

RAÚL P. PELÁEZ

COMPLEX FLUIDS IN THE GPU ERA

Algorithms and simulations

COMPLEX FLUIDS IN THE GPU ERA

Algorithms and simulations

RAÚL P. PELÁEZ



Departamento de Física Teórica de la Materia Condensada
Facultad de Ciencias
Universidad Autónoma de Madrid

2022 – v1.0

Raúl P. Peláez: *Complex fluids in the GPU era, Algorithms and simulations*, © 2022

SUPERVISOR:
Rafael Delgado Buscalioni

Raúl P. Peláez: *Complex fluids in the GPU era, Algorithms and simulations*, © 2022

This thing is complex... so complex not even God gets it.

— Rafa Buscalioni

There is no way you are getting a quote for your thesis

— NVIDIA

ENGLISH ABSTRACT

Complex fluids is an umbrella term for the coexistence between two phases, in our cases of interest these being a solid and a liquid. A wide range of soft matter systems, oftentimes of a biological nature, can be categorized as complex fluids. We often find the presence of a group of certain slow solid, or soft, objects submerged in a solvent comprised by smaller particles governed by faster dynamics. A suspension of colloidal particles in water or a small virus diffusing in the cellular environment are examples of complex fluids. The dynamics of these kind of systems are more often than not influenced (and sometimes straight-up controlled) by thermal fluctuations and hydrodynamic correlations arising from the elimination of the solvent's degrees of freedom. Additionally, the two-phase coupling often imposes geometrical restrictions that fundamentally alter the dynamics of both the solution and the solvent. Both of these effects require special (and in some occasions altogether novel) mathematical and algorithmic consideration. This thesis pushes the boundaries of numerical simulation of complex fluids. In this manuscript, I present an overview of past, recent and novel algorithms adapted to the GPU, a powerful, massively-parallel hardware recently developed. This thesis has culminated in the release of UAMMD, a new software infrastructure for complex fluids simulations in the GPU. Besides the novel tools and methodology, I will also go through some real-life applications which have led to scientific breakthroughs published in research journals.

SPANISH ABSTRACT

El término fluido complejo se refiere a la consistencia entre dos fases, en nuestros casos de estudio habituales siendo una de ellas sólida y la otra líquida. Un amplio rango de sistemas en materia blanda, a menudo de naturaleza biológica, pueden categorizarse como fluidos complejos. Típicamente nos encontramos con un grupo de objetos sólidos, o blandos, cuya dinámica es lenta, que están sumergidos en un solvente compuesto de partículas más pequeñas y con una dinámica mucho más rápida. Algunos ejemplos de fluidos complejos son una suspensión de partículas coloidales en agua o un pequeño virus difundiéndose en el entorno celular. La dinámica de este tipo de sistemas está habitualmente influenciada (o incluso gobernada) por fluctuaciones térmicas y correlaciones hidrodinámicas que surgen de la eliminación de algunos de los grados de libertad del solvente. Adicionalmente, el acoplamiento entre las dos fases a menudo impone restricciones geométricas que alteran fundamentalmente la dinámica de la solución, así como la del solvente. Ambos mecanismos requieren tener en cuenta consideraciones matemáticas y algorítmicas especiales (y en ocasiones completamente novedosas). La presente tesis extiende los límites de la simulación numérica de fluidos complejos. En este manuscrito, presento una serie de algoritmos, pasados y nuevos, adaptados a la GPU, un poderoso hardware extremadamente paralelo recientemente desarrollado. La culminación de esta tesis ha tenido como resultado la publicación de UAMMD, una nueva infraestructura de software para la simulación de fluidos complejos en la GPU. Además de las herramientas y metodología originales, también repasaré algunas aplicaciones prácticas que han resultado en avances científicos publicados en revistas de investigación.

PUBLICATIONS

The following publications resulted from the development of this work

- [1] S. Panzuela, Raúl P. Peláez, and R. Delgado-Buscalioni. “Collective colloid diffusion under soft two-dimensional confinement.” In: *Phys. Rev. E* 95 (1 2017), p. 012602. DOI: [10.1103/PhysRevE.95.012602](https://doi.org/10.1103/PhysRevE.95.012602).
- [2] Raúl P Peláez, Florencio Balboa Usabiaga, Sergio Panzuela, Qiyu Xiao, Rafael Delgado-Buscalioni, and Aleksandar Donev. “Hydrodynamic fluctuations in quasi-two dimensional diffusion.” In: *Journal of Statistical Mechanics: Theory and Experiment* 2018.6 (2018), p. 063207. DOI: [10.1088/1742-5468/aac2fb](https://doi.org/10.1088/1742-5468/aac2fb).
- [3] Paloma Rodríguez-Sevilla, Francisco Sanz-Rodríguez, Raúl P. Peláez, Rafael Delgado-Buscalioni, Liangliang Liang, Xiaogang Liu, and Daniel Jaque. “Upconverting Nanorockers for Intracellular Viscosity Measurements During Chemotherapy.” In: *Advanced Biosystems* 3.10 (2019), p. 1900082. DOI: <https://doi.org/10.1002/adbi.201900082>.
- [4] M. Meléndez, N. Alcázar-Cano, R. P. Peláez, J. J. Sáenz, and R. Delgado-Buscalioni. “Optofluidic control of the dispersion of nanoscale dumbbells.” In: *Phys. Rev. E* 99 (2 2019), p. 022603. DOI: [10.1103/PhysRevE.99.022603](https://doi.org/10.1103/PhysRevE.99.022603).
- [5] Raúl P. Peláez and Rafael Delgado-Buscalioni. “Origin of Tank-Treading and Breathing Dynamics of Star Polymers in Shear Flow.” In: *Macromolecules* 53.7 (2020), pp. 2634–2648. DOI: [10.1021/acs.macromol.9b01968](https://doi.org/10.1021/acs.macromol.9b01968).
- [6] Ondrej Maxian, Raúl P. Peláez, Leslie Greengard, and Aleksandar Donev. “A fast spectral method for electrostatics in doubly periodic slit channels.” In: *The Journal of Chemical Physics* 154.20 (2021), p. 204107. DOI: [10.1063/5.0044677](https://doi.org/10.1063/5.0044677).
- [7] Ondrej Maxian, Raúl P. Peláez, Alex Mogilner, and Aleksandar Donev. “Simulations of dynamically cross-linked actin networks: Morphology, rheology, and hydrodynamic interactions.” In: *PLOS Computational Biology* 17.12 (Dec. 2021), pp. 1–38. DOI: [10.1371/journal.pcbi.1009240](https://doi.org/10.1371/journal.pcbi.1009240).

CONTENTS

List of Figures	xiii
List of Codes	xxviii
I INTRODUCTION	
1 INTRODUCTION	3
1.1 The Graphical Processor Unit	5
1.2 Basic concepts of GPU programming	9
1.3 Software for soft matter simulations	12
II UAMMD: DESIGN AND COMPONENTS	
2 THE FERMATION OF SUBPROBLEMS	17
3 IT'S IN THE NAME	22
3.1 Universality	22
3.2 Adaptability	23
3.3 Multiscale	25
3.3.1 The Fokker-Planck Formalism	28
3.3.2 Fluctuation-Dissipation balance	30
3.3.3 Einstein relation	32
3.4 Molecular Dynamics	33
4 INITIAL REMARKS AND OVERVIEW	34
III A VOYAGE THROUGH NUMERICAL SPACE-TIME	
5 BUILDING AN UAMMD MODULE	40
6 PARTICLEDATA	41
7 A SELF-CONTAINED UAMMD CODE EXAMPLE	44
8 PARTICLE INTERACTIONS	48
8.1 The Interactor interface	48
8.1.1 The simulation domain	51
9 LONG RANGE INTERACTIONS	53
9.1 The NBody algorithm	55
10 SHORT RANGE INTERACTIONS	60
10.1 The Lennard-Jones potential	61
11 NEIGHBOUR LISTS	63
11.1 The NeighbourContainer interface	66
11.2 Cell list	69
11.3 Verlet list	78
11.4 LBVH list	82

11.5 Performance comparisons	85
12 BONDED INTERACTIONS	88
13 PARTICLE DYNAMICS	91
13.1 The Integrator interface	91
14 MOLECULAR DYNAMICS	94
14.1 Finite-difference methods	95
14.1.1 Euler Methods	96
14.2 The velocity Verlet algorithm	98
15 LANGEVIN DYNAMICS	101
15.1 Grønbech-Jensen	104
16 DISSIPATIVE PARTICLE DYNAMICS	106
17 BROWNIAN DYNAMICS	110
17.1 Metropolis Adjusted Langevin Algorithm (MALA)	115
18 HYDRODYNAMIC INTERACTIONS	120
18.1 Connection with Brownian Dynamics	125
19 HYDRODYNAMICS IN OPEN BOUNDARIES	129
19.1 Cholesky	130
19.2 Lanczos	131
20 TRIPLY PERIODIC HYDRODYNAMICS	134
20.1 Force Coupling Method (FCM)	135
20.2 Positively Split Ewald (PSE)	141
20.3 Spatial discretization with staggered grids	146
20.4 Fluctuating Immersed Boundary (FIB) in triply periodic systems	148
20.5 Inertial Coupling Method (ICM)	152
21 THE IMMERSED BOUNDARY METHOD (IBM) KERNELS	155
21.1 Spreading and interpolation algorithms	161
21.2 Comparing the different spreading kernels for BDHI	178
 IV NOVEL ALGORITHMS AND PHYSICS FOR COMPLEX FLUIDS	
22 HYDRODYNAMICS UNDER QUASI TWO-DIMENSIONAL CONFINEMENT	185
22.1 The Force Coupling Method in quasi 2D	188
23 HYDRODYNAMICS IN DOUBLY PERIODIC GEOMETRIES	195
23.1 Free-space solver	198
23.2 Corrections	201
23.3 The zeroth wave number	202

23.4 On spreading and interpolation	204
24 TRIPLY PERIODIC ELECTROSTATICS	208
24.1 Ewald splitting	210
25 DOUBLY PERIODIC ELECTROSTATICS	214
25.1 Solver description	216
25.1.1 Free space solver, ϕ_{DP}	217
25.1.2 Harmonic correction, ϕ_{corr}	218
25.1.3 The $k = 0$ mode	220
25.1.4 Wrapping up	221
25.2 Ewald splitting	221
25.2.1 Near field	223
25.2.2 Far field	223
25.3 How to use in UAMMD	223
V NEW PHYSICS AND APPLICATIONS	
26 MEASURING INTRACELLULAR VISCOSITY	227
27 STAR POLYMER DYNAMICS IN SHEAR FLOW	231
28 HYDRODYNAMICS IN CONFINED GEOMETRIES	232
28.1 From soft to strict confinement	232
28.2 Limit of strict confinement	235
29 OPTOFLUIDIC CONTROL OF THE DISPERSION OF NANOSCALE DUMBBELLS	239
30 ONGOING WORK, FUTURE DIRECTIONS AND CONCLUSIONS	240
31 SPANISH CONCLUSIONS	243
VI APPENDIX	
A BASIC NOTIONS OF CUDA/C++ PROGRAMMING	247
A.1 Basics rules of CUDA programming	260
B DEALING WITH THE FFT IN THE GPU	262
C BOUNDARY VALUE PROBLEM (BVP) SOLVER	266
D UAMMD'S ONLINE DOCUMENTATION	269
E THE TRANSVERSER AND POTENTIAL INTERFACES	271
E.1 The Transverser Interface	271
E.2 The Potential Interface	274
F A FULL UAMMD SIMULATION EXAMPLE	278
G LIST OF CURRENTLY IMPLEMENTED MODULES	280
G.1 Integrators	280
G.2 Interactors	281
G.3 Other modules	281

LIST OF FIGURES

- Figure 1.1 The spatio-temporal landscape and its numerical techniques. Different techniques are applied to simulate the different scales involved in a biological system or, in general, a complex fluid. Zooming in from the bottom-left we have the blood inside a section of a vein, the plasmic environment in a small drop of blood (where cells, platelets, viruses, etc. are present), a virus, a single protein and finally the individual atoms that compose it along with the surrounding fluid particles (water). [4](#)
- Figure 1.2 A picture a GPU (NVIDIA Tesla). This card will grant its owner the computing power of a small CPU cluster for a price of around 1000€. [5](#)
- Figure 1.3 Representation of the thread geometry for a kernel launch with 2 blocks of 4 threads ($blockDim = (2,2)$). Threads inside the same block run concurrently, while different blocks might run desynchronized between them. Each thread has a small local (and private) register memory space. Furthermore, threads inside the same block can share memory among them via the low-latency *shared memory*. All threads can access the same, low-bandwidth, *global memory* space. [10](#)

- Figure 1.4 Representation of a divergent branch. The 32 threads conforming a warp (curvy arrows) encounter a piece of divergent code (a conditional branch that separates threads in the same warp). When divergent code is encountered each branch is executed in serial (as opposed to, for instance, a parallel CPU code), execution does not continue until all threads have concluded processing all branches since all threads in the same warp must execute instructions in lock-step. [11](#)
- Figure 2.1 The basic hierarchy of concepts (and code) in [UAMMD](#), represented by a series of modules connected by arrows that convey the direction of the data flow. *System* holds information about the actual physical hardware, and all the entities below rely on it to interact with the environment. *ParticleData* (a “real class” in the code) stores all the information about the particles in the simulation, such as positions (\mathbf{r}), velocities (\mathbf{v}), mass, etc, as well as other properties like the current forces acting on it (\mathbf{F}), energy (E) or virial (T). Modules below (explained later) can request, at their leisure, a list with any of the particles properties. *Integrators* and *Interactors* are interfaces (called virtual classes in programming terms), are in charge of, respectively, forwarding the simulation one step in time and computing the forces, energies and/or virials acting on each particle. These “Interfaces” (drawn in red) are abstract objects that cannot be instanced on their own but rather must be inherited. For instance, Brownian Dynamics (section. [17](#)) would be an *Integrator*, while a module that computes gravitational forces would be an *Interactor*. [20](#)

- Figure 3.1 A simulation is constructed by putting together several modules. In this instance, the simulation of a virus submerged in water could be constructed by joining a hydrodynamics *Integrator* module with several *Interactors* describing the interactions between particles, such as sterics, bonds, electrostatics,... 23
- Figure 3.2 The spatio-temporal landscape and its numerical techniques. Different techniques are applied to simulate the different scales involved in a biological system or, in general, a complex fluid. Image courtesy of Rafael Delgado-Buscalioni. 25
- Figure 3.3 The different levels of coarse grained description available in UAMMD for soft matter simulations. \mathbf{q}_i represents the positions of particles, \mathbf{u}_i their velocity. $\mathbf{v}(\mathbf{r}, t)$ represents an Eulerian fluid velocity field, $M(\mathbf{q}_{ij})$ a mobility tensor and $\xi(\mathbf{q}_{ij})$ a friction kernel. Arrows represent standard routes for the formal bottom-up derivation using coarse-graining theory. In order, we have: (1) in [25], (2) in [26] and (3) in [27]. 26

- Figure 4.1** A bird's-eye of the different conceptual layers of the UAMMD infrastructure. Many (but not all) of the implemented modules are represented inside the red circle, we will discuss all of them through this manuscript. At the innermost level, the library level, we have the basic tools and algorithms on which the rest of the project is supported. The next layer is the solver layer, with modules for particle interaction and dynamics. In general, all these modules are black-box input/output modules that can be used in any other code in a library-like fashion as external accelerators (especially the ones at the library level). Outside the red circle we find the application layer, in which several of the inner tools are joined to study the physics of a new system. For instance, to study the phenomenon of electrochemical impedance we need electrostatics (interactions) and hydrodynamics (dynamics) in a slit channel (chapters 25 and 23, respectively), the modules inside blue circles are connected first to construct the interaction and dynamics modules, which in turn are used to set up a simulation. **36**
- Figure 8.1** The next level in the *Interactor* branch of the tree presented in Fig. 2.1. Names in blue represent **UAMMD** modules that can be specialized with outside logic. For instance, the bonded interactions module is general to any arbitrarily complex potential, which can be provided to the module via the use of metaprogramming. **49**
- Figure 8.2** Overview of the concepts in the following chapters. Green text represents template arguments, while generic algorithms are purple. Long- and short-ranged interactions can be specialized from the outside via the *Potential* interface (see Appendix E.1). Arrows denote logical dependence. **50**

- Figure 8.3 A representation of periodic boundary conditions. The unit cell, containing the system, will interact with the surrounding copies of itself if periodic boundary conditions are in place. Given two particles in the unit cell, if the **MIC** is considered, they will only interact with the closest images of each other (included the one in the unit cell). [52](#)
- Figure 9.1 Five particles (black circles) must interact with every other (represented by red lines) in a NBody interaction. There are $N(N - 1) = 20$ lines, since self interactions are not depicted here. [54](#)
- Figure 9.2 Representation of algorithms [2](#) (upper half of the figure) and [3](#) (bottom half). Two thread blocks ($b_{0,1}$), with two threads each ($t_{0,1,2,3}$), have to process four particles ($p_{0,1,2,3}$). Each thread must go through all particles and fetch some arbitrary information for each of them (i.e. positions, velocities and/or any other data required by the computation), represented by colored lines. The naive algorithms does not make use of the shared memory space available to threads in the same block and thus each thread needs to read all particles. On the other hand, the shared algorithm benefits from the shared memory space. In particular, each particle must be fetched only once per block (as opposed to once per thread in the naive algorithm). This is depicted by the reduced number of colored lines in the lower side of the figure. [58](#)
- Figure 10.1 The Lennard-Jones potential described in Eq. [10.1](#) [62](#)

- Figure 11.1 A depiction of a neighbour list in a section of a particle distribution (left). Particles inside the blue circle (of radius r_c) are neighbours of the red particle. The Verlet list strategy (section 11.3) defines a second safety radius, r_s , that can be leveraged to reuse the list even after particles have moved. In the worst-case scenario of the red particle and another particle just outside r_s approaching each other (right), the list will be invalidated only when each has moved $r_t = \frac{1}{2}(r_s - r_c)$ since the last rebuild. 64
- Figure 11.2 The three types of neighbour lists in UAMMD. All of them can be used via the same unifying interfaces; *Transverser* (described in Appendix E.1) and *NeighbourContainer* (described in sec 11.1). Additionally, the internal structures of each list can be requested for custom use. 65
- Figure 11.3 Sketch of the cell list algorithm. Space is binned (black grid) and the bin (cell) of each particle is computed. In order to look for the neighbours of the black particle (those inside the green dashed circle) all the particles inside the adjacent cells (bins inside the orange dashed square, 27 cells in three dimensions) are checked. The orange particles are therefore false positives. Finally, the yellow particles are never considered when looking for neighbours of the black one. 69

- Figure 11.4 Representation of the spatial binning for a 2D distribution of particles. The particle marked as i lies in the cell with coordinates $(1, 2)$, these coordinates will then be used to assign a hash to particle i . The dotted red line represents the order given by the space-filling curve, starting at the $(0,0)$ cell. Blue particles lie in arbitrary indexes, aka memory locations, (blue numbers). After hashing the positions according to the red dashed line and sorting, particle positions of particles in the same cell are stored contiguously in *sortPos* (right), particles inside the same cell have an undetermined order. Solid red lines mark the end of a cell or, equivalently, the start of the next one, information that is stored in the *cellStart* and *cellEnd* arrays. 71
- Figure 11.5 Performance of the Verlet list versus the safety cut-off radius, r_s , in a LJ fluid simulation. The system has one million particles. Tests ran on an RTX2080 GPU. 86
- Figure 11.6 Performance comparison of the different neighbour list strategies in a LJ liquid at two different particle concentrations. The number of steps between reconstructions of the Verlet list is highly dependent on any parameter that modifies the diffusivity of the particles (like the temperature or the time step), which directly affects performance. In contrast, the other lists, being reconstructed every step regardless, are oblivious to the temperature or, in general, to changes in the diffusivity of particles. All tests were carried out in a RTX2080 GPU. 87

- Figure 13.1 The next level in the *Integrator* branch of the tree presented in Fig. 2.1. Different methods are available for each of the levels of description (pink) introduced in chapter 3. Methods available as UAMMD modules are in black. 92
- Figure 17.1 Optimal time step in MALA for several acceptance ratios and number of particles using a soft repulsive potential given by Eq. (17.20) (left) and a hard steric WCA interaction (right). Naturally, a higher acceptance ratio (with 1 corresponding to accepting all configurations and 0 to never accepting them) results in a lower step size, δt . A by-product of the MALA algorithm comparing the total energy change is that δt depends on the number of particles so that more particles result in a lower optimal δt . In all cases the number density of particles is kept at $\rho := N/V = 0.1a^{-3}$. 118
- Figure 18.1 Representation of the Oseen mobility in 2D. A force acting on a point (green) is propagated to the fluid via the Oseen tensor (blue lines). The fluid behind the source will be dragged onto it, while the fluid in front will be pushed away. Note that the action of the Oseen tensor decays as the inverse of the distance, a behavior this representation does not convey. 124
- Figure 20.1 Representation of a staggered grid. Each quantity is defined on its own grid, with origin (circles) shifted by $h/2$ between them. Crosses mark cell centers in the various grids. The x coordinates of vectors are defined on the red grid, y coordinates on the green. Tensors are defined on the orange one and finally, scalars are defined on the black grid. 147

- Figure 21.1 A representation of the Immersed Boundary. The blue circle represents a particle (with the green cross marking its center). Some quantity (i.e. the force) acting on it will be spread to the grid points inside its radius of action(red crosses). [156](#)
- Figure 21.2 Two particles (red and blue circles, centered at the green circles) in a one dimensional grid ($N_x = 5, N_y = N_z = 1$). The kernel has a support of $n_s = 3$ cells (top) and 4 cells (bottom). Dashed lines indicate the middle of each cell and crosses represent the cells that need to be visited by the spreading or interpolation operations for each particle. If the support is even, depending on where a particle is inside the cell, some set of neighbours or another will be visited for spreading/interpolation. In this particular example, the shift, P , on the bottom (even) support case will be 2 for the red particle and 1 for the blue one. [163](#)

- Figure 21.3 Performance comparison between the different spreading schemes considered for two different densities. Particles are distributed uniformly on a cubic grid and spread with a support $n_s = 8$ using a Gaussian kernel. A single scalar is spread per particle. The considered algorithms are: A block-per-particle (BPP), the algorithm used in [UAMMD](#). A thread-per-particle (TPP) precomputing the kernel to register memory (register, similar to *GM-sort* in Shih2021 [118]) and recomputing it (recompute). Finally, timing for the BDP algorithm (*SM* in [118]) is also presented. For these tests, particle sorting was not employed, i.e. the particles are randomly distributed in memory with respect to their physical positions. Performance data was gathered in an RTX2080Ti [GPU](#) using single precision. The crossover between BPP and SM happens at around 2 million particles in both cases. [172](#)
- Figure 21.4 The same situation as in Fig. 21.3, but in this case particles are presorted to increase spatial locality in memory (particles close in space are placed close in memory). Both the positions and the spread quantity are sorted. Sorting is carried out using the same Morton hash technique laid out in chapter [11.2](#). Sorting takes a neglegible amount of time and, moreover, it does not need to be done every time spreading occurs. Typically particles are sorted at the start of a given simulation and then every few diffusive times. [173](#)
- Figure 21.5 Interpolation performance for different densities. Particles are distributed uniformly in a cubic grid and one value is interpolated with a support $n_s = 8$ using a Gaussian kernel. Performance data was gathered in an RTX2080Ti [GPU](#) using single precision. All data lies near the 9ns mark. [175](#)

- Figure 21.6 Deviation from the average hydrodynamic radius inside a cell in the x direction. Shown here is the error computed by evaluating Eq. (21.20) inside a range $x = [0, 1]h$ with the mean subtracted. Since the signal is symmetrical only the range $[0.5, 1]h$ is presented. 180
- Figure 21.7 Variance of the hydrodynamic radius inside a cell (2D slice at $z = L/2 + h/2$ of a 3D system) for different kernels in FCM. Tests were carried out on a cubic box of size $L = 32a$. 181
- Figure 22.1 Representation of a colloidal system under soft confinement in the perpendicular (z) direction. Particles are confined via some external potential that distributes them near $z = 0$ with a typical width δ . The confining forces acting on the particles are propagated to the plane via the Oseen tensor (orange lines). 186
- Figure 23.1 Representation of a lipid membrane inside a doubly periodic domain. The BCs can be customized at the domain limits ($z = -H, H$) to be either open or have no-slip walls. 196
- Figure 23.2 Representation of a hybrid regular-Chebyshev grid. In the Doubly Periodic Stokes algorithm, the plane-parallel components (x and y) are described on a regular grid (with nodes at $x = -L/2 + i/NL$ and similarly for y), whereas the perpendicular direction (z) is discretized at the Gauss-Chebyshev-Lobatto points (the extrema of the Chebyshev polynomials), $z = H \cos(\pi i / N_z)$ (where i runs from 0 to N_z). 205

- Figure 23.3 Representation of the image spreading in the DP Stokes algorithm. Forces acting on particles (blue circles) are translated to the fluid as a force density. Due to images, part of the particle's force is zero near walls. Zero fluid forcing is depicted as white. A particle located at exactly the height of a wall has no effect on the fluid. 206
- Figure 25.1 Schematic representation of the doubly periodic domain described by Eqs. 25.1 and 25.3-25.6. Each domain wall is represented with a different color. Blue clouds represent Gaussian charge sources. 214
- Figure 25.2 Enforcing the dielectric jumps using images. The presence of the two dielectric boundaries (at $z = 0$ and $z = H$) causes each charge to have infinite reflections. For instance, the charge q reflects through the bottom with an effective charge q^* , which in turn reflects again through the top wall with q^{**} . We manage to Ewald split the problem into a near and far field contributions in such a way that both parts only need to take into account the first reflections at most. 222
- Figure 26.1 Representation of the simulation environment used to model the experimental setup in [134]. The nanorocker (grey) and the microtubulae (red) are modeled as spring-connected blobs. 228

Figure 26.2 Computed and experimentally measured mean squared angular displacement (MSAD) over time. Comparison is made between simulations of free (brown line) and laser-illuminated (confined, orange line) nanorocker, surrounded by 10 microtubules (MT), and the experimental results (Exp.) after 5 h incubation time with Taxol (purple data). The red dashed line represents the MSAD calculated from Equation (1) using the measured viscosity for an incubation time of 5 h. The good agreement between the red and purple curves validates the approximation of the nanorocker to a disk. The inset shows the mean squared displacement (MSD) for free nanorockers in the presence of 0, 5, and 10 microtubules. The agreement between the experimental and simulation results indicates that the movement of the particle is not affected by the presence of microtubules since their separation is much bigger than the particle. [230](#)

Figure 27.1 A star polymer solution under a shear flow. [231](#)

Figure 28.1 Collective diffusion, $D_c(k)$, of a group of ideal colloids under strict confinement ($\delta \rightarrow 0$) for several densities. Dashed line correspond to $D_c/D_0 = 1 + \frac{\lambda}{2\pi L_h}$. Here $L_h := \frac{2a}{3\phi}$ is the hydrodynamic length and $\lambda := \frac{2\pi}{k}$ is the wavelength of a density perturbation. Note that for $\lambda \rightarrow \infty$, $D_c \rightarrow D_0$, corresponding to the single particle diffusion coefficient, i.e. in this limit there are no collective effects (in the case of ideal particles). [234](#)

Figure 28.2 The enhancement of the collective diffusion (given by the hydrodynamic function $H_c(k)$ in Eq. 28.1) of a group of ideal colloids ($S(k) = 1$) as the confinement goes from non-existent ($\delta \rightarrow \infty$) to stiff ($\delta \rightarrow 0$) trap. Dashed lines correspond to Eq. (18) in [126], from where this figure has been taken. 235

Figure 28.3 Time evolution of a stripe density perturbation initially localized in the middle third of the domain in quasi 2D. Shown here are the density profiles averaged both in ensemble and in the x direction. (Left) The total density, $c^{(1)}(y, t)$, at time $t = 0$ (dashed-solid black line) and at a later point in time (red squares), theory for the quasi 2D line (red) comes from our mean field (Eq. 30 in [71]). The solution to the diffusion equation without hydrodynamic interactions (at the same time) is shown as a solid black line. (Right) We tag particles starting in the middle stripe as green ($c_G^{(1)}$, green lines and squares) and the rest as red ($c_R^{(1)}$, red lines and circles) and plot their density profiles, $c_{R/G}^{(1)}$. Red and green symbols correspond to $c_R^{(1)}$ and $c_G^{(1)}$ at the same point in time as the left pane. Dotted lines correspond to $t = 0$. Our mean field theory is also shown here with solid lines. Dashed lines correspond to the diffusion equation without hydrodynamic interactions. All particles are passive tracers. 236

Figure 28.4 Time evolution (snapshots at times increasing from left to right) of a density perturbation initially localized in the middle third of the domain (blue represents zero concentration of particles). We show snapshots for quasi 2D (top row) and true 2D (bottom row) at the same diffusive times ($t^{t2D/q2D} = tD_0^{t2D/q2D}/a^2$). The images show the number density by counting the number of particles in each cell of a 128^2 grid; the color bar is fixed across all panes from 0(blue) to 0.4 (red). All particles are passive tracers. [237](#)

Figure 28.5 Time evolution (snapshots at times increasing from left to right) of a color (species) density perturbation initially localized in the middle third of the domain. All simulations have a uniform total density, with a packing fraction of $\phi = 1$. We show snapshots in the absence of hydrodynamics (BD, top row), quasi 2D (middle row) and true 2D (bottom row) at the same diffusive times ($t^{BD/t2D/q2D} = tD_0^{BD/t2D/q2D}/a^2$). The images show the number density by counting the number of particles in each cell of a 64^2 grid; the color bar is fixed across all panes from 0(blue) to 0.4 (red). All particles are passive tracers. [238](#)

Figure 28.6 Self diffusion coefficient (measured via the mean square displacement (MSD)) of a single particle in quasi 2D for different packing densities, ϕ . [239](#)

- Figure A.1 The different kinds of C++ iterators. Iterators can be either for input (reading), output (writing) or both (read/write). The most generic type of iterator is called *Random Access* and allows for arbitrary access to its elements. Below, a *Bidirectional* constrains the access to be always next to the previously accessed element. In other words, a *Bidirectional* iterator can only be issued to advance or regress one element at a time. Finally, the most restricted iterator kind, the *Forward* iterator, can only be accessed element by element in order.
[258](#)

LIST OF CODES

- Source Code 1** Template code for the creation of a module. [40](#)
- Source Code 2** Definition of the UAMMD auxiliary structure created for this manuscript. Other examples making use of the *UAMMD* struct need this snippet as a preamble to compile. [41](#)
- Source Code 3** Creating and using an instance of *ParticleData*. [42](#)
- Source Code 4** A constant-energy UAMMD simulation of a collection of Lennard-Jones particles. For simplicity, all parameters are hard-coded (as opposed to, for instance, being read from a file). [45](#)
- Source Code 5** The basic outline of a new *Interactor*. A lot of advanced functionality has been omitted here for simplicity, refer to Appendix D for more information. See chapter 6 for more information about how to access particle properties. [49](#)
- Source Code 6** Using the Box class. [53](#)

- Source Code 7** Usage examples of the three different ways the NBody algorithm is exposed in [UAMMD](#). The *Transverser* and *Potential* interfaces are described in Appendix E. Note that nothing in this example hints at the computation being long ranged, besides of the fact that the NBody algorithm is being used. While this ensures that every pair of particles in the system will be visited, the *Transverser* (or *Potential*, any of which will provide the actual computation logic/-physics) can simply choose to, for instance, ignore any pair further than a certain distance. [59](#)
- Source Code 8** Creating and returning a short range interaction module. The *Potential* interface is described in Appendix E. [63](#)
- Source Code 9** A CUDA kernel that uses a *NeighbourContainer* to go though all the neighbours of each particle (when launched with as many threads as particles). This code is an implementation of the pseudo code in Algorithm 4. [66](#)
- Source Code 10** The different ways of using a Cell List in [UAMMD](#). The *Transverser* interface, already used in chapter 9, is described in detail in Appendix E. [76](#)
- Source Code 11** The different ways of using a Verlet List in [UAMMD](#). The *Transverser* interface, already used in chapter 9, is described in detail in Appendix E. Note that the *Transverer* and *NeighbourContainer* options are identical to the case of a *CellList*. [81](#)
- Source Code 12** Example usage of the LBVH List in [UAMMD](#). The *Transverser* interface is described in detail in Appendix E. Note that the *Transverer* and *NeighbourContainer* options are identical to the case of a *CellList*. [84](#)
- Source Code 13** Constructing and returning a Bonded interaction module. [90](#)

- Source Code 14** The basic outline of a new *Integrator*. Some advanced functionality has been omitted here for simplicity, refer to Appendix D for more information. See chapter 6 for more information about how to access particle properties. A class inheriting from *Integrator* has access to a series of built-in members, in particular, a *ParticleData* instance is always available under the name “pd”. Furthermore, a list of *Interactors* is available in the variable “interactors”. The Integrator defined below implements a simple forward-Euler integration scheme (which will be introduced in sec. 14.1.1). 93
- Source Code 15** Example of the creation of a *VerletNVE Integrator*. 100
- Source Code 16** Example of the creation of a *VerletNVT Integrator* module. 105
- Source Code 17** Creation of an instance of a **DPD** integrator. In **UAMMD**, **DPD** is encoded as a Verlet integrator coupled with a short range interaction. 109
- Source Code 18** Example of the creation of a **BD Integrator** module. 115
- Source Code 19** Example of the creation of a *MALA Integrator* module. 119
- Source Code 20** Example code for the Cholesky **BDHI** method 131
- Source Code 21** Example code for the Lanczos **BDHI** method. 132
- Source Code 22** Usage example of the **FCM** module 139
- Source Code 23** Example of the creation of the **PSE** module. 145
- Source Code 24** Example of the creation of a *FIB Integrator* module. 151
- Source Code 25** Example of the creation of a *ICM Integrator* module. 154

- Source Code 26** Usage example of the spreading and interpolation module. Some data (stored in *dataAtParticlePositions*) located at some positions (stored in the *positions* array) can be spread to a grid (for which the values of each cell will be stored in *dataAtCellPositions*) using the function *spreadWithIBM*. Similarly, the function *interpolateWithIBM* can be used for the interpolation operation. As an example, the **FCM** module (chapter 20.1) uses code similar to this one in order to spread particle forces into a grid and, after solving the Stokes equation, interpolate the velocities on a grid back to the particle positions. 176
- Source Code 27** Example of the creation of a quasi 2D *Integrator*. 194
- Source Code 28** Using the DPStokes module for doubly periodic hydrodynamics. 207
- Source Code 29** Usage example of the triply periodic Poisson module. 213
- Source Code 30** Usage example of the doubly periodic Poisson module. 224
- Source Code 31** A C++ program that does nothing. 248
- Source Code 32** Defining a function in C++. 248
- Source Code 33** The classic Hello World program in C++. Once executed, it will print “Hello world” to the terminal. 249
- Source Code 34** Examples of object creation. 251
- Source Code 35** Examples of a variable going out of scope 252
- Source Code 36** Inheriting a virtual class. 254
- Source Code 37** Using shared pointer to hold many types of polygons. 255
- Source Code 38** Using shared pointer to hold many types of polygons. 256

Source Code 39 Using iterators in C++. In this example a counting iterator for the range 0-5 is constructed from three different sources. For the first two cases a container (vector) is created and filled with the actual integers from 0 to 5 (requiring memory allocation). We then get an iterator to the range of the container via the method *begin* of the vector (which returns an iterator pointing to the first element, see the variable “*vit*”). An equivalent way in this case is to simply get a pointer to the first element of the container (variable “*pit*”). Finally, a counting iterator is constructed using the one available in the thrust library. This special iterator is not associated to an underlying container, instead it is an object that provides an overloaded bracket operator that when given an integer, “*i*”, simply returns that same integer, “*i*”. Note that in this example, “*vit*” and “*pit*” are input/output iterators, while “*cit*” is just an input iterator. The method “*cbegin*” in vector can be used to make “*vit*” an input iterator instead. Additionally, making the type of “*pit*” be “*const int**” will also turn it into an input iterator. On the other hand, “*cit*” cannot be an output iterator, since there is nothing to write to behind it. [259](#)

Source Code 40 Creating vectors allocated in the [GPU](#), populating them and performing the *saxpy* operation using a kernel. [260](#)

Source Code 41 Getting the wave number corresponding to a given linearized index. [264](#)

Source Code 42 Finding out if a wave number corresponds to a Nyquist point given the wave number and the grid size. [265](#)

Source Code 43 A *Transverser* that counts the number of neighbours of each particle. [273](#)

Source Code 44 An example *Potential* that computes Lennard-Jones forces, energies and/or virials. For simplicity, all relevant parameters are hard-coded here. In particular, $\sigma_{lj} = 1$, $\epsilon_{lj} = 1$ and the cut off is set at $r_c = 2.5\sigma = 2.5$. The potential here defined (called *SimpleLJ*) calculates forces, energies and virials. Note, however, that it does so only when provided to a *PairForces Interactor* (see chapter 10) and, subsequently, to an *Integrator*. In other words, we use *Potentials* to define an *Interactor*, which will be used by an *Integrator* to calculate forces, energies, etc. 276

Source Code 45 A simulation of LJ particles liquid using UAMMD. Source codes 16, 8 and 44 serve as a preamble (must be included or copy/pasted) here. To ease initialization, only two particles are simulated. 278

ACRONYMS

AB Adams-Bashforth.

API Application Programming Interface.

BC Boundary Condition.

BD Brownian Dynamics.

BDHI Brownian Dynamics with Hydrodynamic Interactions.

BVP Boundary Value Problem.

DP Doubly Periodic.

DPD Dissipative Particle Dynamics.

EM Euler-Maruyama.

FCM Force Coupling Method.

FCT Fast Chebyshev Transform.

FFT Fast Fourier Transform.

FIB Fluctuating Immersed Boundary.

FPE Fokker-Plank Equation.

GPGPU General Purpose Computing on GPU.

GPU Graphical Processor Units.

IBM Immersed Boundary.

ICM Inertial Coupling Method.

LD Langevin Dynamics.

LJ Lennard-Jones.

MD Molecular Dynamics.

MIC Minimum Image Convention.

ODE Ordinary Differential Equation.

PBC Periodic Boundary Conditions.

PDE Partial Differential Equation.

PSE Positively Split Ewald.

RHS Right Hand Side.

RPY Rotne-Prager-Yamakawa.

SDE Stochastic Differential Equation.

SPH Smoothed Particle Hydrodynamics.

UAMMD Universally Adaptable Multiscale Molecular Dynamics.

Part I
INTRODUCTION

INTRODUCTION

The modeling and simulation of physical systems is always tied to a certain spatio-temporal scale (see Fig. 1.1). Usually, studying a system through the lens of a certain scale prevents the exploration of the others. Choosing the right lens for a problem requires understanding its characteristic times and lengths. Say, for instance, that you want to study the flow of a waterfall throughout the next thousand years. It would be a bad decision to employ quantum mechanics for such a feat, as it is a theoretical framework for events that take femtoseconds and angstroms.

Mathematical frameworks and numerical techniques have been devised for the exploration of different spatio-temporal scales. However, many physical phenomena are intrinsically *multiscale*, as the effect of small and fast scales affect the larger and slower ones. The advent of supercomputing in the last two decades has presented the field with new powerful tools to leverage. For the first time we have more computing power than what our techniques are capable of handling. Developing new techniques (numerical and theoretical) directly aimed to these new technologies will enable the exploration of regimes previously unreachable, opening the doors to new phenomena. Multiscale problems will particularly benefit from the improvements in supercomputing. For instance, [1] to simulate every molecule of a virus capsid submerged in water (6 million atoms) in the span of $1\mu s$ ($5 \cdot 10^8$ simulation steps). As another example of these huge simulations, the authors of [2] use 100 million particles to model a chunk of the interior environment of a bacterium for some hundreds of nanoseconds (around 10^7 steps).

These sort of simulations would be impossible without the aid of a supercomputer and specialized software tools. Indeed, aside from raw computing power, new fast and efficient numerical schemes (adapted to novel computing architectures) are needed to cover larger spatio-temporal windows in physical modeling. This is precisely the objective of this thesis, which is devoted, in particular, to the development of new algorithms and software tools for the

study of complex fluids and biological systems in high-performance computing environments.

Most of the properties of complex fluids often arise from an intermediate regime, called the *mesoscale*. The definition of mesoscale is fuzzy, usually it ranges from a few nanometers (a protein) to $10\mu m$ (the size of a HeLa cell). Characteristic times range from microseconds to even minutes. The mesoscale poses a series of theoretical and numerical challenges, because it stems from the microscopic spatio-temporal scales and yet it interacts with slower processes over large scales: an optimal challenge for a super computer.

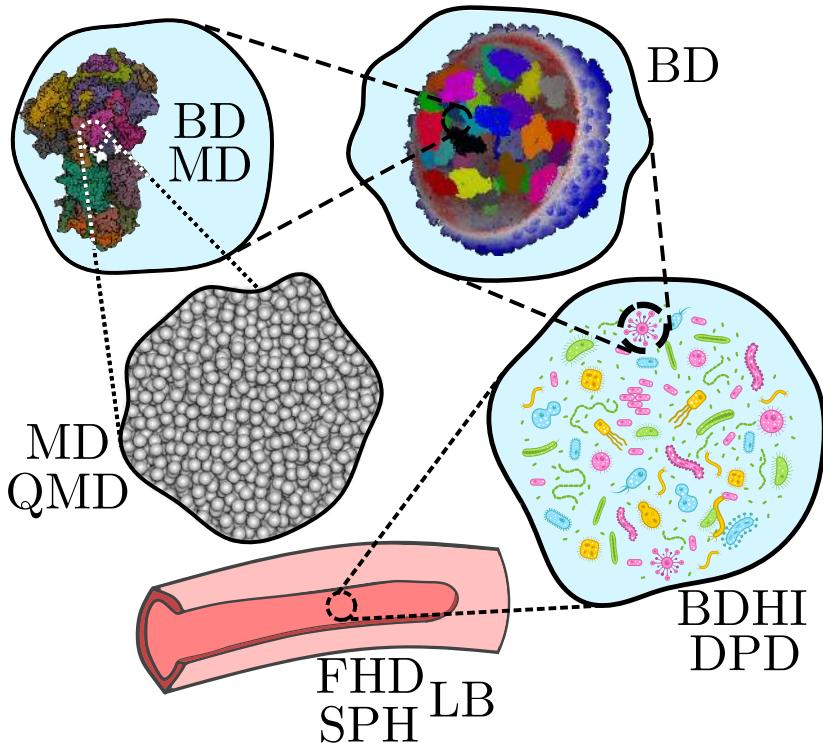


Figure 1.1: The spatio-temporal landscape and its numerical techniques. Different techniques are applied to simulate the different scales involved in a biological system or, in general, a complex fluid. Zooming in from the bottom-left we have the blood inside a section of a vein, the plasmic environment in a small drop of blood (where cells, platelets, viruses, etc. are present), a virus, a single protein and finally the individual atoms that compose it along with the surrounding fluid particles (water).



Figure 1.2: A picture a GPU (NVIDIA Tesla). This card will grant its owner the computing power of a small CPU cluster for a price of around 1000€.

1.1 THE GRAPHICAL PROCESSOR UNIT

Recently, a new paradigm of supercomputing has arisen. The [Graphical Processor Units \(GPU\)](#), a massively parallel co-processor, so powerful that it requires to straight-up rethink our algorithms to take advantage of it.

Yet, the [GPU](#) technology, interpreted as some kind of specialized graphic circuit or co-processor, can be tracked back to the 1970s. Once electronic devices started having screens attached to them, it made sense to have some kind of chip translating the CPU information into the analogical signal of the display. It was then only a matter of time before more functionality was put into them. It started as a way to accelerate the display process in early video game hardware, such as arcade systems. Storing the data that goes into the screen takes a lot of space and, at the time, RAM memory was expensive (it is still, by the way!).

In 1977, the Atari 2600 (one of the first home consoles) could simply not afford to store the contents of its 160x80 pixel display into its 128 bytes of RAM. Of course, these systems employed some

truly creative software tricks to reduce memory usage. However, the numbers just do not add up. Even if we limit ourselves to a monochrome output (so one pixel takes one bit to store), the frame buffer for a 160x80 pixel display takes 1600 bytes¹. The solution was not to have a frame buffer at all, but rather to outsource the display operations to specialized hardware. The Atari 2600's graphic chip had a 128 color palette and allowed developers to use close to 5 sprites to design their 2D games.

These chips were little more than video display chips, but they were the predecessors to the GPU. Throughout the next decade the hardware evolved to support faster and more complex operations, hold more memory, etc. In 1990, a graphic adapter could draw 16 million different colors to a 1028x1024 pixel display and hold 1MB of data.

Graphics adapter were still centered around 2D graphics acceleration, such as graphical user interfaces and 2D games. However, the market was already pushing for real-time 3D graphics in games. Although one could hack the prevailing graphics 2D pipeline to draw 3D environments, without specialized hardware acceleration (as was already common for 2D), the results were far from interactive. Surely, it was a task best suited for the graphics adapter technology already established.

Soon enough, during the early and mid 1990s, several companies were releasing their graphics adapters with 3D acceleration capabilities. In 1994, the term GPU was coined to designate this hardware, which had evolved beyond its original task of just sending pixels to a display. OpenGL[3], one of the first graphics Application Programming Interfaces (APIs)², appeared in the early 1990s attempting to standardize the programming of graphic hardware accelerators, especially for 3D. At the time, OpenGL had quite limited capabilities, allowing a developer to do little more than to feed a fixed pipeline with triangles. OpenGL would then interface with the GPU to turn this geometry into a 2D image that could be displayed. Naturally, this translation involves a great deal of raw computation, further transforming the GPU from a video display card to an independent co-processor. Furthermore, the usual operations required to do this happen to be inherently

¹ In contrast, the NVIDIA GTX 980 (the trusty, consumer-grade, GPU that has accompanied me throughout most of my Ph.D.) has 4 GB of memory and will happily output to several screens with 4k resolution (4096x2160 pixels).

² A software library.

parallel. While the CPU evolved to be formed by a single, powerful core, the **GPU** was born out of a necessity for parallel computing. Thus, a **GPU** tends to have many, less powerful, processing cores. Jumping again to the year 2000, the quality and complexity of computer graphics had exploded. Card manufacturers have included all sorts of new 3D hardware-accelerated operations that go beyond simple triangles and OpenGL has evolved along with them. As it usually happens, people started to hack around, using the graphics pipeline to perform computations not necessarily related to computer graphics. In particular, a new **GPU** was presented by the NVIDIA Corporation in 2001 that allowed to modify the different stages of the graphic pipeline via so-called “programmable shaders”. These shaders were short programs that could intercede between the different stages of drawing. A vertex shader could be written to process each triangle before sending it to a fragment shader, which could process each pixel on the screen before finally sending them to the display. This was the advent of the so-called **General Purpose Computing on GPU (GPGPU)**.

Shaders were enough for the community to realize that drawing polygons was far from the only thing they could do with a **GPU** [4]. Applications exploiting shaders arise everywhere in a variety fields like scientific image processing [5, 6], linear algebra [7–9], physics [10] and even machine learning[11, 12]. One can even find a molecular dynamics code running on the **GPU** of a Sony PlayStation 3 [13].

The next natural step took place in 2007, when the NVIDIA Corporation released CUDA[14]³ alongside its flagship card, the GeForce 8800.

CUDA is a programming model and an extension of the C++ programming language that allows to exploit CUDA-enabled **GPUs** for general-purpose computing. It can be considered as a conceptual generalization of the aforementioned graphic shaders that acknowledges the **GPU** as a general-purpose processor. Via the inclusion of a small set of keywords and extensions to C++, CUDA exposes the massively parallel architecture of the **GPU** under the ecosystem of an already established high-performance programming language. Now developers aiming for the **GPU** did not have

³ Although the original acronym for CUDA was Compute Unified Device Architecture, the platform quickly outgrew this definition. However, the term CUDA was already established and thus it has remained as the name of the platform.

to be experts in hacking a certain graphics [API](#) and could just focus on paralleling the algorithms for its particular architecture.

The birth of CUDA became a pivotal moment in high performance computing, opening the door to a revolution in several fields such as medical analysis, bioinformatics, machine learning, economic market analysis, molecular dynamics and more.

In this thesis, we will push the boundaries of [GPU](#) computing by developing and implementing old and new algorithms for the modeling of soft matter systems into a new infrastructure called [Universally Adaptable Multiscale Molecular Dynamics \(UAMMD\)](#). [UAMMD](#) is designed from scratch with the [GPU](#) in mind and is written in CUDA/C++.

One of the main criticisms of CUDA is that it is a closed source, proprietary environment that can only be compiled for NVIDIA's own [GPUs](#). This restricts CUDA code to only a subset of [GPU](#) hardware. In 2009, an open standard called OpenCL[15] was born (from the creators of OpenGL) as an alternative to CUDA. Although implementations of OpenCL exist for almost any hardware (including GPUs from NVIDIA and AMD, CPUs, FPGAs,...) its adoption has been slow, partly due to an aggressive marketing campaign from NVIDIA to sell CUDA as the better alternative⁴. The general feeling in the community has always been that OpenCL should win as the de facto [GPGPU-API](#) in the long run. However, it simply has not happened yet. Luckily, there are ways to translate CUDA code into OpenCL code⁵ and, given that the paradigms of both [APIs](#) are quite similar, it is fairly straightforward to port CUDA code to OpenCL. Therefore, in the future, porting the software infrastructure presented in this manuscript to OpenCL would be mainly painless and will not require changes to the public [API](#). So, even though at the time of writing [UAMMD](#) is restricted to run on NVIDIA's hardware, it is possible to adapt it so that it runs on virtually any parallel hardware.

⁴ Since the birth of CUDA, NVIDIA has been promoting it by donating GPUs to researchers, giving away a plethora of CUDA courses all around the world, and more.

⁵ In particular, the HIP library works as a thin wrapper over CUDA/OpenCL, allowing to write code that can be compiled for any of the [APIs](#).

1.2 BASIC CONCEPTS OF GPU PROGRAMMING

The **GPU** is a pivotal concept in this work and, as such, it is worth introducing here its basic computing model. Doing so will enable the unfamiliar reader to better grasp the importance of designing algorithms specific for the **GPU** and the new hardships that arise when approaching it from a CPU-centric world. Note, however, that this is merely an introduction to the particularities that come with coding for the **GPU** and will probably not contain enough information for an unfamiliar reader to be able to write **GPU** code. Appendix A contains a more in depth introduction to programming C++ and CUDA, although a plethora of arbitrarily-detailed tutorials and courses are freely available on the web.

The **GPU** (usually referred to simply as *device*) works as a massively parallel co-processor, with thousands of (relatively slow) cores (workers), independent of the rest of the system for most uses and purposes. The **GPU** runs instructions independently (and in principle asynchronously) from the CPU (referred to as the *host*). More importantly, the **GPU** has its own memory space which is separated from the system RAM. The device cannot access the host memory directly (and vice-versa)⁶. If information has to be shared between each other an explicit memory copy has to be issued, which incurs a synchronization barrier and is in general a slow process. In this section, we will discuss how these properties are taken into account and how they affect the programming model.

From a programmatic point of view, we can access the **GPU** capabilities using CUDA, which extends C++ (by introducing new keywords and an **API**) to accommodate for the particular programming model of a **GPU**⁷.

CUDA exposes the many cores of the **GPU** via a series of logical groupings of workers (which in turn map to the hardware cores). The most basic computing unit is called a *thread*, which can execute a single instruction at a time. A group of threads (typically around 128 and inferior to 1024) is called a *thread-block*. Finally, several blocks are grouped into a *grid*. A *grid* is assigned to a special kind

⁶ This is not entirely true anymore for recent CUDA versions via the so-called Unified Virtual Addressing (UVA) system. However, this advanced functionality is beyond the scope of this work and, in any case, **UAMMD** only makes use of it in very specific instances.

⁷ There are other ways to program a **GPU**, such as OpenACC[16] or OpenCL[15], but we will stick to CUDA.

of (**GPU**) function, called a *kernel*, containing instructions that are to be executed by the group of threads.

The main memory space of the **GPU** (the equivalent of the heap memory in the CPU) is called *global memory*. Memory allocated in this memory space is accessible at all times by every thread. Similar to the CPU's heap memory, global memory is slow to access (although a series of caches exists to mitigate this). Each thread has its own small, fast access, local memory space (called *register*, or *local*, memory) which is somewhat equivalent to the *stack* in the CPU. Finally, CUDA exposes another high bandwidth memory space that is private to each block but accessible for all threads within. It is the *shared* memory space, as threads in a block are able to share information directly through it.

See Fig. 1.3 for a schematic representation of the CUDA threading model.

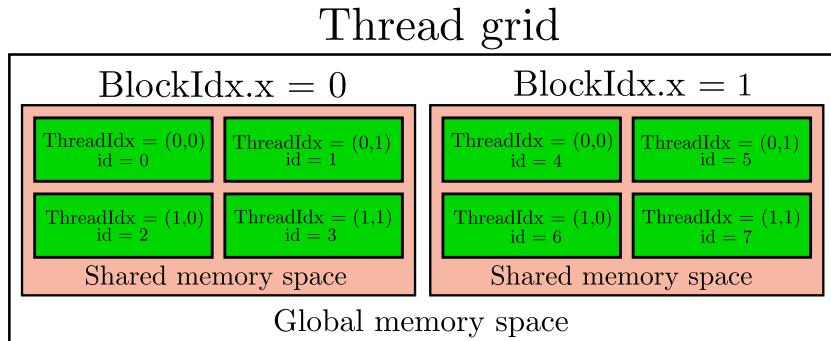


Figure 1.3: Representation of the thread geometry for a kernel launch with 2 blocks of 4 threads ($\text{blockDim} = (2,2)$). Threads inside the same block run concurrently, while different blocks might run desynchronized between them. Each thread has a small local (and private) register memory space. Furthermore, threads inside the same block can share memory among them via the low-latency *shared memory*. All threads can access the same, low-bandwidth, *global memory space*.

Although it is not required by the programming model, threads inside a block excel when executing the same set of instructions in lock-step. In particular, blocks are composed of groups of 32 threads called *warps*. Threads in a warp will execute (at the hardware level) the same instructions at the same time, but not necessarily acting on the same data (in what is known as a SIMD

model). The CUDA compiler will happily process code in which each thread in a warp executes different code (for instance, by using conditional branches), but the actual executable will simply *emulate* this behavior by making all threads execute all branches and discarding the results for the ones that should not have been executed. Naturally, this so-called branch divergence (or simply divergence) greatly hurts performance⁸. Therefore, conditional-branches that result in divergent code in CUDA must be avoided whenever possible. Note that, on the other hand, divergence does not occur when all the threads in a given warp enter the same branch. Figure 1.4 contains a depiction of branch divergence.

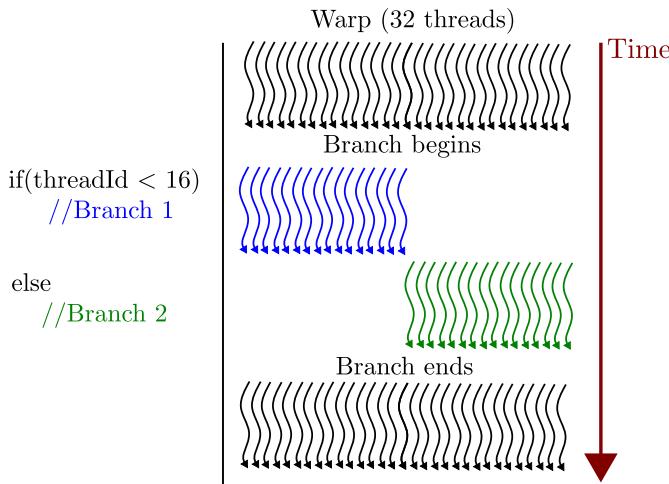


Figure 1.4: Representation of a divergent branch. The 32 threads conforming a warp (curvy arrows) encounter a piece of divergent code (a conditional branch that separates threads in the same warp). When divergent code is encountered each branch is executed in serial (as opposed to, for instance, a parallel CPU code), execution does not continue until all threads have concluded processing all branches since all threads in the same warp must execute instructions in lock-step.

On the other hand, CUDA offers no guarantee regarding the execution order of blocks.

CUDA allows us to write small programs, called kernels, that we can then launch to run on the GPU. The instructions in a kernel are executed by a grid of threads, whose size can be customized

⁸ Furthermore, GPUs do not have branch-prediction as opposed to the CPU, where branch-prediction is one of the pillars of CPU architecture.

per launch. Appendix A offers an introduction to programming GPU code using C++ and CUDA.

1.3 SOFTWARE FOR SOFT MATTER SIMULATIONS

A brief history of molecular dynamics software

Particle based computer modeling of soft matter systems has been around since the advent of computers. The use of computers for physical simulation can be tracked back to as far as the 1940's when researches at the Los Alamos national laboratory described the use of the ENIAC⁹ for simulating some stochastic physical process[17] (the origins of the Monte Carlo method[18]). Shortly after, in the 1950's, computational molecular dynamics was born[19].

One of the first examples of a software package for molecular dynamics is CHARMM, whose initial release dates back to 1983[20]. To put it in perspective, Windows 1.0 was released in 1985, the UNIX operating system was disclosed to the public in 1973 and Linux originated in 1991. It is humbling to realize that since 1983 the basic layout of a fully fledged molecular dynamics software package has not changed that much. As a matter of fact, the infrastructure presented in this manuscript shares many of the basic conceptual sections already present in [20].

It is important to also mention that MPI, one of the first parallel APIs, was not available until 1994. Up to that point, sequential computing was the norm.

Another famous molecular dynamics package is GROMACS [21], which was released in 1991, with the first version using MPI coming shortly after its release.

The GPU revolution

Many of these software packages were born more than a decade before the term GPU existed and were already quite established when the GPU became a mainstream general-purpose computing hardware. Naturally almost all of them have, at least partially, ported their functionalities to be accelerated by a GPU in some way. However, GPU programming poses such a wildly orthogonal paradigm that it makes the porting process slow and painful. Most

⁹ The first programmable, electronic, general-purpose digital computer.

CPU-centric molecular dynamics packages are already so complex that redesigning from the ground up to accommodate for the **GPU** is not an option and thus their inclusion can become awkward, sometimes causing so-called software rot.

It is therefore worth it to start from a clean slate every once in a while considering the new environments. One example of this is the general-purpose molecular dynamics software package HOOMD[22]. Born around the same time as CUDA itself HOOMD was designed from the ground up with the **GPU** in mind. Although HOOMD provides a CPU-only backend it is centered around the **GPU** and can work exclusively in it, this is crucial, as communication between the CPU and the **GPU** is quite expensive, so much that it can often negate the benefits of using a **GPU** altogether.

The **GPU** offers mind-blowing raw computing power when used correctly. Developing new algorithms and software infrastructures from scratch particularly designed for the **GPU** architecture is therefore essential to exploit this amazing technology to the fullest. This starts by taking the whole computation to the **GPU** in order to avoid the communication cost. On the other hand, at a library level, one of the most powerful tools exposed by the **GPU** is the **Fast Fourier Transform (FFT)**. It just so happens that the existing algorithms to compute the discrete Fourier transform numerically are a perfect fit for the **GPU**. We will exploit this by devising pseudo-spectral algorithms in which the majority of the computations take place in Fourier space.

Closed vs. Open ecosystems

Most commercial molecular dynamics packages share one foundational design principle; they are *closed* ecosystems. This means that, for a user, the only option to interface with the packages is to encode a simulation in them in its totality. In other words, it is awkward (or straight up not possible) to use these packages as external accelerators for an already established, in-house, code.¹⁰

A closed ecosystem allows to design the software under less assumptions, easing development. On the other hand, a closed ecosystem misses some opportunities in doing so. For instance,

10 If these software packages were hardware stores, a closed ecosystem would consist of the tools inside the store (the hammer, saw, etc) being chained to the counter, so you would need to bring your furniture into the store in order to fix it.

one of the first problems a soft matter software package must solve is the construction of a short ranged particle interaction list (a so-called neighbour list), which usually governs the overall computational cost of a simulation. Creating a public, library-like, interface exposing this list creation algorithm (in what we would call an open ecosystem) allows an user to take his existing code and accelerate it, as opposed to having to do it the other way around, that is porting the entirety of his simulation to the closed ecosystem.

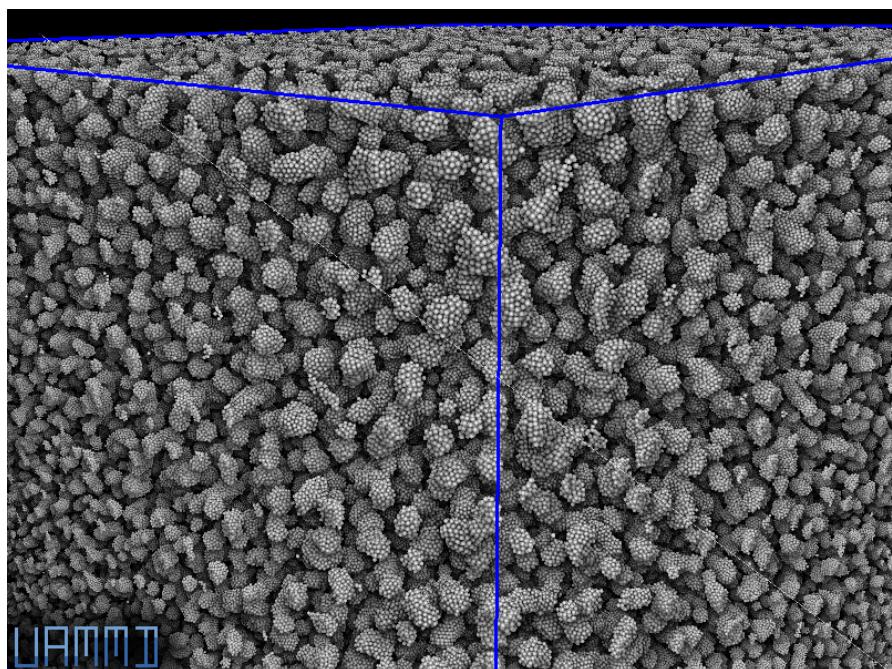
UAMMD attempts to provide an *open* ecosystem, exposing most of its underlying algorithms in a library-like fashion¹¹. Furthermore, while most packages require compilation into a binary or library in order to be used, **UAMMD** is a header-only framework. Meaning that the compilation of a source code using parts of **UAMMD** can be seamlessly blended in during the compilation process of another software.

¹¹ Following the hardware store analogy, the tools of an open ecosystem store would not only be unchained, but also available for rent. Furthermore, this would be an *open-source* hardware store (bear with me) so the cost of renting the tools is zero euros per day. In this kind of store, you can just borrow the tool(s) you need and fix your stuff at home.

Part II

UAMMD: DESIGN AND COMPONENTS

An overview of the UAMMD infrastructure.



2

THE FORMATION OF SUBPROBLEMS

The Trinity test, the first atomic bomb detonation in history, was carried out in 1945. The scientists of the Manhattan project dumbfoundedly observed how the detonation unraveled from a base afar. Among them was Enrico Fermi, the so-called architect of the nuclear age, who was 44 at the time. Fermi had absent-mindedly torn apart a sheet of paper moments before the detonation took place and held the pieces in his hands. Forty seconds after the blast the then withered shock-wave hit the crew. At the first sign of the shockwave Fermi, who had placed his hands above his head in expectation, released the paper bits, which landed a couple of meters past him. After a brief moment of meditation, Fermi announced to the crew that the detonation had released about 10 kilotons of energy. It took weeks of analyzing the data from the plethora of sensors available on site to confirm Fermi's estimate of the energy, which is now believed to have been near 20 kilotons.

Fermi had mastered (and practically invented) these back-of-the-envelope calculations, which are more precise than a guess but are not mathematical proofs. He knew how to subdivide complex, intractable, problems into many small and manageable ones. In order to teach this technique to his students, Fermi prompted them with a kind of question that has become known as a *Fermi problem*. A Fermi problem presents itself as an apparently unanswerable question. For instance: “*If I attach a blood vial to the wheels of my bike, how fast do I need to go and for how long for the plasma to separate from the blood cells?*” or “*How many hairs does a polar bear have?*”. At first sight it seems that we simply do not have enough information at hand to solve these problems. However, if we break them down into small, answerable, subproblems we find that we can give a pretty good estimate in a short time: “*What is the skin area of a bear?*”, “*How many hairs per squared centimeter does a bear have?*”. These can in turn be interpreted as Fermi problems themselves and further subdivided: “*What is the average*

height of a bear?”, “Assuming a bear is somewhat cylindrical, what is its radius?”, “What is the diameter of a hair?”¹, etc.

Surely we can apply Fermi’s teachings and *Fermate* our endeavors into subproblems too. Consider the development of a software capable of reproducing the dynamics of an ionic solution in a cellular membrane channel. This poses a really convoluted simulation, involving electrostatics, fluctuating hydrodynamics, steric interactions, bonded ligatures, etc.². As we will see along this manuscript, some of these algorithms are quite complex on their own, depending on lots of moving parts. The mathematical and theoretical machinery required to model something like that appears to be in the realm of a theory of everything. It sure sounds like a Fermi problem.

Still, we could try to design, from the ground up, a software that can perform this hypothetical simulation. However, this top-down approach is fundamentally flawed. The final product will probably have a lot of internal dependencies, many of which are probably unnecessary. In software this means that the code will be hard to adapt, extend or even scarier, to maintain.

We humans are not good with complex things (which is probably the reason why complex and complicated share the same etymology) and, as can be glimpsed throughout **UAMMD**, my solution to this problem is often to either Fermate it or remove it altogether.

Like we did when counting hairs, we start by dividing the original problem into subproblems in a hierarchical fashion (see Fig. 2.1). The top of the tree represents the most general (or abstract) aspect of the problem, and subdivisions offer increasingly specific concepts. So instead of designing our framework with an ionic solution in a membrane channel in mind, we start at the top and then hierarchically specialize. What constitutes the “top”, or *soul*, of the problem is of course highly subjective and thus the contents

¹ Let us assume bear hairs grow in a near close-packed configuration (it’s cold at the poles, you know) where each hair is on the order of $20\mu\text{m}$ in radius. This yields $1\text{cm}^2/((20\mu\text{m})^2\pi) \sim 100\text{Khairs/cm}^2$. I have not found the precise number, but we can compare with a sea otter (the mammal with the densest fur) which has between $100 - 400\text{Khairs/cm}^2$. This gives me some confidence about the validity of our order of magnitude estimation.

² A simulation easily handled by **UAMMD**, by the way.

of the next sections should be considered little more than my humble take on the problem³.

In particular, we can find the root by removing assumptions until we get to a point where only one (or a small number of them) remains. Therefore, we do not design in terms of “ions”, “cells” or “molecules”, rather in terms of a vague concept called “particles”. Following Fermi’s teachings we embrace the (redundant) complexity of complex fluids instead of trying to fight it.

The foundational concepts of **UAMMD** are supported on a handful of (deliberately) vague assumptions which can be summarize in four:

- S.1** The *System* assumption: The code will run primarily on a **GPU** (the most limiting assumption in the development process).
- S.2** The *ParticleData* assumption: Simulations are based on the state of “particles” (whatever a particle and its state mean).
- S.3** The *Integrator* assumption: The state of these particles changes in time.
- S.4** The *Interactor* assumption: Particles can interact with each other and with an external influence.

These are the four pillars depicted in Fig. 2.1. The software framework exposes these assumptions through four foundational classes (i.e objects in programming), which are, in order; *System*, *ParticleData*, *Integrator* and *Interactor*.

Excluding the first assumption, which is unrelated to physics, it is straightforward to see that our initial ionic solution simulation fits into this criterion. As a bonus, we have ended up with a basis that also describes things like “a gas of argon atoms”, “a group of planets orbiting a star” or “a virus filled with proteins exploding due to the force exerted on it by an AFM tip”⁴.

At the top of the tree in Fig. 2.1 we find the *System* object which takes care of setting up the computational environment for the rest of players. Usually the user does not need to interact with this

³ Note, however, that I am far from the first to come up with this logical separation. Slightly different separations and with different names are present in basically every molecular dynamics software package.

⁴ An oddly specific example describing a real study that is being carried out with **UAMMD** at the moment of writing by a collaborator group.

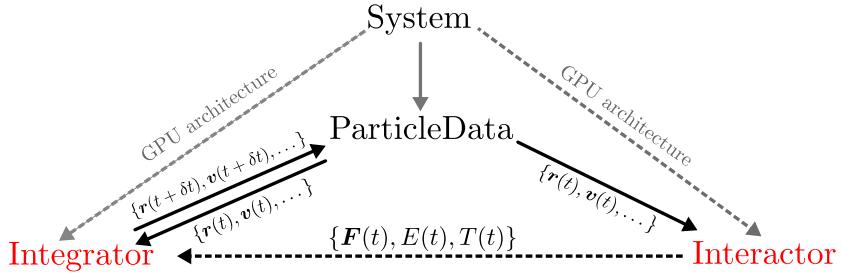


Figure 2.1: The basic hierarchy of concepts (and code) in [UAMMD](#), represented by a series of modules connected by arrows that convey the direction of the data flow. *System* holds information about the actual physical hardware, and all the entities below rely on it to interact with the environment. *ParticleData* (a “real class” in the code) stores all the information about the particles in the simulation, such as positions (r), velocities (v), mass, etc, as well as other properties like the current forces acting on it (F), energy (E) or virial (T). Modules below (explained later) can request, at their leisure, a list with any of the particles properties. *Integrators* and *Interactors* are interfaces (called virtual classes in programming terms), are in charge of, respectively, forwarding the simulation one step in time and computing the forces, energies and/or virials acting on each particle. These “Interfaces” (drawn in red) are abstract objects that cannot be instanced on their own but rather must be inherited. For instance, Brownian Dynamics (section. 17) would be an *Integrator*, while a module that computes gravitational forces would be an *Interactor*.

object, which is automatically started when needed. [UAMMD](#)’s online documentation can be visited to learn more about the *System* object (see Appendix D)

The second assumption is represented in the code base via the *ParticleData* object, which we discuss in 3.4.

Then, we can focus on each of the last two assumptions and apply this separation again. For instance, instead of grouping all interactions (fourth assumption) into a single entity (class) that deals with all possible interactions, we can split the concept between general things like short-ranged, bonded or long-ranged interactions. In section 8 we explore deeper along the *Interactor* branch to discuss the different algorithms and programming interfaces related with particle interactions (see Fig. 8.1).

Similarly, in section 13 we break down the *Integrator* branch (bottom left of Fig. 2.1) into the several ways the state of the particles can evolve (see Fig. 13.1).

During the following chapters, we will discuss in detail this conceptual and programmatic hierarchy and follow its branches down to the leaves. These “final” computational objects representing the leaves of UAMMD encapsulate the actual algorithms and physics in the form of isolated, individual modules. A “module” can be understood as a piece of code which receives information, processes it, and sends back some output. The internal complexity of a module is usually much larger than that related with its input and output. Input and output are the essence of the communication between modules.

In general, these modules will communicate only via the closed loop described by *ParticleData-Integrator-Interactor* in Fig. 2.1.

One of the main unwritten lessons we will glean is that it is never a good idea to introduce dependencies that climb up through the tree. Just to give some examples of this:

- A neighbour list should not assume it is going to be used to compute the forces of a group of repulsive particles.
- The existence of a hydrodynamics integrator module should not incur the modification of the *Integrator* interface.

In other words, conceptual complexity and problem specificity should increase only downwards through the tree. This forces us as developers to modify a certain module only to generalize it, and never to specialize it. If a module appears to require being specialized, it is almost always a sign that either it should actually be generalized or exist as a separate entity beneath the initial one. This is what makes UAMMD so modular and extensible. See figures 8.1 or 13.1, the next lower branches. UAMMD carefully avoids these kind of interdependencies. As a matter of fact, if one were to inspect the commit history of the UAMMD repository, they would find that many times the most relevant changes come in the form of *removing* code, not adding it. What is interesting

3

about this is that, maybe counterintuitively, this code removal almost always has resulted in a more generic module.

IT'S IN THE NAME

It is common knowledge that good software starts with a good acronym. **UAMMD** stands for Universally Adaptable Multiscale Molecular Dynamics in an attempt to abridge the simple-yet-powerful four assumptions in the previous chapter¹. Each of the words carries a different aspect of the framework's philosophy. In this chapter, we will discuss each of them.

3.1 UNIVERSALITY

Complex fluids are a prime examples of a multiphysics system. As such, multiphysics are deeply ingrained in **UAMMD**. Our framework can potentially take into account physical phenomena emerging from seemingly disparate fields. One of the key features for *universality* is to be able to communicate between Eulerian (fields) and Lagrangian ("particles") descriptions. This hybrid approach permits **UAMMD** to efficiently solve problems involving hydrodynamics, electrostatics, magnetism, light-matter interactions, and more...

In chapter 18, we will describe a series of Eulerian-Lagrangian algorithms that incorporate the effect of a solvent in a group of submerged particles (hydrodynamics) by describing it explicitly. On the other hand, we will adapt these techniques to compute electrostatics in chapter 24 by describing an explicit *charge-density field*.

Quantum physics has not been mentioned yet and for the time being we have limited our toolset to the classical world. Thus, staying in the realm of classical physics could be considered a fifth (soft) assumption.

It is however important to stress that the central (class) object in **UAMMD** are the particles (see below). In fact, the fourth assumption in chapter ii states that particles are expected to interact with each other or via an external influence. When particles

¹ Naturally, the fact that the acronym for the Universidad Autonoma de Madrid is there is merely a coincidence.

are submerged in a fluid, its effect on the former can be considered an external influence. Even when the fluid itself is also affected by the presence particles, as is the case with hydrodynamics.

3.2 ADAPTABILITY

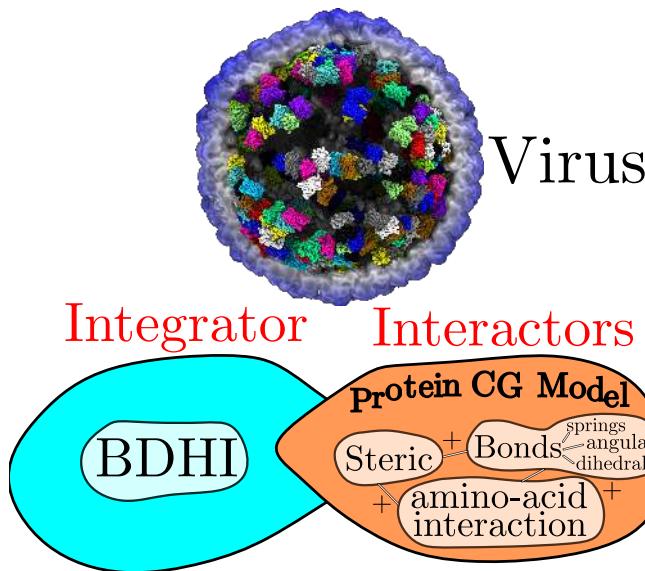


Figure 3.1: A simulation is constructed by putting together several modules. In this instance, the simulation of a virus submerged in water could be constructed by joining a hydrodynamics *Integrator* module with several *Interactors* describing the interactions between particles, such as sterics, bonds, electrostatics,...

Modularity is another central tenet in the philosophy behind **UAMMD**, which provides its adaptability.

Even when the internal complexity of each separate module can be quite great it never bleeds into the rest of the modules, which can remain oblivious to each other. This allows each module to worry only about solving one specific problem (hydrodynamics, electrostatics,...).

Moreover, this makes **UAMMD** extremely adaptable, since new modules can be added without disturbing the ecosystem.

