Rajesh Ramaswamy & Ivo F. Sbalzarini

Particle Methods

Computing with Particles

Lecture Notes

TU Dresden, Faculty of Computer Science
Chair of Scientific Computing for Systems Biology

Prof. Dr. Ivo F. Sbalzarini, TUD & MPI-CBG
Dr. Rajesh Ramaswamy, MPI-PKS

June 2021

This page is intentionally left blank

Contents

List of Figures

Listings

Foreword

Chapter 1

Introduction

In this chapter:

- What is modeling and simulation?
- The four kingdoms of models
- Item-based simulations vs. field-based simulations
- What are particle methods?
- Computational cost of particle methods

Learning goals:

- Know the difference between a model and a simulation
- Know the four kingdoms of models and examples of each
- Know the difference between item-based and field-based simulations, and how they relate to the underlying models
- Be able to explain the concept of discretization
- Know by heart the definition of particle methods and their basic structure
- Be able to analyze the computational cost of a particle method
- Be able to name two examples of near- and far-field interactions and motivate the need to hybrid particle-mesh methods

1.1 What is modeling and simulation?

Models are hypotheses about a process or a system. Any mental image about how we think or believe a system looks or works is a model of that system. Using the model to answer a specific question about the system is called a

simulation. Simulations are hence more akin to experiments than to theory. A simulation will tell how the model behaves in a specific situation (e.g., for specific values of its parameters) or reacts to a specific, given perturbation. Theory, however, would predict how it behaves for *any* parameter for *every possible* perturbation. Simulations are especially useful or necessary in cases that evade theoretical treatment or where the corresponding perturbation could not be applied experimentally to the modeled system. This is often the case for non-linear models or when a parameter is not controllable in the system (e.g., we cannot change the diffusion constant of a molecule at will since it is a physical given) or the system reaction is not observable.

Models need not be mathematical, and simulations need not be computational. Using, e.g., a cardboard *model* of the planned building to predict how light penetrates into the rooms, the architect performs a simulation. Using a *model* organism like mouse to learn something about cancer in humans, the biologist performs a simulation.

Every model is valid only in a limited and well-defined set of cases, called the model's *experimental frame*. Using a model outside its experimental frame produces incorrect predictions or results. While the bespoke cardboard model of a house can be used to simulate light and shadow, it could not be used to simulate earthquake or fire safety. While mice can be used to learn something about cancer, they could not simulate how newborns learn language.

Models can be qualitative or quantitative, physical or phenomenological, discrete or continuous, stochastic or deterministic. Qualitative models do not predict actual values (of, e.g., chemical concentrations), but only qualitative outcomes like whether something is increasing or decreasing (without telling by how much). Physical models are formulated or derived from physical laws and principles, such as conservation of mass or energy. This is in contrast to phenomenological models, which describe mechanisms that do not necessarily adhere to physics. In discrete models, individual real-world entities are directly represented as discrete items in the model, like the colorful plastic balls in high-school molecule models represent individual atoms. In a continuous model, the entities are not explicitly represented, but only their continuous distribution field. For example, describing the population distribution over Germany as a population density field, rather than by representing every individual separately, is a continuous model. The reaction or evolution of a deterministic model is completely determined by the present and previous state of the model, whereas in stochastic models, the output contains a random element. This means that one cannot predict the reaction, but only a probability distribution of it. These different classes of models are summarized in Fig. 1.1, along with typical examples of each class.

If the model is given by mathematical equations, e.g., partial differential equations, a simulation computes a correct numerical solution of these equations. This amounts to *in-silico* reconstitution of the system described by the model. Simulations are hence not "made to fit expectations", but provide a powerful tool to predict system behavior. A simulation can, e.g., be used to check whether the mechanisms included in the model are *sufficient* to produce a certain be-

	continuous	discrete
deterministic	PDEs	interacting particles
deterministic	diffusion	molecular dynamics
stochastic	SDEs	random events
stochastic	reaction-diffusion with low molecule numbers	population dynamics

Figure 1.1: The four types of models with typical examples.

havior. This would be hard to do experimentally, because one could never be sure that the mechanisms in question are really the only ones in the system. In a model, however, one has full control over what is included and what not. The fact that not everything is included in the model is hence the *strength* of the model, and not a weakness. This fact is famously known as "Ockham's razor" [1] stating that the simplest possible models are the most useful ones, since they tell that this simplicity is a sufficient explanation of reality. Whether it is also necessary then remains to be proven experimentally by perturbing these mechanisms in the system and checking if then it stops working.

Discrete models (space and/or time) can directly be simulated in a (digital) computer, whereas continuous ones first need to be discretized since computers have finite memory and can only deal with finite sets. Discretization means that we do not store and compute the value of the field everywhere (because that would make infinitely many points), but only at selected, representative *discretization points*. This is very much like using a finite set of weather stations scattered across the country to monitor the continuous temperature field. Obviously, the weather stations need to be "dense enough" to capture temperature variations. Having them too dense, however, makes no sense (the recorded differences would mostly be noise and measurement errors) and would be wasteful. In discretization theory, "dense enough" means *well sampled*, and the condition is given by the length scale of the field variations. This is the length scale over which the field changes, i.e., over which gradients in the field exist. In particle methods, the discretization points can be distributed arbitrarily, which may help track field variations. In other methods, their spatial arrangement needs to satisfy certain conditions. In finite-difference methods, e.g., the points need to lie on a grid.

1.2 What are particle methods?

Particle methods are numerical schemes that can be used to computationally simulate both discrete and continuous models, either deterministically or stochastically. The simulations are formulated in terms of interacting particles, possibly combined with meshes. When simulating a discrete model, we call the simulation *item-based simulation*. When simulating a discretized continuous model, we call the simulation *field-based simulation*. We use these terms in order to make clear whether the underlying model was discrete or continuous, whereas any simulation is always discrete. For item-based simulations, the case is clear: each item is represented by a particle. For field-based simulations, we need mathematical tools to discretize fields using particles.
In all cases, particles are point-like objects (i.e., zero-dimensional) that are characterized by a *position* and certain *properties*. The position can be in any space, and the property list can be arbitrarily long and contain different data types. For example, a car in the street could be described as a particle whose position corresponds to the GPS coordinates on the map and properties could be the velocity, age of the driver, color of the car, etc. While particles always must have a position, the property list can also be empty. A particle hence is a zero-dimensional data-structure with a position and certain properties. The position is a vector of any data type, depending on the space in which the particles live, and its length is given by the dimension of that space. For example, if the particle is to mark a pixel in an image, then the position is a vector of integers. Thus, the space is discrete. If the particle represents a molecule, the position is a vector of real numbers and the space is continuous.
Particles can do two things: *interact* with other particles and *evolve*. Evolving means that the particle's position and/or properties change. Interaction is a pair-wise interaction of the particle with another particle and yields a contribution to the change of properties and positions as a result. These are the two operations defined for each particle. In pseudo-code[1], a particle hence is:

```
class PARTICLE {
    vector(space-dimension) :: position,positionChange
    struct :: properties,propertiesChange

    method [Kx,Kw] = interact(PARTICLE)
    method evolve()
}
```

Listing 1.1: Particle data structure

An interaction between two particles can at most depend on the positions and properties of the two particles. In particle methods, we assume that the interactions are pair-wise and additive, i.e., that the result of interaction(p,q,r) is identical to interaction(p,q)+interaction(p,r). This assumption is limiting, but part of the definition of particle methods. The dynamic equations governing

[1] We use a loose pseudo-code notation in these lecture notes, which is a mixture of C++ and Fortran, and should be intuitively understandable to the programming-savvy.

any particle method can hence be written as:

$$\begin{bmatrix}\Delta\vec{x}_p \\ \Delta\vec{\omega}_p\end{bmatrix} = \sum_{q=1}^{N} \begin{bmatrix}\vec{K}_x \\ \vec{K}_\omega\end{bmatrix} = \sum_{q=1}^{N} \vec{K}_p(\vec{x}_p, \vec{x}_q, \vec{\omega}_p, \vec{\omega}_q) \tag{1.1}$$

Here, the *interaction kernel* $\vec{K}_p$ encapsulates the model and is the mathematical representation of the `interact` method. It is indexed by p, as different particles can have different interaction kernels. In most cases, however, the same kernel is used across particles, which is why we often omit the index p for clarity. Everything else is generic. This is very nice from an algorithmic point of view, since the same algorithms and software can be used to simulate any particle method. All that algorithm need to be able to do is to compute interactions between pairs of particles and sum the contributions over all particles. This provides the rates of changes or increments (per unit time) in the particle position, $\Delta\vec{x}$ (`positionChange`) and properties $\Delta\vec{\omega}$ (`propertiesChange`). In a concrete software, $\vec{K}$ can for example be a function pointer to an implementation of the `interact` method.

1.2.1 Examples

For illustration, we provide here a few examples of particle methods along with their classification:

- Agent-based simulations are particle methods for item-based models where each particle represents an item or agent. The interaction kernel $\vec{K}$ represents the function or method according to which two agents interact.
- Molecular-dynamics simulations [2] are a particle method for an item-based model where particles represent atoms (or groups of atoms in coarse-grained molecular dynamics). The interaction kernel in this case is the atomic or molecular *force field* or potential.
- Smooth particle hydrodynamics (SPH) [3] is a particle method to simulate the field-based model of continuum fluid flow. The interaction kernel in this case results from discretizing the differential operators in the governing physical equations.
- Evolution strategies are particle methods to solve optimization problems in an item-based model, where each particle represents a sample point and the interaction kernel $\vec{K}$ contains the *mutation* and *evaluation* operators.
- Region competition [4] is a particle methods to solve image-segmentation problems over field-based models where particles mark boundary pixels between image regions and the kernels $\vec{K}_x$ and $\vec{K}_\omega$ are the propagator and imaging model, respectively.
- The discrete element method (DEM) is a particle method to simulate the item-based model for granular flows, where the kernel K is Newton's law of mechanics and the contact deformation model [5] (see Section 4.2.4) .

1.2.2 Generic algorithm

Due to the generic formulation in Eq. 1.1, all particle methods can be encapsulated in a generic algorithm, which amounts to implementing the `interact` and `evolve` methods. The generic algorithm then is:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
    foreach particle q do
        [p.positionChange,p.propertiesChange] += p.interact(q)
    end
end
foreach particle p do
    p.evolve()
end
```

Listing 1.2: Generic particle algorithm

Note that the particles first all interact, in order for each particle to compute its final Δx and $\Delta\omega$. Only then, they all evolve using these rates of change. Evolving particles directly in the first loop would lead to non-deterministic behavior, as the result will depend on the ordering (indexing) of the particles. This algorithm has a runtime complexity in $O(N^2)$ for N particles. This can be significantly reduced if the interaction kernel is local, i.e., decays to zero for particle further apart than a certain *cutoff radius*. In theory, if every particle only interacts with an $O(1)$ set of "neighbors", the overall algorithm reduces to $O(N)$. This is frequently the case. Examples include discrete element methods (see Section 4.2.4), where particles can only collide with nearby partners, and field-based simulations of diffusion, which is a local physical phenomenon. Fast neighbor search algorithms are available for each particle to find its nearby interaction partners in $O(1)$ time (see Chapter 2). This is not trivial, since the particles move all the time and the set of neighbors constantly changes in a potentially unpredictable way (e.g., in stochastic simulations).
In the case there the interaction kernel is not local, all particles contribute to all other particles. This is called a *long-range interaction*. The simulation is then truly $O(N^2)$ per time step. However, efficient approximation algorithms exist also for this case. The seminal Barnes-Hut algorithm can be used to reduce the complexity to $O(N \log N)$ in the average case of uniformly distributed particles [6]. The idea behind this algorithm is that a group of particle far away can be approximated as a single "cluster particle" and one interaction is then sufficient to consider the whole group. This is like looking at a galaxy in the sky at night. Since the galaxy is far away, you do not see all the individual stars in it, but you perceive the whole galaxy as one "particle". For uniformly distributed particles, the cluster tree has a depth of $O(\log N)$, which explains the runtime bound. The bound can be further reduced to $O(N)$ using *Fast Multipole Methods* (FMM) [7, 8], where the action of each cluster of particles onto the other particles is further expanded into a series of spherical harmonics. This allows clusters to directly interact with other clusters, hence the runtime bound. The concept is akin to how international politics work. If the United

States want to negotiate a treaty with the European Union, it is not that every citizen of the US talks to every citizen of the EU ($O(N^2)$, where N is about 450 million). It is also not that every citizen of the US talks to the EU government (i.e., the Barnes-Hut way where the EU government is the cluster representing all people in the EU). Instead, the two governments negotiate (this is $O(1)$) and then, the results are propagated "down" to the individual states, cities, and people within each cluster (which is $O(N)$). While these algorithms are efficient, they are approximate, with the approximation error depending non-trivially on the cluster granularity and the number of expansion coefficients used.

1.2.3 Hybrid particle-mesh methods

An alternative way of evaluating long-range particle interactions is by using a mesh. This defines *hybrid particle-mesh methods*, where particles are combined with an overlaid grid. The grid is usually Cartesian and uniform (i.e., all grid cells are rectangles or boxes of same size). This is because the focus is on computational efficiency, and potential non-uniform or sub-grid-scale phenomena can still be represented directly on the particles. The grid is only thought to provide the "background far field".

The famous historic example for which this framework has originally been developed are plasma physics simulations [9]. In these simulations, individual charge carriers (e.g., electrons) are represented as particles. This is the item-based part of the model. Collectively, all charge carriers induce an electric field, which depends on the spatial distribution of charge carriers. This continuous electric field then defines the force each particle feels, and hence the motion acceleration of it. This is the field-based part of the model.

In a pure particle method, each charge carrier is represented by a particle with position $\vec{x}_p$ and scalar charge ω_p. The charges never change, hence $K_\omega = 0$. The positions of the particles evolve according to the mechanical forces they feel in the induced electric field. These forces are given by the Coulomb law of electrostatics:

$$\vec{F}_p = \sum_{q=1}^{N} \frac{\omega_p \omega_q}{4\pi\varepsilon} \frac{\vec{r}_{pq}}{|\vec{r}_{pq}|^2}, \tag{1.2}$$

where $\vec{r}_{pq}$ is the vector pointing from particle p to particle q, and ε is a physical constant called the *permittivity* of the material. Evaluating the force for all particles requires $O(N^2)$ operations.

In the hybrid particle-mesh formulation, the charges of the particle are interpolated to a regular Cartesian grid. Dividing then the charge of each grid node with the volume of a grid cell (h^d for a grid resolution h in d-dimensional space), one obtains the charge density ρ at each mesh node. The induced electric field can then be computed by solving the partial differential equation

$$\nabla \cdot \vec{E}(\vec{x}) = \frac{\rho(\vec{x})}{\varepsilon} \tag{1.3}$$

for the electric field $\vec{E}$, where $\nabla\cdot$ is the divergence operator from vector calculus. This field is then interpolated back to the particles in order to obtain the force acting on each particle as:

$$\vec{F}_p = \omega_p \vec{E}(\vec{x}_p) \tag{1.4}$$

The last step is $O(1)$. Particle-mesh and mesh-particle interpolation are $O(N)$ each, as discussed in detail in Section 7.4. Solving Eq. 1.3 on a regular Cartesian mesh can be done in $O(M)$ using finite differences (multi-grid methods [10]) or in $O(M \log M)$ using fast Fourier transforms, where M is the total number of mesh nodes. The overall hybrid particle-mesh method is hence $O(N)$ when using multi-grid methods, and $O(N \log N)$ when using fast Fourier transforms. The solution computed using multi-grid methods is approximate with an approximation error that converges with some power of h. The solution computed using Fourier transforms is accurate to machine precision on periodic domains. On unbounded domains it also converges with some power of h [11].
More on efficient long-rage interactions and field solvers in Chapter 8.

1.2.4 Discretizing differential operators on particles

Alternatively to discretizing differential operators and partial differential equations on a mesh, they can also be directly discretized on the particles. Thus, pure particle methods can also be used to solve differential equations. In the most general form, this amounts to an operator involving the properties and positions of all involved particles as a pairwise interaction:

$$D^\beta \omega(\vec{x}_p) \approx \sum_{q \in \mathcal{N}(p)} (\vec{\omega}_q \pm \alpha \vec{\omega}_p) \eta_\epsilon^\beta (\vec{x}_q - \vec{x}_p). \tag{1.5}$$

Here, $\mathcal{N}(p)$ is the neighborhood of particle p, i.e., all other particles that are interaction partners for this operator. The binary number α is 0 for asymmetric operators and 1 for symmetric ones. The sign in the first parenthesis depends on the differential operator D^β to be discretized. The *operator kernel* η^β is scaled to width ϵ as: $\eta_\epsilon^\beta = \epsilon^{-d} \eta^\beta(\vec{x}/\epsilon)$, defining the spatial resolution of the discretization. The specific form of the kernel η^β depends on the differential operator D^β that is to be discretized, and on its order β. More on this topic in Chapter 5.
The operator symmetry deserves special attention. If $\alpha = 1$, then the above expression is symmetric in the sense that the contribution of particle q to particle p is the same (maybe with sign change, depending on $\pm$) as the contribution of p to q. This can be exploited to reduce the computational cost of the method by another factor of two by considering every unique interaction *pair* only once. Symmetric neighbor-search algorithms are available for this, as described in Chapter 2. If ω corresponds to a conserved physical quantity, like mass, then this also guarantees that the method is conservative, i.e., that the mass received by q is exactly the mass sent by p and nothing is lost or created underway.

Chapter 2

Algorithms and Data Structures for Particle Methods

In this chapter:

- Efficient short-range interactions with cell lists?
- Verlet lists
- Exploiting symmetry in interactions

Learning goals:

- Be able to implement and use cell lists and Verlet lists for symmetric and asymmetric short-range interactions
- Know the computational complexity of building and using cell lists and verlet lists
- Be able to quantitatively compare cell and Verlet lists in terms of the cost pre-factor

Evolving particles, in general, requires that particles interact. In order to update the position and properties of particles, each particle needs to interact with other particles in the system. If the number of particles is N, the number of computations to perform to compute the rate of change of position $\Delta\vec{x}_p$ and the rate of change of the properties $\Delta\vec{\omega}_p$ is $O(N^2)$. This can, however, be improved if the computation of $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ only requires a particle to interact with other particles within a local neighborhood. For example, assume that you want to write an algorithm for a video game that requires you to evolve the position of a car traveling with a given speed on a street with other cars. In order to determine the required change of position of the car, one needs to find

other cars in the neighborhood. Knowing the positions of *all* other cars is not necessary. Assuming that you want to control all N_c cars, an efficient algorithm to determine the subsequent position of all cars is $O(N_c)$ and not $O(N_c^2)$. In such cases, we say that computation of $\Delta\vec{x}_p$ for moving particles (and, in general computation of $\Delta\vec{\omega}_p$ for evolving properties) depend on *short-range interactions* where particles only need to interact with other particles within a small (small compared to the size of the system) fixed, local neighborhood defined by a cut-off radius or distance r_c. In such cases where each particle only needs to interact with other particles with a distance of r_c from itself, the number of computational operations for computing $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ can be reduced to $O(N)$ by using *fast neighbor lists*. Fast neighbor lists are data structures providing a list of other particles that a given particle needs to interact with. There are two principal types of fast neighbor lists: *cell list* and *Verlet list*.

2.1 Cell lists

Cell lists are fast neighbour lists in which the computational domain is partitioned into Cartesian cells of length r_c in each dimension. The cell-list data structure stores particle indices that reside in each of these cells. For the sake of simplicity, we introduce cell lists for two-dimensional computational domains and the extension to three-dimensions is straightforward.
Assume a two-dimensional space of length L_x in the x-direction and length L_y in the y-direction. Assume that we have N particles residing in this two-dimensional space. We divide the two-dimensional space into squares of size r_c. The number of squares (or cells) of size r_c required to tile the two-dimensional space in the x-direction is $\left\lceil \frac{L_x}{r_c} \right\rceil$ and in the y-direction is $\left\lceil \frac{L_y}{r_c} \right\rceil$. The total number of such squares is therefore $\left\lceil \frac{L_x}{r_c} \right\rceil \left\lceil \frac{L_y}{r_c} \right\rceil$ and each square is indexed by two indices (i, j) in two-dimensions where $i = 0, \ldots, \left\lceil \frac{L_x}{r_c} \right\rceil - 1$ and $j = 0, \ldots, \left\lceil \frac{L_y}{r_c} \right\rceil - 1$.
Cell list is a data structure that gives the index p of particles and the number of particles in each cell i, j. A simple recipe to build cell list in two-dimensions is as follows:
For each particle $p = 0, \ldots, N-1$, with positions $\vec{x}_p = (x_p, y_p)$ (in 2D):

1. Compute the cell index (i, j) as $i = \left\lfloor \frac{x_p}{r_c} \right\rfloor$ and $j = \left\lfloor \frac{y_p}{r_c} \right\rfloor$.
2. Add particle p to the cell (i, j).

The algorithm for constructing cell list in two-dimensions is as follows:

```
foreach particle p do
    cellIndex = floor(p.position / rc)
    cell(cellIndex).add(p)
end
```

Listing 2.1: Cell list construction

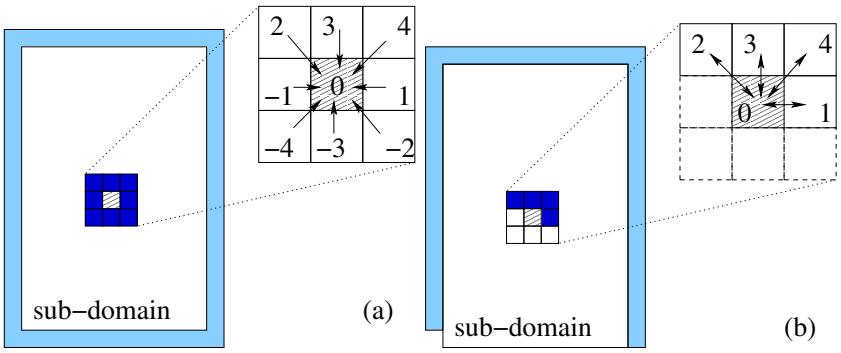


Figure 2.1: Cell list interactions in 2D. (a) In the asymmetric case, each particle interacts with all other particles in the same cell and with all particles in all adjacent cells. As the cell stencil iterates over the entire computational sub-domain, boundary layers (boundary conditions) are needed on all sides. (b) In the symmetric case, particles interact with particle in the same cell that have a bigger index, and with *all* particles in *half* of the neighboring cells, as indicated. The change is then simultaneously applied to both interaction partners. In this case, boundary layers are needed on all but one face of the sub-domain.

2.1.1 Short-range interactions with cell list

Once the cell list has been constructed, every particle p has to interact with all other particles in its cell and all adjacent cells. The number of adjacent cell is 8 in two dimensions and 26 in three dimensions. Figure 2.1(a) shows the adjacent cells of a cell (i, j) in 2D.
The generic particle algorithm presented in Chapter 1 can now be made more efficient for computing $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ in case of short-range interactions by incorporating cell lists. The algorithm for such a particle algorithm is as follows:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
    cellIndex = floor(p.position / rc)
    foreach particle q in cell(cellIndex) do
        [p.positionChange,p.propertiesChange] += p.interact(q)
    end
    foreach adjacentCellIndex do
        foreach particle q in cell(adjacentCellIndex) do
            [p.positionChange,p.propertiesChange] += p.interact(q)
        end
    end
end
foreach particle p do
    p.evolve()
end
```

Listing 2.2: Short-range interaction particle algorithm with cell list

2.1.2 Computational cost

Cost of constructing cell list. The computational cost of constructing is solely dependent on the number of particle N within the computational domain. In addition, the cost is linear in N and is therefore $O(N)$.

Cost of particle algorithm with short-range interactions using cell list. Assume a three-dimensional computational domain of size L in each direction so that the volume is L^3. Given N particles in the computational domain that are distributed homogeneously (or uniformly at random) across the domain, the average number of particles per unit volume is $\frac{N}{L^3}$. Given a cut-off radius r_c for particle interactions, the number of particles per cell is $N(\frac{r_\mathrm{c}}{L})^3$. The number of particles that each particle p needs to interact with is therefore $(N_\mathrm{a}+1)N\left(\frac{r_\mathrm{c}}{L}\right)^3$ where $N_\mathrm{a} = 26$ is the number of adjacent cells in 3D to the cell in which particle p resides. Doing this not only for one particle, but for all N particles, the total number of kernel interaction evaluations to compute $27N^2\left(\frac{r_\mathrm{c}}{L}\right)^3$ and is therefore $O(N^2)$.
In 3D (L^3), the computational cost of any particle algorithm with short range interactions is therefore proportional to $27N^2\left(\frac{r_\mathrm{c}}{L}\right)^3$ which is $C \cdot O(N^2)$ where the pre-factor of the computational cost $C = 27\left(\frac{r_\mathrm{c}}{L}\right)^3$. However, since $r_\mathrm{c} << L$ by definition of short-range interactions, the pre-factor C is very small and therefore a particle algorithm with cell list is fast. In practical cases, r_c is about 1000-times smaller than L, rendering the cell-list algorithm 10^9 times faster than the direct all-against-all interaction. In addition, if the number of particle N is increased while keeping the average number of particles per cell $\frac{Nr_c^3}{L^3}$ constant, the computational cost can be written as $27\frac{N}{L^3}r_\mathrm{c}^3\,N$ which is $C \cdot O(N)$ where $C = 27\frac{N}{L^3}r_\mathrm{c}^3$ is a constant pre-factor owing to constant average number of particles per cell. This is usually the case in field-based simulations, where the cutoff radius $r_\mathrm{c} \propto h$ with h being the distance between particles. This means that having neighbors more nearby implies reducing the interaction cutoff in order to keep the *number* of interaction partners constant. In this case, the average complexity of cell lists is $O(N)$.
It is also worth noting that the computational cost increases if the particle are inhomogeneously distributed within the computational domain. In the worst case, if all particle are within a distance of r_c from each other, then each particle needs to interact with all other particles rendering the computational cost $O(N^2)$ again.

2.2 Verlet list

The Verlet list [12] is a fast neighbor list, where interaction partners of each particle are explicitly stored. In other words, the particle lists are not associated with cells, but directly with the particles themselves. A Verlet list hence is a data structure that stores the indices q of all particles that interact with each

particle p. Efficient construction of Verlet lists utilizes an intermediate cell-list data structure in order to not search over all particles.
Verlet lists are smaller than cell lists since they only contain exactly those particles p needs to interact with. Cell lists are conservative and contain more particles than necessary. The difference is about a factor of 6 in 3D, as discussed below. However, particles constantly move, which also changes the neighbor set of each particle at every time step. Verlet lists would hence have to be updated or recomputed at each time step, which, since they are using an intermediate cell list, renders them less efficient than cell lists. The common remedy to this is to extend the Verlet interaction sphere by a "safety margin" called the *skin*. Then, all particles closer than $r_c + skin$ are added to the Verlet list. This gives up a part of the factor of 6 by including more particles than actually required. However, the lists now only need to be updated once any particles has moved further than the skin thickness. A conservative estimate is to update whenever $2\vec{v}_{max}\delta t > skin$, where $\vec{v}_{max} = \max_p \vec{v}_p$ is the largest velocity occurring in the system. The factor of 2 accounts for the worst case that two particles of maximum velocity are approaching each other head-on.
Verlet lists are particularly beneficial in simulations where the particles do not perform large net movements, but rather jiggle or oscillate around a fixed location. In this case, like molecular dynamics, the Verlet lists almost never need to be updated. The skin thickness used in practice is 10% to 20% of r_c. In some applications, like molecular liquid dynamics, the optimal skin thickness can be predicted from the parameters of the system [13]. This, however, is not possible in general and is up to manual tuning.
The algorithm for constructing Verlet list is as follows:

```
foreach particle p do
    cellIndex = floor(p.position / (rc+skin))
    foreach particle q in cell(cellIndex) do
        ! compute the distance between particles p and q
        if |p.position-q.position| <= (rc+skin)
            verlet(p).add(q)
        end
    end
    foreach particle q in cell(adjacentCellIndex) do
        if |p.position-q.position| <= (rc+skin)
            verlet(p).add(q)
        end
    end
end
```

Listing 2.3: Verlet list construction

In principle, the two inner loops could be fused if a cell is also considered a neighbor of itself. The code would be equivalent. Why we prefer writing the loops separately is for reasons of symmetry, as will be discussed below.