In **UAMMD** a simulation is described by connecting a series of *Interactors* with an *Integrator*. In Fig. 3.1 we can see an example of a simulation which connects a hydrodynamics *Integrator* (such

as the ones described in sec. 18) with an *Interactor* that deals with sterics (see sec. 10), another one that computes electrostatics (see sec. 24), and so on and so forth. The communication between these modules is minimal, being mostly restricted to the closed loop in Fig. 2.1. Although nothing technically prevents two *Interactors* from explicitly sharing information, this never happens in **UAMMD**.

This does not imply that each module should exist in a completely isolated environment, since a plethora of utilities are available in **UAMMD** to aid with the typical problems that arise in complex fluids. For instance, a module in need of a neighbour list does not need to write one. Rather, it can simply make use of one of the solutions already implemented (see chapter 11).

Adaptability also means that, when it makes sense, a module can be specialized from the outside. For instance, the *Interactor* module that computes bonded forces (particles joined by springs or similar ligatures) is not specific to a particular potential. As users, we can specialize it, via the use of C++ templates, to work with a harmonic, FENE[23], Morse[24], or an arbitrarily complex potential.

The use of metaprogramming (in the C++ sense) reaches every corner of the code base, in many cases allowing to customize (or hack) a utility or module beyond its intended scope without having to modify a single line of it.

UAMMD is presented as a CUDA/C++ header-only library and it is mostly self-contained in terms of dependencies. The use of the word library is not chosen lightly here. We have discussed how an *Integrator* is joined with a collection of *Interactors*, maybe suggesting that the latter cannot function without the former. This is not the case since our framework provides a completely open ecosystem, facilitating (and actually encouraging) the incorporation of **UAMMD** modules outside their *normal* environment. All the utilities in **UAMMD**, including *Interactors*, *Integrators*, neighbour lists, linear solvers and many more² can be used in an isolated way from outside the code base. This also means that parts of **UAMMD** can be used as accelerators in already established codes. For instance, an already written simulation code can include one of our fast GPU hydrodynamics implementations into their description with just a few lines of code.

² A list of all the modules in **UAMMD** at the time of writing can be found in Appendix G.

In the past, we have explored this facet of the project by providing to collaborators some of our utilities and solvers as stand-alone Python interfaces.

The modularity of **UAMMD** is reflected directly in the code, where modules can be created independently of each other as it will be evidenced by the example codes throughout this manuscript. Be it an *Integrator* (hydrodynamics, molecular dynamics,...) or *Interactor* (short range, bonded,...), all modules in **UAMMD** can be created and stored individually.

3.3 MULTISCALE

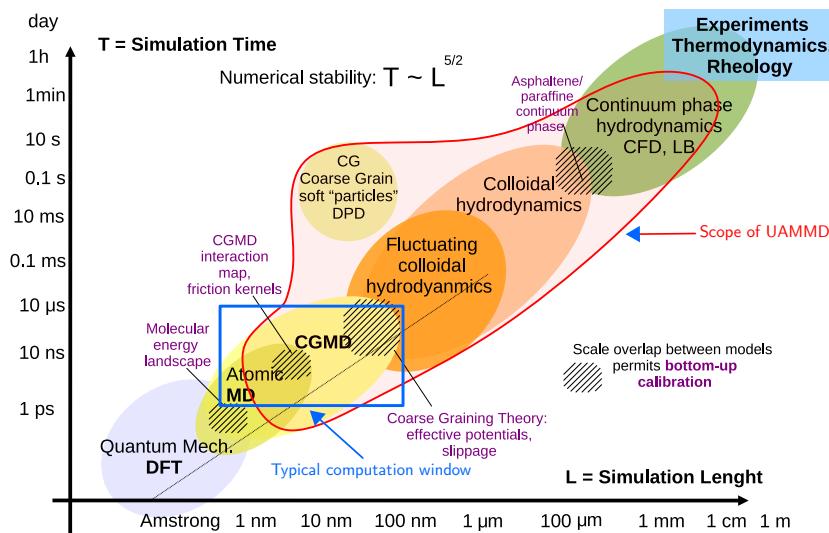


Figure 3.2: The spatio-temporal landscape and its numerical techniques. Different techniques are applied to simulate the different scales involved in a biological system or, in general, a complex fluid. Image courtesy of Rafael Delgado-Buscalioni.

Describing a given system in a reductionist manner, by solving the dynamics of every atom involved, is of course unfeasible. The **UAMMD** infrastructure can accommodate several levels of description by making use of coarse-graining techniques. Coarse graining allows to realistically describe a system using fewer degrees of freedom. In general, a level of description is tied to a certain time

scale and is characterized by a series of relevant variables coupled via a (usually stochastic) dynamic equation.

Although **UAMMD** is not restricted to soft matter systems, this work will concentrate mainly on such systems. In this regard we can identify a series of clearly separated levels of description. It is worthwhile to summarize them here since this manuscript will discuss many of them in detail. They are furthermore available as **UAMMD Integrator** modules. A representation of the different levels of description is available in Fig. 3.3, while the numerical methods employed to numerically study each of them can be seen in Fig. 1.1. Figure 3.2 provides an overview of the different fields of soft matter simulation and its characteristic spatio-temporal scales. We can distinguish between the following levels of description (from most to least detailed):

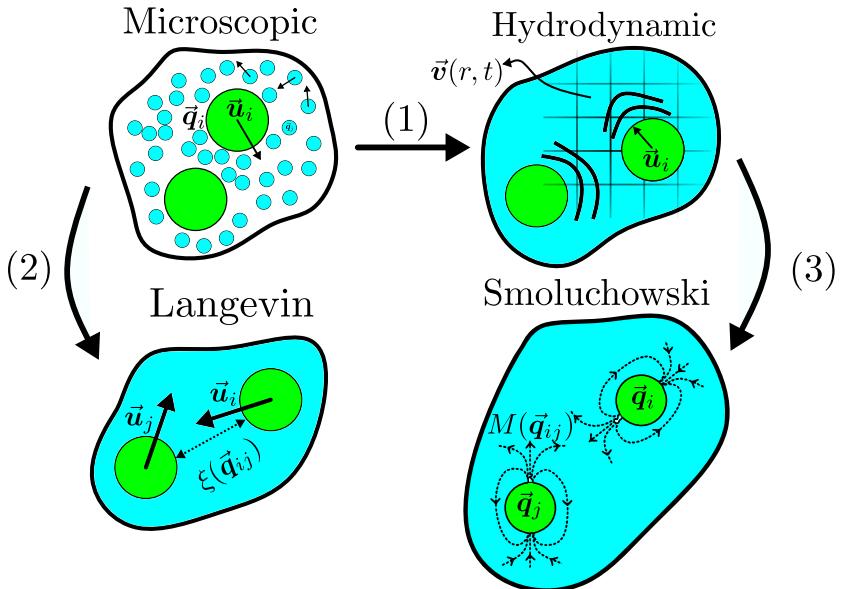


Figure 3.3: The different levels of coarse grained description available in **UAMMD** for soft matter simulations. \vec{q}_i represents the positions of particles, \vec{u}_i their velocity. $\vec{v}(r, t)$ represents an Eulerian fluid velocity field, $M(\vec{q}_{ij})$ a mobility tensor and $\xi(\vec{q}_{ij})$ a friction kernel. Arrows represent standard routes for the formal bottom-up derivation using coarse-graining theory. In order, we have: (1) in [25], (2) in [26] and (3) in [27].

1. Microscopic: The relevant variables are the positions and moments of every particle in the system, including the solvent and solute particles. This level is dominated by the mean collision time between the particles, of the order of $\tau \sim 10^{-12} s$. The (purely Lagrangian) Newtonian equations of motion describe the dynamics of the system and [Molecular Dynamics \(MD\)](#) can be used to solve them. We explore this description in chapter [14](#).
2. Hydrodynamic: The degrees of freedom of the solvent particles are omitted and turned into hydrodynamic fields. The Newtonian equations for the solute particles are coupled with the (fluctuating) Navier-Stokes equations that describe the solvent in what constitutes an Eulerian-Lagrangian description. An infinitesimal region of space represents a group of solvent particles, whose individual degrees of freedom are lost. The characteristic time is now controlled by the speed of sound of the solvent ($\tau \sim 10^{-9} s$) and the vorticity ($\tau \sim 10^{-6} s$). We study the hydrodynamic level in chapter [18](#).
3. Langevin: Particles move so slowly compared to the solvent characteristic times that hydrodynamic interactions are effectively instantaneous. Only the positions and momenta of the particles remain as relevant variables (Lagrangian description once again), which obey a Langevin equation. The characteristic decorrelation time of the particle's velocities governs this scale with $\tau \sim 10^{-5} s$. We explore this inertial level of detail in chapter [15](#).
4. Smoluchowski: Particle positions change slowly enough that their velocity decorrelates effectively instantaneously, inertia can be disregarded and positions are the only relevant variable. The dynamics are described by the [Brownian Dynamics \(BD\)](#) equations of motion (an over-damped Langevin equation). The diffusion time of the particles, $\tau \sim 10^{-3} s$, dominates this level. We study this limit in chapter [17](#).

Above these levels of description particles are no longer relevant. Here we have the Fick level, in which a continuum particle concentration field is the relevant variable, and the thermodynamic level, in which the relevant variables are the macroscopic dynamical

invariants of the system (mass, energy, volume, temperature,...). We can use these descriptions to study time scales of seconds (Fick) up to infinity (thermodynamics). However, given that **UAMMD** is a particle-centric framework, these descriptions will not be considered in this manuscript.

With the exception of the microscopic level, in which particles experience effective interactions that come directly from quantum effects, every other level emerges as a coarse grained description of the previous one.

Going from one level to the next incurs a loss of information (AKA degrees of freedom, e.g the hydrodynamic level cannot track the positions/momenta of the microscopic solvent particles). These eliminated, fast, degrees of freedom are modeled (reintroduced) as a drag coupled with thermal noise. In the following sections, we will see how dissipation is intimately related with thermal noise via the fluctuation-dissipation theorem, such relations ensure that the coarse-grained system attains the correct equilibrium distribution.

3.3.1 The Fokker-Planck Formalism

One of the basic theories for coarse graining is embedded in the Fokker-Planck formalism. Let us focus on a certain relevant slow, degree of freedom (variable) that interacts, in a *random* way, with many other (fast) degrees of freedom. In general, we will eliminate fast variables from our description and reintroduce them as fluctuations. The **Fokker-Plank Equation (FPE)** describes the evolution of the probability distribution, $P(\mathbf{x}, t)$, of a set of relevant or slow variables, $\mathbf{x} = \{x_1, x_2, \dots, x_N\}$ in a quite general way:

$$\partial_t P = -\boldsymbol{\partial}_{\mathbf{x}} \cdot \mathbf{J} \quad (3.1)$$

Where $P_0 := P(\mathbf{x}_0, t_0)$ is known, \mathbf{J} is the flux of probability vector and $\boldsymbol{\partial}_{\mathbf{x}} \cdot \mathbf{J} := \nabla_{\mathbf{x}} \cdot \mathbf{J} = \sum_i \partial J_i / \partial x_i$ represents the divergence (note that the derivation is made with respect to the slow variables by using supervectors, i.e. all the variables and their coordinates[27]). Using the backwards Kramers-Moyal expansion [28] truncated at

the second term³, the probability flux vector, \mathbf{J} , can be expressed as,

$$\mathbf{J}(\mathbf{x}, t) := \widetilde{\mathbf{D}}^{(1)}(\mathbf{x}, t)P - \partial_{\mathbf{x}} \cdot [\mathbf{D}^{(2)}(\mathbf{x}, t)P] \quad (3.2)$$

The first term is called the total drift or transport term ($\widetilde{\mathbf{D}}^{(1)}$ represents the total drift coefficients) and the second one the diffusion term (being $\mathbf{D}^{(2)}$ the diffusion coefficients). It will later come in handy to express Eq. (3.3) using another definition for the drift coefficient, so that

$$\mathbf{J} = (\mathbf{D}^{(1)} - \mathbf{D}^{(2)}\partial_{\mathbf{x}}) P, \quad (3.3)$$

where we used the chain rule to get the diffusion coefficients outside of the derivative, placing the extra diffusive term (referred to as thermal drift) inside the definition of a systematic drift coefficient, $\mathbf{D}^{(1)} := \widetilde{\mathbf{D}}^{(1)} - \partial_{\mathbf{x}} \cdot \mathbf{D}^{(2)}$. The FPE in Eq. (3.1) simply represents the conservation of the conditional probability $P(\mathbf{x}, t)d\mathbf{x}$ of finding a set of variables \mathbf{x} in the range $[\mathbf{x}, \mathbf{x} + d\mathbf{x}]$. The generality of Eq. (3.1) makes it applicable in a wide range of situations (a suspension of colloidal particles, fluctuations in electrical circuits, mechanical oscillators and chemical reactions just to name a few).

We can use the probability distribution P to compute the ensemble average of a certain function $f(\mathbf{x}(t))$ as

$$\langle f \rangle(t) = \int P(\mathbf{x}, t)f(\mathbf{x})d\mathbf{x}. \quad (3.4)$$

Its time evolution can be then expressed as

$$\begin{aligned} \frac{d\langle f \rangle}{dt} &= \int f \partial_t P d\mathbf{x} = \int P \left(\mathbf{D}^{(1)} \partial_{\mathbf{x}} f + \partial_{\mathbf{x}} \cdot [\mathbf{D}^{(2)} \partial_{\mathbf{x}} f] \right) d\mathbf{x} \\ &= \langle \mathbf{D}^{(1)} \partial_{\mathbf{x}} f \rangle + \langle \partial_{\mathbf{x}} \cdot [\mathbf{D}^{(2)} \partial_{\mathbf{x}} f] \rangle, \end{aligned} \quad (3.5)$$

where the second equality comes from using integration by parts and the divergence theorem in an straightforward way [28]. Additionally, we assumed natural boundary conditions (closed system). The choice of names for the coefficients becomes evident if we now

³ The Pawula theorem [29] can be invoked here, which proves that all terms beyond the second are meaningless. In particular, the Pawula theorem states that there are only three possible truncations of the Kramers-Moyal expansion: (1) truncating at the first term, in which case the process is deterministic; (2) truncating at the second term, in which case the process is diffusive; (3) keeping all terms up to infinity. However, keeping any term beyond the second results in a non-positive probability density.

study the temporal evolution of the first moments of the relevant variables. The first one (the mean) yields

$$\frac{d\langle \mathbf{x} \rangle}{dt} = \underbrace{\left\langle \widetilde{\mathbf{D}}^{(1)} \right\rangle}_{\text{Drift}} = \overbrace{\left\langle \mathbf{D}^{(1)} \right\rangle}^{\text{Systematic drift}} + \underbrace{\left\langle \partial_{\mathbf{x}} \cdot \mathbf{D}^{(2)} \right\rangle}_{\text{Thermal drift}}. \quad (3.6)$$

The evolution of the second moment (matrix), $\langle \mathbf{x} \otimes \mathbf{x} \rangle$, can be obtained following a similar route as that leading to Eq. (3.5) (see [28]), the result is

$$\frac{d\langle \mathbf{x} \otimes \mathbf{x} \rangle}{dt} = \left\langle \widetilde{\mathbf{D}}^{(1)} \otimes \mathbf{x} \right\rangle + \left\langle \mathbf{x} \otimes \widetilde{\mathbf{D}}^{(1)} \right\rangle + 2 \underbrace{\left\langle \mathbf{D}^{(2)} \right\rangle}_{\text{Diffusion}}. \quad (3.7)$$

Here \otimes represents the tensor product. In section 15 we will study the Langevin equation, a case in which the relevant variables are the velocities (and positions) of the particles (which are surrounded by a bath of fast moving water molecules). On the other hand, in section 17 we will see that in Brownian Dynamics (the over-damped limit of the Langevin equation), the relevant variables are the positions of the particles (their velocities being the fast, eliminated, degree of freedom). Finally, the applicability of the Fokker-Planck formalism at the hydrodynamic level will be briefly discussed in chapter 18.

3.3.2 Fluctuation-Dissipation balance

A system in contact with a heat bath will experience both friction and fluctuations stemming from the elimination of a series of fast degrees of freedom. Both magnitudes are directly related to each other in equilibrium, as the equipartition theorem states that every degree-of-freedom has $k_B T/2$ kinetic energy, which requires that fluctuation and dissipation are in a certain specific balance.

We will see what the FPE tells us about the fluctuation-dissipation balance by studying the covariance matrix,

$$\boldsymbol{\sigma} := \langle \mathbf{x} \otimes \mathbf{x} \rangle - \langle \mathbf{x} \rangle \otimes \langle \mathbf{x} \rangle = \langle \delta \mathbf{x} \otimes \delta \mathbf{x} \rangle, \quad (3.8)$$

of the relevant variables and how it behaves at equilibrium. Here $\delta \mathbf{x} := \mathbf{x} - \langle \mathbf{x} \rangle$ is a fluctuation of the relevant variables around their averages. The time evolution of the covariance can be written as

$$\partial_t \boldsymbol{\sigma} = \partial_t \langle \mathbf{x} \otimes \mathbf{x} \rangle - \langle \mathbf{x} \rangle \otimes \partial_t \langle \mathbf{x} \rangle - \partial_t \langle \mathbf{x} \rangle \otimes \langle \mathbf{x} \rangle, \quad (3.9)$$

Let us also define $\delta\mathbf{D} := \mathbf{D} - \langle \mathbf{D} \rangle$ as the fluctuations around the average of the transport coefficients. Plugging Eqs. (3.6) and (3.7) into Eq. (3.9) we arrive at a compact expression for the time evolution of the covariance

$$\partial_t \boldsymbol{\sigma} = 2 \left\langle \mathbf{D}^{(2)} \right\rangle + \left\langle \delta\mathbf{x} \otimes \delta\widetilde{\mathbf{D}}^{(1)} \right\rangle + \left\langle \delta\widetilde{\mathbf{D}}^{(1)} \otimes \delta\mathbf{x} \right\rangle. \quad (3.10)$$

Consider now a linearization around the equilibrium state⁴, in which the transport coefficients can be decomposed as

$$\mathbf{D}^{(i)} = \left\langle \mathbf{D}^{(i)} \right\rangle + \left. \partial_{\mathbf{x}} \mathbf{D}^{(i)} \right|_{\text{eq}} \delta\mathbf{x} + \dots \quad (3.11)$$

Where i can refer to the drift or diffusion coefficients. Using Eq. (3.11) we can approximate

$$\delta\widetilde{\mathbf{D}}^{(1)} = \left. \partial_{\mathbf{x}} \widetilde{\mathbf{D}}^{(1)} \right|_{\text{eq}} \delta\mathbf{x} + O(\delta\mathbf{x}^2). \quad (3.12)$$

Let us now define the dynamic matrix as

$$\mathcal{H} := -\partial_{\mathbf{x}} \widetilde{\mathbf{D}}^{(1)}(\bar{\mathbf{x}}, t), \quad (3.13)$$

where $\bar{\mathbf{x}} := \langle \mathbf{x} \rangle_{\text{eq}}$ represents the equilibrium state of the relevant variables. So that

$$\delta\mathbf{x} \otimes \delta\widetilde{\mathbf{D}}^{(1)} = -\mathcal{H} (\delta\mathbf{x} \otimes \delta\mathbf{x}) + O(\delta\mathbf{x}^3). \quad (3.14)$$

In a stationary or equilibrium state we have $\partial_t \boldsymbol{\sigma}|_{\text{eq}} = 0$. Replacing Eq. (3.14) into (3.10) in equilibrium yields

$$2 \left\langle \mathbf{D}^{(2)} \right\rangle = \mathcal{H} \boldsymbol{\sigma}_{\text{eq}} + \boldsymbol{\sigma}_{\text{eq}} \mathcal{H}^T. \quad (3.15)$$

This key relation is known as the Fluctuation-Dissipation Balance, and it can be also used in steady, yet non-equilibrium, states [30].

We can now go a little bit further and investigate the evolution of the relevant variables during a small time interval, dt , which will allow us to derive a [Stochastic Differential Equation \(SDE\)](#). Let us assume that we know the values of the relevant variables at some initial point in time, $t_0 = 0$, so that $P(\mathbf{x}, t_0) = \delta(\mathbf{x} - \mathbf{x}_0)$ (where $\mathbf{x}_0 := \mathbf{x}(t_0)$).

If the time interval, $dt := t - t_0$, is small enough, we can safely assume that the transport coefficients remain mostly constant

⁴ Note that, at equilibrium $\mathbf{J} = 0$ and $P(\mathbf{x}, t) = P_{\text{eq}}(\mathbf{x})$.

($\langle \mathbf{D}^{(i)}(\mathbf{x}, t) \rangle_{t \in [0, t_0]} = \mathbf{D}^{(i)}(\mathbf{x}_0)$). Furthermore, we can neglect any terms beyond dt . Making use of the definition of derivative, we can now write Eq. (3.6) as

$$\langle \mathbf{x} \rangle(t_0 + dt) = \mathbf{x}_0 + \widetilde{\mathbf{D}}^{(1)}(\mathbf{x}_0)dt. \quad (3.16)$$

On the other hand, we have that $\sigma(t_0) = 0$ since we know \mathbf{x}_0 exactly. Using Eq. (3.16), (3.7) and the definition of covariance in (3.8) we can approximate

$$\sigma(t_0 + dt) = \langle \delta \mathbf{x} \otimes \delta \mathbf{x} \rangle = 2\mathbf{D}^{(2)}(\mathbf{x}_0)dt + O(dt^2). \quad (3.17)$$

Finally, we can decompose any function into its average and a fluctuation, so that $d\mathbf{x} = d\langle \mathbf{x} \rangle + \delta \mathbf{x}$. We can then write an SDE (using the Itô interpretation[31]) for the relevant variables

$$d\mathbf{x} = \mathbf{x}(t_0 + dt) - \mathbf{x}_0 = \widetilde{\mathbf{D}}^{(1)}dt + \delta \mathbf{x}. \quad (3.18)$$

Where the fluctuating part must satisfy Eq. (3.17). Furthermore, the diffusion and drag terms in the dynamic matrix, \mathcal{H} , are related via the fluctuation-dissipation balance in Eq. (3.15).

In future chapters, we will apply the Fokker-Planck formalism to the different levels of detail described in Fig. 3.3 and its related discussion.

3.3.3 Einstein relation

An extremely important relation between the transport coefficients appearing in $\mathbf{D}^{(1)}$ and the diffusion $\mathbf{D}^{(2)}$ can be derived from the equilibrium state, $\mathbf{J} = 0$ in Eq. (3.3). This relation was first defined by Einstein[32] with arguments similar to those exposed in what follows.

Consider the zero flux condition $\mathbf{J} = 0$, which is solved by the equilibrium distribution $P_{\text{eq}}(\mathbf{x})$ (independent of time)

$$\mathbf{J} = \mathbf{D}^{(1)}P_{\text{eq}} - \partial_{\mathbf{x}} \cdot (\mathbf{D}^{(2)}P_{\text{eq}}) = 0 \quad (3.19)$$

Thus

$$\partial_{\mathbf{x}} \ln(P_{\text{eq}}) = \frac{\mathbf{D}^{(1)}(\mathbf{x})}{\mathbf{D}^{(2)}(\mathbf{x})} \quad (3.20)$$

Now, quite generally, consider that $\mathbf{D}^{(1)} = -\mathbf{M}(\mathbf{x})\partial_{\mathbf{x}}U(\mathbf{x})$ where $\mathbf{M}(\mathbf{x})$ is the so called “mobility matrix” which relates forces

$(\mathbf{F} = -\partial_{\mathbf{x}} U)$ with the systematic part (mean) of the displacement $d\langle \mathbf{x} \rangle = \mathcal{M}\mathbf{F}dt$.

Note that $U(\mathbf{x})$ is the free energy of the system providing the equilibrium distribution $P_{\text{eq}} = \frac{1}{Z} \exp[-\beta U(\mathbf{x})]$. This relation can also be written as $\partial_{\mathbf{x}} \ln(P_{\text{eq}}) = -\beta \partial_{\mathbf{x}} U(\mathbf{x})$. Inserting $\mathbf{D}^{(1)} = -\mathcal{M}\partial_{\mathbf{x}} U$ into Eq. (3.20) and comparing with the equilibrium result, we conclude that

$$\frac{\mathbf{D}^{(1)}(\mathbf{x})}{\mathbf{D}^{(2)}(\mathbf{x})} = \frac{-\mathcal{M}(\mathbf{x})\partial_{\mathbf{x}} U}{\mathbf{D}^{(2)}(\mathbf{x})} = -\beta \partial_{\mathbf{x}} U \quad (3.21)$$

So that,

$$\mathcal{M}(\mathbf{x}) = \beta \mathbf{D}^{(2)}(\mathbf{x}) \quad \text{or} \quad \mathbf{D}^{(2)}(\mathbf{x}) = k_B T \mathcal{M}(\mathbf{x}) \quad (3.22)$$

Which is the celebrated Einstein relation. Note that the Einstein relation also relates fluctuations (governed by $\mathbf{D}^{(2)}$) with dissipation (determined by the inverse mobility, aka friction).

In colloidal systems $\mathcal{M}(\mathbf{x})$ is calculated from macroscopic (deterministic) hydrodynamics. For instance, the mobility of a single (isolated) sphere with a no-slip surface moving in a fluid at rest with velocity \mathbf{u} can be obtained [27] by studying the total traction created by the fluid stress on its surface, $\mathbf{F} = \oint \boldsymbol{\sigma} \cdot \hat{\mathbf{n}} dS$ where $\boldsymbol{\sigma}$ is the fluid pressure tensor. The result is the Stokes relation

$$\mathbf{F} = -M_0^{-1} \mathbf{u} \quad (3.23)$$

With the so-called “self-mobility” being $M_0 = (6\pi\eta a)^{-1}$ where a is the particle radius and η the fluid viscosity⁵. Note that the inverse of the mobility, $\xi := M_0^{-1}$, is the so-called friction coefficient.

Combining the Einstein (Eq. (3.22)) and Stokes (Eq. (3.23)) relations leads to the Stokes-Einstein diffusion coefficient

$$D_0 = \frac{k_B T}{6\pi\eta a} \quad (3.24)$$

a manifestation of Eq. (3.22) in the case of a single no-slip sphere moving through a fluid at rest.

3.4 MOLECULAR DYNAMICS

Thus far we have used the term “particle” without explicitly defining it. We interpret a particle as an arbitrarily coarse-grained

⁵ Note that perfect slip surfaces lead to $M_0 = (4\pi\eta a)^{-1}$.

4

simulation unit (atoms, groups of atoms, colloids, groups of colloids, buckets of water, planets...). In this sense, the term molecular dynamics does not refer to the numerical simulation technique (described in sec. 14). Rather, it should be interpreted as *dynamics of molecules*, or more appropriately, *particles*.

In UAMMD particles are the relevant simulation entity as specified by the *ParticleData* class (see section 6). When a fluid is present (either implicitly or explicitly) it is only as a medium to move the particles. Hence, *Integrators* work out the time evolution of particles, which interact via *Interactors*. However, our loose description of a particle allows assigning any kind of property to them. For instance, in the simulation of an ideal gas of atoms, describing the positions, velocities and masses of the atoms might be enough. But in a more sophisticated situation, like a solution of charged colloidal particles, we can include forces, energies, charges, torques, angular velocities, radii, etc.

INITIAL REMARKS AND OVERVIEW

This manuscript encases three distinct, somewhat interleaved, narratives: physics, algorithms and their implementations for a GPU (always under UAMMD). The lion's share of the author's contribution to the field has gone into the implementation (i.e UAMMD), which is not thoroughly discussed here. In UAMMD's online documentation (see Appendix D) the weight leans more heavily on implementation. A reader with a special interest in the implementation of a particular module should have the UAMMD code base at hand when going through the module's chapter in this manuscript. The UAMMD codebase is thoroughly documented, many times in an almost pedagogical manner. Most chapters beyond this point have a corresponding UAMMD module and are organized as follows; first the physical problem to solve is introduced and its mathematical foundations are laid out. Then, the relevant algorithms are discussed. If the discussed algorithm contains some novel aspects, its implementation is described in detail. Regardless, each chapter ends with a usage example of the module under the UAMMD infrastructure.

As previously stated the central workpiece in UAMMD are the particles (see Fig. 2.1). The *Integrator* solves the time evolution of

the particles and to that end, it requires the driving interactions (e.g. forces) which are, in turn, provided by the *Interactors*. For this reason, from the code perspective “forces” and “velocities” respectively pertain to *Interactor* and *Integrator* concepts. Grid-based methods generally consist in solving **Partial Differential Equations (PDEs)** for some spatio-temporal field in a mesh, i.e. in an Eulerian framework. **UAMMD** deploys Eulerian solvers for several purposes: i) to obtain the velocity field of the solvent in Eulerian-Lagrangian hydrodynamics and ii) to solve the force field generated by a distribution of source charges. Both schemes are based on pseudo-spectral methods, which shall be first introduced when explaining hydrodynamic *Integrators* in Chapter 18. The *Interactor* dealing with electrostatics in **UAMMD** (pseudospectral Poisson solver) is explained later, in Chapter 24.

Figure 4.1 presents an overview of the different layers of UAMMD, listing most of the tools (modules) that we will discuss in future chapters. These essential tools are enhanced by the power of C++ templates, which allow a user to introduce new potentials, interpolation kernels,... into the existing modules without modifying them. In many instances it is even possible to generalize the scope of these tools. For instance, spreading/interpolation in a non-regular grid (discussed in chapter 21), or abusing a neighbour list’s (chapter 11) packing efficiency to perform some novel sparse matrix multiplication. Inside UAMMD, the communication between the different modules (which is limited to a minimum to reduce internal dependencies) is carried out via the root, *ParticleData*, and the other foundational modules already introduced in chapter 3 (see Fig. 2.1).

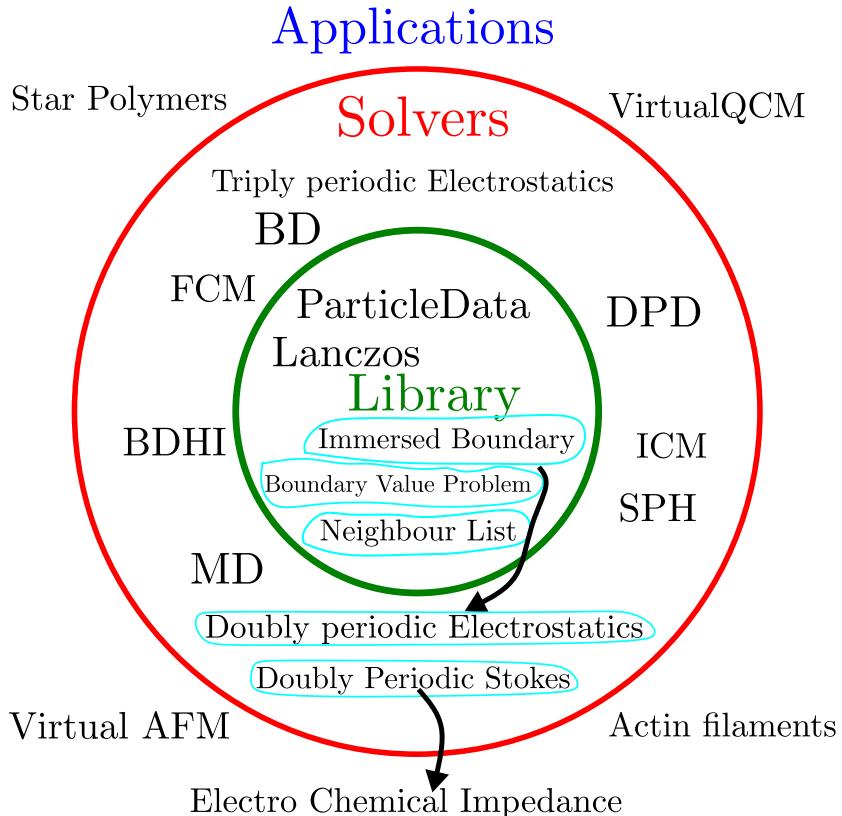


Figure 4.1: A bird's-eye of the different conceptual layers of the UAMMD infrastructure. Many (but not all) of the implemented modules are represented inside the red circle, we will discuss all of them through this manuscript. At the innermost level, the library level, we have the basic tools and algorithms on which the rest of the project is supported. The next layer is the solver layer, with modules for particle interaction and dynamics. In general, all these modules are black-box input/output modules that can be used in any other code in a library-like fashion as external accelerators (especially the ones at the library level). Outside the red circle we find the application layer, in which several of the inner tools are joined to study the physics of a new system. For instance, to study the phenomenon of electrochemical impedance we need electrostatics (interactions) and hydrodynamics (dynamics) in a slit channel (chapters 25 and 23, respectively), the modules inside blue circles are connected first to construct the interaction and dynamics modules, which in turn are used to set up a simulation.

The rest of the manuscript is organized in three main parts: In the first one (part [iii](#)) we will go through the different techniques used to solve particle interactions and dynamics (as described by the different levels of detail introduced in chapter [3.3](#)) with a particular focus on their adaptation and implementation in a [GPU](#) architecture. Whenever a new technique/algorithm is introduced an accompanying code example showing how to use it in [UAMMD](#) will be present. In the second part (part [iv](#)) we will go through a series of novel algorithms that have resulted from the development of this thesis. Like before, [UAMMD](#) code examples will always be accompanying the description of the algorithms. Finally, in the third part (part [v](#)) we will see a summary of works (publications in scientific journals) in which [UAMMD](#) has played a role either as a simulation engine or as an accelerator for an already established external code. The manuscript ends with some future directions and conclusions (chapter [30](#)), listing ongoing research and future endeavors for the [UAMMD](#) infrastructure.

Part III

A VOYAGE THROUGH NUMERICAL SPACE-TIME

Let us go through the different numerical techniques that are used to simulate the different ranges within the spatio-temporal landscape.

5

From this point on in the manuscript we will see descriptions of GPU algorithms and code examples written in CUDA/C++. Although the code examples can be mostly omitted without missing the overall message of the text, understanding the jargon, rationale and details of the described algorithms will require at least basic knowledge of the GPU programming model. If the reader is unfamiliar with either the C++ programming language or the GPU he or she can find an introduction to the basic concepts in Appendix A. While not a full-fledged guide, Appendix A has been designed to provide the bare minimum required to understand most of the GPU and programming related concepts laid out in the rest of the manuscript.

BUILDING AN UAMMD MODULE

Throughout this manuscript, we will typically create a module named “Module” using instructions similar to those in code 1. “Module” will be called, for instance, *VerletNVE* if it is an *MD Integrator* (see sec. 14.2) or *BondedForces* if it is a ligature (joining particles with springs) *Interactor* (see sec. 12).

```
#include <uammd.cuh>
//Additional includes when necessary. The definition
//of the UAMMD structure can be found in the next
//example, the code of which should be included
//here.
//A module named "Module" is created and returned.
auto createModule(UAMMD sim){
    //Some parameter preparations...
    return std::make_shared<Module>(/*Required
    // parameters*/);
}
```

Source Code 1: Template code for the creation of a module.

The argument of the function *createModule* in example code 1, of type *UAMMD*, is simply an aggregate (a *struct* in C++) of recurrent parameters written for this manuscript. In particular, this structure can be defined as in code example 2, which will serve as an implicit preamble code for the rest of the examples in this

manuscript¹. It is important to stress that all the code examples in this manuscript can be compiled (with the exception of source code 1) by including the preamble in code 2 (a.i. copy/pasting it at the start) and defining, inside the `Parameters` struct, the parameters required by the example (for instance, example code 15 will require two parameters: `real dt` and `real targetEnergy`).

```
#include <uammd.cuh>
using namespace uammd;
//An aggregate of parameters required by the relevant
// modules in the code
struct Parameters{
    //real dt;
    //...
};

//This structure packs a Parameters along with a
// ParticleData instance
struct UAMMD{
    std::shared_ptr<ParticleData> pd;
    Parameters par;
};
```

Source Code 2: Definition of the UAMMD auxiliary structure created for this manuscript. Other examples making use of the `UAMMD` struct need this snippet as a preamble to compile.

PARTICLE DATA

6

UAMMD is a GPU software, which requires special considerations. In particular, given that the GPU works as a co-processor with its own separated memory space, when allocating or accessing memory we must specify *where* that memory resides. This location can sometimes be subtle when looking at an isolated piece of code. The separation of memory spaces can become a nuisance especially when dealing with quantities that must naturally be accessed from both the CPU and the GPU.

¹ Incidentally, I often use this same construction when writing simulations using UAMMD, as evidenced by the examples in the UAMMD repository.

The *ParticleData UAMMD* class¹ hides the two memory spaces from the user by handling them internally, ensuring that the CPU and GPU versions of the arrays it provides are synchronized when requested. This is reflected mainly when requesting a particle property from *ParticleData*, which requires to specify both a location (CPU or GPU) and a usage intention (read, write or both). *ParticleData* can then use this information to provide an up-to-date reference to the data in the correct memory space. In CUDA/C++, a function is executed on the CPU unless it is explicitly marked as a GPU one (see Appendix A for more information about CUDA/C++ programming). Accessing GPU memory in a CPU function is thus illegal and results in the program crashing.

ParticleData can provide handles giving access to the different particle properties. When requesting a handle to a property from *ParticleData* with the intention of writing the requesting entity becomes the sole owner of it until the handle is released (destroyed or out of scope). If some other part of the code tries to access the same property (while another part still has the write handle) an error will be issued, since this could result in both versions of the property becoming out of sync. The solution is to simply refrain from storing the *ParticleData*-issued handles (for example as members of a class), requesting the handles just before use and destroying them when they are no longer needed. Furthermore, trying to use a handle for a different intention than requested (choosing the GPU as the device but accesing from the CPU or writing to a handle that was requested for reading) will result in either undefined behavior or the code crashing². The example code 3 showcases the basic usage of the *ParticleData* class. In particular, showing how to access and modify particle properties.

```
#include <uammd.cuh>
using namespace uammd;
int main(){
    //Creation requires a number of particles
    int numberParticles = 1e6;
    auto pd =
        std::make_shared<ParticleData>(numberParticles);
```

1 Note that since UAMMD usually exposes a so-called module via a single C++ class we will often use the words *class* and *module* as synonyms.

2 Whenever is possible UAMMD will throw an exception instead of crashing.

```

//An arbitrary particle index
int someIndex = rand()%numberParticles;
//A handle to any existing
// property can be requested like this
auto positions = pd->getPos(access::cpu,
→ access::write);
//Positions are stored as a 4-dimensional number
→ with elements x,y,z,w
//The fourth element, w, represents a "color" or
→ "type" and is largely ignored by uammd
//Let's set some position to x=1, y=2, z=3 and
→ w=0:
positions[someIndex] = {1,2,3,0}; //Legal access
auto masses = pd->getMass(access::gpu,
→ access::write);
//This would be an illegal access, since masses was
→ requested for the GPU:
masses[someIndex] = 1.0;
auto charges = pd->getCharge(access::cpu,
→ access::read);
//This would be an illegal access, since charges
→ was requested for reading:
charges[someIndex] = -1.0;
return 0;
}

```

Source Code 3: Creating and using an instance of *ParticleData*.

Adding new particle properties in **UAMMD** amounts to including them in a special list (in the source code *ParticleData.cuh*, to be precise). Many particle properties are already available in this list at the time of writing, such as positions, velocities, forces, mass, radius and many more. Each particle is assigned a name when *ParticleData* is created. *ParticleData* refers to this special property as “Id” (and exposes it just like the rest, with a corresponding `getId(access::gpu, access::read);` function). The name, or id, property has the unique quality of being immutable, meaning that it can only be requested for reading. **UAMMD** can under some circumstances change (reorder) the memory location of the particles³ (so that a particle named *id* that started at some index *i* would no longer be located at index *i*). Naturally, it is possible

³ For instance to increase the data locality in memory for particles close in physical space.

7

to query *ParticleData* for the index of a particle given its name⁴ (the reader can refer to the online **UAMMD** documentation, see Appendix D, to learn more about this). Note that in general in a GPU-centric algorithm, when we assign a worker (be it a thread or a thread block) to each particle, given that each worker runs asynchronously with the rest the specific order of the particles in memory becomes irrelevant from an algorithmic point of view.

Besides working as a multi container, *ParticleData* offers a series of functionalities overlooked here for simplicity (the reader can find more information in Appendix D). However, it is worth mentioning that *ParticleData* has the ability to spatially hash and sort particles to increase data locality when accesing the particles' properties.

A SELF-CONTAINED UAMMD CODE EXAMPLE

Before moving on, it is worth laying out here a complete example of an UAMMD simulation in hopes of helping the reader not to loose sight of the framework as a whole. Source code 4 contains the UAMMD code required to perform a constant-energy (NVE ensemble) MD simulation of a collection of **Lennard-Jones (LJ)** particles. Note that most of the concepts (both physical and programming ones) in this example have not been introduced in detail yet. In the course of this manuscript we will introduce the necessary tools required for the reader to understand each part of code 4. It is not the goal of this example to be understood in its entirety at this point of the manuscript, rather offer to the reader an overview, in general terms, of how a complete simulation code using UAMMD might look like. A quick look at the main function at the end of source code 4 reveals the general structure of the program, with most steps executing one of the functions defined before.

⁴ The *ParticleData* member function `getIdOrderedIndices()`; returns an array that stores the memory location (index) of a particle given its name (aka id).

```
# include "uammd.cuh"
# include "utils/InitialConditions.cuh"
# include "Interactor/Potential/Potential.cuh"
# include "Interactor/NeighbourList/CellList.cuh"
# include "Interactor/PairForces.cuh"
# include "Integrator/VerletNVE.cuh"

using namespace uammd;
using std::make_shared;
using std::shared_ptr;

auto initializeParticles(int number_of_particles, Box
→ box){
    auto particles =
        make_shared<ParticleData>(number_of_particles);
    auto position = particles->getPos(access::cpu,
        access::write);
    auto initial = initLattice(box.boxSize,
        number_of_particles, sc);
    std::copy(initial.begin(), initial.end(),
        position.begin());
    return particles;
}

auto createVerletNVE(shared_ptr<ParticleData>
→ particles){
    using Verlet = VerletNVE;
    Verlet::Parameters VerletParams;
    VerletParams.dt = 0.01;
    VerletParams.initVelocities = true;
    VerletParams.energy = 1.0;
    auto integrator = make_shared<Verlet>(particles,
        VerletParams);
    return integrator;
}

auto createLJInteraction(shared_ptr<ParticleData>
→ particles,
    Box box){
    auto LJPotential = make_shared<Potential::LJ>();
    Potential::LJ::InputPairParameters LJParams;
    LJParams.epsilon = 1.0;
    LJParams.sigma = 1.0;
    LJParams.cutOff = 2.5*LJParams.sigma;
    LJParams.shift = true;
```

```

LJPotential->setPotParameters(0, 0, LJParams);
using LJForces = PairForces<Potential::LJ>;
LJForces::Parameters interactionParams;
interactionParams.box = box;
auto interaction = make_shared<LJForces>(particles,
     $\hookrightarrow$  interactionParams, LJPotential);
return interaction;
}

void runSimulation(shared_ptr<Integrator>
 $\hookleftarrow$  integrator){
    int number0fSteps = 1000;
    for(int step = 0; step < number0fSteps; ++step) {
        integrator->forwardTime();
    }
}

void writePositions(shared_ptr<ParticleData>
 $\hookleftarrow$  particles){
    std::string outputFile = "Lennard-Jones.dat";
    std::ofstream out(outputFile);
    auto position = particles->getPos(access::cpu,
         $\hookrightarrow$  access::read);
    const int * index =
         $\hookleftarrow$  particles->getIdOrderedIndices(access::cpu);
    out<<std::endl;
    for(int id = 0; id < number0fParticles; ++id){
        auto pi =
             $\hookrightarrow$  box.apply_pbc(make_real3(position[index[id]]));
        out<<pi<<std::endl;
    }
}

int main(int argc, char *argv[]){
    int number0fParticles = 100000;
    // Simulation box (periodic by default)
    real L = 128;
    Box box(make_real3(L, L, L));
    // Initial configuration
    auto particles =
         $\hookrightarrow$  initializeParticles(number0fParticles, box);
    // Integration scheme (Verlet)
    auto integrator = createVerletNVE(particles);
    // Add interactions (Lennard-Jones)
    auto lj = createLJInteraction(particles, box);
    integrator->addInteractor(lj);
}

```

```
// Numerical integration of the equations of
// motion
runSimulation(integrator);
// Output final configuration
writePositions(particles);
return 0;
}
```

Source Code 4: A constant-energy UAMMD simulation of a collection of Lennard-Jones particles. For simplicity, all parameters are hard-coded (as opposed to, for instance, being read from a file).

8

PARTICLE INTERACTIONS

One of the reasons for our vague definition of particles is that often, from an algorithmic point of view, it does not really matter what a particle represents. Sometimes a particle will be an atom or molecule, other times it will make more sense that our simulation unit is a big colloid, a virus or a whole star.

The important thing about particles is that when several of them are present, they more than often have a certain effect on each other. Stars in a galaxy attract each other over an infinitely long range of space, while two argon atoms will repel each other at close range. Even if particles are somehow oblivious to each other, they might interact with some other thing, like fluid particles being repelled by a wall. Each kind of interaction requires a different approach, and figuring out how to efficiently compute it in a [GPU](#) is a field in itself. In this chapter we will see some of the strategies that can be employed and how they are implemented in [UAMMD](#). Figure 8.1 shows the next level of the *Interactor* branch in Fig. 2.1. Most [UAMMD](#) *Interactors* are generic (i.e. they can be externally specialized), indicated in Fig. 8.1 in blue.

We start with the case of particles interacting with each and every other in chapter 9. In chapter 10 we discuss the optimization opportunities that arise when this interaction has a short range (compared to the typical particle size). After that, in Chapter 12 we explore the case of particles being joined by spring-like ligatures (be it in pairs, triplets, quadruplets, etc).

Let us start by describing [UAMMD](#)'s *Interactor* interface in detail and then proceed to descend to the *Interactor* logical branch in Fig. 2.1, for which the next downward leaves are depicted in Fig. 8.1. In particular, we will focus on discussing the long-range branch in chapter 9 and the short-range branch in chapter 10 (depicted in figure 8.2).

8.1 THE INTERACTOR INTERFACE

Interactor encapsulates the concept of a group of particles interacting, either with each other or with some external influence. An

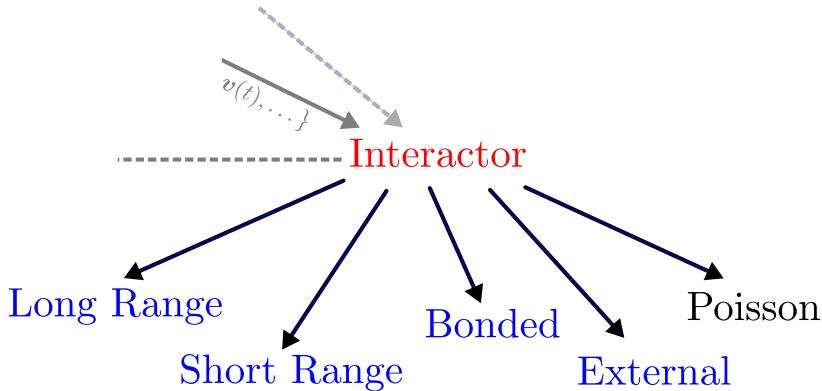


Figure 8.1: The next level in the *Interactor* branch of the tree presented in Fig. 2.1. Names in blue represent UAMMD modules that can be specialized with outside logic. For instance, the bonded interactions module is general to any arbitrarily complex potential, which can be provided to the module via the use of metaprogramming.

Interactor can be issued to compute, for each particle, the forces, energies and/or virial due to a certain interaction. To do so it can access the current state of the particles (like positions, velocities, etc) via *ParticleData*. A minimal example of an *Interactor* can be found in code 5.

```

#include <uammd.cuh>
#include <Interactor/Interactor.cuh>
using namespace uammd;

//A class that needs to behave as
//an UAMMD Interactor must inherit from it
class MyInteractor: public Interactor{
public:
    //The constructor must initialize the base
    // Interactor class, for which a ParticleData
    // instance is required.
    //Other than that, it can take any necessary
    // arguments (such as a group of parameters).
    MyInteractor(std::shared_ptr<ParticleData> pd):
        Interactor(pd, "MyInteractor"){
            //Any required initialization
    }
}
  
```

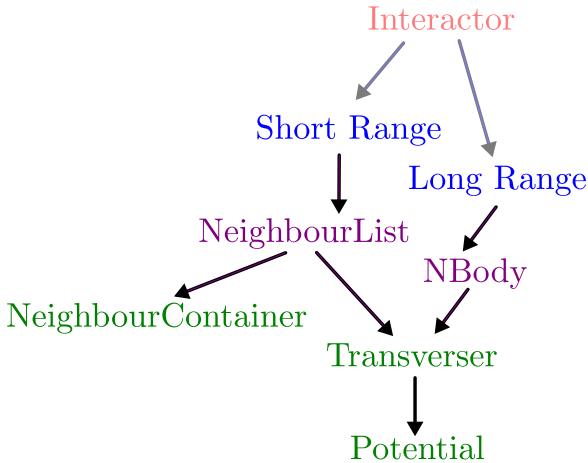


Figure 8.2: Overview of the concepts in the following chapters. Green text represents template arguments, while generic algorithms are purple. Long- and short-ranged interactions can be specialized from the outside via the *Potential* interface (see Appendix E.1). Arrows denote logical dependence.

```

//An Interactor can be issued, mainly
// by Integrators, to sum
// forces, energies and/or virial
// on the particles
void sum(Computables comp, cudaStream_t st)
→ override{
//"sys" and "pd" are provided by the Interactor
→ base class
    sys->log<System::MESSAGE>("Computing
→ interaction");
    if(comp.force){
        //Sum forces to each particle
        //For instance, adding a force to the x
        → coordinate
        // of the first particle
        auto forces = pd->getForces(access::cpu,
→ access::write);
        forces[0].x += 1;
    }
    if(comp.energy){
        //Sum energies to each particle
    }
    if(comp.virial){
        //Sum virial to each particle
    }
}
  
```

```
    }
}
};
```

Source Code 5: The basic outline of a new *Interactor*. A lot of advanced functionality has been omitted here for simplicity, refer to Appendix D for more information. See chapter 6 for more information about how to access particle properties.

The *Computables* type in the *sum* function simply contains a list of boolean values describing the needs of the caller (which will typically be an *Integrator*). As of today, an *Interactor* can be asked to compute only forces, energies and or virials¹ is defined acting on the particles. The *Computables* structure exists also to facilitate the future inclusion of additional quantities to the *Interactor* responsibilities.

Note that *Interactor* is what is called a *pure-virtual* class in C++ (and programming in general). This means that *Interactor* is not a class that can be used by itself (such as, for instance, *ParticleData*). It is a conceptual base class that must be inherited², as in code snippet 5. The same way a *dog* is a type of *animal*, *MyInteractor* in code 5 is a type of *Interactor*. We will see other examples shortly.

Any class inheriting from *Interactor* will have access to an instance of *System* with the name *sys*, that can be used to query properties of the GPU and log messages, and a *ParticleData* instance with the name *pd*.

It is worth mentioning that the *Interactor* interface offers, in addition to what is showcased in code 5, a series of optional functionalities, omitted here for simplicity, allowing more sophisticated communication between modules. The reader can learn about these extra capabilities through UAMMD's online documentation (see Appendix D).

8.1.1 The simulation domain

Particles are often located inside a certain domain. In each direction, we distinguish between the domain being open or periodic. In the first case, particles near opposite walls of the domain will not interact. This can be used to model either a truly open boundary

¹ Where the virial of a particle i is defined as $T_i = \frac{1}{2} \sum_j \mathbf{F}_{ij} \cdot \mathbf{r}_{ij}$.

² See Appendix A for more information about C++ classes and inheritance.

system (if the domain length is infinite) or in general a non periodic domain (e.g. due to the presence of a repulsive wall).

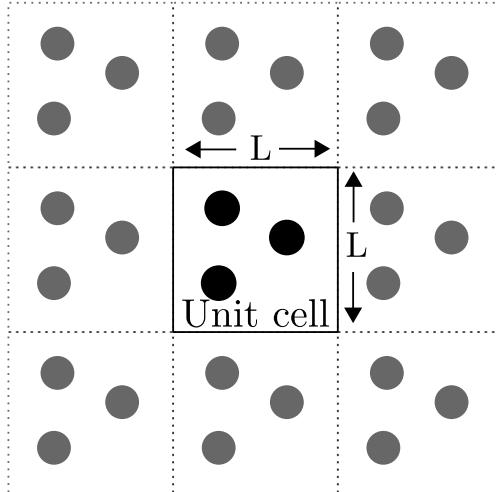


Figure 8.3: A representation of periodic boundary conditions. The unit cell, containing the system, will interact with the surrounding copies of itself if periodic boundary conditions are in place. Given two particles in the unit cell, if the [Minimum Image Convention \(MIC\)](#) is considered, they will only interact with the closest images of each other (included the one in the unit cell).

On the other hand, periodic domains are common in numerical simulation as a way of approximating a large system while simulating only a small region of it (called unit cell, see figure 8.3). In a domain with periodic boundary conditions, particles in the unit cell interact with the particles in the system images. Usually, particles are made to interact only with the closest image of the other particles (so a pair of particles near opposite walls of the system see each other “through” the wall). The so-called [MIC](#) is used to determine the distance of the closest image, the algorithm is laid out in algorithm 1.

Use in UAMMD

In [UAMMD](#), the *Box* class holds information about the domain. When required, individual modules will request a *Box* instance as a parameter (for instance, short ranged interactions require a box

Algorithm 1 Minimum Image Convention, takes a position or displacement vector and returns the coordinates of the image the simulation domain

```
1: function MIC( $r$ ,  $L$ )
2:   return  $r - \text{floor}(r/L + 0.5)*L$ 
3: end function
```

if the domain is periodic). Source code 6 contains a usage example of the *Box* class

```
#include <uammd.cuh>
using namespace uammd;
int main(){
    real lx, ly, lz;
    lx = ly = lz = 32.0;
    //A Box requires the size of the domain in each
    // direction, which can be infinite
    Box box({lx, ly, lz});
    //Periodicity can be set independently for each
    // direction. 1 meaning periodic and 0 aperiodic
    box.setPeriodicity(1,1,0);
    //The unit cell goes from -L/2 to L/2 in each
    // direction.
    //Let's store in a variable a position just outside
    // the unit cell.
    real3 position_outside_box = {0.5*lx+1, 0, 0};
    //Given that we have set the box as periodic in X,
    // the position will be folded to the other side
    // of the domain
    real3 position_in_box =
        box.apply_pbc(position_outside_box);
    //position_in_box holds {-lx*0.5+1,0,0}
    return 0;
}
```

Source Code 6: Using the Box class.



LONG RANGE INTERACTIONS

Say we are set to simulate a galaxy, so far away from others that it can be safely assumed their action is negligible. Say we want

to simulate the dynamics of each of the N stars in this galaxy. Gravity is the dominating interaction between each star, sadly it decays slowly enough to be considered an infinitely ranged one. The gravity potential can be written as follows

$$U(r) = G \frac{m_1 m_2}{r} \quad (9.1)$$

Where G is the gravitational constant and m_1, m_2 are the masses of each particle. In this case, if we want to compute the force acting on a star due to the presence of all the others we must check each and every one of them. This leads to lots of interactions to check, more precisely $N(N - 1)$ of them (see Fig. 9.1). Of course, there are more sophisticated ways of performing such a computation, such as fast multipole methods [33] or Ewald splitting (which will be introduced in section 24). But sometimes these techniques are not a possibility so it is valuable to see how to efficiently handle this computation. Furthermore, this problem is a really good fit for a **GPU** as we are going to see.

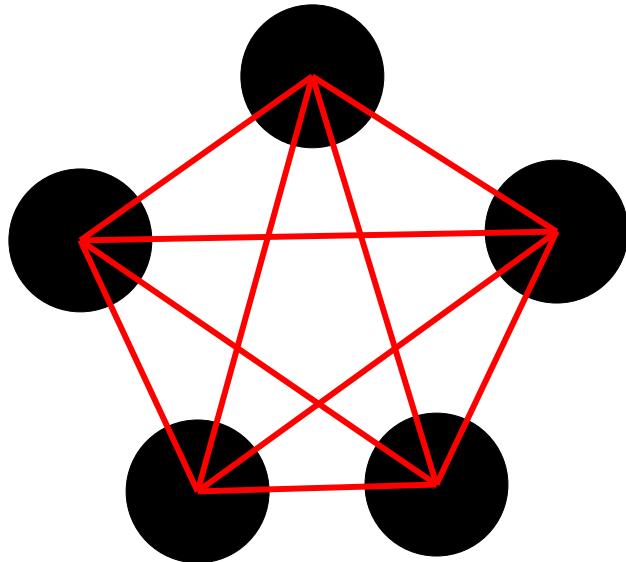


Figure 9.1: Five particles (black circles) must interact with every other (represented by red lines) in a NBody interaction. There are $N(N - 1) = 20$ lines, since self interactions are not depicted here.

9.1 THE NBODY ALGORITHM

Algorithm 2 summarizes the naive parallel approach. With the naive approach we assign a particle (or a group of them) to a thread. Then, each thread iterates over all the other particles.

Algorithm 2 Naive NBody algorithm. Each particle, i , visits all the others.

Input: A list of N particles

Require: A thread is launched per particle

- 1: $i \leftarrow$ thread ID ▷ Particle index
 - 2: **for** $j = 0$ until N **do**
 - ▷ Process $i-j$ pair
 - 3: **end for**
-

This is an embarrassingly parallel operation in which, in principle, a thread can work without collaborating with the others. However the naive algorithm is missing an opportunity in doing so. We can leverage that all threads have to access the same segments of memory. Instead of letting each thread diverge we can enforce that all threads access the same particles at the same time. This can effectively reduce the number of global memory accesses to a certain particle from N to, potentially, just one. This algorithm is based on the *nbody* algorithm originally devised by NVIDIA[34, 35]. It leverages the shared memory capabilities of the GPU¹. Each thread is assigned to a particle, with threads inside a block having consecutive particles. Groups of particles (called tiles) are then loaded collaboratively by the threads in the block into shared memory. The number of particles per tile is restricted by the amount of shared memory available².

Then the tile is processed by the thread block, in principle, with all threads accessing the same shared memory elements in lock-step. This is highly beneficial, since global memory accesses incur a high latency (in the order of hundreds of clock cycles).

¹ See Appendix A for information about the different memory spaces in the GPU.

² Thus far, are CUDA-capable cards above architecture 3.0 provide 48KB of shared memory per thread block. In single precision, the 3D coordinates of a particle take 12 bytes, limiting the tile size to 4096 particles. In practice, we set the tile size equal to the number of threads per block (typically 128) and allow the possibility of storing other particle properties into shared memory (mass, velocity,...).

In contrast, loading a value from shared memory yields a near register-memory performance (~ 4 cycles for shared memory and ~ 1 cycle for register memory). Since each thread's global memory accesses are reused by the rest, the latency of the former is hidden. In particular, if the tile size is equal to the number of threads in a block, N_{th} , (so each threads loads only one particle), the global memory latency is effectively reduced N_{th} -fold.

Once the tile has been processed, the next tile is loaded. This process is repeated until no particles remain (note that the last tile can potentially be smaller than the rest). The optimal size of a tile will be an optimization parameter also depending on the required shared memory per particle, being the number of threads in a block (or a multiple of it) a good default. This results in an overall speedup of 30 compared with the naive algorithm. The algorithm is summarized in algorithm 3 and a schematic representation of the process is presented in Fig. 9.2.

Algorithm 3 Shared memory NBody algorithm GPU kernel. Although the number of particles per tile is unconstrained, for simplicity this pseudocode assumes a tile has a size equal to the number of threads per block, with a number of tiles equal to the number of thread blocks.

Input: A list of N particles (positions, velocities,...)

Require:

A group of N_b thread blocks with N_{th} threads per block is launched.

$$N_b N_{th} \geq N.$$

A shared memory array with N_{th} elements.

Ensure:

Threads with index larger than N do not participate.

- 1: $i \leftarrow$ thread ID \triangleright The index of the particle assigned to this thread.
 - 2: $tid \leftarrow \text{mod}(i, N_{th})$ \triangleright Index of thread in the block.
 - 3: **for** tile = 0 until N_b **do**
 - 4: $\text{loadId} \leftarrow \text{tile} \cdot N_{th} + tid$
 - 5: $\text{shared}[tid] \leftarrow \text{particles}[\text{loadId}]$
 - 6: Synchronize threads in block. \triangleright Ensures the entire tile is loaded.
 - 7: **for** counter = 0 until N_{th} **do**
 - 8: $j \leftarrow \text{tile} \cdot N_{th} + \text{counter}$
 - 9: $pj \leftarrow \text{shared}[\text{counter}]$ \triangleright Read particle j from shared memory.
 - 10: Process $i-j$ pair
 - 11: **end for**
 - 12: Synchronize threads in block. \triangleright Ensures shared memory can be rewritten
 - 13: **end for**
-

Note that the computation as described here is not restricted to computing forces or energies between particles, it might be used for widely different computations that can be encoded as an nbody operation (like a dense matrix-vector product). [UAMMD](#) acknowledges this generality via the *Transverser* interface (see Appendix [E.1](#)), that can be used to specialize these algorithms.

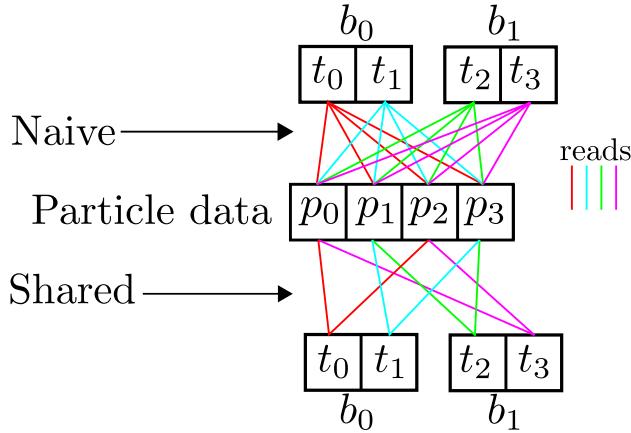


Figure 9.2: Representation of algorithms 2 (upper half of the figure) and 3 (bottom half). Two thread blocks ($b_{0,1}$), with two threads each ($t_{0,1,2,3}$), have to process four particles ($p_{0,1,2,3}$). Each thread must go through all particles and fetch some arbitrary information for each of them (i.e. positions, velocities and/or any other data required by the computation), represented by colored lines. The naive algorithm does not make use of the shared memory space available to threads in the same block and thus each thread needs to read all particles. On the other hand, the shared algorithm benefits from the shared memory space. In particular, each particle must be fetched only once per block (as opposed to once per thread in the naive algorithm). This is depicted by the reduced number of colored lines in the lower side of the figure.

Use in UAMMD

There are three ways to access this algorithm depending on the usage:

1. Processing a group of particles using a *Transverser* outside the **UAMMD** ecosystem (without *ParticleData*)³.
2. Applying a *Transverser* to the particles in *ParticleData*.
3. As an *Interactor* to compute forces, energies and/or virials. This is done by providing the *PairForces* module with a *Po-*

³ A *Transverser* applies a series of operations to each item in a list, taking advantage of the particularities of GPU computing described above. See Appendix E for instructions on how to craft a *Transverser*.

*tential*⁴ with an infinite cut-off distance. We will come back to *PairForces* in the chapter for short-ranged interactions, as it can also work with short ranged potentials.

An example of each one is available in code 7. The *Transverser* and *Potential* interfaces are described in detail in Appendix E.

If the *NeighbourCounter* example in Appendix E is given as a *Transverser*, the number of particles will be stored in the result of each particle.

While the first two functions in source code 7 actually perform a computation (as encoded in the provided *Transverser*), the third merely creates an *Interactor* (in particular, of type *PairForces*). In order for the *Interactor* to compute forces, energies and/or virials, its member function `sum` must be called, either directly or by an *Integrator*.

Source code 7 is the first one in this manuscript that provides a function creating an actual instance of an *Interactor* (in this case, *PairForces*), to place it into context, Appendix F provides an example showcasing how to put together an *Integrator* and an *Interactor* to construct a simulation.

```
#include <uammd.cuh>
using namespace uammd;

#include <Interactor/NBodyBase.cuh>
template<class Transverser>
void transverseWithoutUAMMD(real4* gpupositions,
                             int numberParticles,
                             Transverser tr){
    NBodyBase nb;
    nb.transverse(gpupositions, tr, numberParticles);
}

#include <Interactor/NBody.cuh>
//For each particle, applies a Transverser with all
//other particles.
template<class Transverser>
void transverseWithNBody(UAMMD sim, Transverser tr){
    NBody nb(sim.pd);
    nb.transverse(tr);
}
```

⁴ A *Potential* encases a *Transverser* specifically tailored to compute forces, energies or virials. See Appendix E and related examples for more information.

```

#include <Interactor/PairForces.cuh>
template<class Potential>
auto createPairForcesWithPotential(UAMMD sim,
→ Potential pot){
    using PF = PairForces<Potential>;
    return std::make_shared<PF>(pf(sim.pd, pot));
}

```

Source Code 7: Usage examples of the three different ways the NBody algorithm is exposed in [UAMMD](#). The *Transverser* and *Potential* interfaces are described in [Appendix E](#). Note that nothing in this example hints at the computation being long ranged, besides of the fact that the NBody algorithm is being used. While this ensures that every pair of particles in the system will be visited, the *Transverser* (or *Potential*, any of which will provide the actual computation logic/physics) can simply choose to, for instance, ignore any pair further than a certain distance.

Long range interactions in a periodic domain pose a special challenge. Direct summation would require taking into account the periodic copies of the system. On the other hand, this summation might even be conditionally convergent, as is the case with electrostatics. It is necessary to elaborate special algorithms for these cases. In [UAMMD](#), algorithms to solve the Poisson equation for electrostatics with triple and double periodicity are available. However, these algorithms rely on a mathematical machinery that will be discussed later on in this manuscript and thus we have delayed their description to chapter [24](#).

10

SH O R T R A N G E I N T E R A C T I O N S

Particles interacting closely allow for specific optimization taking advantage of the locality of the required computation. In reality it can happen that the effect of one particle on another is short ranged in nature, as could be the case for interactions stemming directly from quantum effects such as van der Walls forces. It can also happen that a combination of several long-ranged interactions result in an effective short-ranged one, such as a screened electrostatic interaction (like the DLVO potential). Other times when

the interaction cannot be made to decay rapidly using natural arguments we can still partially transform it into a short range one with techniques such as Ewald splitting, which will be discussed in future sections.

The reason for giving short range interactions so much credit is simple: imagine a system with N uniformly distributed particles inside a given domain. If we let each particle interact with every other we need to check N^2 pairs of particles as we already saw. On the other hand, restricting the interaction to a certain distance, such that each particle has a number of neighbours $k \ll N$ reduces the number of checks to kN .

The canonical example of a short range interaction is the Lennard-Jones potential.

10.1 THE LENNARD-JONES POTENTIAL

A standard potential employed to model the short range Van-der-Waals interactions is the LJ potential [36]. A rapidly-decaying repulsive radial potential with an attractive tail.

$$U_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6 \right] \quad (10.1)$$

Which results in this expression for the force at a certain distance \mathbf{r} between two points

$$\mathbf{F}_{LJ}(\mathbf{r}) = -\nabla U_{LJ} = -24\epsilon \left[\left(\frac{\sigma}{r}\right)^7 - 2\left(\frac{\sigma}{r}\right)^{13} \right] \frac{\mathbf{r}}{r} \quad (10.2)$$

Where $r = ||\mathbf{r}||$ is the modulus of the distance vector. The characteristic length, σ , and energy, ϵ , are typically used as natural units in simulations.

Given the rapidly-decaying nature of this otherwise infinite range potential it is standard to truncate it at a certain distance, usually $r_{cut} = 2.5\sigma$. At this distance the potential has a value of $U_{LJ}(r = r_{cut}) = -0.0615\epsilon$. Ignoring the long range effects of this potential changes the equation of state of a LJ fluid in a way that is well understood and tabulated [36]. Doing this allows to devise more efficient algorithms that only need to take into account short range interactions. Finally, in order to avoid jumps in the energy it is also standard to shift the potential in addition to this truncation.

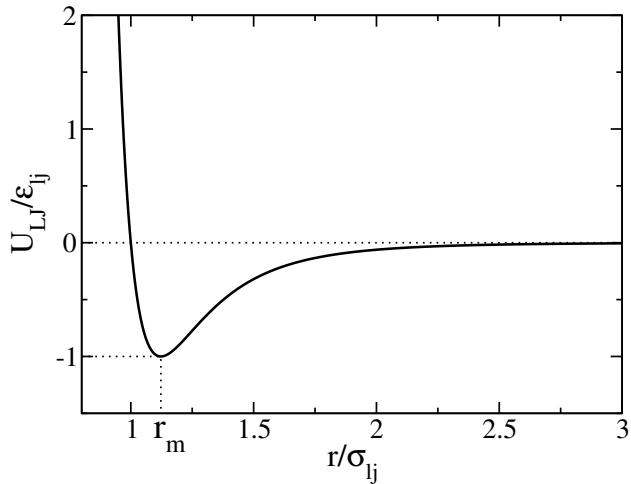


Figure 10.1: The Lennard-Jones potential described in Eq. 10.1

This is referred to as the truncated and shifted Lennard-Jones potential

$$U_{LJTS}(r) = \begin{cases} U_{LJ}(r) - U_{LJ}(r_{cut}) & r < r_{cut} \\ 0 & r \geq r_{cut} \end{cases} \quad (10.3)$$

The expression for the force remains as in Eq. (10.2), simply truncated at $r = r_{cut}$.

Sometimes we are only interested in the repulsive part of the LJ potential, as a way of modeling hard spheres (note that is only an approximation to the behavior of hard spheres). In order to do so we truncate Eq. (10.1) at its minimum, located at a distance $r_m = 2^{1/6}\sigma$, where $U_{LJ} = 0$. The resulting potential is called Weeks-Chandler-Andersen, or WCA, potential. We will refer to it as U_{WCA} .

USE IN UAMMD

A short range *Interactor* is available in **UAMMD** under the name *PairForces*. We have already seen this *Interactor* module when discussing long ranged interactions in example code 7. For a user of this module, the only difference between a long range or short range interaction is the cut-off distance. If it is large enough, the *PairForces* module will decide to use the NBody algorithm instead

of a neighbour list. An example of the creation of a *PairForces* module is available in code 8.

The *PairForces* module can be specialized for a certain *Potential* (see Appendix E.2) and *NeighbourList* (see chapter 11).

In the following chapters, we will discuss in detail the different neighbour list strategies.

```
#include <uammd.cuh>
using namespace uammd;
#include <Interactor/PairForces.cuh>
template<class Potential>
auto createPairForcesWithPotential(UAMMD sim,
→ Potential pot){
    //Any neighbour list can be used
    using NeighbourList = CellList;
    //using NeighbourList = VerletList;
    //using NeighbourList = LBVHLList;
    using PF = PairForces<Potential, NeighbourList>;
    PF::Parameters par;
    //A box can be specified here.
    //Note that periodicity can be set independently
    //in any direction.
    par.box = sim.par.box;
    //If the box is omitted, it is considered
    //aperiodic and infinite in the three directions
    //Optionally, an instance of a neighbour list
    //can also be provided as a parameter
    //par.nl =
    → std::make_shared<NeighbourList>(sim.pd);
    return std::make_shared<PF> pf(sim.pd, par, pot);
}
```

Source Code 8: Creating and returning a short range interaction module. The *Potential* interface is described in Appendix E.

NEIGHBOUR LISTS

Imagine a system composed of a uniform distribution of argon atoms, whose interaction can be quantitatively modeled via the LJ potential. When we discussed the gravitational interaction between

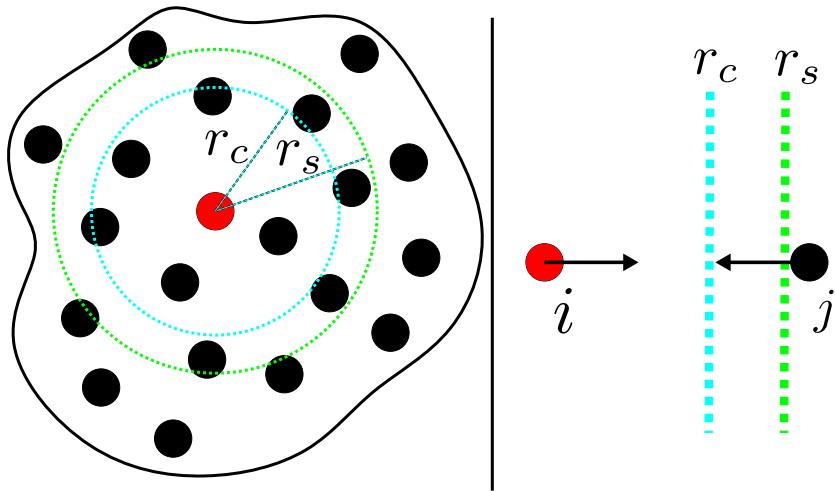


Figure 11.1: A depiction of a neighbour list in a section of a particle distribution (left). Particles inside the blue circle (of radius r_c) are neighbours of the red particle. The Verlet list strategy (section 11.3) defines a second safety radius, r_s , that can be leveraged to reuse the list even after particles have moved. In the worst-case scenario of the red particle and another particle just outside r_s approaching each other (right), the list will be invalidated only when each has moved $r_t = \frac{1}{2}(r_s - r_c)$ since the last rebuild.

stars, it made sense to use an efficient algorithm that checks all star pairs in the system. However, in this new system governed by Eq. (10.3), checking all pairs becomes pointless given that the vast majority of them will contribute practically nothing to the final result. We would like to skip all these pairs that lie beyond a certain cut off distance, r_{cut} , beforehand. A neighbour list contains, for each particle, a list of other particles that are closer than r_{cut} to it. There are a few approaches to constructing neighbour list, and **UAMMD** implements three of them as we will shortly see.

Once a list is constructed, the traversal algorithm depends on the type of neighbour list. However, the logic behind traversal will always be the same (see algorithm 4) and compatible with the *Transverser* interface (which is actually the main reason for its existence). In **UAMMD** neighbour lists can always be used by providing them with a *Transverser*.

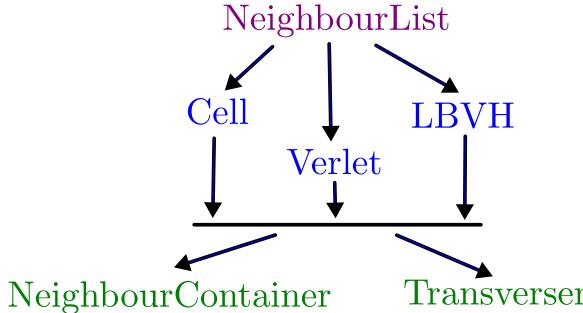


Figure 11.2: The three types of neighbour lists in **UAMMD**. All of them can be used via the same unifying interfaces; *Transverser* (described in Appendix E.1) and *NeighbourContainer* (described in sec 11.1). Additionally, the internal structures of each list can be requested for custom use.

Algorithm 4 Traversing a neighbour list. Each particle, i , visits all the others in its interaction list. In general, instead of launching a thread per particle, it is also possible to launch a thread block per particle and then perform a block reduction to obtain the final result. However, the *NeighbourContainer* interface is restricted to one thread per particle

Input: A list of N particles, A neighbour list
Require: A thread is launched per particle

- 1: $i \leftarrow$ thread ID ▷ Particle index
- 2: **for** j in $\text{nlist}[i]$ **do**
▷ Process $i-j$ pair
- 3: **end for**

This allows to unify any neighbour algorithm into a common interface. However, sometimes we would rather have access to an actual neighbour list. In these cases we generally have three options:

1. Use the *NeighbourContainer* interface, which provides a pseudo-container mimicking a neighbour list.
2. Use a *Transverser* to construct one with whatever format is needed. This is always valid and can be crafted with slight modifications to the neighbour counter example in 43 in Appendix E.

3. Use the internal structures the particular neighbour list construction algorithm provides. We will see how to do this in the following sections on a case by case basis.

11.1 THE NEIGHBOURCONTAINER INTERFACE

A *NeighbourContainer* is an interface that can provide, for each particle, each of its neighbours. It does not provide any guarantees besides this, which means that as long as it provides the neighbours using the common interface, it does not have to actually construct a list¹. Contrary to a *Transverser*, instead of providing the neighbour list with a certain logic, a *NeighbourContainer* allows to request the traversal information *from* the list. A summary of the usage of a *NeighbourContainer* can be found in code 9.

The *NeighbourContainer* interface works under the assumption that the underlying neighbour list can have an internal indexing of the particles and an internal copy of the positions in this order. We will see the rationale of this assumption shortly. It loosely behaves as a C++ standard container, providing *begin* and *end* methods returning forward input iterators to the first and last neighbours. This means that the neighbour container of a certain particle can only be advanced in sequential order starting from the first neighbour. By using this interface we can write code that will work for any neighbour list algorithm.

```
//The actual type name of the container
// will be different for each type of list
template<class NeighbourContainer>
__global__ void traverseNeighbours(NeighbourContainer
→ &nc, int N){
    //This kernel has to be launched with at least N
    → threads so tid goes from 0 to N-1.
    const int tid = blockIdx.x*blockDim.x +
    → threadIdx.x;
    //We have to filter out any surplus threads
    if(tid >= N) return;
    //Set the container to provide the list
    // of neighbours of particle tid.
```

¹ The CellList in sec. 11.2 actually exploits this by traversing the 27 cells around a given particle when needed, instead of reading from a precomputed list of neighbour particles

```

//Note that tid is the internal indexing of the
→ container
nc.set(tid);
//Get the group index of the particle:
const int i = nc.getGroupIndexes()[tid];
//Get the position of a particle given its internal
→ index
real3 pos_i =
→ make_real3(nc.getSortedPositions()[tid]);
//Loop through neighbours
for(auto neigh: nc){
    const int j = neigh.getGroupIndex();
    const real3 pos_j = make_real3(neigh.getPos());
    //Process pair i-j
    ...
}
//Do something with result
//output[i] = result;
}

```

Source Code 9: A CUDA kernel that uses a *NeighbourContainer* to go though all the neighbours of each particle (when launched with as many threads as particles). This code is an implementation of the pseudo code in Algorithm 4.

We are now ready to explore the different neighbour list algorithms, including construction and traversal. Although the latter can be solved via *Transversers* and/or *NeighbourContainers* we will see how each algorithm deals with it. This can be useful for more specialized computations that can leverage the internal workings of the different lists.

Let's start with the naive way to construct a neighbour list; Using algorithm 3 to fill a list of particles closer than r_c to each other. If for some reason we ought to traverse all neighbour pairs several times before the particles change positions this tactic is already a win over simply using algorithm 3. However, it negates one of the reasons for using a neighbour list, reducing the complexity of the overall operation from $O(N^2)$ to $O(kN)$ (with $k \ll N$). We can now traverse the list in $O(kN)$ operations, but construction still requires going through every pair anyway. Still, it is worth to go through the data format of this list since we can make use of it in the future. When it comes to the data format of a neighbour list

we can typically find three ways of storing the neighbours of each particle:

1. Store the list in a particle major format [22][37]. This format stores all the neighbours of each particle contiguously in a matrix of size $N \times n_{max}$, where n_{max} is a maximum number of neighbours allowed for a single particle. This can result in a lot of wasted space if there are large disparities in the number of neighbours per particle. On the other hand, having all the neighbours stored contiguously can be cache friendly in certain architectures or traversal strategies. The number of neighbours per particle can be encoded in a second list or via a special value in the main list (like -1). In a **GPU**, if we are to process the neighbours by assigning a thread block to each particle, this can be cache friendly, since contiguous threads in a warp will access contiguous elements in the list.
2. Store the list in a neighbour major format [21]. This format has the same storage requirements as the previous one. The only difference is that instead of storing the neighbours for each particle contiguously, we store contiguously one single neighbour for all particles. Starting at the N element we find the next neighbour for every particle and so on until the n_{max} neighbour is reached. The same storage waste concerns are present for this format. In a **GPU**, if the traversal is carried out assigning a thread per particle, this is the most cache friendly strategy, since contiguous threads will fetch contiguous elements. In **UAMMD** this is the chosen format when a neighbour list is constructed. As before, the number of neighbours per particle can be stored in an auxiliary array.
3. Compact any of the above lists. With the first **GPU** architectures the thread access patterns (the so-called coherence in CUDA) described in the previous points made a crucial difference. Nowadays the latest architectures are more forgiving about this, furthermore we might need to reduce the memory footprint of the list to the minimum. It is possible to compact any of the above lists so there are no “empty” spaces. This can be beneficial when the list has to be downloaded into the CPU, in which case the size of the memory transfer has to be optimized (and will probably dominate the cost of the overall operation). Nonetheless, I am not aware of any major implementation that chooses to do this.

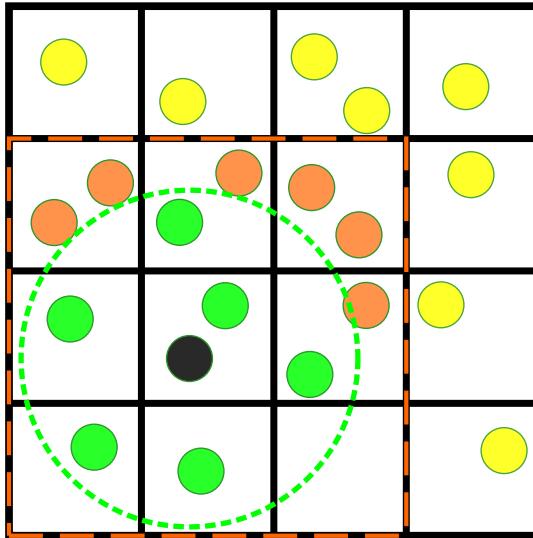


Figure 11.3: Sketch of the cell list algorithm. Space is binned (black grid) and the bin (cell) of each particle is computed. In order to look for the neighbours of the black particle (those inside the green dashed circle) all the particles inside the adjacent cells (bins inside the orange dashed square, 27 cells in three dimensions) are checked. The orange particles are therefore false positives. Finally, the yellow particles are never considered when looking for neighbours of the black one.

Lets now see the different **GPU** algorithms implemented in [UAMMD](#).

11.2 CELL LIST

This algorithm is based on the *particles* algorithm originally devised by NVIDIA and published in the acclaimed book GPU Gems 3[34]. It is reminiscent of the classic CPU linked cells algorithm[38] and in some ways an adaptation of it to the **GPU** architecture. This algorithm is the standard in **GPU** cell list generation and lots of works can be found describing it or variations of it [22][39][40][41]. In particular, our approach to building a cell list shares many similarities with the one described in [42]. The main idea behind the cell list is to perform a spatial binning and assign a hash to each particle according to the bin it is in. If we then sort these hashes we get a list in which all the particles in a given cell are

contiguous. By accessing, for a certain particle, the particles in the 27 surrounding cells we can find its neighbours without checking too many false positives, Fig. 11.3 provides an sketch of the cell list algorithm. Ideally, for a given particle, we would want to check the distance only with the N_{neigh} particles that lie at a distance closer than (or equal to) the cut off distance, r_c , thus requiring to check a volume of $V_{min} = \frac{4}{3}\pi r_c^3$. However, as the cell list partitions the space into cubes of side r_c , a volume of $V_{cl} = 27r_c^3$ around each particle has to be visited. Assuming the particles are uniformly distributed, the cell list will, on average, check a number of unnecessary particles that scales as $N_{cl}/N_{neigh} = V_{cl}/V_{min} \approx 6$. Although the cell list potentially requires visiting more than 6 times the number of particles that would strictly count as neighbours, its construction and traversal are so efficient that it is usually worth it. We are going to describe the algorithm for a rectangular box of side $\mathbf{L} = (L_x, L_y, L_z)$ with a cut-off distance r_c . This requires partitioning the domain into a number of cells $\mathbf{n} = \text{floor}(\mathbf{L}/r_c) = (n_x, n_y, n_z)$. Where *floor* represents the largest integer smaller than the argument. It is important to ensure the resulting cell size is at least the cut-off radius, so that $\mathbf{l} = \mathbf{L}/\mathbf{n} \geq r_c$.

It is possible to reduce the cell size in exchange for visiting more neighbouring cells (126 if $\mathbf{l} = r_c/2$), which will also reduce the number of neighbour candidates that lie beyond the cut-off distance. However, testing suggests that this is in general not worth the effort [22]. The algorithm for the cell list construction can be summarized in three separate steps

1. Hash (label) the particles according to the cell (bin) they lie in.
2. Sort the particles and hashes using the hashes as the ordering label (technically this is known as sorting by key). So that particles with positions lying in the same cell become contiguous in memory.
3. Identify where each cell starts and ends in the sorted particle positions array.

After these steps we end up with enough information to visit the 27 neighbour cells of a given particle.¹ We have to compute the assigned cell of a given position at several points during the algorithm. Doing this is straightforward. For a position inside the domain, $x \in [0, L]$, the bin assigned to it is $i = \text{floor}(x/n_x) \in$

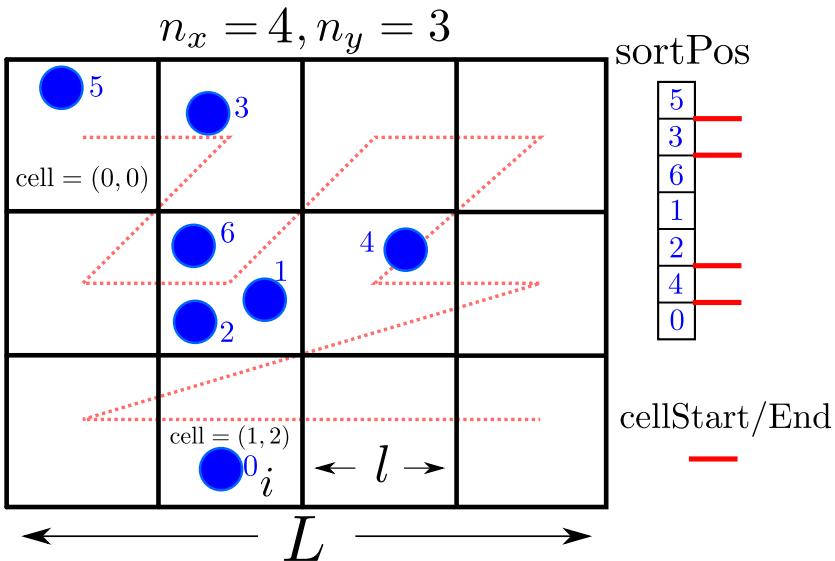


Figure 11.4: Representation of the spatial binning for a 2D distribution of particles. The particle marked as i lies in the cell with coordinates $(1, 2)$, these coordinates will then be used to assign a hash to particle i . The dotted red line represents the order given by the space-filling curve, starting at the $(0,0)$ cell. Blue particles lie in arbitrary indexes, aka memory locations, (blue numbers). After hashing the positions according to the red dashed line and sorting, particle positions of particles in the same cell are stored contiguously in *sortPos* (right), particles inside the same cell have an undetermined order. Solid red lines mark the end of a cell or, equivalently, the start of the next one, information that is stored in the *cellStart* and *cellEnd* arrays.

$[0, n_x - 1]$. It is important to notice that a particle located at exactly $x = L$ will be assigned the cell with index n_x , special consideration must be taken into account to avoid this situation. In particular, in a periodic domain, a particle at $x = L$ should be assigned to the cell $i = 0$ ². Figure 11.4 contains a representation of the binning. Let's describe now each step in detail

Hashing the particles

We want to assign to each position a hash that is unique to the cell in the grid in which it lies (see Fig. 11.4). These are stored in an auxiliary array holding the hashes for each index in the position array.

One possibility is to simply use the linear cell index as a hash, i.e $\text{hash}(i, j, k) := i + (j + kn_y)n_x$. However, we can leverage that the positions will be sorted according to this hash to improve the data locality for traversal later on. Using the cell index as hash will place particles with the same x coordinate close in the final array, but this is not optimal, since we will have to also explore contiguous cells in y and z which will end up far away. The usual solution is to use as hash some kind of space-filling curve that tends to assign similar values to cells that are spatially close. The key idea is to find a function that maps the three dimensional cell index to one dimension while preserving their locality. Any space filling curve, like Peano [43] or Hilbert [44] curves, can be used. We are going to use the Morton hash [45], a popular Z-order curve (see Fig. 11.4). Construction is fairly straightforward, although the implementation can be challenging to understand.

By assigning to each cell coordinate its binary representation and interleaving the three resulting patterns we get a hash that follows the path depicted in Fig. 11.4. In particular, we interleave three 10 bit coordinates in a 32 bit Morton hash (leaving 2 bits unused), which allows us to encode up to 1024 cells per direction, enough for most applications. We can achieve this with algorithm 5.

This algorithm is comprised of two functions; First a *mask* function that uses a series of bit masks to encode the bits of a number between 0 and 1023 in every third bit of a 32 bit unsigned

² Although this might sound evident, it is an easy-to-miss detail and the consequences resulting from its omission will haunt a naive developer for weeks (or so I am told).

integer. Then a *mortonhash* function that takes three numbers masked by the first function and interleaves them bit by bit by shifting them accordingly.

As an example, lets say we want to compute a 6 bit Morton hash from two 3 bit coordinates. In that case a particle in a 2D cell with coordinates (i, j) , encoded as two 3 bit numbers with bits $(i_0 i_1 i_2, j_0 j_1 j_2)$ will first get converted to the two 6 bit numbers $(i_0 0 i_1 0 i_2 0, j_0 0 j_1 0 j_2 0)$ (via two calls to *mask* in algorithm 5) and then converted into a Morton hash (via the *mortonhash* function) by interleaving them into a single 6 bit number, $i_0 j_0 i_1 j_1 i_2 j_2$.

Algorithm 5 Computing a hash from the coordinates of a cell by interleaving three Morton hashes. The symbols \ll (left shift), $|$ (bitwise OR) and $\&$ (bitwise AND) represent the bitwise C operators.

Input: The 3d coordinates of a cell in the grid (i, j, k)

- ▷ Interleave three 10 bit numbers into a 32 bit number

- 1: **function** MORTONHASH(i,j,k)
- 2: **return** id $\leftarrow \text{mask}(i) | (\text{mask}(j) \ll 1) | (\text{mask}(k) \ll 2);$
- 3: **end function**
- ▷ Encode a 10 bit number in every third bit of a 32 bit one
- 4: **function** MASK(i)
- Ensure:** $i < 1024$
- 5: x $\leftarrow i$ ▷ x must be an unsigned 32 bit integer
- 6: x $\leftarrow x \& 0x3FF$
- 7: x $\leftarrow (x | x \ll 16) \& 0x30000FF$
- 8: x $\leftarrow (x | x \ll 8) \& 0x300F00F$
- 9: x $\leftarrow (x | x \ll 4) \& 0x30C30C3$
- 10: x $\leftarrow (x | x \ll 2) \& 0x9249249$
- 11: **return** x
- 12: **end function**

Sorting the hashes

Once each particle position has been assigned a hash we sort them. For our use case, we need to sort the positions according to the values in the auxiliary array with the hashes, this operation is known as sort by key. Since all particles in a given cell are assigned the same, unique, hash this operation results in a list of positions such that all the particles in a given cell are guaranteed to lie

contiguous in memory. The most efficient way to perform this sorting operation in a GPU (as far as the author is aware) is the radix sort algorithm [46–48]. This algorithm is capable of sorting with an $O(n)$ complexity and its implementation happens to be quite a good fit for a GPU. In particular, UAMMD uses the radix sort implementation in the CUDA library *cub* [49]. Radix sort needs to traverse the input a number of times that increases with the number of bits in the keys. Our hashes are encoded in 30 bit numbers and, depending on the geometry of the grid, the rightmost bit set in the largest hash might be even lower. This allows us to further optimize the sorting process. With our implementation of the Morton hash we can find the largest possible hash simply by evaluating the hash of cell $(n_x - 1, n_y - 1, n_z - 1)$. We can leverage the C function *ffs* (find first set) or *clz* (count leading zero) to find the most significant bit set in an integer.

Constructing the cell list

By construction, positions lying in the same cell are contiguous in the array of positions sorted by hash. We can construct a cell list by finding the indexes such that the cell of that index is different from the previous one. If we store the indexes for the first and last particle in each cell, we can later use them to access the range of particles in it. This is an easily parallelizable operation, we can simply assign a thread to each particle and ask whether its cell is different from the cell of the previous particle in the list. If this is the case, we know that this particle is the first in its cell (and store it in the *cellStart* array) and also the last of the previous cell (and store it in the *cellEnd* array). The procedure is listed in algorithm 6.

The cost of the whole construction operation grows linearly with the number of particles and is quite inferior to the cost of traversing. On the other hand, this algorithm partitions the system into equal-sized bins independent of the particle distribution or sizes. The resulting neighbour list is therefore best suited for uniform distributions of particles with equal, or similar, sizes (interaction ranges).

When large range disparities are present (for instance, a big sphere interacting with a solvent made up of much smaller ones) the cell list requires choosing as cut-off distance the largest interaction range, which will yield many false positives for the smaller particles.

Algorithm 6 Constructing a cell list from a list of sorted-by-hash positions.

Input: Grid information, a list of N postions sorted by hash

Output: Arrays `cellStart` and `cellEnd` containing the indexes (in the positions array) of the first and last particles in each cell.

Requires: A thread per particle

```

1:  $i \leftarrow$  thread ID                                 $\triangleright$  Index of a particle
2: if  $id < N$  then
3:    $c_i \leftarrow \text{cellIndex}(\text{pos}[i])$ 
4:    $c_{i-1} \leftarrow \text{cellIndex}(\text{pos}[i-1])$ 
5: end if
6: if  $c_i \neq c_{i-1}$  or  $i = 0$  then     $\triangleright$  Particle  $i$  is the last of it's cell
7:    $\text{cellStart}[c_i] \leftarrow id$ 
8:   if  $i > 0$  then
9:      $\text{cellEnd}[c_{i-1}] \leftarrow id;$ 
10:  end if
11: end if
12: if  $i == N - 1$  then
13:    $\text{cellEnd}[c_i] \leftarrow N;$ 
14: end if
```

This can be alleviated by constructing a list for each interaction range. Then each particle traverses all lists, thus reducing the number of checks. Similar approaches (like stenciled lists) can be found in the literature[40].

Another option is to use a different strategy altogether, as we will see in sec 11.4.

Use in UAMMD

There are two main ways to construct a Cell List in [UAMMD](#) (and the other neighbour lists behave in a similar way):

- Through the [UAMMD](#) ecosystem. We can provide a `CellList` with a `ParticleData` and let it take care of when rebuilding is needed.
- Outside the ecosystem. Some use cases might benefit from a way to access the algorithm as less intrusively as possible. In particular a user might be interested in constructing the cell list for a list of positions not handled by `ParticleData`.

The example code 10 shows how to construct and traverse a cell list using both methods.

```
#include <uammd.cuh>
#include <Interactor/NeighbourList/CellList.cuh>
using namespace uammd;

//Construct a list using the UAMMD ecosystem
void constructListWithUAMMD(UAMMD sim){
    //Create the list object
    //It is wise to create once and store it
    CellList cl(sim.pd);
    //Update the list using the current positions in
    //sim.pd
    cl.update(sim.par.box, sim.par.rcut);
    //Now the list can be used via the
    //various common interfaces
    //Providing a Transverser:
    //cl.transverseList(some_transverser);
    //Requesting a NeighbourContainer
    auto nc = cl.getNeighbourContainer();
    //Or by getting the internal structure of the Cell
    //List
    auto cldata = cl.getCellList();
}

//Construct a CellList without UAMMD
template<class Iterator>
void constructListWithPositions(Iterator positions,
                               int numberParticles,
                               real3 boxSize,
                               int3 numberCells){
    //Create the list object
    //It is wise to create once and store it
    CellListBase cl;
    //CellListBase requires specific cell
    //dimensions for its construction
    Grid grid(Box(boxSize), numberCells);
    //Update the list using the positions
    cl.update(positions, numberParticles, grid);
    //Now the internal structure of the Cell List
    //can be requested
    auto cldata = cl.getCellList();
    //And a NeighbourContainer can be constructed from
    //it
}
```

```

    auto nc = CellList_ns::NeighbourContainer(cldata);
}

```

Source Code 10: The different ways of using a Cell List in UAMMD. The *Transverser* interface, already used in chapter 9, is described in detail in Appendix E.

In both cases the internal structure of the cell list can be requested to facilitate any usage outside the proposed interfaces for neighbour traversing. In the case of the cell list, this means having access to an ordered list containing the indexes of the particle's in each cell. Remembering that the cell list works by binning the domain, assigning a hash to each position according to the bin they are in and then sorting this hash list. As explained in sec. 11.2, the cell list algorithm mainly stores two arrays containing the indexes of the particles for each cell. However, there are some technical implementation details that we will discuss now. As part of the construction algorithm we need to sort the hashed positions into a, potentially, different order than the input positions. This allows us to easily identify the particles in each cell. Furthermore this has the fortunate side effect of providing us with a sorted positions array with the interesting property that particles that are close in space happen to be close (on average) in memory. In order to take advantage of this possibility, *CellList* constructs the list based on the indexes in this auxiliary array and provides it along the rest of the arrays. A really good visual representation of the data locality benefits from using these sorted positions for traversal can be found in Fig. 6 of [42].

Additionally, if the total number of cells is too large in particle configurations, then that results in a lot of empty cells. The process of filling up the *cellStart* array with a default value (marking the cell as "empty") can become a large part of the total runtime. In order to overcome this, instead of cleaning the array (which incurs visiting each element in the array and setting it to a certain value), *CellList* simply invalidates all the list in $O(1)$ time by interpreting any value lower than a certain threshold (called *VALID_CELL*) as the cell being empty. This value starts being 0 in the first update and is increased by the number of particles in each following update. When *VALID_CELL* is close to overload the maximum value representable by a C++ unsigned integer *cellStart* is then cleaned and *VALID_CELL* goes back to 0. So, when the cell list is constructed for the first time, *cellStart* contains values in the

range $[0, N - 1]$, but the next time (when *VALID_CELL* is equal to N), we will have values in the range $[N, 2N - 1]$. Let's describe the full contents of the *struct* returned by the *CellList* method *CellList::getCellListData()*.

- **Grid grid:** The grid data for which the list was constructed, holds the number of cells (n_x, n_y, n_z) and a series of utility functions to, for instance, find the cell of a particle. Also holds a *Box* object (for applying the minimum image convention).
- **uint VALID_CELL:** The threshold value in the cell list, any value in *cellStart* lower than this encodes the cell being empty.
- **real4* sortPos:** Positions sorted to be contiguous if they fall into the same cell.
- **uint* cellStart:** For a given cell with index (i, j, k) . The element $icell = i + (j + kn_y)n_x$ stores the index of the first particle in *sortPos* that lies in that cell.
- **int* cellEnd:** Contrary to *cellStart* this array stores the index of the last particle in the cell.
- **int* groupIndex** Given the index of a particle in *sortPos*, this array returns the index of that particle in *ParticleData*. This indirection is necessary when something other than the positions is needed (like the forces).

11.3 VERLET LIST

This list uses *CellList* to construct a neighbour list up to a distance $r_s > r_{cut}$, in this case the list only has to be reconstructed when any given particle has travelled more than a threshold distance,

$$r_t = \frac{r_s - r_c}{2}. \quad (11.1)$$

See Fig. 11.1.

Constructing the list has therefore a cost given by the cost of traversing the cell list (but larger due to the extra memory writes). Once it is constructed, the number of false positives will depend on the safety radius, r_s . In particular, assuming a uniform distribution of particles, the number of false positives will be given

by $N_s/N_{\text{neigh}} = V_s/V_{\text{neigh}} = r_s^3/r_c^3$. In order to get an optimal number of false positives, similar to the ones checked by the cell list (in which $N_{cl}/N_{\text{neigh}} \approx 6$) we would need to set $r_s > 1.8r_c$, which results in a threshold distance of $r_t = 0.4r_c$. Assuming the cut-off radius is the typical one for the LJ potential ($r_c \sim 2.5\sigma$, see sec. 10.1) this means that the list does not have to be reconstructed until one particle has diffused more than $r_t \sim \sigma$ from its starting position. Assuming each position update typically moves a particle less than $1\%\sigma$ this means that the list is rebuilt only once every several hundred steps.

On the other hand, the cost of constructing the neighbour list from the cell list also increases with r_s . There is therefore a trade-off between the compensated cost of list construction over many steps and the artificially inflated traversal cost. The optimal safety radius should be tuned for each type of simulation by inspecting how the timing depends on it. For instance, in a really dense system where particles are mostly still the optimal safety radius will be wildly different from a gas of rapidly diffusing particles. A good default is usually around $r_s \approx 1.15r_c$.

In the special case of $r_s = r_c$, which yields $r_t = 0$, the Verlet list can be used as a regular neighbour list. The list is constructed by storing a private list of neighbours for each particle in a column-major fashion. In order to achieve a cache-friendly memory pattern this “private” lists are stored in the same contiguous array. Since we do not know in advance how many neighbours each particle has we set up a maximum number of neighbours per particle, N_{max} , and allocate an array of size $N_{\text{max}}N$ elements. Later on we traverse the list by assigning a thread per particle, which prompts for a column-major layout of the list. That is, threads will tend to read contiguous memory locations if we place the first neighbours of all particles contiguously, then the second, and so forth and so on. A row-major layout (in which we place all neighbours of a certain particle contiguously) will be beneficial if we assign a *block* of threads per particle when traversing. Measuring is required to know which strategy is best in each case (thread-per-particle vs block-per-particle). UAMMD chooses a column-major format, as testing suggests this is the better choice in our habitual use-cases.³

³ Nonetheless this fact is abstracted away in the interface and changing between column- and row-major formats can be done easily and without affecting the users code.

Finally, the maximum number of neighbours per particles, which affects both performance and memory consumption, is autotuned at each update to be the nearest multiple of 32 (the CUDA warp size) of the particle with the greatest number of neighbours.

Use in UAMMD

As with the *CellList*, **UAMMD** exposes the Verlet list algorithm as part of the ecosystem and as an external accelerator. If the internal option is used the safety factor is automatically autotuned. Regardless, the safety radius can be modified via a member function call (see code 11).

In both cases, accessing the *VerletList* via the common **UAMMD** interfaces (*Transverser* and *NeighbourContainer*) makes it interchangeable with a *CellList*, as evidenced in the example code 11.

On the other hand, we can request *VerletList* to provide its internal structures for manual traversal. In this case, the format of the returned structure is as follows.

- **real4*** `sortPos`: Positions sorted to be contiguous if they fall into the same cell. Same as with *CellList*.
- **int*** `groupIndex` Given the index of a particle in `sortPos`, this array returns the index of that particle in *ParticleData* (or the original positions array if the list is used outside the ecosystem). This indirection is necessary when something other than the positions is needed (like the forces). Same as with *CellList*.
- **int*** `numberNeighbours`: The number of neighbours for each particle in `sortPos`.
- **StrideIterator** `particleStride`: For a given particle index, i , `particleStride[i]` holds the index of its first neighbour in `neighbourList`. **StrideIterator** is a random access iterator⁴.
- **int*** `neighbourList`: The actual neighbour list. The neighbour j for particle with index i (of a maximum of $j=\text{numberNeighbours}[i]$) is located at `neighbourList[particleStride[i] + j];`.

⁴ In particular, in the current implementation this iterator simply returns, for a given index i , `maxNeighboursPerParticle*i`.

Note that, as evidenced by its type name, the particle stride (or offset) is not necessarily a raw array but rather a generic C++ random access iterator (that behaves as a raw array for most purposes). For instance the offset might just be the same for all particles. In UAMMD's current implementation, the particle stride is simply the maximum number of neighbours per particle (see the initial discussion in chapter 11). However, this interface allows for the possibility of compacting the list in the future, with a similar behavior as the *cellStart* array in *CellList*.

```
#include <uammd.cuh>
#include <Interactor/NeighbourList/VerletList.cuh>
using namespace uammd;

//Construct a list using the UAMMD ecosystem
void constructListWithUAMMD(UAMMD sim){
    //Create the list object
    //It is wise to create once and store it
    VerletList vl(sim.pd);
    //Update the list using the current positions in
    //→ sim.pd
    vl.update(sim.par.box, sim.par.rcut);
    //Now the list can be used via the
    // various common interfaces
    //With a Transverser:
    //vl.transverseList(some_transverser);
    //Requesting a NeighbourContainer
    auto nc = vl.getNeighbourContainer();
    //Or by getting the internal structure of the
    //→ Verlet List
    auto vldata = vl.getVerletList();
    //The safety radius can be specified
    vl.setCutOffMultiplier(sim.par.rsafe/sim.par.rcut);
    //The number of update calls since the last time
    // it was necessary to rebuild the list can be
    // obtained
    int nbuild = vl.getNumberofStepsSinceLastUpdate();
}

//Construct a VerletList without UAMMD
template<class Iterator>
void constructListWithPositions(Iterator positions,
                               int numberPartivles,
                               real3 boxSize,
                               int3
                               → numberParticles){
```

```

//Create the list object
//It is wise to create once and store it
VerletListBase vl;
//The safety radius can be specified
vl.setCutOffMultiplier(sim.par.rsafe/sim.par.rcut);
//Update the list using the positions
vl.update(positions, numberParticles,
           sim.par.box, sim.par.rcut);
//Now the internal structure of the Verlet List
// can be requested
auto vldata = vl.getVerletList();
//And a NeighbourContainer can be constructed from
// it
auto nc =
→ VerletList_ns::NeighbourContainer(vldata);
//The number of update calls since the last time
// it was necessary to rebuild the list can be
// obtained
int nbuild = vl.getNumberOfStepsSinceLastUpdate();
}//Construct a VerletList without UAMMD

```

Source Code 11: The different ways of using a Verlet List in **UAMMD**. The *Transverser* interface, already used in chapter 9, is described in detail in Appendix E. Note that the *Transverer* and *NeighbourContainer* options are identical to the case of a *CellList*.

11.4 LBVH LIST

The Linear Bounding Volume Hierarchy (LBVH) neighbour list works by partitioning space into boxes according to a tree hierarchy in such a way that interacting pairs of particles can be located quickly. The innermost level of partitioning encases single particles, and boxes sharing faces are hierarchically bubbled together to form a tree structure. In contrast, the cell list partitions space in identical cubes independently of the particles positions or their distribution inside the domain. The “object awareness” of the LBVH results in a better handling of systems with large density fluctuations or objects of highly different size⁵. A full detailed description of the algorithm goes beyond the scope of this manuscript. We provide

⁵ In contrast, the cell list is not aware of the size of the particles, only accepting a cut off distance. If a system contains particles of different sizes (and therefore interaction cut offs), the cell list must be constructed using the largest one. In the presence of large size disparities (say a set of large particles interacting

here is a brief break down, which it is beautifully laid out in detail in references [40], [50] and [51], with only minor modifications to them present in UAMMD’s implementation.

1. We start by assigning a different type to each particle based on its size (understanding that if particles have not been assigned a size, they will all have the same type).
2. Then, we sort the particles by assigning a hash to each one in a way such that two given particles of the same type that are close in physical space tend to be close in memory. We achieve this by sorting particles first by type and then by Z-order hash (actually, UAMMD uses the Morton hash presented at 5, encoding the type in the last two bits, which are typically unused).
3. The sorted particle hashes are included in a binary tree structure following Karra’s algorithm [52]. By including the type in the particle hashing, we can generate a single tree, ensuring that a different subtree is constructed for each type. The root of the subtree for each type can be then identified by descending the tree. This appears to scale well with the number of types when compared to generating an entirely new tree for every type as in [40, 50].
4. After, we Assign an Axis Aligned Bounding Box (AABB) to each node of the tree that joins the AABBs of the nodes below it (with the particles, aka leaf nodes, being at the innermost level). This is done using Karra’s algorithm [52]. The AABBs are stored in a “quantized” manner that allows to store a node in a single int4, improving traversal time [50]. The bubbling of boxes stops at the root of every type subtree.
5. Finally, the neighbours of a given particle are found by traversing the AABB subtrees of every type[51].

Tree traversal is carried out in a top-down approach, where each particle starts by checking its distance to the root of a given subtree and subsequently descending as needed. If a particle’s AABB overlaps a node within a given cut off, the algorithm goes

with tiny ones), a lot of unnecessary particle pairs will be visited (in particular when checking the neighbours of a tiny particle).

to the next child node, otherwise it skips to the next node//tree. For a given particle, overlap with the 27 (in 3D) periodic images of the current subtree is computed before traversal of a tree and encoded in a single integer to reduce divergence (except the main box, which is traversed first by default) (see [50]). After a type subtree is entirely processed, the process is repeated with next one until none remain.

In my personal experience, the sheer raw power of the cell list in the GPU makes this algorithm not worth the effort in general. Note, however, that this algorithm is bound to outperform the cell list in certain situations, mainly when the size disparities between the different particles in the simulation is pronounced (as in the largest particle having at least twice the size of the smallest) or when the configuration presents a very low density (in terms of the cut-off distance of the interaction).

Usage in UAMMD

The interface for this neighbour list in UAMMD is more restricted than those of the previously introduced ones. The reason for this being its, yet to be found, applicability in our simulations. Nonetheless, it can be used in any place where *CellList* or *VerletList* can be used.

The internal data structure of the LBVH list can be queried, but we have not discussed in detail the algorithm in this manuscript. A reader who is particularly interested in making use of the LBVH list or a more in-depth understanding of its inner workings is referred to UAMMD's online documentation⁶(see Appendix D).

```
#include <uammd.cuh>
#include <Interactor/NeighbourList/LBVHList.cuh>
using namespace uammd;

//Construct a list using the UAMMD ecosystem
void constructListWithUAMMD(UAMMD sim){
    //Create the list object
    //It is wise to create once and store it
    LBVHList vl(sim.pd);
    //Update the list using the current positions in
    → sim.pd
```

⁶ or the code itself, located at the source file *Interactor/NeighbourList/LBVH.cuh*.

```

vl.update(sim.par.box, sim.par.rcut);
//Now the list can be used via the
// various common interfaces
//With a Transverser:
//vl.transverseList(some_transverser);
//Requesting a NeighbourContainer
auto nc = vl.getNeighbourContainer();
//Or by getting the internal structure of the LBVH
→ List
auto vldata = vl.getLBVHList();
}

```

Source Code 12: Example usage of the LBVH List in [UAMMD](#). The *Transverser* interface is described in detail in Appendix E. Note that the *Transverer* and *NeighbourContainer* options are identical to the case of a *CellList*.

11.5 PERFORMANCE COMPARISONS

It is worth comparing the performance of the three kinds of neighbour list strategies laid out in previous sections. In particular, we will inspect the time each list takes to construct and traverse in order to compute the LJ forces (see section 10.1) of a suspension of particles in a periodic domain. We will integrate the temporal dynamics of the system using a Langevin thermostat (which will be introduced in section 15), which will allow us to see the effect of the particles diffusion on the performance of the Verlet list. Since the other strategies (the cell list and the LBVH list) are reconstructed at each step their performance is largely independent on the temperature.

We start with a suspension of particles in an equilibrium configuration and average the time per step during 1000 steps. The time required to update the particle positions (as well as the rest of the computations besides list construction and traversal) is negligible.

We consider a LJ liquid in a cubic box with periodic boundary conditions and all particles having $\sigma = 2a$, where $a = 1$ (particle radius) represents the units of length, and $\epsilon = k_B T$ (unless stated otherwise). The interaction is cut off at the standard distance, $r_c = 2.5a$. In order to study linear scaling we use a uniform density of particles, either $\rho := N/L^3 = a^{-3}$ or $\rho = 0.1a^{-3}$. The biggest system considered has $N = 16777216$ particles with a cubic size of $L = 322.54a$ (for $\rho = a^{-3}$) and $L = 694.98a$ (for $\rho = 0.1a^{-3}$).

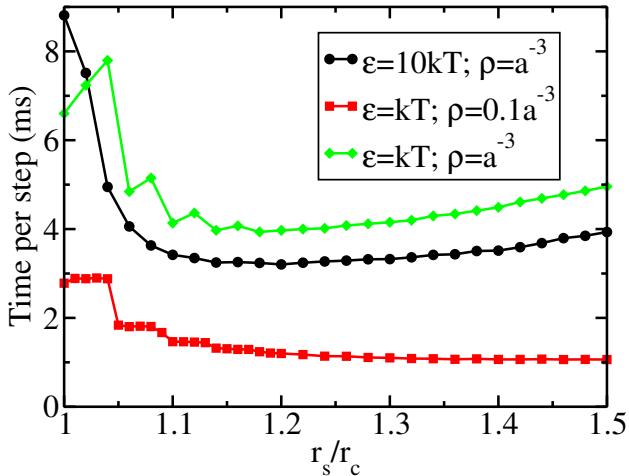


Figure 11.5: Performance of the Verlet list versus the safety cut-off radius, r_s , in a LJ fluid simulation. The system has one million particles. Tests ran on an RTX2080 GPU.

The performance of the Verlet list is affected by the safety cut-off radius, r_s (see Fig. 11.5). Increasing r_s will cause the list to update more sparingly at the cost of more false-positive neighbour checks and an enlarged memory requirement⁷. On the other hand, a small value of r_s will result in a more “optimal” neighbour list (in memory access pattern, false positive neighbours and size) at the expense of more regular updates. Based on Fig. 11.5 we choose $r_s = 1.2r_c$ for all tests. We compare the performance of the three neighbour lists strategies in Figure 11.6. At low system sizes the performance is mostly independent on the number of particles. This is a standard behavior in GPU computing coming from the fact that not all resources in the GPU are being utilized. When the computational load is large enough to fill the GPU (which happens at around 50K particles and beyond) performance exhibits the expected linear scaling, evidencing the fact that all of this strategies show a $O(N)$ complexity.

The tests in this section constitute a very specific benchmark that can be unfair with, for instance, the LBVH list. In particular, a uniform density benefits the cell list and thus, the Verlet list. On the other hand, the cell list excels when all particles have the same interacting radius ($r_c = 2.5a$ in this case) as in these

⁷ The Verlet list requires to reserve an array of size N_{\max} for each particle, where N_{\max} is, at least, the maximum number of neighbours that a particle has.

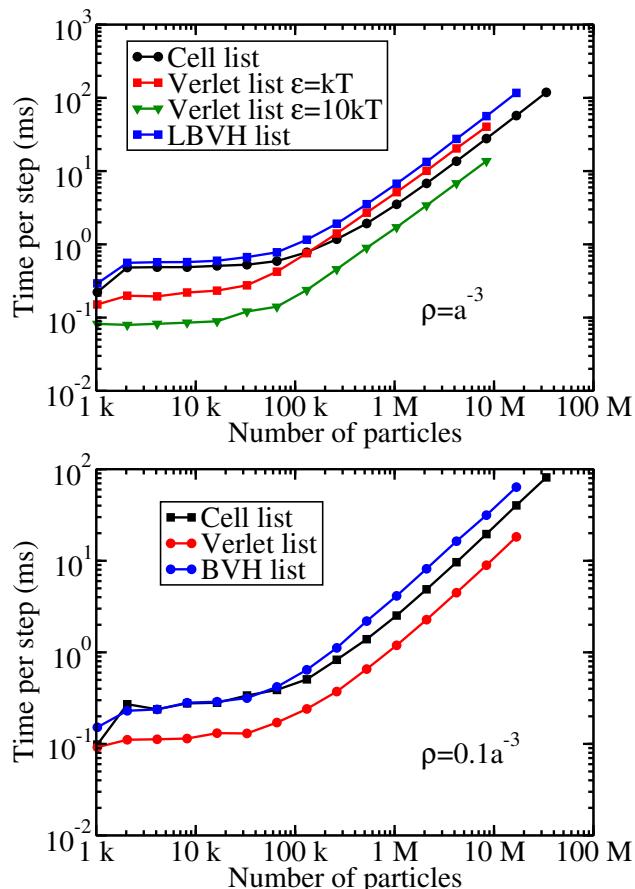


Figure 11.6: Performance comparison of the different neighbour list strategies in a LJ liquid at two different particle concentrations. The number of steps between reconstructions of the Verlet list is highly dependent on any parameter that modifies the diffusivity of the particles (like the temperature or the time step), which directly affects performance. In contrast, the other lists, being reconstructed every step regardless, are oblivious to the temperature or, in general, to changes in the diffusivity of particles. All tests were carried out in a RTX2080 GPU.

12

BONDED INTERACTIONS

As bonds can be expected to present great disparity in the number of bonds per particle, instead of using the same strategy as with the neighbour list, for the bonded interactions **UAMMD** generates a compacted list of bonds per particle similar to the cell list. For each particle, a list of all the bonds it is involved in is stored. Note that this means that the information for a given bond is stored several times (two times for pair bonds, three for angular bonds and so on). This improves the data locality when traversing the bond list (at the expense of extra storage). In contrast with the cell list algorithm, the bond list creation algorithm does not have to be particularly efficient since it will be generated only once at initialization¹.

Regarding traversal, as explained in sec. 11, for a compacted list such as this we need to store the bond list itself (containing a list of bonds for each particle) and then two auxiliary arrays; One with offset information (the index in the bond list where the particular list for a given particle is located) and another with the number of bonds for each particle. Note that it is not necessary to store information about particles that are not involved in any bond. UAMMD employs another auxiliary array storing the indexes of

¹ Creating the bond list only at initialization hinders the ability of the algorithm to efficiently allow for the dynamic inclusion/exclusion of bonds. However, we have not yet devised an efficient and sufficiently generic algorithm for constructing a dynamic bond list.

particles that have at least one bond. The special type of bond that involves a single particle is referred to as a fixed-point bond, usually attaching a particle to a certain point in space. In UAMMD this is encoded by storing the tether point position as part of the bond information.

USE IN UAMMD

The bond *Interactor* can be specialized for a certain number of particles per bond and a potential. Some potentials are already defined for bonds involving two, three (angular bonds) and four (torsional bonds) particles per bond.

The interface for the potential can be found in source code 13. The function `compute` of the bond potential (`HarmonicBond` in this example) requires some additional explanation. This function will be called for every bond read in the bond file and is expected to compute force/energy and/or virial for a certain particle involved in that bond². The arguments of this function are (see the example code 13):

- `bond_index`: The index of the particle to compute force/energy/virial on.
- `ids`: list of indexes of the particles involved in the current bond.
- `pos`: list of positions of the particles involved in the current bond.
- `comp`: computable targets (whether force, energy and/or virial are needed).
- `bi`: bond information for the current bond (as returned by the function `readBond`).

² Note that this implies that not only the information for each bond is stored several times, but the forces/energies/virials for the same bond are queried independently for each of the involved particles (as opposed to a model in which the bond potential would perform the computation for all particles in the bond at once). Although this might seem like a strange decision at first, it makes sense in the context of a GPU for the same reason that we do not make use of the fact that when pair forces are used $F_{ij} = F_{ji}$. Moreover, the computation of the potential can be wildly different for each particle involved in a bond with, for instance, 4 particles (like a torsional bond).

```

#include <uammd.cuh>
#include <Interactor/BondedForces.cuh>
using namespace uammd;

//Harmonic bond for pairs of particles
struct HarmonicBond{
    //Place in this struct whatever static information
    //is needed for the different bonds.
    //In this case spring constant and equilibrium
    //distance.
    //The function readBond below takes care of reading
    //each BondInfo from the file.
    //Naturally, the present class can also store any
    //additional information.
    struct BondInfo{
        real k, r0;
    };

    __device__ ComputeType compute(int bond_index,
                                  int ids[2], real3 pos[2],
                                  Interactor::Computables comp,
                                  BondInfo bi){
        real3 r12 = pos[1]-pos[0];
        real r2 = dot(r12, r12);
        const real invr = rsqrt(r2);
        const real f = -bi.k*(real(1.0)-bi.r0*invr);
        ComputeType ct;
        ct.force = comp.force?f*r12:real3();
        ct.energy = 0.0; //Whatever
        ct.virial = 0.0; //Whatever
        return (r2==real(0.0))?(ComputeType{}):ct;
    }

    //This function will be called for each bond in the
    //bond file and read the information of a bond
    BondInfo readBond(std::istream &in){
        //BondedForces will read i j, readBond has to
        //read the rest of the line
        BondInfo bi;
        in>>bi.k>>bi.r0;
        return bi;
    }
};

```

```

//Create a bond interactor for bonds with two
→ particles, uses the Harmonic potential defined
→ above.
auto createBondInteractor(UAMMD sim){
    //If the bond involves, for instance, 3 particles,
    → the module should be specialized accordingly
    → below.
    //Furthermore, the potential should be modified to
    → take this into account.
    using BF = BondedForces<HarmonicBond, 2>;
    typename BF::Parameters params;
    params.file = "bonds.dat";
    auto bf = std::make_shared<BF>(sim.pd, params);
    return bf;
}

```

Source Code 13: Constructing and returning a Bonded interaction module.

As usual, some additional functionality of this module has been omitted for simplicity. For instance, a shared pointer to an instance of the bond potential can be provided at the *BondedForces* construction to retain control of it (maybe to mutate the properties of the potential over time). See Appendix D for more information.

13

PARTICLE DYNAMICS

Thus far we have dealt with particle interactions. In the following chapters we will see a series of numerical strategies to solve the time evolution dynamics of the particles in the different levels of description introduced in chapter 3. We will do so by descending through the Integrator branch in Fig. 2.1. We can find the leaves below the Integrator branch in Fig. 13.1.

In a bottom-up manner, referring to the complexity of the numerical methods, we start with the microscopic level (via MD) in sec. 14, then Langevin in sec. 15 and finally the hydrodynamics and Smoluchowski levels in chapter 18.

13.1 THE INTEGRATOR INTERFACE

We introduced the *Integrator* concept back in chapter 3, where we adhered to it the responsibility of taking the simulation to the

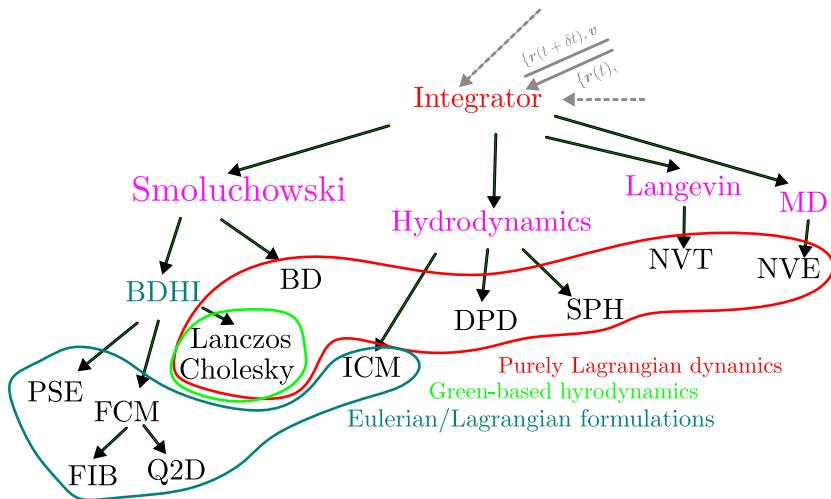


Figure 13.1: The next level in the *Integrator* branch of the tree presented in Fig. 2.1. Different methods are available for each of the levels of description (pink) introduced in chapter 3. Methods available as **UAMMD** modules are in black.

next time step. In this chapter, we will discuss in detail **UAMMD**'s *Integrator* software interface. Similar to *Interactor*, *Integrator* is a purely virtual C++ class that must be inherited (specialized). Instead of providing functions to compute forces (*Interactor*), *Integrator* exposes functions to take the simulation to the next time step. Additionally, *Integrators* hold a list of *Interactors* that they can use at their leisure to obtain up-to-date forces, energies and/or virials for each particle.

An arbitrary number of *Interactor* instances can be added to an *Integrator* (e.g. short-ranged forces, bonded forces...) via its member function `Integrator::addInteractor()`. Source code 14 is an example of how to create a novel *Integrator* (parallel to Example 5 for *Interactors*).

```
#include <uammd.cuh>
#include <Integrator/Integrator.cuh>
using namespace uammd;

//A class that needs to behave as
//an UAMMD Integrator must inherit from it
class MyIntegrator: public Integrator{
    real dt = 0.1; //A time step with a default value
public:
    MyIntegrator(std::shared_ptr<ParticleData> pd):
        Integrator(pd, "MyIntegrator"){
        //Any required initialization,
        //Typically, an Integrator will require
        //a series of input parameters,
        //such as the time step, etc.
    }

    //An Integrator can be issued to take the
    //simulation
    //to the next step in time
    virtual void forwardTime(cudaStream_t st) override{
        //Whatever is required to take particles to the
        //next step.
        { //Reset forces
            auto force = pd->getForce(access::cpu,
                access::write);
            std::fill(force.begin(), force.end(),
                real4(0.0));
        }
        //Typically we will issue the interactors to
        //compute forces, energies and/or virials.
        for(auto i: interactors) i->sum({.force=true}, st)
        //Use computed forces to update positions and
        //vels
        auto pos = pd->getPos(access::cpu,
            access::readwrite);
        auto vel = pd->getVel(access::cpu,
            access::readwrite);
        auto mass = pd->getMass(access::cpu,
            access::read);
        auto force = pd->getForce(access::cpu,
            access::read);
        for(int i = 0; i<pos.size(); i++){
            vel[i] += force[i]*dt/mass[i];
            pos[i] += vel[i]*dt;
        }
    }
}
```

14

M O L E C U L A R D Y N A M I C S

At the lowest level our simulation units are interacting atoms or molecules whose motion can be described using classical mechanics. We refer to the numerical techniques used in this regime as **MD**. In **MD** molecules are moving in a vacuum following the Newtonian equation of motion. If we need to include some kind of solvent, i.e. water, in an **MD** simulation we must do so by explicitly solving the motion of all the involved molecules of water (with some suitable force field). Even so, **MD** represents the basis for all particle-based methods, where the term *particle*, depending on the level of coarse graining, might refer to anything from an atom to a colloidal particle. Although it is not the most fundamental way of expressing the equations of motion, we will stick to a somewhat simplified but still quite general approach. For a system of N molecules interacting via a certain potential, Newton's second law

states that the acceleration experienced by each particle comes from the total force, \mathbf{F} , acting on it.

$$\mathbf{F} = m\ddot{\mathbf{q}} = m\mathbf{a}, \quad (14.1)$$

where m is the mass of the molecule, \mathbf{q} its position in cartesian coordinates and \mathbf{a} its acceleration. The force can usually be expressed as the gradient of an underlying potential energy landscape, U .

$$\mathbf{F} = -\nabla_{\mathbf{q}} U(\{\mathbf{q}_1, \dots, \mathbf{q}_N\}), \quad (14.2)$$

which in general is a function of the positions of the particles.

At their core, Eqs. (14.1) and (14.2) are an expression of the conservation of the total energy of the entire system. Consequently, these equations of motion can be used to perform simulations in the so-called microcanonical ensemble (NVE)¹, where the number of particles (N), the volume of the domain (V) and the total energy (E) are conserved. Note, however, that the energy is only conserved when forces stem from the gradient of a potential.

Our goal is to numerically integrate Eq. (14.1). Starting from the state of the system at a certain time t (where the system is conformed by the positions and velocities of the particles and the forces acting on them) we want to find the state of the system at time $t + \delta t$. We refer to δt as the time step.

The so-called finite-difference methods are the most commonly employed techniques to achieve this.

14.1 FINITE-DIFFERENCE METHODS

Finite-difference methods [53] solve Ordinary Differential Equations (ODEs) or Partial Differential Equations (PDEs) by approximating the derivative operators with finite differences. We can employ this idea to solve both spatial and temporal discretized differentiation operators. Thus far we are only interested in solving temporal derivatives, such as the one in Eq. (14.1). The derivative of a function, f , is defined as

$$\dot{f}(t) = \lim_{\delta t \rightarrow 0} \frac{f(t + \delta t) - f(t)}{\delta t} \quad (14.3)$$

¹ Although other systems can also be simulated with MD. For example, MD could be used to simulate the dynamics a group of gas molecules inside a piston (in which volume is not conserved).

If the function is smooth and analytic (well behaved), we can write its Taylor expansion to approximate $f(t + \delta t)$

$$f(t + \delta t) = f(t) + \dot{f}(t)\delta t + \frac{1}{2}\ddot{f}(t)\delta t^2 + O(\delta t^3) \quad (14.4)$$

For convenience, we will refer to the function evaluated at time t as $f^n := f(t)$ and as $f^{n\pm 1} := f(t \pm \delta t)$ when evaluated at $t \pm \delta t$. This notation can be extended to refer to points separated further in time, in general defining $f^{n\pm j} := f(t \pm j\delta t)$. Solving Eq. (14.4) for the first derivative

$$\dot{f}^n = \frac{f^{n+1} - f^n}{\delta t} - \frac{1}{2}\dot{f}^n\delta t + O(\delta t^2), \quad (14.5)$$

which approximates the definition of derivative for sufficiently small δt , when only the first term remains. Most of the integration techniques we will use make use of this trick. Lets see how we can leverage this technique to solve Eq. (14.1).

14.1.1 Euler Methods

If we truncate Eq. (14.4) at order δt we get the so called Euler method [54], which approximates the integral as

$$f^{n+1} = f^n + \dot{f}^n\delta t \quad (14.6)$$

Which yields a solver with $O(\delta t)$ accuracy [54]. It is worth exploring here a different derivation of the Euler method using the definition of integral instead of derivative. We can apply the fundamental theorem of calculus to get

$$f^{n+1} - f^n = \int_t^{t+\delta t} \dot{f}(t')dt' \quad (14.7)$$

On the other hand, we can approximate the integral using the left Riemann sum with only one interval (which is a good approximation as $\delta t \rightarrow 0$)

$$\int_t^{t+\delta t} \dot{f}(t')dt' \approx \delta t \dot{f}^n \quad (14.8)$$

By combining Eqs. (14.7) and (14.8) we arrive at (14.6) again. This derivation will come in handy later when describing integration schemes for equations with terms that cannot be approximated by Eq. (14.4). Furthermore, Eq. (14.8) opens the door to a family of

numerical integrators based on improving this approximation by using more intervals between t and $t + \delta t$. Some examples are the Runge-Kutta methods or the so-called linear multistep methods, such as the Adams Bashforth family of algorithms [54]. Often these methods improve numerical stability at the cost of evaluating the function derivative (in our case meaning the force and also, often, the velocity which is less expensive computationally) several times per time step. Given that in a MD simulation this is, often by orders of magnitude, the most expensive part of the computation we wont be making use of them for the time being. Finally, there are several modifications to the Euler algorithm that can improve its stability, such as the backward Euler [54] or the exponential Euler[54]. We will expand into Euler methods in section 17. For now, we can use it to integrate Eq. (14.1) to get the velocity and the positions

$$\begin{aligned}\mathbf{u}^{n+1} &\approx \mathbf{u}^n + \mathbf{a}^n \delta t \\ \mathbf{q}^{n+1} &\approx \mathbf{q}^n + \mathbf{u}^n \delta t\end{aligned}\tag{14.9}$$

Where $\mathbf{u} := \dot{\mathbf{q}}$ represents the velocity of the molecule. The Euler integrator as defined above is not symplectic[55] and as such presents several stability issues that make it impractical as compared with other strategies that will be discussed shortly. In particular, not being symplectic makes it unsuitable for our main goal: energy conservation[55]. On the other hand it is a first order method and lowering the time step too much to improve accuracy leads to numerical rounding errors (due to the limits of floating point accuracy). However, we can make a small modification to the Euler method to make it symplectic[55]

$$\begin{aligned}\mathbf{u}^{n+\frac{1}{2}} &\approx \mathbf{u}^n + \frac{1}{2} \mathbf{a}^n \delta t \\ \mathbf{q}^{n+\frac{1}{2}} &\approx \mathbf{q}^n + \frac{1}{2} \mathbf{u}^{n+\frac{1}{2}} \delta t\end{aligned}\tag{14.10}$$

This mid-point scheme leads to the semi implicit Euler scheme. While symplectic, this algorithm effectively requires two force evaluations per step (interpreting step as going from t to $t + \delta t$) and it is still a first order method. In order to efficiently solve Eq. (14.1) we need a symplectic and memory-saving algorithm that offers good numerical stability while requiring to evaluate the forces only once per step. The most widespread algorithm for MD is the so-called velocity Verlet method[38], a second-order algorithm that needs the forces only once per step.

14.2 THE VELOCITY VERLET ALGORITHM

Instead of truncating Eq. (14.4) at $O(\delta t)$ as we did with Euler let's now truncate it at $O(\delta t^3)$ and write the expressions for the positions at $t + \delta t$ and $t - \delta t$

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \mathbf{u}^n \delta t + \frac{1}{2} \mathbf{a}^n \delta t^2 + \mathbf{b}^n \delta t^3 + O(\delta t^4) \quad (14.11)$$

$$\mathbf{q}^{n-1} = \mathbf{q}^n - \mathbf{u}^n \delta t + \frac{1}{2} \mathbf{a}^n \delta t^2 - \mathbf{b}^n \delta t^3 + O(\delta t^4) \quad (14.12)$$

Where \mathbf{b} is just the term accompanying the $O(\delta t^3)$ term, typically referred to as *jerk*. By adding both equations together and solving for \mathbf{q}^{n+1} we get

$$\mathbf{q}^{n+1} = 2\mathbf{q}^n - \mathbf{q}^{n-1} + \frac{1}{2} \mathbf{a}^n \delta t^2 + O(\delta t^4) \quad (14.13)$$

When written like this, this equation can be used to integrate Eq. (14.1) and is known as the Störmer integration method. It can also be seen as an application of the central difference method to the position and as such, Eq. (14.13) is also known as the explicit central difference method. Although Eq. (14.13) presents really good numerical stability ($O(\delta t^4)$) at a small cost, it presents some issues that hinder its applicability. It requires storing the position at two points in time, which is inconvenient and poses the challenge of starting the simulation. Furthermore it eliminates the velocity from the integration, which can be computed from the positions using the central difference method

$$\mathbf{u}^n = \frac{\mathbf{q}^{n+1} - \mathbf{q}^{n-1}}{2\delta t} + O(\delta t^2) \quad (14.14)$$

Mathematically this is a second order approximation to the velocity. However, computing the velocity in this way can result in large roundoff errors stemming from the subtraction of two similar quantities. We can manipulate Eqs. (14.11) and (14.14) to arrive at the more commonly used version of the Verlet method, the so-called velocity Verlet algorithm. We can use the central difference method to evaluate the velocity at half step as $\mathbf{u}^{n+\frac{1}{2}} = \frac{\mathbf{q}^{n+1} - \mathbf{q}^n}{\delta t}$. By replacing \mathbf{q}^{n+1} with Eq. (14.11) and truncating at $O(\delta t^2)$ we get

$$\mathbf{u}^{n+\frac{1}{2}} = \mathbf{u}^n + \frac{1}{2} \mathbf{a}^n \delta t + O(\delta t^2) \quad (14.15)$$

Which allows to get a second order approximation of the position by solving Eq. (14.15) for \mathbf{u}^n and inserting it into Eq. (14.11)

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \mathbf{u}^{n+\frac{1}{2}}\delta t + O(\delta t^3) \quad (14.16)$$

We can then use Eq. (14.15) again to compute the velocity at $t + \delta t$

$$\mathbf{u}^{n+1} = \mathbf{u}^{n+\frac{1}{2}} + \frac{1}{2}\mathbf{a}^{n+1}\delta t \quad (14.17)$$

Note that these equations are just a rewrite of the Störmer-Verlet method above and thus present the same stability and accuracy while eliminating the aforementioned inconveniences presented by the former. Thus the overall algorithm is second-order accurate in the velocity and third-order accurate in the positions. Finally, the velocity Verlet algorithm can be summarized in the following steps

$$\begin{aligned} \mathbf{u}^{n+\frac{1}{2}} &= \mathbf{u}^n + \frac{1}{2}\mathbf{a}^n\delta t \\ \mathbf{q}^{n+1} &= \mathbf{q}^n + \mathbf{u}^{n+\frac{1}{2}}\delta t \\ \mathbf{u}^{n+1} &= \mathbf{u}^{n+\frac{1}{2}} + \frac{1}{2}\mathbf{a}^{n+1}\delta t \end{aligned} \quad (14.18)$$

Note that, with the exception of the first step, there is no need to compute the accelerations twice, since the forces at time t are known from the previous step. Forces at $t + \delta t$ should be evaluated after computing \mathbf{q}^{n+1} , typically overwriting the current forces. The only storage needed are the velocities, positions and accelerations per particle. Thus, the velocity Verlet algorithm presents all the necessary properties for being a popular integrator. It has good numerical stability, a small memory footprint, good energy conservation, guaranteed momentum conservation,...

USE IN UAMMD

The velocity Verlet algorithm in the NVE ensemble is available in [UAMMD](#) as an Integrator module called *VerletNVE*.

As an energy conserving algorithm, VerletNVE offers the possibility of setting a certain energy per particle at creation. The starting kinetic energy of the system will be set to match this energy. Note that this will result in particle velocities being overwritten. Alternatively you can instruct VerletNVE to leave particle velocities untouched, which will result in the initial total energy

being uncontrolled by the module. The only tool used by this module to set the initial total energy is the kinetic energy, which is always positive. As such, an error will be thrown if the target energy is *less* than the potential energy at creation.

```
#include <uammd.cuh>
#include <Integrator/VerletNVE.cuh>
using namespace uammd;
//A function that creates and returns a VerletNVE
→ integrator
auto createIntegratorVerletNVE(UAMMD sim){
    VerletNVE::Parameters par;
    par.dt = sim.par.dt; //The time step
    //Optionally a target energy can be passed
    par.energy = sim.par.targetEnergy;
    //If false, VerletNVE will not initialize
    → velocities
    //par.initVelocities = false;
    return std::make_shared<VerletNVE>(sim.pd, par);
}
```

Source Code 15: Example of the creation of a *VerletNVE Integrator*.

On its own, the *Integrator* object created in source code 15 does not have much use (besides simulating a gas of ideal particles, for instance). As discussed when introducing the *Integrator* interface, we can call the member `Integrator::addInteractor(std::shared_ptr<Interactor> inter)` present in every *Integrator* to add any interaction to it (such as any of the ones laid out in previous chapters and their accompanying source code examples). Regardless, we can call the member `Integrator::forwardTime()` (also present in every *Integrator*) to take the simulation state to the next time step. Appendix F contains an example showcasing how to add *Interactors* to an *Integrator* and forward a simulation in time. During the following sections, we will see different strategies to take into account the effects of the solvent in an implicit manner. In particular, the solvent effect will be coarse-grained in the form of fluctuations and hydrodynamic correlations between the resolved colloidal parti-

cles. The next natural step after MD equations are the so-called Langevin Dynamics (LD).

15

LANGEVIN DYNAMICS

In section 3.3.1 we introduced the Fokker-Planck formalism, which coarse grains the effects of a series of fast variables into drift and diffusion coefficients, enabling us to leave in our description only the slow relevant variables.

At the Langevin level of description, the dynamics of the solvent molecules (e.g. water) are orders of magnitude faster than the solute particles, leaving the positions and momenta of the solute particles as relevant variables. The dynamics of the solute particles are governed by a so-called Langevin equation, a manifestation of the Fokker-Plank Equation (FPE) that is often described as “MD with a thermostat”.

We can write the Langevin Stochastic Differential Equation (SDE) as

$$m d\mathbf{u} = \underbrace{\mathbf{F}}_{\partial_q U(q)} dt - \overbrace{\xi \mathbf{u}}^{\text{Drag}} dt + \underbrace{\beta}_{\text{Fluctuations}}, \quad (15.1)$$

which can be interpreted as a form of Eq. (14.1) coupled with a thermostat so that noise and drag forces are balanced according to the fluctuation-dissipation relation (see section 3.3.2), guaranteeing the correct thermalization of the system (NVT ensemble).

On the other hand, Eq. (15.1) is an expression of the equivalent SDE (Eq. (3.18)) for the FPE with the particle momenta and positions as the relevant variables. The first term in Eq. (15.1), \mathbf{F} , is the sum of the conservative forces acting on particle i (the ones coming from the underlying potential energy landscape), these interactions can be steric, electrostatic, bonded, etc. The second term represents the drag exerted by the solvent particles in the form of a dissipative force. Finally, the third term represents the fluctuations produced by the fast and constant collisions of the solvent particles. Here β is a random increment which must be in fluctuation-dissipation balance with the friction term (see section 3.3.2), we will shortly study its value.

The friction coefficient, ξ , is related with the damping rate as $\gamma = \xi/m$, which represents the decorrelation time of the velocity, $\tau = m/\xi$. Additionally, ξ can be formally derived from a Green-Kubo relation involving the integral of the solvent-solute force time-correlation[56, 57]. Its value is often approximated by the Stokes (macroscopic) value $\xi = 6\pi\eta a$, of a particle of radius a in a solvent with viscosity η (see Eq. (3.23) and discussion in section 3.3.3).

When we introduced the Fokker-Planck formalism in chapter 3.3.1 we saw that when a system dissipates energy, for instance, due to a force like the drag force in Eq. (15.1), the fluctuation-dissipation theorem states that there must be an opposite process that reintroduces this energy via thermal fluctuations.

We can use Eq. (3.15) to elucidate the relation between the drag, ξ and the noise in Eq. (15.1) by defining the single particle drift coefficient as $\widetilde{\mathbf{D}}^{(1)} := -\gamma\mathbf{u}$. The dynamic “matrix” (in this case being just a scalar) is then $H = \gamma$. We can now write Eq. (3.15) in the small time interval limit, which yields

$$D^{(2)} = \gamma\sigma_{\text{eq}}. \quad (15.2)$$

The equipartition theorem can be employed here to obtain the equilibrium covariance $\sigma_{\text{eq}} := \langle \mathbf{u}^2 \rangle$. In the absence of dissipative forces, the Hamiltonian is simply $H = \frac{p^2}{2m}$, where $p = mq$ is the momentum. Let’s apply these arguments to a one-dimensional system for simplicity, understanding they also hold for the three-dimensional case. The equipartition theorem states that for every degree of freedom in the system the following equation must hold,

$$\left\langle p \frac{\partial H}{\partial p} \right\rangle = \left\langle p \frac{p}{m} \right\rangle = k_B T. \quad (15.3)$$

Where k_B is the Boltzmann constant and T the temperature. Brackets represent ensemble average. The previous equation leads to the relation

$$\sigma_{\text{eq}} = \left\langle p^2 \right\rangle = mk_B T. \quad (15.4)$$

Which represents the variance of the particle’s momemtum. We can now use Eqs. (15.2) and (15.4) to identify

$$D^{(2)} := m\gamma k_B T. \quad (15.5)$$

On the other hand, in order to satisfy Eqs. (3.16) and (3.17) the fluctuating term must have zero mean,

$$\langle \beta \rangle = 0 \quad (15.6)$$

and be uncorrelated in time (since we are assuming the time interval is small enough to consider the transport coefficients as constants) with standard deviation given by

$$\langle \beta(t)\beta(t') \rangle = 2D^{(2)}dt\delta(t-t') = 2\xi k_B T dt\delta(t-t'). \quad (15.7)$$

Finally, we can write the Langevin equation as

$$m d\mathbf{u} = \mathbf{F} dt - \xi \mathbf{u} dt + \sqrt{2\xi k_B T} \widetilde{\mathbf{W}}, \quad (15.8)$$

Where we have defined $\beta := \sqrt{2\xi k_B T} \widetilde{\mathbf{W}}$. Technically, Eqs. (15.6) and (15.7) would be compatible with any random distribution for the noise, $\widetilde{\mathbf{W}}$, that has zero mean and variance dt . However, as the noise term comes from the action of countless independent random variables (collisions with the solvent particles) the central limit theorem[58] applies. Thus, it is convenient to choose $\widetilde{\mathbf{W}}$ as a Gaussian white noise (i.e. a Wiener process)[59].

The presence of the noise term with $\widetilde{\mathbf{W}}$ makes Eq. (15.8) an SDE. Since Eq. (15.8) is not analytic, the assumptions in Eq. (14.4) do not hold. In other words, we do not know how to differentiate a random variable. Luckily, we can make assumptions about the integral of a random variable. This allows us to devise a solver for Eq. (15.8) following a strategy similar to what we did in Eqs. (14.7) and (14.8) in sec. 14.1.1.

The challenge in the numerical integration of Eq. (15.8) comes from the discretization of the noise term. A bad discretization of this term can lead to convergence issues, such as in the case of the commonly used BBK (for Brooks-Brünger-Karplus) scheme[60], which is known to provide the correct temperature with an error of the order of $O(\delta t)$ [61]. We can overcome this issue by solving Eq. (15.8) in integral form. Lets start by studying the one dimensional case of a single particle, since the three directions are uncoupled and particles are independent (beyond the conservative forces) making the jump to 3D and many particles is then trivial. Now we integrate Eq. (15.8) between the interval t and $t + \delta t$

$$m \int_t^{t+\delta t} \dot{u} dt' = \int_t^{t+\delta t} F_c dt' - \int_t^{t+\delta t} \xi \dot{q} dt' + \sqrt{2\xi k_B T} d\widetilde{W} \quad (15.9)$$

Where $d\widetilde{W}$ is known as a Wiener increment, defined so that

$$d\widetilde{W} := \int_t^{t+\delta t} \widetilde{W}(t') dt. \quad (15.10)$$

The key now is to realize that we can compute the last term numerically by using the Riemann sum definition of the integral in the Itô sense. In the Itô interpretation, this integral is defined as the limit in probability of a Riemann sum, where the Brownian integral of a certain function f between a certain time interval $[0, t]$ is

$$\int_0^t f(t') d\widetilde{W}(t') = \lim_{n \rightarrow \infty} \sum_{i=0}^n f(t^{i-1}) (\widetilde{W}(t^i) - \widetilde{W}(t^{i-1})) \quad (15.11)$$

Where $t_i \in [0, t]$. It can be shown that this limit converges in probability[31].

We can use the Riemann sum to transform Eq. (15.10) into a sum with an arbitrarily large number of terms. Since in this case $f(t) = \sqrt{2\xi k_B T}$ we are left with a sum of Gaussian distributed random numbers, which also happens to be a Gaussian random number. We can compute $d\widetilde{W}$ as Gaussian random numbers with zero mean and standard deviation $\langle d\widetilde{W}^n d\widetilde{W}^m \rangle = 2\xi k_B T \delta t \delta_{n,m}$ (uncorrelated in time). This realization now opens the door to a family of Verlet-like integration schemes, in particular we are going to describe the algorithm developed by Grønbech-Jensen and Farago [62].

15.1 GRØNBECH-JENSEN

Without approximations, we can write Eq. (15.9) as

$$m(v^{n+1} - v^n) = \int_t^{t+\delta t} F_c dt' - \xi(r^{n+1} - r^n) + \beta^{n+1} \quad (15.12)$$

On the other hand, making use of the definition of velocity

$$\int_t^{t+\delta t} \dot{q} dt' = r^{n+1} - r^n = \int_t^{t+\delta t} v dt' \quad (15.13)$$

Which can be approximated to second order using the Riemann sum with two intervals (similarly as we did for Euler methods in sec. 14.1.1) as

$$r^{n+1} - r^n \approx \frac{\delta t}{2} (v^{n+1} + v^n) \quad (15.14)$$

Replacing v^{n+1} from Eq. (15.12) into (15.14) gets us

$$r^{n+1} - r^n = b\delta t v^n + \frac{b\delta t}{2m} \left(\int_t^{t+\delta t} F_c dt' + \beta^{n+1} \right) \quad (15.15)$$

Where

$$b := \frac{1}{1 + \frac{\xi\delta t}{2}} \quad (15.16)$$

The deterministic force term can be approximated to second order using the same strategy as in Eq. (15.14). However, for Eq. (15.15) only one term of the Riemann sum is needed to make it second order accurate (as it gets multiplied by δt^2). We can then write the final equations for the velocity and position in three dimensions as

$$\mathbf{q}^{n+1} = \mathbf{q}^n + b\delta t \mathbf{u}^n + \frac{b\delta t^2}{2m} \mathbf{F}_c^n + \frac{b\delta t}{2m} \beta^{n+1} \quad (15.17)$$

$$\mathbf{u}^{n+1} = a\mathbf{u}^n + \frac{\delta t}{2m} (a\mathbf{F}_c^n + \mathbf{F}_c^{n+1}) + \frac{b}{m} \beta^{n+1} \quad (15.18)$$

Where

$$a := b \left(1 - \frac{\xi\delta t}{2} \right) \quad (15.19)$$

Note that, since we need the force at $t + \delta t$ to compute the velocities at $t + \delta t$ the forces have to be a function of the positions only (and not velocities).

USE IN UAMMD

The Grønbech-Jensen algorithm described above is available in **UAMMD** as an Integrator module called *GronbechJensen*, under the namespace *VerletNVT*.

This module will initialize the particle's velocities near the equilibrium Boltzmann distribution. Alternatively you can instruct it to leave particle velocities untouched. The system's temperature will then tend to the equilibrium value according to the friction coefficient and each particle's mass.

```
#include <uammd.cuh>
#include <Integrator/VerletNVT.cuh>
using namespace uammd;
//A function that creates and returns a VerletNVT
→ integrator
auto createIntegratorVerletNVT(UAMMD sim){
```

```

using Verlet = VerletNVT::GronbechJensen;
Verlet::Parameters par;
par.dt = sim.par.dt; //The time step
par.friction = sim.par.friction;
par.temperature = sim.par.temperature;
//Optionally, you can force all particles to have
→ the same mass, overriding the one in sim.pd
→ even if it was set.
//par.mass = sim.par.mass;
//If set to true, particle velocities will be left
→ untouched at initialization.
//par.initVelocities = false;
//If 2D is set to true here the integrator will
→ leave the third coordinate of the particles
→ untouched.
//par.is2D = true;
return std::make_shared<VerletNVT>(sim.pd, par);
}

```

Source Code 16: Example of the creation of a *VerletNVT Integrator* module.

The Langevin model makes several assumptions to arrive at Eq. (15.8). At long times, hydrodynamic effects due to momentum propagation of the solvent (leading to long-time tails in the velocity correlation of the particle) are neglected. The dynamics of short times $\tau < \gamma/m$ are also wrong, as memory effects in the noise forces start to matter at fast small scales. While solving these small scale effects requires to explicitly describe the solvent-solute forces using **MD**, we will see in subsequent chapters how to improve the long-time limit introducing the solvent hydrodynamics.