2.2.1 Short-range interactions with Verlet list

Once the Verlet list is constructed, an efficient particle algorithm for short-range interactions is:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
    foreach particle q in verlet(p) do
        [p.positionChange,p.propertiesChange] += p.interact(q)
    end
end
foreach particle p do
    p.evolve()
end
```

Listing 2.4: Short-range interaction particle algorithm with Verlet list

2.2.2 Computational cost

Cost of constructing Verlet list. If the particle are distributed homogeneously in a 3D computational domain of volume L^3, the cost of constructing Verlet list is proportional to $27N^2\left(\frac{r_c}{L}\right)^3$ and is $O(N^2)$ following similar arguments used in Sec. 2.1.2. If the average number of particles per unit volume is kept constant even when N increases, the computational cost is $O(N)$.
The computational cost of constructing Verlet list increases with inhomogeneity in particle distribution over the computational domain. In the worst case, if all particles are within a distance of r_c, the computational cost of constructing Verlet list is $O(N^2)$.

Cost of particle algorithm with short-range interactions using Verlet list. Assume that N particles are homogeneously (or uniformly random) distributed across a 3D computational domain of volume L^3. The average number of particles within a distance of r_c from each particle then is $\frac{N}{L^3}\frac{4}{3}\pi r_c^3$ where $\frac{4}{3}\pi r_c^3$ is the volume of the interaction sphere. The total number of interaction kernel evaluations for computing the interactions for all of the N particles is thus $\frac{N^2}{L^3}\frac{4}{3}\pi r_c^3$, which is $O(N^2)$. This cost is a factor of $\frac{81}{4\pi} \approx 6$ times smaller than the computational cost using cell list, at the expense of the additional memory requirement of Verlet lists.
If the number of particles N increases while keeping the average particle density $\frac{N}{L^3}$ constant, the computational cost of computing particle interactions using Verlet list is $C \cdot O(N)$ where the constant pre-factor $C = \frac{4\pi}{3}\frac{N}{L^3}r_c^3$.
It is also worth noting that the computational cost increases if the particle are inhomogeneously distributed within the computational domain. In the worst case, if all particle are within a distance of r_c from each other, then the computational cost is $O(N^2)$ making Verlet lists worse than cell list given that the cost of constructing Verlet list if also $O(N^2)$ for this case.

2.3 Symmetric interactions

The efficiency of the particle algorithms can be further improved if the interaction kernel $\vec{K}$ is *symmetric*, that is, if $K_p(\vec{x}_p, \vec{x}_q, \vec{\omega}_p, \vec{\omega}_q) = \pm K_q(\vec{x}_q, \vec{x}_p, \vec{\omega}_q, \vec{\omega}_p)$.

Most physics-based interaction kernels are symmetric, i.e., they only depend on the differences of properties and position, but not their absolute values. This symmetry can be exploited to compute every interaction only once, hence reducing the computational cost by a factor of two.
If the change in properties of particle p due to particle q is $\Delta_{p\leftarrow q} = K_p(\vec{x}_p, \vec{x}_q, \vec{\omega}_p, \vec{\omega}_q)$, the change of particle q due to particle p is also known as $\Delta_{q\leftarrow p} = \pm\,\Delta_{p\leftarrow q}$. The sign can be either positive or negative, depending on whether it is a *conserved quantity* (e.g., money transferred between the particles) or a *replicating quantity* (e.g., knowledge shared between agents).
It is also possible that only certain components of $\vec{K}$ are symmetric, and others not. In this case, the symmetric interaction algorithms below can also be used component-wise.

2.3.1 Long-range particle interactions

For long-range particle interactions, the particle algorithm presented in Chapter 1 can be made more efficient by exploiting symmetry as follows:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
end
foreach particle p do
    foreach particle q>=p do
        [kx,kw] = p.interact(q)
        p.positionChange   += kx
        p.propertiesChange += kw
        if p ≠ q
            q.positionChange   ±= kx
            q.propertiesChange ±= kw
        end
    end
    p.evolve()
end
```

Listing 2.5: Long-range symmetric interaction particle algorithm

Notice that `p.evolve()` can now be called inside the loop, since the particle p will not participate in any other interaction after it has been visited. This is in contrast to the asymmetric interactions where first all particles need to interact before all of them can be evolved.

2.3.2 Symmetric short-range interactions using cell list

If the interaction kernels are symmetric and short-range with a cut-off radius r_c that is smaller than the span of the computational domain, we can additionally incorporate cell list. In this scenario, only one half of the adjacent cells need to be looped over in order to exploit symmetry in a consistent fashion. Figure 2.1(b) shows the adjacent cells that need to be looped over for a two-dimensional computational domain. The particle algorithm for symmetric short-range interactions using cell list is:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
end
foreach particle p do
    cellIndex = floor(p.position / rc)
    foreach particle q>=p in cell(cellIndex) do
        [kx,kw] = p.interact(q)
        p.positionChange   += kx
        p.propertiesChange += kw
        if p ≠ q
            q.positionChange   ±= kx
            q.propertiesChange ±= kw
        end
    end
    foreach symmetricAdjacentCellIndex do
        foreach particle q in cell(symmetricAdjacentCellIndex) do
            [kx,kw] = p.interact(q)
            p.positionChange   += kx
            p.propertiesChange += kw
            q.positionChange   ±= kx
            q.propertiesChange ±= kw
        end
    end
end
foreach particle p do
    p.evolve()
end
```

Listing 2.6: Short-range symmetric interaction with cell list

Now, the evolve loop has to be outside again, because p can still be altered by another interaction partner than q if the cell stencil moves over the grid in the corresponding direction.

2.4 Symmetric short-range interactions using Verlet list

Symmetric Verlet lists contain only half the entries of a non-symmetric Verlet list. Every particle-particle interaction is stored only once. The construction of symmetric Verlet lists makes use of only one half of the adjacent cells of the intermediate cell list (see Fig. 2.1(b)) in the cell list, akin to the particle algorithm for symmetric short-range interactions using cell list (Sec. 2.3.2). The construction of symmetric Verlet lists is as follows:

```
foreach particle p do
    cellIndex = floor(p.position / rc)
    foreach particle q>=p in cell(cellIndex) do
        ! compute distance between p and q
        if |p.position-q.position| <= rc
            verlet(p).add(q)
        end
    end
    foreach symmetricAdjacentCellIndex do
```

```
        foreach particle q in cell(symmetricAdjacentCellIndex) do
            if |p.position-q.position| <= rc
                verlet(p).add(q)
            end
        end
    end
end
```

Listing 2.7: Constructing symmetric Verlet list

The particle algorithm using symmetric Verlet list for short-range interactions is as follows:

```
foreach particle p do
    p.positionChange   = 0
    p.propertiesChange = 0
end
foreach particle p do
    foreach particle q in verlet(p) do
        [kx,kw] = p.interact(q)
        p.positionChange   += kx
        p.propertiesChange += kw
        if p ≠ q
            q.positionChange   ±= kx
            q.propertiesChange ±= kw
        end
    end
end
foreach particle p do
    p.evolve()
end
```

Listing 2.8: Short-range symmetric interaction with Verlet lists

Chapter 3

Time stepping algorithms

In this chapter:

- Discrete-time and continuous-time models
- Time stepping schemes for continuous time models
- Discretization error in time stepping schemes
- Numerical stability of time stepping schemes
- Notion of consistency and convergence

Learning goals:

- Know the basic explicit and implicit time stepping schemes
- Be able to define and explain discretization errors and their source
- Define numerical stability and requirements
- Know the difference between explicit and implicit schemes regarding numerical stability
- Be able to choose time stepping schemes based on discretization error and numerical stability

This chapter focusses on the algorithms for the `evolve` method in the particle class introduced in Chapter 1 (see Listing. 1.1). The method `evolve` uses the `p.positionChange` and `p.propertiesChange` variables, which are the rate of position change and rate of property change $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ in Eq. 1.1 summed over all particle interactions. Using these, it updates the position $\vec{x}_p$ and properties $\vec{\omega}_p$ of particle p. For long-range interactions $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ are computed according to the particle algorithm in Listing 1.2. In the case of short-range interactions, fast neighbor lists like cell lists and Verlet lists are used to compute $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ efficiently as presented in Chapter 2.

The implementation of `evolve` depends on whether the model is a discrete-time model or a continuous-time model. We first present the algorithms for discrete-time models before proceeding to those for continuous-time models.
In discrete-time models, the equation for particle p in every particle method is

$$\begin{aligned} \vec{x}_p(n+1) &= \vec{x}_p(n) + \Delta\vec{x}_p(n, \vec{x}(n), \vec{\omega}(n)) \\ \vec{\omega}_p(n+1) &= \vec{\omega}_p(n) + \Delta\vec{\omega}_p(n, \vec{x}(n), \vec{\omega}(n)) \end{aligned} \tag{3.1}$$

where $n \in \mathbb{Z}$ is an integer denoting the discrete time point. The rate of change of position $\Delta\vec{x}_p$ and properties $\Delta\vec{\omega}_p$ (see Eq. 1.1) in a discrete-time model is the same as the change of position and properties respectively, since time is unity in a discrete-time model. In addition, $\Delta\vec{x}_p$ and $\Delta\vec{\omega}_p$ are functions of positions $\vec{x}$ and properties $\vec{\omega}$ of all particles at the n-th time point, and the particle simulation evolves according to the above *difference equations*.
In continuous-time models, the equations of evolution for particle p are

$$\begin{aligned} \frac{\mathrm{d}\vec{x}_p(t)}{\mathrm{d}t} &= \Delta\vec{x}_p = \vec{v}_p(t, \vec{x}(t), \vec{\omega}(t)) \\ \frac{\mathrm{d}\vec{\omega}_p(t)}{\mathrm{d}t} &= \Delta\vec{\omega}_p = \vec{g}_p(t, \vec{x}(t), \vec{\omega}(t)) \end{aligned} \tag{3.2}$$

where $t \in \mathbb{R}$ is a real number denoting continuous time. Here $\Delta\vec{x}_p = \vec{v}_p$ and $\Delta\vec{\omega}_p = \vec{g}_p$ are the rates of change of the position and the properties of particle p, respectively. For convenience, we refer to $\vec{v}_p$ as the *velocity* of particle p and $\vec{g}_p$ as the *property rate* of particle p. In general, the velocity $\vec{v}_p$ and property rate $\vec{g}_p$ are defined as functions of positions $\vec{x}$ and properties $\vec{\omega}$ of all particles at time t. The velocity $\vec{v}_p$ and property rate $\vec{g}_p$ are computed as a sum of interactions between all pairs of particles as given by Eq. 1.1. Therefore, the particle interaction algorithms are the same as in the discrete-time case, albeit with a different interaction kernel $\vec{K}$, and the particle simulation evolves according to the above *differential equations*. In order to numerically approximate the solution of a continuous differential equation in the computer, it needs to be discretized (see Section 1.1).

3.1 Time stepping for discrete-time models

In a discrete-time model, the position of particle p is simply incremented by $\Delta\vec{x}_p$, and the properties are incremented by $\Delta\vec{\omega}_p$ according to Eq. 3.1.
The pseudocode for `evolve` is:

```
method evolve():
    "this" refers to the particle of which the method is a member
    this.position   += this.positionChange
    this.properties += this.propertiesChange
```

Listing 3.1: Method `evolve` of a particle algorithm for a discrete-time model

Hence, the simulation jumps from time point to time point.

3.2 Time stepping for continuous-time models

For continuous-time models (Eq. 5.1), the continuous-time derivatives $\frac{\mathrm{d}\vec{x}_p}{\mathrm{d}t}$ and $\frac{\mathrm{d}\vec{\omega}_p}{\mathrm{d}t}$ need to be discretized. Discretization refers to the process of converting these continuous time derivatives into discrete counterparts so that derivatives can be evaluated on a computer. The conversion enables evaluation of time derivatives at discrete times t_n where $n = 0, \ldots, N$; t_n is the time stamp at the n-th time step. The size of the time step (i.e., how much time advanced since the last discretization point) is $\delta t = t_n - t_{n-1}$. Since time is continuous, the time-step size δt may be different between any two consecutive time steps. For simplicity in the notation, however, we here assume that all time points are equally spaced so that δt is a constant.

3.2.1 Explicit schemes for time stepping

Time discretization schemes are referred to as *explicit* if the values of the quantities of interest ($\vec{x}_p$ and $\vec{\omega}_p$ for particles algorithms) at time t_n can be computed using the values of these quantities at times t_k such that $k < n$. In other words, time discretization scheme are referred to as explicit if future states of the system can be computed using only the current and past states of the system.

3.2.1.1 Explicit Euler scheme

The "explicit Euler" time discretization scheme (also referred to as "forward Euler" scheme) is the one of the simplest ways to discretize time derivatives. In this scheme, time derivative of a quantity y at time t_n is approximated as

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) \approx \frac{y(t_{n+1}) - y(t_n)}{t_{n+1} - t_n} = \frac{y(t_{n+1}) - y(t_n)}{\delta t}. \tag{3.3}$$

Therefore, a differential equation of the form $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(y(t))$ is discretized as

$$\begin{aligned} \frac{y(t_{n+1}) - y(t_n)}{\delta t} &\approx f(y(t_n)) \\ \textit{i.e.,} \qquad y(t_{n+1}) &\approx y(t_n) + \delta t\, f(y(t_n)), \qquad n = 0, 1, \ldots. \end{aligned}$$

This scheme is explicit since $y(t_{n+1})$ can be computed with the knowledge of $y(t_n)$.

Let's now assume one particle residing in a one-dimensional computational domain. The particle moves according to a velocity, and the velocity changes according to an acceleration. That is,

$$\begin{aligned} \frac{\mathrm{d}x(t)}{\mathrm{d}t} &= v(t) \\ \frac{\mathrm{d}v(t)}{\mathrm{d}t} &= a(t), \end{aligned} \tag{3.4}$$

where x is the particle position, v the velocity and a the acceleration. The index p is omitted since there is only one particle. Using the explicit Euler scheme, these equation are approximated as

$$\begin{aligned} x(t_{n+1}) &= x(t_n) + \delta t\, v(t_n), \\ v(t_{n+1}) &= v(t_n) + \delta t\, a(t_n), \qquad n = 0, 1, \ldots. \end{aligned}$$

Therefore, given the initial position $x(0)$, initial velocity $v(0)$, and the acceleration $a(t)$ at all times, we can march forward in time to obtain the position of the particle at all times.

Similarly, we generalize the explicit Euler scheme for the general equations of continuous-time particle methods (Eq. 5.1). Discretizing at time t_n results in

$$\begin{aligned} \frac{\vec{x}_p(t_{n+1}) - \vec{x}_p(t_n)}{\delta t} &\approx \vec{v}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)) \\ \frac{\vec{\omega}_p(t_{n+1}) - \vec{\omega}_p(t_n)}{\delta t} &\approx \vec{g}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)). \end{aligned} \tag{3.5}$$

Therefore,

$$\begin{aligned} \vec{x}_p(t_{n+1}) &\approx \vec{x}_p(t_n) + \vec{v}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n))\, \delta t \\ \vec{\omega}_p(t_{n+1}) &\approx \vec{\omega}_p(t_n) + \vec{g}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n))\, \delta t, \qquad n = 0, 1, \ldots. \end{aligned} \tag{3.6}$$

This scheme is explicit since the next state ($\vec{x}_p(t_{n+1})$ and $\vec{\omega}_p(t_{n+1})$) of particle p can be evaluated using the current state ($\vec{x}_p(t_n)$ and $\vec{\omega}_p(t_n)$) of particle p. The method `evolve` is therefore given by

```
method evolve():
    ! Time step size dt is a parameter
    this.position   += this.positionChange * dt
    this.properties += this.propertiesChange * dt
```

Listing 3.2: Method `evolve` of a particle algorithm for a continuous-time model using explicit Euler time-stepping

Discretization error. In this section, we analyze the error made in discretizing the continuous time derivative using the above discretization schemes. Using the notation for the state of the system $\vec{s} = [\vec{x}_1, \ldots, \vec{x}_p, \ldots, \vec{x}_N, \vec{\omega}_1, \ldots, \vec{\omega}_p, \ldots, \vec{\omega}_N]^{\mathrm{T}}$ and the rate vector of the system $\vec{r}(t, \vec{s}) = [\vec{v}_1, \ldots, \vec{v}_p, \ldots, \vec{v}_N\, \vec{g}_1, \ldots, \vec{g}_2, \ldots, \vec{g}_N]^{\mathrm{T}}$, the equation for a system of particles is

$$\frac{\mathrm{d}\vec{s}}{\mathrm{d}t} = \vec{r}(t, \vec{s}). \tag{3.7}$$

We now analyze the error and the approximation order of discretizing the continuous time derivative using the discretization schemes presented in this chapter. Using the explicit Euler scheme,

$$\vec{s}(t_{n+1}) \approx \vec{s}(t_n) + \delta t\, \vec{r}(t_n, \vec{s}(t_n)). \tag{3.8}$$

Equivalently,

$$\vec{s}(t_n + \delta t) \approx \vec{s}(t_n) + \delta t\, \vec{r}(t_n, \vec{s}(t_n)). \tag{3.9}$$

Expanding $\vec{s}(t_n + \delta t)$ around δt using a Taylor series expansion, assuming that δt is small and $\vec{s}(t)$ is a perfectly smooth function in t, we observe that:

$$\begin{aligned}
\vec{s}(t_n + \delta t) &= \vec{s}(t_n) + \frac{\delta t}{1!}\frac{\mathrm{d}\vec{s}}{\mathrm{d}t}(t_n) + \frac{\delta t^2}{2!}\frac{\mathrm{d}^2\vec{s}}{\mathrm{d}t^2}(t_n) + \frac{\delta t^3}{3!}\frac{\mathrm{d}^3\vec{s}}{\mathrm{d}t^3}(t_n) + \dots \\
&= \vec{s}(t_n) + \frac{\delta t}{1!}\vec{r}(t_n, \vec{s}(t_n)) + \frac{\delta t^2}{2!}\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}(t_n, \vec{s}(t_n)) \\
&\quad + \frac{\delta t^3}{3!}\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2}(t_n, \vec{s}(t_n)) + \dots .
\end{aligned}$$

As terms get successively smaller, the term $\frac{1}{2!}\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}(t_n, \vec{s}(t_n))\, O(\delta t^2)$ dominates the error as δt approaches 0. Therefore,

$$\vec{s}(t_n + \delta t) = \vec{s}(t_n) + \delta t\, \vec{r}(t_n, \vec{s}(t_n)) + \frac{1}{2!}\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}(t_n, \vec{s}(t_n))\, O(\delta t^2). \tag{3.10}$$

Comparing Eqs. 3.9 and 3.10, we see that the discretization error per time-step using the explicit Euler scheme is $O(\delta t^2)$. Therefore, the local order of approximation in an explicit Euler scheme is 2 (the exponent of δt in the leading error term). To compute the state of the system at time $t = T$, we need to perform $N_\mathrm{t} = \frac{T}{\delta t}$ time steps. The overall discretization error is therefore $N_\mathrm{t} O(\delta t^2) = \frac{T}{\delta t} O(\delta t^2) = T O(\delta t) = O(\delta t)$. The global order of approximation of the explicit Euler scheme is thus 1. The explicit Euler method is therefore said to be first-order accurate. This means that as we half the time-step δt, the error decreases by a factor of 2. This may require very small δt to achieve a prescribed error. Therefore, it may be beneficial to have schemes that are of higher order.

3.2.1.2 Leapfrog scheme

The Euler scheme is not the only way of discretizing time. In fact, there are infinitely many time-stepping algorithms that differ with respect to their computational cost and accuracy. Another popular example is the "Leapfrog scheme". In a leapfrog scheme, time derivative of a quantity y at time t_n is approximated as

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) \approx \frac{y(t_{n+1}) - y(t_{n-1})}{t_{n+1} - t_{n-1}} = \frac{y(t_{n+1}) - y(t_{n-1})}{2\,\delta t}.$$

A differential equation of the form $\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(y(t))$ is approximated as

$$\begin{aligned}
\frac{y(t_{n+1}) - y(t_{n-1})}{2\delta t} &\approx f(y(t_n)) \qquad (3.11)\\
i.e., \qquad y(t_{n+1}) &\approx y(t_{n-1}) + 2\delta t\, f(y(t_n)), \qquad n = 0, 1, \dots .
\end{aligned}$$

The leapfrog scheme is therefore explicit but apart from the knowledge of the y at t_n, it also requires the knowledge of y at t_{n-1}.

For a single particle moving according to a velocity field v and acceleration a (Eq. 3.4), the leapfrog scheme results in

$$\begin{aligned} x(t_{n+1}) &= x(t_{n-1}) + 2\delta t\, v(t_n), \\ v(t_{n+1}) &= v(t_{n-1}) + 2\delta t\, a(t_n), \qquad n = 0, 1, \ldots. \end{aligned}$$

To march in time, the leapfrog scheme not only requires the initial position $x(0)$, initial velocity $v(0)$, and the function $a(t)$, but also the position x and velocity v at time point $t_{-1} = -\delta t$. In comparison to explicit Euler, the leapfrog scheme therefore takes into account more information about the past state of the system.

Generalizing the leapfrog scheme for the equations of continuous-time particle methods p (Eq. 5.1), the discretized equations at time t_n are

$$\begin{aligned} \frac{\vec{x}_p(t_{n+1}) - \vec{x}_p(t_{n-1})}{2\,\delta t} &\approx \vec{v}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)) \\ \frac{\vec{\omega}_p(t_{n+1}) - \vec{\omega}_p(t_{n-1})}{2\,\delta t} &\approx \vec{g}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)). \end{aligned} \tag{3.12}$$

Therefore,

$$\begin{aligned} \vec{x}_p(t_{n+1}) &\approx \vec{x}_p(t_{n-1}) + 2\,\vec{v}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n))\,\delta t \\ \vec{\omega}_p(t_{n+1}) &\approx \vec{\omega}_p(t_{n-1}) + 2\,\vec{g}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n))\,\delta t, \qquad n = 0, 1, \ldots. \end{aligned} \tag{3.13}$$

This scheme is explicit since the next state ($\vec{x}_p(t_{n+1})$ and $\vec{\omega}_p(t_{n+1})$) of particle p can be evaluated using the current state ($\vec{x}_p(t_n)$ and $\vec{\omega}_p(t_n)$) of particle p and the previous state ($\vec{x}_p(t_{n-1})$ and $\vec{\omega}_p(t_{n-1})$) of particle p. The leap frog scheme therefore requires storing of the old state of the system in memory.
The method `evolve` using the leap-frog scheme is