16

DISSIPATIVE PARTICLE DYNAMICS

One of the most popular techniques used to reintroduce some of the degrees of freedom lost with **Langevin Dynamics (LD)** is **Dissipative Particle Dynamics (DPD)**. This coarse graining technique can be used to go further in the spatio-temporal scale by choosing groups of fluid particles as the simulation unit, sitting in-between microscopic (as in **MD**) and macroscopic (hydrodynamic) descriptions. In practice **DPD** is a Langevin approach where friction acts by pairs of particles and conserves momentum. **DPD**

equations can be defined from the microscopic Hamiltonian using coarse-graining theory[26]. In the standard DPD, particles interact via a soft potential, modelling the interaction between two large groups of fluid particles. In many instances[63] DPD is used as a momentum-conserving thermostat, which thus permits to include hydrodynamics (contrary to a single Langevin approach). Local momentum conservation results in the emergence of macroscopic hydrodynamic effects. These momentum conserving forces can then be tuned to reproduce not only thermodynamics, but also dynamical and rheological properties of diverse complex fluids. The equations of motion in DPD have the same functional form as LD and can be in fact considered as a momentum-conserving generalization of LD. In fact, it can also be derived from the Fokker-Planck formalism[59]. The equations of motion for DPD read,

$$m\mathbf{a} = \mathbf{F}^c + \mathbf{F}^d + \mathbf{F}^r. \quad (16.1)$$

Where the three forces are traditionally expressed as[64, 65],

$$\begin{aligned} \mathbf{F}_{ij}^c &= \omega(r_{ij})\hat{\mathbf{q}}_{ij} \\ \mathbf{F}_{ij}^d &= -\xi\omega^2(r_{ij})(\mathbf{u}_{ij} \cdot \mathbf{q}_{ij})\hat{\mathbf{q}}_{ij} \\ \mathbf{F}_{ij}^r &= \sqrt{2\xi k_B T}\omega(r_{ij})\widetilde{W}_{ij}\hat{\mathbf{q}}_{ij} \end{aligned} \quad (16.2)$$

Where $\mathbf{u}_{ij} = \mathbf{u}_j - \mathbf{u}_i$ is the relative velocity between particles i and j . Here ξ represents a friction coefficient (as in sec. 15) and is related to the random force strength via fluctuation-dissipation balance in a familiar way[65]. In general ξ can be considered to be a tensorial quantity and even derived from atomistic simulations using dynamic coarse graining theory[26]. The factor \widetilde{W}_{ij} is different from the one in sec. 15 in that it affects pairs of particles (instead of each individual one), it also represents a Gaussian random number with zero mean and unit standard deviation, but must be chosen independently for each pair while ensuring symmetry so that $\widetilde{W}_{ij} = \widetilde{W}_{ji}$. The weight function $\omega(r)$ is a soft repulsive force usually defined as

$$\omega(r) = \begin{cases} \alpha \left(1 - \frac{r}{r_c}\right) & r < r_c \\ 0 & r \geq r_c \end{cases} \quad (16.3)$$

Where r_c is a cut-off distance. The strength parameter, α , can in principle be different for each pair of particles $,i-j$, but for

simplicity we will assume it is the same for every pair. In fact, the conservative force, \mathbf{F}_c can be derived from a bottom-up approach, resulting in the gradient of an effective potential[26].

Numerical integration of the SDE (16.1) is tricky for the same reasons as with Eq. (15.1). We have already described some methods of solving such an equation in sec. 15.1. However, the Grønbech-Jensen method is not valid for DPD since in this case the forces depend on the velocities of the particles. A simple modification can be made, sacrificing stability, by approximating the velocity to just first order in Eq. (15.17), so that the velocity depends only on the force for the current step. Unfortunately, this leads to artifacts in the transport properties and unacceptable temperature drifts. There are several strategies in the literature trying to overcome this, usually presented as modifications of the velocity Verlet algorithm. A good summary can be found in [66]. These methods include empirical tweaks to velocity Verlet via a free parameter[64], a self-consistent correction to the velocity to avoid the temperature drift[67] or a recomputation of the dissipative forces after the velocity update[68]. All of these methods improve the accuracy of the predicted transport properties and response functions in exchange for increased computational cost. One popular approach is to simply use velocity Verlet as described in sec. 14.2 with the forces in Eq. (16.2). This yields “poor” stability and presents certain artifacts[68] due to the mistreatment of the derivative of the noise term incurred by treating Eq. (16.1) as an ODE instead of a proper SDE. However, it is often good enough and while it might require a smaller time step to recover measurables to an acceptable tolerance it is the fastest approach and trivial to implement in a code already providing the velocity Verlet algorithm.

USE IN UAMMD

In UAMMD, DPD is encoded as a *Potential* that can be used to specialize the *PairForces* module in sec. 10, the resulting *Interactor* can then be coupled to a *VerletNVE Integrator* (described in sec. 14.2).

```

#include <uammd.cuh>
#include <Integrator/VerletNVE.cuh>
#include <Interactor/PairForces.cuh>
#include <Interactor/Potential/DPD.cuh>
using namespace uammd;
//A function that creates and returns a DPD
→ integrator
auto createIntegratorDPD(UAMMD sim){
    Potential::DPD::Parameters par;
    par.temperature = sim.par.temperature;
    //The cut off for the weight function
    par.cutOff = sim.par.cutOff;
    //The friction coefficient
    par.gamma = sim.par.friction;
    //The strength of the weight function
    par.A = sim.par.strength;
    par.dt = sim.par.dt;
    auto dpd = make_shared<Potential::DPD>(dpd);
    //From the example in PairForces
    auto interactor =
        → createPairForcesWithPotential(sim, dpd);
    //From the example in the MD section
    // particle velocities should not be initialized
    // by VerletNVE (initVelocities=false)
    auto verlet = createVerletNVE(sim);
    verlet->addInteractor(interactor);
    return verlet;
}

```

Source Code 17: Creation of an instance of a **DPD** integrator. In **UAMMD**, **DPD** is encoded as a Verlet integrator coupled with a short range interaction.

It is worth mentioning that, following the “one *Integrator* with many *Interactors*” philosophy of UAMMD, in addition to the **DPD** potential, one can add any other interactions to the **verlet Integrator** returned by the function in source code (such as bonds or an additional steric repulsion just to name a few).

17

BROWNIAN DYNAMICS

When the viscous forces are much larger than the inertial forces, i.e. $|\xi \mathbf{u}| \gg |m\mathbf{a}|$, the inertial term in Eq. (15.8) becomes irrelevant at very short time scales. **BD** takes advantage of such a time scale separation between the particle velocity fluctuations and its displacement and can be interpreted as the overdamped, or non-inertial, limit of **LD**. The decorrelation time of the velocity, defined as $\tau_l = m/\xi$ is much faster than the time needed for a particle to move farther than its own size. **BD** represents the long time limit of the Langevin equation. This is a powerful property of **BD**, since sampling the probability distributions of the underlying stochastic processes (stemming from the rapid movement of the solute particles) does not require sampling their fast dynamics.

In **BD**, the coupling between the submerged particles and the solvent is instantaneous. Furthermore, since the particle velocities decorrelate instantly, the only remaining relevant variables are the positions of the colloidal particles. We can use the Fokker-Planck formalism to describe the dynamics of the colloidal particles positions. When the **FPE** (3.1) is restricted to Brownian particles it is known as the Smoluchowski Diffusion Equation.

Using Eq. (3.18), we can derive the **SDE** for the positions of a collection of colloidal particles with positions $\mathbf{q} = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_N\}$ from the Smoluchowski description of the two-body level marginal probability [27] as

$$d\mathbf{q} = \widetilde{\mathbf{D}}^{(1)}(\mathbf{q})dt + \delta\mathbf{q} \quad (17.1)$$

Where $\delta\mathbf{q}$ is the set of random displacements which satisfy the two particle covariance given by Eq. (3.17)

$$\langle \delta\mathbf{q} \otimes \delta\mathbf{q} \rangle = 2\mathcal{D}dt \quad (17.2)$$

Following the discussion in section 3.3.3 we can define the drift vector as

$$\mathbf{D}^{(1)} := -\mathcal{M}\partial_{\mathbf{q}}U(\mathbf{q}) + \mathbf{v}_d(\mathbf{q}), \quad (17.3)$$

Here $\mathbf{v}_d(\mathbf{q})$ represents the drift of a base flow moving the particles along. The potential energy, U , acting on the particles can include

pairwise interactions, external fields and/or external non conservative force fields (as in optofluidics [69]). In particular, we can define

$$\mathbf{F} = \partial_{\mathbf{q}} U(\mathbf{q}) + \mathbf{F}^{\text{ext}}(\mathbf{q}) \quad (17.4)$$

Let us focus on the dissipative part of the dynamics, encoded in the mobility tensor. The pairwise mobility tensor, \mathcal{M} , encodes in its elements the hydrodynamic transfer of the force at one point \mathbf{q}_i , to the displacement at another point \mathbf{q}_j . It thus determines the correlations between two particle displacements (see also Eq. (17.2)). The mobility tensor is then related to the diffusion coefficients via the Einstein relation (see sec. 3.3.3),

$$\mathcal{D}(\mathbf{q}) = k_B T \mathcal{M}(\mathbf{q}). \quad (17.5)$$

The Einstein relation coupled with the condition in Eq. (17.2) hints that mobility tensor should be symmetric and positive semi-definite, so that we can define a tensor $\mathcal{B} := \sqrt{\mathcal{M}}$ such that

$$\mathcal{B}\mathcal{B}^T := \mathcal{M}. \quad (17.6)$$

If the system is translationally invariant (isotropic)¹ $\mathcal{M}(\mathbf{q}, \mathbf{q}') = \mathcal{M}(\mathbf{q} - \mathbf{q}')$. To model hydrodynamic couplings, \mathcal{M} is usually taken to be the Rotne-Prager-Yamakawa tensor (describing up to the second hydrodynamic reflection) [27]. Including further reflections explodes the complexity of the formulation and it is only useful in situations in which particles are really close (e.g. high densities) [27], in which case the lubrication approximation is preferable.

We can then write the equations of BD as

$$d\mathbf{q} = \mathbf{v}_d(\mathbf{q}) + \mathcal{M}\mathbf{F}dt + \sqrt{2k_B T \mathcal{M}} d\widetilde{\mathbf{W}} + k_B T \partial_{\mathbf{q}} \cdot \mathcal{M} dt. \quad (17.7)$$

Where \mathbf{F} are the forces acting on the particles and the (so-called) Brownian motion, $d\widetilde{\mathbf{W}}$, is a collection of Wiener increments with zero mean and

$$\langle d\widetilde{W}_i^\alpha(t) d\widetilde{W}_j^\beta(t') \rangle = dt \delta(t - t') \delta_{ij} \delta_{\alpha\beta}. \quad (17.8)$$

It is worth noticing that the physical origin of the noise, $d\widetilde{\mathbf{W}}$, is no longer just a random force coming from an additive action of

¹ Note that there are situations in which this assumption does not hold. Such is the case if a wall is present, where the mobility becomes dependent on the distance to the wall, $\mathcal{M} = \mathcal{M}(\mathbf{q} - \mathbf{q}', z, z')$.

many collisions with solvent particles, but the average of a series of stochastic displacements over a period of time which is short compared to the diffusion time of one particle (but long compared with the relaxation time of the solvent velocity fluctuations via sound or vorticity transport).

The last term in Eq. (17.7), known as the thermal drift term, can be non zero in cases where rigid particle constraints are imposed [70] or due to the presence of boundaries [71, 72]. However, in isotropic and irrotational hydrodynamic fields, this term is usually zero and can thus be omitted from the description.

In practice, we can obtain the actual form of the mobility matrix by directly measuring correlations between random displacements of different particles [73] in an experiment, simply by applying Eqs. (17.2) and (17.5). However, in the case of solutes in a Newtonian fluid it is possible to obtain analytical expressions for \mathcal{M} using standard hydrodynamic theory. In future chapters we will study several analytical instances of \mathcal{M} in various regimes.

For the time being, we will focus on the simple case of the mobility being non-zero only on the diagonal (neglecting hydrodynamic interactions). Furthermore, we assume the self-mobility is equal for all particles so that $\mathcal{M}(\mathbf{q}, \mathbf{q}') = M\mathbb{I} = M_0$. Hydrodynamics will then be reintroduced in chapter 18. This order allows us to introduce the stochastic numerical integration schemes and their use in UAMMD before dealing with the full mobility tensor, which calls for a series of numerical techniques that will be introduced later.

As discussed in sec. 3.3.3, in the case of a no-slip sphere of radius a moving inside a fluid with viscosity η the bare self-mobility is $M_0 = (6\pi\eta a)^{-1}$.

Since the bare self-mobility is just a number when mutual hydrodynamic interactions are not considered, the thermal drift trivially disappears and Eq. (17.7) is greatly simplified, leaving

$$d\mathbf{q} = M\mathbf{F}dt + \sqrt{2k_B T M} d\widetilde{\mathbf{W}}, \quad (17.9)$$

where the stochastic term, $d\widetilde{\mathbf{W}}$, is the origin of the Brownian motion being a Wiener increment (independent random numbers with zero mean and variance dt). A solution to Eq. (17.9) is one that satisfies

$$\mathbf{q}(t) = \mathbf{q}_0 + \int_0^t M\mathbf{F}dt + \int_0^t \sqrt{2k_B T M} d\widetilde{\mathbf{W}} \quad (17.10)$$

Numerical integration

In order to numerically integrate Eq. (17.9) we must discretize the Brownian motion in (15.11). We can do so by integrating it during a small time interval, $[t_n, t_{n+1}]$, where $t_{n+1} - t_n = \delta t$. If the time interval is small enough, we can approximate the Riemann sum by only its first term. Using the discrete time notation we can write

$$\int_{t_n}^{t_{n+1}} f(t') d\tilde{W}(t') \approx f^n \tilde{W}^n = f^n (\tilde{W}^{n+1} - \tilde{W}^n), \quad (17.11)$$

where \tilde{W}^n are independent Gaussian random variables with zero mean and standard deviation δt . Naturally, this approximation is better as $\delta t \rightarrow 0$. With this discretization of the noise we can devise several strategies to numerically integrate Eq. (17.9). The simplest one is the so-called **Euler-Maruyama (EM)** scheme [74].

Euler Maruyama

Using Eq. (17.10) and (17.11) we can write

$$\mathbf{q}^{n+1} = \mathbf{q}^n + M \mathbf{F}^n \delta t + \sqrt{2D} \tilde{\mathbf{W}}^n \quad (17.12)$$

It can be proved that this algorithm is weakly convergent with order 1 and strongly convergent with order 1/2 [75]. Thus the **EM** scheme is not particularly accurate, even in the absence of fluctuations. There are, however, other solvers that yield better accuracy at the same, or similar, computational cost. Let's see some of them.

Adams Bashforth

The **Adams-Bashforth (AB)** scheme uses the forces from the previous step to improve the prediction for the next. This incurs the overhead of storing the previous forces but its computational cost is marginally larger than **EM**. This algorithm mixes a first order method for the noise with a second order method for the force. It yields better accuracy than **EM**, although this comes from experience since as of the time of writing no formal work has been done on its weak accuracy. An empirical demonstration of its better accuracy can be found in [76].

$$\mathbf{q}^{n+1} = \mathbf{q}^n + M \left[\frac{3}{2} \mathbf{F}^n - \frac{1}{2} \mathbf{F}^{n-1} \right] \delta t + \sqrt{2D} \tilde{\mathbf{W}}^n \quad (17.13)$$

Leimkuhler

Another integrator recently developed by Leimkuhler in [77] While also a first order method it seems to yield better accuracy than **AB** and **EM**. I am not aware of any formal studies of its accuracy besides the one in [78]. The update rule is very similar to **EM** but uses noise from two steps (which are generated each time instead of stored)

$$\mathbf{q}^{n+1} = \mathbf{q}^n + M\mathbf{F}^n\delta t + \sqrt{\frac{D}{2}} [\tilde{\mathcal{W}}^n + \tilde{\mathcal{W}}^{n-1}] \quad (17.14)$$

Note that, as stated in [77], while this solver seems to be better than the rest at sampling equilibrium configurations, it does not correctly solve the dynamics of the problem.

Midpoint

Finally, a more sophisticated integrator which we are going to refer to as Mid Point is described in [79]. It is a two step explicit midpoint predictor-corrector scheme. It has a fourth order convergence scaling at the expense of having twice the cost of a single step method, as it requires to evaluate forces twice per update. Noise has to be remembered as well but in practice it is just regenerated instead of stored. Midpoint updates the simulation with the following rule

$$\begin{aligned} \mathbf{q}^{n+1/2} &= \mathbf{q}^n + \frac{1}{2}M\mathbf{F}^n\delta t + \sqrt{D}\tilde{\mathcal{W}}^{n_1} \\ \mathbf{q}^{n+1} &= \mathbf{q}^n + M\mathbf{F}^{n+1/2}\delta t + \sqrt{D}[\tilde{\mathcal{W}}^{n_1} + \tilde{\mathcal{W}}^{n_2}] \end{aligned} \quad (17.15)$$

Where both instances of the noise (n_1 and n_2) are uncorrelated and $\langle \tilde{\mathcal{W}}^{n_i} \rangle^2 = \delta t$.

Use in UAMMD

Using the **BD Integrators** is quite similar to the rest we have seen so far. An example code is available in code 18. The same interface joins all the **BD** integrators we saw in this chapter.

```

#include <uammd.cuh>
#include <Integrator/BrownianDynamics.cuh>
using namespace uammd;
//A function that creates and returns a BD
→ integrator
auto createIntegratorBD(UAMMD sim){
    //Choose the method
    using BD = BD::EulerMaruyama;
    //using BD = BD::MidPoint;
    //using BD = BD::AdamsBashforth;
    //using BD = BD::Leimkuhler;
    BD::Parameters par;
    par.temperature=sim.par.temperature;
    par.viscosity=sim.par.viscosity;
    //If the hydrodynamic radius is not set (or it is
    → equal to -1) the BD Integrator will use the
    → radius of each particle in ParticleData
    par.hydrodynamicRadius=sim.par.hydrodynamicRadius;
    par.dt=sim.par.dt;
    auto bd = std::make_shared<BD>(sim.pd, par);
    return bd;
}

```

Source Code 18: Example of the creation of a **BD Integrator** module.

17.1 METROPOLIS ADJUSTED LANGEVIN ALGORITHM (MALA)

Sometimes we are not interested in the dynamics of a simulation, rather we are looking to sample equilibrium states or simply approaching equilibrium as soon as possible. In these instances the **BD** machinery previously introduced might not be the optimal strategy. We can use Monte Carlo methods to sample equilibrium configurations. Alas Monte Carlo methods often prove to be challenging to parallelize, specially to the degree required in a **GPU**. For instance, proposing a new configuration in the standard Metropolis Monte Carlo algorithm consists in moving just one particle at a time and in order to correctly measure the total energy change of the system the rest of the system must not change during the process, virtually serializing the algorithm. One simple parallelization is to sample new configurations by randomly moving all the particles. However, the chances of this new configuration being more favorable or altogether valid are, at most, slim. The

Metropolis Adjusted Langevin Algorithm (MALA) [80] attempts to increase the acceptance chance of moving all the particles in the system by guiding the sampling process using the particle forces in addition to the total system energy change. In this regard, the MALA algorithm can also be referred to as a Force Biased Metropolis Monte Carlo scheme.

A step in MALA amounts to performing a **BD** simulation step and then using the positions, forces and energies of the old and new configurations to decide whether to accept it or not with a Metropolis-like probability. Let us now study the MALA acceptance rule in detail.

We can interpret the Euler-Maruyama Brownian Dynamics scheme in Eq. (17.12) (with unit self mobility, $M = 1$) as a Markov chain with a transition kernel following a strictly positive probability transition density defined as

$$p(\mathbf{Q}^n, \mathbf{Q}^{n+1}) = (4\pi k_B T \delta t)^{3/2} \exp\left(\frac{|\mathbf{Q}^{n+1} - \mathbf{Q}^n + \delta t \mathbf{F}^n|^2}{4k_B T \delta t}\right) \quad (17.16)$$

Where $\mathbf{Q}^n := \mathbf{q}_1^n, \mathbf{q}_2^n, \dots, \mathbf{q}_N^n$ represents the positions of all particles at time step n . On the other hand, the Langevin equation (17.9) at equilibrium presents a probability density given by

$$\pi(\mathbf{Q}) = Z^{-1} \exp\left(-\frac{U(\mathbf{Q})}{kT}\right) \quad (17.17)$$

Where the entropy is defined as $Z = \int \exp(-U(\mathbf{Q})/k_B T)$ and $U(\mathbf{Q})$ is the internal energy (such that $\mathbf{F} = -\nabla U$).

Once we have a proposed transition kernel (Eq. (17.16)) and a target probability density (the equilibrium one in Eq. (17.17)) we can use the Metropolis-Hastings method to design an updating scheme, referred to as MALA [81] [82].

Instead of sampling configurations using the Euler-Maruyama scheme (as discussed in sec. 17) we now propose a new configuration using Eq. (17.12) as

$$\tilde{\mathbf{Q}}^{n+1} = \mathbf{Q}^n + \delta t \mathbf{F}^n + \sqrt{2k_B T \delta t} \tilde{\mathcal{W}}^n \quad (17.18)$$

And accept it with a Metropolis probability given by

$$\alpha(\mathbf{Q}^n, \tilde{\mathbf{Q}}^{n+1}) = \min\left(\frac{p(\tilde{\mathbf{Q}}^{n+1}, \mathbf{Q}^n) \pi(\tilde{\mathbf{Q}}^{n+1})}{p(\mathbf{Q}^n, \tilde{\mathbf{Q}}^{n+1}) \pi(\mathbf{Q}^n)}\right) \quad (17.19)$$

Numerically, this can be done by drawing a uniform random number $\xi^n \in [0, 1]$ and accepting the new configuration only if $\xi^n < \alpha(Q^n, \tilde{Q}^{n+1})$.

In contrast to the naive Metropolis-Hastings algorithm, that blindly proposes configurations, MALA proposes configurations that always have a higher probability in π , increasing the chances of acceptance.

We are not restricted to using the same value for δt in all steps² since in an MCMC algorithm δt has lost its meaning as a *time step* and it is now simply a parameter describing the *distance* in phase space between two given configurations. As such, we can tune it to get a desired configuration acceptance ratio. With MALA we are not solving the dynamics of the system, but rather sampling equilibrium configurations (or moving the system towards equilibrium). We can use the MALA algorithm and its ability to handle any δt for thermalization or to get a running simulation “unstuck”. For instance, if a simulation of hard spheres (modeled with some repulsive potential, for instance the WCA in sec. 10.1) has reached an invalid configuration due to thermal fluctuations (i.e. two particles overlapping, which promptly causes the simulation to “explode”) the MALA algorithm could be used to safely move the simulation away from this state (since MALA will mostly modify the state of the configuration only when a new configuration is more favorable).

Although MALA proves to be useful for thermalising or initializing simulations, testing suggests that hard steric repulsion between particles (like LJ) requires such a small time step to ensure a reasonable acceptance ratio that the algorithm becomes unusable in practice. In these cases using **BD** directly is preferable.

In order to tune the jump step (δt) to ensure a certain acceptance ratio on average, we measure the acceptance ratio every once in a while, if it is lower than the target the jump step is decreased (and increased if it is larger). We increase the jump step slowly (a 2% increase) and decrease it fast (a 10%) as it is usually done in optimization algorithms.

Another disadvantage of MALA is that usually the energy change coming from moving all particles in the system grows, on average, with the number of particles. This results in the optimal jump

² Theoretically the step size should not be adjusted dynamically for correct sampling (as it violates detailed balance).

step decreasing with the number of particles. Thus defeating the purpose.

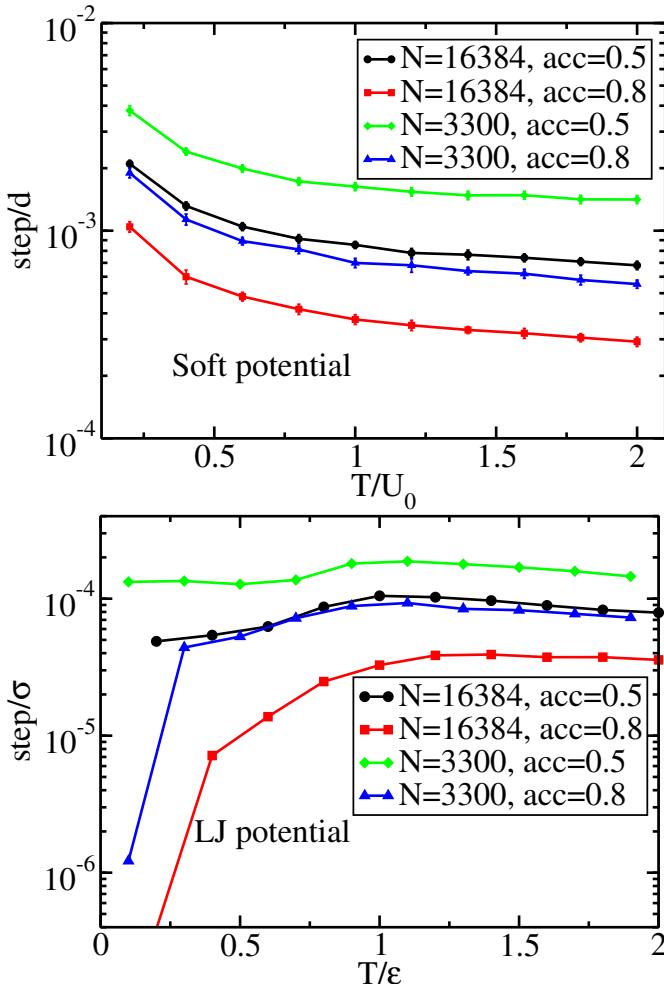


Figure 17.1: Optimal time step in MALA for several acceptance ratios and number of particles using a soft repulsive potential given by Eq. (17.20) (left) and a hard steric WCA interaction (right). Naturally, a higher acceptance ratio (with 1 corresponding to accepting all configurations and 0 to never accepting them) results in a lower step size, δt . A by-product of the MALA algorithm comparing the total energy change is that δt depends on the number of particles so that more particles result in a lower optimal δt . In all cases the number density of particles is kept at $\rho := N/V = 0.1a^{-3}$.

In Fig. 17.1 we compare the optimal step size of the MALA algorithm for different acceptance ratios in different scenarios. In particular, we compare results using a hard-repulsion WCA potential (truncated LJ, see sec. 10.1) with a soft core repulsive interaction given by the potential

$$U(r) = U_0 \begin{cases} \exp(-(r-d)/b) & r > d \\ 1 - (r-d)/b & r \leq d \end{cases} \quad (17.20)$$

Where U_0 represents the repulsive strength, d is a typical repulsive radius and we choose the scale factor $b = 0.1$.

Use in UAMMD

The MALA algorithm is available in **UAMMD** as an *Integrator* module. Its usage is similar to the ones we have seen thus far. The MALA module will query its *Interactors* for both force and energy (whereas, for instance, **BD**, will only request forces). MALA will autotune the step size (δt) to ensure a given acceptance ratio.

```
#include <uammd.cuh>
#include <Integrator/MonteCarlo/ForceBiased.cuh>
using namespace uammd;
//A function that creates and returns a MALA
// integrator
auto createIntegratorMALA(UAMMD sim){
    MC::ForceBiased::Parameters par;
    //Inverse of temperature
    par.beta = 1.0/sim.par.temperature;
    //Initial step length.
    //It will be optimized according to the target
    // acceptance ratio
    par.stepSize = sim.par.dt;
    //Desired acceptance ratio
    par.acceptanceRatio = sim.par.accRatio;
    auto mala =
        std::make_shared<MC::ForceBiased>(sim.pd, par);
    return mala;
}
```

Source Code 19: Example of the creation of a MALA *Integrator* module.

18

HYDRODYNAMIC INTERACTIONS

In our discussion about **BD** in chapter 17 we neglected hydrodynamic interactions by considering only the diagonal terms of the mobility tensor in Eq. (17.7) (the so-called self-mobility). We considered the effect of the solvent on the submerged particles without considering the opposite interaction. Particles disturb the fluid around them, which in turn has an effect on the rest. When the hydrodynamic displacements are no longer negligible (for instance, due to the density of colloidal particles increasing), the full form of the mobility tensor must be taken into account. Alternatively, hydrodynamic effects can be taken into account (to some extent) by particle solvers equipped with local momentum conservation (such as **DPD**, see chapter 16). However, **DPD** (and **Smoothed Particle Hydrodynamics (SPH)**) explicitly solve the solvent particle dynamics (or some coarse-grained version of them). Furthermore, the **DPD** and **SPH** methods lie in the inertial regime of the solvent, forcing us to describe a fast time scale that might be of no interest in a given model, in particular the sound and viscous momentum propagation (vorticity). Thus far we have taken a bottom-up approach, starting by modeling the dynamics of the solvent particles themselves and then eliminating degrees of freedom aided by the Fokker-Planck formalism. This enabled us to elucidate Eq. (17.7), giving us some clues about the mobility matrix and, more importantly, relating it with the diffusion via the fluctuation-dissipation balance. It is worth now to take a top-down approach by considering the hydrodynamics of the solvent and then bridge the gap back to **BD**, which will give us more information about the actual form of the mobility.

FLUID DYNAMICS

We start by considering the more general incompressible steady Navier-Stokes equations. Then, we will introduce the action of the particles as a force-density on the fluid (propagated by a smoothed delta-like function centered around each particle). We now study how this force propagates to the rest of the fluid.

We will take the non-inertial regime for the dynamics of both the fluid and the particles to connect with BD while taking into account hydrodynamic correlations. In section 20.5 we consider the fluid inertial regime.

The incompressible Navier-Stokes equations can be written as [83]

$$\underbrace{\rho \partial_t \mathbf{v} + \rho \nabla \cdot (\mathbf{v} \otimes \mathbf{v})}_{\substack{\text{fluid inertia} \\ \text{time} \\ \text{convection}}} + \underbrace{\nabla \pi}_{\substack{\text{internal stress}}} = \underbrace{\eta \nabla^2 \mathbf{v}}_{\substack{\text{momentum diffusion}}} + \underbrace{\mathbf{f} + \nabla \cdot \mathcal{Z}}_{\substack{\text{external forces} \\ \text{and} \\ \text{thermal noise}}}$$

$$\underbrace{\nabla \cdot \mathbf{v} = 0}_{\text{incompressibility}}$$
(18.1)

Where $\mathbf{v}(\mathbf{r}, t)$ represents the velocity field of the fluid, π the pressure, ρ its density and η its viscosity. \mathbf{f} is some localized force density acting on the fluid (which can arise from the presence of submerged particles).

The fluctuating hydrodynamic description (Landau-Lipshitz) includes a fluctuating stress tensor, $\mathcal{Z}(\mathbf{r}, t)$, which must comply with the fluctuation-dissipation balance (see sec. 3.3.2) according to the following statistical properties [30]

$$\begin{aligned}
 \langle \mathcal{Z}_{ij} \rangle &= 0 \\
 \langle \mathcal{Z}_{ik}(\mathbf{r}, t) \mathcal{Z}_{jm}(\mathbf{r}', t') \rangle &= 2k_B T \eta (\delta_{ij} \delta_{km} + \delta_{im} \delta_{kj}) \delta(\mathbf{r} - \mathbf{r}') \delta(t - t')
 \end{aligned}$$
(18.2)

Where i, j, k, m represent the different coordinates. In order to generalize the notation for Eq. (18.1), we will sometimes introduce the fluctuations into the fluid forcing term, defining $\tilde{\mathbf{f}} := \mathbf{f} + \nabla \cdot \mathcal{Z}$. We can also write Eq. (18.1) as

$$\rho \partial_t \mathbf{v} = -\nabla \cdot \boldsymbol{\sigma} + \tilde{\mathbf{f}}, \quad (18.3)$$

with

$$\boldsymbol{\sigma} := \pi \mathbb{I} - \eta (\nabla \mathbf{v} + \nabla \mathbf{v}^T) + \rho (\mathbf{v} \otimes \mathbf{v}) \quad (18.4)$$

PARTICLE DYNAMICS

To derive the equation of motion of the particle, we impose the kinetic constraint $\mathbf{u} = \int_{V_p} \mathbf{v}(\mathbf{r}, t) d\mathbf{r}$, where V_p is the particle volume. The kinetic constraint imposes instantaneous coupling between

the particles and the surrounding fluid (this condition represents no-slip of the fluid on the “real” particle’s surface). For the purpose of this derivation, we keep fluctuations out of the description and integrate over the particle volume in Eq. (18.3),

$$\rho \int_{V_p} \partial_t \mathbf{v} d\mathbf{r} = - \int_{V_p} \nabla \cdot \boldsymbol{\sigma} d\mathbf{r} + \int_V \mathbf{f} d\mathbf{r}. \quad (18.5)$$

Note that

$$\rho \int_{V_p} \partial_t \mathbf{v} d\mathbf{r} = \rho \partial_t \int_{V_p} \mathbf{v} d\mathbf{r} = \rho \partial_t \mathbf{u}$$

and using the divergence theorem

$$- \int_{V_p} \nabla \cdot \boldsymbol{\sigma} d\mathbf{r} = \oint_S \boldsymbol{\sigma} \cdot \hat{\mathbf{n}} d\mathbf{r} = \mathbf{F}_{\text{drag}}$$

is the fluid traction on the particle (drag force). We also note that $\int_{V_p} \mathbf{f} d\mathbf{r} = \boldsymbol{\lambda}$ is the total force exerted by the particle on the fluid (induced force [84]). Hence, $m_f \partial_t \mathbf{u} = \mathbf{F}_{\text{drag}} + \boldsymbol{\lambda}$, with m_f being the mass of the fluid displaced by the particle. On the other hand, $m_p \dot{\mathbf{u}} = \mathbf{F}_{\text{drag}} + \mathbf{F}$ where \mathbf{F} is any other force acting on the particle (aside from the fluid drag). Subtracting and introducing the Archimedean excess mass, $m_e := m_p - m_f$,

$$m_e \dot{\mathbf{u}} = \mathbf{F} - \boldsymbol{\lambda}. \quad (18.6)$$

Which implies, for $m_e = 0$, $\boldsymbol{\lambda} = \mathbf{F}$. Thus any force on the particle goes to the fluid. In fact, this is another important assumption in Brownian dynamics; neglect the particle inertia.

THE STOKES LIMIT

If we take the overdamped limit of Eq. (18.1), where the momentum of the fluid can be eliminated as a fast variable (allowing to neglect the transient term $\partial_t \mathbf{v}$ as well as the convection¹²⁾) we get the so-called Stokes equations

$$\begin{aligned} \nabla \pi - \eta \nabla^2 \mathbf{v} &= \tilde{\mathbf{f}}, \\ \nabla \cdot \mathbf{v} &= 0. \end{aligned} \quad (18.7)$$

¹ Neglecting convection is valid for small Reynolds number hydrodynamics, i.e., $\text{Re} = \frac{\eta v}{\rho L} \ll 1$ with L the smallest characteristic length of the system (e.g. particle radius).

² Moreover, we assume that the Schmidt number is very large, $S_c = \eta/(\rho D_0) \gg 1$, where $D_0 = k_B T / (6\pi\eta a)$ is the typical diffusion coefficient of a submerged particle (see sec. 3.3.3), which implies that fluid momentum propagates much faster than particle diffusion. For $S_c \gg 1$ the transient term $\rho \partial_t \mathbf{v}$ can be neglected, which is a sane approximation (even for proteins in water).

We can eliminate the pressure from the description by using the projection method. Let's take the divergence of the first equation

$$\nabla^2 \pi - \eta \nabla \cdot (\nabla^2 \mathbf{v}) = \nabla \cdot \tilde{\mathbf{f}} \quad (18.8)$$

Since the divergence of the velocity is zero the second term vanishes, so we can formally write

$$\pi = \nabla^{-2}(\nabla \cdot \tilde{\mathbf{f}}) \quad (18.9)$$

Replacing the pressure in Eq. (18.7)

$$\eta \nabla^2 \mathbf{v} = \nabla \left[\nabla^{-2}(\nabla \cdot \tilde{\mathbf{f}}) \right] - \tilde{\mathbf{f}} = -\mathcal{P} \tilde{\mathbf{f}} \quad (18.10)$$

Where the projection operator, \mathcal{P} , is formally defined as

$$\mathcal{P} := \mathbb{I} - \nabla \nabla^{-2} \nabla. \quad (18.11)$$

\mathcal{P} projects onto the space of divergence-free velocity. \mathbb{I} represents the identity. In the particular case of an unbounded domain with fluid at rest at infinity, all the differential operators in Eq. (18.11) commute in Fourier space, so that

$$\widehat{\mathcal{P}}(\mathbf{k}) = \mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \quad (18.12)$$

Where \mathbf{k} are the wave numbers. Finally, we can identify

$$\mathcal{L} := -\nabla^{-2} \mathcal{P} \quad (18.13)$$

as the Stokes solution operator to arrive at

$$\mathbf{v} = \eta^{-1} \mathcal{L} \tilde{\mathbf{f}} \quad (18.14)$$

The Green's function, \mathcal{G} , of Eq. (18.14) in the case of an unbounded domain can be written in Fourier space as

$$\eta^{-1} \mathcal{L}(\mathbf{k}) \rightarrow \widehat{\mathcal{G}}(\mathbf{k}) := \eta^{-1} k^{-2} \widehat{\mathcal{P}}(\mathbf{k}) \quad (18.15)$$

The inverse transform of Eq. (18.15) can be computed analytically (with the help of some known mathematical relations [85]) to get

$$\mathcal{O}(\mathbf{r}) := \frac{1}{8\pi\eta r} \left(\mathbb{I} - \frac{\mathbf{r} \otimes \mathbf{r}}{r^2} \right) \quad (18.16)$$

This solution is known as the Oseen tensor [85], the response of a three dimensional unbounded fluid at rest at infinity to a delta forcing. Figure 18.1 contains a visual representation of the Oseen tensor.

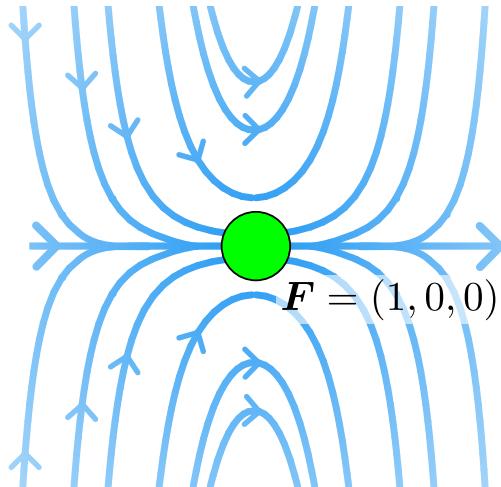


Figure 18.1: Representation of the Oseen mobility in 2D. A force acting on a point (green) is propagated to the fluid via the Oseen tensor (blue lines). The fluid behind the source will be dragged onto it, while the fluid in front will be pushed away. Note that the action of the Oseen tensor decays as the inverse of the distance, a behavior this representation does not convey.

COUPLING WITH PARTICLES

Eqs. (18.14) and (18.15) constitute a formalism to translate forces to velocities and permits unveiling their correlated displacements.

In the following derivation we are going to describe the particle effect on the fluid using a generalized delta function, $\delta_a(\mathbf{r})$. Using a monopolar approximation, we set the force distribution as $\mathbf{f}(\mathbf{r}) = \lambda\delta_a(\mathbf{r})$. Here, $\delta_a(\mathbf{r})$ is a distribution of compact support (usually a smooth smeared delta function). The subscript a refers to the hydrodynamic radius of the particle, related to the typical size of the particle around which the kernel distributes the particle force, or any other extensive quantity (e.g acoustic forces [86], torques, etc).

The velocity of a certain particle $i \in [1, N]$ is computed as

$$\mathbf{u}_i = \mathcal{J}_{\mathbf{q}_i} \mathbf{v} = \int \delta_a(\mathbf{q}_i - \mathbf{r}) \mathbf{v}(\mathbf{r}) d\mathbf{r}. \quad (18.17)$$

So that $\delta_a(\mathbf{r})$ is also used to interpolate fluid properties to the particle “language”. The operator \mathcal{J} is called interpolation operator.

Eq. (18.14) states that the velocity of the particle is equal to the local average of the fluid velocity.

The reverse of interpolation is spreading. As stated, we compute the force density acting on the fluid at a given point \mathbf{r} in space by spreading the forces of all particles according to

$$\mathbf{f}(\mathbf{r}) = \mathcal{S}(\mathbf{r})\mathbf{F} = \sum_i \delta_a(\mathbf{q} - \mathbf{r}_i)\mathbf{F}_i, \quad (18.18)$$

where we have used the supervector notation [27], $\mathbf{F} := \{\mathbf{F}_1, \dots, \mathbf{F}_N\}$. The benefit of using the same kernel to spread and interpolate is that it makes the operations adjoint to each other, $\mathcal{J} = \mathcal{S}^*$, which is required in the following derivation [87].

The physical volume ΔV of a particle is related to the kernel via

$$\Delta V = (\mathcal{JS} 1)^{-1} = \left(\int \delta_a^2(\mathbf{r}) d\mathbf{r} \right)^{-1}, \quad (18.19)$$

where \mathcal{S} is applied to the unit constant function³⁴. We will discuss spreading and interpolation in a regular grid in more detail in sec. 21.

18.1 CONNECTION WITH BROWNIAN DYNAMICS

We can now write an equation for the displacements (velocities) of a group of particles using Eqs. (18.14) and (18.15) as

$$\frac{d\mathbf{q}_i}{dt} = \mathbf{u}_i = \eta^{-1} \mathcal{J}_{\mathbf{q}_i} \mathcal{L}(\mathcal{S}\mathbf{F} + \nabla \cdot \mathcal{Z}). \quad (18.20)$$

This equation is reminiscent of Eq. (17.7) by defining the mobility as

$$\mathcal{M} = \eta^{-1} \mathcal{J} \mathcal{L} \mathcal{S}. \quad (18.21)$$

and its “square root” as

$$\mathcal{M}^{1/2} = \eta^{-1/2} \mathcal{J} \mathcal{L} \nabla \cdot . \quad (18.22)$$

³ We abuse the notation here and assume that the coupling operators can be applied to a scalar, a vector or a tensor field, understanding that the same spreading/interpolation operation is applied to each component independently.

⁴ Note that we applied both \mathcal{J} and \mathcal{S} to a system with only one particle.

It can be easily proved, by using that $-\mathcal{L}\nabla^2\mathcal{L} = \mathcal{L}$ and that $\mathcal{J}^* = \mathcal{S}^5$ [87]⁶, that this definition of the square root of the mobility satisfies the fluctuation-dissipation balance⁷ with \mathcal{Z} given by Eq. (18.2). Naturally, the definition in Eq. (18.22) satisfies

$$\mathcal{M} = \mathcal{M}^{1/2} (\mathcal{M}^{1/2})^*. \quad (18.23)$$

The definition in Eq. 18.21 allows us to realize that Eq. (18.20) and (17.7) are, in fact, equivalent.

We are now ready to extract more information about the mobility matrix, which is in general a configuration-dependent tensor. In particular, writing Eq. (18.21) as the double convolution with the kernel at the position of two particles yields a pairwise mobility, depending only on the positions of the particles as

$$\mathcal{M}_{ij} = \eta^{-1} \iint \delta_a(\mathbf{q}_j - \mathbf{r}) \mathcal{L}(\mathbf{r}, \mathbf{r}') \delta_a(\mathbf{q}_i - \mathbf{r}') d\mathbf{r} d\mathbf{r}'. \quad (18.24)$$

Here \mathcal{L} is the Green's function for Eq. (18.7) with the particular boundary conditions.

The mobility tensor definition in Eq. (18.24) is general for any geometry. For an unbounded fluid, the Green's function is the Oseen tensor in Eq. (18.16) and the resulting mobility (when using $\delta_a := \delta$) is referred to as the Oseen mobility, so that $\mathcal{M}_{ij} = \mathcal{O}(\mathbf{q}_j - \mathbf{q}_i)$.

The Oseen mobility presents several problems when particles get too close stemming from the point particle assumption. For instance, the tensor becomes ill-formed (and non positive-definite) at close distances, hindering the applicability of the assumption in Eq. (18.23). On the other hand, the Oseen mobility diverges at zero distance, which can result in numerical issues in the absence of steric repulsions and requires treating the self mobility separately. The Oseen mobility can be in a way interpreted as the mobility for colloidal particles with size $a \rightarrow 0$.

In practice we would like to take into account the finite size of the submerged particles, which amounts to choosing a different kernel in Eq. (18.24) or, in the purely particle based description

⁵ The scalar product in the particle and fluid domains are related via $(\mathcal{J}\mathbf{v}) \cdot \mathbf{u} = \int \mathbf{v} \cdot (\mathcal{S}\mathbf{u}) d\mathbf{r} = \int \delta_a(\mathbf{q} - \mathbf{r})(\mathbf{v} \cdot \mathbf{u}) d\mathbf{r}$.

⁶ In particular, because negative divergence is the adjoint of the gradient and $\nabla^2 = \nabla \cdot \nabla = -\nabla \cdot (\nabla \cdot)^*$, we can write $(\mathcal{J}\mathcal{L}\nabla \cdot)(\mathcal{J}\mathcal{L}\nabla \cdot)^* = \mathcal{J}\mathcal{L}\nabla \cdot (\nabla \cdot)^* \mathcal{L}^* \mathcal{J}^* = -\mathcal{J}\mathcal{L}\nabla^2 \mathcal{L}^* \mathcal{S} = \mathcal{J}\mathcal{L}\mathcal{S}$.

⁷ $\langle \delta\mathbf{q}_i \otimes \delta\mathbf{q}_j \rangle = 2k_B T \mathcal{M}(\mathbf{q}_{ij}) \delta t$

of Eq. (17.7), a different (pairwise) approximation to the mobility altogether.

We will see in future sections how to solve the dynamics of the particles without the need for the explicit form for the Green's function by discretizing Eq. (18.17) directly. This can be useful for special geometries in which the differential operators in the Stokes solution operator do not commute or when Eq. (18.24) cannot be solved analytically. For the moment we will stick to imposing a mobility tensor explicitly.

One of the usual choices for the mobility tensor is the **Rotne-Prager-Yamakawa (RPY)** tensor (derived around the same time independently in [88, 89]), describing the hydrodynamic interaction of two spheres of radius a . We can compute it by integrating Eq. (18.24) with the Stokes solution operator (the Oseen tensor) on the surface, S , of two spheres of radius a centered at the positions of the particles (\mathbf{q}_i and \mathbf{q}_j),

$$\begin{aligned}\mathcal{M}_{ij}^{\text{RPY}} &= \frac{\eta^{-1}}{(4\pi a^2)^2} \int_{S_i} dS_i \int_{S_j} dS_j \int d\mathbf{k} \exp(i\mathbf{k} \cdot \mathbf{q}_{ij}) \hat{\mathcal{L}}(\mathbf{k}) \\ &= \eta^{-1} \int \frac{\exp(i\mathbf{k} \cdot \mathbf{q}_{ij})}{k^2} (\text{sinc}(ka))^2 \left(\mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \right) d\mathbf{k}. \end{aligned}\quad (18.25)$$

Expressing this integral in Fourier space allows us to express it in a compact form and, as we will see later, doing so will facilitate its extension to periodic environments. A detailed solution for this integral can be found in [90]. It is important to note that the integral must be handled separately when the two particles overlap. Finally, the free-space RPY mobility tensor can be written in real space as [90]

$$\mathcal{M}^{\text{RPY}}(\mathbf{r} = \mathbf{q}_i - \mathbf{q}_j) = M_0 \begin{cases} \left(\frac{3a}{4r} + \frac{a^3}{2r^3} \right) \mathbb{I} + \left(\frac{3a}{4r} - \frac{3a^3}{2r^3} \right) \frac{\mathbf{r} \otimes \mathbf{r}}{r^2} & r > 2a \\ \left(1 - \frac{9r}{32a} \right) \mathbb{I} + \left(\frac{3r}{32a} \right) \frac{\mathbf{r} \otimes \mathbf{r}}{r^2} & r \leq 2a \end{cases} \quad (18.26)$$

The self mobility, $M_0 := \mathcal{M}(0) = (6\pi\eta a)^{-1}$ is, by no coincidence, equal to the drag for a sphere moving through a Stokesian fluid as given by the Stokes-Einstein relation [27] (see sec. 3.3.3).

This approximation is valid for dilute suspensions (and) to leading order in the far field. The RPY tensor has been proven to accurately capture hydrodynamics even when particles overlap beyond their hydrodynamic radius [91] [90]. Given its popularity

lots of works attempting to generalize or adapt the RPY tensor for different geometries and use cases can be found [90, 92–94]. In particular, we will make use of the generalization in [95], which allows for each particle to have a different hydrodynamic radius.

Notice that, as revealed by Eqs. (18.16) and (18.26), the approximations for the pairwise mobility usually come under the form

$$\mathcal{M}(\mathbf{r}) = f(r)\mathbb{I} + g(r)\frac{\mathbf{r} \otimes \mathbf{r}}{r^2}. \quad (18.27)$$

This is an expression of the fact that the mobility is usually isotropic and translationally invariant.

The mobility tensors elucidated thus far have zero divergence (due to the incompressibility of the three-dimensional flow). However, it is important to take the thermal drift term in Eq. (17.7) into account since even for these cases, certain spatio-temporal discretizations could result in the mobility having a spurious thermal drift. Furthermore, divergence will be non-zero in other geometries (like the confined systems in sec. 22). Although in the use cases throughout this manuscript thermal drift does not pose a significant computational challenge, certain solvers (for instance, when computing hydrodynamic correlations of rigid bodies [70]), for which the application of the Stokes solution operator (or alternatively the evaluation of the mobility) is particularly expensive can result in the thermal drift term having a non-trivial cost. One of the most widespread generic approaches for computing the thermal drift term is to use the so-called Random Finite Differences (RFD) [87], where the divergence of the mobility is computed as

$$\frac{1}{\delta} \left\langle \mathcal{M} \left(\mathbf{q} + \frac{\delta}{2} \widetilde{\mathbf{W}} \right) \widetilde{\mathbf{W}} - \mathcal{M} \left(\mathbf{q} - \frac{\delta}{2} \widetilde{\mathbf{W}} \right) \widetilde{\mathbf{W}} \right\rangle = \partial_{\mathbf{q}} \cdot \mathcal{M}(\mathbf{q}) + O(\delta^2). \quad (18.28)$$

Here $\widetilde{\mathbf{W}}$ is a vector of random Gaussian numbers with zero mean and unit standard deviation and δ is a small parameter (generally chosen as small as possible without incurring in numerical issues). Note that evaluating Eq. (18.28) requires two Stokes solves (or mo-

bility evaluations). It is possible to skip one of them by evaluating the derivative of the kernel only once [87].

19

H Y D R O D Y N A M I C S I N O P E N B O U N D A R I E S

In the following chapters, we will discuss numerical algorithms for hydrodynamic interactions in domains with different periodicities based on the mathematical framework laid out in chapter 18. Let us start by examining systems with fully open boundaries. In this instance, we will describe fully Lagrangian algorithms based on Green's functions in which the whole computation takes place in real space (the standard approach in BD).

Although we are not limited to it, thus far the mobility matrices we have computed explicitly (in Eqs. (18.26) and (18.16)) correspond to systems with open boundaries (since they come from applying the unbounded Green's function in Eq. (18.15)). Let's first see how to solve Eq. (17.7) (rewritten here for the readers convenience),

$$dq = \mathcal{M}Fdt + \sqrt{2k_B T} \mathcal{M}d\tilde{W} + k_B T \partial_q \cdot \mathcal{M}dt,$$

by using the unbounded RPY mobility and then explore other strategies that can be used when assuming periodic boundary conditions.

The fluctuating term in Eq. (17.7) involves finding the square root of the mobility as defined in Eq. (18.23). This requires that the mobility matrix is positive definite. Note that the RPY tensor is positive definite for all particle configurations, whereas the Oseen tensor becomes non positive definite at short distances. Solving Eq. (17.7) numerically by direct evaluation results in an algorithm with a complexity of at least $O(N^2)$ due to the matrix-vector multiplication of the mobility tensor and the forces. Furthermore, finding the square root of the mobility will typically dominate the cost of the algorithm. Even so, there might be situations in which such an algorithm could be the best option. Temporal integration can be achieved with any of the methods described in chapter 17. However, it is important to consider the extreme cost of computing and/or storing both terms in Eq. (17.7) when choosing an algorithm. As such, in UAMMD we usually stick to the Euler-Maruyama method.

The deterministic term can be computed via a library call to a matrix-vector multiplication function. However, this incurs $O(N^2)$ storage. If the full mobility matrix is not needed for a later stage, it can be computed on the fly by using the *NBody* algorithm in sec. 9.

19.1 CHOLESKY

The classic strategy for computing the square root of the mobility, originally proposed by Ermak [91], is by direct Cholesky factorization. This operation requires $O(N^3)$ operations, rendering this algorithm unsuitable for large numbers of particles (above 10^4). Additionally, it has $O(N^2)$ storage requirements, since the full mobility matrix has to be stored. However, the sheer raw power of the GPU can make this a valid option. In UAMMD the Cholesky factorization is accomplished via a single library call to NVIDIA's cuSolver function *potrf* [96]. On the other hand, since the mobility matrix has to be stored anyway, the rest of the algorithm can be coded via a few function calls to a linear algebra library. In particular, the Cholesky module in UAMMD uses the matrix-vector multiplications in the *cuBLAS* library [97]. Taking into account the symmetric form of the mobility matrix, only the upper half needs to be computed and stored, *cuBLAS* (and most linear algebra libraries) provide subroutines that leverage this. In this regard, there is not much possibility for optimization.

Use in UAMMD

In UAMMD, Brownian Dynamics with Hydrodynamic Interactions (BDHI) algorithms are separated between temporal integration schemes and strategies for computing the deterministic and stochastic displacements. Both pieces are joined to form an *Integrator* that can be used as usual. Here is an example of the Euler-Maruyama integration scheme being specialized with the Cholesky decomposition algorithm for the fluctuations.

```

#include <uammd.cuh>
#include <Integrator/BDHI/BDHI_EulerMaruyama.cuh>
#include <Integrator/BDHI/BDHI_Cholesky.cuh>
using namespace uammd;
//A function that creates and returns a BDHI
→ integrator
// using Cholesky decomposition for the noise
auto createIntegratorBDHICholesky(UAMMD sim){
    //A strategy is mixed with an integration scheme
    using Cholesky =
        → BDHI::EulerMaruyama<BDHI::Cholesky>;
    Cholesky::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    //For Cholesky the radius is optional.
    //If not selected, the module will use the
    → individual radius of each particle.
    //par.hydrodynamicRadius =
    → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
    auto bdhi = std::make_shared<Cholesky>(sim.pd,
    → par);
    return bdhi;
}

```

Source Code 20: Example code for the Cholesky **BDHI** method

19.2 LANCZOS

Fixman proposed a method based on Chebyshev polynomials [98] to compute the square root of the mobility. This method requires approximating the extremal eigenvalues of the mobility. Many strategies can be employed to find out these eigenvalues, with complexities ranging from $O(N^3)$ (thus beating the purpose) to $O(N^{2.25})$ [99]. More recently, a family of iterative algorithms based on Krylov subspace decompositions (using the Lanczos algorithm) have emerged [100, 101] showcasing algorithmic complexities in the order $O(kN^2)$, being k the number of required iterations (which is usually around the order of 10 depending on the desired tolerance). In **UAMMD** the technique developed in [100] is implemented.

Another benefit of this method over Cholesky is that it is not required to store the full mobility matrix in order to compute

the fluctuations. The product of the mobility tensor by a vector (the forces in the deterministic term and a random noise in the fluctuating one) can be computed by recomputing the necessary terms. This will be particularly useful later, when most elements in the mobility tensor become zero, reducing the complexity of the computation for both terms. In particular, **UAMMD**'s implementation of the Lanczos iterative method is templated for any object capable of providing the product of any given vector with the mobility matrix. In the current instance we use the NBody algorithm (chapter 9) coupled with a *Transverser* because the mobility is a dense matrix. However, in section 20.2, we only need to compute the square root of a sparse mobility matrix (because the pairwise mobility is short ranged) and thus we can couple the same *Transverser* with a neighbour list instead.

Use in UAMMD

Using the Lanczos strategy in **UAMMD** is similar to using Cholesky (see sec. 19.1). With the difference that now, being an iterative algorithm, a tolerance can be selected. Code 21 contains an example of the Euler-Maruyama integration scheme being specialized with the Lanczos iterative algorithm for the fluctuations.

```
#include <uammd.cuh>
#include <Integrator/BDHI/BDHI_EulerMaruyama.cuh>
#include <Integrator/BDHI/BDHI_Lanczos.cuh>
using namespace uammd;
//A function that creates and returns a BDHI
→ integrator
// using Lanczos decomposition for the noise
auto createIntegratorBDHILanczos(UAMMD sim){
    //A strategy is mixed with an integration scheme
    using Lanczos = BDHI::EulerMaruyama<BDHI::Lanczos>;
    Lanczos::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    //For Lanczos the radius is optional.
    //If not selected, the module will use the
    → individual radius of each particle.
    //par.hydrodynamicRadius =
    → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
```

```
//The tolerance for the stochastic term
→ computation
par.tolerance = sim.par.tolerance;
auto bdhi = std::make_shared<Lanczos>(sim.pd, par);
return bdhi;
}
```

Source Code 21: Example code for the Lanczos [BDHI](#) method.

20

TRIPLY PERIODIC HYDRODYNAMICS

It is often convenient to impose [Periodic Boundary Conditions \(PBC\)](#) in our complex fluids simulations. However, as the hydrodynamic interactions are long ranged in nature, special considerations are required to elaborate efficient algorithms. When discussing open boundary hydrodynamics in the past chapter the most efficient algorithm had a complexity of $O(N^2)$. Intuitively, it would seem that imposing [PBCs](#) would but make things even worse. Luckily, we can leverage the mathematical machinery laid out in chapter 18 to devise a family of triply periodic, Eulerian-Lagrangian, pseudo-spectral algorithms on regular meshes with complexities that will, in fact, scale linearly with the number of particles. In the following sections we will go through a bunch of GPU-aware algorithms that allow to compute hydrodynamic displacements (owing to the different terms in Eq. (17.7)) in triply periodic environments. In particular, we will solve Eq. (18.7) (or Eq. (18.1) in sec. 20.5) directly. As we saw in chapter 18, sampling the local fluid velocity to compute hydrodynamic displacements of each particle is equivalent to solving the particle's dynamics with Eq. (17.7) (or equivalently Eq. (18.20)).

All the algorithms we will learn about share some common themes, mainly: They are variations of the Immersed Boundary Method [102] that take the gross of the computation to Fourier-space in some way (pseudo-spectral algorithms). Many complex operations (differentiations, convolutions...) become simple algebraic ones in Fourier space. One of the main motivations to devise pseudo-spectral algorithms is the existence of the [FFT](#) algorithm which can take a signal, evaluated at equidistant points (i.e. a regular grid), to Fourier space in just $O(N \log(N))$ operations. It would be pointless to develop a spectral algorithm because it reduces the complexity of certain operations if we need $O(N^2)$ operations to transform a signal.

In order to numerically solve Eq. (18.7) (via the Green formalism, see Eq. (18.15)) in Fourier space we need to evaluate the fluid velocity on a grid. A grid poses an immediate challenge when we want to compute the displacements of a group of arbitrarily

located particles because we have to transform the forces acting on them to a smooth force density field defined in the same grid as the fluid. In chapter 18 we saw how this conversion is carried out by the spreading and interpolation operators via a smoothed delta function. We will describe how to efficiently apply the spreading and interpolation operators on a GPU in chapter 21.

20.1 FORCE COUPLING METHOD (FCM)

The Force Coupling Method (FCM) [103] is an Immersed-Boundary-like Eulerian-Lagrangian pseudo-spectral method initially devised for the computation of the hydrodynamic displacements of a colloidal suspension in a triply periodic environment. In future chapters, we will see how the FCM is quite general and can be applied outside of its originally intended applications.

Both *Cholesky* and *Lanczos* methods use the explicit form of an open boundary mobility. This makes them open boundary algorithms since they do not take into account the periodic images of the system in any way. Furthermore, the computational complexity of these methods is restrictive. Luckily, we can manage to do it in $O(N)$ operations if we consider periodic boundary conditions. In particular by solving Eq. (18.20) directly in Fourier space via the Force Coupling Method [103]. In doing so, we get the added benefit (and disadvantage) of not imposing a specific mobility tensor, which will arise naturally according to the convolution between the Green's function and the spreading kernel.

In order to do this we first spread particle forces to the fluid (SF in Eq. (18.20)) and transform them to Fourier space. Then the Green's function (like Eq. (18.15)) for periodic boundary conditions) is used to obtain the velocities in the fluid in Fourier space via multiplication. The fluid velocity is then transformed back to real space and Eq. (18.17) is applied to get the particle displacements by interpolation. The kernel used in the original description of the FCM is the Gaussian kernel in Eq. (21.14), in particular

$$\delta_a(r = \|\mathbf{q}_i - \mathbf{r}\|) := \frac{1}{(2\pi\sigma_a)^{3/2}} \exp\left(\frac{-r^2}{2\sigma_a^2}\right) \quad (20.1)$$

Where $\sigma_a = a/\sqrt{\pi}$. This naturally regularizes the Oseen tensor at short distances, yielding RPY-like behavior. In fact, the mobility

tensor for an unbounded three dimensional domain can be computed analytically in this case by solving the double convolution in Eq. (18.24). In particular, we can write the real space expressions for $f(r)$ and $g(r)$ in Eq. (18.27)

$$\begin{aligned} f(r) &= \frac{1}{8\pi\eta r} \left[\left(1 + 2 \frac{a^2}{\pi r^2} \right) \operatorname{erf} \left(\frac{r\sqrt{\pi}}{2a} \right) - 2 \frac{a}{\pi r} \exp \left(-\frac{\pi r^2}{4a^2} \right) \right] \\ g(r) &= \frac{1}{8\pi\eta r} \left[\left(1 - 6 \frac{a^2}{\pi r^2} \right) \operatorname{erf} \left(\frac{r\sqrt{\pi}}{2a} \right) + 6 \frac{a}{\pi r} \exp \left(-\frac{\pi r^2}{4a^2} \right) \right] \end{aligned} \quad (20.2)$$

The fluctuating term, $\nabla \cdot \mathcal{Z}$, is computed directly in Fourier space in the fluid via ik differentiation. This makes evaluating the Brownian motion effectively cost free. The computation of the fluid velocities is independent of the number of particles (since the first step is to spread the forces to the fluid). The complexity of the spreading and interpolation operations grow linearly with the number of particles. Thus the overall complexity of the algorithm is $O(N)$.

Summarizing, we can express the fluid velocity in Fourier space as

$$\hat{\mathbf{v}}(\mathbf{k}) = \eta^{-1} \hat{\mathbf{G}} (\mathbf{k} \cdot \hat{\mathcal{Z}} + \hat{\mathbf{f}}) \quad (20.3)$$

Where

$$\hat{\mathbf{G}} = B(k) \left(\mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \right) \quad (20.4)$$

is the fourier representation of the Stokes Green's function¹. Here $B(k)$ is a factor in Fourier space, i.e $B := \frac{1}{k^2}$ for the Oseen tensor in Eq. (18.15). Note that we have defined the divergence of the noise as $\mathbf{k} \cdot \hat{\mathcal{Z}}$, without the complex i . We can do this given the fact that if we define $\hat{\mathcal{Z}} := i\hat{\mathcal{Z}}_1$ (being $\hat{\mathcal{Z}}_1$ a different noise with the same properties as $\hat{\mathcal{Z}}$) we still get white noise with identical properties.

The **FCM** was originally developed for a Gaussian kernel, however, note that any smooth, closely-supported kernel can be used, which will regularize differently the near field hydrodynamics. In particular, any of the kernels described in sec. 21 are valid here.

¹ This separation is made here to emphasize the generality of this method for any Green's function as long as its Fourier transform is available.

Spatio-temporal discretization

Assuming a cubic box (the description can be then generalized to non-cubic boxes easily) we solve the velocity of the fluid on a grid with size h , with a number of cells in each size $N_c = L/h$. We use FFT to discretize the Fourier transform, this requires us to evaluate the properties of the fluid in a grid. This grid must be fine enough to correctly describe the Gaussian interpolator in Eq. (20.1) (similarly to the discrete description in sec. 24). On the other hand, the kernel in Eq. (20.1) has an infinite range and to make the overall spreading/interpolation have a constant cost for each particle (independent of the size of the domain) it is necessary to truncate it at a certain distance, r_c . In particular, the author in [103] suggests $r_c = m\sigma_a$, with $m = 3\sqrt{\pi} \approx 5.3$ being the number of standard deviations of the Gaussian that are taken into account (thus making $\delta_a(r > r_c) = 0$), and $\sigma_a/h > 1.86$, suggesting that the error is less than machine precision for $\sigma_a/h = 14.89$. We can then set a number of support cells for the kernel as $n_s = 2 \text{ ceil}(r_c/h) + 1$. We will later study these tolerance considerations.

The noise can be computed in Fourier space by generating Gaussian random numbers with zero mean and standard deviation given by

$$\langle \widehat{\mathcal{Z}}_{ik}(\mathbf{k}) \widehat{\mathcal{Z}}_{jm}^*(-\mathbf{k}) \rangle = \frac{2k_B T \eta N_c^3}{h^3 \delta t} (\delta_{ij} \delta_{km} + \delta_{im} \delta_{kj}) \quad (20.5)$$

Special care must be taken to enforce this condition, in particular by ensuring the uncoupled modes, also known as Nyquist points, are real (equal to their conjugate). Appendix B provides a summary of dealing with spectral methods in numerical implementations, including these details.

Making an analogy with Eq. (18.20), the whole process of going from forces acting on the particles to particle displacements can be summarized as follows:

Force Coupling Method

1. Spread particle forces to the grid: $\mathbf{f} = \mathcal{S}\mathbf{F}$
2. Transform fluid forcing to Fourier space: $\widehat{\mathbf{f}} = \mathfrak{F}\mathcal{S}\mathbf{F}$
3. Multiply by the Green's function to get $\eta^{-1} \widehat{\mathcal{G}} \mathfrak{F}\mathcal{S}\mathbf{F}$

4. Sum the stochastic forcing in Fourier space: $\hat{\mathbf{v}} = \eta^{-1} \hat{\mathcal{G}}(\mathfrak{F} \mathcal{S} \mathbf{F} + \mathbf{k} \hat{\mathcal{Z}})$
5. Transform back to real space: $\mathbf{v} = \eta^{-1} \mathfrak{F}^{-1} \hat{\mathcal{G}}(\mathfrak{F} \mathcal{S} \mathbf{F} + \mathbf{k} \hat{\mathcal{Z}})$
6. Interpolate grid velocities to particle positions: $\mathbf{u} = \mathcal{J} \mathbf{v}$

Here \mathfrak{F} represents the Fourier transform operator. Spreading and interpolation can be done with the algorithms and kernels described in chapter 21.

It is important to note that the steps above constitute quite a general protocol whose range of applicability extends far beyond triply periodic hydrodynamics by reinterpreting/bending the different terms and operators. As a matter of fact, we will employ a reimagination of the Force Coupling Method to compute electrostatics in chapter 24. In another instance, when discussing the [Inertial Coupling Method \(ICM\)](#) (chapter 20.5), we will see how the Green's function, \mathcal{G} , does not have to be analytic and can, in fact, be computed numerically. For the time being, in the following chapters we will describe several modifications to the [FCM](#) to compute hydrodynamics in different regimes.

Once the particle velocities are computed, the dynamics can be integrated using, for instance, the Euler-Maruyama scheme devised for [BD](#) in sec. 17². The update rule in the case of Euler-Maruyama is

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \mathbf{u}^n \delta t, \quad (20.6)$$

where the particle velocities already include the stochastic displacements.

Regarding torques

Torques acting on the particles can easily be introduced in the algorithm by adding them as an external force on the fluid. We can redefine the fluid forcing to include torques

$$\mathbf{f} = \mathcal{S} \mathbf{F} + \frac{1}{2} \nabla \times (\mathcal{S}_\tau \mathbf{T}) \quad (20.7)$$

² The arguments used to arrive at Eq. (18.21) can also be employed here to interpret $\mathbf{u} = \mathcal{J} \mathbf{v}$ as the equations of Brownian Dynamics.

Where \mathcal{S}_τ is the kernel used to spread the torques, \mathbf{T} . If a Gaussian is used (see Eq. (20.1)), its width is related to the hydrodynamic radius as $\sigma_\tau = a/(6\sqrt{\pi})^{1/3}$.

The curl of the spread torques can easily be computed in Fourier space, where it is transformed into a vector product

$$\hat{\mathbf{f}} = \mathfrak{F} \mathcal{S} \mathbf{F} + \frac{1}{2} i \mathbf{k} \times (\mathfrak{F} \mathcal{S}_\tau \mathbf{T}) \quad (20.8)$$

Similarly, once the fluid velocities are obtained in Fourier space, the angular velocities, $\boldsymbol{\omega}$ can be computed from the local fluid vorticity

$$\boldsymbol{\omega} = \frac{1}{2} \mathcal{J}_\tau \mathfrak{F}^{-1} (i \mathbf{k} \times \hat{\mathbf{v}}) \quad (20.9)$$

Use in UAMMD

Using the **FCM** strategy in **UAMMD** is similar to using Cholesky (see sec. 19.1). Being a non-exact algorithm³, a tolerance has to be specified and since now the domain is periodic, a domain size is also needed.

In contrast to the open boundary algorithms in sections 19.1 and 19.2 this implementation does not allow to set a different hydrodynamic radius for each particle. Here is an example of the Euler-Maruyama integration scheme being specialized with **FCM**.

```
#include <uammd.cuh>
#include <Integrator/BDHI/BDHI_EulerMaruyama.cuh>
#include <Integrator/BDHI/BDHI_FCM.cuh>
using namespace uammd;
//A function that creates and returns a BDHI
→ integrator
// using Force Coupling Method
auto createIntegratorBDHIFCM(UAMMD sim){
    //A strategy is mixed with an integration scheme
    using FCM = BDHI::EulerMaruyama<BDHI::FCM>;
    FCM::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    par.hydrodynamicRadius =
        → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
```

³ Due to the spatial discretization and the quadrature errors incurred when spreading/interpolating with a truncated kernel.

```
    par.tolerance = sim.par.tolerance;
    par.box = sim.par.box;
    auto bdhi = std::make_shared<FCM>(sim.pd, par);
    return bdhi;
}
```

Source Code 22: Usage example of the [FCM](#) module

20.2 POSITIVELY SPLIT EWALD (PSE)

In **FCM** the size of the grid is tied to the hydrodynamic radius of the particles. Hindering its ability to simulate either large domains or small particles. We can apply an Ewald splitting strategy to overcome this limitation as described in [94].

In particular, we will solve Eq. (17.7) using the **RPY** mobility with periodic boundary conditions. We can write the Stokes solution operator in Eq. (18.13) for a periodic system (with $\{\mathbf{k}\}$ allowed wavevectors) in real space using the same strategy as in [104]

$$\mathcal{L}(\mathbf{r}) = \frac{1}{V} \sum_{\mathbf{k}} \frac{\exp(i\mathbf{k}\mathbf{r})}{k^2} \left(\mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \right) \quad (20.10)$$

Where $V = L^3$ is the volume of the periodic domain.

We can now write the periodic **RPY** tensor, which amounts to replacing the integral in Eq. (18.25) by a sum [94]⁴.

$$\mathcal{M}_{ij}^{\text{RPY}}(\mathbf{r}) = \frac{1}{\eta V} \sum_{\mathbf{k}} \frac{\exp(i\mathbf{k}\mathbf{r})}{k^2} (\text{sinc}(ka))^2 \left(\mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \right) \quad (20.11)$$

In **Positively Split Ewald (PSE)**, the **RPY** tensor is expressed a sum of two symmetric, positive definite, operators. The first of them is spatially local and can be evaluated exactly using the algorithm in sec. 19.2 (with the added benefit that most elements in the mobility tensor will be zero). The second operator is non local and its contribution can be taken into account using the same spectral algorithm as in sec. 20.1. Thus we are looking for a mobility tensor in the form

$$\mathcal{M}_{ij}^{\text{RPY}} = \mathcal{M}_{ij}^{\text{far}} + \mathcal{M}_{ij}^{\text{near}} \quad (20.12)$$

We use the Ewald sum splitting of Hasimoto [104] for Eq. (20.11) so that

$$\mathcal{M}_{ij}^{\text{far}} = \frac{1}{\eta V} \sum_{\mathbf{k}} \frac{\exp(i\mathbf{k}\mathbf{r})}{k^2} (\text{sinc}(ka))^2 H(k, \xi) \left(\mathbb{I} - \frac{\mathbf{k} \otimes \mathbf{k}}{k^2} \right) \quad (20.13)$$

Where the Hasimoto splitting function is defined as

$$H(k, \xi) = \left(1 + \frac{k^2}{4\xi^2} \right) \exp \left(-\frac{k^2}{4\xi^2} \right) \quad (20.14)$$

⁴ The trick here is to take the open-boundaries **RPY** tensor and sum over the system images in Fourier space, which can then be interpreted as the definition of the discrete Fourier transform.

Here ξ is the splitting parameter (see sec. 24), an arbitrary parameter larger than zero that allows to shift the weight of the mobility from one term to the other. Thus, the far field contribution to the mobility decays rapidly in Fourier space.

On the other hand, the integral for the inverse transform of the near part can be computed analytically, this term can be written as

$$\mathcal{M}_{ij}^{\text{near}} = F(r, \xi) \mathbb{I} - G(r, \xi) \left(\frac{\mathbf{r} \otimes \mathbf{r}}{r^2} \right) \quad (20.15)$$

Where the functions F and G are two rapidly decaying functions with really convoluted expressions that can be found in Appendix A of [94] or in the UAMMD source code⁵. In essence, this means that the near field contribution to the mobility decays rapidly, i.e it is a compactly supported interaction.

It is worth mentioning that both contributions to the mobility in (20.15) and (20.13) are guaranteed to be positive definite for all particle configurations, which is the main contribution of the original work describing the PSE [94]. For our purposes this means that we can use the techniques we have already devised in previous sections to evaluate each term. The deterministic term (the multiplication of the mobility by the forces acting on the particles) can simply be computed separately (in the far and near fields) and then added.

The fluctuations pose more of a challenge if we want to refrain from computing and storing the square root of the mobility. Nonetheless, we can split the fluctuating contribution in Eq. (17.7) into a far and near field ones by defining

$$(\mathcal{M}^{\text{RPY}})^{1/2} d\widetilde{\mathbf{W}} := (\mathcal{M}^{\text{near}})^{1/2} d\widetilde{\mathbf{W}}_1 + (\mathcal{M}^{\text{far}})^{1/2} d\widetilde{\mathbf{W}}_2, \quad (20.16)$$

Where $\widetilde{\mathbf{W}}_{1,2}$ are independent Wiener processes. It can be easily proven (by studying the covariance of the noise) that this expression does indeed provide the correct fluctuation-dissipation balance with the mobility in Eq. (20.12), since both Wiener processes in the right hand side are uncorrelated.

Let's see in more detail how to adapt our previous algorithms to solve each part of the problem

⁵ These expressions are really long and convoluted. The corresponding code evaluating the near field in Eq. (20.15) is located in the UAMMD source file *RPY_PSE.cuh*, which is written to facilitate its copy-pasting for other implementations.

The far field

The far field computation is identical to the previously laid out **FCM** with a couple of trivial changes. First, the Green's function has to be modified, in particular, we need to redefine the Fourier factor in Eq. (20.3)

$$B(k) := \frac{1}{k^2 V} H(k, \xi) \exp\left(\frac{-k^2 \lambda}{4\xi^2}\right) \text{sinc}(ka)^2, \quad (20.17)$$

which describes the Green's function associated with the far field **RPY** mobility in Eq. (20.13). For the **FCM** based on a Gaussian kernel (see Eq. (20.1)) we have introduced a second splitting parameter, λ , to split the exponential in H (a common strategy [105] [106]). Thus, this corresponds to a Gaussian (**FCM**) kernel with variance $\sigma_a := \frac{\sqrt{\lambda}}{2\xi}$ (in real space). On the other hand, this restricts the far field spreading kernel to a Gaussian, without the possibility of using other more closely supported kernels easily, since the splitting requires knowledge of the Fourier transform of the kernel.

The cell size has to be chosen according to a certain cut-off wave number in Eq. (20.13). The authors in [107] give an upper bound for the truncation error of the fourier sum as $E_f \approx \exp(-k_{\text{cut}}^2/4\xi^2)$. Thus we can set E_f to be less than the tolerance, ϵ , and set

$$k_{\text{cut}} = 2\xi \sqrt{-\log(\epsilon)} \quad (20.18)$$

From here we can compute the necessary grid cell size as $h = \pi/k_{\text{cut}}$.

As evidenced by this new definition of σ_a , the spreading kernel is now uncoupled from the hydrodynamic radius. The width and support of the kernel can now be chosen to minimize quadrature and truncation errors. In particular the authors in [105] suggest choosing $m = C\sqrt{\pi n_s}$, where $C = 0.976$ is an empirical parameter. The number of standard deviations, m , can be computed by making the quadrature error, $E_q \approx \text{erfc}(m/\sqrt{2})$, less than a certain tolerance, $E_q < \epsilon$ (see sec. 3.2 in [105]). We can then set the splitting parameter as $\lambda = \left(\frac{hn_s \xi}{m}\right)^2$.

As demonstrated in section 18.1, there is a one-to-one relation between the purely Lagrangian Brownian Dynamics representation (via the mobility tensor) and the Green formalism used by the

FCM. Naturally, this equivalence also includes the fluctuations. We compute the far field fluctuating contribution via the stochastic stress tensor, \mathcal{Z} , (see, for instance, Eq. (20.3) and related discussion) which is already included in the **FCM** in a natural way. The treatment of the fluctuating stress tensor in **FCM** holds for any hydrodynamic Green's function as long as the divergence operator operator ($\nabla \cdot$) can be applied (which is the case in **PSE** since we are using triply periodic boundary conditions).

The near field

For the near field, we can use the infrastructure in the *Lanczos* algorithm (see sec. 19.2) with the mobility in Eq. (20.15). We can leverage here the fact that only local terms in the mobility matrix are non-zero (i.e. elements owing to close particles) to accelerate the matrix-vector multiplication. In **UAMMD**, this is achieved by using a neighbour list with a *Transverser* that takes the forces acting on the particles and computes the product with the mobility for each particle (see sec. E.1). The same *Transverser* is then used to compute the fluctuations via the *Lanczos* algorithm.

The functions F and G in Eq. (20.15) are evaluated in double precision to reduce numerical errors and then tabulated⁶. Both of them are truncated up to a certain distance, $r_{\text{cut}}^{\text{nf}}$. According to [105] the real space truncation error is bounded by $E_n \approx \exp(-\xi^2 r_{\text{cut}}^{\text{nf}})$. Thus, we can choose

$$r_{\text{cut}}^{\text{nf}} = \frac{\sqrt{-\log(\epsilon)}}{\xi} \quad (20.19)$$

Although the cost of computing the near field can sometimes negate the potential gains of using **PSE** over **FCM** (mostly due to the overhead of using an iterative Lanczos algorithm for the noise), it is important to note that if the near field cut off radius is less than the minimum distance allowed for a pair of particles this part of the algorithm can be skipped altogether (with the exception of the self terms in the mobility matrix). For instance, in the presence of some kind of steric repulsion (a la **LJ** or **WCA**).

Finally the optimal Hasimoto splitting parameter, ξ , must be tuned on a case by case basis. In general, lower values of ξ will work

⁶ **UAMMD** provides an infrastructre for tabulating functions called *Tabulated-Function*. Check its documentation for more information.

better for systems with low density (and the other way around). However, this will also depend on the specific implementation of the near and far field components besides particle configuration. In **UAMMD**, choosing $\xi a \approx 0.6$ is usually a good default.

Use in UAMMD

Usage is similar to **FCM**, with the difference that now a splitting parameter (ξ in Eq. (20.14)) can be selected to shift the weight of the algorithm between the near and far field. In particular, lower values ξa give more weight to the near field, while higher values give more importance to the far field.

```
#include <uammd.cuh>
#include <Integrator/BDHI/BDHI_EulerMaruyama.cuh>
#include <Integrator/BDHI/BDHI_PSE.cuh>
using namespace uammd;
//A function that creates and returns a BDHI
→ integrator
// using Positively Split Ewald
auto createIntegratorBDHIPSE(UAMMD sim){
    //A strategy is mixed with an integration scheme
    using PSE = BDHI::EulerMaruyama<BDHI::PSE>;
    PSE::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    par.hydrodynamicRadius =
        → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
    par.tolerance = sim.par.tolerance;
    par.box = sim.par.box;
    //The Ewald splitting parameter
    par.psi = sim.par.psi;
    auto bdhi = std::make_shared<PSE>(sim.pd, par);
    return bdhi;
}
```

Source Code 23: Example of the creation of the **PSE** module.

20.3 SPATIAL DISCRETIZATION WITH STAGGERED GRIDS

Thus far our spatial discretization has assumed a regular cartesian grid with all properties (fluid velocity and forcing) defined in the centers of the cell. This is known as a collocated grid. In non trivial geometries (such as in the presence of walls) the discretization of the different differential operators in Eq. (18.1) could lead to the projection operator, \mathcal{P} not being exactly idempotent, so that $\mathcal{P}^2 \neq \mathcal{P}$. This can lead to inaccuracies in the temporal discretization and ultimately to the discrete fluctuation-dissipation balance not being satisfied. Additionally, a collocated grid can lead to numerical artifacts in the presence of boundaries (such as walls) due to the fluid properties not being defined exactly on them and other aliasing issues that can affect high frequency phenomena [108, 109].

Furthermore, when studying transport phenomena in compressible fluids, collocated grids present spurious dependencies with the wave vector [110].

An alternative is to use a so-called MAC (Marker and cell), or staggered, grid [111].

Although it can be tricky to work with a staggered grid in a numerical implementation, the key is to interpret that each quantity is defined on a different grid, shifted $h/2$ with respect to the others (see Fig. 20.1). When we need to work on quantities defined on different grids, we simply use linear interpolation. From the point of view of the “main” grid (in which scalars are defined at the centers), vector quantities are defined on cell faces, while tensors are located at cell corners. If we need to multiply, for instance, a scalar, s , and a vector, \mathbf{v} , we interpolate the scalar at the same location as each coordinate of the vector. For example, in the x direction:

$$g_{i,j}^x = s_{i-1/2,j} v_{i,j}^x = \frac{1}{2}(s_{i,j} + s_{i-1,j}) v_{i,j}^x \quad (20.20)$$

Where i,j represent the indexes in the relevant “subgrid” (the one for the x direction in this case, marked as red in Fig. 20.1). We can use this to discretize the different operators in Eq. (18.13) via finite differences. The divergence of a vector is a scalar, defined in the centers of the main grid (black in Fig. 20.1)

$$(\nabla \cdot \mathbf{v})_i = \frac{1}{h} \sum_{\alpha} (v_{i+\hat{\alpha}}^{\alpha} - v_i^{\alpha}) \quad (20.21)$$

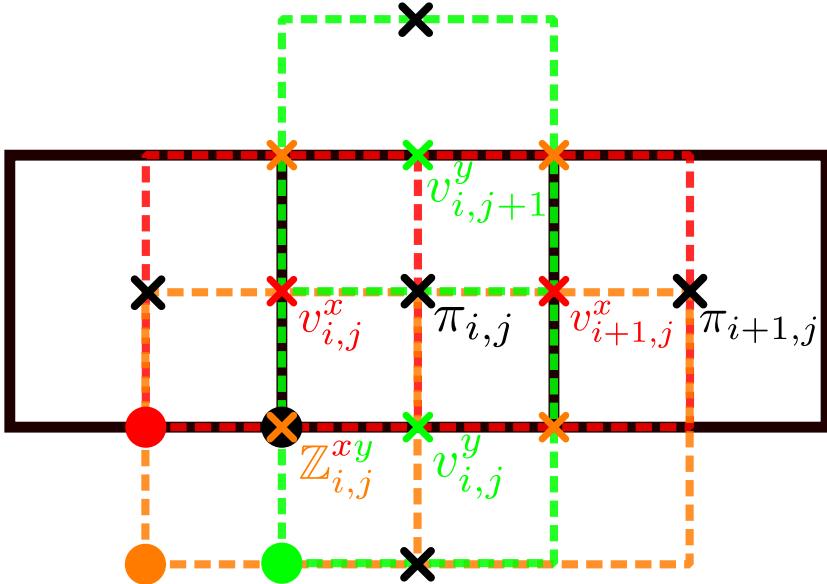


Figure 20.1: Representation of a staggered grid. Each quantity is defined on its own grid, with origin (circles) shifted by $h/2$ between them. Crosses mark cell centers in the various grids. The x coordinates of vectors are defined on the red grid, y coordinates on the green. Tensors are defined on the orange one and finally, scalars are defined on the black grid.

Where $\mathbf{i} := (i, j, k)$ and $\hat{\alpha} := (\delta_{x\alpha}, \delta_{y\alpha}, \delta_{z\alpha})$ is only non-zero in the direction α .

The gradient of a scalar is a vector, in which each component is defined on a different cell face

$$(\nabla s)_{\mathbf{i}}^{\alpha} = \frac{1}{h}(s_{\mathbf{i}} - s_{\mathbf{i}-\hat{\alpha}}) \quad (20.22)$$

Joining these two operators we can compute the Laplacian

$$(\nabla^2 \mathbf{v})_{\mathbf{i}}^{\alpha} = \nabla \cdot (\nabla v_{\mathbf{i}}^{\alpha}) = \frac{1}{h^2} \sum_{\beta} (v_{\mathbf{i}+\hat{\beta}}^{\alpha} - 2v_{\mathbf{i}}^{\alpha} + v_{\mathbf{i}-\hat{\beta}}^{\alpha}) \quad (20.23)$$

And the gradient of the divergence

$$[\nabla(\nabla \cdot \mathbf{v})]_{\mathbf{i}}^{\alpha} = \frac{1}{h^2} \sum_{\beta} (v_{\mathbf{i}+\hat{\beta}}^{\beta} - v_{\mathbf{i}}^{\beta} + v_{\mathbf{i}-\hat{\alpha}+\hat{\beta}}^{\beta} - v_{\mathbf{i}-\hat{\alpha}}^{\beta}) \quad (20.24)$$

Where we recall that each component of a vector is defined on its own grid (see Fig. 20.1) (so a component of a vector with index i is

defined at a different location from a scalar with the same index). The specific index notation here, assigning the same indexes (i, j) to the different quantities defined on the grid, aims to facilitate a computer implementation. With this notation everything can be stored as if it were defined on a collocated grid. Then, when applying the operators above, each element (vector or scalar) can be accessed with the same indexes as in these equations.

20.4 FLUCTUATING IMMERSED BOUNDARY (FIB) IN TRIPLY PERIODIC SYSTEMS

Recently, a new spatio-temporal solver for the Navier-Stokes equation has been presented, based on the [Immersed Boundary \(IBM\)](#) and similar to the [FCM](#) [87]. This framework, referred to as [Fluctuating Immersed Boundary \(FIB\)](#), is generic for any geometry (via direct discretization of the differential operators in Eq. (18.13)). However, here we will only take a few key ideas from it to implement another triply periodic solver for equation (18.7). In particular, we will make use of the new temporal integrators introduced in [87]. Another difference between this algorithm and [FCM](#) (as presented in sec. 20.1) is the use of a staggered grid (see sec. 20.3).

Using [PBC](#) greatly simplifies the algorithm in [87] to something very similar to the [FCM](#), since, as discussed in sec. 18 the Stokes operator has a straightforward form in Fourier space (see Eq. (18.15)).

Summarizing, for our purposes, we can describe the [FIB](#) as a variation of the [FCM](#) for staggered grids and a more sophisticated temporal integration algorithm. The authors of [79] developed the [FIB](#) for a three point Peskin kernel (see Eq. (21.7)) but, similarly to what was discussed in sec. 20.1, any of the kernels in sec. 21 can be used.

The discretization of the Stokes operator in Fourier space is straight-forward for a collocated grid by using the Fourier representation of the differential operators in Eq. (18.13) (a technique known as *ik* differentiation). However, for a staggered grid we have defined the differential operators using finite differences (see Eqs. (20.21) and (20.22)). It is then necessary to redefine the discretized Fourier operators. We can leverage a property of the Fourier transform for this; a shift in space results in a shift in phase in Fourier space. So from the Fourier transform of some scalar quantity (defined in the centers of the main grid in Fig.

[20.1](#)), we can compute the Fourier transform of the gradient by shifting the scalar to $x + h$ and then shifting the result back to $x + h/2$ (where vectors like the gradient are defined).

So by taking the Fourier transform of Eq. [\(20.22\)](#)

$$\widehat{\nabla s} = \frac{\exp(-i2\pi\mathbf{k}h/2)}{h} [\widehat{s}_{\mathbf{k}} - \exp(i2\pi\mathbf{k}h)\widehat{s}_{\mathbf{k}}] = \frac{2i}{h} \sin\left(\frac{\mathbf{k}h}{2}\right) \widehat{s}_{\mathbf{k}} = i\mathbf{k}_{\text{eff}} \widehat{s}_{\mathbf{k}} \quad (20.25)$$

Where the effective wave vector

$$\mathbf{k}_{\text{eff}} = \frac{2}{h} \sin\left(\frac{\mathbf{k}h}{2}\right) \quad (20.26)$$

Allows to make an equivalence with the discrete differential operators for a collocated grid.

Note that if the gradient of a vector, like the velocity, is to be computed this way, each element must first be shifted to the scalar grid (by applying a phase of $\pm h/2$ in Fourier space).

The divergence can be computed using a similar procedure:

$$\widehat{\nabla \cdot \mathbf{v}} = \frac{2i}{h} \sum_{\alpha} \sin\left(\frac{k_{\alpha}h}{2}\right) \widehat{v}_{\mathbf{k}}^{\alpha} = i \sum_{\alpha} k_{\text{eff}}^{\alpha} \widehat{v}_{\mathbf{k}}^{\alpha} \quad (20.27)$$

And finally the Laplacian:

$$\widehat{(\nabla^2 \mathbf{v})^{\alpha}} = \frac{-4}{h^2} \sin^2\left(\frac{k_{\alpha}h}{2}\right) \widehat{v}_{\mathbf{k}}^{\alpha} = -(k_{\text{eff}}^{\alpha})^2 \widehat{v}_{\mathbf{k}}^{\alpha} \quad (20.28)$$

For convenience, we can now write the discrete version of the projection operator in Eq. [\(18.11\)](#) in Fourier space for a staggered grid in a triply periodic system as

$$\widehat{\mathcal{P}}(\mathbf{k}) = \left(\mathbb{I} - \frac{\mathbf{k}_{\text{eff}} \otimes \mathbf{k}_{\text{eff}}}{k_{\text{eff}}^2} \right) \quad (20.29)$$

Temporal integration

The **FIB**, as presented in [\[87\]](#), comes with two new temporal integrator schemes particularly aimed at easing the discretization of thermal drift term. However, in our triply periodic implementation the thermal drift term is strictly zero. Nonetheless we will describe these integration schemes since they can come in handy in the future. On the other hand, even when the thermal drift is mathematically zero in the continuum limit it is possible that our

discretization causes some spurious drift, in which case it should also be included in our description. Taking into account that there are other sources of error it might be worth avoiding the extra work involved in the thermal drift computation (which amounts to two extra spreading operations).UAMMD's implementation acknowledges this by computing the thermal drift using random finite differences (as described in section 18.1) and providing a parameter to switch its inclusion on or off.

Simple predictor corrector

This first-order weak accurate midpoint scheme requires a single Stokes solve and interpolating fluid velocities twice (at the current time, $t = n$, and the mid point, $t = n + 1/2$).

$$\begin{aligned}\mathbf{v} &= \eta^{-1} \mathfrak{F}^{-1} \widehat{\nabla^{-2}} \widehat{\mathcal{P}} (\mathfrak{F} \mathcal{S}^n \mathbf{F}^n + \nabla \cdot \mathcal{Z}^n) \\ \mathbf{q}^{n+\frac{1}{2}} &= \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^n \mathbf{v} \\ \mathbf{q}^{n+1} &= \mathbf{q}^n + \delta t \mathcal{J}^{n+\frac{1}{2}} \mathbf{v}\end{aligned}\quad (20.30)$$

Note that we are again abusing the operator notation by writing the Fourier representation of the inverse of the Laplacian as $\widehat{\nabla}^{-2}$. Being the inverse Laplacian an operator that returns the solution to the equation $\nabla^2 \mathbf{x} = \mathbf{f}$. Since we are describing a triply periodic algorithm the Laplacian is self-adjoint and this equation has a trivial solution in Fourier space as $\widehat{v}^\alpha = (k_{\text{eff}}^\alpha)^{-2} \widehat{f}^\alpha$ (following the discrete definition of the Laplacian in a staggered grid from Eq. (20.28)).

Improved predictor corrector

We can achieve second-order accuracy in exchange for an additional Stokes solve by improving the estimation of the velocity at the midpoint. The update rule in this case is

$$\begin{aligned}\mathbf{v} &= \eta^{-1} \mathfrak{F}^{-1} \widehat{\nabla^{-2}} \widehat{\mathcal{P}} \mathfrak{F} (\mathcal{S}^n \mathbf{F}^n + \nabla \cdot \mathcal{Z}^{n_1}) \\ \mathbf{q}^{n+\frac{1}{2}} &= \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^n \mathbf{v} \\ \tilde{\mathbf{v}} &= \eta^{-1} \mathfrak{F}^{-1} \widehat{\nabla^{-2}} \widehat{\mathcal{P}} \mathfrak{F} \left[\mathcal{S}^{n+\frac{1}{2}} \mathbf{F}^{n+\frac{1}{2}} + \frac{1}{2} \nabla \cdot (\mathcal{Z}^{n_1} + \mathcal{Z}^{n_2}) \right] \\ \mathbf{q}^{n+1} &= \mathbf{q}^n + \delta t \mathcal{J}^{n+\frac{1}{2}} \tilde{\mathbf{v}}\end{aligned}\quad (20.31)$$

The divergence operator, whose application is required in real space to compute the noise term, is discretized according to the spatial discretization (see for instance Eq. (20.21) in a staggered grid). Similarly for the rest of the differential operators which are applied in Fourier space.

Since in a staggered grid each component of the velocity is defined on a different grid but with the exact same geometry there is no need for special considerations when Fourier transforming the fluid forcing or velocities using the **FFT** (see Appendix B).

Use in UAMMD

Usage of the **FIB** is reminiscent of other **BD Integrators** we have seen thus far. In the current case there is an additional parameter, **scheme**, which allows to choose between either of the two integration schemes described in the previous section.

```
#include <uammd.cuh>
#include <Integrator/BDHI/FIB.cuh>
using namespace uammd;
//A function that creates and returns an FIB
→ integrator
auto createIntegratorFIB(UAMMD sim){
    BDHI::FIB::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    par.hydrodynamicRadius =
        → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
    par.box = sim.par.box;
    //The integration scheme
    par.scheme = BDHI::FIB::IMPROVED_MIDPOINT;
    //par.scheme = BDHI::FIB::MIDPOINT;
    auto fib = std::make_shared<BDHI::FIB>(sim.pd,
        → par);
    return fib;
}
```

Source Code 24: Example of the creation of a *FIB Integrator* module.

20.5 INERTIAL COUPLING METHOD (ICM)

Thus far we have neglected the inertial terms in Eq. (18.1) and focused on the Stokes level (in what we call **BDHI**). In this last chapter about triply periodic hydrodynamics we will explore an algorithm, referred to as **ICM** [83], that allows to reintroduce inertia. In particular, we will consider both the temporal and convective inertial terms in Eq. (18.1).

Note, however, that we will not consider particle inertial terms coming from an excess mass of the particles, i.e. we consider neutrally buoyant particles with $m_e := m - \rho\Delta V = 0$. This implies that the particles still follow the local fluid velocity exactly and particles reach terminal velocity instantaneously. As demonstrated in [83] particle excess mass (inertial) effects can be reintroduced as a constraint via a Lagrange multiplier.

We will use the same staggered grid spatial discretization as in **FIB** (see chapter 20.3). Using the projection method described in chapter 18, we can write Eq. (18.1) as

$$\dot{\mathbf{v}} = \rho^{-1} \mathcal{P} (\mathbf{f} + \tilde{\mathbf{f}}). \quad (20.32)$$

Where we have introduced a new fluid forcing,

$$\mathbf{f} = -\rho \nabla \cdot (\mathbf{v} \otimes \mathbf{v}) + \eta \nabla^2 \mathbf{v}, \quad (20.33)$$

that includes the advective and diffusive terms to simplify the notation. We apply the projection operator in Fourier space, as we did in, for instance, sec. 20.1. Since we now have to solve the temporal variation of the velocity and we have non-linear terms, the diffusive and advective terms will be evaluated in real space. In the **ICM**, the divergence of the noise is also evaluated in real space.

We use a second-order accurate⁷ predictor-corrector scheme for temporal discretization. We can discretize the coupled fluid-particle equations as

$$\begin{aligned} \mathbf{q}^{n+\frac{1}{2}} &= \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^n \mathbf{v}^n, \\ \rho \frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\delta t} &= \mathcal{P} \left(\mathbf{f}^{n+\frac{1}{2}} + \tilde{\mathbf{f}}^{n+\frac{1}{2}} \right), \\ \mathbf{q}^{n+1} &= \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^{n+\frac{1}{2}} \left(\mathbf{v}^{n+1} + \mathbf{v}^n \right). \end{aligned} \quad (20.34)$$

⁷ A non-zero excess mass will denote the scheme to first-order accuracy.

Which requires evaluating the non-linear fluid forcing terms at mid step (i.e advection and diffusion). This discretization is similar to the midpoint predictor-corrector schemes in [FIB](#) but the convective term is discretized using a second order explicit Adams-Bashforth method (Eq. 35 in [83]),

$$\nabla \cdot (\mathbf{v} \otimes \mathbf{v})^{n+\frac{1}{2}} = \frac{3}{2} \nabla \cdot (\mathbf{v} \otimes \mathbf{v})^n - \frac{1}{2} \nabla \cdot (\mathbf{v} \otimes \mathbf{v})^{n-1}. \quad (20.35)$$

Advection is therefore stored each step to be reused in the next. The diffusive term is similarly discretized to second-order by

$$\nabla^2 \mathbf{v}^{n+\frac{1}{2}} = \frac{1}{2} \nabla^2 (\mathbf{v}^{n+1} + \mathbf{v}^n). \quad (20.36)$$

Replacing both equations into (20.34) and solving for the velocity at $n+1$ leads to the full form of the velocity solve, depending only on the velocity from previous time steps

$$\begin{aligned} \mathbf{v}^{n+1} = \tilde{\mathcal{P}} \mathbf{g}^n &= \tilde{\mathcal{P}} \left[\left(\frac{\rho}{\delta t} \mathbb{I} + \frac{\eta}{2} \nabla^2 \right) \mathbf{v}^n - \right. \\ &\quad \frac{3\delta t}{2} \nabla \cdot (\mathbf{v} \otimes \mathbf{v})^n - \frac{\delta t}{2} \nabla \cdot (\mathbf{v} \otimes \mathbf{v})^{n-1} + \\ &\quad \left. \mathcal{S} \mathbf{F}^{n+\frac{1}{2}} + \nabla \cdot \mathcal{Z}^n \right], \end{aligned} \quad (20.37)$$

where the modified projection operator is defined as

$$\tilde{\mathcal{P}} := \left(\frac{\rho}{\delta t} \mathbb{I} - \frac{\eta}{2} \nabla^2 \right)^{-1} \mathcal{P} \quad (20.38)$$

and is applied in Fourier space. The full algorithm can be summarized as follows:

1. Take particle positions to time $n + \frac{1}{2}$: $\mathbf{q}^{n+\frac{1}{2}} = \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^n \mathbf{v}^n$.
2. Spread forces on particles to the staggered grid: $\mathcal{S} \mathbf{F}^{n+\frac{1}{2}}$.
3. Compute and store advection: $\nabla \cdot (\mathbf{v} \otimes \mathbf{v})^n$.
4. Compute the rest of the terms in \mathbf{g} in Eq. (20.37), using the advective term just computed in addition to the one stored in the previous step.
5. Take \mathbf{g} to Fourier space and apply $\tilde{\mathcal{P}}$: $\hat{\mathbf{v}}^{n+1} = \tilde{\mathcal{P}} \hat{\mathbf{g}}$.
6. Take $\hat{\mathbf{v}}^{n+1}$ back to real space.

7. Evaluate particle positions at $n + 1$ by interpolating: $\mathbf{q}^{n+1} = \mathbf{q}^n + \frac{\delta t}{2} \mathcal{J}^{n+\frac{1}{2}} (\mathbf{v}^{n+1} + \mathbf{v}^n)$.

Since we are describing a triply periodic algorithm, we can use the discrete form of the differential operators for a staggered grid devised in previous sections (see sec. 20.3 and 20.4).

Use in UAMMD

Usage of the *ICM Integrator* requires a list of the familiar parameters for hydrodynamics thus far plus the fluid density, which for the first time plays a role.

```
#include <uammd.cuh>
#include <Integrator/Hydro/ICM.cuh>
//A function that creates and returns an ICM
→ integrator
auto createIntegratorICM(UAMMD sim){
    Hydro::ICM::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity; //Fluid
    → viscosity
    par.density = sim.par.density;    //Fluid density
    par.hydrodynamicRadius =
        → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
    par.box = Box(sim.par.L);
    return std::make_shared<Hydro::ICM>(sim.pd, par);
}
```

Source Code 25: Example of the creation of a *ICM Integrator* module.

All the algorithms laid out in this chapter make use of the spreading and interpolator operators first introduced in chapter 18.

For this matter, let us delve into the Eulerian-Lagrangian coupling found in the IBM and how to efficiently encode it in a GPU.

21

THE IMMERSED BOUNDARY METHOD (IBM) KERNELS

If we want to directly solve Eq. (18.20) instead of imposing a specific mobility (perhaps because we do not know the analytical form of the Green's function for a particular geometry), we need to discretize the spreading and interpolation operators in Eqs. (18.18) and (18.17). However, we cannot directly discretize deltas and we would like to take into account the size of the particles. We could smear the delta into a Gaussian, but the Gaussian kernels come with some inconveniences. In particular, their infinite range forces us to truncate them at a certain length. We would like to reduce the support of our kernels as much as possible without damaging accuracy. We can borrow some lessons from the IBM to help us in this regard.

The IBM [112] [102] is a mathematical framework for simulation of fluid-structure interaction. It is commonly used to discretize the spreading and interpolation operations we have seen in sec. 18¹. In IBM the forces acting on a certain marker (particle) are distributed (spread) to the nearby fluid grid points via some smeared delta function (see figure 21.1). IBM offers us a way to discretize the spreading (Eq. (18.18)) and interpolation (Eq. (18.17)) operators via some sophisticated δ_a kernels often referred to as Peskin kernels². Furthermore, the spreading and interpolation algorithms that we are going to devise for its implementation will also be applicable to other situations outside its intended purpose. For instance, to spread charges and interpolate electric fields when solving the Poisson equation (see sec. 24 or 25). In general, we can use the spreading and interpolation algorithms here for transforming between a Lagrangian (particles) and an Eulerian (grid) description.

We will discretize the interpolation operator in Eq. (18.17) on a grid as

¹ Which incidentally happen to be identical to the ones for electrostatics (sec. 24).

² In honor of their creator [112].

$$\mathcal{J}_{\mathbf{q}_i} \mathbf{v} = \sum_j \delta_a(\mathbf{q}_i - \mathbf{r}_j) \mathbf{v}_j dV(j) \quad (21.1)$$

Where the sum goes over all the cells, j , in the grid, with centers at \mathbf{r}_j . The function $dV(j)$ are the quadrature weights (i.e the volume of the cell). For a regular grid with cell size h , the quadrature weights are simply $dV(j) := h^3$, but we will see other geometries in chapter 25.

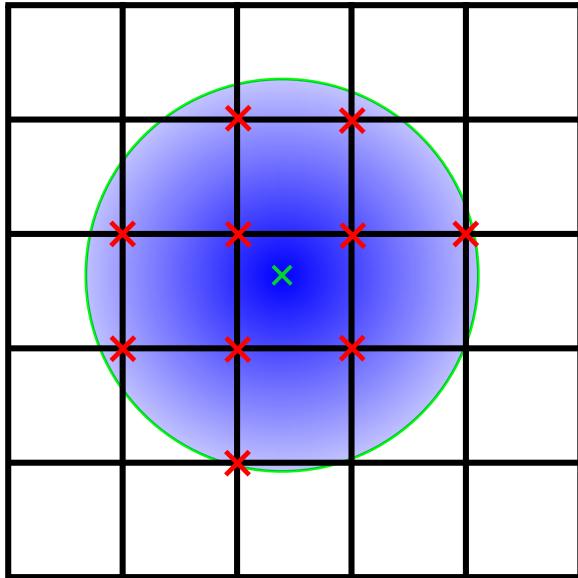


Figure 21.1: A representation of the Immersed Boundary. The blue circle represents a particle (with the green cross marking its center). Some quantity (i.e. the force) acting on it will be spread to the grid points inside its radius of action(red crosses).

A thorough description of the whole IBM framework can be found at [102], including the equations of motion for an arbitrary structure submerged in an incompressible fluid. However, we will only make use of the properties devised for the kernels.

In particular, Peskin kernels must abide by a series of postulates that intend to maximize computational efficiency (which translates to closer support) while minimizing the discretization effect of the grid (such as translational invariance).

For the sake of simplicity, the first postulate consists in assuming the kernel can be separated as

$$\delta_a(\mathbf{r} = (x, y, z)) = \frac{1}{h^3} \phi\left(\frac{x}{h}\right) \phi\left(\frac{y}{h}\right) \phi\left(\frac{z}{h}\right) \quad (21.2)$$

This allows to state the postulates regarding the one-dimensional function, ϕ . Additionally, this form yields $\delta_a \rightarrow \delta$ as $h \rightarrow 0$. The second postulate is that $\phi(r)$ must be continuous for $r \in \mathbb{R}$, avoiding jumps in the quantities spread to, or interpolated from, the grid. The close support postulate says that $\phi(r > r_c) = 0$, being r_c a cut off radius. This is our main means for seeking computational efficiency, since reducing the support of the kernel by one cell reduces dramatically the required operations. In particular, if the kernel has a support of n_s cells in each direction, spreading or interpolating requires visiting n_s^3 nearby cells, so a support of $n_s = 5$ requires 125 cells while a support of 3 requires just 27. Note that the support, n_s must be large enough to include all cells within r_c of the point to spread. In the case of a regular grid, this can be achieved by choosing $n_s \geq 2r_c/h + 1$.

The last basic postulate is required for the kernel to conserve the communicated quantities and it is simply a discrete expression of the fact that the kernel must integrate to unity.

$$\sum_j \phi(r - j) = 1 \text{ for } r \in \mathbb{R} \quad (21.3)$$

Where j are the centers or the cells inside the support. The next postulate intends to enforce the translational invariance of the distributed quantities as much as possible.

$$\sum_j (\phi(r - j))^2 = C \text{ for any } r \in \mathbb{R} \quad (21.4)$$

Where C is some constant to be determined. Eq. (21.4) is a weaker version of the condition for exact grid translational invariance

$$\sum_j \phi(r_1 - j) \phi(r_2 - j) = \Phi(r_1 - r_2) \quad (21.5)$$

Which states that the coupling between any two points must be a function of their distance. However, it can be shown that satisfying this condition is incompatible with a compact support [102]. Eq. (21.4) attempts to guarantee some degree of translational invariance by imposing a condition on the point with maximum coupling, $r_1 = r_2$.

Finally, we can impose conditions on the conservation of the first n moments to get increasingly higher order accuracy interpolants (at the expense of wider support)

$$\sum_j (r - j)^n \phi(r - j) = K_n \quad (21.6)$$

Where K_n are some constants. Note that the zeroth moment condition corresponds to Eq. (21.3) with $K_0 = 1$. By solving the system of equations given by these conditions, different kernels can be found.

3-point Peskin kernel

In particular, enforcing only the condition for the first moment (with $K_1 = 0$) we arrive at the so-called 3-point Peskin kernel.

$$\phi_{p3}(|r|) = \begin{cases} \frac{1}{3} \left(1 + \sqrt{1 - 3r^2} \right) & r < 0.5 \\ \frac{1}{6} \left(5 - 3r - \sqrt{1 - 3(1-r)^2} \right) & r < 1.5 \\ 0 & r > 1.5 \end{cases} \quad (21.7)$$

Where the argument $|r|$ represents the fact that the above expression must be evaluated for the absolute value of the separation (since the kernel is symmetrical).

4-point Peskin kernel

We can add a more restrictive condition on the integration to unity postulate

$$\sum_{j \text{ even}} \phi(r - j) = \sum_{j \text{ odd}} \phi(r - j) = \frac{1}{2} \quad (21.8)$$

Which smooths the contributions of the kernel when using a central difference discretization for the gradient operator. Solving for ϕ with this extra condition yields the classic 4-point Peskin kernel

$$\phi_{p4}(|r|) = \begin{cases} \frac{1}{8} \left(3 - 2r + \sqrt{1 + 4r(1-r)} \right) & r < 1 \\ \frac{1}{8} \left(5 - 2r - \sqrt{-7 + 12r - 4r^2} \right) & r < 2 \\ 0 & r > 2 \end{cases} \quad (21.9)$$

The main advantage of this kernel is that it interpolates linear functions exactly, and smooth functions are interpolated to second order accuracy.

6-point Peskin kernel

Recently, a new 6-point kernel has been developed that satisfies the moment conditions up to $n = 3$ for a special choice of K_2 [113]. Additionally, it also satisfies the even-odd condition in Eq. (21.8), it is three times differentiable and offers a really good translational invariance compared to similarly supported kernels.

This kernel sets $K_1 = K_3 = 0$ and

$$K_2 = \frac{59}{60} - \frac{\sqrt{29}}{20} \quad (21.10)$$

Solving for ϕ using these conditions, and defining the following

$$\alpha = 28$$

$$\begin{aligned} \beta(r) &= \frac{9}{4} - \frac{3}{2}(K_2 + r^2) + \left(\frac{22}{3} - 7K_2\right)r - \frac{7}{3}r^3 \\ \gamma(r) &= -\frac{11}{32}r^2 + \frac{3}{32}(2K_2 + r^2)r^2 + \frac{1}{72}\left((3K_2 - 1)r + r^3\right)^2 + \\ &\quad + \frac{1}{18}\left((4 - 3K_2)r - r^3\right)^2 \\ \chi(r) &= \frac{1}{2\alpha}\left(-\beta(r) + \text{sgn}\left(\frac{3}{2} - K_2\right)\sqrt{\beta(r)^2 - 4\alpha\gamma(r)}\right) \end{aligned} \quad (21.11)$$

We get the expression for the 6-point kernel

$$\phi_{p_6}(|r|) = \begin{cases} 2\chi(r) + \frac{5}{8} + \frac{1}{4}(K_2 + r^2) & r < 1 \\ -3\chi(r-1) + \frac{1}{4} - \frac{1}{6}\left((4 - 3K_2) + (r-1)^2\right)(r-1) & r < 2 \\ \chi(r-2) - \frac{1}{16} + \frac{1}{8}\left(K + (r-2)^2\right) - \\ \quad - \frac{1}{12}\left((3K_2 - 1) - (r-2)^2\right)(r-2) & r < 3 \\ 0 & r > 3 \end{cases} \quad (21.12)$$

Given its complexity it is advisable to tabulate ϕ_{p_6} .

Barnett-Magland (BM) kernel

A new kernel, called “exponential of the semicircle”(ES) and here referred to as BM, has been recently developed to improve the efficiency of non-uniform FFT methods [114]. This kernel has been used for electrostatics [115], but we will also apply it to the Stokes equation. This kernel has a simple mathematical expression

$$\phi_{BM}(r, \{\beta, w\}) = \begin{cases} \frac{1}{S} \exp \left[\beta(\sqrt{1 - (r/w)^2} - 1) \right] & |r|/w \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad (21.13)$$

Where β and w are parameters related to the shape and support (width) of the kernel. The parameter $S(\beta, w)$ is the necessary normalization to ensure that Eq. (21.13) integrates to unity. Since Eq. (21.13) does not have an analytic integral, this factor must be computed numerically. One advantage of BM kernel is that it decays faster than a Gaussian in Fourier space, which is beneficial in spectral methods [114].

One disadvantage of the kernels above is that we do not know their analytical Fourier transform (in the case of the BM kernel this stems from it not having an analytical integral), which hinders our ability to elaborate analytical expressions for the mobility in Eq. (18.24). For our purposes, this means that we have to numerically estimate the relation between the width of the kernels and the hydrodynamic radius, which will sometimes subject also to the size of the grid. On the other hand, all of them will present Oseen-like behavior at long distances and will be regularized in a similar way to the RPY mobility at short distances. We will investigate this in more detail shortly.

Gaussian kernel

Finally, we can include here for completeness the Gaussian kernel, which can be defined as

$$\phi_G(r, \{\sigma\}) = \frac{1}{(2\pi\sigma)^{3/2}} \exp \left(\frac{-r^2}{2\sigma^2} \right) \quad (21.14)$$

Where σ is the width of the Gaussian. In this case it is possible to compute the double convolution in Eq. (18.24) analytically, allowing to relate the hydrodynamic radius, a , with the width of the Gaussian as $\sigma := a/\sqrt{\pi}$. Other Peskin-like kernels can be

found by enforcing other conditions, see for example [116]. Section 21.2 presents a comparison of the different kernels showcased in this section.

21.1 SPREADING AND INTERPOLATION ALGORITHMS

In this chapter we will see how to efficiently spread and interpolate in a GPU. Let us consider that the communication will take place between a set of N particles (aka markers) with positions $\{\mathbf{q}_0, \dots, \mathbf{q}_{N-1}\}$ inside a cubic box of size L and a discrete regular grid with dimensions n in each direction (representing the field). Therefore, each cell in the grid is a cube of size $l := n/L$. The kernel is truncated at some distance r_c , which translates to a number of support cells $n_s := r_c/h$, ensuring that each particle only has to communicate with n_s^3 cells in its vicinity. Note that the algorithms described in the following sections can be generalized then to non-cubic boxes, non-regular grids, etc in a straightforward way.

For convenience we will assume that particles will spread forces, represented with $\mathbf{F} := \{\mathbf{F}_0, \dots, \mathbf{F}_{N-1}\}$, into a force density field, \mathbf{f}_j (where j represents a given cell), in the grid. Similarly particles will interpolate a velocity field, \mathbf{v}_j , from the grid into a set of per-particle velocities, $\mathbf{u} := \{\mathbf{u}_0, \dots, \mathbf{u}_{N-1}\}$. Naturally, the algorithms will be generic for any per-particle and per-cell quantity.

We want to devise algorithms for two purposes:

1. Spreading (as defined in Eq. (18.18)) for a discrete set of points:

$$\mathbf{f}_j = \mathcal{S}\mathbf{F} = \sum_i \delta_a(\mathbf{r}_j - \mathbf{q}_i)\mathbf{F}_i. \quad (21.15)$$

2. Interpolation (as defined in Eq. (21.16)) from a discrete set of points:

$$\mathbf{u}_i = \mathcal{J}_{\mathbf{q}_i}\mathbf{v} = \sum_j \delta_a(\mathbf{q}_i - \mathbf{r}_j)\mathbf{v}_j dV \quad (21.16)$$

Where, owing to the assumption of a regular grid, the quadrature weights, dV , are assumed to be the same for all cells in the grid.

In a regular grid, we can find the cell in which a given particle with position $q^{(\alpha)} \in [-L/2, L/2]$ (being α any direction) lies using

$c^{(\alpha)} = \text{floor}((q^{(\alpha)}/L + 0.5)N_\alpha)$, where N_α is the number of cells in the direction α . From there, it is straightforward to find the coordinates of the n_s^3 cells in the support of the particle, which are given by

$$\mathbf{c}_s \in [\mathbf{c}_q - \mathbf{P}, \mathbf{c}_q - \mathbf{P} + \mathbf{n}_s].$$

Where \mathbf{c}_q represents the cell in which a particle with position \mathbf{q} is located. The factor \mathbf{P} is defined, in each direction, as

$$P^{(\alpha)} := n_s/2 + s^{(\alpha)}$$

The shift, $s^{(\alpha)}$, has to be defined depending on whether the support, n_s , is even or odd (see Fig. 21.2). In particular, we have

$$s^{(\alpha)} := \begin{cases} 0 & \text{if } n_s \text{ is odd} \\ \text{floor}(q^{(\alpha)}/h - c_q^{(\alpha)} + 0.5) & \text{if } n_s \text{ is even} \end{cases} \quad (21.17)$$

Note that this rule for computing $s^{(\alpha)}$ only works for a regular grid. In general, we will need to find a rule that returns 0 if a particle is in one side with respect to the cell center and 1 otherwise.

It is important to note that if the support is even, the value of the factor \mathbf{P} in each direction will depend on the specific position of the particle inside the cell (see Fig. 21.2). On the other hand the definition of \mathbf{P} works seamlessly even if PBC are used, in that case we would simply need to fold the neighbour cells to the $[0, N]$ range (for instance, the rightmost neighbour cell in the x direction would be $c_j^{(x)} = \text{mod}(c_q^{(x)} + P, N_x)$).

The interaction between a cell j and a particle i is mediated via the kernel $\delta_a(\mathbf{q}_i - \mathbf{r}_j)$, in principle, this would incur n_s^3 evaluations of the kernel for spreading or interpolation. However, if the kernels are separable (so that $\delta_a(\mathbf{r}) = \phi_x(r_x)\phi_y(r_y)\phi_z(r_z)$ by the first postulate in Eq. (21.2)) we can reduce it to just $3n_s$ by precomputing and storing $\phi_X(q_x^i - r_x^j)$ (and similarly for the other directions) for each direction before spreading/interpolating each particle. We can then just multiply the stored evaluations of $\phi_{x/y/z}$ in each direction to get δ_a . Algorithm 7 shows how to take advantage of this optimization via two functions: The first one takes a position inside the unit box and its corresponding cell shift P and stores (in some arrays with an undetermined location) the $3n_s$ necessary kernel evaluations. The second function takes the 3D index of a neighbour cell ($[0, n_s]$ in each direction) and uses the stored kernel evaluations (which assumes the first function has been called already) to compute and return the weight for that neighbour cell.

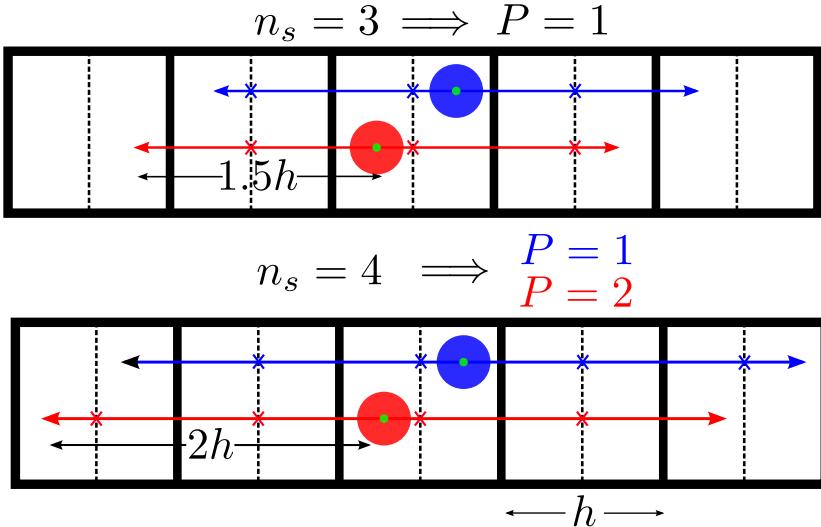


Figure 21.2: Two particles (red and blue circles, centered at the green circles) in a one dimensional grid ($N_x = 5, N_y = N_z = 1$). The kernel has a support of $n_s = 3$ cells (top) and 4 cells (bottom). Dashed lines indicate the middle of each cell and crosses represent the cells that need to be visited by the spreading or interpolation operations for each particle. If the support is even, depending on where a particle is inside the cell, some set of neighbours or another will be visited for spreading/interpolation. In this particular example, the shift, P , on the bottom (even) support case will be 2 for the red particle and 1 for the blue one.

Algorithm 7 Precomputing the kernel and accessing it via linear indexes. We used a similar strategy when discussing the cell list, see alg. 8.

```

1: function PRECOMPUTEKERNEL( $\mathbf{q} \in [0, L]_{\mathbb{R}^3}, \mathbf{P}_s$ )
2:    $\mathbf{c} \leftarrow$  cell of  $\mathbf{q}$ 
3:   for  $i = 0$  until  $n_s$  do
4:      $\mathbf{c}_s \leftarrow \mathbf{c} - \mathbf{P}_s + (i, i, i)$ 
5:      $\mathbf{r}_s \leftarrow$  center of  $\mathbf{c}_s$   $\triangleright$  Center position of cell  $\mathbf{c}_s$  in each
       direction
6:      $\mathbf{r} \leftarrow \text{dist}(\mathbf{r}_s, \mathbf{q})$   $\triangleright$  Distance to cell  $\mathbf{c}_s$  in each direction
7:     Store  $\phi_X[i] \leftarrow \phi(\mathbf{r}_x/h)/h$ 
8:     Store  $\phi_Y[i] \leftarrow \phi(\mathbf{r}_y/h)/h$ 
9:     Store  $\phi_Z[i] \leftarrow \phi(\mathbf{r}_z/h)/h$ 
10:   end for
11: end function
12: function FETCHKERNEL( $i_x \in [0, n_s], i_y \in [0, n_s], i_z \in [0, n_s]$ )
13:   return  $\phi_X[i_x]\phi_Y[i_y]\phi_Z[i_z]$ 
14: end function

```

In a serial implementation of spreading, we can simply sum the contribution, $\delta_a(\mathbf{q}_i - \mathbf{r}_j)\mathbf{F}_i$, of each particle, i , to each of the cells, j , in its support by looping. In the case of interpolation a similar strategy can be employed, where each particle accumulates the quantity $\delta_a(\mathbf{q}_i - \mathbf{r}_j)\mathbf{v}$ from its nearby cells. Algorithm 8 summarizes the strategy to spread or interpolate a given particle thus far.

Algorithm 8 Visiting the neighbour cells of a particle, i , located at $\mathbf{q}_i \in [0, L]_{\mathbb{R}^3}$ for spreading or interpolation. We define $\mathbf{u} := (1, 1, 1)$ for convenience.

```

1:  $\mathbf{c}_i \leftarrow$  cell of particle  $i$ 
2: if  $n_s$  is even then
   ▷ Shift for even supports. Can be 0 or 1 in each direction
3:    $\mathbf{s} \leftarrow \text{floor}(\mathbf{q}_i/h - \mathbf{c}_i + 0.5\mathbf{u})$       ▷ Valid for regular grids
4: else
5:    $\mathbf{s} \leftarrow (0, 0, 0)$ 
6: end if
7:  $\mathbf{P}_s \leftarrow n_s/2\mathbf{u} - \mathbf{s}$ 
8: precomputeKernel( $\mathbf{q}_i, \mathbf{P}_s$ )                                ▷ See alg. 7
9: for  $j = 0$  until  $n_s^3$  do
10:    $i_x \leftarrow \text{mod}(j, n_s)$ 
11:    $i_y \leftarrow \text{mod}(j/n_s, n_s)$ 
12:    $i_z \leftarrow j/n_s^2$ 
13:    $\delta \leftarrow \text{fetchKernel}(i_x, i_y, i_z)$       ▷  $\delta_a(\mathbf{q}_i - \mathbf{r}_j)$ , see alg. 7
14:    $\mathbf{c}_j \leftarrow \text{fold}(\mathbf{c}_i + (i_x, i_y, i_z) - \mathbf{P}_s)$  ▷ Neighbour cell  $\in [0, \mathbf{n}]$ .
15:   if Spreading then
16:     cellQuantity[ $\mathbf{c}_j$ ]  $\leftarrow= dV\delta$  particleQuantity[ $\mathbf{q}_i$ ]
17:   else if Interpolating then
18:     particleQuantity[ $\mathbf{p}_i$ ]  $\leftarrow= \delta$  cellQuantity[ $\mathbf{c}_j$ ]
19:   end if
20: end for
```

In a serial version, we can simply apply algorithm 8 to each particle to get a fairly efficient algorithm. However, a direct parallelization of this approach by assigning a worker to each particle is not possible for spreading. In this case two particles could write at the same time to the same cell, resulting in a race condition. This race condition can be circumvented by ensuring that line 16 in alg. 8 is an *atomic* operation³.

3 In a parallel environment, an *atomic* operation enforces that only one thread performs the operation a time. Allowing several workers (threads) to modify

Another solution is to traverse per cell instead of per particle, usually referred to as mesh-based approaches. Each cell is assigned to a worker, which then traverses all nearby particles. This requires to construct a cell list. While this approach is free of atomic operations (which can hurt performance if lots of collisions happen) it makes the algorithm scale with the number of cells and it will be inefficient in low density systems (or in configurations with large density disparities), where many cells are empty. On the other hand, CUDA atomic operations have come a long way, presenting almost no overhead when there are no collisions. Furthermore, a per-cell approach does not allow to take advantage of Eq. (21.2) easily, increasing the number of kernel evaluations to n_s^3 . We usually have to spread quasi uniform configurations in 3D and we typically deal with a few million number of particles, which makes a per-particle approach the overall best option. Note, however, that there is not a best-for-all algorithm regarding spreading and a mesh based approach will probably have the upper-hand in some situations (probably when highly dense systems are considered). For our typical use, the few corner cases in which the per-cell approach is more optimal are not worth the effort to consider it as an option. A review of several mesh-based methods is available at [117].

It is worth mentioning the specific geometry of the loop in line 9 of algorithm 8. When looping through the cells inside the support of a particle, using the strategy in algorithm 8 instead of the naive three nested loops results in more integer operations, but reduces the operation to a single loop, reducing conditional logic pressure and facilitating loop unrolling, also giving the compiler more chances for optimization. Furthermore, the traversal order can be tweaked easily by modifying the relation between the linear neighbour index and the cell offsets in lines 10 and below from algorithm 8.

(for instance by reading, modifying and then writing) the same memory location at the same time can result in some of the information being lost (for instance when a second thread reads while the first is still modifying). CUDA comes with a great GPU atomic library, which has become more and more efficient with time. CUDA atomics have come to a point in which, in the absence of collisions (two threads trying to access the same memory location), the overhead of using atomics is almost negligible.

Spreading algorithm

Regarding GPU friendly parallelizations of particle-based spreading algorithms, we will consider three main strategies that we will refer to as Block Per Domain (BPD), Block Per Particle (BPP) and Thread Per Particle (TPP). After describing the different algorithms we will compare their performance.

Thread Per Particle (TPP)

Let us start with the *naive* version of the algorithm in which we assign a thread to each particle and then simply apply algorithm 8, summing the particles contributions atomically to the grid data as previously discussed. Each thread must then evaluate the kernel as many times as needed for its assigned particle. We deal with the kernel evaluations in two different ways:

1. **TPP (register):** Store the $3n_s$ kernel evaluations (making use of the optimization in Algorithm 7) in each thread's *register* memory. Given the scarcity of register memory, storing $3n_s$ values in it could easily result in register spilling⁴. This algorithm is referred to as *GM* in [118].
2. **TPP (recompute):** We do not take advantage of the kernel separation at all and simply recompute the kernel whenever needed (in particular n_s^3 times). This naturally increases arithmetic pressure compared with the register version. However, as a general rule, GPUs are much more forgiving to extra arithmetic operations than they are to extra memory traffic. Thus exchanging memory for arithmetics is usually a good call.

While assigning a thread to a particle sounds like a reasonable separation of tasks, in doing so we are missing out on the intra-thread-block collaborative capabilities of the GPU (such as the shared memory space, see sec. 1.2). Furthermore, in TPP threads in the same block are dealing with different particles, so there is no mechanism in place reducing the chance of atomic collisions. If the particles assigned to two adjacent threads happen to be close

⁴ In a given CUDA kernel, when more register memory than available is requested, the compiler stores the excess in global memory. This is known as *register spilling*. Naturally, register spilling should be avoided at all costs, since accessing global memory is orders of magnitude slower than register memory.

in space, there will be a high chance that at some point these two threads try to write to the same cells in the grid. Luckily there is a yet more fine-grained parallelization hidden in this algorithm that happens to be perfectly suited for the GPU architecture. This parallelization emerges when assigning an entire thread block to a particle as we will discuss now.

Block Per Particle (BPP)

We suggest a particle-based approach, presenting subtle differences with the existing ones [118] [117] [94], specifically tailored for the GPU. We assign a thread block to each particle and then make each thread in the block spread to a different cell atomically. Since all threads in the block handle the same particle, all the per-particle information can be stored in shared memory. This includes the $3n_s$ evaluations of the kernel, which can be precomputed into shared memory collaboratively by all threads in the same block (see algorithm 9, a slight modification to algorithm 7 that takes this parallelization into account). The available shared memory is much larger than register memory so in this instance storing the $3n_s$ evaluations is not an issue. We refer to this method as block-per-particle (BPP)⁵.

In BPP, only one of the threads (typically the first one) needs to fetch the particle position and quantity to spread and compute its information (lines 1 to 7 in alg. 8). One benefit of assigning a block per particle is that it is guaranteed that no atomic collisions will occur for threads in the same block. Since the execution order of blocks is not guaranteed the chances of atomic collisions between blocks are reduced on average. In a thread-per-particle approach, if the particles are sorted randomly in memory chances of atomic collision are reduced at the expense of a worse access pattern (both when reading particles and writing to cells).

We might also consider sorting particles in memory by spatial hash (so that close in space also means close in memory) which results in faster fetching of the grid data, but increased chance of atomic overlap⁶. In general, there will be a trade-off between these two effects.

5 Changing algorithm 8 into a block-per-particle geometry amounts to assigning different elements of the cell loop to different threads, in the same way as in passing from algorithm 7 to 9.

6 In UAMMD, particles can be sorted using the Z-Morton hash (see chapter 11.2) easily.

Algorithm 9 Precomputing the kernel in GPU shared memory collaboratively. This algorithm is quite similar to alg. 7, but each element in the loop is assigned to a different thread in the block. The arrays $\phi_{X/Y/Z}$ are located in shared memory.

```

1: function PRECOMPUTEKERNELSM( $\mathbf{q} \in [0, L]_{\mathbb{R}^3}$ ,  $\mathbf{P}_s$ )
2:    $\mathbf{c} \leftarrow$  cell of  $\mathbf{q}$ 
3:   for  $i = \text{threadIdx.x}$  until  $n_s$  by  $\text{blockIdx.x}$  do
4:      $\mathbf{c}_s \leftarrow \mathbf{c} - \mathbf{P}_s + (i, i, i)$ 
5:      $\mathbf{r}_s \leftarrow \text{center of } \mathbf{c}_s$      $\triangleright$  Center position of cell  $\mathbf{c}_s$  in each
       direction
6:      $\mathbf{r} \leftarrow \text{dist}(\mathbf{r}_s, \mathbf{q})$      $\triangleright$  Distance to cell  $\mathbf{c}_s$  in each direction
7:     Store  $\phi_X[i] \leftarrow \phi(\mathbf{r}_x/h)/h$ 
8:     Store  $\phi_Y[i] \leftarrow \phi(\mathbf{r}_y/h)/h$ 
9:     Store  $\phi_Z[i] \leftarrow \phi(\mathbf{r}_z/h)/h$ 
10:   end for
11:    $\_\_\_ \text{syncthreads}()$ 
12: end function
```

However, our block-per-particle algorithm hides these issues (effect of particle sorting and atomic overlap) to an extent, since only one thread in each block will fetch a particle and atomic overlap is reduced by design.

Still, testing suggests that particle sorting via spatial hashing dramatically improves performance (an overall speedup of $\times 7$ for TPP and $\times 2$ for BPP). Suggesting that atomic overlap is far less important than a cache-friendly memory access pattern⁷. The storage layout of the grid data can also influence the performance of the algorithm. In particular, using a space-filling curve (as the one in sec. 11.2) could improve the access pattern in both spreading and interpolation [118]. However, this has not been considered in this work, where a linear index is always used (so that the data for cell with indexes (i, j, k) is located at element $i + (j + kn_y)n_x$ in the relevant array)⁸. This ordering comes imposed by the usual relation between spreading/interpolation and spectral methods in fluctuating hydrodynamics, which requires using FFT libraries that benefit from (or are restricted to) it.

⁷ Which goes to show how far have CUDA atomics have come.

⁸ Note, however, that the software interface for this module allows for an arbitrary ordering of the grid data, as we will later see.

Block Per Domain (BPD)

Recently, a new **GPU**-friendly algorithm has been developed [118] that uses a hybrid strategy, lying in between particle and grid based approaches, by breaking the domain into subdomains (called subproblems) and assigning each one to a thread block. Subproblems with more than a certain number of particles are broken again until each thread block deals with a subproblem with up to a maximum number of particles (see Fig. 1 in [118]) to aide with load-balancing. Subproblems are then spread into shared memory copies of the subdomains, which are then reduced and added atomically into global memory. Since overlap between subproblems is reduced to a shell of ghost cells atomic collisions are minimized. One particular advantage of this method is that its generalization to multi-gpu via domain decomposition comes naturally, as the algorithm already works by subdividing the domain. Testing suggest that, for a single **GPU**, the new strategy in [118] is only worth it for large and dense systems (with more than 2 million particles) and 2D grids (see Fig. 21.3). Although it is arguably the best algorithm at the time of writing if one wants to take advantage of several GPUs. We refer to this algorithm as Block Per Domain (BPD), although the authors of [118] refer to it as *SM* (from shared memory). It is worth noting that the authors of the BPD algorithm do take into account the previously introduced separable-kernel optimization.

Performance comparison of the spreading algorithms

It is worth mentioning that, although particle sorting usually takes a negligible amount of the runtime, in the **UAMMD** implementation sorting is carried out by *ParticleData* independently of whether spreading/interpolation is present (since sorting is beneficial in several other sections of the codebase, like neighbour traversing). In this sense, the overhead of sorting is hidden. Nonetheless, for the sake of fairness, sorting time is included as part of the runtime in the performance figures when applicable.

Finally, the **UAMMD** implementation presents another benefit when compared to others seen in the wild. In particular when referring to spreading/interpolating several quantities at once (such as forces/velocities). The general strategy when dealing with vectorial quantities (or simply several scalars) is to run the algorithm as

many times as there are components. This is related to the AoS (array-of-structures) vs SoA (structure-of-arrays) debate. In general terms, it is best to take an SoA approach in the GPU, which calls for the usual spread-many-times strategy. However, there is one crucial caveat to this rule worth taking into account: In complex fluid simulations we usually deal with vectorial quantities with 3 or 4 elements (i.e. positions, forces+energies...), which happen to be special in NVIDIA GPUs⁹. In particular, CUDA presents special load/store instructions that fetch/write 128 bits from memory (the space for 4 floats). In SoA we would store and treat each component separately, so that reading the (for instance) position of a particle would take 3 instructions. If we store the positions in an array composed by groups of 4 floats¹⁰ (an AoS approach) we only need a single instruction to load all three coordinates (and we get a fourth number to leverage for free). Using vector types can potentially make the cost of spreading/interpolation 4 values at the same time much cheaper than repeating the operation 4 times. **UAMMD**'s spreading/interpolation implementation interface is templated for any random access iterator¹¹ containing the per-particle and per-cell data as long as the type of the elements follows a certain set of sane rules (in particular, that the values of the iterator can be added together and that the per-particle type can be converted to the per-cell type when required¹²).

We compare the TPP and BPP algorithms with the hybrid BPD (or *SM* in [118]) in figures 21.3 and 21.4, measuring the performance of spreading one scalar per particle for a uniform distribution of positions at different densities (an average of 1 (left) and 0.1 (right) particles per cell). The Gaussian kernel is used in Fig. 21.3 for TPP and BPP, while [118] uses the *BM* kernel. However, as the kernel is precomputed with $3n_s$ evaluations which are then used n_s^3 times, the cost of evaluating the kernel is negligible in any case. In the case of recomputing the kernel our implementation could have an unfair advantage here, but since it happens to be the slower option in general, we will not consider it.

⁹ And probably in the GPU architecture in general.

¹⁰ A series of vectorial types names *realX* (with X being 2,3 or 4) are available in **UAMMD**. For instance, the type *real4* containing 4 scalars.

¹¹ See Appendix A for more information on C++ iterators.

¹² **UAMMD**'s vector types come with a large set of algebraic overloads. In general, any operation that can be used in a scalar also works for vector types in a per-element manner. For instance, the result of multiplying two *real4* values *a* and *b* will be, $c = (a_x b_x, a_y b_y, a_z b_z, a_w b_w)$.

In any case, if the kernel evaluation becomes a problem due to it being expensive, it is always a possibility to tabulate the kernel in a global memory array¹³. By doing so, the cost of computing the kernel becomes constant regardless of its mathematical expression, amounting to fetching some value from memory.

The following remarks about performance trends should be taken with a grain of salt, since with time the GPU architecture tends to become more and more forgiving about things like the memory access pattern, atomic operations, etc. The compiler also grows smarter with time, being able to better optimize the code. However, it is worth mentioning here at least some of the trends that will probably hold. Performance measurements were carried out using a NVIDIA RTX 2080Ti GPU, which has the *sm_75* architecture.

The TPP strategies particularly benefit from the AoS approach, where testing suggest that spreading 4 numbers at once is overall $5\times$ faster than spreading a single number. Alas, the overall more performant BPP strategy does not seem to benefit from this particular optimization and, in fact, the cost of spreading 4 numbers in an AoS manner is slightly higher than spreading 4 times separately.

Spatial hashing and sorting dramatically improves performance for all of our algorithms¹⁴ for medium to large problem sizes (for BPP, the performance crossover between unsorted and sorted happens at around $n_x = n_y = n_z = 100$ regardless of density). For small sizes, we elucidate the increased atomic overlap caused by sorting hurts the internal atomic handling system in the particular architecture at hand. Furthermore, the performance of the sorting algorithm (radix sort) increases with the number of particles. Figure 21.4 presents the same situation as figure 21.3 but with particle pre-sorting.

It is worth mentioning that, in the case of BPP, the optimal number of threads per block will in general depend on the number of support cells (512 cells in total for the showcased examples), besides other hardware considerations. For the data presented in figures 21.3 and 21.4 the optimal block size was 32, the minimum meaningful block size (since a warp, the minimum hardware com-

¹³ The UAMMD utility *TabulatedFunction* can be employed here. This tool mimics the old CUDA textures, allowing to tabulate a function inside a range and then interpolate it at any point in between.

¹⁴ We observe a speedup of up to $10\times$ for BPP and as high as $400\times$ for TPP.

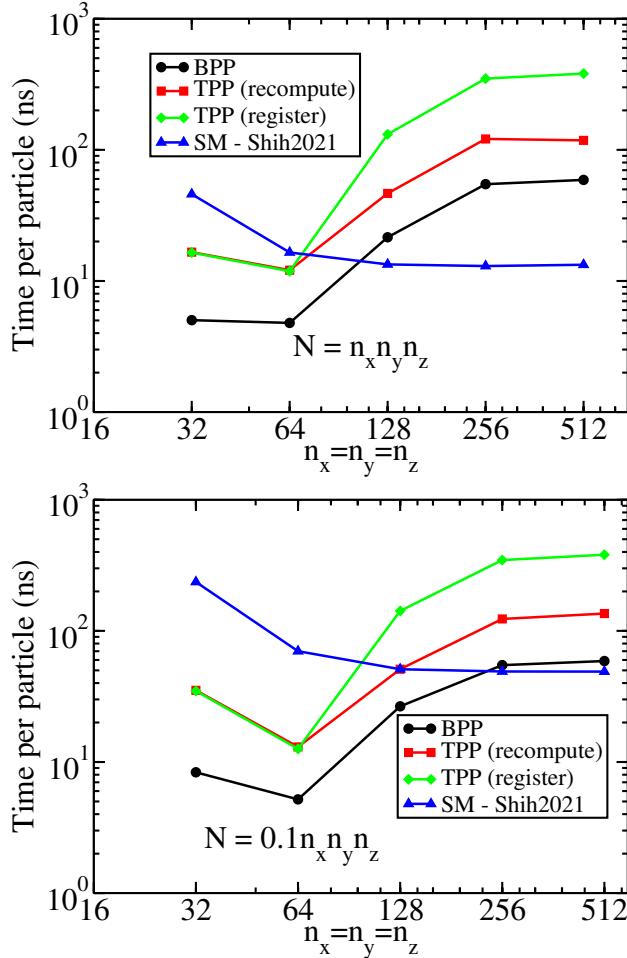


Figure 21.3: Performance comparison between the different spreading schemes considered for two different densities. Particles are distributed uniformly on a cubic grid and spread with a support $n_s = 8$ using a Gaussian kernel. A single scalar is spread per particle. The considered algorithms are: A block-per-particle (BPP), the algorithm used in UAMMD. A thread-per-particle (TPP) precomputing the kernel to register memory (register, similar to *GM-sort* in Shih2021 [118]) and recomputing it (recompute). Finally, timing for the BDP algorithm (*SM* in [118]) is also presented. For these tests, particle sorting was not employed, i.e. the particles are randomly distributed in memory with respect to their physical positions. Performance data was gathered in an RTX2080Ti GPU using single precision. The crossover between BPP and SM happens at around 2 million particles in both cases.

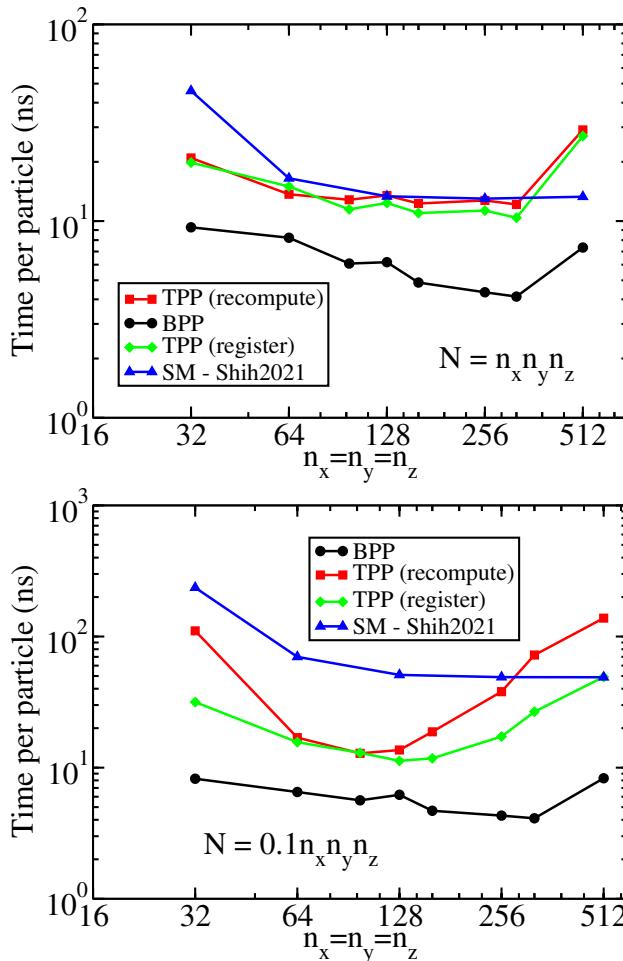


Figure 21.4: The same situation as in Fig. 21.3, but in this case particles are presorted to increase spatial locality in memory (particles close in space are placed close in memory). Both the positions and the spread quantity are sorted. Sorting is carried out using the same Morton hash technique laid out in chapter 11.2. Sorting takes a negligible amount of time and, moreover, it does not need to be done every time spreading occurs. Typically particles are sorted at the start of a given simulation and then every few diffusive times.

puting unit in CUDA GPUs, is composed of 32 threads). Finally, if sorting is used, the performance of BPP appears to be robust to changes in density. Whereas the TPP will typically be more performant with a higher density. Without sorting, neither of the algorithms seems to be affected by density.

According to the carried out tests, the overall better choice is therefore the BPP algorithm with particle sorting. Sizes beyond 512 cells per direction (in a cubic domain) start to suffer from memory issues ($n = 512$ with density 1 takes up at least 4GB of memory) and the sheer number of particles involved, $N \sim 10^8$ for density 1, render them impractical for most purposes.

Interpolation algorithm

Interpolation is inherently a more efficient operation than spreading, since no atomic operations are required and, moreover, we only need to read the grid data (see alg. 8). The worker assigned to a particle can accumulate the result of the interaction with nearby particles into a local variable and then write it to the global result array. Furthermore, interpolation requires only N write operations, while spreading requires to perform $n_s^3 N$ atomic writes. The same TPP vs BPP arguments from spreading can be broadly employed here, prompting a BPP approach. In this case, each thread in the block assigned to a given particle will compute the interaction with one (or several) cells, storing the result in a thread local register. When all threads in the block have finished, the local results are reduced to a final result, which is then written to global memory by a single thread (usually the first one). A naive block reduction involves the first thread fetching the local values from the rest of the threads (via shared memory or with CUDA shuffle intrinsics) and then performing a simple addition. However, using *cub*'s [49] *BlockReduce* utility proves to be a vastly superior alternative¹⁵.

Figure 21.5 shows the performance of BPP interpolation in the same situation as Fig. 21.3. Timings present a good linearity with the number of particles and are almost independent of the density.

¹⁵ With *cub* the reduction algorithm will be different depending on a series of factors (GPU architecture, data type, number of elements...).

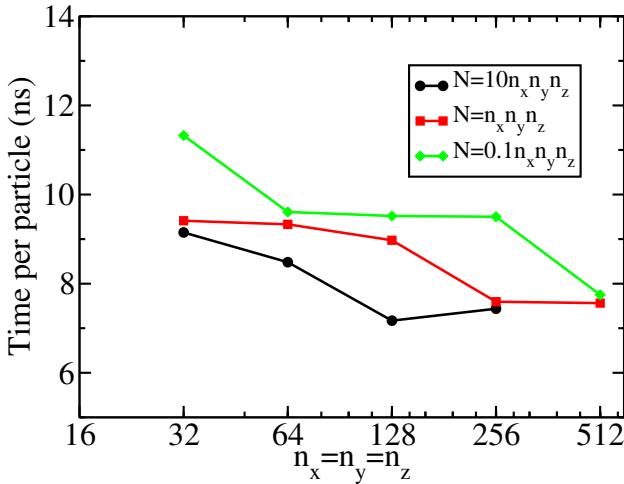


Figure 21.5: Interpolation performance for different densities. Particles are distributed uniformly in a cubic grid and one value is interpolated with a support $n_s = 8$ using a Gaussian kernel. Performance data was gathered in an RTX2080Ti GPU using single precision. All data lies near the 9ns mark.

Use in UAMMD

The **IBM** module in **UAMMD** exposes the spreading and interpolation algorithms described in the previous sections.

This interface is quite generic, allowing to spread or gather in a wide variety of situations. The geometry of the grid is abstracted via the *Grid* object (see Appendix D) so non-regular grids can be used. By default, the module will assume that data for cell (i, j, k) is located at index $i + (j + kn_y)n_x$, but any indexing can be used by providing the module with a functor whose parenthesis operator takes (i, j, k) and returns the index in the grid data array.

The types of the grid and particle data arrays are templated. In fact any random access iterator can be provided as long as the value type of the grid data is convertible to the corresponding variable in the particle data (and vice-versa) and any two given elements can be summed. So one can, for instance, using a *transform iterator*, compute the quantity to spread on the fly without storing or reading anything. For additional information on C++ templates and iterators, see Appendix A.

The gather operation can take an additional functor whose parenthesis operator takes the coordinates of a given cell and a grid object and returns the quadrature weights for that cell.

The default for each cell is taken from the cell volume via the *Grid* object (`dV=grid.getCellVolume(cell)`).

The kernel can also be specified as a template argument. The kernel interface allows to set ϕ in each direction separately and can also be different depending on the particle. The support can also be set per particle.

An implementation of all the kernels in sec. 21 can be found at file *IBM_kernels.cuh*.

A lot of extra functionality (omitted here for simplicity) is laid out in [UAMMD](#)'s online documentation, here we provide an example function that spreads and/or gathers an isotropic Gaussian kernel into a regular Cartesian grid. Representing the simplest, yet most common, use case.

```
#include <uammd.cuh>
#include <misc/IBM.cuh>
using namespace uammd;

//A simple Gaussian kernel compatible with the IBM
// module.
class Gaussian{
    const real prefactor;
    const real tau;
    const int support;
public:
    Gaussian(real width, int support):
        prefactor(pow(2.0*M_PI*width*width, -0.5)),
        tau(-0.5/(width*width)),
        support(support){}
    __device__ int3 getSupport(real3 pos, int3 cell){
        return {support, support, support}
    }
    __device__ real phi(real r, real3 pos) const{
        return prefactor*exp(tau*r*r);
    }
};

template<class Iter1, class Iter2>
void spreadWithIBM(Grid grid, real4* positions,
```

```

        Iter dataAtCellPositions,
        Iter2 dataAtParticlePositions,
        int numberParticles){

const real width = 1; //An arbitrary width
const int support = 8;//An arbitrary support
auto kernel = std::make_shared<Gaussian>(width,
    → support);
IBM<Gaussian> ibm(kernel, grid);
//Spreads dataAtParticlePositions into
→ dataAtCellPositions
ibm.spread(positions, dataAtParticlePositions,
    → dataAtCellPositions, numberParticles);

}

template<class Iter1, class Iter2>
void interpolateWithIBM(Grid grid, real4* positions,
    Iter dataAtCellPositions,
    Iter2
        → dataAtParticlePositions,
        int numberParticles){
const real width = 1; //An arbitrary width
const int support = 8;//An arbitrary support
auto kernel = std::make_shared<Gaussian>(width,
    → support);
IBM<Gaussian> ibm(kernel, grid);
//Interpolates dataAtCellPositions into
→ dataAtParticlePositions
ibm.gather(positions, dataAtParticlePositions,
    → dataAtCellPositions, numberParticles);
}

```

Source Code 26: Usage example of the spreading and interpolation module. Some data (stored in *dataAtParticlePositions*) located at some positions (stored in the *positions* array) can be spread to a grid (for which the values of each cell will be stored in *dataAtCellPositions*) using the function *spreadWithIBM*. Similarly, the function *interpolateWithIBM* can be used for the interpolation operation. As an example, the **FCM** module (chapter 20.1) uses code similar to this one in order to spread particle forces into a grid and, after solving the Stokes equation, interpolate the velocities on a grid back to the particle positions.

21.2 COMPARING THE DIFFERENT SPREADING KERNELS FOR BDHI

As explained in previous sections using different kernels will yield slightly different regularizations of the near field hydrodynamics. In particular, the specific shape of the kernel will have an effect on the hydrodynamic radius when performing the convolution in Eq. (18.24). In this section we will explore the relation between the different kernels showcased in section 21 and the resulting hydrodynamic radius.

The spatial discretization (collocated vs. staggered grid) will also influence the hydrodynamic radius, so we will make a separation between them. Finally, some kernels must be truncated at a certain distance damaging accuracy. Since the domain is periodic in the three directions it is straightforward to see that any possible effect of the kernels on the grid will be the same for any cell. Thus, we just need to measure in one of the cells, or more appropriately, fold our results to a unit cell and average. In particular, we will study the variance of the hydrodynamic radius inside a cell to determine both the accuracy (translational invariance) and the relation between the grid size, h , and the hydrodynamic radius, a ¹⁶. It is possible that the following relations are slightly renormalized in the case of non-cubic cells. However, the self mobility is not well-defined anyway in non-cubic periodic domains. As a matter of fact, if a particle is being pulled on the plane in a box with increasingly large size in the perpendicular direction its in-plane self mobility monotonically increases towards infinity [119]¹⁷.

The self mobility of a single particle being pulled inside a periodic cubic domain of size L is computed using the well known periodic correction for the Stokes drag of a sphere [104],

$$M_0(a, L) = \frac{1}{6\pi\eta a} \left[1 - b \frac{a}{L} + \frac{4\pi}{3} \left(\frac{a}{L} \right)^3 - c \left(\frac{a}{L} \right)^6 \right]. \quad (21.18)$$

Where the factors b and c are

$$\begin{aligned} b &:= 2.83729748, \\ c &:= \frac{16\pi^2}{45} + 23.85. \end{aligned} \quad (21.19)$$

¹⁶ We are assuming a cubic domain in which cells in all directions have the same length.

¹⁷ An effect that is well captured by all the periodic methods we have seen thus far.

We test accuracy by measuring the translational invariance through the self mobility of a particle being pulled in an arbitrary direction. In particular, we measure the variance of the hydrodynamic radius (computing $a = 6\pi\eta M_0$) inside a grid cell. We define the error as

$$E_a(\mathbf{r}) = \left| 1 - \frac{a(\mathbf{r})M_0}{6\pi\eta} \right|. \quad (21.20)$$

Regular grids

We will test using the [FCM](#) with the different kernels since the [PSE](#) is tied to the Gaussian kernel.

Being defined for specific support cells, the Peskin kernels are tied to the grid size, h . When performing the double convolution with the hydrodynamic Green's function (see Eq. (18.24)) the resulting self mobility will be regularized differently for each one and will be dependent on h . Since we cannot perform the double convolution in Eq. (18.24) analytically in the case of Peskin kernels we measure the self mobility (either dynamically as explained in the previous section or by looking at fluctuations) and then renormalize the hydrodynamic radius so that $M_0(L \rightarrow \infty) = \frac{1}{6\pi\eta a}$. In particular, for the Peskin kernels introduced in section 21, the resulting hydrodynamic radii are:

- Peskin 3pt gives $a = (1 \pm 0.01)h$
- Peskin 4pt gives $a = (1.31 \pm 0.001)h$
- Peskin 6pt gives $a = (1.5195 \pm 0.0002)h$

It is important to note that the Peskin kernels prevent us from tweaking the translational invariance of the hydrodynamic radius (as evidenced by the error ranges in the above list).

In contrast to the Peskin kernels for the Gaussian we are free to choose how fine the grid is (which we refer to as upsampling) and a truncation distance. Both parameters will affect the overall accuracy of the spreading and interpolation. The Gaussian gives $a = h\sqrt{\pi}g_u$, where we set the upsampling, g_u , large enough to satisfy a given tolerance. In particular we set

$$g_u(\epsilon) := \min(0.55 - 0.11 \log_{10}(3\epsilon), 1.65) \quad (21.21)$$

Which is just an empirical fit. We truncate the Gaussian kernel at a distance r_c so that $\phi_G(r_c) < \epsilon$. The author of [103] provides

an in-depth analysis of the discretization errors of the Gaussian kernel. Figure 21.6 shows the average error inside a fluid cell for the Peskin kernels, compared to similarly supported Gaussian kernels. At first sight, it might seem like the error is not much better for the Peskin kernels as compared with the Gaussian ones. However, the Gaussian kernels require a greater upsampling (i.e more cells, see Eq. (21.21)) than the Peskin ones to achieve the same tolerance and thus the latter are in general a better choice. Figure 21.7 shows the hydrodynamic radius variance inside a cell for some kernels, but in a slice of the cell (at the center height).

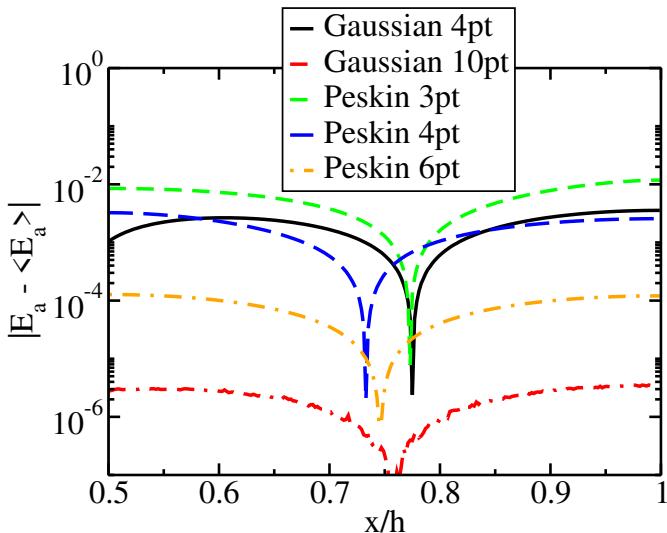


Figure 21.6: Deviation from the average hydrodynamic radius inside a cell in the x direction. Shown here is the error computed by evaluating Eq. (21.20) inside a range $x = [0, 1]h$ with the mean subtracted. Since the signal is symmetrical only the range $[0.5, 1]h$ is presented.

Finally, in the case of BM the quadrature error and hydrodynamic radius are different for each n_s , β and h . In a way, a new kernel is obtained for each support (which relates to the tolerance) and β must be tweaked to achieve a particular hydrodynamic radius. The actual relations are still being tested and thus we refrain from laying them out here for the time being. It is worth mentioning that, provided the same support, the BM kernel tends to greatly outperform the Gaussian (by yielding a lower quadrature error). Still, there are some considerations that must be taking

into account when using the BM kernel for immersed boundary, in particular coming from the fact that it is not separable. A more in depth discussion about the BM kernel can be found in [114].

Staggered grids

We perform the same tests using the **FIB**, which uses a staggered grid.

- Peskin 3pt gives $a = (0.910 \pm 0.005)h$
- Peskin 4pt gives $a = (1.265 \pm 0.002)h$
- Peskin 6pt gives $a = (1.4830 \pm 0.0001)h$

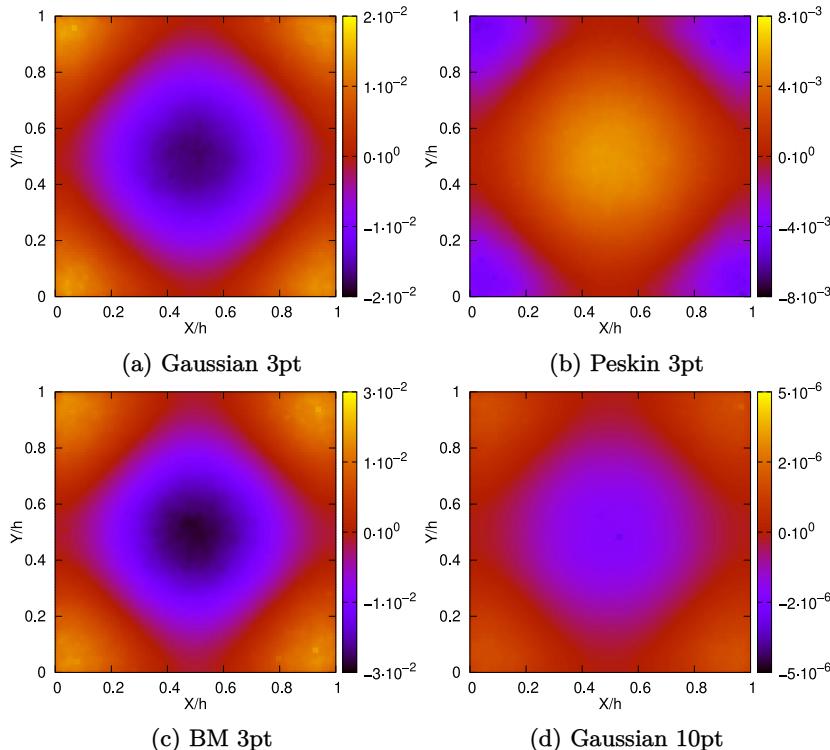


Figure 21.7: Variance of the hydrodynamic radius inside a cell (2D slice at $z = L/2 + h/2$ of a 3D system) for different kernels in **FCM**. Tests were carried out on a cubic box of size $L = 32a$.

In general using a staggered grid requires a smaller grid to get the same hydrodynamic radius (with a similar translational

invariance) than a collocated one. On the other hand, the access patterns in a staggered grid are usually more complex and can result in less performant code, even when the cost of the FFT will be slightly inferior. In the case of a triply periodic system, using a staggered grid via the [FIB](#) is not worth it in general as compared to the [FCM](#). All tests were carried out with single precision (to measure the error in a “real-life” setting) using an RTX3080 [GPU](#).

Part IV

NOVEL ALGORITHMS AND PHYSICS FOR COMPLEX FLUIDS

In this part of the manuscript we will go through several novel, GPU-focused, algorithms that have been developed during this thesis. In particular, we will discuss four new algorithms; two of them solving the Stokes equation (see Eq. (18.7)) for hydrodynamics and the other two solving the, not yet introduced, Poisson equation (see Eq. (24.1)) for electrostatics. In chapter 22 we introduce a novel algorithm for solving the Stokes equation in quasi 2D geometries, in which particles are restricted to move in a periodic plane while embedded in a fluid that is either open (quasi 2D) or non-existent (true 2D) in the third direction. Later, in chapter 23 we solve the Stokes equation again but this time letting the particles move freely inside a doubly periodic domain (periodic in the plane and open in the third direction), introducing the possibility of placing walls at the domain limits, creating slit or walled geometries. The doubly periodic Stokes algorithm will be the last methodology for hydrodynamics in this manuscript. We will use the two subsequent chapters in this part to introduce two algorithms for electrostatics (by solving the Poisson equation)¹⁸. We will start by describing a method for computing the electrostatic interactions of a group of charged particles in a triply periodic environment in chapter 24. Finally, in a similar way as done with Stokes, we will compute electrostatic interactions in a doubly periodic environment (with the possibility of placing surface charged walls at the domain limits).

22

HYDRODYNAMICS UNDER QUASI TWO-DIMENSIONAL CONFINEMENT

In this section we will develop algorithms to study the dynamics of a group of colloidal particles whose movement is constrained in the z direction by some external force (see Fig. 22.1) while submerged in an otherwise unbounded three-dimensional fluid. We refer to this geometry as quasi 2D (q2D for short), as opposed to a situation in which both the particles *and* the fluid are restricted to the plane (true 2D or t2D) or when both are unbounded in the three directions (true 3D or t3D). In the limit when the confining force is infinitely stiff, particles move in a strictly two-dimensional

¹⁸ The algorithms for electrostatics make use of the toolset developed for FCM and PSE in previous chapters, which is the reason why its introduction was delayed up to this point.

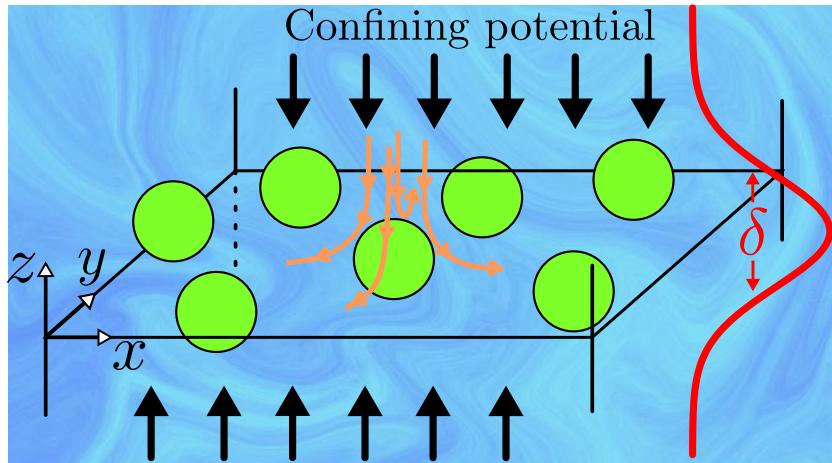


Figure 22.1: Representation of a colloidal system under soft confinement in the perpendicular (z) direction. Particles are confined via some external potential that distributes them near $z = 0$ with a typical width δ . The confining forces acting on the particles are propagated to the plane via the Oseen tensor (orange lines).

plane. This can be used to model, for instance, the transport properties of systems at fluid-fluid or air-fluid interfaces.

Besides being common in subcellular biology, there are several examples of these constrained dynamics systems in a wide variety of industrial applications, like food, creams, or crude oil (e.g., asphaltenes near water interfaces). On the other hand, the transverse diffusion of proteins embedded in lipid bilayers controls their biological function [120]. Note that although confined to the membrane plane, these proteins will be subject to relatively large spatial fluctuations.

In general, diffusion of colloidal particles confined to two-dimensional surfaces is a key transport mechanism in several contexts of technological and biological significance.

As to the physical phenomenon that actually constrains the particles to a plane we have, for instance, the so-called Pickering emulsions, which are stabilized by the spontaneous absorption of colloidal particles to the fluid-fluid interface.

Colloids might also be trapped in fluid-fluid interfaces and interact via capillary [121] or electrostatic [122] forces. By placing walls, particles can be forced to diffuse in a plane [123].

A standing pressure wave with hundreds of megahertz or more creates an ultrasound potential which moves heavy colloidal particles towards two-dimensional traps formed at the nodes of the pressure wave (or the valleys if particles are lighter than the fluid) [86, 124]. The resulting ultrasound potential is harmonic, leading to a Gaussian colloidal dispersion around the pressure node [86].

Similar harmonic traps can be obtained by laser tweezers [125]. Other forms of (non-Gaussian) traps can be prepared using electric fields (maybe leading to barometric-like density profiles).

In future sections, we will study diffusive phenomena arising in this special geometry. For now, let's lay out the algorithmic machinery necessary to simulate it.

Let's start by considering a suspension of colloidal particles confined near $z = 0$ by a strong confining potential submerged in an otherwise unbounded three-dimensional incompressible fluid. As we saw in chapter 18, forces acting on the particles are propagated to the whole fluid, due to its incompressibility, via the Oseen tensor. This results in part of the momentum introduced in the fluid by the confining force acting on one particle (which acts in the normal direction) being propagated over the plane. As we will soon see, the resulting flow source tends to expel other particles around it (see Fig. 22.1). This can be interpreted as the flow in the plane having a positive divergence, as seen from within the plane. Indeed, we will see that this effect naturally arises in a particle-based description (BDHI) through an effective plane mobility with non-zero divergence [126].

There are several ways to gradually transform a perfect 2D confinement into an isotropic 3D distribution and, as illustrated in Fig. 22.1, we model the case where the 2D interface of colloids becomes Gaussianly blurred. In particular, we consider a suspension of colloids confined in the z direction around some height, $z = 0$, by a harmonic potential of the form

$$U_{\text{conf}}(z) = \frac{1}{2}K_s z^2 \quad (22.1)$$

applied to each individual particle. The spring constant K_s controls the width, δ , of the Gaussian distribution of particles in the z direction.

$$\delta = \left(\frac{k_B T}{K_s} \right)^{1/2} \quad (22.2)$$

We can use this as a slider to go from t3D, in the limit $K_s \rightarrow 0$, to q2D when $K_s \rightarrow \infty$. The dynamics of the system are then governed by Eq. (18.20) with an Oseen-like mobility (like the RPY one). Note that solvent inertial effects can also be incorporated by using the ICM in sec. 20.5.

Thus far, all the algorithms we have developed for solving (18.20) (or more generally Eq. (18.1)) deal with either fully open or triply periodic boundary conditions. However, now the perpendicular direction should be open while leaving the plane periodic.

To be fair, our mathematical infrastructure allows to discretize the Stokes operator in any geometry to accommodate any boundary conditions. But since our implementations rely on the FFT to be efficient and GPU-friendly these modifications are not straightforward.

One approach to deal with this is to simply use the triply periodic algorithms with a box with width, $L_z \gg \delta$, large enough to neglect finite-size effects. However, since hydrodynamics are long ranged in nature we will be forced to perform a full finite-size analysis to ensure the convergence of the measured properties.

In future sections, we will describe a family of pseudo-spectral algorithms capable of dealing with non periodic boundary conditions. For the moment, let's adapt the already developed FCM to the limit of strict confinement (q2D). Let's see how we can compute the Fourier expression of a Green's function specific for q2D.

22.1 THE FORCE COUPLING METHOD IN QUASI 2D

Suppose that all particles are subject to a force coming from the potential in Eq. (22.1). We want to take the mathematical limit of Eq. (18.20) (using $\mathbf{F} = -\partial_{\mathbf{q}} U_{\text{conf}}$) when $K_s \rightarrow \infty$ (hence particles move strictly in the plane $z = 0$). Although it is possible to take this limit formally the general theory for it is quite complex. Luckily we can elaborate some simple arguments and assumptions to make this in a trivial way (we provide a more in depth analysis in [71]).

To understand the enhancement of collective diffusion in confined geometries, we first consider the Smoluchowsky equation for the particle concentration field $\rho(\mathbf{r}, t) = \sum_i \delta(\mathbf{r} - \mathbf{q}_i)$ of ideal (i.e, non-interacting via potential forces) colloids,

$$\partial_t \rho = -\boldsymbol{\partial}_r \cdot [-D_0 \boldsymbol{\partial} \rho + \rho \mathbf{v}_d]. \quad (22.3)$$

Here D_0 is the bare diffusion coefficient and \mathbf{v}_d is the drift velocity, given by

$$\mathbf{v}_d(\mathbf{r}, t) = \int \mathcal{M}(\mathbf{r} - \mathbf{r}') f(\mathbf{r}') d\mathbf{r}'. \quad (22.4)$$

Where $f(\mathbf{r})$ is the confining force density, which can be expressed as the gradient of an osmotic pressure. For an ideal gas of colloids, the pressure is $\pi := k_B T \rho(\mathbf{r})$, then

$$\mathbf{f}(\mathbf{r}') = -\boldsymbol{\partial}_{\mathbf{r}} \pi = -k_B T \boldsymbol{\partial}_{\mathbf{r}} \rho = -k_B T \sum_i \partial_z \delta(\mathbf{r} - \mathbf{q}_i) \hat{\mathbf{e}}_z, \quad (22.5)$$

where the last equality particularizes the equation for the in-plane confinement. The drift velocity is then,

$$\begin{aligned} \mathbf{v}_d(\mathbf{r}) &= -\int \mathcal{M}(\mathbf{r} - \mathbf{r}') \boldsymbol{\partial}_{\mathbf{r}} \pi d\mathbf{r}' \\ &= -k_B T \int \mathcal{M}(\mathbf{r} - \mathbf{r}') \sum_i \partial_z \delta(\mathbf{r} - \mathbf{q}_i) d\mathbf{r}' \stackrel{\text{(By parts)}}{=} \\ &= k_B T \int [\partial_{z'} \mathcal{M}(\mathbf{r} - \mathbf{r}')] \sum_i \delta(\mathbf{r} - \mathbf{q}_i) d\mathbf{r}' \\ &= -k_B T \sum_i \int [\partial_z \mathcal{M}(\mathbf{r} - \mathbf{s}')] |_{z=0} \delta(\mathbf{s}' - \mathbf{q}_i) d\mathbf{s}'. \end{aligned} \quad (22.6)$$

Where we introduced the plane coordinates, \mathbf{s} , such that $\mathbf{r} = \mathbf{s} + z \hat{\mathbf{e}}_z$. Here δ represents the Dirac delta. Now, the flow is incompressible in 3D, which implies

$$(\boldsymbol{\partial}_{\mathbf{s}} + \partial_z) \mathcal{M} = 0 \rightarrow \partial_z \mathcal{M} = -\boldsymbol{\partial}_{\mathbf{s}} \mathcal{M}, \quad (22.7)$$

thus, substituting $\partial_z \mathcal{M}$ in Eq. (22.6) with Eq. (22.7),

$$\mathbf{v}_d(\mathbf{s}) = k_B T \sum_i \int (\boldsymbol{\partial}_{\mathbf{s}} \mathcal{M})_{z=0} \delta(\mathbf{s}' - \mathbf{q}_i) d\mathbf{s}' = k_B T (\boldsymbol{\partial}_{\mathbf{q}} \cdot \mathcal{M})_{z=0}, \quad (22.8)$$

which indicates that $(\boldsymbol{\partial}_{\mathbf{q}} \cdot \mathcal{M})_{z=0}$ is acting as a flow source term in the plane¹. Notably, the resulting hydrodynamic interaction between two particles 1 and 2 is similar to an electrostatic repulsion,

$$\mathbf{v}_d(\mathbf{q}_1) = k_B T (\boldsymbol{\partial}_{\mathbf{q}_2})_{z_1=z_2=0} \mathcal{M} \approx \frac{1}{8\pi\eta q_{12}^3} \mathbf{q}_{12}, \quad (22.9)$$

¹ Note that we have restored back to supervector notation, i.e., $\boldsymbol{\partial}_{\mathbf{q}} \cdot \mathcal{M} = \sum_i \boldsymbol{\partial}_{\mathbf{q}_i} \cdot \mathcal{M}(\mathbf{s} - \mathbf{q}_i)$.

where we have used the Oseen mobility for \mathcal{M}^2 .

The collective effect on the colloidal density $\rho(\mathbf{r}, t)$ can be analyzed from eq. (22.3).

$$\partial_t \rho = \partial_{\mathbf{r}} \cdot \left[D_0 \partial_{\mathbf{r}} \rho + k_B T \bar{\rho} \int \mathcal{M}(\mathbf{r} - \mathbf{r}') \partial_{\mathbf{r}} \rho d\mathbf{r}' \right] \quad (22.10)$$

Where we have approximated ρ by its average, $\bar{\rho}$, in the second term (linear perturbation approximation). Taking the Fourier transform in space,

$$\partial_t \hat{\rho}(\mathbf{k}, t) = \left[D_0 k^2 + k_B T \bar{\rho} \mathbf{k} \cdot \mathcal{M} \cdot \mathbf{k} \right] \hat{\rho}, \quad (22.11)$$

which solution is given by

$$\hat{\rho}(\mathbf{k}, t) = \hat{\rho}(\mathbf{k}, 0) \exp \left(-k^2 D_c(\mathbf{k}) t \right), \quad (22.12)$$

where D_c is the short time collective diffusion coefficient³,

$$D_c(\mathbf{k}) = D_0 + \bar{\rho} k_B T \frac{\mathbf{k} \cdot \mathcal{M} \cdot \mathbf{k}}{k^2}. \quad (22.13)$$

In 3D, $D_c = D_0$ (since the divergence of the mobility is null), as expected for ideal particles. However, for q2D it is not difficult to show [71, 127] that,

$$D_c = D_0 + (L_n k)^{-1} + O(k), \quad (22.14)$$

with

$$L_n := \frac{2}{3\pi a \bar{\rho}_{2D}}, \quad (22.15)$$

where $\bar{\rho}_{2D}$ is the surface density. Thus collective diffusion increases as the inverse of the wave number. This analysis indicates that to impose the strict⁴ confinement condition in q2D of Brownian

² This interaction could be equivalent to an effective electrostatic potential $U_{\text{eff}} \approx \frac{3a}{4r} k_B T$. However, unlike a potential interaction, the thermal drift $\partial_s \cdot \mathcal{M}$ does not induce any distortion in the equilibrium distribution. On the contrary, the addition of the drift term $k_B T \partial_s \cdot \mathcal{M}$ precisely ensures that the equilibrium distribution is consistent with the Boltzmann presumption $P(\mathbf{q}) \propto \exp(-\beta H(\mathbf{q}))$; given by the effective Hamiltonian of the system (see chapter 3.3.1).

³ The collective diffusion coefficient determines the decorrelation time $(D_c(k) k^2)^{-1}$ of a density fluctuation of typical wavenumber k .

⁴ If $U = \frac{1}{2} K_s z^2$ is used, there is no need of $\nabla_s \cdot \mathcal{M}$, because we resolve the whole 2D flow.

particles one needs to include the thermal or spurious drift related to the divergence of the (effective) q2D mobility, $(\partial_s \cdot \mathcal{M})_{z=0}$ (according to eq. (22.8)). The resulting particle dynamics **at the confining plane** are described by eq. (17.7). This means that we can simply remove the z component in our description and write

$$d\mathbf{q} = \mathcal{M}\mathbf{F}dt + \sqrt{2k_B T \mathcal{M}} d\widetilde{\mathbf{W}} + k_B T (\partial_q \cdot \mathcal{M}) dt, \quad (22.16)$$

where the particle positions and forces, the mobility and the noise are now defined only in the plane. In order to implement eq. (22.16) within the **FCM** procedure (introduced in section 20.1) we use the second equality of eq. (22.6), including also putative forces \mathbf{F} (per particle) arising from colloidal interactions and/or external fields. In such a case the force density is,

$$\mathbf{f}(\mathbf{r}) = \mathcal{S}\mathbf{F} + \partial_q \mathcal{S} [k_B T]. \quad (22.17)$$

We have generalized the Dirac delta by using the spreading operator, \mathcal{S} (as we did when introducing spreading and interpolation in chapter 18). The second term corresponds to the osmotic force, $-\partial_r \pi = -k_B T \partial_r \mathcal{S} = k_B T \partial_q \mathcal{S}$. The kernel \mathcal{S} spreads forces while its derivative⁵ spreads internal energies⁶.

Following the **FCM**, we will set \mathcal{S} as a Gaussian, and in order to implement a purely 2D computational set up, we need to integrate out the effect of the third direction, z , from the equations of motion. To that end, we recall that $\mathcal{M} = \mathcal{J}\mathcal{G}\mathcal{S}$, where \mathcal{G} is the Green function for the fluid velocity field. The particle kernels are separable ($\mathcal{S} = \phi(x)\phi(y)\phi(z)$) which permits including them into the 2D Fourier transform of the Green function, to yield,

$$\hat{\mathcal{G}}_{\text{q2D}}(\mathbf{k} = (k_x, k_y)) = \frac{1}{2\pi} \int_{k_z=-\infty}^{\infty} \hat{\phi}(k_z)^2 \hat{\mathcal{G}}_{\text{3D}}(\mathbf{k}; k_z) dk_z. \quad (22.18)$$

Using a Gaussian kernel for ϕ and the Oseen tensor for \mathcal{G}_{3D} ,

$$\begin{aligned} \hat{\mathcal{G}}_{\text{q2D}}(\mathbf{k}) &= \frac{1}{2\pi\eta} \int_{k_z} \frac{dk_z}{(k')^2} \exp\left(-\frac{a^2 k_z^2}{\pi}\right) \left(\mathbb{I} - \frac{\mathbf{k}' \otimes \mathbf{k}'}{(k')^2}\right) \\ &= \eta^{-1} (g_k(ka) \mathbf{k}_\perp \otimes \mathbf{k}_\perp + f_k(ka) \mathbf{k} \otimes \mathbf{k}). \end{aligned} \quad (22.19)$$

⁵ In fact, in the case of a Gaussian, we can spread a single quantity by using $\partial\phi_G(r) = 2\sigma_a r \phi_G(r)$ and spreading $\mathbf{F} + 2k_B T \sigma_a \mathbf{r}$.

⁶ This latter idea was generalized in [86] to model ultrasound forces on compressible particles, with energy $\psi(\rho)k_B T$.

Where $\mathbf{k}' = (k_x, k_y, k_z)$ is the three dimensional wavenumber and we recall that unless stated otherwise, in quasi 2D vectors are defined in \mathbb{R}^2 . Additionally $\mathbf{k}_\perp := \mathbf{k} \times \hat{\mathbf{z}} = (k_y, -k_x)$ is a vector perpendicular to \mathbf{k} . The convolution with the (x, y) kernel is performed explicitly via spreading in [FCM](#). Note that this has restricted our algorithm to a Gaussian kernel, with its advantages and disadvantages. However, a different kernel can be used if it has an analytical Fourier form that allows the integral in Eq. (22.19) to be solved analytically⁷. Solving Eq. (22.19) yields

$$\begin{aligned} g_k(K) &= \frac{1}{2K^3} \left[1 - \text{erf} \left(\frac{K}{\sqrt{\pi}} \right) \right] \exp \left(\frac{K^2}{\pi} \right) \\ f_k(K) &= \left(\frac{1}{2} - \frac{K^2}{\pi} \right) g_k(K) - \frac{1}{2\pi K^3} \end{aligned} \quad (22.20)$$

We can also write the Green's function for true 2D by redefining f_k and g_k (just the Oseen tensor in 2D)

$$\begin{aligned} g_k(K) &= 0 \\ f_k(K) &= \frac{a}{K^4} \end{aligned} \quad (22.21)$$

The Green's function (22.19) is purely two-dimensional, allowing to reuse the [FCM](#) machinery without the third direction altogether.

Using Eq. (22.21) instead of Eq. (22.20) allows for our algorithm to simulate a true 2D system. In fact, any 2D hydrodynamic kernel can be used as long as f_k and g_k are known⁸.

The particle dynamics can thus be formally written as

$$\frac{d\mathbf{q}_i}{dt} = \mathcal{J}_{\mathbf{q}_i} [\mathcal{G}_{\text{q2D}} (\mathcal{S}\mathbf{F} + \partial\mathcal{S}(k_B T)) + \mathbf{w}(\mathbf{r}, t)]. \quad (22.22)$$

Here we have put the fluctuating part, \mathbf{w} , aside. The fluctuating stress tensor (\mathcal{Z} in Eq. (18.20)) and its divergence are described in three dimensions. In the quasi 2D case we do not have an easy mechanism to project the stress tensor into the plane in order to keep a fully two dimensional description. Instead, we will use the fluctuation-dissipation balance to generate a stochastic fluid velocity directly.

⁷ The analytical form of Eq. (22.19) is not actually needed if somehow f_k and g_k are tabulated.

⁸ For instance, the Saffman kernel for membrane hydrodynamics [[Brown20112](#).]

As previously discussed (see chapter 18.1), the mobility tensor corresponding to Eq. (22.19) can be written as

$$\mathcal{M}_{\text{q2D}} = \mathcal{J}\mathcal{G}_{\text{q2D}}\mathcal{S}. \quad (22.23)$$

On the other hand, we know that fluctuation-dissipation (discussed in chapter 3.3.2) mandates that the stochastic particle displacements,

$$\tilde{\mathbf{u}}_i := \mathcal{J}_{q_i} \mathbf{w}, \quad (22.24)$$

must be related with the mobility as

$$\langle \tilde{\mathbf{u}}_i \otimes \tilde{\mathbf{u}}_i \rangle = \frac{2k_B T}{dt} \mathcal{M}_{\text{q2D}}. \quad (22.25)$$

Equivalently, we can write the fluctuation-dissipation relation for the fluid velocity in Fourier space,

$$\langle \hat{\mathbf{w}} \otimes \hat{\mathbf{w}} \rangle = \frac{2k_B T}{dt} \hat{\mathcal{G}}_{\text{q2D}}. \quad (22.26)$$

It is straightforward to devise a functional form for the stochastic fluid velocity that satisfies Eq. (22.26). We can use a similar trick as we did for PSE in Eq. (20.16) to separate the contribution of the noise in two,

$$\hat{\mathbf{w}}(\mathbf{k}, t) := \sqrt{\frac{2k_B T}{\eta}} \left(\sqrt{f_k(ka)} \mathbf{k}_\perp \hat{\mathcal{Z}}_k^1 + \sqrt{g_k(ka)} \mathbf{k} \hat{\mathcal{Z}}_k^2 \right). \quad (22.27)$$

Where $\hat{\mathcal{Z}}_k^{1,2}$ are independent Wiener processes. The same concerns about the delicate conjugacy properties of the noise we discussed in sec. 20.1 are applicable here.

We now have all the necessary ingredients to apply the FCM framework and solve Eq. (22.22) for the particle velocities.

Use in UAMMD

The quasi 2D FCM described in the previous section is available as an *Integrator* in UAMMD. The particular module, called `BDHI::BDHI2D` is templated for any 2D Green's function by providing f_k and g_k (from Eq. (22.19)) through a special structure (refer the online documentation for more information). The aliases `BDHI::True2D` and `BDHI::Quasi2D` are provided as specializations of the base class for the hydrodynamic kernels described in the

previous section (true 2D and quasi 2D respectively). The translational invariance of the hydrodynamic radius is kept below a certain threshold via the tolerance parameter, which controls the support of the Gaussian kernels and the dimensions of the grid. The rest of the input parameters are similar to the ones required for the rest of the hydrodynamic modules we have seen thus far.

```
#include <uammd.cuh>
#include <Integrator/Hydro/BDHI_quasi2D.cuh>
using namespace uammd;
//A function that creates and returns a quasi 2D
//integrator
auto createIntegratorQ2D(UAMMD sim){
    //Choose the hydrodynamic kernel
    using Hydro2D = BDHI::Quasi2D;
    //using Hydro2D = BDHI::True2D;
    Hydro2D::Parameters par;
    par.temperature = sim.par.temperature;
    par.viscosity = sim.par.viscosity;
    par.hydrodynamicRadius =
        → sim.par.hydrodynamicRadius;
    par.dt = sim.par.dt;
    par.tolerance = sim.par.tolerance;
    par.box = sim.par.box;
    auto q2d = std::make_shared<Hydro2D>(sim.pd, par);
    return q2d;
}
```

Source Code 27: Example of the creation of a quasi 2D *Integrator*.

HYDRODYNAMICS IN DOUBLY PERIODIC GEOMETRIES

Thus far we have seen how to compute hydrodynamic displacements in domains that are either triply periodic (periodic in the three directions) or two dimensional and periodic in the plane. In the case of quasi 2D, where the particles are constrained to a plane, the boundary conditions for the fluid in the perpendicular directions are open, while in true 2D both the fluid and the particles exist in the plane. We refer to a domain as **Doubly Periodic (DP)** when it is periodic in the plane directions (x, y) but aperiodic in the perpendicular one (z). In this sense the quasi 2D algorithm in chapter 22 can be regarded as a **DP** algorithm. However, the aforementioned algorithm only allows to simulate environments in which the submerged particles diffuse in two dimensions (for instance, a fluid-fluid interface). If we are looking to model a system lying in between the triply periodic and quasi 2D regimes, for instance a lipid membrane (see Fig. 23.1), currently our only option is to perform a triply periodic simulation and do some kind of finite-size analysis, measuring the relevant properties with an increasingly large simulation domain size in the perpendicular direction. Of course, given that the algorithmic complexity of all our pseudo-spectral algorithms scales linearly with the volume of the domain (or more appropriately, with the number of cells in the grid), the triply periodic approach is not optimal.

We propose a novel pseudo-spectral **FCM**-like algorithm, with a wildly different approach to the ones we have seen thus far, that allows to compute the hydrodynamic displacements of a suspension of colloids inside a **DP** domain with either open or no-slip conditions in the third direction. In particular, we will see how to solve the incompressible Stokes equation (18.7) without fluctuations in a **DP** domain of size $L_{x,y}$ (we will assume $L_x = L_y$, which can then be easily generalized). Thus, we want to solve

$$\begin{aligned} \nabla p_{\text{DP}} - \eta \nabla^2 \mathbf{v}_{\text{DP}} &= \mathbf{f} \\ \nabla \cdot \mathbf{v}_{\text{DP}} &= 0, \end{aligned} \tag{23.1}$$

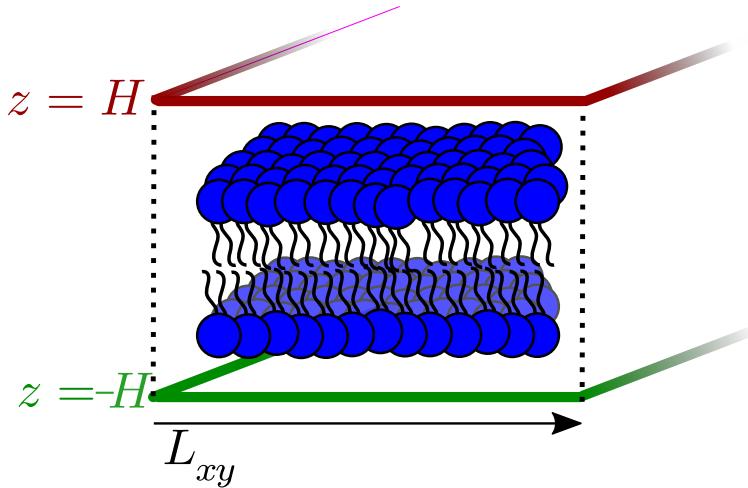


Figure 23.1: Representation of a lipid membrane inside a doubly periodic domain. The [Boundary Conditions \(BCs\)](#) can be customized at the domain limits ($z = -H, H$) to be either open or have no-slip walls.

with $z \in (-\infty, \infty)$ and a certain set of [BCs](#) (that will be introduced shortly) in z , assuming that \mathbf{v}_{DP} is bounded as $|z| \rightarrow \infty$. The plane (x, y) is periodic as in quasi 2D.

Here p is the pressure and the fluid forcing includes the particle forces and torques (similarly to the section on [FCM](#), see sec. 20.1)

$$\mathbf{f}(\mathbf{r}) = \mathcal{S}\mathbf{F} + \frac{1}{2}\nabla \times \mathcal{S}_\tau\boldsymbol{\tau}. \quad (23.2)$$

We assume that the fluid forcing is zero outside a domain $z \in [-H, H]$. In the case of an unbounded fluid this domain is arbitrary and physically meaningless, serving only to define a numerical domain for our grid-based solver. The same thing happens in the case of a bottom wall with the top limit at $z = H$. Furthermore, assuming $\mathbf{f}(|z| > H) = 0$ allows us to define a set of [BCs](#) more easily.

Our solver is periodic in the plane (x, y) (as in quasi 2D). In the z direction, by incorporating different [BCs](#), we distinguish between three modes:

1. Unbounded in z , $z \in (-\infty, \infty)$. \mathbf{v}_{DP} is bounded as $|z| \rightarrow \infty$.

2. A no-slip wall at the bottom of the domain, $z = -H$ (unbounded at $z > H$). $z \in [-H, \infty)$. The no-slip BC for the wall is simply

$$\mathbf{v}_{\text{DP}}(z = -H) = 0. \quad (23.3)$$

3. A slit channel, with no-slip walls located at $z = -H$ and $z = H$. $z \in [-H, H]$. Besides the BC for the bottom wall, we add a similar one for the top one

$$\mathbf{v}_{\text{DP}}(z = H) = 0. \quad (23.4)$$

The unbounded mode allows to generalize our quasi2D simulations to systems where particles are not restricted to a plane. Additionally the other modes can be used in a variety of simulations. For instance, the bottom wall mode can be used to model the hydrodynamics of a Quartz Crystal Microbalance (QCM), improving existing models [128].

The present DP solver is vastly different from the FCM family of methods we have seen thus far.

Our approach consists in solving the Stokes equation in an unbounded domain, adding the effects of the walls afterwards as a correction. Thus we first solve Eq. (23.1) with free-space BCs and $z \in (-\infty, \infty)$. Then we compute the final result in the presence of one or two walls as

$$\begin{aligned} \mathbf{v} &= \mathbf{v}_{\text{DP}} + \mathbf{v}_{\text{corr}} \\ \mathbf{p} &= \mathbf{p}_{\text{DP}} + \mathbf{p}_{\text{corr}}. \end{aligned} \quad (23.5)$$

For the corrections we solve analytically Eq. (23.1) in the absence of forces with one or two slip walls. In particular we have

$$\begin{aligned} \nabla p_{\text{corr}} - \eta \nabla^2 \mathbf{v}_{\text{corr}} &= 0 \\ \nabla \cdot \mathbf{v}_{\text{corr}} &= 0, \end{aligned} \quad (23.6)$$

solved in a domain with $z \in [-H, \infty)$ in the case of a bottom wall and $z \in [-H, H]$ in the case of a slit channel. Finally, we set slip BCs at the walls so

$$\begin{aligned} \mathbf{v}_b &:= -\mathbf{v}_{\text{corr}}(z = -H) = \mathbf{v}_{\text{DP}}(z = -H) \\ \mathbf{v}_t &:= -\mathbf{v}_{\text{corr}}(z = H) = \mathbf{v}_{\text{DP}}(z = H), \end{aligned} \quad (23.7)$$

where the second condition, at $z = H$, is imposed only in the case of a slit channel geometry.

In Appendix C we describe a fast and GPU-friendly Boundary Value Problem (BVP) solver that solves one-dimensional PDEs

in the Chebyshev basis. Our strategy in the next sections will be to write equations for the velocity and pressure in a way that is compatible with this solver.

We will first discuss the free-space solver ($\mathbf{v}_{\text{corr}} = 0$ and $p_{\text{corr}} = 0$) and then see how to solve the correction in each case.

23.1 FREE-SPACE SOLVER

Let us start by writing the equations for the pressure (Eq. (18.9)) in Fourier space only in the plane (x, y) , which makes it easier to incorporate arbitrary BCs in the z direction.

Pressure solve

By taking the divergence of Eq. (23.1) we eliminate the velocity

$$\nabla^2 p = \nabla \cdot \mathbf{f}. \quad (23.8)$$

Transforming to Fourier space in the plane we get a one dimensional problem for each wave number

$$(\partial_z^2 - k^2)\hat{p} = [i\mathbf{k}, \partial_z] \cdot \mathbf{f}, \quad (23.9)$$

where the wave vector, $\mathbf{k} := (k_x, k_y)$, is defined on the plane and has modulus $k^2 = k_x^2 + k_y^2$. The forces, $\mathbf{f} = (f^x, f^y, f^z)$ can be non-zero only inside a certain domain with $z \in [-H, H]$. Note that the pressure (and the forces) are Fourier transformed only in the plane, so that $\hat{p} := \hat{p}(\mathbf{k}, z)$. As a matter of fact, the [Fast Chebyshev Transform \(FCT\)](#) is employed here to transform the signals in the z direction to the Chebyshev basis, as this allows to use the [BVP](#) solver and facilitates the evaluation of the derivatives in this direction (by using the relations for the Chebyshev coefficients introduced in Appendix C). Therefore we will work with the Chebyshev coefficients, \hat{p}_n (or $\hat{\mathbf{f}}_n$), of the different quantities.

We can solve the pressure outside the domain using that

$$(\partial_z^2 - k^2)\hat{p} = 0, \quad \text{for } z \notin [-H, H]. \quad (23.10)$$

Whose solution is

$$\hat{p}(k, z) = C_1 \exp(-kz) + C_2 \exp(kz). \quad (23.11)$$

By using the boundedness of the pressure at $|z| \rightarrow \infty$ we can write the solution outside the domain

$$\begin{aligned}\hat{p}(k, z \leq -H) &= C_2 \exp(kz) \\ \hat{p}(k, z \geq H) &= C_1 \exp(-kz).\end{aligned}\quad (23.12)$$

This implies

$$(\partial_z \pm k)\hat{p}(k, z = \pm H) = 0. \quad (23.13)$$

For a given wave number, k , the system in Eq. (23.9) along the BCs in Eq. (23.13) constitutes a BVP that can be solved in Chebyshev space with the solver described in Appendix C.

Since this solver requires to define z in the Chebyshev basis, we can apply ∂_z in linear time by using recurrent relations on the Chebyshev coefficients (in this case of \hat{f}^z) instead of via ik differentiation. As discussed in Appendix C, the Fourier-Chebyshev¹ coefficients $\hat{p}(\mathbf{k}, z)$ are computed via a hybrid FFT-FCT. Which amounts to performing the 3D FFT of \mathbf{f} periodic extended in z .

Velocity solve

Once the Chebyshev coefficients for the pressure for each wave number are known we can write the equations for the three components of the velocity in Fourier space as

$$\eta \left(\partial_z^2 - k^2 \right) \hat{\mathbf{v}} = \begin{bmatrix} i\mathbf{k} \\ \partial_z \end{bmatrix} \hat{p} - \hat{\mathbf{f}}, \quad (23.14)$$

where the derivative of the pressure, $\partial_z \hat{p}$, can be computed via the Chebyshev coefficients of the pressure. For the BCs we know that outside the domain, where $\hat{\mathbf{f}} = 0$, the velocity satisfies

$$\eta \left(\partial_z^2 - k^2 \right) \hat{\mathbf{v}} = \begin{bmatrix} i\mathbf{k} \\ \partial_z \end{bmatrix} \hat{p}, \quad \text{for } z \notin [-H, H]. \quad (23.15)$$

The BCs for Eq. (23.15) must be computed independently for the plane, \mathbf{v}^\parallel , and perpendicular, \mathbf{v}^\perp , velocities.

¹ Fourier in the plane (x, y) and Chebyshev in z .

Parallel velocity solve

Using the boundness of the velocity at $\pm\infty$, the solution of Eq. (23.15) in the plane for $z \leq -H$ and $z \geq H$] is, respectively

$$\hat{\mathbf{v}}^{\parallel} = \mp \frac{C_1 \mathbf{k} \exp(\pm kz) (2zk \mp 1)}{4\eta i k^2} + C_2 \exp(\pm zk). \quad (23.16)$$

The derivative of Eq. (23.16) can be written as

$$\partial_z \hat{\mathbf{v}}^{\parallel} = \pm k \hat{\mathbf{v}}^{\parallel} + \frac{C_1 i \mathbf{k} \exp(\pm kz)}{2\eta k}. \quad (23.17)$$

We can make use of the previously computed solution for the pressure outside the domain,

$$\hat{p}(\mathbf{k}, z \leq -H \text{ or } z \geq H) = C_1 \exp(\pm zk), \quad (23.18)$$

to finally write the BCs for the parallel velocity as

$$(\partial_z \pm k) \hat{\mathbf{v}}^{\parallel}(\mathbf{k}, \pm H) = \mp \frac{i \mathbf{k}}{2\eta k} \hat{p}(\mathbf{k}, \pm H). \quad (23.19)$$

Note that the pressure at the domain limits, $\hat{p}(\mathbf{k}, \pm H)$, can be easily computed from the already available Chebyshev coefficients of the pressure.

Perpendicular velocity solve

Following a similar strategy as with the parallel velocity, we can write the solution for the perpendicular velocity outside the domain as

$$\hat{\mathbf{v}}^{\perp} = \frac{C_1 \mathbf{k} \exp(\pm kz) (2zk \mp 1)}{4\eta k} + C_2 \exp(\pm zk) \quad (23.20)$$

and its derivative

$$\partial_z \hat{\mathbf{v}}^{\perp} = \pm k \hat{\mathbf{v}}^{\perp} + \frac{C_1 \exp(\pm kz)}{2\eta}. \quad (23.21)$$

Identifying the already known solution for the pressure (see Eq. (23.18)) we can finally write the BCs for the perpendicular velocity as

$$(\partial_z \pm k) \hat{\mathbf{v}}^{\perp}(\mathbf{k}, \pm H) = \frac{1}{2\eta} \hat{p}(\mathbf{k}, \pm H). \quad (23.22)$$

Thus far we have shown how to compute, for all wave numbers, the Chebyshev coefficients for the pressure and the three directions of the velocity in the free-space case. We will now study the correction in the bottom wall and slit channel geometries.

23.2 CORRECTIONS

We will compute, for each wave number, the analytical solution of the velocity and pressure inside the domain due to the presence of the walls and sum it to the main solution.

Let us start with the bottom wall case.

Bottom wall

We want to solve Eq. (23.6) with BCs in Eq. (23.3).

For simplicity, let us define and $z' := (z + H)/2$. It is straightforward to prove that the following solutions for the velocity and the pressure satisfy Eq. (23.6) with BCs in Eq. (23.3):

$$\hat{\mathbf{v}}_{\text{corr}}(\mathbf{k}, z) = \left(\hat{\mathbf{v}}_b - \begin{bmatrix} \mathbf{k} \\ ik \end{bmatrix} ([\mathbf{k}, ik] \cdot \hat{\mathbf{v}}_b) \frac{z'}{k} \right) \exp(-kz'), \quad (23.23)$$

$$\hat{p}_{\text{corr}}(\mathbf{k}, z) = 2\eta [-i\mathbf{k}, k] \cdot \hat{\mathbf{v}}_b \exp(-kz'). \quad (23.24)$$

The velocity at the bottom, $\hat{\mathbf{v}}_b$, can be trivially evaluated via its already available Chebyshev coefficients. Once the solution is computed for every wave number, the Chebyshev coefficients can be obtained by applying the FCT to each of them. Finally, we compute the total solution using Eq. (23.5).

Slit channel

The double wall case proves to be a little more convoluted. We want to solve Eq. (23.6) with BCs in Eq. (23.3) and (23.4). The general solution for the correction in this case is

$$\begin{aligned} \hat{\mathbf{v}}_{\text{corr}}(\mathbf{k}, z) &= \left(\frac{C_0}{2\eta k} z' \begin{bmatrix} -i\mathbf{k} \\ k \end{bmatrix} + \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \right) \exp(-kz') + \\ &\quad \left(\frac{D_0}{2\eta k} z' \begin{bmatrix} i\mathbf{k} \\ k \end{bmatrix} + \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix} \right) \exp(kz') \end{aligned} \quad (23.25)$$

$$\hat{p}_{\text{corr}}(\mathbf{k}, z) = C_0 \exp(-kz') + D_0 \exp(kz') \quad (23.26)$$

The six equations above are not enough to compute the values of the eight unknown coefficients, $\mathbf{C} = C_{[0,1,2,3]}$ and $\mathbf{D} = D_{[0,1,2,3]}$. For the remaining two equations we can use the incompressibility condition in Eq. (23.1),

$$[i\mathbf{k}, \partial_z] \cdot \hat{\mathbf{v}}_{\text{corr}}(\mathbf{k}, z) = 0, \quad (23.27)$$

evaluated at both the domain limits, $z = -H, H$. Evaluating Eq. (23.27) and replacing $\partial_z \hat{v}_{\text{corr}}^z(\mathbf{k}, z = \pm H)$ from Eq. (23.25) yields the last two required equations,

$$\begin{aligned} \frac{C_0}{2\eta} + \frac{D_0}{2\eta} - C_3 k + D_3 k &= -ik_x \hat{v}_b^x - ik_y \hat{v}_b^y \\ \left[(1 - 2kH) \frac{C_0}{2\eta} - kC_3 \right] \exp(-2kH) + \\ \left[(1 + 2kH) \frac{D_0}{2\eta} + kD_3 \right] \exp(2kH) &= -ik_x \hat{v}_t^x - ik_y \hat{v}_t^y. \end{aligned} \quad (23.28)$$

We now have a linear system of 8 equations and 8 unknowns (\mathbf{C} and \mathbf{D}). Once this system is solved² we can evaluate the correction and compute its Chebyshev coefficients. Finally we sum the correction to the free domain solution, obtaining the final result as per Eq. (23.5).

23.3 THE ZEROTH WAVE NUMBER

Instead of computing the zeroth mode for each of the subproblems separately (the free space solver and the correction solve) we treat them as a unique solver for simplicity. The free space solver (no walls) does not require a correction and handling the zeroth mode amounts to simply setting $\mathbf{v}(\mathbf{k} = 0, z) = 0$. Let us go through the bottom wall and slit channel geometries.

Bottom wall

For the zeroth mode, the equation for total pressure is reduced to

$$\partial_z \hat{p}(\mathbf{k} = 0, z) = \hat{f}_z(\mathbf{k} = 0, z), \quad (23.29)$$

² The system can be written in matrix form and partially precomputed (via matrix inversion) so that solving the slit correction at runtime amounts to a simple 8x8 matrix-vector multiplication. The related functions in [UAMMD](#) can be found in the file *StokesSlab/Correction.cuh*.

the solution of which is

$$\hat{p}(\mathbf{k} = 0, z) = \int_{-H}^z \hat{\mathbf{f}}(\mathbf{k} = 0, z') dz' + C. \quad (23.30)$$

We can see that the constant $C = 0$ by choosing $\hat{p}(\mathbf{k} = 0, z = -H) = 0$ and write a **BVP** for the pressure that can be solved as usual,

$$\partial_z^2 \hat{p}(\mathbf{k} = 0, z) = \partial_z f_z(\mathbf{k} = 0, z), \quad (23.31)$$

with **BCs** given by

$$\hat{p}(\mathbf{k} = 0, z = -H) = 0, \quad \hat{p}(\mathbf{k} = 0, z = H) = \int_{-H}^H \hat{\mathbf{f}}(\mathbf{k} = 0, z') dz'. \quad (23.32)$$

On the other hand, the in-plane velocity satisfies

$$\partial_z^2 \hat{\mathbf{v}}^\parallel = -\eta^{-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \hat{f}_z. \quad (23.33)$$

For the **BCs** we can use the fact that the bottom wall imposes a no-slip condition to elucidate $\hat{\mathbf{v}}^\parallel(\mathbf{k} = 0, z = -H) = 0$. Given that the forces are bound to the inside of the domain, we can assume that $\hat{\mathbf{v}}^\parallel(\mathbf{k} = 0, z \geq H)$ is constant. Using the continuity of its derivative at the domain limit, we can obtain the second **BC** as $\partial_z \hat{\mathbf{v}}^\parallel(\mathbf{k} = 0, z = H) = 0$.

Finally, the equation for the perpendicular velocity when $\mathbf{k} = 0$ is simplified to

$$\partial_z \hat{\mathbf{v}}^\perp(\mathbf{k} = 0, z) = 0. \quad (23.34)$$

Making use of the no-slip wall, we can simply choose $\hat{\mathbf{v}}^\perp(\mathbf{k} = 0, z) = 0$.

Slit channel

Since the difference between the boundary conditions of the slit channel and bottom wall cases only affect the velocity at the boundaries, the computation of the pressure is equivalent in both cases. Furthermore, since for the bottom wall we chose $\hat{v}_z(\mathbf{k} = 0, z) = 0$ just by using the no-slip condition at the bottom wall, this component is also equivalent³.

³ Any other value would result in a net flow across the no-slip walls, which is of course impossible.

The [BVP](#) for the parallel velocities can be laid out following similar arguments as with the bottom wall case. In particular, the zeroth mode satisfies

$$\partial_z^2 \hat{\mathbf{v}}^\parallel(\mathbf{k} = 0, z) = -\eta^{-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \hat{f}_z. \quad (23.35)$$

For the [BCs](#), due to the no-slip condition, we have $\hat{\mathbf{v}}^\parallel(\mathbf{k} = 0, z = \pm H) = 0$.

23.4 ON SPREADING AND INTERPOLATION

Using the Chebyshev basis to discretize the z direction makes the grid of our solver non-regular. In particular, we will have a regular binning in the plane directions and bins defined at the extrema of the Chebyshev polynomials in z .

Similarly to all the Force Coupling Method-like methods described thus far, the communication between the particles information and the grid (in this case to construct \mathbf{f} in Eq. (23.1) and interpolate the velocities back to the particle positions) is carried out via the Immersed Boundary module in UAMMD (see chapter 20.5). Note however, that there is a fundamental difference with respect to the rest of the use cases for the [IBM](#) up to this point. In particular, now the grid is not regular in the z direction, being a Chebyshev grid instead (see Fig. 23.2).

The Chebyshev spacing in the z direction has several consequences with respect to our usual regular grid. First, the non-regularity of the grid makes the number of support cells of the kernel vary with the height of the particle. Secondly, the quadrature weights (i.e. the sizes of each cell) are different for each height. In particular, we have to use the so-called Clenshaw-Curtis weights [129], so that a cell centered at the Chebyshev root $z = H \cos(\pi i / N_z)$ will have a quadrature weight, $w(z)$, of⁴

$$w(z = H \cos(\pi i / N_z)) = \frac{2}{N_z} \sum_{k=0}^i \cos\left(\frac{2ik\pi}{N_z}\right) \begin{cases} 1/2 & i = 0, N_z \\ 1 & \text{otherwise} \end{cases} \quad (23.36)$$

In the plane directions, the quadrature weights are the cell sizes, h , and are the same everywhere. The [UAMMD IBM](#) interface can be

⁴ The [UAMMD](#) source code containing the special considerations for Chebyshev grids can be found at `src/utils/ChebyshevUtils.cuh`.

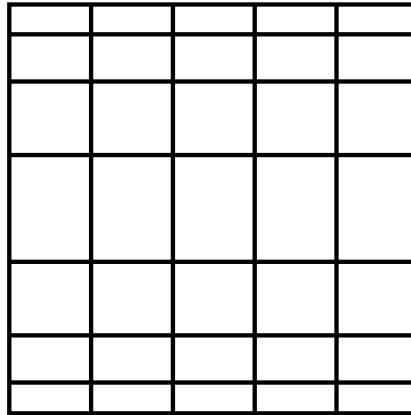


Figure 23.2: Representation of a hybrid regular-Chebyshev grid. In the Doubly Periodic Stokes algorithm, the plane-parallel components (x and y) are described on a regular grid (with nodes at $x = -L/2 + i/NL$ and similarly for y), whereas the perpendicular direction (z) is discretized at the Gauss-Chebyshev-Lobatto points (the extrema of the Chebyshev polynomials), $z = H \cos(\pi i/N_z)$ (where i runs from 0 to N_z).

customized (via templates) to accept arbitrary quadrature weights and non regular grids as well as different supports for each particle according to its position.

Furthermore, given that the domain is not periodic in z , we need special considerations when spreading and interpolating near the boundaries. In the free-space case we can simply define the limits of the domain far enough from the particles so that $\mathbf{f}(z \notin [-H, H]) = 0$. However, in the presence of a wall we must ensure that the force goes smoothly to zero at the wall. We enforce this by subtracting the envelope of a given particle centered at its image about the wall. We can thus redefine the kernel close to the wall⁵ as

$$\tilde{\delta}_a(\mathbf{r} - \mathbf{q}_i) = \delta_a(\mathbf{r} - \mathbf{q}_i) - \delta_a(\mathbf{r} - \mathbf{q}_i^{\text{img}}), \quad (23.37)$$

where $\mathbf{q}_i^{\text{img}}$ is the particle's point of reflection about the wall, defined as

$$\begin{aligned} \mathbf{q}_i^{\text{img}_b} &= \mathbf{q}_i - 2(H + r_z)\hat{\mathbf{e}}_z, & \text{For the bottom wall,} \\ \mathbf{q}_i^{\text{img}_t} &= \mathbf{q}_i + 2(H - r_z)\hat{\mathbf{e}}_z, & \text{For the top wall.} \end{aligned} \quad (23.38)$$

⁵ Closer than the kernel's support, which effectively means everywhere.

Here \hat{e}_z is a unitary vector in the z direction. The no-slip condition is then imposed by the fact that a particle at the height of a wall does not affect (nor is it affected by) the fluid. A visual representation of the effect of the images is available in Fig. 23.3.

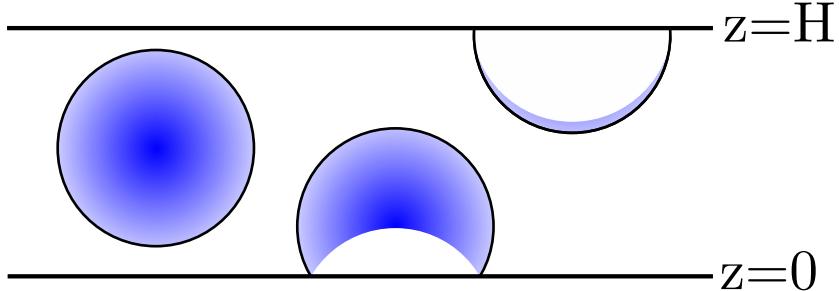


Figure 23.3: Representation of the image spreading in the DP Stokes algorithm. Forces acting on particles (blue circles) are translated to the fluid as a force density. Due to images, part of the particle's force is zero near walls. Zero fluid forcing is depicted as white. A particle located at exactly the height of a wall has no effect on the fluid.

USE IN UAMMD

At the time of writing, the Doubly Periodic Stokes module is still in development (although in its final stages) and thus its interface is bound to change in the near future. Furthermore, its applicability is hindered by the absence of fluctuations, which make this module not suitable for Brownian Dynamics simulations in its current state (note, however, that the addition of fluctuations is a work in progress). For the time being this module is not available as an integrator and can only be used to compute the hydrodynamic displacements of a series of particles given the forces and/or torques acting on them. The heuristics for the different parameters required by the module are still being actively tested and developed, so many implementation details leak into the parameter list. In particular, the BM kernel parameters (mainly β and the support, w) and the grid size information must be provided.

```

#include
→ <Integrator/BDHI/DoublyPeriodic/DPStokesSlab.cuh>
using namespace uammd::DPStokesSlab_ns;
auto createDPStokesModule(Parameters par){
    DPStokes::Parameters par;
    par.nx      = par.nx;
    par.ny      = par.%;
    par.nz      = par.nz;
    par.viscosity = par.viscosity;
    par.Lx      = par.Lx;
    par.Ly      = par.Ly;
    par.H       = par.H;
    par.w = par.w; //support for the forces
    par.beta = par.beta; //beta for the forces
    par.w_d = par.w_d; //suport for the torques
    par.beta_d = par.beta_d; //beta for the torques
    par.mode = WallMode::none; //Can also be bottom or
    → slit
    auto dpstokes = std::make_shared<DPStokes>(par);
    return dpstokes;
}

auto computeHydrodynamicDisplacements(UAMMD sim,
→ std::shared_ptr<DPStokes> dpstokes){
    auto pos = sim.pd->getPos(access::gpu,
    → access::read);
    auto force = sim.pd->getForce(access::gpu,
    → access::read);
    auto torques = sim.pd->getTorque(access::gpu,
    → access::read);
    int numberParticles = pos.size();
    //The forces or torques can be replaced by a
    → nullptr, which will spare the related
    → computations.
    auto displacements = dpstokes->Mdots(pos.begin(),
    → force.begin(), torques.begin(),
    → numberParticles);
    //The result of Mdots contains the linear and
    → dipolar displacements:
    //auto MF = displacements.first; //linear
    → displacements
    //auto MT = displacements.second; //angular
    → displacements
    return displacements;
}

```

Source Code 28: Using the DPStokes module for doubly periodic hydrodynamics.

24

TRIPLY PERIODIC ELECTROSTATICS

We are going to describe a fast spectral solver for the Poisson equation with periodic boundary conditions and Gaussian sources of charge with arbitrary widths centered at the particles' locations. Our approach is similar to the one presented at [105]. However, the authors of [105] make use of the so-called Fast Gaussian Gridding (FGG) to accelerate spreading/interpolation while UAMMD's implementation uses the algorithms laid out in chapter 21. While FGG reduces the number of evaluations of the kernel, it is restricted to the Gaussian one and, since our implementation is generic for any kernel, we deem FGG unworthy for our purposes. The Ewald splitting framework, already introduced in chapter 20.2 for the Stokes equation, follows closely from the mathematical machinery laid out in [130]. One notable thing about the following algorithm is that it can be seen as merely a reinterpretation of terms in the non-fluctuating version of the **FCM**¹. In general, we want to solve the Poisson equation in a periodic domain in the presence of a charge density $f(\mathbf{r} = (x, y, z))$,

$$\varepsilon \Delta \phi = -f. \quad (24.1)$$

Here ε represents the permittivity of the medium and f accounts for N Gaussian charges of strength Q_i located at \mathbf{q}_i ,

$$f(\mathbf{r}) = \mathcal{S}(\mathbf{r})\mathbf{Q} = \sum_i Q_i \delta_a(||\mathbf{r} - \mathbf{q}_i||). \quad (24.2)$$

Let us denote with the vector containing all charges as $\mathbf{Q} = \{Q_1, \dots, Q_N\}$. We will use the spreading operator, \mathcal{S} , (introduced in chapter 18) to transform the particles charges to a smooth charge density field. We use the Gaussian kernel,

$$\delta_a(\mathbf{r}) = \frac{1}{(2\pi a^2)^{3/2}} \exp\left(\frac{-r^2}{2a^2}\right), \quad (24.3)$$

¹ In fact, UAMMD's implementation of triply periodic electrostatics shares most of the code with the **FCM** implementation. Mainly the spreading/interpolation and the **FFT**.

Identifying a as the width the charges (notice that the case $a \rightarrow 0$ corresponds to point charges). Once Eq. (24.1) is solved we have the value of the potential in every point in space and can be evaluated at the charges locations via the interpolation operator (introduced in chapter 18)

$$\phi_{\mathbf{q}_i} = \mathcal{J}_{\mathbf{q}_i} \phi = \int Q_i \delta_a(\mathbf{q}_i - \mathbf{r}) \phi(\mathbf{r}) d\mathbf{r}. \quad (24.4)$$

Here \mathcal{J} represents the interpolation operator, that averages a quantity defined in space to a charge's location. The electrostatic energy can be computed as

$$U = \frac{1}{2} \sum_i \phi_{\mathbf{q}_i}. \quad (24.5)$$

In a similar way we compute the electrostatic force, $\mathbf{F}_i = -Q_i \nabla_i \phi$, acting on each charge from the electric field

$$\mathbf{E} = -\nabla \phi. \quad (24.6)$$

Interpolating again we get

$$\mathbf{E}_i = \mathcal{J}_{\mathbf{q}_i} \mathbf{E}(\mathbf{r}). \quad (24.7)$$

Thus, the electrostatic force acting on particle i is

$$\mathbf{F}_i = Q_i \mathcal{J}_{\mathbf{q}_i} \mathbf{E}. \quad (24.8)$$

Given that Eq. (24.3) has in principle an infinite support evaluating Eq. (24.2) at every point in space, as well as computing the averages of the electric potential and field in Eqs. (24.5) and (24.7) can be highly inefficient. In practice we overcome this limitation by truncating Eq. (24.3) at a certain distance according to a desired tolerance.

Basic Algorithm

Eq. (24.1) can be easily solved in Fourier space by convolution with the Poisson's Greens function,

$$\hat{\phi}(\mathbf{k}) = \frac{\hat{f}(\mathbf{k})}{\varepsilon k^2}. \quad (24.9)$$

We can reuse the methodological machinery devised for the **FCM** (see chapter 20.1) identifying $\hat{\mathcal{G}} := \frac{1}{k^2}$ as the Green's function and

interpreting the viscosity as the permittivity. Naturally, in the current case fluctuations are not present.

The electric field can be derived from the potential in fourier space via ik differentiation,

$$\hat{\mathbf{E}} = i\mathbf{k}\hat{\phi}. \quad (24.10)$$

Similarly to the section on [FCM](#), Eq. (24.9) can be discretized using 3D [FFT](#) in a grid with spacing fine enough to resolve the Gaussian charges in Eq. (24.2).

The whole algorithm, going from particle charges to forces, can be summarized as follows

1. Spread charges to the grid, $f = \mathcal{S}Q$.
2. Fourier transform $\hat{f} = \mathfrak{F}f$.
3. Multiply by the Poisson's Greens function to obtain the potential, $\hat{\phi} = \frac{\hat{f}}{\varepsilon k^2}$.
4. Compute field via ik differentiation, $\hat{\mathbf{E}} = i\mathbf{k}\hat{\phi}$.
5. Transform potential and field back to real space $\phi = \mathfrak{F}^{-1}\hat{\phi}$; $\mathbf{E} = \mathfrak{F}^{-1}\hat{\mathbf{E}}$.
6. Interpolate energy and/or force to charge locations, $\phi_i = \mathcal{J}\phi$; $\mathbf{E}_i = \mathcal{J}_{q_i}\mathbf{E}$.

24.1 EWALD SPLITTING

The main problem with the approach in the previous section is that the grid needs to be fine enough to correctly describe the Gaussian sources. Thus a small width results in a high number of grid cells. This hinders the ability to simulate large domains (in terms of a) or narrow (or even point) sources. In order to overcome this limitation we use an Ewald splitting technique [130] (reminiscent of the use case introduced with [PSE](#) in chapter 20.2).

We can write the potential as

$$\phi = (\phi - \gamma^{1/2} \star \psi) + \gamma^{1/2} \star \psi = \phi^{\text{near}} + \phi^{\text{far}}, \quad (24.11)$$

where \star represents convolution and the intermediate solution ψ satisfies

$$\varepsilon\Delta\psi = -f \star \gamma^{1/2}. \quad (24.12)$$

The splitting function γ is defined as

$$\gamma^{1/2} = \frac{8\xi^3}{(2\pi)^{3/2}} \exp(-2r^2\xi^2). \quad (24.13)$$

Here the splitting parameter, ξ , is an arbitrary factor that is chosen to optimize performance. Given that the Laplacian commutes with the convolution we can divide the problem in two separate parts, denoted as near and far field

$$\varepsilon\Delta\phi^{\text{far}} = -f \star \gamma, \quad (24.14)$$

$$\varepsilon\Delta\phi^{\text{near}} = -f \star (1 - \gamma). \quad (24.15)$$

The convolution of two Gaussians is also a Gaussian, so in the case of the far field the RHS results in wider Gaussian sources that can be interpreted as smeared versions of the original ones. The far field RHS thus decays exponentially in Fourier space and is solved as in the non Ewald split case by effectively modifying the width of the Gaussian sources from a to

$$g_t = \sqrt{\frac{1}{4\xi^2} + a^2}. \quad (24.16)$$

Notice that this overcomes the difficulties for the case of point sources, since we can arbitrarily increase ξ to work with an arbitrarily large Gaussian source.

In contrast the near field effective charges are sharply peaked and narrower than the originals, rapidly decaying to zero. We can compute the Green's function for Eq. (24.15) analytically by convolving the original Poisson Green's function with the effective charges in Eq. (24.15) in Fourier space, yielding

$$\hat{\mathcal{G}}^{\text{near}}(k) = \frac{(1 - \hat{\gamma})\hat{f}}{\varepsilon k^2}. \quad (24.17)$$

The inverse transform of this kernel gives us the potential at any point of the grid as

$$\phi^{\text{near}}(\mathbf{r}) = \sum_i Q_i \mathcal{G}^{\text{near}}(||\mathbf{r} - \mathbf{q}_i||). \quad (24.18)$$

However, we are only interested in the potential averaged at the charges locations. Instead of storing and computing the potential in a grid and then interpolating as in the far field we can compute

this analytically. Given that $\mathcal{J}\phi^{\text{near}} = f \star \phi^{\text{near}}$ we can define a pre-interpolated interaction kernel as

$$\hat{\mathcal{G}}_{\mathcal{J}}^{\text{near}}(k) = \hat{f} \hat{\mathcal{G}}^{\text{near}} = \frac{(1 - \gamma)\hat{f}^2}{\varepsilon k^2}. \quad (24.19)$$

This expression is radially symmetric, which allows to compute the inverse Fourier transform as

$$\begin{aligned} \mathcal{G}_{\mathcal{J}}^{\text{near}}(r) &= \frac{1}{2\pi^2 r} \int \hat{\mathcal{G}}_{\mathcal{J}}^{\text{near}} k \sin(kr) dk \\ &= \frac{1}{4\pi\epsilon r} \left(\operatorname{erf}\left(\frac{r}{2a}\right) - \operatorname{erf}\left(\frac{r}{2g_t}\right) \right), \end{aligned} \quad (24.20)$$

which decays exponentially and can be truncated at a certain cut off radius r_c . Furthermore this expression requires special consideration at short distances where numerical issues could arise. In these cases the Taylor expansion of Eq. (24.20) can be used instead. We can use this expression to compute the potential or energy at the charges locations as

$$U_i = Q_i \mathcal{J}_{\mathbf{q}_i} \phi^{\text{near}}(\mathbf{r}) = Q_i \sum_j Q_j \mathcal{G}_{\mathcal{J}}^{\text{near}}(||\mathbf{q}_i - \mathbf{q}_j||). \quad (24.21)$$

Similarly we compute the electric field or force acting on each charge,

$$\mathbf{F}_i = Q_i \mathcal{J}_{\mathbf{q}_i} \mathbf{E}^{\text{near}} = Q_i \sum_j Q_j \partial_r G_{\mathcal{J}}^{\text{near}}(r_{ij}) \frac{\mathbf{r}_{ij}}{r_{ij}}, \quad (24.22)$$

where $\mathbf{r}_{ij} = \mathbf{q}_i - \mathbf{q}_j$ and $r_{ij} = ||\mathbf{r}_{ij}||$. The derivative of Eq. (24.20) can be computed analytically [131].

Accuracy

There are several parameters than can be tweaked to control the overall accuracy of the algorithm: The grid cell size $h := L/n$ (being L the box size and n the number of grid cells²) controls how fine the Gaussian sources are described (providing a cut off wave number for the fourier description of the Poisson's Green's function). Empirically I have found that $h = [1.3 - \min(-0.1 \log_{10}(\epsilon), 0.9)] g_t$ ensures enough accuracy to ensure a relative error below the desired

² A cubic box is considered, but the arguments can be extended easily to any domain dimensions.

tolerance, ϵ . Note that n should be chosen to be an FFT-friendly number (see Appendix B).

On the other hand, we also truncate the Gaussian kernel, ϕ_G , up to a certain distance, r_c , such that $\phi_G(r_c) < \epsilon\phi_G(0)$.

A cut off distance is also required for the near field, where we choose r_c^{nf} such that $\mathcal{G}_{\mathcal{T}}^{\text{near}}(r_c) < \epsilon\mathcal{G}_{\mathcal{T}}^{\text{near}}(0)$.

Use in UAMMD

This algorithm is exposed in UAMMD via the *SpectralEwaldPoisson Interactor* module.

```
#include <uammd.cuh>
#include <Interactor/SpectralEwaldPoisson.cuh>
using namespace uammd;
//Creates and returns a triply periodic Poisson
//solver Interactor
auto createTPPoissonInteractor(UAMMD sim){
    Poisson::Parameters par;
    par.box = sim.par.box;
    //Permittivity
    par.epsilon = sim.par.epsilon;
    //Gaussian width of the sources
    par.gw = sim.par.gw;
    //Overall tolerance of the algorithm
    par.tolerance = sim.par.tolerance;
    //If a splitting parameter is passed
    // the code will run in Ewald split mode
    //Otherwise, the non Ewald version will be used
    //par.split = 1.0;
    return std::make_shared<Poisson>(sim.pd, par);
}
```

Source Code 29: Usage example of the triply periodic Poisson module.

The tolerance parameter is the maximum relative error allowed in the potential for two charges. The potential for $L \rightarrow \infty$ is extrapolated and compared with the analytical solution. Also in

25

Ewald split mode the relative error between two different splits is less than the tolerance.

DOUBLY PERIODIC ELECTROSTATICS

We present a novel algorithm for computing the electrostatic energies and forces for a collection of charges in a doubly-periodic environment (a slab). Our algorithm can account for arbitrary dielectric jumps across the boundaries of the slab and an arbitrary distribution of surface charge at the domain walls (in the open direction).

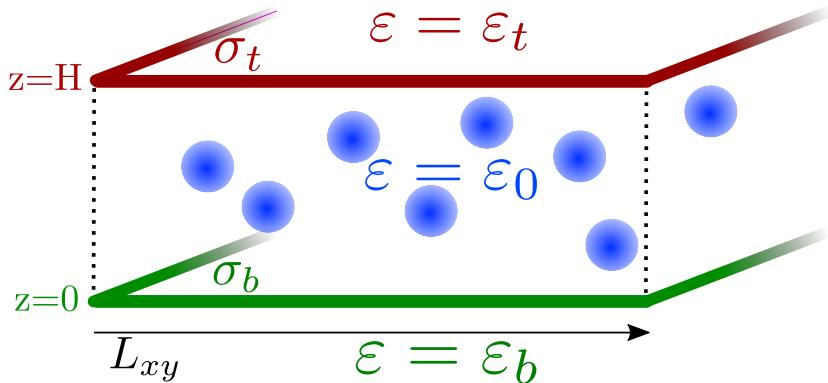


Figure 25.1: Schematic representation of the doubly periodic domain described by Eqs. 25.1 and 25.3-25.6. Each domain wall is represented with a different color. Blue clouds represent Gaussian charge sources.

A complete description of the algorithm can be found in [131]. We model charges as Gaussian sources (as opposed to point charges). Even so our algorithm provides spectral accuracy via Ewald splitting, which naturally decouples the width of the sources and the grid size. This allows us to use an arbitrarily narrow Gaussian to simulate point charges if needed. By using Gaussian charges we avoid divergent self-interactions while more closely modelling electrolyte solutions, in which solvated ions are not point charges but rather have a certain characteristic charge size that we can mimic via the Gaussian width. We want to solve the Poisson equa-

tion with Gaussian charge sources in a doubly periodic domain of width H and size L_{xy} in the plane. In particular, we seek to solve

$$\varepsilon_0 \Delta \phi(\mathbf{x} = (x, y, z)) = -f(\mathbf{x}) = -\mathcal{S}(\mathbf{x})\mathbf{Q}, \quad (25.1)$$

where

$$\mathcal{S}(\mathbf{x})\mathbf{Q} = \sum_i Q_i \left(2\pi a^2\right)^{-3/2} \exp\left(-\frac{\|\mathbf{q}_i - \mathbf{x}\|^2}{2a^2}\right), \quad (25.2)$$

where a is the width of the Gaussian charges with strength Q_i ¹. We impose that the sources do not overlap the boundaries in the z direction, $f(z > H \text{ or } z < 0) = 0$, so that the charge density integrates to one inside the slab. Given that the Gaussian is not compactly supported we truncate it at $n_\sigma a \geq 4a$ to overcome this, ensuring that the integral is at least 99.9% of the charge Q .

Finally, we solve (25.1) with the following set of BCs for the potential

$$\phi(x, y, z \rightarrow 0^+) = \phi(x, y, z \rightarrow 0^-) \quad (25.3)$$

$$\phi(x, y, z \rightarrow H^-) = \phi(x, y, z \rightarrow H^+) \quad (25.4)$$

And for the electric field

$$\varepsilon_0 \frac{\partial \phi}{\partial z}(x, y, z \rightarrow 0^+) - \varepsilon_b \frac{\partial \phi}{\partial z}(x, y, z \rightarrow 0^-) = -\sigma_b(x, y) \quad (25.5)$$

$$\varepsilon_0 \frac{\partial \phi}{\partial z}(x, y, z \rightarrow H^-) - \varepsilon_t \frac{\partial \phi}{\partial z}(x, y, z \rightarrow H^+) = \sigma_t(x, y) \quad (25.6)$$

We introduce, via these BCs, the possibility of having arbitrary surface charges at the walls, σ_b and σ_t for the bottom and top respectively. Additionally, we can set different permittivities inside the slab (ε_0) above (ε_t) and below (ε_b) it. See figure 25.1 for a representation of the described set up. Finally, we assume that the domain is overall electroneutral,

$$\sum_{k=1}^N Q_k + \int_0^{L_{xy}} \int_0^{L_{xy}} (\sigma_b(x, y) + \sigma_t(x, y)) dx dy = 0. \quad (25.7)$$

Similarly to the triply periodic algorithm described in chapter 24, we are interested in computing the system energy and the particle

¹ In UAMMD's implementation, the permittivity inside the domain is set to unity and the charges are rescaled as $\mathbf{Q}' = \mathbf{Q}/\varepsilon_0$.

forces. Once the potential is known the total electrostatic energy (inside the simulation domain) can be computed as

$$U = \frac{1}{2} \sum_i Q_i \mathcal{J}_{q_i} \phi + \frac{1}{2} \sum_{wall=b,t} \int_{x,y} \sigma_{wall} \phi(x, y, wall) dx dy \quad (25.8)$$

Where \mathcal{J} represents the interpolation operator as in chapter 24. In the same way as with the triply periodic algorithm, we can compute the electric field using Eq. (24.6) and interpolate it to the particles positions with Eq. (24.7). Finally, we compute the force acting on each particle with Eq. (24.8).

Since we do not have periodic boundaries in the z direction, the application of the nabla operator on the potential to compute the electric field is not as straightforward as in the triply periodic case. In particular, as we will study in the following sections, our solver works in the Fourier-Chebyshev space (similar to the doubly periodic Stokes solver in chapter 23).

25.1 SOLVER DESCRIPTION

We use a grid-based solver to get an algorithm with linear complexity with the number of particles. We evaluate the **Right Hand Side (RHS)** of (25.1) in a grid by spreading the charges (as in Chapter 24), then solve the potential on that grid and finally interpolate the required quantities back to the particle positions (i.e the energy in Eq. (25.8) or the force with Eq. (24.8)). In order to accurately describe the charge density $f(x)$ in the grid we must choose a sufficiently small grid size, h . Since the correct description of the charge density field produced by the Gaussian sources requires $h \sim a$ this method will be inefficient for point-like charges ($a \rightarrow 0$). To overcome this limitation (similarly to the triply periodic case described in chapter 24) we make use of Ewald splitting.

Let us start with the non-Ewald split case by considering that the distance between charges is comparable to their width, a .

In a move reminiscent of the DP Stokes algorithm in Chapter 23 we start by separating the problem into two sub-problems. First we solve (25.1) in free space (no dielectric jumps or surface charges) and then introduce a harmonic correction to account for the jump BCs (section 25.1.2). In particular, we separate the potential as

$$\phi = \phi_{DP} + \phi_{corr} \quad (25.9)$$

We first find the free-space potential, ϕ_{DP} , by solving the Poisson equation (25.1) with free-space BCs and uniform permittivity ϵ_0 (see Eq. (25.10)). Then we compute the correction, ϕ_{corr} , by solving a Laplace equation that satisfies the BCs in Eqs. (25.3)-(25.6). Notice that on the one hand the free-space potential is oblivious to the presence of the surface charged walls and, on the other, the harmonic correction does not include the Gaussian sources in its formulation. This means that each separate problem is not necessarily electroneutral, we address electroneutrality further on in section 25.1.3.

25.1.1 Free space solver, ϕ_{DP}

Let us start by describing the free-space solver. We want to solve

$$\epsilon_0 \Delta \phi_{DP}(\mathbf{r}) = -f(\mathbf{r}) \quad \text{for } 0 < z < H \quad (25.10)$$

bounded at infinity so that $\partial_z \phi_{DP}(z \rightarrow \pm\infty) \rightarrow 0$ and with periodic boundary conditions in the plane. We can use the fact that the source charges are contained inside the domain $z \in [0, H]$ to find the necessary BCs to constitute a BVP, which can then be solved by the method in Appendix C (first introduced in chapter 23 for the Stokes equation). In particular, we can write

$$\epsilon_0 \Delta \phi_{DP} = 0 \quad \text{for } z \leq 0 \quad \text{or} \quad z \geq H. \quad (25.11)$$

this equation can be solved analytically by taking its Fourier transform in the plane, so that for each wavenumber we have

$$(\partial_z^2 - k^2) \hat{\phi}_{DP}(\mathbf{k}, z) = 0. \quad (25.12)$$

Which has an analytical solution

$$\hat{\phi}_{DP}(\mathbf{k}, z \leq 0) \propto \exp(kz) \quad \text{and} \quad \hat{\phi}_{DP}(\mathbf{k}, z \geq H) \propto \exp(-kz), \quad (25.13)$$

hinting at the following BCs for Eq. (25.10),

$$(\partial_z + k) \hat{\phi}_{DP}(\mathbf{k}, H) = 0 \quad \text{and} \quad (\partial_z - k) \hat{\phi}_{DP}(\mathbf{k}, 0) = 0, \quad (25.14)$$

owing to the continuity of ϕ_{DP} and $\partial_z \phi_{DP}$ at the boundaries. We can now write Eq. (25.10) in Fourier space in the plane and couple it with the BCs in Eq. (25.14) to constitute a two point BVP that

can be efficiently solved in the **GPU** with the solver described in Appendix C. We can thus rewrite Eq. (25.10) as

$$\varepsilon_0(\partial_z^2 - k^2)\hat{\phi}_{\text{DP}}(\mathbf{k}, z) = -\hat{f}(\mathbf{k}, z) \quad \text{for } 0 \leq z \leq H. \quad (25.15)$$

The $k = 0$ mode requires special treatment, since the **BVP** in Eqs. (25.15) with **BCs** in Eq. (25.14) is not necessarily electroneutral (since it does not consider the surface charges), we will consider it later along with the correction (given that the problem for ϕ is indeed electroneutral).

25.1.2 Harmonic correction, ϕ_{corr}

We now correct the solution ϕ_{DP} , for which we did not take into account the **BCs** in Eqs. (25.3)-(25.6), mainly neglecting the surface charges $\sigma_{b/t}$. In contrast, the correction potential does not require taking into account the Gaussian sources, which were included in ϕ_{DP} . Thus, the correction will have a Laplace equation

$$\Delta\phi_{\text{corr}} = 0 \quad (25.16)$$

for all space, $z \in (-\infty, \infty)$. For the **BCs** we use the four jump conditions at the domain limits (two for the potential and two for the field). In general, we will have a mismatch at the boundaries in both the potential and the field given by

$$\begin{aligned} \phi_{\text{corr}}(x, y, z \rightarrow 0^+) - \phi_{\text{corr}}(x, y, z \rightarrow 0^-) &= -m_\phi^b(x, y), \\ \phi_{\text{corr}}(x, y, z \rightarrow H^-) - \phi_{\text{corr}}(x, y, z \rightarrow H^+) &= -m_\phi^t(x, y), \\ \varepsilon_0 \partial_z \phi_{\text{corr}}(x, y, z \rightarrow 0^+) - \varepsilon_b \partial_z \phi_{\text{corr}}(x, y, z \rightarrow 0^-) &= -m_E^b(x, y) \\ \varepsilon_0 \partial_z \phi_{\text{corr}}(x, y, z \rightarrow H^-) - \varepsilon_t \partial_z \phi_{\text{corr}}(x, y, z \rightarrow H^+) &= -m_E^t(x, y). \end{aligned} \quad (25.17)$$

Using $\phi = \phi_{\text{DP}} + \phi_{\text{corr}}$ we can write them in terms of the already computed ϕ_{DP} ,

$$\begin{aligned} m_\phi^b &= \phi_{\text{DP}}(z \rightarrow 0^+) - \phi_{\text{DP}}(z \rightarrow 0^-) = 0, \\ m_\phi^t &= \phi_{\text{DP}}(z \rightarrow H^-) - \phi_{\text{DP}}(z \rightarrow H^+) = 0, \\ m_E^b &= (\varepsilon_0 - \varepsilon_b) \partial_z \phi_{\text{DP}}(z = 0) + \sigma_b, \\ m_E^t &= (\varepsilon_0 - \varepsilon_t) \partial_z \phi_{\text{DP}}(z = H) - \sigma_t. \end{aligned} \quad (25.18)$$

Where we have used the continuity of ϕ_{DP} and its derivative at the domain limits to simplify the expressions for the field mismatch.

Additionally, the potential mismatch is zero in the current case, nonetheless we keep it in the description, since it will become non-zero in the Ewald split case (see section 25.2).

We can write the general solution for the correction potential inside the domain in Fourier space (as usual, transformed only in the plane) as

$$\hat{\phi}_{\text{corr}}(\mathbf{k}, z) = A(\mathbf{k}) \exp(kz) + B(\mathbf{k}) \exp(-kz), \quad \text{for } 0 \leq z \leq H \quad (25.19)$$

Using the BCs in Eq. (25.17) it can be shown that the solution is

$$\begin{aligned} \hat{\phi}_{\text{corr}}(\mathbf{k}, z) = \alpha(k) & \left[(\varepsilon_t + 1) \exp(-kz) (\hat{m}_E^b - \varepsilon_b k \hat{m}_\phi^b) \right. \\ & - (\varepsilon_b + 1) \exp(k(z - H)) (\hat{m}_E^t + \varepsilon_t k \hat{m}_\phi^t) \\ & + (\varepsilon_b - 1) \exp(-k(H + z)) (\varepsilon_t k \hat{m}_\phi^t + \hat{m}_E^t) \\ & \left. - (\varepsilon_t - 1) \exp(k(z - 2H)) (\hat{m}_E^b - \varepsilon_b k \hat{m}_\phi^b) \right]. \end{aligned} \quad (25.20)$$

Where we have defined for convenience the helper function, $\alpha(k)$, as

$$\alpha(k) := (k(\varepsilon_b + 1)(\varepsilon_t + 1) - (\varepsilon_b \varepsilon_t + \varepsilon_b + \varepsilon_t - 1) \exp(-2Hk))^{-1} \quad (25.21)$$

The expression for the correction potential in Eq. (25.20) has been carefully laid out to minimize overflow errors in a numerical implementation (due to the evaluation of the exponential terms). In particular, UAMMD's implementation evaluates this expression in double precision regardless of the overall precision mode, since the standard math library allows for a maximum exponent of 709. Exponential overflow is not an issue in the non-Ewald split case, but as we will see in the following sections, the Ewald-split case requires evaluating Eq. (25.20) below $z = -H_e$, which results in the term $\exp(kH_e)$ being evaluated.

The only missing piece in our description is the $k = 0$ mode, which must be computed for directly for ϕ , since this is the only way to ensure the solution is well defined (which requires electroneutrality, i.e the integral of the overall charge being zero).

25.1.3 The $k = 0$ mode

For the zeroth mode we can write the equation for the uncorrected potential as

$$\partial_z^2 \hat{\phi}_{\text{DP}}(\mathbf{k} = \mathbf{0}, z) = -\varepsilon_0^{-1} \hat{f}(\mathbf{0}, z). \quad (25.22)$$

Which can be solved by integrating the zeroth mode of the charge density twice, making use of the Chebyshev coefficients of \hat{f} . Alternatively, our **BVP** solver can handle Eq. (25.15) with the (homogeneous) **BCs** in Eq. (25.14) in the case $k = 0$ ². The potential in Eq. (25.22) is only valid up to a linear correction, given by the solution of Eq. (25.19) for $k = 0$,

$$\hat{\phi}_{\text{corr}}(\mathbf{0}, 0 \leq z \leq H) = A_0 z + B_0. \quad (25.23)$$

Let us also define the correction potential outside the boundaries as

$$\hat{\phi}_{\text{corr}}(\mathbf{0}, z > H) = A_0^t z + B_0^t \quad \text{and} \quad \hat{\phi}_{\text{corr}}(\mathbf{0}, z < 0) = A_0^b z + B_0^b. \quad (25.24)$$

The final potential, $\phi(\mathbf{0}, z) = \phi_{\text{DP}}(\mathbf{0}, z) + \phi_{\text{corr}}(\mathbf{0}, z)$ must satisfy the **BCs** in Eqs. (25.3)-(25.6), replacing the equations for the partial potentials and using the mismatches yields a system of equations for the unknown coefficients as

$$\hat{\phi}_{\text{DP}}(\mathbf{0}, 0) + B_0 = B_0^b = 0, \quad (25.25)$$

$$A_0 H + B_0 + \hat{\phi}_{\text{DP}}(\mathbf{0}, H) = A_0^t H + B_0^t, \quad (25.26)$$

$$m_E^b + \varepsilon_0 A_0 - \varepsilon_b A_0^b = 0, \quad (25.27)$$

$$m_E^t + \varepsilon_0 A_0 - \varepsilon_t A_0^t = 0. \quad (25.28)$$

We only need the values of A_0 and B_0 in order to compute $\hat{\phi}(\mathbf{0}, z)$ inside the slab. We can obtain B_0 from Eq. (25.25). For A_0 we can use the decay of the field outside the slab, which requires that the correction potentials outside the boundaries cancel the one produced by $\hat{\phi}_{\text{DP}}$, so

$$A_0^b = -\partial_z \hat{\phi}_{\text{DP}}(\mathbf{0}, z \leq 0), \quad \text{and} \quad A_0^t = -\partial_z \hat{\phi}_{\text{DP}}(\mathbf{0}, z \geq H). \quad (25.29)$$

² Which is how **UAMMD** computes the linear mode.

Using the above expressions we can arrive at two solutions, which are identical for electroneutral slabs, for A_0 ,

$$\begin{aligned} A_0 &= -\varepsilon_0^{-1}(m_E^b + \varepsilon_b \partial_z \hat{\phi}_{\text{DP}}(\mathbf{0}, 0)) = \\ &= -\varepsilon_0^{-1}(m_E^t + \varepsilon_t \partial_z \hat{\phi}_{\text{DP}}(\mathbf{0}, H)). \end{aligned} \quad (25.30)$$

In [UAMMD](#)'s implementation we average both expressions to minimize the discretization errors³

25.1.4 Wrapping up

Once the correction is evaluated for all wavenumbers (except the zeroth mode, which as we have seen is handled independently), we apply the [FCT](#) to each one, obtaining the Chebyshev coefficients that can then be summed to the uncorrected solution, $\hat{\phi}_{\text{DP}}$ to finally get the total potential, ϕ , via Eq. (25.9).

In order to compute the electric field we can use ik differentiation in the plane directions and use the Chebyshev coefficients in z for each wavenumber to compute its derivative (see Appendix C).

Then we apply the inverse Fourier-Chebyshev transform to get the potential and/or field, finally interpolating to the charges positions (see Eq. (24.7) and (24.5)).

25.2 EWALD SPLITTING

In order to reuse the Ewald splitting machinery laid out for the triply periodic case (see chapter 24) we must shift our approach and describe the problem using image charges instead of boundary conditions.

The method of images

In the presence of a single wall with a jump in permittivity (for instance, the bottom wall at $z = 0$, below of which the permittivity is ε_b), the method of images [132] states that the potential (and field) created by a charge, q , at the location, (x, y, z) , is equivalent to that of a medium with uniform permittivity, ε_0 , containing the

³ In [UAMMD](#), the code for the zeroth mode correction can be found in the mismatch compute source file, *PoissonSlab/MismatchCompute.cuh*.

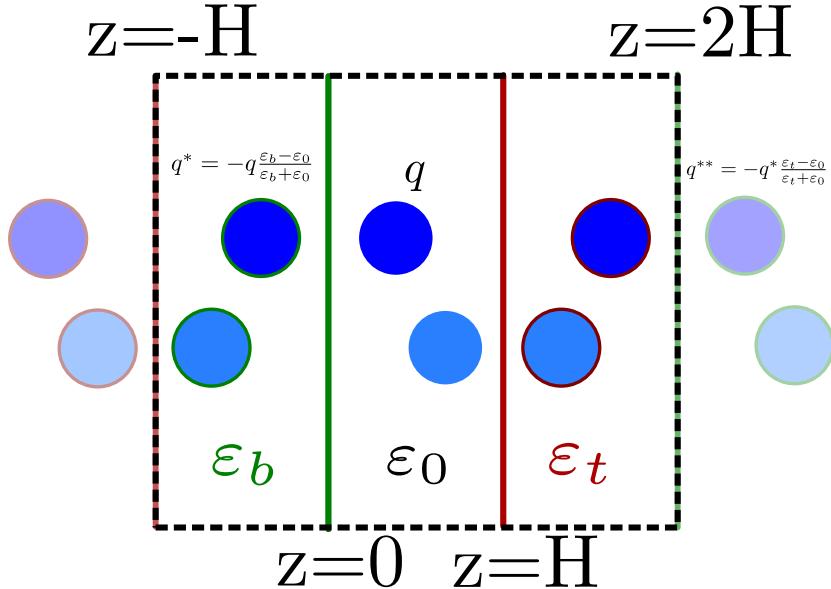


Figure 25.2: Enforcing the dielectric jumps using images. The presence of the two dielectric boundaries (at $z = 0$ and $z = H$) causes each charge to have infinite reflections. For instance, the charge q reflects through the bottom with an effective charge q^* , which in turn reflects again through the top wall with q^{**} . We manage to Ewald split the problem into a near and far field contributions in such a way that both parts only need to take into account the first reflections at most.

original charge along with an image charge reflected across the wall, $(x, y, -z)$, with charge

$$q^* = -q \frac{\varepsilon_b - \varepsilon_0}{\varepsilon_b + \varepsilon_0}. \quad (25.31)$$

Additionally, the potential and field at the position of the image charge, $(x, y, -z)$, is equivalent to the medium having permittivity ε_0 in the presence of a single charge, located at (x, y, z) with charge

$$q_{\text{img}}^* = q \frac{2\varepsilon_0}{\varepsilon_b + \varepsilon_0}. \quad (25.32)$$

With two walls the situation becomes much more complex, since the image charges further reflect through the opposite wall (see Fig. 25.2) and so on and so forth, resulting in infinite images. The image construction enforces the BCs in Eqs. (25.3)-(25.6) without

surface charges (which we will introduce later) even when the particles are Gaussian clouds instead of point-like charges. In the following discussion, we will see how to deal with images when we split the problem into a near and far field contributions (using the Ewald-splitting tools in chapter 24).

25.2.1 Near field

As discussed in sec 24.1, the range of the near field Green's function is related to the splitting parameter, ξ , and we can truncate it at a certain distance, r_c , in order to meet a certain tolerance requirement. We can take advantage of this and truncate G_J^{near} to limit the near field computation to just the first set of images above or below the walls. We can achieve this by imposing $r_c < H$, we further restrict $r_c < L_{xy}/2$ in order to apply the MIC in the plane.

Besides having to include the first images, this part of the algorithm is then identical to the triply periodic case in chapter 24.1.⁴.

25.2.2 Far field

The treatment for the far field is quite convoluted due to the requirement of solving the potential outside and inside the domain while taking into account the image charges in a spectrally accurate manner. The algorithm for the far field is carefully laid out in [131].

25.3 HOW TO USE IN UAMMD

The creation of the Doubly Periodic Poisson Interactor is similar to that of the triply periodic case. With the exception that now the box size is communicated separately in the parallel and perpendicular directions and the permittivity can be different inside and outside the domain. Besides the parameters in the source code example 30, additional ones are available to fine-tune several inter-

⁴ In UAMMD's implementation, the near field images are handled via a special *Transverser* that checks, for a given neighbour, its position and that of its two images (top and bottom wall). The related code can be found at the file *Interactor/DoublyPeriodic/PoissonSlab/NearField.cuh*.

nal precision parameters (such as support, upsampling or overall tolerance). By default, the module will provide an overall tolerance of around 4 digits, which is the study case in the original work describing the doubly periodic algorithm [131]. Additionally, a special functor can be provided specifying the surface charges. The description of the surface charge parameter is left for UAMMD's online documentation (see Appendix D). In all instances, the surface charge will enforce overall electroneutrality inside the domain. For instance, if a single positive charge of strength Q is located inside the domain, each wall will be assigned a constant charge of $-Q/2$.

```
#include <Interactor/DoublyPeriodic/DPoissonSlab.cuh>

auto createDPoissonInteractor(UAMMD sim){
    DPoissonSlab::Parameters par;
    par.Lxy = sim.par.Lxy;
    par.H = sim.par.H; //Domain height
    DPoissonSlab::Permitivity perm;
    perm.inside = sim.par.permInside;
    perm.top = sim.par.permTop;
    perm.bottom = sim.par.permBottom;
    par.permitivity = perm;
    par.gw = sim.par.gw; //Width of the Gaussian
    // sources
    par.split = gw*0.1; //Splitting parameter
    auto poisson = make_shared<DPoissonSlab>(sim.pd,
    // par);
    return poisson;
}
```

Source Code 30: Usage example of the doubly periodic Poisson module.

Part V

NEW PHYSICS AND APPLICATIONS

Scientific publications using UAMMD.

In this part we showcase the works, published in scientific journals, that resulted from the development of this thesis. We will see new physics and works in which UAMMD played a fundamental role either as a simulation engine or accelerator. This section does not intend to be self-contained (the readers are referred to the published material), but is rather a way to illustrate the capabilities of UAMMD in disparate physical scenarios.

Besides the works presented in this part of the manuscript, it is worth mentioning [133], a recent publication in which the author of this manuscript appears as a collaborator, in which UAMMD acts as an external accelerator library. In particular, an already established, in-house code written in python by the author for simulations of dynamically cross-linked actin networks offloads the construction of a neighbour list to UAMMD (using the cell list). Thanks to the inclusion of UAMMD, the previous bottle-neck of their implementation became effectively instantaneous and allowed them to reach larger simulation times and systems.

26

MEASURING INTRACELLULAR VISCOSITY

In [134] we use PSE (chapter 20.2) to model the environment of a cell. In particular, we study how the presence of microtubulae affects the viscosity measured by a nanorocker marker. The experimental collaborators in [134] were testing the effects of two drugs, colchicine and Taxol, on the internal structure of HeLa cells (which are 20-40 μm in diameter). These drugs are so-called tubulin interactors, meaning that they either promote the polymerization (Taxol) or depolymerization (colchicine) of microtubulae (i.e the cell's cytoskeleton). In particular, local mesoscopic intracellular viscosity was being probed via a non-spherical upconverting nanorocker particle (a $\beta - \text{NaYF}_4$ hexagonal-shaped nanoparticle with a 400nm thickness, see fig 26.1). The mean squared angular displacement (MSAD) of the nanorocker is sampled using a polarized spectroscopy technique that allows to track the orientation of the nanoparticle in real time [135]. The orientational fluctuations (a.i. the MSAD) of the nanorocker can then be related directly

to the local viscosity using the well-known relation for the time evolution of the MSAD for a disk-like particle,

$$\text{MSAD}(t) = \frac{k_B T}{3\eta V f/f_0} t. \quad (26.1)$$

Where V is the volume of the particle and f/f_0 is the so-called Perrin's friction coefficient, which relates the friction of the non-spherical object with that of a sphere of equivalent volume.

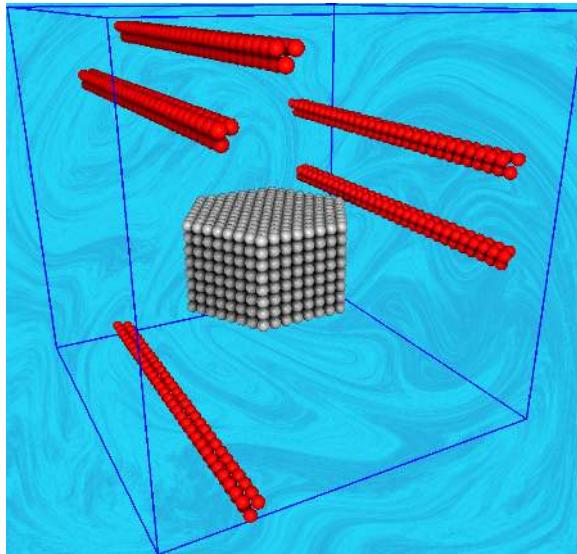


Figure 26.1: Representation of the simulation environment used to model the experimental setup in [134]. The nanorocker (grey) and the microtubulae (red) are modeled as spring-connected blobs.

Experiments with Taxol and colchicine revealed a strong and unintuitive relation between the concentration of microtubulae and the mesoscopic intracellular viscosity. In particular, experiments showed a ten-fold reduction in the viscosity after Taxol administration, corresponding to the presence of a higher concentration of microtubulae in the cellular environment. Consistently, the posterior administration of colchicine, which depolymerizes the microtubulae into a suspension of tubulin heterodimers, induces an increase in viscosity.

We infer the origin of the viscosity reduction lies in the specific sampling size of the nanorocker (with an average size of 400nm). For a typical concentration of tubulin (25-100 μM) in the cytoplasm

the volume fraction occupied by the microtubulae is between $5 \cdot 10^{-3}$ and $5 \cdot 10^{-2}$. At these concentrations we find a polydisperse ensemble of rods that form a fractal gel with pores typically smaller than the average length of the rods. A tracer particle diffusing in this scenario is known to have a severely reduced translational and rotational diffusion [136, 137]. However, after Taxol administration, the microtubule network is formed by much larger rods, resulting in the nanorocker diffusing through much larger structures. Colchicine produces the opposite effect, reducing the typical rod size and, consequently, the average length of the cytoplasm components that drag the sampling particle. Our hypothesis relies on the expectation that, when the typical distance between microtubules in the ordered phase (after Taxol administration) becomes larger than the nanorocker size the steric hidrance of the microtubulae is significantly reduced. In order to validate this, we setup a series of simulation using the PSE method (see chapter 20.2) with various concentration of microtubulae such that the average free space between rods is larger than the rocker's size (see figure 26.2).

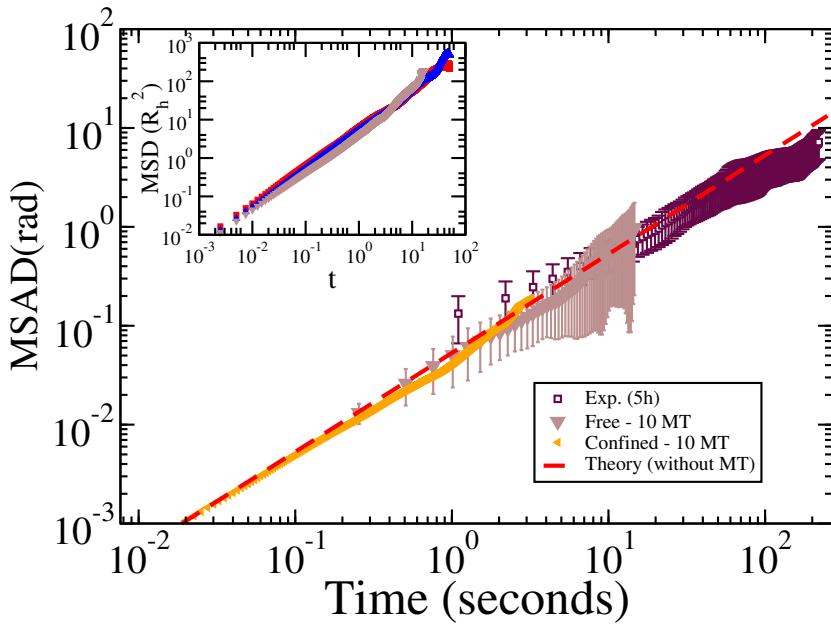


Figure 26.2: Computed and experimentally measured mean squared angular displacement (MSAD) over time. Comparison is made between simulations of free (brown line) and laser-illuminated (confined, orange line) nanorocker, surrounded by 10 microtubules (MT), and the experimental results (Exp.) after 5 h incubation time with Taxol (purple data). The red dashed line represents the MSAD calculated from Equation (1) using the measured viscosity for an incubation time of 5 h. The good agreement between the red and purple curves validates the approximation of the nanorocker to a disk. The inset shows the mean squared displacement (MSD) for free nanorockers in the presence of 0, 5, and 10 microtubules. The agreement between the experimental and simulation results indicates that the movement of the particle is not affected by the presence of microtubules since their separation is much bigger than the particle.

We perform simulations with and without taking into account the translational confining effect of the probing laser on the nanorocker (by trapping the nanorocker with a soft potential well to a domain about 10% larger than its size) and find no discernable effect on its rotational diffusion. In the confined case, any modification of the particle rotation is only due to the hydrodynamic effects of the microtubulæ. This evidences that the translational confinement of the probing beam does not affect its orientational fluctuating

motion. Our simulations yield MSAD curves that match the theory and simulations in the absence of microtubulae to within statistical uncertainty. This evidences that the presence of microtubulae, at these concentrations, has practically no effect on the viscosity sampled by the nanorocker. From this, we conclude that, once all microtubulae are formed due to the effect of Taxol, there is less amount of intracellular material of comparable size to the nanorocker, which effectively reduces the viscosity sensed by it.

STAR POLYMER DYNAMICS IN SHEAR FLOW

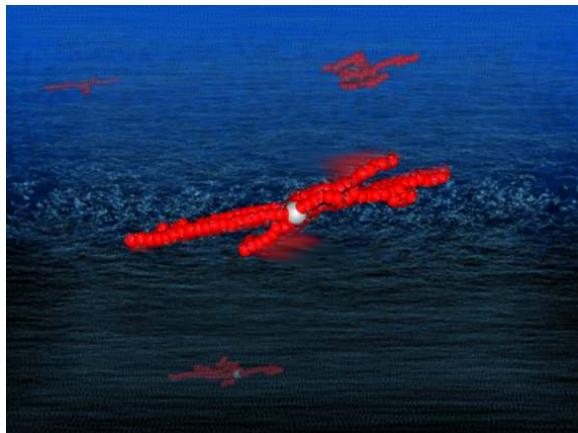


Figure 27.1: A star polymer solution under a shear flow.

In [138] we use **BDHI** via Cholesky (see chapter 19.1) to study the rheological properties of a low density solution of star polymers in shear flow. We investigate tank-treading and breathing dynamics of individual star molecules under shear flow and their relation with the macromolecule architecture. Tank-treading consists of the rotation of the arms of the star around a molecule's center, whereas breathing consists in expansions and contractions of the whole molecule at a certain characteristic frequency. We derive scaling arguments for the trends of the frequency and decorrelation rates of both rotation and breathing modes versus the shear rate $\dot{\gamma}$, which are supported by extensive Brownian hydrodynamics simulations. We find that breathing occurs if $\dot{\gamma}$ is made faster

28

HYDRODYNAMICS IN CONFINED GEOMETRIES

Let us start by discussing two works that study hydrodynamics in confined geometries. First in section 28.1 when particles are confined to the plane via a soft harmonic potential and then in section 28.2 when the confinement is strict (in the quasi 2D regime introduced in chapter 22).

28.1 FROM SOFT TO STRICT CONFINEMENT

In [126] we study the hydrodynamics-enhanced collective diffusion of a group of colloids under soft two dimensional confinement as the confinement becomes stiff. Colloids are embedded in an unbounded fluid but their movement in the Z direction is constrained via a harmonic potential (see Fig. 22.1) as we introduced in chapter 22. As the spring constant of the confining (parabolic) potential is decreased, in this work we study the crossover between purely 3D hydrodynamics and quasi2D both numerically and theoretically.

It is well-known that, in quasi2D, collective diffusion is enhanced due to the effective compressibility of the in-plane hydrodynamics. This is evidenced in Eq. (22.9) (and related discussion), where we saw how the divergence of the in-plane mobility can be interpreted as the colloids interacting via a Coulomb-like repulsive potential in an incompressible fluid. In [126] we find that the enhancement of the collective diffusion is quite robust and remains significant when moving from strict to soft confinement.

The enhancement, being collective in nature, decreases with the concentration of colloids. As an example, for a projected surface fraction, $\phi = \pi a^2 N / L^2$, of about 0.4, a rather soft confining

potential of width $\delta = 3a$ will induce more than a ten-fold increase in collective diffusion for a density perturbation of typical size given by a wavelength $ka \sim 6 \cdot 10^{-2}$ (corresponding to a wavelength around $100a$). For nanoparticles (of about 10 nm) these lengths are not small (order 10 microns). We can find several experimental set-ups where the confinement is even stiffer than our example, $\delta \sim a$ (and thus the increase in diffusion is more pronounced). For instance, in the presence of walls (note that the mobility will be regularized by the presence of a wall) depletion forces can constrain the out-of-plane diffusion of colloids near the wall to $\delta \sim a$. If the surface and the colloids are charged the confinement will be related to the Debye length (nanometers). On the other hand, the theory presented in [126] can be directly applicable to confinement by ultrasound [86]. The pressure waves generated by high frequency (MHz) ultrasounds (an effective quadratic potential) can confine micron-sized colloids to a width $\delta \sim a$. In Fig. 28.1 we show the collective diffusion coefficient at short times, given by [27],

$$D_c(k) = D_0 \frac{H(k)}{S(k)}, \quad (28.1)$$

for systems with several densities. In Eq. (28.1) $S(k)$ is the structure factor (which is 1 for ideal particles, i.e no structure). $H(k) := H_s + H_c(k)$ is the so-called hydrodynamic mobility function, which further divides into a self (H_s) and collective (H_c) contributions.

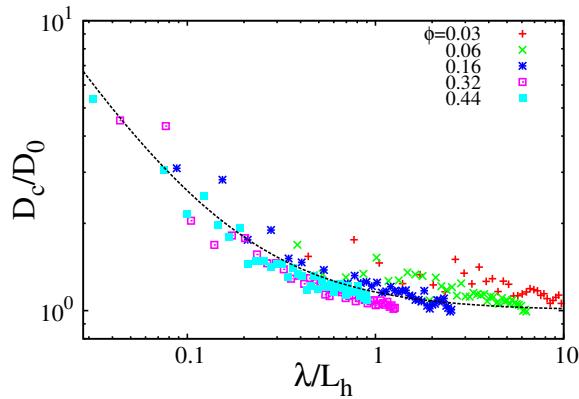


Figure 28.1: Collective diffusion, $D_c(k)$, of a group of ideal colloids under strict confinement ($\delta \rightarrow 0$) for several densities. Dashed line correspond to $D_c/D_0 = 1 + \frac{\lambda}{2\pi L_h}$. Here $L_h := \frac{2a}{3\phi}$ is the hydrodynamic length and $\lambda := \frac{2\pi}{k}$ is the wavelenght of a density perturbation. Note that for $\lambda \rightarrow \infty$, $D_c \rightarrow D_0$, corresponding to the single particle diffusion coefficient, i.e. in this limit there are no collective effects (in the case of ideal particles).

Under strict two-dimensional confinement (infinitely stiff trap) the collective colloidal diffusion is enhanced and diverges at zero wave number (like k^{-1}), due to the hydrodynamic propagation of the confining force across the layer. Physically this results in increasingly large density perturbations “repelling” with increasing strength.

At intermediate and short wavelengths, we study to what extent the hydrodynamic enhancement of diffusion is masked by the conservative forces between colloids, which make the structure factor different from 1.

Notably, at very large wavelengths, the collective diffusion becomes even faster than the solvent momentum diffusive transport and a transition from Stokesian dynamics to inertial dynamics takes place, which we study using the **ICM** described in chapter 20.5.

Figure 28.2 shows the gradual transition from quasi-2D (enhanced) collective diffusion and 3D diffusion, as the stiffness of the trap is decreased. The trap stiffness length is given by $\delta = \left(\frac{k_B T}{K_{\text{trap}}}\right)^{\frac{1}{2}}$, where $U_{\text{trap}} = \frac{1}{2} K_{\text{trap}} z^2$. As expected $H(k) \rightarrow 0$ as the diffusion becomes 3D. In other words $D_c(k) = D_0$ for ideal particles in 3D.

We observe that the analytic solution for the collective diffusion of colloids under a Gaussian trap of width δ still shows enhanced diffusion for large wavelengths $k\delta < 1$, and a gradual transition to normal diffusion for $k\delta > 1$ (see Fig. 28.2).

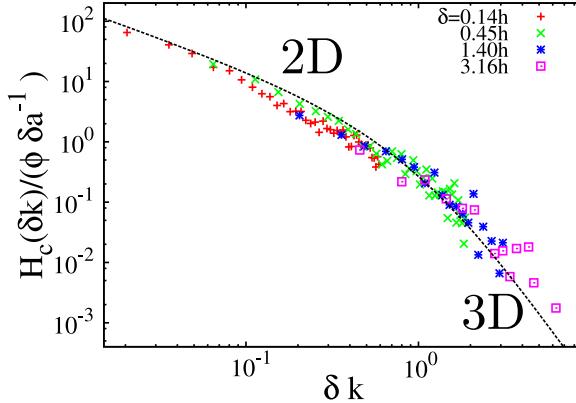


Figure 28.2: The enhancement of the collective diffusion (given by the hydrodynamic function $H_c(k)$ in Eq. 28.1) of a group of ideal colloids ($S(k) = 1$) as the confinement goes from non-existent ($\delta \rightarrow \infty$) to stiff ($\delta \rightarrow 0$) trap. Dashed lines correspond to Eq. (18) in [126], from where this figure has been taken.

28.2 LIMIT OF STRICT CONFINEMENT

In [71] we focus on the limit $\delta \rightarrow 0$ by taking the mathematical limit of the confining force becoming infinite. We already introduced this limit in chapter 22 by introducing a constraint into the mobility $M = M(z)$, leading to a thermal drift $\partial_z M \neq 0$. We saw how the confining force propagates to the fluid resulting in an effectively compressible in-plane mobility (which is the origin of the anomalous collective diffusion). In fact, as represented in Fig. 28.1 the collective diffusion coefficient diverges like the inverse of the wavenumber.

In [71] we extend the previously existing hydrodynamic theory to account for a species/color labeling of the particles (which is needed to model experiments based on fluorescent techniques). We then study the magnitude and dynamics of the density and color density fluctuations theoretically (using linearized fluctuating hydrodynamics) and simulations (with our novel BDHI quasi 2D GPU algorithm, see chapter 22). The action of the effective repul-

sion fundamentally changes the dynamics in quasi 2D as evidenced in figs. 28.3 and 28.4, where we place striped over-densities in the middle of the domain and study their time evolution.

We measure concentration as the number density defined as

$$c^{(1)}(\mathbf{r}) = \sum_i \delta(\mathbf{r} - \mathbf{q}_i) \quad (28.2)$$

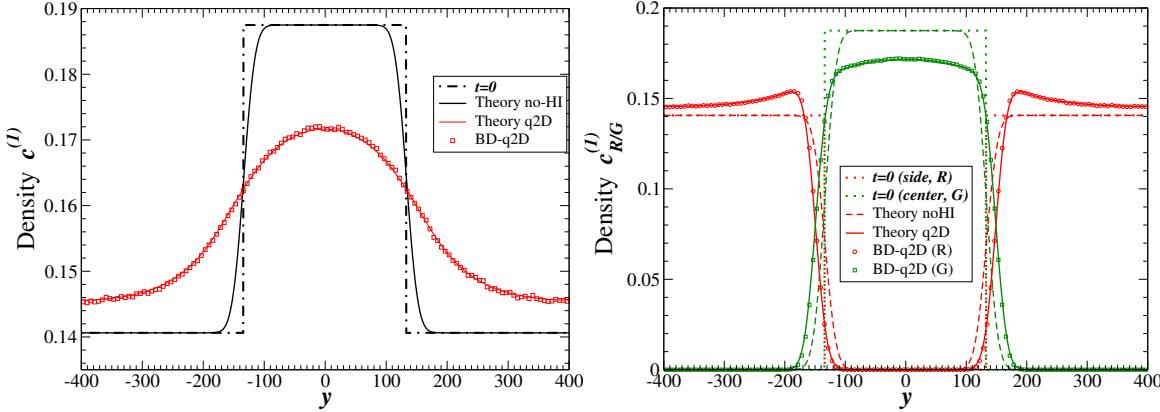


Figure 28.3: Time evolution of a stripe density perturbation initially localized in the middle third of the domain in quasi 2D. Shown here are the density profiles averaged both in ensemble and in the x direction. (Left) The total density, $c^{(1)}(y, t)$, at time $t = 0$ (dashed-solid black line) and at a later point in time (red squares), theory for the quasi 2D line (red) comes from our mean field (Eq. 30 in [71]). The solution to the diffusion equation without hydrodynamic interactions (at the same time) is shown as a solid black line. (Right) We tag particles starting in the middle stripe as green ($c_G^{(1)}$, green lines and squares) and the rest as red ($c_R^{(1)}$, red lines and circles) and plot their density profiles, $c_{R/G}^{(1)}$. Red and green symbols correspond to $c_R^{(1)}$ and $c_G^{(1)}$ at the same point in time as the left pane. Dotted lines correspond to $t = 0$. Our mean field theory is also shown here with solid lines. Dashed lines correspond to the diffusion equation without hydrodynamic interactions. All particles are passive tracers.

Fig. 28.3 shows the ensemble average of the time evolution of the density profiles, evidencing the effect of the enhanced collective diffusion (marked as q2D in the figure). On the other hand, we study non-equilibrium giant fluctuations by looking at the time

evolution of a single realization of the diffusing stripes. In Fig. 28.4 we look at the time evolution of an over-density, while 28.5 shows simulations with “color” density gradients. In the latter simulations the system has uniform total density across the whole domain, but we mark (color) particles starting in the middle third of the domain and study the evolution of their concentration.

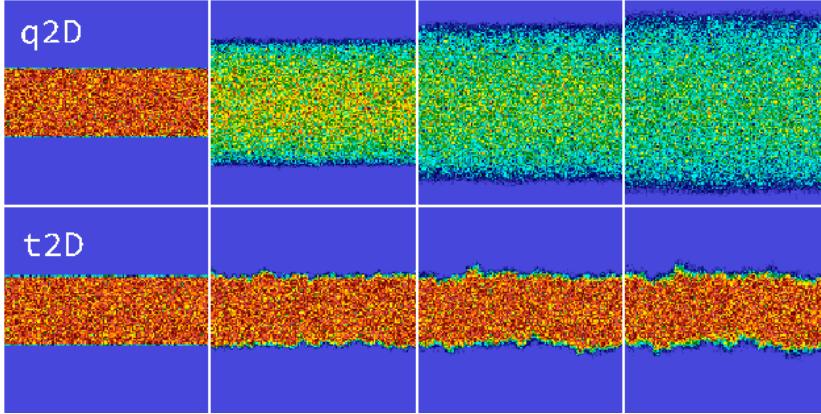


Figure 28.4: Time evolution (snapshots at times increasing from left to right) of a density perturbation initially localized in the middle third of the domain (blue represents zero concentration of particles). We show snapshots for quasi 2D (top row) and true 2D (bottom row) at the same diffusive times ($t^{t2D/q2D} = tD_0^{t2D/q2D}/a^2$). The images show the number density by counting the number of particles in each cell of a 128^2 grid; the color bar is fixed across all panes from 0(blue) to 0.4 (red). All particles are passive tracers.

Giant fluctuations (evident in the bottom row of Fig. 28.4 for true 2D) have disappeared in quasi 2D in the presence of a density gradient. However, in Fig. 28.4 giant fluctuations are being masked by the vast dispersion enhanced by the collective diffusion in the presence of density gradients.

The visual appearance of the true 2D panes is similar in figs. 28.4 and 28.5 as expected, given that in both cases particles are passive, non-interacting, tracers. Furthermore, we observe giant fluctuations in both density and color gradient experiments for true 2D. On the other hand, giant fluctuations can only be seen for quasi 2D in the color gradient case.

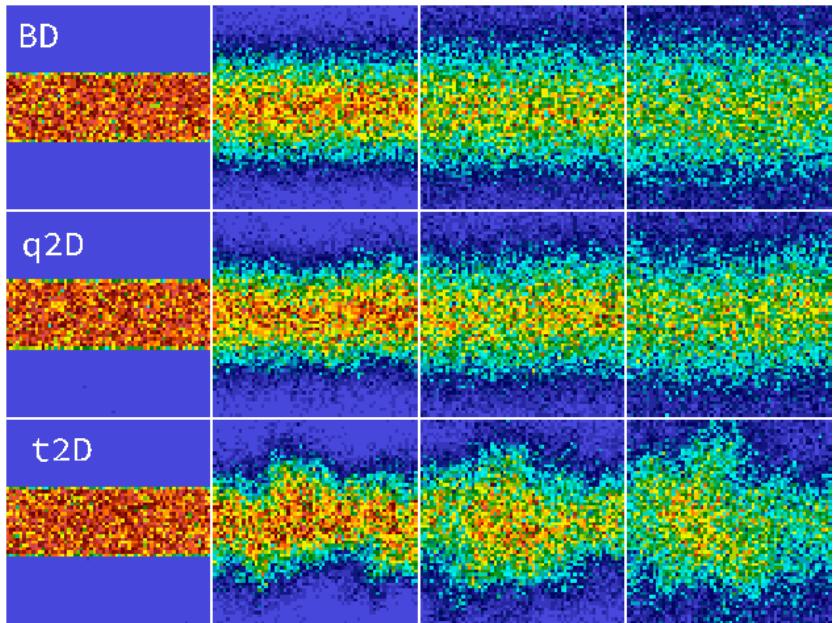


Figure 28.5: Time evolution (snapshots at times increasing from left to right) of a color (species) density perturbation initially localized in the middle third of the domain. All simulations have a uniform total density, with a packing fraction of $\phi = 1$. We show snapshots in the absence of hydrodynamics (BD, top row), quasi 2D (middle row) and true 2D (bottom row) at the same diffusive times ($t^{BD}/t^{2D/q2D} = tD_0^{BD/t^{2D/q2D}}/a^2$). The images show the number density by counting the number of particles in each cell of a 64^2 grid; the color bar is fixed across all panes from 0(blue) to 0.4 (red). All particles are passive tracers.

Another puzzling fact discovered in this work is the effect of q2D dynamics on the single particle diffusion (ideal tracers). In contrast to the diverging collective diffusion we find that the long time self diffusion coefficient is reduced (see Fig. 28.6). Although the effect is small (less than 15% reduction of the self diffusion coefficient, see Fig. 28.6) its physics are an example of coupling between density fluctuations and single particle diffusion, in systems where the divergence of mobility is not zero. The phenomena has also been observed in charged particles, where the electrostatic potential has the same role as the thermal drift $\partial_z M$ here [71].

The mathematical formalism for these type of phenomena is not trivial, and reference [71] provides a first sketch of it.

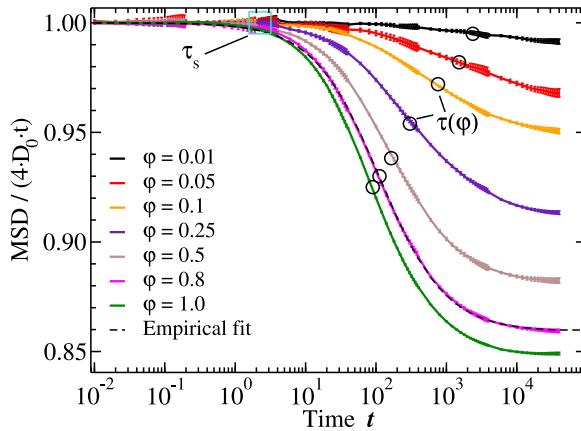


Figure 28.6: Self diffusion coefficient (measured via the mean square displacement (MSD)) of a single particle in quasi 2D for different packing densities, ϕ .

29

OPTOFLUIDIC CONTROL OF THE DISPERSION OF NANOSCALE DUMBBELLS

In [69] UAMMD is used in conjunction with an already established in-house solver for interaction between particles and optical fields to study the hydrodynamically-enhanced dispersion of a collection of nanoscale dumbbells in a vortex lattice formed by an incident laser [139]. In particular, the system is simulated with open boundaries via the Cholesky decomposition approach in chapter 19.1. In some simulations hydrodynamics are turned off to isolate their effect by using plain BD (introduced in chapter 17).

Previous research has shown that gold nanoparticles immersed in water in an optical vortex lattice formed by the perpendicular intersection of two standing light waves with a $\pi/2$ rad phase difference will experience enhanced dispersion that scales with the intensity of the incident laser. We show that flexible nanoscale dumbbells (created by attaching two such gold particles by means of a molecule chain such as a DNA or oligomer) in the same field display different types of motion depending on the chain length and field intensity. We have not disregarded the secondary optical forces due to light scattering. The dumbbells may disperse, rotate, or remain trapped. For some values of the parameters, the (enhanced)

30

dispersion possesses a displacement distribution with exponential tails and exhibiting time-dependent diffusion, including anomalous (though Brownian) diffusion.

The interested reader is referred to [69] for details. For the sake of this dissertation, the main outcome of this work (and research line) is to highlight the ability of UAMMD to include other type of interactions between fields and particles, which can be expressed in terms of Green functions. In particular, secondary optical forces (arising from light scattered by the nanoparticles) are evaluated from the optical Green function (see [69, 139]). The techniques developed here for the hydrodynamic propagators can also be used for the optical problem, and this subject is part of ongoing research, the resulting “opto-hydrodynamic” computational framework promises to be of great usefulness to analyze recent experiments with plasmonic nanoparticles [140].

ONGOING WORK, FUTURE DIRECTIONS AND CONCLUSIONS

We are living in the birth of the GPU era, in which new computing technologies evolve every day and the software development kits (SDK) that allow to use them evolve alongside them. Lagging behind both hardware and SDK are the applications that make use of them, such as UAMMD. Finally, one step behind all of them lie the novel algorithms, some of which have been developed during this thesis. Climbing up the proverbial ladder requires constant work and research into these new technologies, as if we were walking a mechanical escalator in reverse. In order to successfully survive the constant climb it is worth to, in my opinion, lose the fear of returning back to basics, to reinvent the wheel. We chose CUDA for UAMMD because it was the best choice at the moment, but we did so knowing that this might change over time. In UAMMD reinventing the wheel is easy by design. The base modules (*Interactor*, *Integrator*, *ParticleData*,...) are simple and short, allowing for extension or modification of even the most basic assumptions.

New programming interfaces

Recently SYCL, a cross-platform C++ programming model for heterogeneous computing, has been gaining a lot of traction. The SYCL standard was introduced by the Khronos group (the same group behind OpenCL and OpenGL) and several implementations already exist (like Intel's DPC++ and hipSYCL just to name a few). SYCL aims to provide a common specification allowing to write code that will run in any massively parallel accelerator, such as a GPU or CPU, by having backends to accomodate the different hardware vendors (CUDA for NVIDIA, ROCm for AMD, oneAPI for Intel...). In the future, it is within the realm of possibilities to port UAMMD to SYCL, which will enable it to run on any hardware and, moreover, would make it more future-proof.

multi-GPU

Currently UAMMD is a single GPU library. Once a GPU is selected at initialization every other GPU in the system is ignored. As a matter of fact, in the description of the different algorithms we have not acknowledged the existence of multi-GPU. While some algorithms, such as the neighbour lists or the immersed boundary method, present almost linear scalings when using several processors, others, mainly the ones involving the Fourier transform, require an all-to-all communication in a multi-GPU environment. Many novel algorithms in UAMMD are based on the **FFT** and, given that a single GPU can already fit quite a large system, the extra software complexity and maintenance burden associated with a multi-GPU implementation was deemed not worthy of the effort. Furthermore, our simulations are often governed by thermal fluctuations and require us to launch the same simulation many times to gather statistics. In this case the multi-GPU parallelization is trivial. Nonetheless, the codebase was designed with the possibility of going multi-GPU in mind. In the future it would be possible for a developer to incrementally make the different basic UAMMD modules multi-GPU-aware, but it is not something being considered at the moment.

Outreach

UAMMD offers a plethora of tools that researchers can leverage to accelerate their simulations, however, making use of them requires

certain knowledge and resources that might not be in the skillset of a typical research group, much less an experimental group. One of our current lines of work consists in bridging the gap between us, producers of scientific software, and their potential users, such as experimental research groups, in particular by offering easy-to-use graphical user interfaces (GUI) tailored for the (quantitative) reproduction of specific experiments or experimental techniques. At the moment, we are pursuing this idea by iterating, with the help of a research group at CSIC, towards a GUI for a “virtual QCM” which will aid in the characterization of samples for this sensor.

What others are doing with UAMMD

At the moment, extensions to UAMMD are being actively developed by several collaborators. I still retain my role as the lead developer and routinely push fixes, improvements and new modules to the main repository, but other researchers have been forking, extending and hacking UAMMD for some time now. It is worth mentioning here some of the ongoing projects that are being carried out by collaborators using UAMMD.

- Pablo Ibañez is studying mechanical properties of virus capsids under the stress of an atomic force microscope’s tip.
- Marc Meléndez and collaborators are studying the dynamical properties of a Quartz Crystal Microbalance (QCM) biosensor for sample characterization, which involves embedding an oscillating wall in a fluid with inertial effects. The subtle coupling between a sample (placed on top of the wall) and the oscillating motion of the wall is studied to provide insights about the sample.
- Pablo Palacios is studying the response of bioconjugated magnetic nanoparticles to an alternating magnetic field for biomarker detection. This endeavor requires the development of a new [FCM](#)-like algorithm for magnetism. Furthermore, the aforementioned nanoparticles have a direction, requiring to take torques and angular displacements into account when adding hydrodynamics.
- Collaborators at the New York University are using UAMMD’s hydrodynamic and electrostatics modules for several works,

such as the study of the rheological properties of a fiber suspension (actin networks), charge diffusion in ionic channels, and more.

- Salvatore Assenza is incorporating coarse-grained models for simulations of ADN strand suspensions.

Finally, several works exploring UAMMD’s facet as an external, GPU-enabled, accelerator library are ongoing, mostly making use of the hydrodynamic solvers, electrostatic solvers and neighbour list capabilities of UAMMD. In particular, I routinely provide external collaborators with python interfaces for the different UAMMD modules required by their research.

Novel algorithms

New algorithms have been developed that enable simulations of physical regimes that were previously unachievable. These algorithms are accompanied by highly performant GPU implementations in UAMMD, boosting their applicability. The bulk of this thesis contribution to this end deals with hydrodynamics and electrostatics in confined geometries. Furthermore, an efficient, GPU-enabled, algorithm for particle-grid communication has been developed and shown to outperform existing alternatives in the common use cases in complex fluid simulations.

New physics

Although this thesis is more centered around developing new techniques and software than finding and/or explaining new physical phenomena per se, several works presenting new physics have been shown. In particular, works studying diffusion in confined geometries, rheological properties of star polymer suspensions, intracellular viscosity and opto-hydrodynamics have been discussed. Some of these lines of work are still active and more publications are expected.

SPANISH CONCLUSIONS

Estamos viviendo los orígenes de la era de la GPU, en la que las nuevas tecnologías evolucionan cada día, así como las herramientas

de software que permiten hacer uso de ellas. Un paso por detrás se encuentran las aplicaciones que hacen uso de la GPU, tales como UAMMD. Finalmente, aún más atrás están los algoritmos novedosos, algunos de los cuales han sido desarrollados en esta tesis. Trepas por la escalera proverbial requiere trabajo e investigación constantes en estas nuevas tecnologías, como si quisieramos subir por una escalera mecánica en sentido contrario. Para sobrevivir con éxito esta constante subida es necesario, en mi opinión, perder el miedo a volver a explorar las asunciones más básicas y a reinventar la rueda. Elegimos CUDA para UAMMD porque era la opción más adecuada en el momento, pero lo hicimos sabiendo que esto podría cambiar en el futuro. En UAMMD reinventar la rueda es fácil por diseño. Los módulos base (*Interactor*, *Integrator*, *ParticleData*,...) son simples y cortos, permitiendo la extensión y/o modificación de incluso sus principios más fundamentales.

Nuevos algoritmos

Durante esta tesis se han desarrollado nuevos algoritmos que permiten la simulación de regímenes físicos que eran previamente inalcanzables. Estos algoritmos están acompañados de implementaciones de alto rendimiento en GPU dentro de UAMMD, impulsando su aplicabilidad. El grueso de los algoritmos desarrollados en esta tesis lidian con la hidrodinámica y electrostática en geometrías confinadas. Además, se ha desarrollado e implementado en UAMMD un algoritmo GPU para la comunicación entre partículas y una malla que muestra ser más performante que las alternativas existentes en los casos de uso habituales en las simulaciones de fluidos complejos.

Nueva Física

Aunque esta tesis no está centrada en encontrar/explicar nuevos fenómenos físicos per se, se han mostrado varios trabajos presentando nueva física. En particular se han mencionado trabajos que estudian difusión en geometrías confinadas, propiedades rehológicas de polímeros estrella en suspensión, viscosidad intra celular y opto-hidrodinámica. Algunas de estas líneas de trabajo están aún activas y se esperan más publicaciones.

Part VI
APPENDIX

A

BASIC NOTIONS OF CUDA / C++ PROGRAMMING

UAMMD uses the C++14 standard¹, which presents subtle but sometimes important differences with the previous iteration of the language (C++11). Note however, that this standard is wildly different to “classic” C++98, which could be considered another language altogether. Furthermore, **UAMMD** uses the CUDA extensions to the language, which introduce some new keywords and subtle rules.

In this chapter, we will give a few hints to ease the interpretation of the code examples throughout this manuscript for the uninitiated. Note that this is not intended to be a thorough description or tutorial of the language. For that the reader is redirected to the plethora of resources available online regarding CUDA and modern C++. This chapter summarizes most of the tools from the C++ language (the bare minimum) that a reader needs to be aware of in order to follow the logic in the examples scattered throughout this manuscript.

CUDA/C++ is a compiled language, meaning that before executing a source code, it must be transformed into an executable binary by an external program called the *compiler*. We will also refrain from discussing the compilation of CUDA codes².

Let’s start with some basic concepts in C++ and then go to the **GPU**. The entry point for execution in all C++ codes starts with a function called “main”. This function must be defined by the user with the name “main” and a returning type int (see example code 31).

¹ The only restriction in adopting more modern standards, like C++20, is the adoption rate of CUDA. Up to the date of this writing, the latest CUDA toolkit supports up to C++17.

² **UAMMD**’s online resources shed light on this process, which might change over time for reasons outside of the author’s control.

```
//This is a comment, which is ignored by the
→ compiler
//The main function must be present in all C++
→ programs
int main(){
    //main must return an integer, encoded as 0 if the
    // program terminated successfully, and 1
    → otherwise.
    return 0;
}
```

Source Code 31: A C++ program that does nothing.

The auto keyword

The keyword *auto* can be seen in almost every code example in this manuscript. In contrast with other loosely typed languages, C++ forces the developer to always state the actual type of a variable when defining it. Similarly, the return type of a function must be specified. In some situations, for instance when dealing with metaprogramming (which we will discuss shortly), knowing the name of the relevant type can become cumbersome and verbose. Luckily, the latest standards allow to replace the type name by the keyword *auto*. When the compiler sees a variable with the type *auto*, it will automatically deduce the correct type for it. Example code 32 introduces a couple of use cases for the *auto* keyword.

Functions

Main is an example of a function (albeit a special one), the syntax for defining custom functions is similar, see code 32. The special return type **void** can be used to signify that a function returns nothing.

```
//We can define a function with the syntax below
//The returning type, in this case int, can be
→ replaced
// by the keyword auto.
int foo(int a){
    //This function takes an integer, referred to as
    → "a"
    // and returns its value multiplied by 2
```

```

    return a*2;
}
//The main function will simply call the foo function
→ with an
// arbitrary number and store the result in a
→ variable
int main(){
    int value = foo(12);
    //value holds the integer 24
    //The type of the variable, int, can be replaced by
    → auto.
    //For instance,
    auto value2 = foo(12);
    //The type of value2 is automatically deduced to be
    → "int"
    return 0;
}

```

Source Code 32: Defining a function in C++.

Including other codes

By default, a C++ source code will have access only to the basic rules of the language. We can add libraries via the *include* directive, which will make available to us every function, class and variable defined in another file. For instance, in example code 33 we can see how to include the standard library header file that provides printing functionality.

```

#include<iostream> //Includes std::cout and
→ std::endl
//Uncommenting the line below permits to omit writing
→ std::
//using namespace std;
int main(){
    std::cout<<"Hello world"<<std::endl;
    return 0;
}

```

Source Code 33: The classic Hello World program in C++. Once executed, it will print “Hello world” to the terminal.

Namespaces and the using keyword

In order to avoid naming collisions (two libraries or files defining entities with the same name) C++ provides the concept of *namespaces*. We have already seen the standard library namespace, `std`, in action in example 33. In order to reference an entity inside a given namespace, the “`::`” operator is used. In the aforementioned example, we access the object `cout` of the `std` namespace with `std::cout`.

This can become a nuisance when a given namespace is constantly used. C++ provides the `using namespace` construct to mitigate this. For instance, throughout this manuscript, the line `using namespace uammd;` is used to omit writing `uammd::` every time an `UAMMD` entity is referenced.

The `using` keyword has a second usage, it allows to “rename” a type. In C++ type names can become quite convoluted and long. The `using` keyword can be used in these cases. For instance, in example code 8 we write the line `using PF = PairForces<Potential, NeighbourList>;`, allowing to use simply “`PF`” when we want to refer to the longer type name.

Objects

Although C++ can work as a functional language, it is heavily object oriented. The `struct` and `class` keywords are used to define objects in C++.

An object can hold both variables and functions, which are referred to as *members*. A member function in a class has access to all other members of the object. In general, object members (functions or variables) can have either private or public visibility. From the outside, only public members can be accessed, whereas private members can only be accessed by other members of the same class.

In the author’s personal programming style, structs are usually employed to serve as an aggregate of variables, whereas classes are used as a more complex object (with member functions, etc). Note however, that the `struct` and `class` keywords are almost interchangeable³.

³ The only difference between a struct and a class is the default visibility of its members. By default, members in a struct have public visibility, whereas a class defaults to private visibility.

```

#include <iostream>
//The syntax for creating a struct with type name
→ Parameters, holds two double variables.
struct Parameters{
    double var1;
    double var2;
};
//A class with public visibility is equivalent to a
→ struct.
class MyClass{
    //The public keyword marks every defined member
    → below as accessible from outside
    public:
        void print(){
            std::cout<<"Hello"<<std::endl;
        }
};
int main(){
    //Create an instance of the struct Parameters
    Parameters par;
    //Set the value of one of its variables
    par.var1 = 1.0;
    //Create an instance of the object MyClass
    MyClass c;
    //This "c" is created here as an instance of
    → MyClass
    //Call its member "print".
    c.print();
    return 0;
}

```

Source Code 34: Examples of object creation.

Given that objects can be created and passed around as regular variables, we can use them to encapsulate and transport logic between different parts of the code.

In **UAMMD**, every *Interactor* and *Integrator* is provided as a C++ **class**.

Scope

A name (be it of a function, class, variable, etc...) is only visible inside a certain portion (or portions) of the source code. This portion is called the *scope* of the name. There are several scopes

in a C++ source code, however we will limit our introduction to discuss the lifespan of the variables we create.

In particular, when a new instance of an object (or in general any variable) is created, as we saw in example 34, it will live until it goes out of *scope*. At that point, the object/variable will be destroyed. Destruction involves the invalidation of the variable and the release of the memory allocated for it. Additionally, in the case of objects the destructor function, if defined for that object, will be called. When a name goes out of scope it is made available for use again (i.e. we cannot have two variables with the same name, unless is out of scope).

The portion of code between two curly brackets is called a *block*. As a rule of thumb, a variable will live until the code block in which it has been defined is closed. In UAMMD this is used, for instance, to release the handles to the particle properties provided by *ParticleData*. When the destructor of one of these handles is called, it signals *ParticleData* for it to take it into account (for instance, by synchronizing the CPU and GPU versions of the data).

Let us showcase the lifespan of a variable by creating some instances inside different types of code blocks.

```
//Let us define a class and give it a destructor
class MyClass{

public:
//The destructor is a special member function that
//has ~[class name] as signature.
~MyClass(){
    std::cerr<<"This is printed when the instance
    goes out of scope"<<std::endl;
}
//Additionally, a constructor can be defined.
//The constructor can even have input arguments.
MyClass(){
    std::cerr<<"This is printed when an instance is
    created"<<std::endl;
}
//If either the constructor or destructor are not
//present, they are automatically defined by the
//compiler as just empty functions
};
```

```

//A function that just creates an instance of
→ MyClass
void foo(){
    MyClass c; // "c" is created and its constructor is
    → called
}//"c" goes out of scope and its destructor is
→ called

int main(){
    { //This is a code block
        MyClass c; // "c" is created and its constructor
        → is called
        //Note that even though we used the name "c" in
        → the function foo, we can use it again here,
        → since the variable "c" inside foo went out of
        → scope
    }// "c" goes out of scope here and its destructor
    → is called
    float someVariable; //Let's create some variable
    → here
    //A loop also works as a code block
    for(int i=0; i<10; i++){
        MyClass c; // "c" is created
        //Again, the name "c" is available because it was
        → not currently in scope
    }// "c" goes out of scope

    //MyClass someVariable; //This is an illegal name,
    → because there is already a variable in scope
    → with this name
    MyClass someVariable2;
    return 0;
}//someVariable2 is released here and its destructor
→ is called

```

Source Code 35: Examples of a variable going out of scope

Object inheritance

In C++, objects can inherit the functionality of other classes, and even override some of it. For instance, every polygon has an area, we can create a class called *Polygon* that provides a member function called *area* (returning the area). However, a polygon is just a concept that has no area per se. We can then create another

class called *Square* that inherits from *Polygon* and overrides the *area* function, providing its squared side.

Given that the area of a *Polygon* object is meaningless, we can mark it's *area* function as **virtual** and add the suffix = 0; to its signature. This transforms the *Polygon* class into a *virtual base class*, which means that it cannot be instanced directly, rather it must be inherited and the virtual members must be overriden. See example code 36.

```
//This is a pure virtual class. A "Polygon" cannot be
→ instantiated, only classes that inherit from it
→ are instantiable.
class Polygon{
public:
    virtual double area() = 0;
};

//This class is a kind of Polygon (inherits from it)
class Square: public Polygon{
    double side = 2.0; //An arbitrary value for the
→ side
public:
    //We can override the area member of Polygon
    virtual double area() override{
        return side*side;
    }
};

//This class is a kind of Polygon (inherits from it)
class Circle: public Polygon{
    double radius = 2.0; //An arbitrary value for the
→ radius
public:
    //We can override the area member of Polygon
    virtual double area() override{
        return 3.1415*radius*radius;
    }
};

int main(){
    Square s;
    auto a = s.area();
    Circle c;
    auto a2 = c.area();
    //Polygon pol; //Illegal, a pure virtual class
    → cannot be instanciated.
}
```

```

    return 0;
}

```

Source Code 36: Inheriting a virtual class.

In [UAMMD](#), *Interactor* and *Integrator* are virtual classes such as *Polygon*. One of the benefits of using inheritance is that it allows to provide a general logical interface. In general C++ an inherited class can masquerade as an instance of the parent class (but not the other way around). So a *Square* can be used as a *Polygon* but not viceversa. This only happens, however, for pointers (since the virtual base class cannot be instanced directly).

Let us discuss now the type of pointer used throughout this manuscript, the `std::shared_ptr`.

Shared pointers

Without going into details (the reader can learn more online), a C++ shared pointer can hold ownership of an object through a pointer. In the [UAMMD](#) examples in this manuscript we use shared pointers mainly to pass around *Interactors* and *Integrators*. For instance, in example 8 we create and return an instance of *PairForces*, a class inherited from *Interactor*. On the other hand, the *Integrator* member function `addInteractor()` will take in its argument a pointer to an *Interactor*, meaning that the *PairForces* instance created in example 8 can be passed to an *Integrator* (such as the one created in example 15) directly.

```

#include <memory> //std::shared_ptr
#include <vector> //std::vector
//Using the Polygon declarations of the previous
//example
//This function takes a pointer to a polygon and
//returns its area
auto getAreaOf(std::shared_ptr<Polygon> pol){
    return pol->area();
}
int main(){
    //Create pointers to a square and a circle
    auto s = std::make_shared<Square>();
    auto c = std::make_shared<Circle>();
    //The getAreaOf function works for pointers of any
    //class that inherits from Polygon
}

```

```

double as = getAreaOf(s);
double ac = getAreaOf(c);
//We can also aggregate pointers to Polygon
→ children in a vector
std::vector<std::shared_ptr<Polygon> > vec({s, c});
return 0;
}

```

Source Code 37: Using shared pointer to hold many types of polygons.

Metaprogramming

In addition to inheritance, C++ offers another way of writing generic code called templates. A template is a function or object that is generic for any type (sometimes the type is restricted under some assumptions). For instance, let us rewrite the function `getAreaOf` in example 38 using metaprogramming instead of inheritance.

```

//This function takes any type and returns its area.
→ Of course, it will not compile if the provided
→ type has no member function called area.
//When we call this function, T will be replaced by
→ the type of the provided argument
template<class T>
auto getAreaOf(T pol){
    return pol.area();
}
int main(){
    Square s;
    Circle c;
    //The getAreaOf function now works any type that
    → provides a function called area
    double as = getAreaOf(s);
    double ac = getAreaOf(c);
    double var = 1.0;
    //The code would be invalid if a double is passed,
    → since the function expects the member area to
    → exists.
    //double avar = getAreaOf(var);
    return 0;
}

```

Source Code 38: Using shared pointer to hold many types of polygons.

In [UAMMD](#), templates are used extensively to generalize modules. For instance, the *PairForces* module (see example code [8](#)) is templated for both the potential and the neighbour list. In another example, the [IBM](#) module is templated for several aspects of the algorithm, like the spreading kernel, the particle and grid quantities or the quadrature weights.

Iterators

In C++, *Iterators* are a type of object that *points* to an element in a range of elements (such as a container or pseudo-container) and has the ability to *iterate* through the elements of that range. The canonical example of an iterator is a pointer to the first element of an array. We can use a pointer to iterate through the contents of the array. However, like many things in programming, the internal workings of a certain iterator are irrelevant as long as the iterator works “as is”. For instance, a *constant* input iterator works by always returning the same value, regardless of what “element” in the “container” it is pointing to, as if it was iterating over an infinite sized array filled with the same number. In practice, a constant input iterator would be implemented as simply providing the same number all the time, without requiring any underlying memory to be allocated or present.

Several categories of iterators exists depending on the functionality they provide, figure [A.1](#) provides an overview. Iterators play a central role in connecting algorithms with containers in C++ and allow for deep customization. For instance, say that a given algorithm requires as input an ordered range of numbers and, after some operations, fills with some results the contents of two output arrays. Furthermore, say we are only interested in one of the two output arrays. Instead of allocating three vectors and filling the first with the range of numbers we could use some fancy iterators to save a lot of work; For the input range of numbers, we could pass an input *counting iterator* (see example code [39](#)). For the non-required output, we could provide the algorithm with an output *discard iterator*⁴ that simply ignores any value that is assigned to any of its elements (thus preventing unnecessary memory copies).

⁴ The thrust library provides a series of fancy iterators like the ones mentioned in this paragraph (counting, constant, discard...).

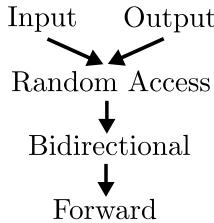


Figure A.1: The different kinds of C++ iterators. Iterators can be either for input (reading), output (writing) or both (read/write). The most generic type of iterator is called *Random Access* and allows for arbitrary access to its elements. Below, a *Bidirectional* constrains the access to be always next to the previously accessed element. In other words, a *Bidirectional* iterator can only be issued to advance or regress one element at a time. Finally, the most restricted iterator kind, the *Forward* iterator, can only be accessed element by element in order.

In another instance an algorithm may require to iterate through the elements of a container in reverse (starting at the last element and iterating up to the first), a *reverse* iterator could be used instead of reordering the container prior to the execution of the algorithm.

Iterators appear many times in [UAMMD](#). For instance, any *random access* iterator can be provided for the input per-particle and per-grid quantities in the [IBM](#) module (see chapter 21). The only restriction here would be that depending on whether we are spreading or interpolating, the iterators will be required to be either input or output ones. A simple example case of this would be, for instance, spreading the charges of a collection of equally charged particles. A *constant iterator* could be provided in this case.

```

#include <iostream>
#include <vector>
#include <thrust/iterator/counting_iterator.h>
//Three ways of making a counting iterator
int main(){
    std::vector<int> v{0,1,2,3,4,5}; //A vector
    ↪ containing the integers from 0 to 5
    auto vit = v.begin(); //Vector iterator
    int* pit = &v[0];      //Pointer to the first
    ↪ element of the vector
    thrust::counting_iterator<int> cit(0); //A counting
    ↪ iterator that starts at 0
    //All three iterators return the same numbers in
    ↪ this range
    //Note that only "cit" is accessible beyond i=5,
    ↪ since the vector only has 6 elements
    for(int i = 0; i<v.size(); i++){
        std::cout<<vit[i]<< " "<<pit[i]<<""
        ↪ "<<cit[i]<<std::endl;
    }
    return 0;
}

```

Source Code 39: Using iterators in C++. In this example a counting iterator for the range 0-5 is constructed from three different sources. For the first two cases a container (vector) is created and filled with the actual integers from 0 to 5 (requiring memory allocation). We then get an iterator to the range of the container via the method *begin* of the vector (which returns an iterator pointing to the first element, see the variable “*vit*”). An equivalent way in this case is to simply get a pointer to the first element of the container (variable “*pit*”). Finally, a counting iterator is constructed using the one available in the thrust library. This special iterator is not associated to an underlying container, instead it is an object that provides an overloaded bracket operator that when given an integer, “*i*”, simply returns that same integer, “*i*”. Note that in this example, “*vit*” and “*pit*” are input/output iterators, while “*cit*” is just an input iterator. The method “*cbegin*” in vector can be used to make “*vit*” an input iterator instead. Additionally, making the type of “*pit*” be “*const int**” will also turn it into an input iterator. On the other hand, “*cit*” cannot be an output iterator, since there is nothing to write to behind it.

Other keywords

The **const** keyword simply informs the reader (and compiler) that the value of the accompanying variable cannot be modified after its declaration. When appearing in the signature of an object's member function, it conveys that the function will not modify the state of the object in any way.

A.1 BASICS RULES OF CUDA PROGRAMMING

CUDA is an extension of C++. All the language rules we have seen thus far also apply to CUDA. CUDA introduces some new keywords and an [API](#).

We use the `cudaMalloc*` family of functions to request global memory, although in practice we have higher-level containers available, such as `thrust::device_vector<T>`. Nowadays, it is not considered good practice to call `cudaMalloc` directly, similarly to using `malloc`/`new` vs. `std::vector<T>` in the CPU. We can populate the allocated global memory copying data from the CPU via the `cudaMemcpy*` family of functions. As with allocation, a series of higher-level functions are available as alternatives. In particular we have `thrust::copy`, which handles CPU-GPU copies. Additionally, kernels can write to or read freely from global memory arrays, with the atomic considerations proper to a parallel environment. We use the keyword `__global__` to mark a C++ function as a CUDA kernel, which must return a `void` (i.e not return a value), a kernel must be called with a special syntax and specifying the thread grid geometry (see example code 40). Additionally, kernels might call other functions, which must be adorned with the `__device__` attribute⁵.

```
#include <thrust/device_vector.h>
#include <iostream>

//Example of a device function, returns a*x+y
__device__ float saxpy_element(float a, float x,
→ float y){
    returns a*x + y;
}
```

⁵ The `__host__` attribute also exists. A function adorned with both device and host attributes can be called from both the device and the host.

```
//Performs the operation  $y = a*x + y$ , where  $y$  and  $x$ 
//are vectors of size  $n$  and  $a$  is a number
//Each thread will handle one element
//This kernel must be launched with at least  $n$ 
//threads
__global__ void saxpy(float a, const float *x, float
→ *y, int n){
    //Built-in variables are available in kernels to
    → compute things like a unique id for each thread
    → in the grid
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    //The variable "id" lives in the thread-local, aka
    → register, memory space.
    //In case the kernel is launched with more threads
    → than elements.
    if(id>=n) return;
    //The contents of the input arrays,  $x$  and  $y$ , live
    → in the global memory space
    y[id] = saxpy_element(a, x[id], y[id]);
}

int main(){
    const int n = 1<<20;
    //Allocate two vectors in GPU global memory
    thrust::device_vector<float> y(n), x(n);
    //Fill the vectors with some arbitrary number
    //These operations will be performed in the GPU
    thrust::fill(y.begin(), y.end(), 1.123f);
    thrust::fill(x.begin(), x.end(), 2.123f);
    float a = 1.0f;
    //Configure and launch the kernel
    int blockSize = 128; //Number of threads per
    → block
    int nBlocks = n/blockSize + 1; //Number of blocks
    //Get pointers to the global memory
    float* y_ptr = thrust::raw_pointer_cast(y.data());
    float* x_ptr = thrust::raw_pointer_cast(x.data());
    //Writing or reading from these pointers from the
    → CPU will result in a crash, since the CPU
    → cannot access the device memory (and
    → viceversa).
    //y_ptr[0] = 1.0; //← Not allowed
    //Launch the kernel with  $nBlocks$  blocks of
    → blockSize threads each
    saxpy<<<nBlocks, blockSize>>>(a, x_ptr, y_ptr, n);
```

```

//Let us download the results to the CPU
std::vector<float> y_host(n);
//The thrust library knows this is a device to host
→ copy
thrust::copy(y.begin(), y.end(), y_host.begin());
//Now we can print the contents of the CPU copy
std::cout<<y_host[0]<<<std::endl;
return 0;
}

```

Source Code 40: Creating vectors allocated in the GPU, populating them and performing the saxpy operation using a kernel.

In a CUDA code it is important to always know the provenance of a pointer in order to know where to access it from⁶. This is the original motivation for the *ParticleData* **UAMMD** object.

B

DEALING WITH THE FFT IN THE GPU

Many of the numerical techniques in this manuscript make use of the **FFT**, mostly in order to easily compute convolutions. The GPU is a very powerful hardware for **FFT**, but using the available APIs can become a real nuisance. In this chapter we will see how **UAMMD** makes use of the CUDA library *cuFFT* [141] and how to work with data in Fourier space. While the information hereafter focuses on the *cuFFT* library, most lessons are applicable to other popular **FFT** libraries (such as *FFTW*). The documentation of the *cuFFT* library is sometimes obscure (and even lacking). There are some particular details that have made me waste countless hours. Hopefully with the help of this chapter you can save some.

Let's assume we want to work on some three-dimensional¹ data defined on the nodes (cells) of a mesh (grid) with size $\mathbf{n} = (n_x, n_y, n_z)$. For instance, the forces spread to the grid in **FCM**. A one dimensional transform has size $\mathbf{n} := \mathbf{n} = (n_x, 1, 1)$.

⁶ There is a special kind of memory, called *managed memory*, that can be accessed from both devices freely. However, it comes with its own downsides and is beyond the scope of this introduction. **UAMMD** sometimes defaults to managed memory when compiled in debug mode.

¹ The dimensionality of the transformation is tipically referred to as *rank*, so a 3D transform has rank 3.

We will store the data in a linear array of size $n_x n_y n_z$, and access the value of cell $\mathbf{c} = (c_x, c_y, c_z)$ at the element $i = c_x + (c_y + c_z n_y) n_x$. Where $c_{x/y/z} \in [0, n_{x/y/z} - 1]$.

CHOOSING AN EFFICIENT GRID SIZE

FFT libraries will typically process transformations of any sizes, however, many algorithms are dramatically more efficient when the size meets some conditions. Often the most efficient transform sizes are the powers of two ($n = 2^i$), but in general a multiple of the first prime numbers will work as well

$$n = 2^i 3^j 5^k 7^l 11^m \quad (\text{B.1})$$

Where i, j, k, l, m are positive integers. A number that can be expressed like this is referred to as an **FFT**-friendly number. Typically we will look for the next friendly number given a certain target size. One way to find this number is to simply generate all numbers, limiting the exponents to some low arbitrary numbers, that meet Eq. (B.1), sort them and find the nearest one to a target².

Although this is a good rule of thumb, we can expect the performance to vary between implementations, hardware and version. The best strategy is to test on a case by case basis. This effort may be worth it for long simulations, since this means that a bigger transformation can sometimes be faster. Given that in many spectral algorithms a bigger grid size means more accuracy, we could get a more accurate run with a shorter runtime. As a side note, in my personal experience, virtually every **FFT** implementation will perform much better when $i \leq 1$ in Eq. (B.1).

COMPLEX TO REAL (C2R) AND REAL TO COMPLEX (R2C) TRANSFORMS

One particular optimization commonly employed is to take advantage of the fact that, in a C2R or R2C transform, half the wave numbers are redundant. In these cases the signal in Fourier space, $\hat{s}(\mathbf{k})$, must meet that

$$\hat{s}(\mathbf{k}) = \hat{s}(\mathbf{n} - \mathbf{k})^* \quad (\text{B.2})$$

² The code in **UAMMD** that performs this operation is at *utils/Grid.cuh*.

In particular *cuFFT* will only store the first floor($n/2$) + 1 wavenumbers in the x direction. By default, an R2C transform will take an input of size \mathbf{n} real numbers and return a complex array of size $\hat{\mathbf{n}} := (\text{floor}(n_x/2) + 1, n_y, n_z)$ complex numbers³. Similarly, a C2R transform will take a $\hat{\mathbf{n}}$ sized complex array and return a size \mathbf{n} array with real numbers.

In principle, *cuFFT* allows to store the input and output of a C2R or R2C transform in the same array in what it's called an *in-place* transform. However, in my experience this results in unexpected behavior, so I advise against it⁴.

DATA LAYOUT

Since most spectral algorithms in [UAMMD](#) require C2R and R2C transforms it is worth giving here some more details about the correspondence of wave numbers and elements in the data array.

Here is a convenience function that returns the wave number in each direction that corresponds to a certain index in the data array.

```
//Returns the wave number in each direction given a
→ linear index, i, and a grid size, nk.
int3 indexToWaveNumber(int i, int3 nk){
    int ikx = i%(nk.x/2+1);
    int iky = (i/(nk.x/2+1))%nk.y;
    int ikz = i/((nk.x/2+1)*nk.y);
    ikx -= nk.x*(ikx >= (nk.x/2+1));
    iky -= nk.y*(iky >= (nk.y/2+1));
    ikz -= nk.z*(ikz >= (nk.z/2+1));
    return make_int3(ikx, iky, ikz);
}
```

Source Code 41: Getting the wave number corresponding to a given linearized index.

³ Note that since a complex number is stored as of two real numbers, the storage of both arrays is similar

⁴ The documentation hints that in these cases the input size in the x direction for an in-place C2R transform should be $2(\text{floor}(n_x/2) + 1)$ to accomodate for all necessary complex values in the output. However, this does not seem to be the same rule for a C2R transform and has caused some problems to me in the past

Most **FFT** libraries provide some kind of advanced interface allowing to customize the data layout to some extent. In *cuFFT* this is called the *Advanced Data Layout*. In **UAMMD** this is used extensively to compute three or four transformations in a single batch using interleaved data. For example, we can transform in a single call the three directions of the forces spread to the grid in **FCM** (see sec. 20.1), which are stored in a vector with *real3*⁵ type elements. For example, if we want to transform the three coordinates for the forces in an interleaved array, we can instruct the library to interpret this data as three signals, each starting after the other and strided by three elements.

NYQUIST POINTS

The complex data resulting of a R2C transform (or the input data of a C2R transform) meet the condition $\text{data}(\mathbf{i}) = \text{data}(\mathbf{i} - \mathbf{n})^*$. A consequence of this is that, in a transformation with an even number of points in some direction, there are some points that are the conjugates of themselves. We call these uncoupled modes Nyquist points.

In **UAMMD**, we take this into account mainly when including random fluctuations in Fourier space (see for instance the **FCM** in chapter 20.1).

There are 8 nyquist points at most (including the $\mathbf{k} = (0, 0, 0)$ mode), corresponding to the vertices of the inferior left quadrant of the grid. Here is a convenience function that determines if a certain wave number is a Nyquist point:

```
bool isNyquistWaveNumber(int3 ik, int3 n){
    //Is the current wave number a nyquist point?
    const bool isXnyquist = (ik.x == n.x - ik.x);
    const bool isYnyquist = (ik.y == n.y - ik.y);
    const bool isZnyquist = (ik.z == n.z - ik.z);
    const bool nyquist =
        (isXnyquist and ik.y==0 and ik.z==0) or //1
        (isXnyquist and isYnyquist and ik.z==0) or //2
        (ik.x==0 and isYnyquist and ik.z==0) or //3
        (isXnyquist and ik.y==0 and isZnyquist) or //4
        (ik.x==0 and ik.y==0 and isZnyquist) or //5
        (ik.x==0 and isYnyquist and isZnyquist) or //6
```

⁵ The *real3* type stores three scalars contiguously.

```

        (isXnyquist and isYnyquist and isZnyquist);    //7
        return nyquist;
    }

```

Source Code 42: Finding out if a wave number corresponds to a Nyquist point given the wave number and the grid size.

C

BOUNDR Y VALUE PROBLEM (BVP) SOLVER

In sections 23 and 25 we laid out the basic PDEs as BVPs with the following generic form

$$\begin{aligned} (\partial_z^2 - k^2)y(z) &= f(z) \\ (\partial_z \pm k)y(\pm 1) &= \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \end{aligned} \quad (\text{C.1})$$

Where $\partial_z y := \frac{\partial y}{\partial z} := y'(z)$ and $\partial_z^2 y := \frac{\partial^2 y}{\partial z^2} := y''(z)$

The domain is here defined in the $[-1, 1]$ range for simplicity. We can later generalize to any size via a simple change in units. For instance by defining $z = z'/H$ (with $z' \in [-H, H]$) and $k = k'H$.

In this section we describe a generic, GPU-friendly, spectral solver for the set of equations in (C.1) based on the spectral integration method described in [142].

We will work with the discrete Chebyshev transform of Eq. (C.1). In general, we can expand a given function, $g(z)$ with $z \in \mathcal{R}$ into the first N terms of its Chebyshev series so that

$$g(z) = \sum_{n=0}^{N-1} \hat{g}_n T_n(z) \quad (\text{C.2})$$

Where we use the subscript n to distinguish between the Chebyshev and Fourier coefficients (since throughout this manuscript we have used the subscript k to denote Fourier coefficients).

In general, the Chebyshev polynomials of the first kind, T_n , are defined via a recurrent relation with

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \end{aligned} \quad (\text{C.3})$$

On the other hand, the Chebyshev polynomials can be interpreted as a cosine series under a change of variables. Given that by definition $T_n(\cos(\theta)) = \cos(n\theta)$, evaluating the function at the extrema points $x_n = \cos(n\pi/(N-1))$ transforms the expansion in Eq. (C.2) into a cosine series. We can then leverage the **FFT** to perform a discrete cosine transform by transforming the periodic extension of a signal. This allows to obtain the Chebyshev coefficients of a signal in $O(N \log(N))$ operations. We refer to this technique as the **FCT** (see chapter 8 of [143]).

We can then leverage several interesting properties of the Chebyshev series to easily (and exactly) derivate and integrate the different quantities.

In particular, we can use the well known indefinite integrals of the Chebyshev polynomials to integrate \hat{y}_n'' twice and get \hat{y}_n . First by integrating \hat{y}_n'' :

$$\begin{aligned}\hat{y}'_1 &= \frac{1}{2} (2\hat{y}_0'' - \hat{y}_2'') \\ \hat{y}'_n &= \frac{1}{2n} (\hat{y}_{n-1}'' - \hat{y}_{n+1}''), \quad n \geq 1.\end{aligned}\tag{C.4}$$

And then integrating again

$$\begin{aligned}\hat{y}_1 &= \frac{1}{2} (2\hat{y}_0' - \hat{y}_2') = \hat{y}_0' - \frac{1}{8} (\hat{y}_1'' - \hat{y}_3'') \\ \hat{y}_2 &= \frac{1}{4} (\hat{y}_1' - \hat{y}_3') = \frac{1}{4} \left[\frac{1}{2} (2\hat{y}_0'' - \hat{y}_2'') - \frac{1}{6} (\hat{y}_2'' - \hat{y}_4'') \right] \\ \hat{y}_n &= \frac{1}{2n} (\hat{y}_{n-1}' - \hat{y}_{n+1}') = \\ &\quad \frac{1}{2n} \left[\frac{1}{2n-2} (\hat{y}_{n-2}'' - \hat{y}_n'') - \frac{1}{2n+2} (\hat{y}_n'' - \hat{y}_{n+2}'') \right], \quad n \geq 3.\end{aligned}\tag{C.5}$$

This formulation gives two free parameters \hat{y}_0 and \hat{y}_0' , which are obtained using the **BCs**. Additionally, we consider $\hat{y}_n'' = \hat{y}_n' = 0$ for $n > N-1$ when calculating \hat{y}_n using (C.5).

Now, we can reformulate the boundary value problem using the Chebyshev series representations in (C.2) as

$$\sum_{n=0}^{N-1} (\hat{y}_n'' - k^2 \hat{y}_n) T_n(z) = \sum_{n=0}^{N-1} \hat{f}_n T_n(z).\tag{C.6}$$

Matching modes gives a system of $N + 2$ equations for the Chebyshev coefficients

$$\begin{aligned}\hat{y}_n'' - k^2 \hat{y}_n &= \hat{f}_n \quad n = 0, \dots, N-1 \\ \sum_{n=0}^{N-1} (\hat{y}'_n + k\hat{y}_n) &= \alpha \\ \sum_{n=0}^{N-1} (\hat{y}'_n - k\hat{y}_n)(-1)^n &= \beta\end{aligned}\tag{C.7}$$

Where we have $N + 2$ unknowns $\hat{y}_0, \hat{y}'_0, \hat{y}''_0, \dots, \hat{y}''_{N-1}$ and equations. We solve the algebraic system of equations (C.7) for the second derivative coefficients $\hat{y}''_0, \dots, \hat{y}''_{N-1}$ and integration constants \hat{y}_0 and \hat{y}'_0 , then determine \hat{y}_n by integrating twice using (C.5).

For the $k = 0$ mode, the system reduces to the trivial system of equations

$$\hat{y}_n'' = \hat{f}_n \quad n = 0, \dots, N-1.\tag{C.8}$$

The factors α and β , which in general can be different for each k , are zero for $k = 0$, so in this case we enforce

$$\sum_{n=0}^{N-1} \hat{y}_n = 0, \quad \sum_{n=0}^{N-1} \hat{y}_n(-1)^n = 0.\tag{C.9}$$

We use a Schur complement approach to solve the algebraic system of equations (C.7). We can write the system in block form as

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{y}}'' \\ \hat{\mathbf{y}}'_0 \\ \hat{\mathbf{y}}_0 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{f}} \\ \alpha \\ \beta \end{bmatrix}.\tag{C.10}$$

Here \mathbf{B} is $N \times 2$, \mathbf{C} is $2 \times N$, \mathbf{D} is 2×2 , and \mathbf{A} is an $N \times N$ *pentadiagonal* matrix. In particular, the matrix \mathbf{A} , referred to as the second integral matrix, has only three non zero diagonals ($i = j$ and $i = j \pm 2$) which allows to use a specialized factorization algorithm. In UAMMD we use the so-called *KBPENTA* in [144], modified for the special case of only three nonzero diagonals.¹

We start by solving the 2×2 system

$$(\mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{D}) \begin{bmatrix} \hat{y}_0 \\ \hat{y}'_0 \end{bmatrix} = \mathbf{C}\mathbf{A}^{-1}\mathbf{f} - \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\tag{C.11}$$

¹ The matrix \mathbf{A} is pre factorized (incurring an auxiliar storage of $3N$ elements for a given k). Using this our special pentadiagonal system can be solved in just $2N$ operations.

for \hat{y}_0 and \hat{y}'_0 . The 2×2 matrix in the left hand side as well as $\mathbf{C}\mathbf{A}^{-1}$ ($2 \times N$ size) are precomputed and stored for each k . Since the inverse of \mathbf{A} is only needed at the precomputation stage no special performance considerations are required for it².

Finally we obtain the coefficients $\hat{\mathbf{y}}'' = (\hat{y}_0'', \dots, \hat{y}_{N-1}'')$ by solving

$$\mathbf{A}\hat{\mathbf{y}}'' = \left(\mathbf{f} - \mathbf{B} \begin{bmatrix} \hat{y}_0 \\ \hat{y}'_0 \end{bmatrix} \right) \quad (\text{C.12})$$

using the pre-factorized modified *KBPENTA* algorithm.

Note that we need to solve this system for each wave number, k . Our solver requires the evaluation of several recurrent relations, making it mostly a serial algorithm not worth parallelizing. In order to take advantage of the **GPU**, we assign a thread to each k . As described above, much of the problem can be precomputed at the expense of memory storage (around $5N_k N$ values in total, being N_k the number of wave vectors³). Except for the $k = 0$ wave number, every solve is identical in operations, meaning that each thread will present the same memory access pattern in the, precomputed, auxiliar arrays. We take advantage of this by storing them in a coherent (strided) way, where a given element of an auxiliar array is stored contiguously (and ordered) for all wave numbers (so that the thread access pattern is cache friendly)⁴. Testing suggests that our **BVP** solver yields similar performance as the **FCT** for small N_k , becoming increasingly faster for increasing N_k (as expected given that our solver has linear scaling as opposed to the $O(N \log(N))$ operation count of the **FCT**).

U A M M D ' S O N L I N E D O C U M E N T A T I O N



This manuscript is based on UAMMD version 2.0 (which was released at the same time as this manuscript). Its contents, as well as the code examples, should be valid at least until version 3.0, the next release that might contain API breaking changes.

² In **UAMMD**, the CPU *getrf* BLAS square matrix inversion routine is used.

³ Since we use R2C **FFT** we will typically have $N_k = (N_x/2 + 1)N_y$.

⁴ I wrote a special **GPU** auxiliar storage handler for this that allows to switch between a strided and contiguous pattern. It can be found in the file *BVP-Memory.cuh* in **UAMMD**. In general a strided access provides a much better performance.

At the time of writing, the **UAMMD** codebase is hosted as a git repository on the github platform (currently the de facto standard git hosting platform), at <https://github.com/RaulPPelaez/UAMMD>. Note that the UAMMD repository offers a git branch for each major version. Be sure to check out the branch *v2.x* to have a version that corresponds to this manuscript. A summary of changes between releases can be found at the *CHANGELOG* file included in UAMMD's repository.

Platforms change over time and github might not exist at the time you are reading this¹. If that is the case and a google search does not help you locate its new hosting you have two options; either find me and request it directly or look in the wayback machine. At the time of writing, the internet archive has already scraped the uammd repository at least once, you can find the archived versions at http://web.archive.org/web/202*/https://github.com/RaulPPelaez/UAMMD.

This very manuscript is focused on the underlying theory behind the many algorithms exposed by the code as well as the design choices involved in its creation. However, it is not intended as a usage guide for **UAMMD** (although several code examples are included). A user wanting to learn how to use **UAMMD** can also benefit from the online resources. In particular, **UAMMD** offers three distinct sources of knowledge:

1. **Code examples.** The repository itself contains a plethora of organized usage examples showcasing the different **UAMMD** functionalities. In particular, the *basic_concepts* folder contains a series of tutorial-like examples intended as an introduction to the **UAMMD** ecosystem. This is the recommended next step into UAMMD for the reader of this manuscript.
2. **Wiki.** A wiki accompanying the codebase provides extensive information about all the available modules, as well as detailed information about the compilation process.
3. **The book,** “A Painless Introduction to Programming UAMMD Modules” by Marc Meléndez [145]. This book provides an introduction to **UAMMD** from the very ground up, while

¹ As a side note, **UAMMD** was included among github's Artic Code Vault program repositories. Thus, a physical copy of the codebase can be found 250 meters under the permafrost in a vault inside an abandoned coal mine in the Svalbard archipelago. This means that the codebase should be retrievable for at least the next thousand years.

making little assumptions about the readers familiarity with programming, complex fluids and their related numerical methods. Note, however, that at the time of writing [145] introduces UAMMD “v1.x”. Therefore some modifications might be required to translate its lessons into “v2.0” (the *CHANGELOG* can help with this).



THE TRANSVERSER AND POTENTIAL INTERFACES

E.1 THE TRANSVERSER INTERFACE

Many of the algorithms described in this manuscript require some kind of particle traversal. Say, for instance, that for each particle we want to visit the rest of the particles that are closer than a certain distance. Or simply all of the other particles. More generally, we might want to perform some kind of operation equivalent to a matrix-vector multiplication, for which in order to compute one element of the result, the vector needs to go through a row of the matrix. In these cases, a *Transverser*¹ is used.

A *Transverser* holds information about what to do with a pair of particles, what information is needed to compute this interaction, and what to do when a particle has interacted with all pairs it is involved in.

Being such a general concept, a *Transverser* is used as a template argument, and therefore cannot be a base virtual class that can be inherited. This is why it is a "concept". No assumption can be made about the return types of each function, or the input parameters, the only common things are the function names.

For each particle to be processed the *Transverser* will be called for:

- Setting the initial value of the interaction result (function *zero*)
- Fetching the necessary data to process a pair of particles (function *getInfo*)

¹ The word *Transverser* was chosen to convey that it is used to traverse and transform

- Compute the interaction between the particle and each of its neighbours (function *compute*)
- Accumulate/reduce the result for each neighbour (function *accumulate*)
- Set/write/handle the accumulated result for all neighbours (function *set*)

The same *Transverser* instance will be used to process every particle in an arbitrary order. Therefore, the Transverser must not assume it is bound to a specific particle.

The *Transverser* interface requires a given class/struct to provide the following public device (unless, “prepare”, that must be a host function) member functions:

- Compute `compute(real4 position_i, real4 position_j, Info info_i, Info info_j);`

For a pair of particles characterized by position and info this function must return the result from the interaction for that pair of particles. The last two arguments must be present only when *getInfo* is defined. The returning type, *Compute*, must be a POD type (just an aggregate of plain types), for example a real when computing energy.

- `void set(int particle_index, Compute &total);`

After calling *compute* for all neighbours this function will be called with the contents of "total" after the last call to "accumulate". Can be used to, for example, write the final result to main memory.

- `Compute zero();`

This function returns the initial value of the computation, for example 0,0,0 when computing the force. The returning type, *Compute*, must be a POD type (just an aggregate of plain types), for example a real when computing energy. Furthermore it must be the same type returned by the "compute" member. This function is optional and defaults to zero initialization (it will return *Compute()* which works even for POD types).

- `Info getInfo(int particle_index);`

Will be called for each particle to be processed and returns the per-particle data necessary for the interaction with another particle (except the position which is always available). For example the mass in a gravitational interaction or the particle index for some custom interaction. The returning type, Info, must be a POD type (just an aggregate of plain types), for example a real for gravitation. This function is optional and if not present it is assumed the only per-particle data required is the position. In this case the function "compute" must only have the first two arguments.

- **void accumulate**(Compute &total, **const** Compute ¤t);

This function will be called after "compute" for each neighbour with its result and the accumulated result. It is expected that this function modifies "total" as necessary given the new data in "current". The first time it is called "total" will be have the value as given by the "zero" function. This function is optional and defaults to summation: total = total + current. Notice that this will fail for non trivial types.

- **void prepare**(std::shared_ptr<ParticleData> pd);

This function will be called one time on the CPU side just before processing the particles. This function is optional and defaults to simply nothing.

Example code 43 contains a very bare-bones instance of a *Transverser*. In particular, *NeighbourCounter* relies on as much default behavior as possible, presenting only a *compute* and *set* functions. If we apply the *NeighbourCounter Transverser* to one of the neighbour lists in [UAMMD](#) (see chapter 11), the output ("nneigh" array) will hold, for each particle, the number of neighbour particles.

```
struct NeighbourCounter{
    int *nneigh;
    real rc;
    Box box;
    NeighbourCounter(Box i_box, real i_rc, int *nneigh):
        rc(i_rc), box(i_box),
        nneigh(nneigh){}
    //There is no "zero" function so the total result
    → starts being 0.
```

```

//For each pair computes counts a neighbour
//if the particle is closer than rcut
__device__ auto compute(real4 pi, real4 pj){
    const real3 rij =
        → box.apply_pbc(make_real3(pj)-make_real3(pi));
    const real r2 = dot(rij, rij);
    if(r2>0 and r2< rc*rc){
        return 1;
    }
    return 0;
}
//There is no "accumulate"
// the result of "compute" is added every time.
//The "set" function will be called with the
→ accumulation
// of the result of "compute" for all neighbours.
__device__ void set(int index, int total){
    nneigh[index] = total;
}
};

```

Source Code 43: A *Transverser* that counts the number of neighbours of each particle.

Alternatively, if we apply the *Transverser* in the code example 43 to the *NBody* module (see chapter 9) each particle will go through every other one, and thus all the elements of the *NeighbourCounter* output will be equal to the total number of particles.

E.2 THE POTENTIAL INTERFACE

This interface is just a connection between the *Transverser* and *Interactor* concepts. Additionally, *Potential* aids with one limitation of the CUDA programming language and **GPU** programming in general. On the one hand, register memory in a **GPU** is quite limited, so it is not a good idea to use large objects in a kernel. On the other, there are some technical details that prevent certain objects from existing in a **GPU** kernel. For example, objects are provided by value to a kernel, which can incur undesired copies and/or destructors being called. Thus, it is sometimes worth it to make a conceptual and programmatic separation between CPU and **GPU** objects. In this regard, *Transversers* are **GPU** objects, while *Interactors* or *Potentials* are meant to be used in the CPU. Furthermore, while *Transverser* describes a very general

computation, *Potential* only holds the logic on how to compute forces, energies and/or virials. *Potentials* are used to provide force-, energy- and/or virial-calculating *Transversers* to an *Interactor*² (alternatively, the *Transverser* provided by a *Potential* could be used with a neighbour list directly). In turn, this *Interactor* can be either used on its own to compute directly, or provided to a *Integrator*.

The *Potential* interface is straightforward, requiring only two functions:

- `real getCutOff();`

This function must return the highest cut off distance required by the interaction.

- `Transverser getTransverser(Interactor::Computables comp, Box box, std::shared_ptr<ParticleData> pd);`

This function must provide an instance of a *Transverser* that, using the provided *ParticleData* and *Box* instances, computes anything requested by the *Computables* list (mainly forces, energies and/or virials, see chapter 8.1 for more information). The return type of this function, called *Transverser* here, can be any valid *Transverser* (see Appendix E.1) with only one restriction: The return type of the *compute* function (called *Compute* in Appendix E.1) must be *ForceEnergyVirial* (a simple POD type that holds members for the force, energy and virial). See example code 44.

² In particular, the *PairForces* module (see chapter 10) needs a *Potential* encoding the specific particle interaction.

```

//Some functions to compute forces/energies
__device__ real lj_force(real r2){
    const real invr2 = real(1.0)/r2;
    const real invr6 = invr2*invr2*invr2;
    const real fmoddinvr = (real(-48.0)*invr6 +
    ↵ real(24.0))*invr6*invr2;
    return fmoddinvr;
}

__device__ real lj_energy(real r2){
    const real invr2 = real(1.0)/r2;
    const real invr6 = invr2*invr2*invr2;
    return real(4.0)*(invr6 - real(1.0))*invr6;
}

//A Transverser for computing, energy, virial and
→ force (or just some of them).
//It is the simplest form of Transverser, as it
→ only provides the "compute" and "set"
→ functions
//When constructed, if the i_force, i_energy or
→ i_virial pointers are null that computation
→ will be avoided.
struct LJTransverser{
    real4 *force;
    real *virial;
    real* energy;
    Box box;
    real rc;
    LJTransverser(Box i_box, real i_rc, real4*
    → i_force, real* i_energy, real* i_virial):
    box(i_box), rc(i_rc), force(i_force),
    → virial(i_virial), energy(i_energy){
        //All members will be available in the device
        → functions
    }
    //For each pair computes and returns the LJ force
    → and/or energy and/or virial based only on the
    → positions
    __device__ ForceEnergyVirial compute(real4 pi,
    → real4 pj){
        const real3 rij =
        → box.apply_pbc(make_real3(pj)-make_real3(pi));
        const real r2 = dot(rij, rij);
        if(r2>0 and r2< rc*rc){

```

```

    real3 f;
    real v, e;
    f = (force or
        → virial)?lj_force(r2)*rij:real3();
    v = virial?dot(f, rij):0;
    e = energy?lj_energy(r2):0;
    return {f,e,v};
}
return {};
}

//Note that we are making use of the default
→ behaviors by not defining an accumulate or
→ zero functions.
__device__ void set(int id, ForceEnergyVirial
→ total){
    //Write the total result to memory if the
→ pointer was provided
    if(force) force[id] += make_real4(total.force,
        → 0);
    if(virial) virial[id] += total.virial;
    if(energy) energy[id] += total.energy;
}
};

//A simple LJ Potential, can compute force, energy,
→ virial or all at the same time using the above
→ Transverser.
struct SimpleLJ{
    real rc = 2.5;
    //A function returning the maximum required cut off
→ for the interaction
    real getCutOff(){
        return rc;
    }
    //This function is required to provide a
→ Transverser that has the ability to compute the
→ requested Computables.
    auto getTransverser(Interactor::Computables comp,
    Box box,
    std::shared_ptr<ParticleData> pd){
        auto force = comp.force?pd->getForce(access::gpu,
            → access::readwrite).raw():nullptr;
        auto energy =
            → comp.energy?pd->getEnergy(access::gpu,
            → access::readwrite).raw():nullptr;
    }
};

```

```

    auto virial =
    ↵ comp.virial?pd->getVirial(access::gpu,
    ↵ access::readwrite).raw():nullptr;
    return LJTransverser(box, rc, force, energy,
    ↵ virial);
}

};

```

Source Code 44: An example *Potential* that computes Lennard-Jones forces, energies and/or virials. For simplicity, all relevant parameters are hardcoded here. In particular, $\sigma_{lj} = 1$, $\epsilon_{lj} = 1$ and the cut off is set at $r_c = 2.5\sigma = 2.5$.

The potential here defined (called *SimpleLJ*) calculates forces, energies and virials. Note, however, that it does so only when provided to a *PairForces Interactor* (see chapter 10) and, subsequently, to an *Integrator*. In other words, we use *Potentials* to define an *Interactor*, which will be used by an *Integrator* to calculate forces, energies, etc.

F

A FULL UAMMD SIMULATION EXAMPLE

The UAMMD repository contains many examples encoding full-fledged simulations¹ making use of the tools described in this manuscript. Nonetheless, it is worth to lay out here a putting-it-all-together example that shows how to set up and run a simulation from scratch using UAMMD. In particular, source code 45 makes use of some of the example codes written throughout this manuscript to craft a simulation of a gas of LJ atoms (for instance, modeling argon).

```

#include <uammd.cuh>
using namespace uammd;
//In order to use the rest of the examples, we need
↪ to define all the necessary parameters inside the
↪ Parameters structure.
//For simplicity, let us simply hardcode here the
↪ parameters

```

¹ Most notably, the example *generic_md.cu* shows off virtually every module and concept described in this thesis.

```
struct Parameters{
    real dt = 0.01;
    real temperature = 0.1;
    real friction = 1.0;
    //A periodic cubic box of size L=32
    Box box = Box({32, 32, 32});
};

struct UAMMD{
    std::shared_ptr<ParticleData> pd;
    Parameters par;
};

//Include here the source codes listed in the
→ caption

int main(){
    UAMMD sim;
    //We start by initializing ParticleData
    int numberParticles = 2;
    sim.pd =
        → std::make_shared<ParticleData>(numberParticles);
    //Now we can set the positions of the two
    → particles
    { //Our LJ Potential example hardcodes sigma = 1,
    → lets place our particles at a distance of
    → 2sigma=2.
        auto positions = sim.pd->getPos(access::cpu,
        → access::write);
        positions[0] = {0,0,0,0};
        positions[1] = {2.0,0,0,0};
    } //Remember that the handle, positions, must be
    → destroyed ASAP
    //We can now create our integrator using the
    → function we wrote when describing Langevin
    → dynamics
    auto nvt = createIntegratorVerletNVT(sim);
    //Using the SimpleLJ Potential defined in a
    → previous example we can create a PairForces
    → Interactor
    auto ljpot = SimpleLJ(); //The SimpleLJ constructor
    → does not have any arguments
    auto lj = createPairForcesWithPotential(sim,
    → ljpot);
    //We now add the newly created interactor to nvt.
    nvt->addInteractor(lj);
    //Any number of Interactors can be added this way.
```

```

//Finally, we can advance the simulation as many
→ times as needed, one dt at a time:
int numberSteps = 10;
for(int step = 0; step<numberSteps; step++){
    nvt->forwardTime();
    //Whenever needed, we can request the particles
    → state via ParticleData, for instance, to
    → print the positions.
{
    auto positions = sim.pd->getPos(access::cpu,
    → access::read);
    std::cout<<"First particle is at:
    → "<<positions[0];
    std::cout<<" at time
    → "<<step*sim.par.dt<<std::endl;
}
}
return 0;
}

```

Source Code 45: A simulation of LJ particles liquid using UAMMD. Source codes 16, 8 and 44 serve as a preamble (must be included or copy/pasted) here. To ease initialization, only two particles are simulated.

G

LIST OF CURRENTLY IMPLEMENTED MODULES

For convenience, we present here a list of all the modules available in [UAMMD](#) at the time of writing. Each of them can be created and used on its own. Alternatively, several of them can be joined to construct a simulation.

Most of the modules listed below have a corresponding chapter in this manuscript. The ones that have not been described are available in [UAMMD](#)'s online resources.

G.1 INTEGRATORS

- Molecular Dynamics
- Langevin Dynamics

- Dissipative Particle Dynamics
- Smoothed-particle Hydrodynamics
- Brownian Dynamics
- Brownian Hydrodynamics
 - Open Boundaries
 - * Cholesky
 - * Lanczos
 - Triply Periodic
 - * Force Coupling Method
 - * Positively Split Ewald
- Fluctuating Hydrodynamics
 - Fluctuating Immersed Boundary
 - Inertial Coupling Method
 - Quasi 2D
- Lattice Boltzmann

G.2 INTERACTORS

- Short-ranged
- Long-ranged (nbody)
- Triply and Doubly periodic electrostatics
- Bonds (particle-particle, particle-point, angular, torsional...)
- External interactions

G.3 OTHER MODULES

- Neighbour lists
 - Cell list
 - Verlet list
 - LBVH list

- Immersed Boundary Method (spreading and interpolation between markers and a grid)
- Lanczos iterative Krylov decomposition
- Boundary Value Problem solver
- Tabulated functions

B I B L I O G R A P H Y

- [1] Jodi A Hadden et al. “All-atom molecular dynamics of the HBV capsid reveals insights into biological function and cryo-EM resolution limits.” In: *eLife* 7 (2018). Ed. by Axel T Brunger, e32478. ISSN: 2050-084X. DOI: [10.7554/elife.32478](https://doi.org/10.7554/elife.32478).
- [2] Jean-David Rochaix. “Dynamic Modeling of a 100-Million-Atom Organelle at the Source of Life.” In: *Cell* 179.5 (2019), pp. 1012–1014. DOI: [10.1016/j.cell.2019.10.023](https://doi.org/10.1016/j.cell.2019.10.023).
- [3] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [4] C. J. Thompson, Sahngyun Hahn, and M. Oskin. “Using modern graphics architectures for general-purpose computing: a framework and analysis.” In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings*. 2002, pp. 306–317. DOI: [10.1109/MICRO.2002.1176259](https://doi.org/10.1109/MICRO.2002.1176259).
- [5] Philippe Colantoni, Nabil Boukala, and Jérôme Da-Rugna. “Fast and Accurate Color Images Processing Using 3D Graphics Cards.” In: 2003, pp. 383–390.
- [6] Jose Gonzalez-Mora, Nicolas Guil, and Emilio L. Zapata. “Using Graphic Processing Units for Tracking Algorithms.” In: *AIP Conference Proceedings* 860.1 (2006), pp. 310–317. DOI: [10.1063/1.2361233](https://doi.org/10.1063/1.2361233).
- [7] E. Scott Larsen and David McAllister. “Fast Matrix Multiplies Using Graphics Hardware.” In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC ’01. Denver, Colorado: Association for Computing Machinery, 2001, p. 55. ISBN: 158113293X. DOI: [10.1145/582034.582089](https://doi.org/10.1145/582034.582089).
- [8] Jeff Bolz et al. “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.” In: *ACM Trans. Graph.* 22.3 (2003), pp. 917–924. ISSN: 0730-0301. DOI: [10.1145/882262.882364](https://doi.org/10.1145/882262.882364).

- [9] Jeff Bolz et al. “Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.” In: *ACM Trans. Graph.* 22.3 (2003), pp. 917–924. ISSN: 0730-0301. DOI: [10.1145/882262.882364](https://doi.org/10.1145/882262.882364).
- [10] Zhe Fan et al. “GPU Cluster for High Performance Computing.” In: *SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. 2004, pp. 47–47. DOI: [10.1109/SC.2004.26](https://doi.org/10.1109/SC.2004.26).
- [11] Zhongwen Luo et al. “Self-Organizing Maps computing on Graphic Process Unit.” In: 2005, pp. 557–562.
- [12] Christian-A. Bohn. “Kohonen Feature Mapping through Graphics Hardware.” In: *In Proceedings of Int. Conf. on Compu. Intelligence and Neurosciences*. 1998, pp. 64–67.
- [13] Edgar Luttmann et al. “Accelerating molecular dynamic simulation on the cell processor and Playstation 3.” In: *Journal of Computational Chemistry* 30.2 (2009), pp. 268–274. DOI: <https://doi.org/10.1002/jcc.21054>.
- [14] John Nickolls et al. “Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?” In: *Queue* 6.2 (2008), pp. 40–53. ISSN: 1542-7730. DOI: [10.1145/1365490.1365500](https://doi.org/10.1145/1365490.1365500).
- [15] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.” In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73. DOI: [10.1109/MCSE.2010.69](https://doi.org/10.1109/MCSE.2010.69).
- [16] CAPS Enterprise. *Cray Inc. and NVIDIA and the Portland Group: The openacc application programming interface, v1.0 (November 2011)*.
- [17] Cuthbert C. Hurd. “A Note on Early Monte Carlo Computations and Scientific Meetings.” In: *Annals of the History of Computing* 7.2 (1985), pp. 141–155. DOI: [10.1109/MAHC.1985.10019](https://doi.org/10.1109/MAHC.1985.10019).
- [18] A.M. Johansen. “Monte Carlo Methods.” In: *International Encyclopedia of Education (Third Edition)*. Ed. by Penelope Peterson, Eva Baker, and Barry McGaw. Third Edition. Oxford: Elsevier, 2010, pp. 296–303. ISBN: 978-0-08-044894-7. DOI: <https://doi.org/10.1016/B978-0-08-044894-7.01543-8>.

- [19] Jacopo De Tullio. “The Fermi-Pasta-Ulam model: the birth of numerical simulation.” In: *Lettera Matematica* 4.1 (2016), pp. 41–48. ISSN: 2281-5937. DOI: [10.1007/s40329-016-0126-4](https://doi.org/10.1007/s40329-016-0126-4).
- [20] Bernard R. Brooks et al. “CHARMM: A program for macromolecular energy, minimization, and dynamics calculations.” In: *Journal of Computational Chemistry* 4.2 (1983), pp. 187–217. DOI: <https://doi.org/10.1002/jcc.540040211>.
- [21] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. “GROMACS: A message-passing parallel molecular dynamics implementation.” In: *Computer Physics Communications* 91.1 (1995), pp. 43–56. ISSN: 0010-4655. DOI: [https://doi.org/10.1016/0010-4655\(95\)00042-E](https://doi.org/10.1016/0010-4655(95)00042-E).
- [22] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. “General purpose molecular dynamics simulations fully implemented on graphics processing units.” In: *Journal of Computational Physics* 227.10 (2008), pp. 5342–5359. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2008.01.047>.
- [23] Harold R. Warner. “Kinetic Theory and Rheology of Dilute Suspensions of Finitely Extendible Dumbbells.” In: *Industrial & Engineering Chemistry Fundamentals* 11.3 (1972), pp. 379–387. DOI: [10.1021/i160043a017](https://doi.org/10.1021/i160043a017).
- [24] Philip M. Morse. “Diatom Molecules According to the Wave Mechanics. II. Vibrational Levels.” In: *Phys. Rev.* 34 (1 1929), pp. 57–64. DOI: [10.1103/PhysRev.34.57](https://doi.org/10.1103/PhysRev.34.57).
- [25] Diego Camargo et al. “Nanoscale hydrodynamics near solids.” In: *The Journal of Chemical Physics* 148.6 (2018), p. 064107. DOI: [10.1063/1.5010401](https://doi.org/10.1063/1.5010401).
- [26] Carmen Hijón et al. “Mori–Zwanzig formalism as a practical computational tool.” In: *Faraday Discuss.* 144 (0 2010), pp. 301–322. DOI: [10.1039/B902479B](https://doi.org/10.1039/B902479B).
- [27] J.K.G. Dhont. *An Introduction to Dynamics of Colloids*. ISSN. Elsevier Science, 1996. ISBN: 9780080535074.
- [28] H. Risken and T. Frank. *The Fokker-Planck Equation: Methods of Solution and Applications*. Springer Series in Synergetics. Springer Berlin Heidelberg, 2012. ISBN: 9783642615443. DOI: [10.1007/978-3-642-96807-5](https://doi.org/10.1007/978-3-642-96807-5).

- [29] M. Reza Rahimi Tabar. “Kramers–Moyal Expansion and Fokker–Planck Equation.” In: *Analysis and Data-Based Reconstruction of Complex Nonlinear Dynamical Systems: Using the Methods of Stochastic Processes*. Cham: Springer International Publishing, 2019, pp. 19–29. ISBN: 978-3-030-18472-8. DOI: [10.1007/978-3-030-18472-8_3](https://doi.org/10.1007/978-3-030-18472-8_3).
- [30] José M. Ortiz de Zárate and Jan V. Sengers. “Chapter 3 - Fluctuations in fluids in thermodynamic equilibrium.” In: *Hydrodynamic Fluctuations in Fluids and Fluid Mixtures*. Ed. by José M. Ortiz de Zárate and Jan V. Sengers. Amsterdam: Elsevier, 2006, pp. 39–62. ISBN: 978-0-444-51515-5. DOI: <https://doi.org/10.1016/B978-044451515-5/50003-X>.
- [31] S.N. Cohen and R.J. Elliott. *Stochastic Calculus and Applications*. Probability and Its Applications. Springer New York, 2015. ISBN: 9781493936816.
- [32] A. Einstein. “Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen.” In: *Annalen der Physik* 322.8 (1905), pp. 549–560. DOI: [10.1002/andp.19053220806](https://doi.org/10.1002/andp.19053220806).
- [33] L. Greengard and V. Rokhlin. “A Fast Algorithm for Particle Simulations.” In: *J. Comput. Phys.* 73.2 (1987), pp. 325–348. ISSN: 0021-9991. DOI: [10.1016/0021-9991\(87\)90140-9](https://doi.org/10.1016/0021-9991(87)90140-9).
- [34] H. Nguyen and NVIDIA Corporation. *GPU Gems 3*. Lab Companion Series v. 3. Addison-Wesley, 2008. ISBN: 9780321515261.
- [35] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013. ISBN: 9780321809469.
- [36] Monika Thol et al. “Equation of State for the Lennard-Jones Truncated and Shifted Model Fluid.” In: *International Journal of Thermophysics* 36.1 (2015), pp. 25–43. ISSN: 1572-9567. DOI: [10.1007/s10765-014-1764-4](https://doi.org/10.1007/s10765-014-1764-4).
- [37] Steve Plimpton. “Fast Parallel Algorithms for Short-Range Molecular Dynamics.” In: *Journal of Computational Physics* 117.1 (1995), pp. 1–19. ISSN: 0021-9991. DOI: <https://doi.org/10.1006/jcph.1995.1039>.

- [38] Michael P. Allen and Dominic J. Tildesley. *Computer Simulation of Liquids: Second Edition*. eng. 2nd ed. Oxford: Oxford University Press, 2017, p. 640. ISBN: 9780198803195. DOI: [10.1093/oso/9780198803195.001.0001](https://doi.org/10.1093/oso/9780198803195.001.0001).
- [39] J. M. Domínguez et al. “Neighbour lists in smoothed particle hydrodynamics.” In: *International Journal for Numerical Methods in Fluids* 67.12 (2011), pp. 2026–2042. DOI: <https://doi.org/10.1002/fld.2481>.
- [40] Michael P. Howard et al. “Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units.” In: *Computer Physics Communications* 203 (2016), pp. 45–52. ISSN: 0010-4655. DOI: <https://doi.org/10.1016/j.cpc.2016.02.003>.
- [41] W. Michael Brown et al. “Implementing molecular dynamics on hybrid high performance computers – short range forces.” In: *Computer Physics Communications* 182.4 (2011), pp. 898–911. ISSN: 0010-4655. DOI: <https://doi.org/10.1016/j.cpc.2010.12.021>.
- [42] Yu-Hang Tang and George Em Karniadakis. “Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications.” In: *Computer Physics Communications* 185.11 (2014), pp. 2809–2822. ISSN: 0010-4655. DOI: <https://doi.org/10.1016/j.cpc.2014.06.015>.
- [43] G. Peano. “Sur une courbe, qui remplit toute une aire plane.” In: *Mathematische Annalen* 36 (1890), pp. 157–160. ISSN: 1432-1807. DOI: [10.1007/BF01199438](https://doi.org/10.1007/BF01199438).
- [44] David Hilbert. *Dritter Band: Analysis. Grundlagen der Mathematik. Physik Verschiedenes*. Vol. 36. Springer, Berlin, Heidelberg, 1935, pp. 157–160. ISBN: 9783662384527. DOI: [10.1007/978-3-662-38452-7_1](https://doi.org/10.1007/978-3-662-38452-7_1).
- [45] Guy Macdonald Morton. “A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing.” In: (1966).
- [46] Linh Ha, Jens Krüger, and Claudio Silva. “Fast 4-way parallel radix sorting on GPUs.” In: *Comput. Graph. Forum* 28 (2009), pp. 2368–2378. DOI: [10.1111/j.1467-8659.2009.01542.x](https://doi.org/10.1111/j.1467-8659.2009.01542.x).

- [47] Dhirendra Singh, Ishan Joshi, and Jaytrilok Choudhary. “Survey of GPU Based Sorting Algorithms.” In: *International Journal of Parallel Programming* 46 (2018). DOI: [10.1007/s10766-017-0502-5](https://doi.org/10.1007/s10766-017-0502-5).
- [48] Duane Merrill and Andrew Grimshaw. “High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing.” In: *Parallel Processing Letters* 21 (2011), pp. 245–272. DOI: [10.1142/S0129626411000187](https://doi.org/10.1142/S0129626411000187).
- [49] Duane Merrill and NVIDIA Corporation. *CUB: Cooperative primitives for CUDA C++*. 2011.
- [50] Michael P. Howard et al. “Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units.” In: *Computational Materials Science* 164 (2019), pp. 139–146. ISSN: 0927-0256. DOI: <https://doi.org/10.1016/j.commatsci.2019.04.004>.
- [51] Roberto Torres, Pedro J. Martín, and Antonio Gavilanes. “Ray Casting Using a Roped BVH with CUDA.” In: *Proceedings of the 25th Spring Conference on Computer Graphics. SCCG ’09*. Budmerice, Slovakia: Association for Computing Machinery, 2009, pp. 95–102. ISBN: 9781450307697. DOI: [10.1145/1980462.1980483](https://doi.org/10.1145/1980462.1980483).
- [52] Tero Karras. “Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees.” In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics. EGHH-HPG’12*. Paris, France: Eurographics Association, 2012, pp. 33–37. ISBN: 9783905674415.
- [53] Christian Grossmann, Hans-Görg Roos, and Martin Stynes. *Numerical treatment of partial differential equations. Revised translation of the 3rd German edition of ‘Numerische Behandlung partieller Differentialgleichungen’ by Martin Stynes*. 2007. ISBN: 978-3-540-71582-5. DOI: [10.1007/978-3-540-71584-9](https://doi.org/10.1007/978-3-540-71584-9).
- [54] John C. Butcher. *Numerical Methods for Ordinary Differential Equations*. Third. Hoboken, New Jersey: Wiley, 2016. ISBN: 9781119121503. DOI: [10.1002/9781119121534](https://doi.org/10.1002/9781119121534).

- [55] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric numerical integration. Structure-preserving algorithms for ordinary differential equations.* 2nd ed. Vol. 31. 2006. ISBN: 3-540-30663-3. DOI: [10.1007/3-540-30666-8](https://doi.org/10.1007/3-540-30666-8).
- [56] Melville S. Green. “Markoff Random Processes and the Statistical Mechanics of Time-Dependent Phenomena. II. Irreversible Processes in Fluids.” In: *The Journal of Chemical Physics* 22.3 (1954), pp. 398–413. DOI: [10.1063/1.1740082](https://doi.org/10.1063/1.1740082).
- [57] Ryogo Kubo. “Statistical-Mechanical Theory of Irreversible Processes. I. General Theory and Simple Applications to Magnetic and Conduction Problems.” In: *Journal of the Physical Society of Japan* 12.6 (1957), pp. 570–586. DOI: [10.1143/JPSJ.12.570](https://doi.org/10.1143/JPSJ.12.570).
- [58] V. Feller and W. Feller. *An Introduction to Probability Theory and Its Applications, Volume 1.* A Wiley publication in mathematical statistics. Wiley, 1968. ISBN: 9780471257080.
- [59] BURKHARD DÜNWEG and WOLFGANG PAUL. “BROWNIAN DYNAMICS SIMULATIONS WITHOUT GAUSSIAN RANDOM NUMBERS.” In: *International Journal of Modern Physics C* 02.03 (1991), pp. 817–827. DOI: [10.1142/S0129183191001037](https://doi.org/10.1142/S0129183191001037).
- [60] Axel Brünger, Charles L. Brooks, and Martin Karplus. “Stochastic boundary conditions for molecular dynamics simulations of ST2 water.” In: *Chemical Physics Letters* 105.5 (1984), pp. 495–500. ISSN: 0009-2614. DOI: [https://doi.org/10.1016/0009-2614\(84\)80098-6](https://doi.org/10.1016/0009-2614(84)80098-6).
- [61] Wei Wang and Robert D. Skeel. “Analysis of a few numerical integration methods for the Langevin equation.” In: *Molecular Physics* 101.14 (2003), pp. 2149–2156. DOI: [10.1080/0026897031000135825](https://doi.org/10.1080/0026897031000135825).
- [62] Niels Grønbech-Jensen and Oded Farago. “A simple and effective Verlet-type algorithm for simulating Langevin dynamics.” In: *Molecular Physics* 111.8 (2013), pp. 983–991. DOI: [10.1080/00268976.2012.760055](https://doi.org/10.1080/00268976.2012.760055).
- [63] J. Sablić, R. Delgado-Buscalioni, and M. Praprotnik. “Application of the Eckart frame to soft matter: rotation of star polymers under shear flow.” In: *Soft Matter* 13.39 (2017), pp. 6988–7000.

- [64] Robert D. Groot and Patrick B. Warren. “Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation.” In: *The Journal of Chemical Physics* 107.11 (1997), pp. 4423–4435. DOI: [10.1063/1.474784](https://doi.org/10.1063/1.474784).
- [65] P Español and P Warren. “Statistical Mechanics of Dissipative Particle Dynamics.” In: *Europhysics Letters (EPL)* 30.4 (1995), pp. 191–196. DOI: [10.1209/0295-5075/30/4/001](https://doi.org/10.1209/0295-5075/30/4/001).
- [66] Benedict Leimkuhler and Xiaocheng Shang. “On the numerical treatment of dissipative particle dynamics and related systems.” In: *Journal of Computational Physics* 280 (2015), pp. 72–95. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2014.09.008>.
- [67] I Pagonabarraga, M. H. J Hagen, and D Frenkel. “Self-consistent dissipative particle dynamics algorithm.” In: *Europhysics Letters (EPL)* 42.4 (1998), pp. 377–382. DOI: [10.1209/epl/i1998-00258-6](https://doi.org/10.1209/epl/i1998-00258-6).
- [68] Gerhard Besold et al. “Towards better integrators for dissipative particle dynamics simulations.” In: *Phys. Rev. E* 62 (6 2000), R7611–R7614. DOI: [10.1103/PhysRevE.62.R7611](https://doi.org/10.1103/PhysRevE.62.R7611).
- [69] M. Meléndez et al. “Optofluidic control of the dispersion of nanoscale dumbbells.” In: *Phys. Rev. E* 99 (2 2019), p. 022603. DOI: [10.1103/PhysRevE.99.022603](https://doi.org/10.1103/PhysRevE.99.022603).
- [70] Timothy A Westwood, Blaise Delmotte, and Eric E Keaveny. *A generalised drift-correcting time integration scheme for Brownian suspensions of rigid particles with arbitrary shape*. 2021.
- [71] Raúl P Peláez et al. “Hydrodynamic fluctuations in quasi-two dimensional diffusion.” In: *Journal of Statistical Mechanics: Theory and Experiment* 2018.6 (2018), p. 063207. DOI: [10.1088/1742-5468/aac2fb](https://doi.org/10.1088/1742-5468/aac2fb).
- [72] Florencio Balboa Usabiaga et al. “Hydrodynamics of Suspensions of Passive and Active Rigid Particles: A Rigid Multiblob Approach.” In: *Communications in Applied Mathematics and Computational Science* 11 (2016). DOI: [10.2140/camcos.2016.11.217](https://doi.org/10.2140/camcos.2016.11.217).
- [73] S. Panzuela and R. Delgado-Buscalioni. “Solvent Hydrodynamics Enhances the Collective Diffusion of Membrane Lipids.” In: *Phys. Rev. Lett.* 121 (4 2018), p. 048101. DOI: [10.1103/PhysRevLett.121.048101](https://doi.org/10.1103/PhysRevLett.121.048101).

- [74] Desmond J. Higham. “An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations.” In: *SIAM Review* 43.3 (2001), pp. 525–546. DOI: [10.1137/S0036144500378302](https://doi.org/10.1137/S0036144500378302).
- [75] P.E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*. Stochastic Modelling and Applied Probability. Springer Berlin Heidelberg, 2011. ISBN: 978-3-662-12616-5.
- [76] Florencio Balboa Usabiaga, Blaise Delmotte, and Aleksandar Donev. “Brownian dynamics of confined suspensions of active microrollers.” In: *The Journal of Chemical Physics* 146.13 (2017), p. 134104. DOI: [10.1063/1.4979494](https://doi.org/10.1063/1.4979494).
- [77] Benedict Leimkuhler, Charles Matthews, and Gabriel Stoltz. “The computation of averages from equilibrium and nonequilibrium Langevin molecular dynamics.” In: *IMA Journal of Numerical Analysis* 36.1 (2015), pp. 13–79. ISSN: 0272-4979. DOI: [10.1093/imanum/dru056](https://doi.org/10.1093/imanum/dru056).
- [78] B. Leimkuhler, C. Matthews, and M. V. Tretyakov. “On the long-time integration of stochastic gradient systems.” In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 470.2170 (2014), p. 20140120. DOI: [10.1098/rspa.2014.0120](https://doi.org/10.1098/rspa.2014.0120).
- [79] Steven Delong et al. “Temporal integrators for fluctuating hydrodynamics.” In: *Phys. Rev. E* 87 (3 2013), p. 033302. DOI: [10.1103/PhysRevE.87.033302](https://doi.org/10.1103/PhysRevE.87.033302).
- [80] Nawaf Bou-Rabee and Eric Vanden-Eijnden. “Pathwise accuracy and ergodicity of metropolized integrators for SDEs.” In: *Communications on Pure and Applied Mathematics* 63.5 (2010), pp. 655–696. DOI: <https://doi.org/10.1002/cpa.20306>.
- [81] G. O. Roberts and R. L. Tweedie. “Geometric Convergence and Central Limit Theorems for Multidimensional Hastings and Metropolis Algorithms.” In: *Biometrika* 83.1 (1996), pp. 95–110. ISSN: 00063444.
- [82] JE Besag. “Comments on “Representations of knowledge in complex systems” by U. Grenander and MI Miller.” In: *J. Roy. Statist. Soc. Ser. B* 56 (1994), pp. 591–592.

- [83] Florencio Balboa Usabiaga et al. “Inertial coupling method for particles in an incompressible fluctuating fluid.” In: *Computer Methods in Applied Mechanics and Engineering* 269 (2014), pp. 139–172. ISSN: 0045-7825. DOI: <https://doi.org/10.1016/j.cma.2013.10.029>.
- [84] P. Mazur and D. Bedeaux. “A generalization of Faxén’s theorem to nonsteady motion of a sphere through an incompressible fluid in arbitrary flow.” In: *Physica* 76.2 (1974), pp. 235–246. ISSN: 0031-8914. DOI: [https://doi.org/10.1016/0031-8914\(74\)90197-9](https://doi.org/10.1016/0031-8914(74)90197-9).
- [85] Maciej Lisicki. *Four approaches to hydrodynamic Green’s functions – the Oseen tensors*. 2013.
- [86] F. Balboa Usabiaga and R. Delgado-Buscalioni. “Minimal model for acoustic forces on Brownian particles.” In: *Phys. Rev. E* 88 (6 2013), p. 063304. DOI: [10.1103/PhysRevE.88.063304](https://doi.org/10.1103/PhysRevE.88.063304).
- [87] Steven Delong et al. “Brownian dynamics without Green’s functions.” In: *The Journal of Chemical Physics* 140.13 (2014), p. 134110. DOI: [10.1063/1.4869866](https://doi.org/10.1063/1.4869866).
- [88] Jens Rotne and Stephen Prager. “Variational Treatment of Hydrodynamic Interaction in Polymers.” In: *The Journal of Chemical Physics* 50.11 (1969), pp. 4831–4837. DOI: [10.1063/1.1670977](https://doi.org/10.1063/1.1670977).
- [89] Hiromi Yamakawa. “Transport Properties of Polymer Chains in Dilute Solution: Hydrodynamic Interaction.” In: *The Journal of Chemical Physics* 53.1 (1970), pp. 436–443. DOI: [10.1063/1.1673799](https://doi.org/10.1063/1.1673799).
- [90] Eligiusz Wajnryb et al. “Generalization of the Rotne–Prager–Yamakawa mobility and shear disturbance tensors.” In: *Journal of Fluid Mechanics* 731 (2013), R3. DOI: [10.1017/jfm.2013.402](https://doi.org/10.1017/jfm.2013.402).
- [91] Donald L. Ermak and J. A. McCammon. “Brownian dynamics with hydrodynamic interactions.” In: *The Journal of Chemical Physics* 69.4 (1978), pp. 1352–1360. DOI: [10.1063/1.436761](https://doi.org/10.1063/1.436761).
- [92] Zhi Liang et al. “A fast multipole method for the Rotne–Prager–Yamakawa tensor and its applications.” In: *Journal of Computational Physics* 234 (2013), pp. 133–139. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2012.09.021>.

- [93] W. Guan et al. “RPyFMM: Parallel adaptive fast multipole method for Rotne–Prager–Yamakawa tensor in biomolecular hydrodynamics simulations.” In: *Computer Physics Communications* 227 (2018), pp. 99–108. ISSN: 0010-4655. DOI: <https://doi.org/10.1016/j.cpc.2018.02.005>.
- [94] Andrew M. Fiore et al. “Rapid sampling of stochastic displacements in Brownian dynamics simulations.” In: *The Journal of Chemical Physics* 146.12 (2017), p. 124116. DOI: [10.1063/1.4978242](https://doi.org/10.1063/1.4978242).
- [95] Paweł Zuk et al. “Rotne–Prager–Yamakawa approximation for different-sized particles in application to macromolecular bead models.” In: *Journal of Fluid Mechanics* 741 (2014), R5. DOI: [10.1017/jfm.2013.668](https://doi.org/10.1017/jfm.2013.668).
- [96] NVIDIA Corporation. *The NVIDIA cuSolver CUDA library*. 2014.
- [97] NVIDIA Corporation. *The NVIDIA cuBLAS CUDA library*. 2014.
- [98] Marshall Fixman. “Construction of Langevin forces in the simulation of hydrodynamic interaction.” In: *Macromolecules* 19.4 (1986), pp. 1204–1207. DOI: [10.1021/ma00158a043](https://doi.org/10.1021/ma00158a043).
- [99] Richard M. Jendrejack, Michael D. Graham, and Juan J. de Pablo. “Hydrodynamic interactions in long chain polymers: Application of the Chebyshev polynomial approximation in stochastic simulations.” In: *The Journal of Chemical Physics* 113.7 (2000), pp. 2894–2900. DOI: [10.1063/1.1305884](https://doi.org/10.1063/1.1305884).
- [100] Tadashi Ando et al. “Krylov subspace methods for computing hydrodynamic interactions in Brownian dynamics simulations.” In: *The Journal of Chemical Physics* 137.6 (2012), p. 064106. DOI: [10.1063/1.4742347](https://doi.org/10.1063/1.4742347).
- [101] Amir Saadat and Bamin Khomami. “Computationally efficient algorithms for incorporation of hydrodynamic and excluded volume interactions in Brownian dynamics simulations: A comparative study of the Krylov subspace and Chebyshev based techniques.” In: *The Journal of Chemical Physics* 140.18 (2014), p. 184903. DOI: [10.1063/1.4873999](https://doi.org/10.1063/1.4873999).
- [102] Charles S. Peskin. “The immersed boundary method.” In: *Acta Numerica* 11 (2002), pp. 479–517. DOI: [10.1017/S0962492902000077](https://doi.org/10.1017/S0962492902000077).

- [103] Eric E. Keaveny. “Fluctuating force-coupling method for simulations of colloidal suspensions.” In: *Journal of Computational Physics* 269 (2014), pp. 61–79. ISSN: 0021-9991. DOI: [10.1016/j.jcp.2014.03.013](https://doi.org/10.1016/j.jcp.2014.03.013).
- [104] H. Hasimoto. “On the periodic fundamental solutions of the Stokes equations and their application to viscous flow past a cubic array of spheres.” In: *Journal of Fluid Mechanics* 5.2 (1959), pp. 317–328. DOI: [10.1017/S0022112059000222](https://doi.org/10.1017/S0022112059000222).
- [105] Dag Lindbo and Anna-Karin Tornberg. “Spectral accuracy in fast Ewald-based methods for particle simulations.” In: *Journal of Computational Physics* 230.24 (2011), pp. 8744–8761. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2011.08.022>.
- [106] Mu Wang and John F. Brady. “Spectral Ewald Acceleration of Stokesian Dynamics for polydisperse suspensions.” In: *Journal of Computational Physics* 306 (2016), pp. 443–477. ISSN: 0021-9991. DOI: [10.1016/j.jcp.2015.11.042](https://doi.org/10.1016/j.jcp.2015.11.042).
- [107] Dag Lindbo and Anna-Karin Tornberg. “Spectrally accurate fast summation for periodic Stokes potentials.” In: *Journal of Computational Physics* 229.23 (2010), pp. 8994–9010. ISSN: 0021-9991. DOI: [10.1016/j.jcp.2010.08.026](https://doi.org/10.1016/j.jcp.2010.08.026).
- [108] Joel H Ferziger, Milovan Perić, and Robert L Street. *Computational methods for fluid dynamics*. Vol. 3. Springer, 2002.
- [109] H.F. Meier, J.J.N. Alves, and M. Mori. “Comparison between staggered and collocated grids in the finite-volume method performance for single and multi-phase flows.” In: *Computers & Chemical Engineering* 23.3 (1999), pp. 247–262. ISSN: 0098-1354. DOI: [https://doi.org/10.1016/S0098-1354\(98\)00270-1](https://doi.org/10.1016/S0098-1354(98)00270-1).
- [110] Florencio Balboa Usabiaga. “Minimal models for finite particles in fluctuating hydrodynamics.” PhD thesis. Universidad Autonoma de Madrid, 2014.
- [111] Florencio Balboa Usabiaga et al. “Staggered Schemes for Fluctuating Hydrodynamics.” In: *Multiscale Modeling & Simulation* 10.4 (2012), pp. 1369–1408. DOI: [10.1137/120864520](https://doi.org/10.1137/120864520).

- [112] Charles S Peskin. “Numerical analysis of blood flow in the heart.” In: *Journal of Computational Physics* 25.3 (1977), pp. 220–252. ISSN: 0021-9991. DOI: [https://doi.org/10.1016/0021-9991\(77\)90100-0](https://doi.org/10.1016/0021-9991(77)90100-0).
- [113] Yuanxun Bao, Jason Kaye, and Charles S. Peskin. “A Gaussian-like immersed-boundary kernel with three continuous derivatives and improved translational invariance.” In: *Journal of Computational Physics* 316 (2016), pp. 139–144. ISSN: 0021-9991. DOI: <10.1016/j.jcp.2016.04.024>.
- [114] Alexander H. Barnett, Jeremy Magland, and Ludvig af Klinteberg. “A Parallel Nonuniform Fast Fourier Transform Library Based on an “Exponential of Semicircle” Kernel.” In: *SIAM Journal on Scientific Computing* 41.5 (2019), pp. C479–C504. DOI: <10.1137/18M120885X>.
- [115] D. S. Shamshirgar, J. Bagge, and A.-K. Tornberg. “Fast Ewald summation for electrostatic potentials with arbitrary periodicity.” In: *The Journal of Chemical Physics* 154.16 (2021), p. 164109. DOI: <10.1063/5.0044895>.
- [116] Xiaolei Yang et al. “A smoothing technique for discrete delta functions with application to immersed boundary method in moving boundary simulations.” In: *Journal of Computational Physics* 228.20 (2009), pp. 7821–7836. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2009.07.023>.
- [117] Xiangyu Guo et al. “Efficient Particle-mesh Spreading on GPUs.” In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 120–129. ISSN: 1877-0509. DOI: <10.1016/j.procs.2015.05.210>.
- [118] Yu hsuan Shih et al. “cuFINUFFT: a load-balanced GPU library for general-purpose nonuniform FFTs.” In: (2021).
- [119] Martin Vögele and Gerhard Hummer. “Divergent Diffusion Coefficients in Simulations of Fluids and Lipid Membranes.” In: *The Journal of Physical Chemistry B* 120.33 (2016). PMID: 27385207, pp. 8722–8732. DOI: <10.1021/acs.jpcb.6b05102>.

- [120] Frank L. H. Brown. “Continuum simulations of biomembrane dynamics and the importance of hydrodynamic effects.” In: *Quarterly Reviews of Biophysics* 44.04 (2011), pp. 391–432.
- [121] Alvaro Domínguez, Martin Oettel, and S. Dietrich. “Dynamics of colloidal particles with capillary interactions.” In: *Phys. Rev. E* 82 (1 2010), p. 011402. DOI: [10.1103/PhysRevE.82.011402](https://doi.org/10.1103/PhysRevE.82.011402).
- [122] Peng Gao et al. “Influence of an Additive-Free Particle Spreading Method on Interactions between Charged Colloidal Particles at an Oil/Water Interface.” In: *Langmuir* 32.19 (2016), pp. 4909–4916. DOI: [10.1021/acs.langmuir.6b01362](https://doi.org/10.1021/acs.langmuir.6b01362).
- [123] Binhua Lin et al. “Divergence of the long-wavelength collective diffusion coefficient in quasi-one-and quasi-two-dimensional colloidal suspensions.” In: *Physical Review E* 89.2 (2014), p. 022303.
- [124] Mikkel Settnes and Henrik Bruus. “Forces acting on a small particle in an acoustical field in a viscous fluid.” In: *Physical Review E* 85.1 (2012). DOI: [10.1103/physreve.85.016327](https://doi.org/10.1103/physreve.85.016327).
- [125] Xiaoyun Ding et al. “Tunable patterning of microparticles and cells using standing surface acoustic waves.” In: *Lab on a Chip* 12.14 (2012), p. 2491. DOI: [10.1039/c2lc21021e](https://doi.org/10.1039/c2lc21021e).
- [126] S. Panzuela, Raúl P. Peláez, and R. Delgado-Buscalioni. “Collective colloid diffusion under soft two-dimensional confinement.” In: *Phys. Rev. E* 95 (1 2017), p. 012602. DOI: [10.1103/PhysRevE.95.012602](https://doi.org/10.1103/PhysRevE.95.012602).
- [127] J. Bleibel, Alvaro Domínguez, and M. Oettel. “Onset of anomalous diffusion in colloids confined to quasimonolayers.” In: *Phys. Rev. E* 95 (3 2017), p. 032604.
- [128] Marc Meléndez Schofield and Rafael Delgado-Buscalioni. “Quantitative description of the response of finite size adsorbates on a quartz crystal microbalance in liquids using analytical hydrodynamics.” In: *Soft Matter* 17 (35 2021), pp. 8160–8174. DOI: [10.1039/D1SM00492A](https://doi.org/10.1039/D1SM00492A).
- [129] C. W. Clenshaw and A. R. Curtis. “A method for numerical integration on an automatic computer.” In: *Numerische Mathematik* 2.1 (1960), pp. 197–205. ISSN: 0945-3245. DOI: [10.1007/BF01386223](https://doi.org/10.1007/BF01386223).

- [130] Anna-Karin Tornberg. “The Ewald sums for singly, doubly and triply periodic electrostatic systems.” In: *Advances in Computational Mathematics* 42 (2015), pp. 227,248. DOI: [10.1007/s10444-015-9422-3](https://doi.org/10.1007/s10444-015-9422-3).
- [131] Ondrej Maxian et al. “A fast spectral method for electrostatics in doubly periodic slit channels.” In: *The Journal of Chemical Physics* 154.20 (2021), p. 204107. DOI: [10.1063/5.0044677](https://doi.org/10.1063/5.0044677).
- [132] D.J. Griffiths. *Introduction to Electrodynamics*. Pearson Education, 2014. ISBN: 9780321972101.
- [133] Ondrej Maxian et al. “Simulations of dynamically cross-linked actin networks: Morphology, rheology, and hydrodynamic interactions.” In: *PLOS Computational Biology* 17.12 (2021), pp. 1–38. DOI: [10.1371/journal.pcbi.1009240](https://doi.org/10.1371/journal.pcbi.1009240).
- [134] Paloma Rodríguez-Sevilla et al. “Upconverting Nanorockers for Intracellular Viscosity Measurements During Chemotherapy.” In: *Advanced Biosystems* 3.10 (2019), p. 1900082. DOI: <https://doi.org/10.1002/adbi.201900082>.
- [135] Paloma Rodríguez-Sevilla et al. “Optical Torques on Up-converting Particles for Intracellular Microrheometry.” In: *Nano Letters* 16.12 (2016), pp. 8005–8014. ISSN: 1530-6984. DOI: [10.1021/acs.nanolett.6b04583](https://doi.org/10.1021/acs.nanolett.6b04583).
- [136] Michael J. Solomon and Patrick T. Spicer. “Microstructural regimes of colloidal rod suspensions, gels, and glasses.” In: *Soft Matter* 6 (7 2010), pp. 1391–1400. DOI: [10.1039/B918281K](https://doi.org/10.1039/B918281K).
- [137] Nerea Alcázar-Cano and Rafael Delgado-Buscalioni. “A general phenomenological relation for the subdiffusive exponent of anomalous diffusion in disordered media.” In: *Soft Matter* 14 (48 2018), pp. 9937–9949. DOI: [10.1039/C8SM01961D](https://doi.org/10.1039/C8SM01961D).
- [138] Raúl P. Peláez and Rafael Delgado-Buscalioni. “Origin of Tank-Treading and Breathing Dynamics of Star Polymers in Shear Flow.” In: *Macromolecules* 53.7 (2020), pp. 2634–2648. DOI: [10.1021/acs.macromol.9b01968](https://doi.org/10.1021/acs.macromol.9b01968).
- [139] R. Delgado-Buscalioni et al. “Emergence of collective dynamics of gold nanoparticles in an optical vortex lattice.” In: *Phys. Rev. E* 98 (6 2018), p. 062614. DOI: [10.1103/PhysRevE.98.062614](https://doi.org/10.1103/PhysRevE.98.062614).

- [140] Bingtian Ye et al. “Emergent Hydrodynamics in Nonequilibrium Quantum Systems.” In: *Phys. Rev. Lett.* 125 (3 2020), p. 030601. DOI: [10.1103/PhysRevLett.125.030601](https://doi.org/10.1103/PhysRevLett.125.030601).
- [141] NVIDIA Corporation. *The NVIDIA cuFFT CUDA library*. 2014.
- [142] L. Greengard. “Spectral Integration and Two-Point Boundary Value Problems.” In: *SIAM Journal on Numerical Analysis* 28.4 (1991), pp. 1071–1080. DOI: [10.1137/0728057](https://doi.org/10.1137/0728057).
- [143] Lloyd N. Trefethen. *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics, 2000. DOI: [10.1137/1.9780898719598](https://doi.org/10.1137/1.9780898719598).
- [144] A. A. Karawia. “Two algorithms for solving general backward pentadiagonal linear systems.” In: *International Journal of Computer Mathematics* 87.12 (2010), pp. 2823–2830. DOI: [10.1080/00207160802326507](https://doi.org/10.1080/00207160802326507).
- [145] Marc Meléndez. *A Painless Introduction to Programming UAMMD Modules*. 2020.