```
method evolve():
    ! the time-step size dt is a parameter
    this.position   = this.position_old + this.positionChange * dt * 2
    this.properties = this.properties_old + this.propertiesChange * dt * 2
```

Listing 3.3: Method `evolve` of a particle algorithm for a continuous-time model using leap-frog time-stepping

Discretization error. The leapfrog scheme for Eq. 3.7 is given by

$$\vec{s}(t_n + \delta t) \approx \vec{s}(t_n - \delta t) + 2\delta t\, \vec{r}(t_n, \vec{s}(t_n)). \tag{3.14}$$

Expanding $\vec{s}(t_n + \delta t)$ around δt:

$$\begin{aligned} \vec{s}(t_n + \delta t) &= \vec{s}(t_n) + \frac{\delta t}{1!}\,\vec{r}(t_n, \vec{s}(t_n)) + \frac{\delta t^2}{2!}\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}(t_n, \vec{s}(t_n)) \\ &\quad + \frac{\delta t^3}{3!}\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2}(t_n, \vec{s}(t_n)) + \frac{\delta t^4}{4!}\frac{\mathrm{d}^4\vec{r}}{\mathrm{d}t^4}(t_n, \vec{s}(t_n)) + \ldots. \end{aligned}$$

Similarly, expanding $\vec{s}(t_n - \delta t)$ around δt:

$$\begin{aligned}\vec{s}(t_n - \delta t) &= \vec{s}(t_n) - \frac{\delta t}{1!}\vec{r}(t_n, \vec{s}(t_n)) + \frac{\delta t^2}{2!}\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}(t_n, \vec{s}(t_n)) \\ &\quad -\frac{\delta t^3}{3!}\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2}(t_n, \vec{s}(t_n)) + \frac{\delta t^4}{4!}\frac{\mathrm{d}^4\vec{r}}{\mathrm{d}t^4}(t_n, \vec{s}(t_n)) + \ldots.\end{aligned}$$

Therefore,

$$\begin{aligned}\vec{s}(t_n + \delta t) - \vec{s}(t_n - \delta t) &= 2\frac{\delta t}{1!}\vec{r}(t_n, \vec{s}(t_n)) + 2\frac{\delta t^3}{3!}\frac{\mathrm{d}^2\vec{r}}{\mathrm{d}t^2}(t_n, \vec{s}(t_n)) + \ldots \\ &= 2\delta t\,\vec{r}(t_n, \vec{s}(t_n)) + O(\delta t^3).\end{aligned}$$

Equivalently,

$$\vec{s}(t_n + \delta t) = \vec{s}(t_n - \delta t) + 2\delta t\,\vec{r}(t_n, \vec{s}(t_n)) + O(\delta t^3). \tag{3.15}$$

Comparing Eqs. 3.14 and 3.15, we see that the local discretization error of the leapfrog scheme is $O(\delta t^3)$ and therefore the local order of approximation is 3. This means that the global discretization error is $O(\delta t^2)$ and therefore the global order of approximation is 2. Leapfrog time-stepping is therefore a second-order accurate. This means that as we half the time step δt, the error decreases by a factor of 4.

3.2.1.3 Runge-Kutta-4 scheme

So far, time stepping was always done in a single forward operation. This can be generalized to multi-stage schemes where the accuracy can be arbitrarily increased. Runge-Kutta schemes are a particularly popular family of explicit multi-stage time-stepping schemes. In multi-stage time-stepping schemes, evaluation of the future state at t_{n+1} given the current state at t_n requires evaluating $\vec{v}_p$ and $\vec{g}_p$ at intermediate times between t_n and t_{n+1}. Among multi-stage Runge-Kutta schemes, Runge-Kutta-4 (RK4) or the four-stage Runge-Kutta scheme is the most popular.
The RK4 time-stepping scheme for an equation

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t) = f(t, y(t)) \tag{3.16}$$

is

$$y(t_{n+1}) \approx y(t_n) + \frac{\delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where (3.17)

$$k_1 = f(t_n, y(t_n)) \tag{3.18}$$

$$k_2 = f(t_{n+\frac{1}{2}}, y(t_n) + \frac{\delta t}{2}k_1) \tag{3.19}$$

$$k_3 = f(t_{n+\frac{1}{2}}, y(t_n) + \frac{\delta t}{2}k_2) \tag{3.20}$$

$$k_4 = f(t_{n+1}, y(t_n) + \delta t\,k_3). \tag{3.21}$$

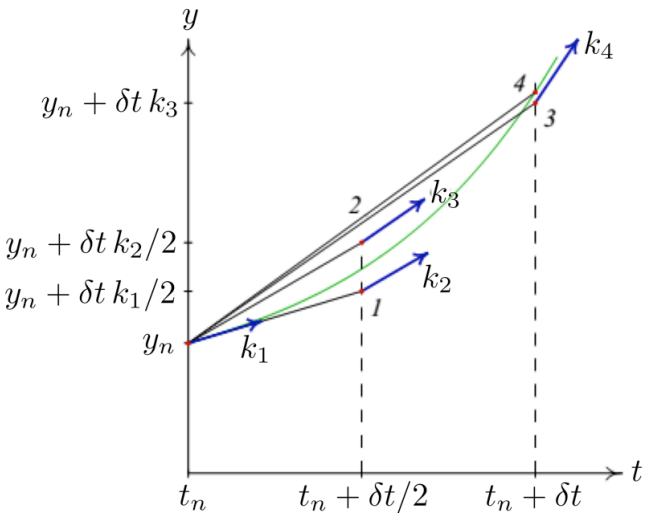


Figure 3.1: Illustration of the Runge-Kutta-4 scheme corresponding to Eqs. 3.17–3.21 for the model equation in Eq. 3.16. (Adapted from figure in wikipedia, CC license)

This is illustrated in Fig. 3.1 and can be understood as a sequence of three intermediate Euler steps.
Applying the RK4 scheme to the general equations of continuous-time particle methods, $\vec{x}_p(t_{n+1})$ and $\vec{\omega}_p(t_{n+1})$ are is written as

$$\begin{aligned}
\vec{x}_p(t_{n+1}) &\approx \vec{x}_p(t_n) + \frac{\delta t}{6}\left(\vec{a}_{1\mathrm{p}} + 2\vec{a}_{2\mathrm{p}} + 2\vec{a}_{3\mathrm{p}} + \vec{a}_{4\mathrm{p}}\right) \\
\vec{\omega}_p(t_{n+1}) &\approx \vec{\omega}_p(t_n) + \frac{\delta t}{6}\left(\vec{b}_{1\mathrm{p}} + 2\vec{b}_{2\mathrm{p}} + 2\vec{b}_{3\mathrm{p}} + \vec{b}_{4\mathrm{p}}\right), \qquad (3.22)
\end{aligned}$$

where

$$\begin{aligned}
\vec{a}_{1\mathrm{p}} &= \vec{v}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)) && (3.23)\\
\vec{a}_{2\mathrm{p}} &= \vec{v}_p(t_{n+\frac{1}{2}}, \vec{x}(t_n) + \frac{\delta t}{2}\vec{a}_1, \vec{\omega}(t_n) + \frac{\delta t}{2}\vec{b}_1) && (3.24)\\
\vec{a}_{3\mathrm{p}} &= \vec{v}_p(t_{n+\frac{1}{2}}, \vec{x}(t_n) + \frac{\delta t}{2}\vec{a}_2, \vec{\omega}(t_n) + \frac{\delta t}{2}\vec{b}_2) && (3.25)\\
\vec{a}_{4\mathrm{p}} &= \vec{v}_p(t_{n+1}, \vec{x}(t_n) + \delta t\,\vec{a}_3, \vec{\omega}(t_n) + \delta t\,\vec{b}_3) && (3.26)\\
\vec{b}_{1\mathrm{p}} &= \vec{g}_p(t_n, \vec{x}(t_n), \vec{\omega}(t_n)) && (3.27)\\
\vec{b}_{2\mathrm{p}} &= \vec{g}_p(t_{n+\frac{1}{2}}, \vec{x}(t_n) + \frac{\delta t}{2}\vec{a}_1, \vec{\omega}(t_n) + \frac{\delta t}{2}\vec{b}_1) && (3.28)\\
\vec{b}_{3\mathrm{p}} &= \vec{g}_p(t_{n+\frac{1}{2}}, \vec{x}(t_n) + \frac{\delta t}{2}\vec{a}_2, \vec{\omega}(t_n) + \frac{\delta t}{2}\vec{b}_2) && (3.29)\\
\vec{b}_{4\mathrm{p}} &= \vec{g}_p(t_{n+1}, \vec{x}(t_n) + \delta t\,\vec{a}_3, \vec{\omega}(t_n) + \delta t\,\vec{b}_3), && (3.30)
\end{aligned}$$

with

$$\vec{a}_i = [\vec{a}_{i1}, \ldots, \vec{a}_{ip}, \ldots, \vec{a}_{iN}]^{\mathrm{T}}, \qquad i = 1, \ldots, 4$$
$$\vec{b}_i = [\vec{b}_{i1}, \ldots, \vec{b}_{ip}, \ldots, \vec{b}_{iN}]^{\mathrm{T}}, \qquad i = 1, \ldots, 4.$$

Denoting the state of the system as $\vec{s} = [\vec{x}_1, \ldots, \vec{x}_p, \ldots, \vec{x}_N, \vec{\omega}_1, \ldots, \vec{\omega}_p, \ldots, \vec{\omega}_N]^{\mathrm{T}}$ and the rate vector of the system $\vec{r}(t, \vec{s}) = [\vec{v}_1, \ldots, \vec{v}_p, \ldots, \vec{v}_N \vec{g}_1, \ldots, \vec{g}_p, \ldots, \vec{g}_N]^{\mathrm{T}}$, the equation of for a system of particles can be written as

$$\frac{\mathrm{d}\vec{s}}{\mathrm{d}t} = \vec{r}(t, \vec{s}). \tag{3.31}$$

In this formulation, the above Runge-Kutta scheme can be written in a concise way:

$$\vec{s}(t_{n+1}) \approx \vec{s}(t_n) + \frac{\delta t}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4) \tag{3.32}$$

where

$$\vec{k}_1 = \vec{r}(t_n, \vec{s}(t_n)) \tag{3.33}$$
$$\vec{k}_2 = \vec{r}(t_{n+\frac{1}{2}}, \vec{s}(t_n) + \frac{\delta t}{2}\vec{k}_1) \tag{3.34}$$
$$\vec{k}_3 = \vec{r}(t_{n+\frac{1}{2}}, \vec{s}(t_n) + \frac{\delta t}{2}\vec{k}_2) \tag{3.35}$$
$$\vec{k}_4 = \vec{r}(t_{n+1}, \vec{s}(t_n) + \delta t\, \vec{k}_3). \tag{3.36}$$

Discretization error. Following similar derivation as for the previous methods, we can see that the local discretization error and the global discretization error for RK4 is $O(\delta t^5)$ and $O(\delta t^4)$ respectively. The RK4 scheme is therefore fourth-order accurate.

3.2.2 Implicit scheme

Time discretization schemes are referred to as *implicit* if the values of dynamical quantities of interest ($\vec{x}_p$ and $\vec{\omega}_p$ for particles algorithms) at time t_n is computed using the values of these quantities at times t_k such that at least one $k \geq n$. In other words, time discretization scheme are referred to as implicit if at least the present or one future state of the system is used to compute the present state. This hence leads to an implicit equation that needs to be solved.

3.2.2.1 Implicit Euler scheme

The simplest implicit scheme is the implicit variant of the Euler scheme, the "Implicit Euler" time discretization scheme (also referred to as "backward Euler" scheme). Therein, the time derivative of a quantity y at time t_{n+1} is

approximated as

$$\frac{\mathrm{d}y}{\mathrm{d}t}(t_{n+1}) \approx \frac{y(t_{n+1}) - y(t_n)}{t_{n+1} - t_n} = \frac{y(t_{n+1}) - y(t_n)}{\delta t}. \tag{3.37}$$

Applying this scheme the general equations of continuous-time particle methods, the equation for particle p at time t_{n+1} is approximated as

$$\begin{aligned} \frac{\vec{x}_p(t_{n+1}) - \vec{x}_p(t_n)}{\delta t} &\approx \vec{v}_p(t_{n+1}, \vec{x}(t_{n+1}), \vec{\omega}(t_{n+1})) \\ \frac{\vec{\omega}_p(t_{n+1}) - \vec{\omega}_p(t_n)}{\delta t} &\approx \vec{g}_p(t_{n+1}, \vec{x}(t_{n+1}), \vec{\omega}(t_{n+1})). \end{aligned} \tag{3.38}$$

It is evident from the above equation, that the next state of the system is dependent on $\vec{v}_p$ and $\vec{g}_p$ evaluated at the next state, a typical consequence if implicit schemes. Therefore, implicit schemes always result in a system of equations that needs to be numerically solved to obtain $\vec{x}_p(t_{n+1})$ and $\vec{\omega}_p(t_{n+1})$. Here, we provide a complete example of a implicit scheme using the notation $\vec{s} = [\vec{x}_1, \ldots, \vec{x}_p, \ldots, \vec{x}_N, \vec{\omega}_1, \ldots, \vec{\omega}_p, \ldots, \vec{\omega}_N]^{\mathrm{T}}$ and the rate vector of the system $\vec{r}(t, \vec{s}) = [\vec{v}_1, \ldots, \vec{v}_p, \ldots, \vec{v}_N\, \vec{g}_1, \ldots, \vec{g}_p, \ldots, \vec{g}_N]^{\mathrm{T}}$. We further assume that the interaction kernel $K(\vec{x}_p, \vec{x}_q, \vec{\omega}_p, \vec{\omega}_q) = a_{p,p}\vec{x}_p + a_{p,q}\vec{x}_q + a_{p,N+p}\vec{\omega}_p + a_{p,N+q}\vec{\omega}_q$, that is, the interaction kernel in linear in particle position and particle properties. In this case, $\vec{r} = \mathbf{A}\,\vec{s}$, where the elements $a_{i,j}$ of the matrix are given by the coefficients defining the interaction kernels. In addition, we impose that the coefficients of $\mathbf{A}$ are independent of time t. In this case, the equation for the system of particles using implicit Euler is approximated as

$$\frac{\vec{s}(t_{n+1}) - \vec{s}(t_n)}{\delta t} \approx \mathbf{A}\,\vec{s}(t_{n+1}). \tag{3.39}$$

Therefore,

$$(\mathbf{I} - \delta t\,\mathbf{A})\,\vec{s}(t_{n+1}) \approx \vec{s}(t_n), \tag{3.40}$$

where $\mathbf{I}$ is the identity matrix. This equation is a system of linear equations that needs to be solved to obtain $\vec{s}(t_{n+1})$ by computing the inverse of matrix $(\mathbf{I} - \delta t\,\mathbf{A})$:

$$\vec{s}(t_{n+1}) \approx (\mathbf{I} - \delta t\,\mathbf{A})^{-1}\,\vec{s}(t_n), \tag{3.41}$$

where $(\mathbf{I} - \delta t\,\mathbf{A})^{-1}$ is the inverse of matrix $(\mathbf{I} - \delta t\,\mathbf{A})$.
It is worth noting that matrix $(\mathbf{I} - \delta t\,\mathbf{A})$ is a $2N \times 2N$ matrix and the computational complexity of inverting a dense matrix of size $2N \times 2N$ is $O(N^3)$ using Gaussian elimination. Using state-of-the-art algorithms, this complexity can be reduced to $O(N^{2.373})$. In contrast, all the explicit scheme presented in this chapter have a computational complexity of $O(N)$. Even if the matrix is sparse (typically for short-range interactions), the computational complexity is

$O(N^a)$ where $1 \leq a \leq 2$. Therefore, the computational complexity using an implicit scheme is $\Omega(N)$ and $O(N^3)$ as opposed to all explicit schemes presented in this chapter for which the computational complexity is $\Theta(N)$. As we will see next, implicit schemes however have advantageous stability properties, which may amortize the increased computational cost.

Discretization error. The local discretization error and the global discretization error for the implicit Euler scheme are $O(\delta t^2)$ and $O(\delta t)$, respectively. This follows from the same derivation as for the explicit Euler scheme.

3.2.3 Numerical stability

In the previous section, we have learned about different time-stepping schemes and their discretization errors. Here, we analyze the numerical stability of these schemes. Numerical stability is a property required for all time-stepping schemes. This property requires that when the exact solution of the equation that is discretized is bounded for all times, then the discretization also remains bounded at all times. In other words, if we are simulating stable dynamics, the simulation should also be stable, i.e., not diverge to infinity. Numerical stability is usually analyzed in the linear case ("linear stability") using the linear differential equation:

$$\frac{\mathrm{d}x}{\mathrm{d}t}(t) = \lambda\, x(t), \tag{3.42}$$

where $\lambda = \lambda_\mathrm{R} + i\,\lambda_\mathrm{I}$ is in general complex with λ_R being the real part and λ_I the imaginary part. The solution of this equation is

$$\begin{aligned} x(t) &= x(0)\,\mathrm{e}^{\lambda t} \\ &= x(0)\,\mathrm{e}^{\lambda_\mathrm{R} t}\,\mathrm{e}^{i\,\lambda_\mathrm{I} t} \\ &= x(0)\,\mathrm{e}^{\lambda_\mathrm{R} t}\,(\cos\lambda_\mathrm{I} t + i \sin\lambda_\mathrm{I} t). \end{aligned}$$

This solution is bounded for all times t if $\lambda_\mathrm{R} \leq 0$ and $\lambda_\mathrm{I} \in \mathbb{R}$. If λ is purely real, $x(t)$ merely decays to 0 as t becomes large. If λ is purely imaginary $x(t)$ shows oscillatory behavior as a function of t.

An important theorem (Strang Theorem) states that if a time-stepping scheme is stable for this simple linear test problem, then it is also stable for all other problems. It is hence sufficient (but not necessary) for stability, to ensure that the discretized version of the above model problem remains bounded for all times.

Note that linear stability is a conservative requirement. If the time-stepping is linearly stable, it is also stable for all other cases. However, it can be that it is stable for non-linear cases without being stable in the linear case (Lyapunov stability). Linear stability therefore implies Lyapunov stability, but not vice versa.

3.2.3.1 Explicit Euler

The explicit Euler scheme for Eq. 3.42 gives

$$x(t_{n+1}) = x(t_n) + \delta t\, \lambda\, x(t_n) = (1 + \delta t \lambda) x(t_n).$$

This means that the solution is multiplied by the number $(1 + \delta t \lambda)$ at each time step. After $(n + 1)$ time steps, we hence have:

$$\begin{aligned} x(t_{n+1}) &= (1 + \lambda\, \delta t)^{n+1} x(0), \\ &= \rho^{n+1} x(0), \end{aligned} \tag{3.43}$$

where $\rho = (1 + \lambda\, \delta t)$ is referred to as the *amplification factor* of the time-stepping scheme. For the discretized system to be bounded, and therefore stable, at all times, it is easy to see that we require

$$|\rho| \leq 1. \tag{3.44}$$

That is,

$$\begin{aligned} |(1 + \lambda\, \delta t)| &\leq 1, \\ i.e., \quad |[1 + (\lambda_{\mathrm{R}} + i\, \lambda_{\mathrm{I}})\, \delta t]| &\leq 1, \\ i.e., \quad \sqrt{(1 + \lambda_{\mathrm{R}} \delta t)^2 + (\lambda_{\mathrm{I}} \delta t)^2} &\leq 1, \\ i.e., \quad (1 + \lambda_{\mathrm{R}} \delta t)^2 + (\lambda_{\mathrm{I}} \delta t)^2 &\leq 1. \end{aligned} \tag{3.45}$$

This inequality defines the *region of stability* of explicit Euler discretization (see Fig. 3.2).
Since $\lambda_{\mathrm{R}} < 0$ for our model Eq. 3.42 to be bounded, we can write $\lambda_{\mathrm{R}} = -\,|\lambda_{\mathrm{R}}|$. Making this substitution in Eq. 3.45

$$\begin{aligned} (1 - |\lambda_{\mathrm{R}}|\, \delta t)^2 + (\lambda_{\mathrm{I}} \delta t)^2 &\leq 1 \\ 1 - 2|\lambda_R| \delta t + |\lambda_R|^2 \delta t^2 + \lambda_I^2 \delta t^2 &\leq 1 \\ (|\lambda_R|^2 + \lambda_I^2) \delta t^2 &\leq 2|\lambda_R| \delta t \\ i.e., \quad \delta t &\leq \tfrac{2\,|\lambda_{\mathrm{R}}|}{|\lambda_{\mathrm{R}}|^2 + \lambda_{\mathrm{I}}^2}. \end{aligned} \tag{3.46}$$

Therefore, for the explicit Euler scheme of Eq. 3.42 to be numerically stable, δt must fulfil Eq. 3.46. We call the explicit Euler scheme *conditionally stable*, because the sufficient condition for stability is given by Eq. 3.46.
If λ is purely real (i.e., $\lambda_{\mathrm{I}} = 0$), then

$$\delta t \leq \frac{2}{|\lambda_{\mathrm{R}}|}. \tag{3.47}$$

The explicit Euler scheme is therefore *conditionally stable* when λ is purely real, where the condition for stability is given by Eq. 3.47.
If λ is purely imaginary (i.e., $\lambda_{\mathrm{R}} = 0$), then

$$\delta t \leq 0. \tag{3.48}$$

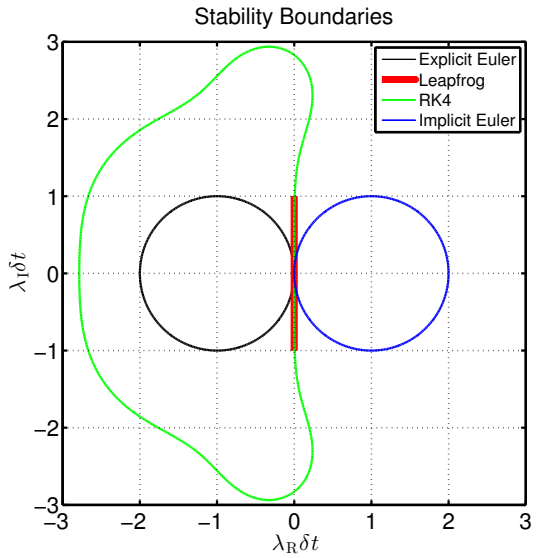


Figure 3.2: The numerical stability boundaries for explicit Euler, leapfrog, Runge-Kutta 4, and Implicit Euler. The region of stability for explicit Euler, leapfrog and Runge-Kutta 4 is within the boundary, whereas that of Implicit Euler is outside its boundary.

This condition, however, cannot be satisfied since δt must be greater than 0 in order to advance time. Therefore, the explicit Euler scheme is *unconditionally unstable* when λ is purely imaginary. In other words, explicit Euler scheme is always linearly unstable for purely oscillatory dynamics (but not necessarily Lyapunov-unstable!).

3.2.3.2 Leapfrog

The leapfrog scheme for Eq. 3.42 leads to

$$x(t_{n+1}) = x(t_{n-1}) + 2\delta t\, \lambda\, x(t_n). \tag{3.49}$$

It is not easy to factor something out here in order to get at the amplification factor. Nevertheless, we can simply assume that such a factor exists, i.e., there is a number with which the solution gets multiplied at each step. From this definition of the amplification factor ρ, we thus substitute $x(t_{n+1})$ with $\rho^2 x(t_{n-1})$, and $x(t_n)$ with $\rho x(t_{n-1})$ in Eq. 3.49. Factoring out and dividing by $x(t_{n-1})$, this substitution results in the following equation:

$$\rho^2 - 2\delta t \lambda \rho - 1 = 0.$$

This is the equation that the amplification factor needs to fulfill. Solving this quadratic equation, we find the following two amplification factors:

$$\rho_{1,2} = \lambda\delta t \pm \sqrt{\lambda^2\delta t^2 + 1}.$$

Since both could be true, we can only be sure of stability if both are less than one. The condition for stability hence is:

$$|\rho_1| \leq 1 \quad \text{and} \quad |\rho_2| \leq 1.$$

We now find the condition on δt that fulfills the above requirements. We start by observing that the product of ρ_1 and ρ_2 is:

$$\begin{aligned} \rho_1\rho_2 &= (\lambda\delta t + \sqrt{\lambda^2\delta t^2 + 1})(\lambda\delta t - \sqrt{\lambda^2\delta t^2 + 1}) \\ &= \lambda^2\delta t^2 - \lambda\delta t\sqrt{\lambda^2\delta t^2 + 1} + \lambda\delta t\sqrt{\lambda^2\delta t^2 + 1} - \lambda^2\delta t^2 - 1 \\ &= -1. \end{aligned} \tag{3.50}$$

Therefore, $|\rho_1\rho_2| = |\rho_1||\rho_2| = 1$. As a consequence, if $|\rho_1| > 1$, then $|\rho_2| < 1$ and vice versa. Therefore, the only possibility of fulfilling the stability criteria is when $|\rho_1| = |\rho_2| = 1$. That is,

$$|\rho_1| = |\lambda\delta t + \sqrt{\lambda^2\delta t^2 + 1}| = 1 \quad \text{and} \quad |\rho_2| = |\lambda\delta t - \sqrt{\lambda^2\delta t^2 + 1}| = 1.$$

Setting $\lambda = -|\lambda_{\mathrm{R}}| + i\lambda_{\mathrm{I}}$, we find that $|\rho_1| = |\rho_2| = 1$ only when $\lambda_{\mathrm{R}} = 0$ and $|\lambda_{\mathrm{I}}|\,\delta t \leq 1$. Therefore, the leapfrog scheme is *conditionally stable* only for purely oscillatory dynamics.

3.2.3.3 RK4

Following the same methodology as in the previous section, one finds that the amplification factor ρ for the RK4 scheme is:

$$\rho = \sum_{k=0}^{4} \frac{(\delta t\,\lambda)^k}{k!}.$$

The region of stability for RK4 is shown in Fig. 3.2. RK4 is conditionally stable for both non-oscillatory and purely oscillatory dynamics.

3.2.3.4 Implicit Euler

The amplification factor ρ for implicit Euler is found as:

$$\rho = \frac{1}{1 - \lambda\delta t}. \tag{3.51}$$

Thus, for stability we have to require:

$$\begin{aligned} |\rho| &\leq 1 \\ \frac{1}{|1 - \lambda\delta t|} &\leq 1 \\ 1 &\leq |1 - \lambda\delta t| \end{aligned}$$

This is the stability region shown in Fig. 3.2. It is the *outside* of a circle with center 1 and radius 1. This includes the entire negative half-plane, meaning that discretized function always remains bounded when the exact function is bounded. This means that the implicit Euler scheme is always stable, for any δt. Implicit Euler is therefore called *unconditionally stable*. It should, however, be noted that stability does not guarantee accuracy since the error of implicit Euler is $O(\delta t)$ globally. Therefore, if δt is large then the error of implicit Euler is large. But the scheme will never be unstable. This unconditional stability is the prime reason for using an implicit scheme.

3.2.4 Consistency and convergence

Consistency of a discretization scheme requires that the error ϵ between the discretized time derivative and the true time derivative reduces to zero for all times as δt is reduced to zero. For accuracy of order p, this means that the error has to scale with δt as:

$$\epsilon(t) = C(t)\delta t^p + O(\delta t^{p+1}), \tag{3.52}$$

where $C(t)$ is a bounded time-dependent pre-factor. This is the truncation error of the discretization scheme. As we have already derived above, the error $\epsilon(t)$ for explicit Euler is

$$\begin{aligned} \epsilon(t) &= \frac{\delta t}{2!}\frac{\mathrm{d}^2\vec{s}}{\mathrm{d}t^2}(t) + \frac{\delta t^2}{3!}\frac{\mathrm{d}^3\vec{s}}{\mathrm{d}t^3}(t) + \dots && (3.53)\\ &= C(t)\,\delta t + O(\delta t^2), && (3.54)\end{aligned}$$

where $C(t) = \frac{1}{2}\frac{\mathrm{d}^2\vec{s}}{\mathrm{d}t^2}(t)$. The explicit Euler scheme is therefore first-order accurate and consistent. Note that we have already arrived at the same order of accuracy from the global error of explicit Euler. Similarly, we can show that the other discretization schemes presented in this chapter are consistent with a certain order of accuracy. As we have seen above, leapfrog is second-order accurate, RK4 is fourth-accurate, and implicit Euler is first-order accurate.

Convergence of order p requires that the error between the numerical solution at all times and the exact solution goes to zero as δt goes to zero as:

$$error(t) = C(t)\delta t^p + O(\delta t^{p+1}),$$

where $C(t)$ is now a different pre-factor.

The *Lax equivalence theorem* combines the notions of consistency, stability, and convergence. It states that given a properly posed initial condition and a discretization scheme that is consistent, stability is a necessary and sufficient condition for convergence. Additionally, if a numerical scheme is convergent, then the order of convergence is equal to the order of accuracy of the discretization scheme.

Chapter 4

Particle Methods for Item-based Simulations

In this chapter:

- Item-based particle simulation of discrete models
- Random number generation for stochastic item-based simulations
- Stochastic agent-based simulations
- Stochastic discrete-space simulation of chemical kinetics
- Verlet time-stepping for deterministic item-based simulations
- The symplectic velocity-Verlet integrator
- Discrete element simulations of granular flow
- Lennard-Jones molecular dynamics simulations

Learning goals:

- Know the difference between aleatory and epistemic randomness in a simulation
- Be able to sample pseudo-random numbers from a given probability distribution
- Be able to implement and use stochastic item-based simulations in continuous and discrete spaces
- Know what a symplectic integrator is
- Be able to implement the Leapfrog and velocity-Verlet integrators for item-based simulations

- Be able to implement and use deterministic item-based simulations in continuous and discrete spaces

In item-based simulations, the modeled real-world entities are individually represented as discrete items. Examples include traffic simulations where individual road agents (cars, bikes, etc.) are represented, or molecular dynamics simulations where individual atoms are represented.
Using particle methods for item-based simulations is straightforward: every item is represented by a particle. The particles then evolve according to the interactions between the items. These interactions directly define the kernel $\vec{K}$ and can be of any type. They need not be mathematical equations, but can also be rule systems, subroutines, or data files. They may be deterministic or stochastic. In the deterministic case, a particle will always do the same thing in the same situation. In the stochastic case, the interactions contain randomness. We illustrate these cases using concrete examples in both continuous and discrete spaces in this chapter.
The distinction between item-based and field-based simulations may not always be obvious without knowing the model that is being simulated. Consider the example of simulating a flexible polymer represented by multiple particles that are connected by springs. This can be either an field-based or an item-based simulation, depending on the underlying model. If the underlying model is a continuum-mechanics description of the filament, then the simulation is field-based, as the particles discretize the continuous model. The spring characteristics in this case are chosen (in fact, need to be chosen) so as to correctly converge to the continuum model when the number of particles is increased. If, however, the underlying model is a filament consisting of solid rods that are connected by flexible linkers, then the simulation is item-based, as each solid rod is represented by a particle.

4.1 Stochastic dynamics

In stochastic item-based simulations, the items interact non-deterministically. This means that the result of an interaction can not be predicted with certainty, but is a realization of a random process. Therefore, only the probability of a certain outcome can be quantified.
Stochastic models can describe systems that are indeed stochastic also in the real world. Examples include the quantum mechanics of electrons, the time between two radioactive decays, and the chemical reaction between molecules that meet in space. This type of "real" stochasticity is called *aleatory randomness* (from latin *alea* = dice) and naturally results in stochastic particle interactions.
Stochastic models can also describe systems that are deterministic in reality, but too complex for us to fully model them. In this case, the stochasticity models "absence of knowledge" or "uncertainty". Examples include the weather forecast providing a probability for rain because there are not enough measurements available to fully determine the atmospheric physics. Another example is individual-based population dynamics where particle representing animals

stochastically decide to breed or eat when they meet. In reality, this process is deterministic, but the ideas, psyche, and mental processes that lead to the decision are unknown to us. This type of stochasticity in models is called *epistemic randomness* (from greek $\varepsilon\pi\iota\sigma\tau\eta\mu\eta$, *episteme* = knowledge).

4.1.1 Random number generation

Stochastic simulations require generating random numbers. However, deterministic computers cannot produce truly random numbers, which is why *random number generators* (RNG) produce streams of pseudo-random or quasi-random numbers [14]. This means that the number sequence is deterministic and reproducible, but depends on a *seed* in a non-trivial way. For different seeds, different number sequences are generated, which are indistinguishable from random by a statistical test. Most programming languages and libraries provide RNGs for the uniform probability distribution between 0 and 1.
If random numbers from a different distribution are required, they have to be computed from the uniform ones by a transformation.
The classic transformation to generate standard-normal pseudo-random numbers is the *Box-Muller transform* [15]. It takes two pseudo-random numbers from a uniform RNG over the unit interval $(0, 1]$ and computes two independent pseudo-random numbers from the standard normal distribution, that is the Gaussian distribution with mean 0 and variance 1. The method is based on the Euler formula for polar coordinates in the complex plane $\mathrm{e}^{iz} = \cos(z) + i\sin(z)$ when uniformly sampling Cartesian complex numbers in the unit square. This is illustrated in Fig. 4.1. Taking two independent uniformly distributed random numbers r_1, $r_2 \in \mathcal{U}(0, 1]$, the transform generates two standard normal variates z_1, $z_2 \in \mathcal{N}(0, 1)$ as:

$$z_1 = \sqrt{-2\ln r_1}\cos(2\pi r_2) \tag{4.1}$$

$$z_2 = \sqrt{-2\ln r_1}\sin(2\pi r_2). \tag{4.2}$$

The standard normal variates can then be scaled and shifted to any desired mean μ and standard deviation σ as: $\sigma z + \mu$.
The smallest 32-bit floating-point non-zero number r that can be represented is 2^{-32}. This maps to a random number z of 6.66. The Box-Muller transform will hence never return samples more than 6.66 standard deviations from the mean. This means that a probability mass of about 10^{-11} is lost due to numerical truncation.
The Box-Muller transform is a special case (for the normal distribution) of the general *inversion method*. The inversion method provides a general algorithm for generating pseudo-random variates from any distribution with known cumulative distribution function (CDF). The CDF of a probability distribution is:

$$\mathrm{CDF}(x) = \int_{-\infty}^{x} \mathrm{PDF}(y)\,\mathrm{d}y \tag{4.3}$$

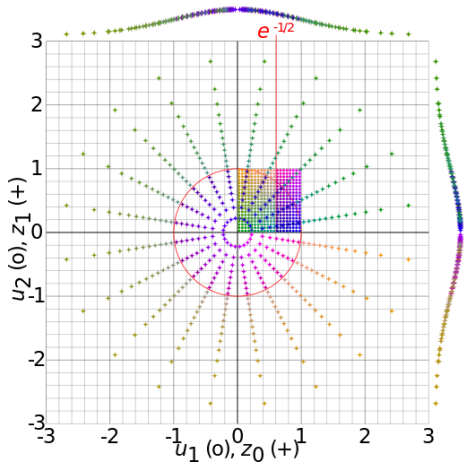


Figure 4.1: Visualization of the Box-Muller transform. The colored points in the unit square are uniformly distributed random numbers between 0 and 1 (r_1, r_2), (circles). They are mapped to a 2D Gaussian (z_1, z_2), drawn as crosses. The plots at the margins are the probability distribution functions of z_1 and z_2. Note that z_1 and z_2 are unbounded, but appear to be in $[-3, 3]$ due to the choice of the illustrated points. (Figure source: wikipedia, CC license)

and is a strictly monotone function with values in $(0, 1]$. Because the value of the CDF is always in $(0, 1]$, $Y = \mathrm{CDF}(X) \in \mathcal{U}(0, 1]$ under the CDF of X. Inverting the CDF of X hence provides a way of transforming uniform random variates into pseudo-random numbers that are distributed with the given CDF. The algorithm is:

```
! generate uniform random number over unit interval
r = uniform_RNG(0,1)
! invert the CDF analytically or numerically
compute x such that CDF(x)=r
```

Listing 4.1: Inversion method for RNG

For simple distributions, the CDF can be analytically inverted. For example, exponentially distributed random numbers with parameter λ can be computed as $x = -(1/\lambda)\ln(r)$, where ln is the natural logarithm. This is the analytical inverse of the CDF of the exponential distribution, which is $1 - \mathrm{e}^{-\lambda x}$. Solving for x yields the inverse $-(1/\lambda)\ln(1-r)$. But since $r \in \mathcal{U}(0, 1]$, also $(1-r) \in \mathcal{U}(0, 1]$. For distributions where the CDF is not analytically computable or invertible, the inversion can be done numerically, e.g., by line search or *regula falsi*.

4.1.2 Example: Agent-based ecosystem simulation

When simulating the spatiotemporal dynamics of ecosystems, individual animals are often represented as *agents* that can move around the habitat and

perform certain actions. In the absence of deeper knowledge about the decision processes of the animals, these actions are often modeled stochastically, which is an example of epistemic stochasticity.
In this case, each particles models an animal agent. The particle position is the position of the agent in the habitat. Properties may include the species, age, or weight of an animal. Let's consider a simple example: a 2D continuous-space square habitat $\Omega \in [-1, 1]^2 \subset \mathbb{R}^2$ with discrete time. In this habitat, we want to simulate the classic predator-prey model of Lotka-Volterra. The simulation starts from 100 particles distributed in the domain uniformly at random. A random 10% of them are predatory, the remaining 90% are prey.
Prey behave according to the following rules:

- They perform a random walk in the domain with diffusion constant D_{prey}.
- They replicate with probability rate ρ_{prey}.

Predators behave according to the following rules:

- They perform a random walk in the domain with diffusion constant D_{predator}.
- They eat prey they encounter, with probability α.
- They die if they do not eat for a time $> T$.
- They replicate with probability ρ_{predator} every time they ate.

This model is admittedly artificial, as for example prey would live forever if it does not get eaten. Moreover, animals can spontaneously replicate without needing a partner. Nevertheless, it is an instructive example, as it corresponds to a spatial variant of the classical Lotka-Volterra model, which is very well studied. It is a non-linear model (due to the predator-prey interactions) that can show a number of different behaviors depending on how the parameters are chosen: extinction of one or both species, spatial waves of population density, temporal oscillations of population sizes, or stable steady states. An agent-based stochastic simulation of this model has been implemented by Hiroki Sayama and is available on the web as part of the free Wolfram Demonstrations Project[1].
Particles have the following properties with the corresponding initializations:

```
properties = STRUCT{
    constant :: type = [predator, prey]
    integer :: dinnertime = 1   ! time steps since last eaten
    boolean :: isDead = false   ! dead or alive
}
```

Listing 4.2: Particle properties structure of each agent

This is a discrete-time model. The `interact()` method shown below is only called for the predators, interacting with prey within a cutoff radius r_c ("radius of encounter"). This can efficiently be done using cell lists for the prey. An encounter/interaction happens whenever a predatory and a prey get closer to

[1] http://demonstrations.wolfram.com/PredatorPreyEcosystemARealTimeAgentBasedSimulation/

each other than a small cutoff radius r_c. Then, an interaction is done according to the following method:

```
method interact(q):   ! called for all predators
    if q.properties.type == prey     ! predators only interact with prey
        r = uniform_RNG(0,1)
        if r < α ! check for eating event
            ! this eats q
            this.properties.dinnertime = 0
            q.properties.isDead = true
        end
    end
end
```

Listing 4.3: Predator-prey interaction method

The interactions are symmetric, since a predator eating a prey also directly sets the prey to dead. The `evolve()` method is always called for all particles, as the simulation proceeds in discrete time steps $1, 2, 3, \ldots$. The interaction only determines state changes, but does not apply them. All state changes are applied in the `evolve()` method:

```
method evolve():
    if this.properties.type == prey
        if this.properties.isDead == true
            delete this
            return
        end
        ! check for replication event
        r = uniform_RNG(0,1)
        if r < ρ_prey
            ! replicate
            new PARTICLE p
            p.position = this.position
            p.properties.type = prey
        end
        ! move with random walk
        this.position += [Gauss_RNG(0,2D_prey),Gauss_RNG(0,2D_prey)]
    end
    if this.properties.type == predator
        if this.properties.dinnertime > T
            ! die of hunger
            delete this
            return
        end
        if this.properties.dinnertime = 0
            ! if just eaten, may proliferate
            r = uniform_RNG(0,1)
            if r < ρ_predator ! check for replication event
                ! replicate
                new PARTICLE p
                p.position = this.position
                p.properties.type = predator
            end
        end
        ! move with random walk
        this.position += [Gauss_RNG(0,2D_predator),Gauss_RNG(0,2D_predator)]
```

```
        this.properties.dinnertime += 1
    end
```

Listing 4.4: Agent evolution method

This provides an interesting ecosystem with some of the behaviors shown in Fig. 4.2. Note that in order to simulate the item-based model, we do not need to known the resulting macroscopic field-based equations (in this case the Lotka-Volterra equations). Moreover, it is very easy in such a simulation to try the effect of different interaction rules and influences.

4.1.3 Example: Stochastic chemical kinetics

Consider a chemical reaction network, such as a cell signaling pathway or a gene regulatory network. There is no spatial information in the network model. All we know is which chemical can engage in reactions with each other, and how many molecules are in the system of each chemical. The model does not care about where they are in space, which is a good assumption when the system is small enough to be well-mixed, i.e., diffusion is so fast that every molecule has equal probability of being found anywhere in space.
Chemical reactions are truly random. Molecules undergo thermal motion and randomly bump into each other. A collision randomly leads to a reaction following the laws of statistical mechanics, which depend, e.g., on the energy and the orientation of the collision. Chemical reaction networks are hence most accurately modeled as stochastic processes in the aleatory sense of randomness.
Chemical reactions convert molecules of one species into molecules of another species. The amount of each species present in the system is given by the number of molecules of that species. Since molecule numbers are always integers, this defines a discrete space. If there are S different chemical species in the reaction system, the space is S-dimensional. Since molecule numbers also cannot be negative, the space is $\vec{x}_p \in \mathbb{Z}_0^{S+}$. This is an example where the space in which the particle positions live is not the physical, geometric space, but is the state space of the molecular reaction network.
A particle hence represents a state of the reaction system, and its motion is due to chemical reactions converting molecules of one species into another. At each reaction event, the particle makes a discrete jump in the species lattice. Since chemical reactions are stochastic, the set of particles $\{\vec{x}_p\}_{p=1}^N$ is a sample over possible states of the system. The density of particles in state space is proportional to the probability that the real system is in that state at that time. Since individual realizations of the random process are uncorrelated, the dynamics of all particles are independent. Every particle thus only interacts with itself, and there is no need for cell lists or Verlet lists to find nearby interaction partners.
Particle density in space and time is governed by the chemical master equation (CME), which is derived from conservation of probability and tells how the state-space probability distribution evolves. The set of particles hence is a discrete and finite sample from the solution of the CME. When using exact

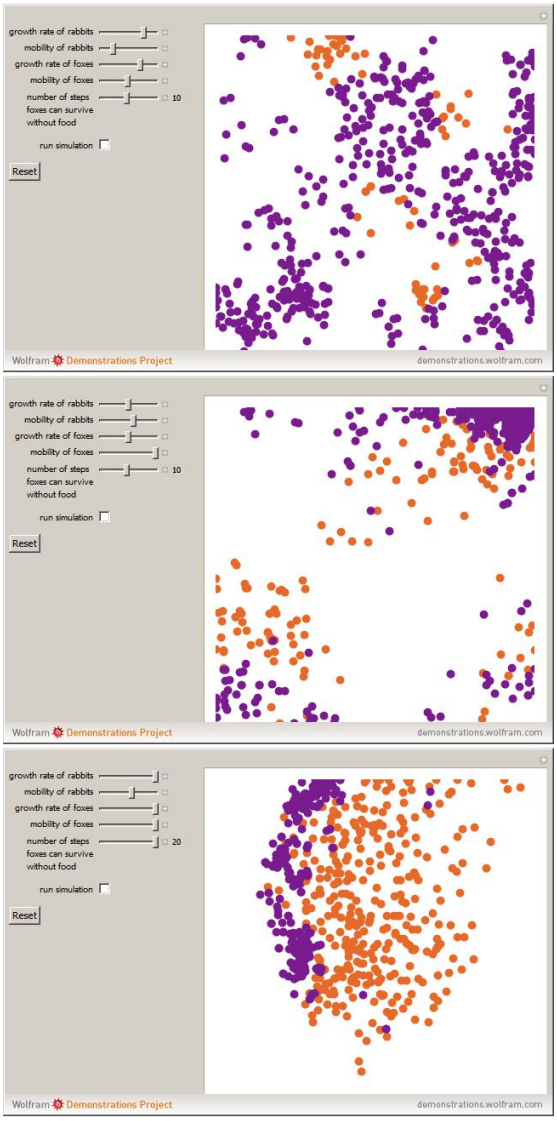


Figure 4.2: Visualization snapshots of the ecosystem simulation as implemented by Hiroki Sayama on Wolfram Demonstrations Project. Predators are shown in orange, prey in purple. The interaction cutoff radius r_c is shown as the radius of the individual disks. Top: example with parameters leading to extinction of predators. Middle: example leading to extinction of prey. Bottom: example showing traveling waves of predator density chasing prey density. (Figure source: http://demonstrations.wolfram.com/PredatorPreyEcosystemARealTimeAgentBasedSimulation/)

stochastic simulation algorithms (SSA) [16, 17], the sample is drawn from the exact, analytical solution of the CME, albeit without explicitly computing it. This is fundamental, since the solution of the CME is analytically intractable for all but the simplest cases.
Consider a chemical network with S species $s = 1, \ldots, S$ and R reactions $\mu = 1, \ldots, R$. In SSA, each reaction μ is characterized by its *propensity* $a_\mu(\vec{x}_p)$, which is the rate at which reaction μ happens (i.e., $a_\mu \mathrm{d}t$ is the probability of reaction μ happening in the next infinitesimal time interval $\mathrm{d}t$). The probability distribution of reaction events μ and waiting times between reactions τ is [16]:

$$p(\tau, \mu | \vec{x}_p(t)) = a_\mu \mathrm{e}^{a\tau} \,, \tag{4.4}$$

where $a = \sum_\mu a_\mu$ is the total propensity in the system. The reaction propensities a_μ are functions of the population of the involved species, i.e., they are functions of $\vec{x}_p$.
Gillespie's original *Direct Method* SSA samples from the above probability distribution in two steps: First, the index of the next reaction is sampled proportionally to the reaction propensities as:

$$\mu = \arg\min_{\mu'} \left[r_1 a < \sum_{m=1}^{\mu'} a_m \right] . \tag{4.5}$$

In words, the next reaction to happen is the smallest index μ' for which the cumulative sum of propensities exceeds the threshold $r_1 a$ for the first time, where $r_1 \in \mathcal{U}(0, 1]$ is a uniformly distributed random number between 0 and 1. This is easily understood by imagining the interval $[0, a]$ subdivided into R sub-intervals such that the μ-th sub-interval has exactly length a_μ. Intervals exclude the lower boundary and include the upper boundary. If one then draws a uniform random number between 0 and a (which is what $r_1 a$ is), the probability for this number to be in interval μ is proportional to a_μ. Hence, this sampling chooses reactions at rates given by their respective propensities.
In the second step of the algorithm, the expected waiting time until the chosen reaction happens is computed. From statistical physics, it is known that the waiting times between events that occur with a given frequency is exponentially distributed. The propensity of *any* reaction happening is a (we know from before that it will be reaction μ). The waiting time hence is:

$$\tau = -\frac{1}{a} \ln r_2 \,, \tag{4.6}$$

where ln is the natural logarithm, and $r_2 \in \mathcal{U}(0, 1]$ is a second uniformly distributed random number between 0 and 1. This amounts to sampling τ from the exponential distribution with parameter a using the inversion method.
These two steps are done independently for each particle. Then, all particles move according to the change in the molecular population $\vec{x}_p$ and they update their time as $t_p = t_p + \tau$. This is also an example where time is a particle property, since each particle has its own clock. In this example, the time is the

only property of a particle. The simulation ends when all particles have passed a certain final time T. Since the particles do not move between reaction events, $\vec{x}_p$ is a piecewise constant function of time. The cloud of particles $\{\vec{x}_p\}_{p=1}^N$, i.e. their density, hence provides a numerical approximation to the exact solution of the CME at any time. The complete algorithm is:

```
foreach particle p with p.properties.time< T do
    p.interact(p)
    p.evolve()
end
```

Listing 4.5: Particle SSA simulation

Each particle only interacts with itself, as we are simulating independent realizations of the random process:

```
method interact(q)
    sample μ from Eq. 4.5
    sample τ from Eq. 4.6
    compute Δx_p    ! consumed and produced molecules
    this.positionChange = Δx_p
    this.propertiesChange.time = τ
```

Listing 4.6: Particle SSA interaction

No neighbor lists are required. The `evolve()` method in this discrete-time simulation simply is:

```
method evolve():
    this.position += this.positionChange
    this.properties.time += this.propertiesChange.time
```

Listing 4.7: Particle SSA evolution

4.2 Deterministic dynamics

In deterministic item-based dynamics, the particle velocities or the forces acting on the particles are given by the interaction kernels, i.e., by particle-particle interactions. These interactions can be contact-less, as in molecular dynamics where they are mediated by molecular force fields. They can also be mediated by direct particle-particle contacts or collisions, as in granular flows. In either case, a time-stepping scheme is used to advance the particles (see Chapter 3). Ideally, the time-stepping scheme preserves the physical properties of the discrete model, such as time-reversibility (thermodynamic equilibrium) and conservation of energy. This is guaranteed by so-called *symplectic integrators*. Since the particle-particle interactions are typically the computationally most expensive part, they should be kept to a minimum. Multi-stage time-stepping schemes, such as Runge-Kutta, are hence not often used for deterministic item-based particle simulations.

4.2.1 The Verlet time-stepping method

The most popular symplectic time-stepping scheme for item-based particle simulations is the Verlet method, which is a special case of the leapfrog scheme introduced in Section 3.2.1.2. The method was first used in 1791 by French astronomer Jean Baptiste Joseph Delambre to compute the motion of planets. Carl Størmer used it in 1907 to approximate the motion of charged particles in an electric field. It was popularized in 1967 by Loup Verlet when he used it for molecular-dynamics simulations of Lennard-Jones fluids [12].
The Verlet method integrates the ODE:

$$\frac{\mathrm{d}^2\vec{x}_p}{\mathrm{d}t^2} = \vec{a}_p(t) \tag{4.7}$$

for the particle positions $\vec{x}_p$ as they move under the mechanical force $\vec{F}_p = m_p\vec{a}_p$ acting on particle p with mass m_p and acceleration $\vec{a}_p$. The Verlet scheme then reads:

$$\vec{x}_p(t_{n+1}) = 2\vec{x}_p(t_n) - \vec{x}_p(t_{n-1}) + \delta t^2\vec{a}_p(t_n). \tag{4.8}$$

It is derived by adding the Taylor expansions of $\vec{x}_p(t)$ around the two time points t_{n+1} and t_{n-1} as follows:

$$\begin{aligned} & \vec{x}_p(t_{n+1} = t_n + \delta t) = \vec{x}_p(t_n) + \delta t\vec{v}_p + \frac{1}{2}\delta t^2\vec{a}_p + \frac{1}{6}\delta t^3\frac{\mathrm{d}^3\vec{x}_p}{\mathrm{d}t^3} + \dots \\ + \quad & \vec{x}_p(t_{n-1} = t_n - \delta t) = \vec{x}_p(t_n) - \delta t\vec{v}_p + \frac{1}{2}\delta t^2\vec{a}_p - \frac{1}{6}\delta t^3\frac{\mathrm{d}^3\vec{x}_p}{\mathrm{d}t^3} + \dots \end{aligned}$$

Adding these two series yields Eq. 4.8 up to a local error of $O(\delta t^4)$. In order to compute the global error, we also need to take into account that the acceleration $\vec{a}_p$ is not known exactly but is itself computed from the approximate particle positions. A reasonably involved error consideration (see e.g., wikipedia page for "Verlet integration") shows that this leads to a global error of $O(\delta t^2)$. One order is lost due to the fact that reaching final time T requires $O(\delta t^{-1})$ time steps, another order is lost due to the accelerations being approximate themselves. Adding or subtracting Taylor series around different points in order to derive a higher-order scheme is a common technique in scientific computing, known as *Richardson extrapolation*.
One problem is that the particle velocity $\vec{v}_p$ cancels out when adding the two Taylor series. The Verlet time-stepping scheme directly updates the positions based on the acceleration or force acting on the particles. This may be undesirable, since the velocity could be an important property that the particles carry, e.g., in order to compute kinetic energy. The usual remedy is to compute velocity as a centered finite difference from the positions as:

$$\vec{v}_p(t_n) = \frac{\vec{x}_p(t_{n+1}) - \vec{x}_p(t_{n-1})}{2\delta t}, \tag{4.9}$$

which is also second-order accurate. The overall accuracy of the method is hence second order.

The Verlet time-stepping method has the following advantages and disadvantages:

+ centered around t_n and hence time-reversible

+ positions are directly moved from accelerations or forces in one step

+ conservation of momentum and energy (the scheme is symplectic)

— velocity is not directly available and needs to be approximated separately

— the scheme is prone to numerical extinction because it involves adding a very small number ($\delta t^2 \vec{a}_p$ is very small for small δt) into a much larger one. This severely limits how small one can choose δt, because δt^2 must be larger than machine epsilon for the method to advance at all.

These drawbacks led to the development of two modifications: the leapfrog scheme and the velocity-Verlet scheme, as discussed next.

4.2.2 Leapfrog time-stepping for deterministic item dynamics

The *leapfrog scheme* has already been introduced in Section 3.2.1.2. It is algebraically equivalent to the Verlet method, but is not affected by numerical extinction problems. The price one pays for this is that the velocity and position values are not available at the same time points, but at staggered, shifted times. As discussed earlier, the Leapfrog scheme is only stable for purely oscillatory linear systems. Here, we are solving Eq. 4.7. For linear stability analysis, we have $\vec{a}(\vec{x}) = \lambda \vec{x}$. The ODE then describes an undamped oscillator and hence is purely oscillatory. The leapfrog method is hence stable (and symplectic) if the time step is chosen such that $|\delta t \lambda_i| < 1$, which can always be achieved. As we know, stability in the linear case implies stability also for nonlinear right-hand sides. The leapfrog scheme is hence a good choice for item-based particle simulations.
For item-based deterministic particle dynamics, where the particle property is $\vec{\omega}_p = \vec{v}_p$, the leapfrog scheme reads:

$$\begin{aligned} \vec{v}_p(t_{n+1/2}) &= \vec{v}_p(t_{n-1/2}) + \delta t \vec{a}_p(t_n) \\ \vec{x}_p(t_{n+1}) &= \vec{x}_p(t_n) + \delta t \vec{v}_p(t_{n+1/2}), \end{aligned} \tag{4.10}$$

where the positions of the particles are available at integer time steps, but the particle velocities are available only at the half-steps $t_{n\pm1/2} = t \pm \frac{1}{2}\delta t$ (hence the name "leapfrog"). The particles carry these half-step velocities as one of their properties. The time-staggering can be a problem when the velocities and positions are needed at coinciding time points, for example to compute the kinetic and potential energies of the system at identical times in order to check the evolution of the total energy.

This is identical to the leapfrog scheme as we derived it by Taylor expansion in Section 3.2.1.2 when halving the time step size. Indeed, replacing $\delta t \to \frac{1}{2}\delta t$ in Eq. 3.13 leads to:

$$\begin{aligned}\vec{v}_p(t_{n+1/2}) &= \vec{v}_p(t_{n-1/2}) + \delta t \vec{a}_p(t_n) \\ \vec{x}_p(t_{n+1/2}) &= \vec{x}_p(t_{n-1/2}) + \delta t \vec{v}_p(t_n),\end{aligned} \tag{4.11}$$

which becomes identical to Eq. 4.10 when shifting the index in the second line by $+\frac{1}{2}$.
It is also easy to see that Leapfrog is identical to the Verlet method in Eq. 4.8: Substituting the first line of Eq. 4.10 into the second gives:

$$\begin{aligned}\vec{x}_p(t_{n+1}) &= \vec{x}_p(t_n) + \delta t(\vec{v}_p(t_{n-1/2}) + \delta t \vec{a}_p(t_n)) \\ &= \vec{x}_p(t_n) + \delta t \vec{v}_p(t_{n-1/2}) + \delta t^2 \vec{a}_p(t_n) \\ &= 2\vec{x}_p(t_n) - \vec{x}_p(t_{n-1}) + \delta t^2 \vec{a}_p(t_n),\end{aligned}$$

where we have used in the last step: $\delta t \vec{v}_p(t_{n-1/2}) = \vec{x}_p(t_n) - \vec{x}_p(t_{n-1})$ according to the second line of Eq. 4.10 with the index shifted by -1.

4.2.3 The velocity-Verlet time-stepping method

The *velocity-Verlet scheme* is a variant of the leapfrog and the Verlet methods that computes positions and velocities at coinciding time points, hence avoiding the main drawback of the leapfrog scheme. It is also algebraically equivalent to the Verlet method and hence symplectic and time-reversible. However, it is only applicable if the acceleration or force depends only on the particle position, and not on the velocity. This is typically the case in friction-less systems. If the acceleration also has a velocity-dependent component, e.g. friction, the scheme is not explicit any more. The velocity-Verlet scheme reads:

$$\begin{aligned}\vec{x}_p(t_{n+1}) &= \vec{x}_p(t_n) + \delta t \vec{v}_p(t_n) + \frac{1}{2}\delta t^2 \vec{a}_p(t_n) \\ \vec{v}_p(t_{n+1}) &= \vec{v}_p(t_n) + \frac{1}{2}\delta t \left[\vec{a}_p(t_n) + \vec{a}_p(t_{n+1})\right] .\end{aligned} \tag{4.12}$$

Again, it is easy to see that this is identical to the Verlet method in Eq. 4.8: Shifting the index of the second line of Eq. 4.12 by -1, and substituting the resulting expression for $\vec{v}_p(t_n)$ into the first line, we find:

$$\begin{aligned}\vec{x}_p(t_{n+1}) &= \vec{x}_p(t_n) + \delta t \left(\vec{v}_p(t_{n-1}) + \frac{1}{2}\delta t \left[\vec{a}_p(t_{n-1}) + \vec{a}_p(t_n)\right]\right) + \frac{1}{2}\delta t^2 \vec{a}_p(t_n) \\ &= \vec{x}_p(t_n) + \delta t \vec{v}_p(t_{n-1}) + \frac{1}{2}\delta t^2 \vec{a}_p(t_{n-1}) + \delta t^2 \vec{a}_p(t_n) \\ &= 2\vec{x}_p(t_n) - \vec{x}_p(t_{n-1}) + \delta t^2 \vec{a}_p(t_n),\end{aligned}$$

where we have in the last step used the fact that: $\delta t \vec{v}_p(t_{n-1}) + \frac{1}{2}\delta t^2 \vec{a}_p(t_{n-1}) = \vec{x}_p(t_n) - \vec{x}_p(t_{n-1})$ according to the first line of Eq. 4.12 with index shifted by -1.

In practice, the scheme is often implemented as:

$$\begin{aligned}
\vec{v}_p(t_{n+1/2}) &= \vec{v}_p(t_n) + \frac{1}{2}\delta t \vec{a}_p(t_n) \\
\vec{x}_p(t_{n+1}) &= \vec{x}_p(t_n) + \delta t \vec{v}_p(t_{n+1/2}) \\
&\text{compute } \vec{a}_p(t_{n+1}) \text{ from } \vec{x}_p(t_{n+1}) \\
\vec{v}_p(t_{n+1}) &= \vec{v}_p(t_{n+1/2}) + \frac{1}{2}\delta t \vec{a}_p(t_{n+1})\,,
\end{aligned} \tag{4.13}$$

which avoids the numerical extinction problem when computing δt^2 and saves memory by re-using the same three memory locations. The value $\vec{a}(t_{n+1})$ can also be stored and re-used in the subsequent iteration. This is identical to Eq. 4.12, as can easily be verified by substituting the expression for $\vec{v}_p(t_{n+1/2})$ from the first line into both the second and last to eliminate $\vec{v}_p(t_{n+1/2})$.

The velocity-Verlet scheme is the standard time-stepping method for item-based deterministic particle dynamics with forces that only depend on the particle positions. The scheme is symplectic and second-order accurate. If the forces also depend on the velocities, the Leapfrog scheme is the standard choice instead. We see examples of both in the reminder of this chapter.

4.2.4 Example: discrete element method for granular flows

The *discrete element method* (DEM) is a deterministic particle method for the item-based description of granular flows [18]. Granular flows are flows of granular materials, such as sand, salt, or mining stones. The collective motion of the individual granules can make the material flow, like sand dunes migrate in the wind, snow avalanches flow down the slope, and sand drizzles down an hourglass [19]. The governing equations for the macroscopic flow fields (i.e., the velocity, density, and pressure fields) are unknown and pose a famous open problem in physics [20, 21, 22, 23]. Besides experiments, item-based particle simulations are the prevalent tool of study for such systems.

In discrete element methods, every grain of the granular material is explicitly represented as a particle. Each particle has a position $\vec{x}_p(t)$ and carries the properties: velocity $\vec{v}_p$(t), angular velocity $\vec{\omega}_p$ (t), elastic deformation $\vec{u}_p(t)$, radius R_p of the grain it represents, mass m_p of the grain, polar moment of inertia I_p of the grain. Particles interact with each other in direct-contact collisions. These collisions are classically modeled according to [24] with the correction from [25]. The collision force has two components: a radial component due to elastic deformation of the colliding grains, and a tangential component due to friction between the colliding grains. The radial, elastic deformation is given by:

$$d_{pq} = (R_p + R_q) - |\vec{x}_p - \vec{x}_q|\,. \tag{4.14}$$

The radial and tangential components of the relative velocity between the two

particles at the point of collision are:

$$\vec{v}^r_{pq} = ((\vec{v}_p - \vec{v}_q) \cdot \vec{n}_{pq})\vec{n}_{pq} \tag{4.15}$$

$$\vec{v}^t_{pq} = \vec{v}_p - \vec{v}_q - \vec{v}^r_{pq} - (\vec{\omega}_p R_p + \vec{\omega}_q R_q) \times \vec{n}_{pq}\,, \tag{4.16}$$

where $\vec{n}_{pq} = (\vec{x}_p - \vec{x}_q)/|\vec{x}_p - \vec{x}_q|$ is the unit normal vector onto the plane of contact. The evolution of the elastic tangential deformation is integrated over the duration of the contact as:

$$\frac{\mathrm{d}\vec{u}_{pq}}{\mathrm{d}t} = \vec{v}^t_{pq} \tag{4.17}$$

with initial condition $\vec{u}_{pq}(t=0) = 0$ at the time of first contact. The radial and tangential forces acting on the colliding particles then are:

$$\vec{F}^r_{pq} = \sqrt{\frac{d_{pq}}{R_p + R_q}}\left(k_r d_{pq}\vec{n}_{pq} - \gamma_r m_{\mathrm{eff}}\vec{v}^r_{pq}\right) \tag{4.18}$$

$$\vec{F}^t_{pq} = \sqrt{\frac{d_{pq}}{R_p + R_q}}\left(-k_t\vec{u}_{pq} - \gamma_t m_{\mathrm{eff}}\vec{v}^t_{pq}\right)\,, \tag{4.19}$$

where k_r and k_t are the radial and tangential elastic constants of the grains, and γ_r and γ_t the radial and tangential friction constants of the grains. The effective collision mass is $m_{\mathrm{eff}} = m_p m_q/(m_p + m_q)$.
One physical problem is that the elastic tangential deformation $\vec{u}_p$ cannot grow indefinitely, since the grains will at some point start to slide against each other with no further deformation induced. In order to model this sliding limit, the tangential displacement is truncated as given by Coulomb's law $|\vec{F}^t_{pq}| < \mu|\vec{F}^r_{pq}|$. This is simply done by rescaling the tangential force as:

$$\vec{F}^t_{pq} \leftarrow \vec{F}^t_{pq}\frac{\mu|\vec{F}^r_{pq}|}{|\vec{F}^t_{pq}|} \tag{4.20}$$

and adjusting the displacement as:

$$\vec{u}_{pq} = -\frac{1}{k_t}\left(\vec{F}^t_{pq}\sqrt{\frac{R_p + R_q}{d_{pq}}} + \gamma_t m_{\mathrm{eff}}\vec{v}^t_{pq}\right)\,. \tag{4.21}$$

The total resultant force on particle p is then computed by summing the forces from all particles q it is currently in contact with:

$$\vec{F}_p = m_p\vec{g} + \sum_q(\vec{F}^r_{pq} + \vec{F}^t_{pq})\,, \tag{4.22}$$

where $\vec{g}$ is the acceleration due to gravity or any other body force. The total torque acting on particle p is similarly computed as:

$$\vec{T}_p = -R_p\sum_q(\vec{n}_{pq} \times \vec{F}^t_{pq})\,. \tag{4.23}$$

Implementing these particle interaction laws leads to the following implementation of the `interact` method:

```
method interact(q):
    ! "this" refers to the particle of which the method is a member
    dx  = this.position - q.position
    dist = sqrt(dx · dx)   ! scalar product
    diam = this.properties.radius + q.properties.radius
    ! skip this interaction if the particles do not touch
    if (dij > diam*diam) return [0, 0]
    ! compute overlap of particles (Eq. 4.14)
    dpq = diam - dist
    ! unit normal vector of contact
    dist = 1.0/dist
    normal = dx*dist

    ! relative velocity at contact point
    vpq = this.properties.velocity - q.properties.velocity
    ! radial component of the relative velocity (Eq. 4.15)
    vpqr = vpq · normal   ! scalar product
    ! tangential component of the relative velocity (Eq. 4.16)
    a = this.properties.angularvelocity * this.properties.radius +
        q.properties.angularvelocity * q.properties.radius
    vpqt = vpq - vpqr - a × normal  ! cross product

    ! integrate elastic deformation for this contact (Eq. 4.17) using explicit Euler
    ! deformation is stored separately for each contact pair
    this.properties.deformation(q) += upq(cidx+1,ipt) + dt*vpqt

    ! radial contact force (Eq. 4.18)
    factor = sqrt(dpq/diam)
    meff = this.properties.mass * q.properties.mass/
        (this.properties.mass + q.properties.mass)
    gnm = gamma_n*meff
    Fpqr = factor*(k_n*dpq*normal - gnm*vpqr)
    ! tangential contact force (Eq. 4.19)
    gtm = gamma_t*meff
    Fpqt = factor*(-k_t*this.properties.deformation(q)-gtm*vpqt)

    ! truncate displacement to satisfy Coulomb yield criterion (Eq. 4.20)
    a = mu*mu*(Fpqr · Fpqr)   ! scalar product
    b = Fpqt · Fpqt  ! scalar product
    if b>a
        a = sqrt(a/b)
        ! truncate the force magnitude accordingly
        Fpqt = Fpqt * a
        ! compute the corresponding u_pq from Eq. 4.21
        factor = 1.0/factor
        this.properties.deformation(q) = -k_tinv*(factor*Fpqt + gtm*vpqt)
    end

    ! total force on the particle (Eq. 4.22)
    kx = Fpqr + Fpqt
    ! total torque on the particle (Eq. 4.23)
    kw = normal × Fpqr  ! cross product

    return [kx, kw]
```

Listing 4.8: Interaction method for DEM

One may then use any of the algorithms in Chapter 2 to compute all pair-

wise particle interactions within a cutoff radius of $r_c = 2 \max_p R_p$, which are all particles that a particle can possibly collide with. This yields the final `propertiesChange.velocity`$= \Delta\vec{x}_p = \sum$`kx` and `propertiesChange.angularvelocity`$= \Delta\vec{\omega}_p = \sum$`kw`, i.e., the total force and torque acting on each particle. In order to include the effect of gravity, however, the force should not be initialized to zero, but to $m_p\vec{g}$ for each particle before starting the interaction loop.
Using the leapfrog scheme for time-stepping, the particle positions and properties are updated as:

$$\vec{\omega}_p(t_{n+1}) = \vec{\omega}_p(t_n) + \frac{\delta t}{I_p}\vec{T}_p(t_n) \tag{4.24}$$

$$\vec{v}_p(t_{n+1/2}) = \vec{v}_p(t_{n-1/2}) + \frac{\delta t}{m_p}\vec{F}_p(t_n) \tag{4.25}$$

$$\vec{x}_p(t_{n+1}) = \vec{x}_p(t_n) + \delta t\vec{v}_p(t_{n+1/2}) \tag{4.26}$$

The angular velocity $\vec{\omega}$ is integrated using the explicit Euler scheme. In this model, the radius R_p, mass m_p, and polar moment of inertia I_p of all particles remain constant throughout the simulation. Note that we cannot use velocity-Verlet time-stepping for this simulation, because the friction makes the acceleration depend on the velocity.
In the practical implementation, the particles store the half-step velocities in their property variable $\vec{v}_p$. This is just naming, and the physical meaning of half time points remains. The implementation of the `evolve` method then becomes:

```
method evolve():
    this.properties.angularvelocity +=
        this.propertiesChange.angularvelocity*dt/this.properties.polarinertia
    this.properties.velocity +=
        this.propertiesChange.velocity*dt/this.properties.mass
    this.position += dt*this.properties.velocity
```

Listing 4.9: Particle evolution for DEM

Using the PPM Library [26, 27] to implement these methods on distributed-memory parallel computers, DEM simulations of sand avalanches have been done using more than a hundred million particles on 192 processors [5].
The model presented here is very general and includes many of the important physical phenomena (grains of different sizes, friction, elasticity, deformations, rotation, Hertz pressure). The main limitation is that all grains are spheres. Extending to more complex grain shapes is, however, challenging. Already for ellipsoids in 3D there is no analytical formula for collision detection known (for ellipses in 2D there is one, though). One then has to resort to representing the surfaces of the grains using triangulations or level sets, and detecting collisions numerically. This, however, renders the simulation very time-consuming.

4.2.5 Example: Lennard-Jones molecular dynamics

Molecular dynamics [2] is an item-based model to study atomic and molecular processes in simulations. Since systems at this time and length scale are

extremely difficult to study experimentally, the molecular dynamics simulation method is fundamental to our understanding of molecular systems, which is why its inventors Martin Karplus, Michael Levitt, and Arieh Warshel were awarded to 2013 Nobel Prize in chemistry.

In molecular dynamics, atoms or molecules are explicitly represented by particles that take positions in continuous space. The continuous forces acting on them are given by atomic force fields or pairwise potentials. The atoms then move according to Newtonian mechanics. The surprising fact that this reproduces the correct physics even at those length scales is part of the magic of the molecular dynamics method.

While the basic algorithm is always the same, the force fields differ from application to application, and it is something between an art and a science to design and validate new force field models for certain molecular systems, including lipids, proteins, or DNA. The simplest historic example of a force field is the *Lennard-Jones potential* [12]. It approximates the interaction between electrically neutral inert atoms (e.g., noble gases) as:

$$U_{\mathrm{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{4.27}$$

where r is the distance between the two atoms, ε is the depth of the potential well, and σ is the distance at which the interaction potential becomes zero (see Fig. 4.3). The minimum of the potential hence has value $U_{\mathrm{LJ}}(r_m) = -\varepsilon$ and is reached at $r = r_m = 2^{1/6}\sigma$. The r^{12} term is repulsive, describing short-range Pauli exclusion due to electron orbitals starting to overlap. The r^6 term models the attraction from van-der-Waals forces. The exponents 6 and 12 are chosen for efficient computation, since the r^{12} term can be computed as the square of the r^6 term.

The force acting between two atoms that are a distance of r apart is given by the negative gradient of the interaction potential, thus:

$$\vec{F}(r) = -\nabla_r U_{\mathrm{LJ}}(r) = 24\varepsilon \left(2\sigma^{12}/r^{13} - \sigma^6/r^7\right). \tag{4.28}$$

In order to efficiently compute this term across all particle pairs, we precompute and define $s = \sigma^6$ and $\rho = r^{-2}$. Then, the above can be rewritten as:

$$\vec{F}(r) = 24\varepsilon r \left(2s^2\rho^7 - s\rho^4\right). \tag{4.29}$$

Since long-range interactions are negligibly small, we introduce a cutoff radius $r_c = 2.5\sigma$, beyond which the potential is smaller than $\varepsilon/60$. Nevertheless, truncating the potential at r_c introduces a discontinuity at $r = r_c$ where the potential jumps to zero and the force will be infinite. In order to avoid the instabilities this would introduce in the simulation, the potential is shifted upward so that it is exactly zero at $r = r_c$. This shift is inconsequential for the particle dynamics, because the force only depends on the potential gradient, which is invariant to shifts. The final, truncated and shifted potential then is:

$$U_{\mathrm{LJ,trunc}}(r) = \begin{cases} U_{\mathrm{LJ}}(r) - U_{\mathrm{LJ}}(r_c) & r \le r_c \\ 0 & r > r_c \end{cases}. \tag{4.30}$$

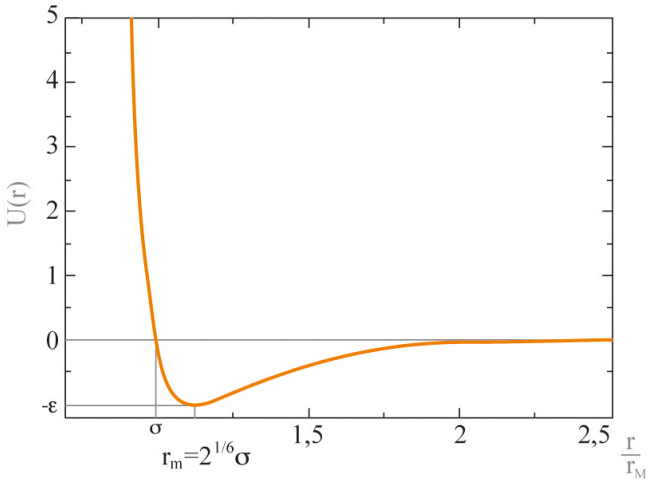


Figure 4.3: Plot of the Lennard-Jones potential function. (Figure source: thesaurus.rusnano.com)

The force is still given by Eq. 4.29 for $r \leq r_c$. For $r > r_c$, the force is zero and the interactions are not considered in the simulation. While the shifted potential has no jump at $r = r_c$, the force (i.e., its gradient) still jumps. This, however, is generally harmless. Only in special cases, like when computing gas-liquid critical points, this force discontinuity can be a problem. In this case, more elaborate smooth truncations (so-called tail corrections) are used, which, however, slightly change the model.

When simulating a Lennard-Jones fluid with particles, each particle represents one atom. The particles have positions $\vec{x}_p$ and velocity $\vec{v}_p$ as their only property. All particles are atoms of the same type and hence all have the same mass m.

The particles hence interact according to the following method:

```
method interact(q):
    r = |this.position - q.position|
    rho = 1.0/(r*r)
    s = sigma^6
    F = 2.0*s^2*rho^7 - s*rho^4
    F = 24.0*epsilon*r*F
    kx = 0          ! there is no direct velocity
    kw = F/m    ! acceleration
    return [kx,kw]
```

Listing 4.10: Particle interaction for Lennard-Jones MD

A typical sanity check for Lennard-Jones molecular dynamics simulations is to

check that the total energy

$$E_{\text{tot}} = E_{\text{kin}} + E_{\text{pot}} \tag{4.31}$$

$$E_{\text{kin}} = \frac{1}{2} m \sum_p \vec{v}_p \cdot \vec{v}_p \tag{4.32}$$

$$E_{\text{pot}} = 4\varepsilon \sum_p \sum_{q, r_{pq} < r_c} \left[\left(\frac{\sigma}{r_{pq}} \right)^{12} - \left(\frac{\sigma}{r_{pq}} \right)^{6} \right], \tag{4.33}$$

is conserved. It is therefore common practice to use a symplectic time-stepping scheme that conserves energy, such as the velocity-Verlet scheme (see Sec. 4.2.3). This can be used here because the force acting on a particle only depends on the position of the particles and not on their velocities. The `evolve` method then becomes:

```
method evolve():
    this.properties.velocity +=
        dt*0.5*this.propertiesChange.velocity
    this.properties.position += this.properties.velocity*dt
    compute new accelerations of the particles
    this.properties.velocity += 0.5*dt*this.propertiesChange.velocity
```

Listing 4.11: Particle evolution for Lennard-Jones MD

The interaction partners within the cutoff radius r_c are commonly found using Verlet lists. Verlet lists are a good choice for Lennard-Jones simulations, because the atoms do not move far, but rather jiggle around their equilibrium position. Therefore, the simulation is first run until the system has equilibrated from the initial particle placement, i.e., until the potential and kinetic energies plateau. The particles are initially commonly placed on a Cartesian lattice with a given density, defining the density of the Lennard-Jones fluid to be simulated.

Chapter 5

Discretizing Linear Differential Operators on Particles

In this chapter:

- Field-based particle simulations of continuous models
- Discretizing linear differential operators over particles
- Smooth particle function approximation
- Smooth particle hydrodynamics
- Particle strength exchange
- Discretization-corrected Particle strength exchange
- Moment conditions
- Overlap condition
- Diffusion operators

Learning goals:

- Be able to approximate a smooth function using particles
- Know SPH, PSE, and DC-PSE for discretizing linear differential operators over particle function approximations
- Know the advantages and drawbacks of these methods
- Be able to derive PSE kernels for any given linear differential operator

- Be able to implement the resulting discretization schemes
- Be able to implement the DC-PSE kernel system

Recall the equation of motion for particle p for continuous-time models introduced in Chapter 3:

$$\begin{aligned} \frac{\mathrm{d}\vec{x}_p}{\mathrm{d}t}(t) &= \vec{v}_p(t, \vec{x}(t), \vec{\omega}(t)) \\ \frac{\mathrm{d}\vec{\omega}_p}{\mathrm{d}t}(t) &= \vec{g}_p(t, \vec{x}(t), \vec{\omega}(t)). \end{aligned} \tag{5.1}$$

The velocity $\vec{v}_p$ and/or the property rate $\vec{g}_p$ in general contain spatial differential operators acting on $\vec{x}_p$ and/or $\vec{\omega}_p$. For example, one way (the Fick'ean way) of describing diffusion would be to set $\vec{v}_p = 0$ and $\vec{g}_p = D\nabla^2\vec{\omega}_p$, where D is the diffusion constant. In order to represent this on particles, the differential operators need to be discretized over the particle locations. In Chapter 3, we presented numerical methods to discretize the temporal differential operator $\frac{\mathrm{d}}{\mathrm{d}t}$. In the present Chapter, we introduce discretization schemes for spatial differential operators:

$$D^\beta = \frac{\partial^{|\beta|}}{\partial x_1^{\beta_1} \partial x_2^{\beta_2} \cdots \partial x_d^{\beta_d}}, \tag{5.2}$$

where d is the dimension of space. $\beta = (\beta_1, \beta_2, \ldots, \beta_d)$ is a multi-index such that $|\beta| = \beta_1 + \beta_2 + \ldots + \beta_d$. For example, $d = 1$ and $\beta = (1)$ is the first derivative along x, i.e., $\frac{\partial}{\partial x}$.

5.1 Smooth Particle Hydrodynamics: SPH

SPH is a particle method invented originally to solve astrophysical problems in three dimensions. The governing equations of such problems emerge from classical Newtonian hydrodynamics. Therefore, this method is referred to as *smooth particle hydrodynamics* or SPH in short. The method, however, has subsequently been used as a general method to discrete differential operators on particles.

In SPH, the process of discretizing a spatial derivative starts with the *integral representation of the function*. Any function $f(\vec{x})$ can be represented as a convolution of the function with the Dirac delta distribution:

$$f(\vec{x}) = \int_\Omega f(\vec{y})\delta(\vec{x} - \vec{y})\mathrm{d}\vec{y},$$

where Ω represents the complete domain of $f(\cdot)$, and the Dirac delta $\delta(\cdot)$ is defined as

$$\delta(\vec{x} - \vec{y}) = \begin{cases} \infty, & \text{if } \vec{x} = \vec{y} \\ 0, & \vec{x} \neq \vec{y}. \end{cases} \tag{5.3}$$

This is an identity and is exact. However, it is not useful for practical computation, since the Dirac delta is infinite and discontinuous.
Therefore, the Dirac delta function is replaced by a smooth function with a smoothing length ϵ, then the function $f(\vec{x})$ can be *approximated* by $f_\epsilon(\vec{x})$ where:

$$f(\vec{x}) \approx f_\epsilon(\vec{x}) = \int_\Omega f(\vec{y}) W_\epsilon(\vec{x} - \vec{y}) \mathrm{d}\vec{y}, \tag{5.4}$$

where $W_\epsilon(\cdot)$ is referred to as the *smoothing function*, or *smoothing kernel*, or *kernel function*, or *mollification kernel*, or simply *kernel*. The variable ϵ is the smoothing length defining the length scale of smoothing, and the approximation f_ϵ is referred to as the *mollified approximation* of the function f. For the mollified approximation f_ϵ to be a proper approximation of the function $f(\vec{x})$ according to Eq. 5.4, the kernel function must fulfill the following properties:

1. The kernel $W_\epsilon(\cdot)$ must be even:

$$W_\epsilon(\vec{z}) = W_\epsilon(-\vec{z}).$$

2. The kernel must be normalized:

$$\int_\Omega W_\epsilon(\vec{z}) \mathrm{d}\vec{z} = 1.$$

3. The kernel must converge to the Dirac delta:

$$\lim_{\epsilon \to 0} W_\epsilon(\vec{z}) = \delta(\vec{z}).$$

One standard choice for $W_\epsilon(\cdot)$ that fulfills these properties is:

$$W_\epsilon(\vec{z}) = \frac{1}{\epsilon^d} W\left(\frac{\vec{z}}{\epsilon}\right), \tag{5.5}$$

where W is any even, normalized, local function.
As mentioned earlier, f_ϵ is an approximation of the true function f. We would now like to find the approximation error. This is, however, cumbersome to derive in d dimensions. We, therefore, assume a one-dimensional domain in order to derive the approximation error. The results from this derivation, however, are also valid in higher dimensions. For a one-dimensional domain, Eq. 5.4 can be rewritten as

$$f_\epsilon(x) = \int_\Omega f(y) W_\epsilon(x - y) \mathrm{d}y,$$

Taylor-expanding $f(y)$ around x:

$$f(y) = f(x) + \frac{1}{1!} \frac{\mathrm{d}f(x)}{\mathrm{d}x}(y - x) + \frac{1}{2!} \frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}(y - x)^2 + \dots$$

and inserting this into the above mollified function approximation:

$$\begin{aligned} f_\epsilon(x) &= \int_\Omega [f(x) + \frac{1}{1!}\frac{\mathrm{d}f(x)}{\mathrm{d}x}(y-x) + \frac{1}{2!}\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}(y-x)^2 + \ldots] W_\epsilon(x-y)\mathrm{d}y \\ &= f(x)\int_\Omega W_\epsilon(x-y)\mathrm{d}y + \frac{\mathrm{d}f(x)}{\mathrm{d}x}\int_\Omega (y-x)W_\epsilon(x-y)\mathrm{d}y + \\ &\quad \frac{1}{2}\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}\int_\Omega (y-x)^2 W_\epsilon(x-y)\mathrm{d}y + \ldots \\ &= f(x)\int_\Omega W_\epsilon(x-y)\mathrm{d}y + \sum_{n=1}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega (y-x)^n W_\epsilon(x-y)\mathrm{d}y. \end{aligned}$$

Substituting in the integral $y - x = z$, we get

$$\begin{aligned} f_\epsilon(x) &= f(x)\int_\Omega W_\epsilon(-z)\mathrm{d}z + \sum_{n=1}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n W_\epsilon(-z)\mathrm{d}z \\ &= f(x)\int_\Omega W_\epsilon(z)\mathrm{d}z + \sum_{n=1}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n W_\epsilon(z)\mathrm{d}z \\ &= f(x)\int_\Omega W_\epsilon(z)\mathrm{d}z + \sum_{n=2,4,\ldots}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n W_\epsilon(z)\mathrm{d}z \\ &= f(x) + \sum_{n=2,4,\ldots}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n W_\epsilon(z)\mathrm{d}z. \end{aligned}$$

The first line in the above equation is merely a consequence of substituting $y - x = z$. The second line is a consequence of $W_\epsilon(\cdot)$ being even. The third line is a consequence of $\int_\Omega z^n W_\epsilon(z) = 0$ for odd n, for which $z^n W_\epsilon(z)$ is an odd function. The fourth line is a consequence of the normalization condition of $W_\epsilon(\cdot)$.
The approximation error between the mollified approximation f_ϵ and the true function f therefore is

$$\begin{aligned} error &= |f_\epsilon(x) - f(x)|, \\ &= \left|\sum_{n=2,4,\ldots}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n W_\epsilon(z)\mathrm{d}z\right|, \\ &= \left|\sum_{n=2,4,\ldots}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\int_\Omega z^n \frac{1}{\epsilon}W\left(\frac{z}{\epsilon}\right)\mathrm{d}z\right|, \qquad \left(\text{since } W_\epsilon(z) = \frac{1}{\epsilon}W\left(\frac{z}{\epsilon}\right)\right) \\ &= \left|\sum_{n=2,4,\ldots}^{\infty}\frac{1}{n!}\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}\epsilon^n\int_\Omega \alpha^n W(\alpha)\,\mathrm{d}\alpha\right|, \qquad \left(\text{setting } \frac{z}{\epsilon} = \alpha\right). \end{aligned}$$

Therefore, the approximation error between the mollified approximation f_ϵ and the true function f is $O(\epsilon^2)$ since $\frac{1}{2}\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}\epsilon^2\int_\Omega \alpha^2 W(\alpha)\,\mathrm{d}\alpha$ is the dominant error

term in the above expression for small ϵ. The second moment of W is just a constant. This results in a second-order approximation of the function f.
A frequently used choice for W is a Gaussian kernel:

$$W_\epsilon(z) = \frac{1}{\epsilon\sqrt{2\pi}} \mathrm{e}^{\frac{-z^2}{2\epsilon^2}}.$$

Note that the Gaussian kernel is strictly positive, i.e., the value of the Gaussian kernel is always greater than zero.
If, however, the smoothing kernel $W_\epsilon(\cdot)$ in addition also fulfills the following conditions:

$$\int_\Omega z^n W_\epsilon(z) \mathrm{d}z = 0, \qquad \text{for } n = 2, 4, \ldots r-2, \tag{5.6}$$

then the approximating error between the mollified approximation f_ϵ and the true function f becomes $O(\epsilon^r)$. It is easy to see that this can never be achieved by strictly positive, even, normalized kernels. The Gaussian kernel can hence be at most second order accurate. Kernels whose order of approximation is greater than 2 cannot be strictly positive.
The condition prescribed by Eq. 5.6 imposes that the *moments*[1] of W_ϵ be zero for the second, fourth, up to the $(r-2)^{\text{th}}$ moment. Such conditions prescribing values for the moments of a kernel are referred to as *moment conditions*.
This provides us with a smooth approximation $f_\epsilon(x)$ of the function $f(x)$ to order $O(\epsilon^r)$. However, the function is still continuous. Discretizing it over the particles starts from Eq. 5.4, where the integral is replaced by a sum. This amounts to using quadrature (numerical integration)

$$f(\vec{x}) \approx f_\epsilon(\vec{x}) = \int_\Omega f(\vec{y}) W_\epsilon(\vec{x}-\vec{y}) \mathrm{d}\vec{y} \approx \sum_p f(\vec{x}_p) W_\epsilon(\vec{x}-\vec{x}_p) V_p = f(\vec{x})^h_\epsilon. \tag{5.7}$$

This now only requires knowing the function value at the particle locations $\vec{x}_p$. However, the numerical integration introduced the particle volume V_p, which is the integration element or the weight of the quadrature scheme. This is a limitation in practice, as it has to be computed. One way is to do a Voronoi tessellation of the particles in order to find the portion of space for which any given particle is responsible. Mostly, however, particles are simply initialized on a regular Cartesian grid, which renders all volumes equal to h^d, where h is the grid spacing.
This discretization introduces another error, the *quadrature error*, which depends on the quadrature scheme used. For midpoint quadrature (i.e., the rectangular rule), we have:

$$\omega_p = f(\vec{x}_p) V_p.$$

The discretization (quadrature) error then becomes:

$$f^h_\epsilon(\vec{x}) = f_\epsilon(\vec{x}) + O\left(\frac{h}{\epsilon}\right)^s,$$

[1]The n^{th} moment of a function $f(x)$ is $\int_\Omega x^n f(x) \mathrm{d}x$.

where h is the distance between nearest-neighbor particles and s is the convergence order of the quadrature scheme used ($s = 1$ for the above midpoint quadrature). From this expression, we see that in order for the overall error to decrease for higher-order quadrature, we have to require that

$$\frac{h}{\epsilon} < 1\,.$$

This condition means that the kernel widths must be greater than the distance between nearest particles. The condition is thus frequently called *overlap condition* because it states that "particles must overlap." This makes sense because otherwise the value of the function f at off-particle locations can not be computed any more, as all information is missing there, which also no longer allows to bound the approximation error.

As opposed to simply sampling the function at the particle locations, the above formulation with the overlapping smoothing kernels has an important advantage: it is smooth. This means that it can be evaluated at any location, also between particles. Moreover, it allows computing differential operators, which require the function to have an appropriate level of smoothness.

Evaluating a differential operator is then done by exploiting its linearity:

$$D^\beta f(x) \approx D^\beta f_\epsilon^h(x) \approx \sum_p f(\vec{x}_p) D^\beta W_\epsilon(\vec{x} - \vec{x}_p) V_p\,, \tag{5.8}$$

since $f(\vec{x}_p)$ and V_p are constants with respect to $\vec{x}$. Evaluating the differential operator applied to $f(x)$ at particle p hence becomes:

$$D^\beta f_\epsilon^h(\vec{x}_p) \approx \sum_q f(\vec{x}_q) \left[D^\beta W_\epsilon\right](\vec{x}_p - \vec{x}_q) V_q = \sum_q \vec{K}(\vec{x}_p, \vec{x}_q)\,, \tag{5.9}$$

This of course only works for linear differential operators. There are also nonlinear ones, such as Schwarzian derivatives, but they are much less common in practice. SPH hence provides a simple recipe for approximating linear differential operators over functions represented on particles: use the (usually analytically known) derivative of the smoothing kernel in the function approximation. This also immediately shows that the smoothing kernel (and hence the whole function approximation) must be sufficiently smooth, as otherwise the derivative would not exist.

However, there are three drawbacks with this way of approximating differential operators in particle methods: (1) The particle interactions are not symmetric. This is because $\vec{K}(\vec{x}_p, \vec{x}_q) \neq -\vec{K}(\vec{x}_q, \vec{x}_p)$. This means that the represented quantity (e.g., mass) is not exactly conserved, and that using symmetric neighbor lists will not lead to a reduction in the number of kernel evaluations. This is akin to other asymmetric non-conservative schemes, like Fishelove's scheme [28]. (2) The approximation loses one order of accuracy with every degree of derivative. This is because of the inner derivative of the kernel $W_\epsilon = \epsilon^{-d} W(z/\epsilon)$. It's first derivative is $W'_\epsilon = \epsilon^{-(d+1)} W'(z/\epsilon)$. So, with every derivative we get an additional pre-factor of $1/\epsilon$, which cancels an order in the ϵ^r pre-factor of the

leading error order. Therefore, if the kernel W_ϵ fulfills the moment conditions to order r, the $|\beta|$-th derivative of W_ϵ will only fulfill them to order $r-|\beta|$. Note that the higher moments are not exactly zero, because of the quadrature error. For sufficiently high derivatives, this eventually leads to an approximation error that is constant or even grows with increasing particle number. The classical SPH formulation as presented here is therefore strictly-speaking inconsistent, as it does not converge for all derivatives. While various "corrected SPH" formulations attempt to alleviate this problem, we present in the next section a different approach that avoids the problem altogether. (3) The kernels in an r_c-neighborhood from a boundary are wrong, because some of the interaction partners are missing. While the operator can still be computed, the result is going to be wrong. A common remedy is to place mirror particles outside the domain (i.e., mirroring all particles in an r_c-neighborhood from the boundary at the boundary) and then computing the full interaction spheres including the mirror particles. This allows imposing homogeneous boundary conditions only. For homogeneous Neumann conditions, the mirror particles are given the same value as the respective source particle. For homogeneous Dirichlet boundary conditions, the sign is flipped. Besides its limitation to homogeneous boundary conditions, this so-called *method of images* is only first-order accurate (on general boundaries; at flat boundaries it is exact), hence reducing the convergence order of the method to 1 in the L_∞-norm.

5.2 Particle Strength Exchange (PSE)

One obvious remedy for point (2) above is to independently derive different kernels for the different derivatives, instead of using derivatives of the same kernel. This then allows to impose the moment conditions independently for each kernel, engineering all of them to the same order of accuracy. At the same time, the derivation can also be made symmetric and hence conservative. The function approximation is still done in the same way as in SPH, and in particular the overlap condition still holds. But differential operators are approximated using different, symmetric particle interactions.

One such method is Particle Strength Exchange (PSE), which has originally been developed as a deterministic pure particle method to simulate diffusion in the continuum (macroscopic) description. The method was introduced by Degond and Mas-Gallic in 1989 [29, 30, 31]. Just like SPH, it is also based on a deterministic integral approximation of the diffusion (i.e., differential) operator. Moreover, PSE also uses a smooth particle function approximation, which allows recovering the field values everywhere in space. The difference to SPH is that we are using a different kernel η^β for every differential operator β, where $\eta^\beta \neq D^\beta \eta$. Since PSE is a pure particle method, we look for an integral operator approximation with a certain kernel η. This will then lead to a particle-particle interaction scheme as outlined in Section 1.2.4.

Isotropic Homogeneous Diffusion

We start deriving PSE with the classic case of isotropic, homogeneous diffusion, where we want to approximate the Laplacian on scattered particle locations such that mass is conserved. For simplicity, we consider the 1D case. The derivation in d dimensions is analogous.
In 1D, the diffusion equation is

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}\,,$$

where D is the diffusion constant.
We start by expansion of the concentration field $u(y,t)$ into a Taylor series around point x:

$$u(y) = u(x) + (y-x)\frac{\partial u}{\partial x} + \frac{1}{2}(y-x)^2\frac{\partial^2 u}{\partial x^2} + \frac{1}{6}(y-x)^3\frac{\partial^3 u}{\partial x^3} + \dots\,. \tag{5.10}$$

We then subtract $u(x)$ on both sides, multiply with the (unknown) kernel $\eta_\epsilon = \epsilon^{-1}\eta(x/\epsilon)$ and integrate over the entire domain of solution Ω in order to arrive at an integral operator approximation:

$$\begin{aligned}\int_\Omega (u(y)-u(x))\eta_\epsilon(y-x)dy = &\int_\Omega (y-x)\frac{\partial u}{\partial x}\eta_\epsilon(y-x)dy \\ &+ \frac{1}{2}\int_\Omega (y-x)^2\frac{\partial^2 u}{\partial x^2}\eta_\epsilon(y-x)dy \\ &+ \frac{1}{6}\int_\Omega (y-x)^3\frac{\partial^3 u}{\partial x^3}\eta_\epsilon(y-x)dy + \dots\end{aligned} \tag{5.11}$$

The term we want is the $\frac{\partial^2 u}{\partial x^2}$ on the right-hand side. We thus design the kernel η such that this term is the only one remaining on the right-hand side, up to a certain order r. This requires that:

- η be even $\Leftrightarrow$ all integrals over odd powers vanish
- $\int z^2\eta(z)dz \stackrel{!}{=} 2 \Leftrightarrow$ second term becomes $\frac{\partial^2 u}{\partial x^2}\cdot\frac{1}{2}\cdot 2\cdot\epsilon^2$
- $\int z^s\eta(z)dz \stackrel{!}{=} 0 \quad \forall\, 2 < s \leq r+1 \Leftrightarrow$ higher-order terms vanish up to order $r+1$

The moment conditions are based on the change of variables $z = (y-x)/\epsilon$, hence $dy = \epsilon dz$, for which the second expansion order becomes:

$$\begin{aligned}\frac{1}{2}\int_\Omega (y-x)^2\frac{\partial^2 u}{\partial x^2}\eta_\epsilon(y-x)dy &= \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\int_\Omega z^2\epsilon^2\frac{1}{\epsilon}\eta(z)\epsilon dz \\ &= \frac{1}{2}\frac{\partial^2 u}{\partial x^2}\epsilon^2\int_\Omega z^2\eta(z)dz\,.\end{aligned} \tag{5.12}$$

Using such an η, the only terms remaining are:

$$\int_\Omega (u(y) - u(x))\eta_\epsilon(y-x)dy = \frac{\partial^2 u}{\partial x^2}\epsilon^2 + O(\epsilon^{r+2})\,. \tag{5.13}$$

The factor ϵ^2 comes from the fact that the differential operator is found with the second moment of $\eta(z)$ in the Taylor expansion. We now solve this equation for the desired term, which is the right-hand side of the diffusion equation:

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{\epsilon^2}\int_\Omega (u(y) - u(x))\eta_\epsilon(y-x)dy + O(\epsilon^r)\,. \tag{5.14}$$

$$\tag{5.15}$$

This is the integral operator approximation to the 1D diffusion operator. Any kernel function η that fulfills the above three moment conditions can be used. The next step is to **discretize** this integral operator as a quadrature over the set of N particles, thus:

$$\frac{\partial^2 u^h}{\partial x^2}(x_p^h) = \frac{1}{\epsilon^2}\sum_{q=1}^{N} V_q(u_q^h - u_p^h)\eta_\epsilon(x_q^h - x_p^h)\,. \tag{5.16}$$

If all particles have the same volume, the difference in the first parenthesis can simply be computed over the strengths $\omega_p = V_p u(x_p)$. This is the discrete form of the diffusion operator. In d dimensions, the operator looks exactly the same. Even the pre-factor ϵ^{-2} remains the same because it comes from the order of the approximated differential operator and not from the dimension. The only thing that changes is that a different η has to be used, namely one that satisfies the above moment conditions in dD, for the respective d.
The final PSE method is again formulated in terms of particles and the dynamics of their properties. Particles have positions $\vec{x}_p$ and properties (in PSE traditionally called "strengths") $\omega_p(t) = V_p u(\vec{x}_p, t)$. The particle positions and strengths then evolve according to:

$$\begin{cases} \dfrac{d\vec{x}_p}{dt} = \vec{0} \\ \dfrac{d\omega_p}{dt} = \dfrac{V_p D}{\epsilon^2}\displaystyle\sum_{q=1}^{N}(\omega_q - \omega_p)\eta_\epsilon(\vec{x}_q - \vec{x}_p)\,. \end{cases} \tag{5.17}$$

if all particles have the same volumes.
PSE has an intuitive interpretation in terms of Fick's law of diffusion. Fick's law states that in diffusion, mass is flowing against the concentration gradient. This is exactly what the PSE operator does: the first parenthesis computes the mass difference between a pair of interacting particles whereas the kernel η converts this difference into a flux of mass depending on the distance between the two particles.

Frequently used second-order accurate kernels in 1D and 3D are:

$$\eta_\epsilon(x) = \frac{1}{2\epsilon\sqrt{\pi}} e^{-\frac{x^2}{4\epsilon^2}} \qquad x \in \mathbb{R}$$
$$\eta(\vec{x}) = \frac{15}{\pi^2} \frac{1}{|\vec{x}|^{10} + 1} \qquad \vec{x} \in \mathbb{R}^3$$

The first kernel is a Gaussian, which naturally follows from the transition density (Green's function) of diffusion being a Gaussian and the differential equation (i.e., the diffusion equation) being linear (Superposition principle). The second example shows that there can also be other kernels that fulfill the moment conditions. Polynomial kernels such as the one here are computationally more efficient than Gaussians because we don't need to evaluate an exponential function. Both of these kernels have order $r = 2$ and are strictly positive. Like in SPH, strictly positive kernels cannot achieve orders $r > 2$. However, these kernels are independent of the function-approximation kernel W and they are not equal to the second derivative of W. Unlike in SPH, it is possible in PSE to, e.g., use a Gaussian kernel for W in combination with a polynomial kernel for η.

Anisotropic Inhomogeneous PSE (Optional Material)

So far we have focused on isotropic and homogeneous diffusion for simplicity. The same derivations, however, can also be made for anisotropic and inhomogeneous diffusion, where $\mathbf{D}$ is a full matrix. We then need to find an integral approximation to the operator $\nabla \cdot (\mathbf{D}\nabla)$ rather than the Laplacian Δ. In d dimensions, this leads to the integral operator approximation:

$$\nabla \cdot (\mathbf{D}\nabla u) \approx Q_\epsilon u(\vec{x}, t) = \epsilon^{-2} \textstyle\int_\Omega (u(\vec{y}) - u(\vec{x}))\sigma_\epsilon(\vec{x}, \vec{y}, t) d\vec{y} \tag{5.18}$$
$$\tag{5.19}$$

(we skip the details of the derivation because there is nothing conceptually new) and the PSE scheme remains

$$\begin{cases} \dfrac{d\omega_p}{dt} = \dfrac{V_p}{\epsilon^2} \displaystyle\sum_{q=1}^{N} (\omega_q - \omega_p)\sigma_\epsilon(\vec{x}_p, \vec{x}_q, t) \qquad \text{for } V_p = V_q \\ \dfrac{d\vec{x}_p}{dt} = \vec{0}\,. \end{cases} \tag{5.20}$$

The operator kernel σ is now a bit more complicated and has the form:

$$\sigma_\epsilon(\vec{x}_p, \vec{x}_q, t) = \underbrace{\epsilon^{-2}\overline{\eta}_\epsilon(\vec{x}_p - \vec{x}_q)}_{\text{isotropic part}} \quad \underbrace{\sum_{i,j=1}^{d} \mathbf{M}_{ij}(\vec{x}_p, \vec{x}_q, t)(\vec{x}_p - \vec{x}_q)_i (\vec{x}_p - \vec{x}_q)_j}_{\text{anisotropic}} \tag{5.21}$$

While the isotropic part of the operator (NOT of **D**!) looks analogous to isotropic PSE, there is a second part, which depends on the space directions

i and j. It contains the mapping tensor $\mathbf{M}$, which maps distance to strength in a direction-dependent way (before this was just the scalar η). In order for the method to conserve mass, $\mathbf{M}$ must be symmetric, such that $\mathbf{M}(\vec{x}_p, \vec{x}_q) = \mathbf{M}(\vec{x}_q, \vec{x}_p)$ for any pair of interacting particles p and q. The simplest way to ensure this is to set:

$$\mathbf{M}(\vec{x}_p, \vec{x}_q, t) = \frac{1}{2}(\mathbf{m}(\vec{x}_p, t) + \mathbf{m}(\vec{x}_q, t)) \,, \tag{5.22}$$

where $\mathbf{m}$ is related to the diffusion tensor as:

$$\mathbf{m}(\vec{x}, t) = \mathbf{D}(\vec{x}, t) - \frac{1}{d+2}\mathrm{Tr}(\mathbf{D}(\vec{x}, t)) \cdot \mathbb{1} \,. \tag{5.23}$$

Subtracting the trace from the diffusion tensor leaves the anisotropic part. This is correct because the isotropic part has already been accounted for in the prefactor to the sum in σ.
A frequently used choice in 3D for the radially symmetric isotropic kernel $\overline{\eta}(r)$ is:

$$\overline{\eta}_\epsilon(\vec{x}_p - \vec{x}_q) = \frac{4}{\epsilon^3 \pi \sqrt{\pi}} e^{-\frac{|\vec{x}_p - \vec{x}_q|_2^2}{\epsilon^2}} \qquad \text{in } \mathbb{R}^3 \,. \tag{5.24}$$

This kernel is second-order accurate, as it fulfills the moment conditions for $r = 2$.

5.2.1 Example

We show an example is using PSE to simulate isotropic homogeneous diffusion, as a benchmark, we compare to the method of Random Walk (RW), where particles never change their strength, but perform Brownian motion by adding Gaussian random numbers to their positions. The method of Random Walk is inspired by the microscopic interpretation of diffusion (i.e., Brownian motion), and is stochastic item-based simulation method (Well, at the same time it is also a field-based method using Monte-Carlo integration to perform the quadrature in the operator discretization). We solve the following benchmark problem on the one-dimensional ($d = 1$) ray $\Omega = [0, \infty)$, subject to the following initial and boundary conditions:

$$\begin{cases} u(x, t = 0) = u_0(x) = x e^{-x^2} & x \in [0, \infty),\ t = 0 \\ u(x = 0, t) = 0 & x = 0,\ 0 < t \leqslant T \,. \end{cases} \tag{5.25}$$

The exact analytic solution of this problem is

$$u_{\mathrm{ex}}(x, t) = \frac{x}{(1 + 4Dt)^{3/2}} e^{-x^2/(1+4Dt)} \,. \tag{5.26}$$

Both RW and PSE simulations of this benchmark case are performed with varying numbers of particles in order to study convergence. The boundary condition at $x = 0$ is satisfied using the method of images.

For the PSE we use the 2^{nd} order accurate Gaussian kernel

$$\eta_\epsilon(x) = \frac{1}{2\epsilon\sqrt{\pi}} e^{-x^2/4\epsilon^2}\,, \tag{5.27}$$

which fulfills the moment conditions in one dimension to order $r = 2$. The continuous-time equations of evolution for the particle positions and strengths are given by Eq. 5.17. We place N particles regularly spaced in the interval $[0, X]$, where the right-most particle location X is chosen such that the solution value never exceeds machine epsilon there. All particle volumes are then identical to $V_p = X/N$. The particles interact with all neighbors within a cutoff radius $r_c = 5\epsilon$, beyond which the interaction kernel becomes negligibly small. The `interact` method hence simply is:

```
method interact(q):
    z = (q.position - this.position)
    eta = Exp(-z*z/(4*epsilon*epsilon))
    eta = eta/(2*epsilon*Sqrt(pi))
    kw = (q.properties.strength - this.properties.strength)*eta
    kw = kw*((X/N)*D/(epsilon*epsilon))
    kx = 0
    return [kx,kw]
```

Listing 5.1: 1D PSE interaction method

Using explicit Euler for time integration of the particle strength (their only property), the `evolve` method is:

```
method evolve():
    ! Time step size dt is a parameter
    this.properties.strength += this.propertiesChange.strength * dt
```

Listing 5.2: 1D PSE evolution method

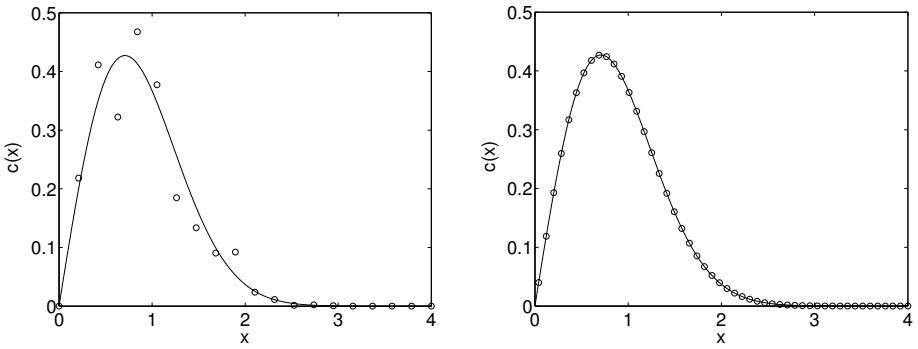
The particle positions never change, due to the PSE formulation.
The concentration values at particle locations x_p and simulation time points $t_n = n\delta t$ are recovered as

$$u^{\text{PSE}}(x_p, t^n) = \omega_p^n \cdot N/X = \omega_p/V_p\,,$$

as all particle volumes are same. For RW, binning of the particles is used to recover the concentration field in a piecewise constant approximation.
Figure 5.1 shows the RW and PSE solutions in comparison to the exact solution at a final time of $T = 10$ for $N = 50$ particles and a diffusion constant of $D = 10^{-4}$. The accuracy of the simulations for different numbers of particles is assessed by computing the final L_2 *error*

$$L_2 = \left[\frac{1}{N}\sum_{p=1}^{N}\left(u_{\text{ex}}(x_p, T) - u(x_p, T)\right)^2\right]^{1/2} \tag{5.28}$$

for each N. The resulting convergence curves are shown in Fig. 5.2.



(a) RW (b) PSE

Figure 5.1: Comparison of RW (a) and PSE (b) solutions of the benchmark case. The solutions at time $T = 10$ are shown (circles) along with the exact analytic solution (solid line). For both methods $N = 50$ particles, a time step of $\delta t = 0.1$, and $\nu = 10^{-4}$ are used. The RW solution is binned in $M = 20$ intervals of $\delta x = 0.2$. For the PSE a core size of $\epsilon = h$ is used.

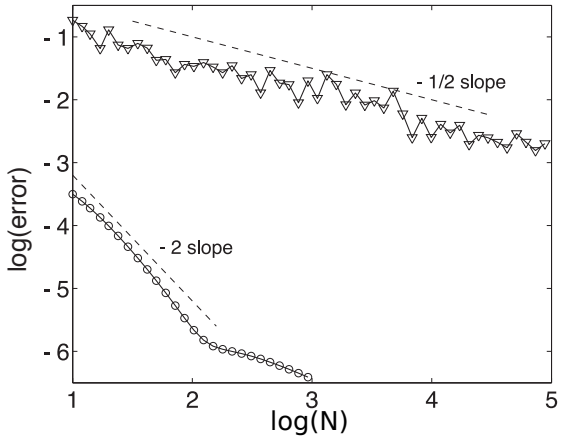


Figure 5.2: Convergence curves for RW and PSE. The L_2 error versus the number of particles for the RW (triangles) and the PSE (circles) solutions of the benchmark case at time $T = 10$ are shown. For both methods a time step of $\delta t = 0.1$ and $\nu = 10^{-4}$ are used. The RW solution is binned in $M = 20$ intervals of $\delta x = 0.2$ and for the PSE a core size of $\epsilon = h$ is used. The machine epsilon is $\mathcal{O}(10^{-6})$.

For RW we observe the characteristic slow convergence of $O(1/\sqrt{N})$. For PSE, a convergence of $O(1/N^2)$ is observed, in agreement with the employed 2^{nd} order kernel function. Below an error of 10^{-6}, machine precision is reached

(the simulations were done in single precision, which yields 23 bits of significant precision). It can be seen that the error of a PSE simulation is several orders of magnitude lower than the one of the corresponding RW simulation with the same number of particles. Using only 100 particles, PSE is already close to machine precision. It is evident from these results that large numbers of particles are necessary to achieve reasonable accuracy using RW in complex-shaped domains.

5.2.2 PSE for arbitrary differential operators

The concept of PSE can also be extended to arbitrary linear differential operators [32]. The idea is that using appropriate moment conditions, *any* derivative from the right-hand side of Eq. 5.11 can be isolated.
PSE operators approximate any spatial derivative

$$D^\beta f(\vec{x}) = \frac{\partial^{|\beta|} f(\vec{x})}{\partial x_1^{\beta_1} \partial x_2^{\beta_2} \dots \partial x_d^{\beta_d}} \tag{5.29}$$

of a (sufficiently smooth) field f by an integral operator over scattered particle locations [32]:

$$Q^\beta f(\vec{x}) = \frac{1}{\epsilon^{|\beta|}} \int_{\mathbb{R}^d} \left(f(\vec{y}) \pm f(\vec{x})\right) \eta_\epsilon^\beta(\vec{x} - \vec{y}) \, \mathrm{d}y \;=\; D^\beta f(\vec{x}) + \mathcal{O}(\epsilon^r) \,. \tag{5.30}$$

The operator kernel $\eta_\epsilon^\beta(\vec{z}) = \epsilon^{-d} \eta^\beta(\vec{z}/\epsilon)$ is scaled to width ϵ (kernel width) and chosen such as to fulfill continuous moment conditions [32]. The sign in Eq. 5.30 is positive for odd $|\beta|$ and negative for even $|\beta|$. This is because for odd $|\beta|$, the kernel η is also odd (then, all even terms in the Taylor expansion vanish) and thus $\eta(x_p - x_q) = -\eta(x_q - x_p)$ gives the sign change for symmetry. If all volumes are the same, the operator thus remains symmetric and conservative. Since for odd $|\beta|$, the 0-th moment vanishes, the sign of the constant term in the Taylor expansion can be chosen arbitrarily, because it is zero.
The integral operator in Eq. 5.30 is discretized by midpoint quadrature over the particles, thus,

$$Q_h^\beta f(\vec{x}_p) = \frac{1}{\epsilon^{|\beta|}} \sum_{q \in \mathcal{N}(\vec{x}_p)} V_q \left(f(\vec{x}_q) \pm f(\vec{x}_p)\right) \eta_\epsilon^\beta(\vec{x}_q - \vec{x}_p) \,, \tag{5.31}$$

where $\vec{x}_p$ and V_p are the position and the volume of particle p, respectively, and $\mathcal{N}(\vec{x})$ is the set of all particles in an r_c-neighborhood around $\vec{x}$. The cutoff radius r_c of the operator is defined such that $\mathcal{N}(\vec{0})$ approximates the support of η_ϵ^β with a certain accuracy. The resolution of the discretization is given by the characteristic interparticle spacing h, defined as the d^{th} root of the average particle volume.
The PSE method overcomes two of the main limitations of SPH: the resulting scheme is symmetric and hence conservative, provided all particles have the same volume $V_p = V_q = V$ or PSE is used in strong form. Also, the PSE kernels have the same order of convergence for all degrees of derivatives. The former is

trivial to see, since the PSE interaction operator for identical volumes only depends on the difference between the two interacting particles. The effect of p on q is hence symmetric with the effect of q on p. This symmetry allows us to use symmetric cell lists and Verlet lists in order to reduce the computational cost by a factor of two. Moreover, it renders the method conservative in the sense that the total strength in the system is always conserved to machine precision. This is because particles only *exchange* strength (hence then name "particle strength exchange"), but no strength is lost or created during an interaction. The latter point is due to the fact that different differential operators are approximated using different kernels. This provides the degrees of freedom necessary to engineer them all to the same order of convergence.
The main drawbacks of PSE are:
(1) Boundary conditions are difficult to impose. For particles in an r_c-neighborhood from the boundary, some of the interaction partners are missing. While the operator can still be computed, the result is going to be wrong. As a remedy, it has been proposed to use one-sided kernels at all particles in an r_c-neighborhood from the boundary [32]. These then only interact with partners in the half of the Verlet sphere toward the interior of the domain. This, however, introduces a conditional statement into the inner loop of the method, preventing it from vectorizing. Moreover, the one-sided kernels fail if the boundary is curved on the length scale of r_c or below. A second possibility is to place mirror particles outside the domain (i.e., mirroring all particles in an r_c-neighborhood from the boundary at the boundary) and then computing the full interaction spheres including the mirror particles. This allows imposing homogeneous boundary conditions only. For homogeneous Neumann conditions, the mirror particles are given the same value as the respective source particle. For homogeneous Dirichlet boundary conditions, the sign is flipped. Besides its limitation to homogeneous boundary conditions, this so-called *method of images* is only first-order accurate (on general boundaries; at flat boundaries it is exact), hence reducing the convergence order of the method to 1 in the L_∞-norm. The most general method is to solve a separate boundary integral problem for the particles near the boundary and then modifying their strength such that the boundary condition is satisfied when evaluating the PSE operator there. This can be done using a heat-panel method [33] or by extrapolation [34]. While this works for arbitrary boundaries, it is the most involved method.
(2) The PSE method is still inconsistent, like SPH, but for a different reason. This is because the overlap condition requires that h and ϵ are proportional. Hence, h/ϵ is a constant, and the quadrature error of $(h/\epsilon)^s$ is also a constant for any quadrature with finite s (like the midpoint quadrature). This means that even when increasing the resolution of a simulation (i.e., decreasing h), the quadrature error remains constant. Sooner or later in a convergence plot, the error hits this plateau and does not decrease any further. The method is hence, strictly speaking, inconsistent. The only way around this is to simultaneously increase the number of particles in the support of the kernel, as h decreases. This means that h decreases faster than ϵ and the computational cost of the method increases over-proportional with N. This is also the case for SPH,

which also has an error plateau from the quadrature. However, in SPH the error from losing one order with every derivative usually dominates, making it rather inconsistent with β. PSE is consistent with β, but still inconsistent with ϵ.
(3) Both SPH and PSE are sensitive to distortions in the particle arrangement ("Lagrangian grid distortion"). They work well for particles that are symmetrically and more or less evenly distributed. In that case, the continuous moment conditions used to derive the kernels also hold to a good approximation in the discrete, i.e., when actually evaluating the kernels only at the particle locations. For sufficiently distorted or irregular particle distributions, however, this is no longer the case. Then, the discrete moments one actually gets when evaluating the kernels over a given particle distribution may be too different from the continuous moments, hence making leading-order terms in the Taylor expansion dominate the error again. This is the point where DC-PSE comes in.

5.3 DC-PSE

DC-PSE was originally introduced as a discretization correction to PSE in order to address the above two limitations of PSE. DC-PSE operators transparently handle boundaries and are not limited by any quadrature error. The latter is achieved by getting rid of the quadrature altogether by directly formulating the operator in the discrete domain. Satisfying the moment conditions discretely on the *given* particle distribution also guarantees that the method converges at full order for irregular particle distributions. Since on irregular particle distributions, the local neighborhood of each particle looks different, also the discrete moment conditions around different particles can be different. This means that in DC-PSE, we do not only have a different kernel for each differential operator, but a different kernel for each particle. This provides the necessary degrees of freedom to satisfy the moment conditions everywhere. But it also creates three problems: (1) kernels cannot be analytically pre-computed any more and need to be determined at runtime, and re-determined every time the particles moved. This creates additional computational cost, which may, however, be amortized by the higher accuracy and stability of the resulting simulation [35, 36]. (2) For certain "pathological" particle distributions (e.g., all particles are on a line) in the neighborhood of any particle, the system of moment conditions may not have full rank or be ill-conditioned. Then, the method does not work. (3) Since the kernel is different for every particle, the method is no longer symmetric, and hence also not conservative. It can be made conservative only for first-order convergence. For all higher convergence orders symmetric DC-PSE operators do not exist, and symmetric cell- or Verlet-lists cannot be used. While simulations of equilibrium models may benefit from exact conservation, DC-PSE is well-suited for simulations of non-equilibrium models or open systems where no conservation laws exist. It is also well suited for simulations in moving or complex-shaped domains and near boundaries, since the kernels automatically adjust to the actual particle distribution at runtime.

We exemplify the derivation in 2D here by considering a differential operator, of arbitrary order, for a sufficiently smooth field $f(\boldsymbol{x}) = f(x, y)$ at point $\boldsymbol{x}_p = \{x_p, y_p\}$ of a particular particle set

$$\boldsymbol{D}^{m,n} f(\boldsymbol{x}_p) = \frac{\partial^{m+n}}{\partial x^m \partial y^n} f(x, y)\bigg|_{x=x_p, y=y_p} \tag{5.32}$$

where m and n are integers that determine the order of the differential operator. The DC PSE operator for the spatial derivative $\boldsymbol{D}^{m,n} f(\boldsymbol{x}_p)$ looks like the standard PSE operator from Eq. 5.31:

$$\boldsymbol{Q}^{m,n} f(\boldsymbol{x}_p) = \frac{1}{\epsilon(\boldsymbol{x}_p)^{m+n}} \sum_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)} \left(f(\boldsymbol{x}_q) \pm f(\boldsymbol{x}_p)\right) \eta\left(\frac{\boldsymbol{x}_p - \boldsymbol{x}_q}{\epsilon(\boldsymbol{x}_p)}\right). \tag{5.33}$$

The difference is that the kernel η now satisfies discrete moment conditions and that $\epsilon(\boldsymbol{x})$ is a function of space, since we also allow irregular particle distributions where h is different for different particles. The original weak-form PSE formulation also includes a particle volume V_p and a dimension-dependent normalization factor for the particle volume $\epsilon(\boldsymbol{x}_p)^{-d}$, where d is the spatial dimension, providing a normalization of the integration length, area, or volume for the particle. As we now allow each particle to have a different $\epsilon_p = \epsilon(\boldsymbol{x}_p)$, both the particle volume and the normalization pre-factor of the kernel can be absorbed into ϵ_p. Since ϵ_p is determined numerically at runtime, there is no need in DC-PSE to care about particle volumes. $\mathcal{N}(\boldsymbol{x}_p)$ is the set of points in the support of the kernel function. Just as in standard PSE, the sign in Eq. (5.33) is positive for odd $(m + n)$, and negative for even.

We want to construct the DC-PSE operators so that as we decrease the spacing between particles, $h(\boldsymbol{x}_p) \to 0$, the operator converges to the spatial derivative $\boldsymbol{D}^{m,n} f(\boldsymbol{x}_p)$ with an asymptotic rate r for all positions $\boldsymbol{x}_p$:

$$\boldsymbol{Q}^{m,n} f(\boldsymbol{x}_p) = \boldsymbol{D}^{m,n} f(\boldsymbol{x}_p) + \mathcal{O}(h(\boldsymbol{x}_p)^r), \tag{5.34}$$

where it is convenient to explicitly define the component-wise average neighbor spacing as $h(\boldsymbol{x}_p) = \frac{1}{N} \sum_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)} (|x_p - x_q| + |y_p - y_q|)$, where N is the number of particles in the support of $\boldsymbol{x}_p$.

Therefore, we need to find a kernel function $\eta(\boldsymbol{x})$ and a scaling relation $\epsilon(\boldsymbol{x}_p)$ that satisfy Eq. (5.34). To achieve this, we replace the term $f(\boldsymbol{x}_q)$ in Eq. (5.33) with its Taylor expansion around $\boldsymbol{x}_p$:

$$\boldsymbol{Q}^{m,n} f(\boldsymbol{x}_p) = \frac{1}{\epsilon(\boldsymbol{x}_p)^{m+n}} \sum_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)} \left(\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \frac{(x_p - x_q)^i (y_p - y_q)^j (-1)^{i+j}}{i! j!} \boldsymbol{D}^{i,j} f(\boldsymbol{x}_p) \right.$$
$$\left. \pm f(\boldsymbol{x}_p) \right) \eta\left(\frac{\boldsymbol{x}_p - \boldsymbol{x}_q}{\epsilon(\boldsymbol{x}_p)}\right). \tag{5.35}$$

This can be rewritten as:

$$\boldsymbol{Q}^{m,n} f(\boldsymbol{x}_p) = \left(\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \frac{\epsilon(\boldsymbol{x}_p)^{i+j-m-n}(-1)^{i+j}}{i!j!} \boldsymbol{D}^{i,j} f(\boldsymbol{x}_p) Z^{i,j}(\boldsymbol{x}_p) \right) \pm Z^{0,0}(\boldsymbol{x}_p)\epsilon(\boldsymbol{x}_p)^{-m-n} f(\boldsymbol{x}_p), \tag{5.36}$$

where

$$Z^{i,j}(\boldsymbol{x}_p) = \sum_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)} \frac{(x_p - x_q)^i (y_p - y_q)^j}{\epsilon(\boldsymbol{x}_p)^{i+j}} \eta\left(\frac{\boldsymbol{x}_p - \boldsymbol{x}_q}{\epsilon(\boldsymbol{x}_p)} \right) \tag{5.37}$$

are the *discrete moments* of η. In order to keep the number of neighbors of each particle bounded by a constant (for computational efficiency), we require the scaling parameter $\epsilon(\boldsymbol{x}_p)$ to converge at the same rate as the average spacing between points $h(\boldsymbol{x}_p)$, that is

$$\frac{h(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)} \in \mathcal{O}(1), \tag{5.38}$$

then we find that the discrete moments $Z^{i,j}$ are $\mathcal{O}(1)$ as $h(\boldsymbol{x}_p) \to 0$ and $\epsilon(\boldsymbol{x}_p) \to 0$. This is because the terms $\frac{(x_p - x_q)^i (y_p - y_q)^j (-1)^{i+j}}{\epsilon(\boldsymbol{x}_p)^{i+j}}$ are $\mathcal{O}(1)$ from the scaling relation and definition of $h(\boldsymbol{x}_p)$. Further, the second term $\eta\left(\frac{\boldsymbol{x}_p - \boldsymbol{x}_q}{\epsilon(\boldsymbol{x}_p)} \right)$ is $\mathcal{O}(1)$, through normalization of the function argument. Therefore, the scaling behavior of Eq. (5.36) is determined solely by the $\epsilon(\boldsymbol{x}_p)^{i+j-m-n}$ term of smallest power with non-zero coefficient. Note that Eq. (5.38) is a much looser constraint on the average spacing of particles than the overlap condition of the PSE method. We no longer need to require $h/\epsilon < 1$, but it can be bounded by any other constant, also > 1.

Given Eq. (5.38), the convergence rate r of the DC PSE operator $\boldsymbol{Q}^{m,n}$ (Eqs. 5.34 and 5.36) is determined by the coefficients of the terms $\epsilon(\boldsymbol{x}_p)^{i+j-m-n}$ in Eq. 5.36. This coefficient is required to be 1 when $i = m$ and $j = n$, and 0 otherwise as long as $i + j - m - n < r$. This results in the following set of conditions for the discrete moments,

$$Z^{i,j}(\boldsymbol{x}_p) = \begin{cases} i!j!(-1)^{i+j} & i = m, j = n \\ 0 & \alpha_{\min} \leq i + j < r + m + n \\ < \infty & \text{otherwise} \end{cases} \tag{5.39}$$

where $\alpha_{\min}$ is 1 if $m + n$ is even and 0 if odd. This is due to the zeroth moment $Z^{0,0}$ trivially canceling out for odd $m + n$, whereas it *cannot* (and must not) be zero for even $m + n$. Note that the pre-factor $\epsilon(\boldsymbol{x}_p)^{-m-n}$ in Eq. (5.33) simplifies the expression of the moment conditions.

For the kernel function $\eta(\boldsymbol{x})$ to be able to satisfy the l conditions given in Eq. (5.39) for arbitrary particle distributions, the operator must have l degrees of freedom. This leads to the requirement that the support $\mathcal{N}(\boldsymbol{x})$ of the kernel

function has to include at least l neighboring particles. It is common to use kernel functions of the form [35]

$$\eta(\boldsymbol{x}) = \begin{cases} \sum\limits_{i,j}^{i+j<r+m+n} a_{i,j} x^i y^j e^{-x^2-y^2} & \sqrt{x^2+y^2} < r_c \\ 0 & \text{otherwise.} \end{cases} \tag{5.40}$$

This is a monomial basis multiplied by an exponential window function, where r_c sets the kernel support and the $a_{i,j}$ are scalars to be determined to satisfy the moment conditions in Eq. (5.39). The cut-off radius r_c should be set to include at least l particles in the support $\mathcal{N}(x)$. A simple choice is to set r_c to include the $l-1$ nearest neighbors of each particle.
If $\alpha_{\min} = 1$, the $a_{0,0}$ coefficient is a free parameter and can be used to increase the numerical robustness of solving the linear system of equations for the remaining $a_{i,j}$ [35].
Since the kernels are different for different particles and can hence not be precomputed, they are determined numerically at runtime. This means that the coefficients $a_{i,j}$ are found by solving a linear system of equations resulting from the moment conditions. With the above choice of kernel function we have,

$$\boldsymbol{Q}^{m,n} f(\boldsymbol{x}_p) = \frac{1}{\epsilon(\boldsymbol{x}_p)^{m+n}} \sum_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)} \left(f(\boldsymbol{x}_q) \pm f(\boldsymbol{x}_p)\right) \boldsymbol{p}\left(\frac{\boldsymbol{x}_p - \boldsymbol{x}_q}{\epsilon(\boldsymbol{x}_p)}\right) \boldsymbol{a}^T(\boldsymbol{x}_p) e^{\frac{-(x_p-x_q)^2-(y_p-y_q)^2}{\epsilon(\boldsymbol{x}_p)^2}}, \tag{5.41}$$

where $\boldsymbol{p}(\boldsymbol{x}) = \{p_1(\boldsymbol{x}), p_2(\boldsymbol{x}), \ldots, p_l(\boldsymbol{x})\}$ and $\boldsymbol{a}(\boldsymbol{x})$ are vectors of the monomial basis and of their coefficients in Eq. (5.40), respectively.
Using this formulation, the operator system becomes straightforward. For example, if we set $r = 2$ and approximate the first spatial derivative in the x direction, $D^{1,0}$, we have $l = 6$ moment conditions ($r + m + n = 3$, $\alpha_{\min} = 0$: $(i,j) = \{(0,0),(1,0),(0,1),(1,1),(0,2),(2,0)\}$) and the monomial basis is $\boldsymbol{p}(x,y) = \{1, x, y, yx, x^2, y^2\}$. The linear system for the kernel coefficients then is:

$$\boldsymbol{A}(\boldsymbol{x}_p)\boldsymbol{a}^T(\boldsymbol{x}_p) = \boldsymbol{b}, \tag{5.42}$$

where

$$\boldsymbol{A}(\boldsymbol{x}_p) = \boldsymbol{B}(\boldsymbol{x}_p)^T \boldsymbol{B}(\boldsymbol{x}_p) \in \mathbb{R}^{l \times l} \tag{5.43}$$

$$\boldsymbol{B}(\boldsymbol{x}_p) = \boldsymbol{E}(\boldsymbol{x}_p)^T \boldsymbol{V}(\boldsymbol{x}_p) \in \mathbb{R}^{k \times l} \tag{5.44}$$

$$\boldsymbol{b} = (-1)^{m+n} \boldsymbol{D}^{m,n} \boldsymbol{p}(\boldsymbol{x})|_{x=0} \in \mathbb{R}^{l \times 1}. \tag{5.45}$$

The scalar number $k \geq l$ is the number of particles in the support of the operator, l the number of moment conditions to be satisfied, and $\boldsymbol{V}(\boldsymbol{x}_p)$ the Vandermonde matrix constructed from the monomial basis $\boldsymbol{p}(\boldsymbol{x}_p)$. $\boldsymbol{E}(\boldsymbol{x}_p)$ is a diagonal matrix containing the square roots of the values of the exponential window function at

the neighboring particles in the operator support. Further, for particle $\boldsymbol{x}_p$ we define $\{\boldsymbol{z}_q(\boldsymbol{x}_p)\}_{q=1}^k = \{\boldsymbol{x}_p - \boldsymbol{x}_q\}_{\boldsymbol{x}_q \in \mathcal{N}(\boldsymbol{x}_p)}$, the set of vectors pointing to $\boldsymbol{x}_p$ from all neighboring particles $\boldsymbol{x}_q$ in the support of $\boldsymbol{x}_p$. Then explicitly

$$\boldsymbol{V}(\boldsymbol{x}_p) = \begin{pmatrix} p_1\left(\frac{\boldsymbol{z}_1(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & p_2\left(\frac{\boldsymbol{z}_1(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & \cdots & p_l\left(\frac{\boldsymbol{z}_1(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) \\ p_1\left(\frac{\boldsymbol{z}_2(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & p_2\left(\frac{\boldsymbol{z}_2(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & \cdots & p_l\left(\frac{\boldsymbol{z}_2(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ p_1\left(\frac{\boldsymbol{z}_k(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & p_2\left(\frac{\boldsymbol{z}_k(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) & \cdots & p_l\left(\frac{\boldsymbol{z}_k(\boldsymbol{x}_p)}{\epsilon(\boldsymbol{x}_p)}\right) \end{pmatrix} \in \mathbb{R}^{k \times l} \tag{5.46}$$

$$\boldsymbol{E}(\boldsymbol{x}_p) = \operatorname{diag}\left(\left\{ e^{\frac{-|\boldsymbol{z}_q(\boldsymbol{x}_p)|^2}{2\epsilon(\boldsymbol{x}_p)^2}} \right\}_{q=1}^{k}\right) \in \mathbb{R}^{k \times k}. \tag{5.47}$$

Once the matrix $\boldsymbol{A}(\boldsymbol{x}_p)$ is constructed at each particle $\boldsymbol{x}_p$, the linear systems can be solved for the coefficients $\boldsymbol{a}(\boldsymbol{x}_p)$ used in the DC PSE operators at each particle as in Eq. (5.41). The matrix $\boldsymbol{A}(\boldsymbol{x}_p)$ only depends on the number of moment conditions l and the local distribution of particles in $\mathcal{N}(\boldsymbol{x}_p)$. Therefore, if the system in Eq. (5.42) is solved using a decomposition (such as LU) of $\boldsymbol{A}(\boldsymbol{x}_p)$. this form can be re-used for multiple right-hand sides, i.e., for different differential operators (albeit with different convergence rates r). The matrix $\boldsymbol{A}$ contains information about the spatial distribution of the particles around the center particle $\boldsymbol{x}_p$. The invertibility of $\boldsymbol{A}$ depends entirely on that of the Vandermonde matrix $\boldsymbol{V}$, due to $\boldsymbol{E}$ being a diagonal matrix with non-zero entries. The condition number of $\boldsymbol{A}$ depends on both $\boldsymbol{V}$ and $\boldsymbol{E}$ and determines the robustness of the numerical inversion.

5.3.1 Finite-differences are a limit case of DC-PSE (Optional Material)

For uniform Cartesian particle distributions with spacing h and a finite operator support of radius r_c, the DC-PSE operator (Eq. 5.33) can be rewritten as

$$Q_h^{\boldsymbol{\beta}} f(\boldsymbol{x}) = \frac{c^n}{\varepsilon^{|\boldsymbol{\beta}|}} \sum_{|\boldsymbol{k}|^2=0}^{\lfloor r_c^2/h^2 \rfloor} \left(f(\boldsymbol{x} + \boldsymbol{k}h) \pm f(\boldsymbol{x})\right) \eta^{\boldsymbol{\beta}}(-c\boldsymbol{k}), \quad \boldsymbol{k} \in \mathbb{Z}^n\,. \tag{5.48}$$

Using the kernel template given in Eq. 5.40, the value of the DC kernel function at $-c\boldsymbol{k}$ is

$$\eta^{\boldsymbol{\beta}}(-c\boldsymbol{k}) = \left(\sum_{\substack{|\boldsymbol{\gamma}|=\alpha_{\min} \\ \boldsymbol{\beta}+\boldsymbol{\gamma} \text{ even}}}^{|\boldsymbol{\beta}|+r-1} a_{\boldsymbol{\gamma}} (-c\boldsymbol{k})^{\boldsymbol{\gamma}} \right) e^{-c^2|\boldsymbol{k}|^2} \tag{5.49}$$

and the discrete moments become

$$Z_h^{\boldsymbol{\alpha}} = c^n \sum_{|\boldsymbol{k}|^2=0}^{\lfloor r_c^2/h^2 \rfloor} \sum_{\substack{|\boldsymbol{\gamma}|=\alpha_{\min} \\ \boldsymbol{\beta}+\boldsymbol{\gamma} \text{ even}}}^{|\boldsymbol{\beta}|+r-1} a_{\boldsymbol{\gamma}} (c\boldsymbol{k})^{\boldsymbol{\alpha}+\boldsymbol{\gamma}} e^{-c^2|\boldsymbol{k}|^2}\,. \tag{5.50}$$

Here, "$\boldsymbol{\beta}+\boldsymbol{\gamma}$ even" stands for all multiindices $\boldsymbol{\gamma}$ for which $\boldsymbol{\beta}+\boldsymbol{\gamma}$ contains only even elements. All other $\boldsymbol{\gamma}$ need not be considered since the corresponding coefficients $a_{\boldsymbol{\gamma}}$ can *a priori* be set to zero.
The DC PSE operators for $c \to \infty$ can be derived from Eqs. 5.48 to 5.50 and the moment conditions. For the second-order accurate DC PSE operator approximating the first derivative along dimension i ($r=2$, $\boldsymbol{\beta}=\boldsymbol{e}_i$), for example, the DC kernel function can be written as

$$\eta^{\boldsymbol{e}_i}(-c\boldsymbol{k}) = \frac{k_i e^{-c^2|\boldsymbol{k}|^2}}{c^{n+1}\sum_{|\boldsymbol{l}|^2=0}^{\lfloor r_c^2/h^2 \rfloor} l_i^2 e^{-c^2|\boldsymbol{l}|^2}}\,. \tag{5.51}$$

Using this kernel, the operator (Eq. 5.48) becomes

$$Q_h^{\boldsymbol{e}_i} f(\boldsymbol{x}) = \frac{\sum_{|\boldsymbol{k}|^2=0}^{\lfloor r_c^2/h^2 \rfloor} \left(f(\boldsymbol{x}+\boldsymbol{k}h)+f(\boldsymbol{x})\right) k_i e^{-c^2|\boldsymbol{k}|^2}}{h\sum_{|\boldsymbol{k}|^2=0}^{\lfloor r_c^2/h^2 \rfloor} k_i^2 e^{-c^2|\boldsymbol{k}|^2}}\,.$$

This is a FD stencil with extent and weights that can be adjusted by the choice of the cutoff radius r_c and the ratio c. Letting $c \to \infty$ yields

$$\lim_{c\to\infty} Q_h^{\boldsymbol{e}_i} f(\boldsymbol{x}) = \frac{f(\boldsymbol{x}+\boldsymbol{h}_i)-f(\boldsymbol{x}-\boldsymbol{h}_i)}{2h}\,, \quad \boldsymbol{h}_i = h\boldsymbol{e}_i\,, \tag{5.52}$$

for any value of $r_c \geq h$. This is the classical centered difference stencil for the first derivative of f.
Following the same procedure, the second-order DC PSE operator approximating the Laplacian $\Delta f(\boldsymbol{x}) = \nabla^2 f(\boldsymbol{x})$ becomes

$$\lim_{c\to\infty} Q_h^{\mathrm{Lap}} f(\boldsymbol{x}) = \lim_{c\to\infty} \sum_{i=1}^{n} Q_h^{2\boldsymbol{e}_i} f(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} \left[f(\boldsymbol{x}+\boldsymbol{h}_i)-2f(\boldsymbol{x})+f(\boldsymbol{x}-\boldsymbol{h}_i)\right]}{h^2} \tag{5.53}$$

and the fourth-order DC PSE approximation of the first derivative along $\boldsymbol{e}_i$ yields

$$\lim_{c\to\infty} Q_h^{\boldsymbol{e}_i} f(\boldsymbol{x}) = \frac{-f(\boldsymbol{x}+2\boldsymbol{h}_i)+8f(\boldsymbol{x}+\boldsymbol{h}_i)-8f(\boldsymbol{x}-\boldsymbol{h}_i)+f(\boldsymbol{x}-2\boldsymbol{h}_i)}{12h}\,. \tag{5.54}$$

On Cartesian particle distributions, all these classical compact FD stencils can hence be interpreted as DC PSE operators with a kernel width ε tending to zero (grid points).

Chapter 6

Eulerian Particle Methods for Field-based Models

In this chapter:

- Eulerian particle methods
- Numerical scheme using Eulerian particle methods
- Numerical stability of Eulerian methods
- Numerical stability due to advection in Eulerian particle methods

Learning goals:

- Be able to devise Eulerian numerical schemes
- Be able to analyze numerical stability of Eulerian particle methods
- Know the advantages and disadvantages of Eulerian particle methods

In this chapter, we use the techniques presented in the Chapters 3 and 5 to develop particle methods for numerically solving continuous-time field-based models. Recall the general equation of motion for such particle methods that was presented in Chapters 3 and 5:

$$\begin{aligned} \frac{\mathrm{d}\vec{x}_p(t)}{\mathrm{d}t} &= \vec{v}_p(t, \vec{x}(t), \vec{\omega}(t)) \\ \frac{\mathrm{d}\vec{\omega}_p(t)}{\mathrm{d}t} &= \vec{g}_p(t, \vec{x}(t), \vec{\omega}(t)), \end{aligned} \tag{6.1}$$

where $\vec{x}_p(t)$ is the position, $\vec{\omega}_p(t)$ is the property, $\vec{v}_p$ is the velocity, and $\vec{g}_p$ is the property rate of particle p. Using this framework, we will develop Eulerian particle methods for field-based models in this chapter. In Eulerian particle

methods, position of particles do not change in time and only the properties evolve as a function of time. That is,

$$\begin{aligned} \frac{\mathrm{d}\vec{x}_p(t)}{\mathrm{d}t} &= 0 \\ \frac{\mathrm{d}\vec{\omega}_p(t)}{\mathrm{d}t} &= \vec{g}_p^{\,\mathrm{Eul}}(t, \vec{x}(t), \vec{\omega}(t)), \end{aligned} \tag{6.2}$$

where $\vec{g}_p^{\,\mathrm{Eul}}$ denotes the property rate in Eulerian particle methods. Since the particle positions do not change in time, the equation of motion for any Eulerian particle method is simply:

$$\frac{\mathrm{d}\vec{\omega}_p(t)}{\mathrm{d}t} = \vec{g}_p^{\,\mathrm{Eul}}(t, \vec{x}(t), \vec{\omega}(t)). \tag{6.3}$$

We will apply this framework (Eq. 6.3) to develop Eulerian particle methods to numerically solve a model equation. As a model equation, we choose a general transport equation, namely the advection-diffusion equation.

6.1 Model equation: Advection-diffusion

The general equation describing the transport of a scalar quantity u due to advection and diffusion processes is given by

$$\frac{\partial u}{\partial t} + \boldsymbol{\nabla} \cdot (\boldsymbol{v}\, u) = \boldsymbol{\nabla} \cdot (\mathbf{D} \boldsymbol{\nabla} u) + S(\vec{x}, t). \tag{6.4}$$

Here u is the scalar quantity being transported. This could be concentration in a description for mass transfer, or temperature in a description for heat transfer. The tensor $\mathbf{D}$ is the diffusivity. The quantity $\boldsymbol{v}$ is the velocity with which the scalar quantity u is advected. If $\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$, the flow field or the velocity field $\boldsymbol{v}$ is incompressible. If $\boldsymbol{v}$ is incompressible, then the advection term $\boldsymbol{\nabla} \cdot (\boldsymbol{v}\, u)$ simplifies to $\boldsymbol{v} \cdot \boldsymbol{\nabla} u$. S represents a source term. The source term could for example be chemical reaction terms in a mass transfer description. $\boldsymbol{\nabla}$ is the spatial gradient or nabla operator. In three-dimensional coordinates (x, y, z), $\boldsymbol{\nabla} = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$. We consider an isotropic, spatially homogeneous diffusivity so that $\mathbf{D} = D$.
Rewriting Eq. 6.4 after expanding the advection term ($\boldsymbol{\nabla} \cdot (\boldsymbol{v}\, u)$) and considering only an isotropic, homogeneous diffusion, Eq. 6.4 simplifies to:

$$\frac{\partial u}{\partial t} = D \nabla^2 u - \boldsymbol{v} \cdot (\boldsymbol{\nabla}\, u) - u(\boldsymbol{\nabla} \cdot \boldsymbol{v}) + S(\vec{x}, t). \tag{6.5}$$

We now discretize the spatial differential operators using the methods presented in Chapter 5. Specifically, we use PSE for discretizing for the sake of illustration over DC-PSE. In DC-PSE, most things happen numerically at runtime and there is not much we could illustrate here on paper. Moreover, the advection-diffusion equation is derived from conservation of mass, so one may prefer PSE over DC

PSE in practical applications due to conservation properties. The results and the methodology used in the rest of chapter is, however, applicable to DC-PSE and other discretization methods. Using PSE and the explicit Euler time stepping scheme presented in Chapter 3, we now attempt to develop a stable numerical scheme for numerically solving Eq. 6.5. For this we first consider only the diffusion term and then consider the advection terms alone before combining our results. Note that the source term $S(\vec{x}, t)$ contains no differential operators and is simply evaluated point-wise.

6.2 Only diffusion

Setting $\boldsymbol{v} = 0$ in Eq. 6.5, we obtain the diffusion equation:

$$\frac{\partial u}{\partial t} = D\nabla^2 u. \tag{6.6}$$

Using PSE to discretize the Laplacian operator, the equation of motion for the scalar quantity u carried by particle p is (refer to Chapter 5) is

$$\frac{\mathrm{d}u_p}{\mathrm{d}t} = \frac{D}{\epsilon^2}\sum_{q=1}^{N} V_q(u_q - u_p)\eta_\epsilon(\vec{x}_p - \vec{x}_q). \tag{6.7}$$

Using explicit Euler time-stepping scheme (see Chapter 3), the above equation can be written as

$$u_p^{n+1} = u_p^n + \delta t\frac{D}{\epsilon^2}\sum_{q=1}^{N} V_q(u_q^n - u_p^n)\eta_\epsilon(\vec{x}_p - \vec{x}_q), \tag{6.8}$$

where $u_p^n = u_p(t = n\delta t)$. The kernel η_ϵ is given by:

$$\begin{aligned} \eta_\epsilon(x) &= \frac{1}{2\epsilon\sqrt{\pi}} e^{-\frac{x^2}{4\epsilon^2}} \qquad x \in \mathbb{R} \\ \eta_\epsilon(\vec{x}) &= \frac{1}{4\epsilon^2\pi} e^{-\frac{|\vec{x}|^2}{4\epsilon^2}} \qquad \vec{x} \in \mathbb{R}^2. \end{aligned} \tag{6.9}$$

Order of accuracy. Using these kernels, the numerical scheme given in Eq. 6.8 to numerical solve Eq. 6.6 is second-order accurate in space and first-order accurate in time.

6.2.1 Stability

We now investigate the numerical stability of Eq. 6.8. For this we make use of von Neumann stability analysis assuming a periodic domain. We consider the general integral PSE formulation where second derivative can be approximated as [32]

$$\nabla^2 u(\vec{x}) \approx \frac{1}{\epsilon^2}\int_\Omega (u(\vec{y}) - u(\vec{x}))\eta_\epsilon(\vec{x} - \vec{y})d\vec{y}, \tag{6.10}$$

where the kernel is given by Eq. 6.9. (Note that in practice the integral is approximated by a numerical quadrature leading to Eq. 6.8).
We consider a one-dimensional domain for the sake of simplicity. The numerical scheme in one dimension to numerically solve Eq. 6.6 with explicit Euler time stepping, and Eq. 6.10 for approximating the second derivative is

$$u_p^{n+1} = u_p^n + \frac{D\delta t}{\epsilon^2}\int_{-\infty}^{\infty}(u^n(y) - u_p^n)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{4\epsilon^2}}dy. \tag{6.11}$$

Now we postulate that

$$u = \hat{u} + \theta,$$

where $\hat{u}$ is the exact solution of Eq. 6.11, u is the numerical solution, and θ is the error. Since $\hat{u}$ is the exact solution of Eq. 6.11, it satisfies the equation by definition. Therefore, the error θ also satisfies Eq. 6.11. That is,

$$\theta_p^{n+1} = \theta_p^n + \delta t\frac{D}{\epsilon^2}\int_{-\infty}^{\infty}(\theta^n(y) - \theta_p^n)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{4\epsilon^2}}dy. \tag{6.12}$$

We now use Fourier series to represent the error θ such that

$$\theta(x,t) = \sum_{m=1}^{M} e^{a_m t}e^{ik_m x},$$

where k_m are the spatial Fourier wave numbers and $e^{a_m t}$ is the amplification factor of the Fourier mode with wavenumber k_m. Substituting the above ansatz in Eq. 6.12 yields:

$$\sum_{m=1}^{M} e^{a_m t_{n+1}}e^{ik_m x_p} = \sum_{m=1}^{M} e^{a_m t_n}e^{ik_m x_p} + \frac{\delta t D}{\epsilon^2}\int_{-\infty}^{\infty}\left(\sum_{m=1}^{M} e^{a_m t_n}e^{ik_m y} - \right. \tag{6.13}$$

$$\left. \sum_{m=1}^{M} e^{a_m t_n}e^{ik_m x_p}\right)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{4\epsilon^2}}dy. \tag{6.14}$$

This equation has to be true for each Fourier modes m individually in order for the sequences on both sides of the equality to be the same. Therefore:

$$e^{a_m t_{n+1}}e^{ik_m x_p} = e^{a_m t_n}e^{ik_m x_p} + \frac{\delta t D}{\epsilon^2}\int_{-\infty}^{\infty}\left(e^{a_m t_n}e^{ik_m y} - e^{a_m t_n}e^{ik_m x_p}\right)\eta dy \tag{6.15}$$

$$e^{a_m t_n}e^{a_m \delta t}e^{ik_m x_p} = e^{a_m t_n}e^{ik_m x_p} + \frac{\delta t D}{\epsilon^2}\int_{-\infty}^{\infty}\left(e^{a_m t_n}(e^{ik_m y} - e^{ik_m x_p})\right)\eta dy \tag{6.16}$$

where we used $t_{n+1} = t_n + \delta t$ in the first step. Dividing the whole equation with the non-zero number $e^{a_m t_n}e^{ik_m x_p}$, we find:

$$e^{a_m \delta t} = 1 + \frac{D\delta t}{\epsilon^2}\int_{-\infty}^{\infty}(e^{ik_m(y-x_p)} - 1)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{4\epsilon^2}}\,\mathrm{d}y, \tag{6.17}$$

$$= 1 + \frac{D\delta t}{\epsilon^2}\int_{-\infty}^{\infty}(e^{ik_m x} - 1)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{x^2}{4\epsilon^2}}\,\mathrm{d}x, \tag{6.18}$$

$$= 1 + \frac{D\delta t}{\epsilon^2}(e^{-\epsilon^2 k_m^2} - 1), \tag{6.19}$$

where we substituted $x = y - x_p$ in the second step. Numerical stability requires that the amplification factor $|e^{a_m \delta t}| \leq 1$ for all m in order for the amplitudes of all spatial Fourier modes of θ to be bounded. This requires

$$\left|1 + \frac{D\delta t}{\epsilon^2}(e^{-\epsilon^2 k_m^2} - 1)\right| \leq 1. \tag{6.20}$$

Since $\frac{D\delta t}{\epsilon^2}(e^{-\epsilon^2 k_m^2} - 1) \leq 0$ always, the above condition imposes that

$$\frac{D\delta t}{\epsilon^2}(e^{-\epsilon^2 k_m^2} - 1) \geq -2, \tag{6.21}$$

$$i.e., \;\; \delta t \leq \frac{2\epsilon^2}{D(1-e^{-\epsilon^2 k_m^2})} \quad \text{since} \quad -1 \leq (e^{-\epsilon^2 k_m^2} - 1) \leq 0. \tag{6.22}$$

This condition should be fulfilled for all modes m. The most limiting constraint for δt is obtained when $D(1 - e^{-\epsilon^2 k_m^2})$ has the largest possible absolute value. The largest possible value of $D(1 - e^{-\epsilon^2 k_m^2})$ is D when $k_m \to \infty$. Therefore, the condition ensuring that the amplitudes of all spatial Fourier modes are bounded is

$$\delta t \leq \frac{2\epsilon^2}{D}. \tag{6.23}$$

The above condition is a necessary and sufficient condition for our numerical scheme to be stable, making the numerical scheme in Eq. 6.8 conditionally stable.

6.3 Only advection

Setting $D = 0$ in Eq. 6.5, we obtain:

$$\frac{\partial u}{\partial t} = -\boldsymbol{v} \cdot (\boldsymbol{\nabla}\, u) - u(\boldsymbol{\nabla} \cdot \boldsymbol{v}). \tag{6.24}$$

Here, we assume that the flow field $\boldsymbol{v}$ is known analytically and therefore the analytical expression for $(\boldsymbol{\nabla} \cdot \boldsymbol{v})$ can be substituted in Eq. 6.24.
Using PSE (see [32] for details) and explicit Euler time stepping scheme:

$$u_p^{n+1} = u_p^n - \delta t \frac{\boldsymbol{v}_p}{\epsilon} \cdot \sum_{q=1}^{N} V_q (u_q^n + u_p^n) \boldsymbol{\eta}_\epsilon(\vec{x}_p - \vec{x}_q) - \delta t\, u_p (\boldsymbol{\nabla} \cdot \boldsymbol{v})_p, \tag{6.25}$$

where $\boldsymbol{v}_p$ is the velocity field at the location of particle p and $(\boldsymbol{\nabla} \cdot \boldsymbol{v})_p$ is the divergence of the velocity field at the location of particle p. The kernel $\boldsymbol{\eta}_\epsilon$ is a scalar in one-dimension and a vector in higher dimensions. It is given by:

$$\begin{aligned} \eta_\epsilon(x) &= \frac{-2x}{\epsilon^2\sqrt{\pi}} e^{-\frac{x^2}{\epsilon^2}} \qquad x \in \mathbb{R} \\ \boldsymbol{\eta}_\epsilon(\vec{x} = (x, y)) &= \left(\frac{-2x}{\epsilon^3 \pi} e^{-\frac{|\vec{x}|^2}{\epsilon^2}},\; \frac{-2y}{\epsilon^3 \pi} e^{-\frac{|\vec{x}|^2}{\epsilon^2}} \right) \qquad \vec{x} \in \mathbb{R}^2. \end{aligned} \tag{6.26}$$

Using the same procedure employed in Sec. 6.2.1, we find that the numerical scheme presented in Eq. 6.25 is unconditionally unstable. This unfavorable property is similar to those of finite difference difference schemes using central differences to approximate the advection operator.

6.3.1 Upwind PSE scheme

In order to overcome the unfavorable stability criteria of the numerical scheme in Eq. 6.25, we devise a scheme called a upwind scheme using PSE, analogous to upwind schemes in finite differences. According to the upwind scheme (written for two-dimensions where $\vec{x} = (x, y)$),

$$\begin{aligned} u_p^{n+1} = u_p^n &- \delta t \frac{\max(v_{p,x}, 0)}{\epsilon} \sum_{q; x_q < x_p} V_q (u_q^n + u_p^n) \eta_\epsilon^{L(1,0)}(\vec{x}_p - \vec{x}_q) \\ &- \delta t \frac{\max(v_{p,y}, 0)}{\epsilon} \sum_{q; y_q < y_p} V_q (u_q^n + u_p^n) \eta_\epsilon^{L(0,1)}(\vec{x}_p - \vec{x}_q) \\ &- \delta t \frac{\min(v_{p,x}, 0)}{\epsilon} \sum_{q; x_q > x_p} V_q (u_q^n + u_p^n) \eta_\epsilon^{R(1,0)}(\vec{x}_p - \vec{x}_q) \\ &- \delta t \frac{\min(v_{p,y}, 0)}{\epsilon} \sum_{q; y_q > y_p} V_q (u_q^n + u_p^n) \eta_\epsilon^{R(0,1)}(\vec{x}_p - \vec{x}_q), \\ &- \delta t \, u_p (\boldsymbol{\nabla} \cdot \boldsymbol{v})_p, \end{aligned} \tag{6.27}$$

where the kernels η_ϵ^L and η_ϵ^R are one-sided kernels. Specifically, η_ϵ^L denotes a left-sided kernel and η_ϵ^R is a right-sided kernel. The superscript $(1, 0)$ denotes one sided kernel along the x-direction and $(0, 1)$ denotes one-sided along the y-direction. For example, $\eta_\epsilon^{L(1,0)}$ denotes a left-sided kernel along the x-direction. One choice for these kernels are:

$$\begin{aligned} \eta_\epsilon^{L(1,0)}(x) &= \eta_\epsilon^{R(1,0)}(x) = \frac{-4x}{\epsilon^2 \sqrt{\pi}} e^{-\frac{x^2}{\epsilon^2}}, \qquad x \in \mathbb{R}. \\ \eta_\epsilon^{L(1,0)}(\vec{x}) &= \eta_\epsilon^{R(1,0)} = \frac{-4x}{\epsilon^3 \pi} e^{-\frac{|\vec{x}|^2}{\epsilon^2}}, \; \eta_\epsilon^{L(0,1)} = \eta_\epsilon^{R(0,1)} = \frac{-4y}{\epsilon^3 \pi} e^{-\frac{|\vec{x}|^2}{\epsilon^2}}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vec{x} \in \mathbb{R}^2. \end{aligned} \tag{6.28}$$

Order of accuracy. Using these kernels, the numerical scheme given in Eq. 6.27 to numerical solve Eq. 6.24 is first-order accurate in space and first-order accurate in time. For higher order kernels please refer [32].

6.3.1.1 Stability

We now investigate the numerical stability of Eq. 6.27 in one dimension. We again consider the general integral PSE formulation [32] using which

$$\begin{aligned} v\partial_x u(x) &\approx \frac{\max(v,0)}{\epsilon}\int_{-\infty}^{x}(u(y)+u(x))\eta_\epsilon^{L(1,0)}(x-y)dy \\ &+ \frac{\min(v,0)}{\epsilon}\cdot\int_{x}^{\infty}(u(y)+u(x))\eta_\epsilon^{R(1,0)}(x-y)dy \end{aligned} \tag{6.29}$$

where the kernel is given by Eq. 6.28. The scheme to numerically solve Eq. 6.24 in one dimension using explicit Euler time stepping and Eq. 6.29 to approximate the advection term is

$$\begin{aligned} u_p^{n+1} &= u_p^n - \delta t\frac{\max(v_p,0)}{\epsilon}\int_{-\infty}^{x_p}(u^n(y)+u_p^n)\eta_\epsilon^{L(1,0)}(x_p-y)dy \\ &- \delta t\frac{\min(v_p,0)}{\epsilon}\int_{x_p}^{\infty}(u^n(y)+u_p^n)\eta_\epsilon^{R(1,0)}(x_p-y)dy \\ &- \delta t\, u_p^n(\partial_x v)_p. \end{aligned} \tag{6.30}$$

Using Eq. 6.12, the error θ is given by

$$\begin{aligned} \theta_p^{n+1} &= \theta_p^n - \delta t\frac{\max(v_p,0)}{\epsilon}\int_{-\infty}^{x_p}(\theta_p^n(y)+\theta_p^n)\eta_\epsilon^{L}(x_p-y)dy \\ &- \delta t\frac{\min(v_p,0)}{\epsilon}\int_{x_p}^{\infty}(\theta_p^n(y)+\theta_p^n)\eta_\epsilon^{R}(x_p-y)dy \\ &- \delta t\, u_p^n(\partial_x v)_p. \end{aligned} \tag{6.31}$$

Using the von Neumann ansatz (Eq. 6.13), we get

$$\begin{aligned} e^{a_m\delta t} &= 1 - \delta t\frac{\max(v_p,0)}{\epsilon}\int_{-\infty}^{x_p}(e^{ik_m(y-x_p)}+1)\frac{(-4)(x_p-y)}{\epsilon^2\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{\epsilon^2}}dy \\ &- \delta t\frac{\min(v_p,0)}{\epsilon}\int_{x_p}^{\infty}(e^{ik_m(y-x_p)}+1)\frac{(-4)(x_p-y)}{\epsilon^2\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{\epsilon^2}}dy \\ &- \delta t\,(\partial_x v)_p. \end{aligned} \tag{6.32}$$

Without any loss of generality, we assume that flow velocity $v_p > 0$ and carry out the derivation to find the stability criteria. For $v_p > 0$,

$$\begin{aligned} e^{a_m\delta t} &= 1 - \delta t\frac{v_p}{\epsilon}\int_{x_p}^{\infty}(e^{ik_m(y-x_p)}+1)\frac{(-4)(x_p-y)}{\epsilon^2\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{\epsilon^2}}dy - \delta t\,(\partial_x v)_p, \\ &= (1-\delta t\,(\partial_x v)_p) - \frac{v_p\delta t}{\epsilon}\frac{4}{\sqrt{\pi}} + \frac{v_p\delta t}{\epsilon}P(k_m\epsilon) - i\frac{v_p\delta t}{\epsilon}Q(k_m\epsilon), \end{aligned}$$

with

$$\begin{aligned} P(x) &= xe^{\frac{-x^2}{4}}\,\mathrm{Erfi}\left(\frac{x}{2}\right), \\ Q(x) &= xe^{\frac{-x^2}{4}}, \end{aligned} \tag{6.33}$$

where Erfi is the imaginary error function. In order to enforce $|e^{a_m \delta t}| \leq 1$ we need

$$\left|(1-\delta t\,(\partial_x v)_p) - \frac{v_p \delta t}{\epsilon}\frac{4}{\sqrt{\pi}} + \frac{v_p \delta t}{\epsilon}\,P(k_m\epsilon) - i\frac{v_p \delta t}{\epsilon}Q(k_m\epsilon)\right| \leq 1,$$

$$i.e., \left[(1-\delta t\,(\partial_x v)_p) - \frac{v_p \delta t}{\epsilon}\frac{4}{\sqrt{\pi}} + \frac{v_p \delta t}{\epsilon}\,P(k_m\epsilon)\right]^2 + \left[\frac{v_p \delta t}{\epsilon}Q(k_m\epsilon)\right]^2 \leq 1,$$

$$\delta t \leq \frac{2(\partial_x v)_p + \frac{2v_p}{\epsilon}\left[\frac{4}{\sqrt{\pi}} - P(\alpha_m)\right]}{2[(\partial_x v)_p]^2 + \frac{2v_p(\partial_x v)_p}{\epsilon}\left[\frac{4}{\sqrt{\pi}} - P(\alpha_m)\right] + \frac{v_p^2}{\epsilon^2}\left\{\left[\frac{4}{\sqrt{\pi}} - P(\alpha_m)\right]^2 + [Q(\alpha_m)]^2\right\}} \quad (6.34)$$

where $\alpha_m = k_m\epsilon$. For any value of α_m, $P(\alpha_m)$ and $Q(\alpha_m)$ are bounded. We numerically determine that for any value of α_m, $0 \leq P(\alpha_m) \lesssim 1.45$ and analytically determine that $|Q(\alpha_m)| \leq \sqrt{2}e^{\frac{-1}{2}}$. Using these bounds and the possible values for v_p in a simulation, we can numerically determine the most restrictive upper bound for δt according to Eq. 6.34.

In case of incompressible flow ($(\partial_x v)_p = 0$ for all p), we can make use of the bounds for $P(\cdot)$ and $Q(\cdot)$ to compute an analytical upper bound for δt as:

$$\delta t \leq \frac{2\left[\frac{4}{\sqrt{\pi}} - P(\alpha_m)\right]}{\frac{v_p}{\epsilon}\left\{\left[\frac{4}{\sqrt{\pi}} - P(\alpha_m)\right]^2 + [Q(\alpha_m)]^2\right\}}. \quad (6.35)$$

The most restrictive upper bound for δt is when $|Q|$ is maximum ($= \sqrt{2}e^{\frac{-1}{2}}$) and $P = 0$. Therefore,

$$\delta t \leq \frac{4e}{2e+\sqrt{\pi}}\frac{\epsilon}{v_p} \approx 1.5\frac{\epsilon}{v_p}. \quad (6.36)$$

This is also referred to as the Courant-Levy-Friedrich condition. All Eulerian schemes have such an upper bound on δt which is of the form:

$$\delta t v < C\epsilon. \quad (6.37)$$

The value of C is specific to the discretization scheme used and the process simulated.

Combining the schemes derived in Secs. 6.2 and 6.3.1, we obtain the complete Eulerian particle method for numerically solving the advection-diffusion equation. The resulting scheme has an upper bound for the value of time-step size δt and is therefore conditionally stable.

The existence of the CFL condition is the biggest drawback of Eulerian methods, and it is addressed in the Lagrangian formulation as discussed next.

Chapter 7

Lagrangian Particle Methods for Field-based Models

In this chapter:

- Lagrangian particle methods
- Numerical scheme using Lagrangian particle methods
- Numerical stability of Lagrangian methods
- Numerical stability due to advection in Lagrangian particle methods
- Remeshing
- Particle to mesh moment-conserving interpolation schemes

Learning goals:

- Be able to devise Lagrangian numerical schemes
- Be able to decide when to use Lagrangian particle methods over Euler particle methods
- Know the advantages and disadvantages of Lagrangian particle methods

In this chapter, we develop Lagrangian particle methods. In contrast to Eulerian methods, the particle positions in Lagrangian particle methods change as a function of time. The general equation of motion governing Lagrangian particle methods is

$$\begin{aligned} \frac{\mathrm{d}\vec{x}_p(t)}{\mathrm{d}t} &= \vec{v}_p(t, \vec{x}(t), \vec{\omega}(t)) \\ \frac{\mathrm{d}\vec{\omega}_p(t)}{\mathrm{d}t} &= \vec{g}_p(t, \vec{x}(t), \vec{\omega}(t)), \end{aligned} \tag{7.1}$$

where $\vec{v}_p$ is the velocity of particle p and $\vec{g}_p$ is the property rate corresponding to the Lagrangian scheme.

7.1 Concept behind Lagrangian particle methods

Consider a general conservation law governing a scalar field u:

$$\frac{Du(\vec{x}(t))}{Dt} = G(u) - u(\boldsymbol{\nabla} \cdot v), \tag{7.2}$$

where $\frac{Du(\vec{x})}{Dt}$ is the Lagrangian time derivative (also referred to as the total time derivative, or material derivative) defined as

$$\frac{Du(\vec{x}(t))}{Dt} = \frac{\partial u}{\partial t} + \boldsymbol{v} \cdot (\boldsymbol{\nabla} u). \tag{7.3}$$

The quantity $G(u)$ is a rate function that can account for rate of change of u due to diffusion, reaction or other processes. The term $u(\boldsymbol{\nabla} \cdot \boldsymbol{v})$ is the rate of change of u due to compressibility of the velocity field. if the velocity field is incompressible ($\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$), this term does not contribute to any change in u. Mathematically, Eq. 7.2 can equivalently be written as two equations

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{v}, \tag{7.4}$$

$$\frac{\mathrm{d}u(\boldsymbol{x})}{\mathrm{d}t} = G(u) - u(\boldsymbol{\nabla} \cdot \boldsymbol{v}). \tag{7.5}$$

The interpretation of these equations is as follows. Consider a infinitesimally small packet of scalar quantity u at position $\boldsymbol{x}$. Eq. 7.4 is the equation of motion for the position of the packet. Eq. 7.5 is the rate of change of the scalar quantity carried by the moving packet. In summary, Eq. 7.5 is the equation of motion of the scalar quantity u of a infinitesimally small container at position $\boldsymbol{x}$ where the position $\boldsymbol{x}$ of the container changes according to Eq. 7.4. This is contrast to the equation of motion in the Eulerian frame of reference where the particles do not move (see Chapter 6) given by

$$\frac{\mathrm{d}u(\boldsymbol{x})}{\mathrm{d}t} = G(u) - u(\boldsymbol{\nabla} \cdot \boldsymbol{v}) - (\boldsymbol{v} \cdot \boldsymbol{\nabla} u). \tag{7.6}$$

Note, however, that the description in the Eulerian (Eq. 7.6) and the Lagrangian frames of reference (Eqs. 7.4 and 7.5) are mathematically equivalent and describe the same conservation law given by Eq. 7.2.

We will now develop Lagrangian particle methods to numerically solve a model equation. As in Chapter 6, as a model equation, we choose a general transport equation, namely one describing advection-diffusion (Eq. 6.5).

7.2 Advection-diffusion in the Lagrangian frame of reference

For the advection-diffusion process (Eq. 6.5), rate function $G(u)$ in Eq. 7.5 is the rate of change of concentration due to diffusion, and is equal to $D\nabla^2 u$. The advection-diffusion equation in the Lagrangian framework is therefore given by

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} &= \boldsymbol{v}, \\ \frac{\mathrm{d}u(\boldsymbol{x})}{\mathrm{d}t} &= D\nabla^2 u - u(\boldsymbol{\nabla}\cdot\boldsymbol{v}). \end{aligned} \tag{7.7}$$

By discretizing the spatial differential operators using any of the methods presented in Chapter 5, and using time-stepping schemes presented in Chapter 3, we can devise a family of Lagrangian particle methods for numerically solving Eq. 7.7. In the rest of this chapter, we will use PSE to discretize spatial differential operators and explicit Euler for time stepping to devise a particular instance of a Lagrangian particle method for numerically solving the advection-diffusion equation.

7.3 Lagrangian particle method for advection-diffusion

Applying PSE for discretizing spatial differential operators, and explicit Euler for time stepping the particle method in the Lagrangian frame of reference is given by

$$\begin{aligned} \vec{x}_p^{\,n+1} &= \vec{x}_p^{\,n} + \delta t\, \boldsymbol{v}_p^n, \\ u_p^{n+1} &= u_p^n + \delta t \frac{D}{\epsilon^2} \sum_{q=1}^{N} V_q (u_q^n - u_p^n)\eta_\epsilon(\vec{x}_p - \vec{x}_q) - \delta t u_p^n (\boldsymbol{\nabla}\cdot\boldsymbol{v})_p^n. \end{aligned} \tag{7.8}$$

Here a superscript n denotes the time step number corresponding to time $t = n\delta t$, δt being the time step size used in explicit Euler. $\vec{x}_p$ is the position of particle p and u_p is the scalar quantity carried by particle p. η_ϵ is the kernel used in PSE to discretize the Laplacian operator (Eq. 6.9). Here, we have considered a case where the flow field $\boldsymbol{v}$ is know analytically, so that $\boldsymbol{\nabla}\cdot\boldsymbol{v}$ can be computed analytically, and therefore does not have to be approximated through a spatial discretization scheme. In cases where $\boldsymbol{v}$ is not known analytically, $\boldsymbol{\nabla}\cdot\boldsymbol{v}$ can be discretized using PSE or DC-PSE as presented in Chapter 5.

Owing to the expression of the kernel η_ϵ given by Eq. 6.9, the Lagrangian particle method given by Eq. 7.8 is second-order accurate in space. In time, the explicit Euler scheme renders the method first-order accurate. Next, we study the stability condition for the numerical method given by Eq. 7.8.

7.3.1 Stability

As in Chapter 6, we restrict ourselves to a one-dimensional domain to carry out the stability analysis. As in Sec. 6.2.1, we rewrite the numerical scheme (Eq. 7.8) using the integral PSE formulation to approximate the Laplacian, and the expression for η_ϵ given by Eq. 6.9. Subsequently, the numerical scheme for one-dimension advection-diffusion is

$$\begin{aligned} x_p^{n+1} &= x_p^n + \delta t\, v_p^n, & (7.9)\\ u_p^{n+1} &= u_p^n + \frac{D\delta t}{\epsilon^2}\int_{-\infty}^{\infty}(u^n(y) - u_p^n)\frac{1}{2\epsilon\sqrt{\pi}}e^{-\frac{(x_p-y)^2}{4\epsilon^2}}dy \\ &\quad -\delta t u_p(\partial_x v)_p^n. & (7.10)\end{aligned}$$

Since the above two equations can be evaluated independently to compute the x_p and u_p at the next time point given the positions and property of all particles at the current time, we can carry out the linear stability analysis for each equation independently.

First, let us evaluate the stability criteria of Eq. 7.10. Using the same procedure presented in Chapter 6, we find that numerical stability of Eq. 7.10 requires

$$\delta t \leq \frac{1}{(D/\epsilon^2) + (\partial_x v)}. \tag{7.11}$$

Now, we evaluate the stability condition of Eq. 7.9. The trajectory of $x_p(t)$ according to Eq. 7.9 gives the streamlines of particles. Streamlines of particles are never allowed to cross, and this condition acts as the stability criterion. This criterion is ensured if the distance between any two particles p and q is always greater than zero. We therefore write the equation of motion for the distance $x_{pq} = x_p - x_q$ between particles p and q. Using Eq. 7.9, we find that

$$x_{pq}^{n+1} = x_{pq}^n + \delta t(v_p^n - v_q^n). \tag{7.12}$$

We now use Taylor expansion of $v_q = v(x_q)$ around $v_p = v(x_p)$, which is given by

$$v_q = v_p - x_{pq}\,(\partial_x v)_p + O(x_{pq}^2). \tag{7.13}$$

Using the above expression for v_q in Eq. 7.12, the equation reduces to:

$$\begin{aligned} x_{pq}^{n+1} &= x_{pq}^n + \delta t\, x_{pq}^n\,(\partial_x v)_p^n, \\ &= x_{pq}^n(1 + \delta t\,(\partial_x v)_p^n). \end{aligned}$$

Ensuring that particles never cross requires

$$\begin{aligned} &\left(1 + \delta t\,(\partial_x v)_p^n\right) > 0, \\ i.e., \quad &\delta t(\partial_x v)_p^n > -1. \end{aligned} \tag{7.14}$$

At a given time, in order to satisfy that every pair of particles do not cross

$$\delta t > 0, \quad \text{if } \min(\partial_x v) \geq 0, \tag{7.15}$$

$$\delta t < \frac{1}{\max(-\partial_x v)_p}, \text{if } \min(\partial_x v) < 0. \tag{7.16}$$

Here the min and max represents the minimum and maximum values over all particles in the entire computational domain. This condition for the time step is referred to as the Lagrangian CFL condition. In a two or three-dimensional domain, the Lagrangian CFL condition is given by

$$\delta t ||\boldsymbol{\nabla v}||_\infty \leq C, \tag{7.17}$$

where $\boldsymbol{\nabla v}$ is the velocity gradient matrix and $||\boldsymbol{\nabla v}||_\infty$ is the maximum norm of the matrix. It is worth noting that in contrast to the Eulerian CFL condition (Eq. 6.37), the Lagrangian CFL condition is potentially much less restrictive. Therefore, numerical stability of the scheme in Eq. 7.8 requires fulfilling Eqs. 7.11 and 7.17.

7.4 Remeshing

As mentioned in the previous section, the Lagrangian CFL condition (Eq. 7.17) compares favorably with the Eulerian CFL condition (Eq. 6.37). Moving particles, however, has the potential of violating the overlap condition required for SPH and PSE. Even for DC-PSE, moving particles can lead to situations where certain parts of the computational domain are not populated by particles, resulting in the value of the field quantities being unknown in those regions. In order to avoid this unfavorable scenario, Lagrangian particle methods employ remeshing, consisting of:

- interpolating the particle properties to a regular mesh. In the case of SPH or PSE, the spacing h of the regular mesh should be less than ϵ in order to satisfy the overlap condition.
- deleting the old set of particles, and
- creating new particles at the location of the non-zero mesh nodes, carrying the node weights as their new strengths.

Therefore, the crucial ingredient for remeshing are interpolation schemes to go from particles to a regular mesh.

7.4.1 Particle-Mesh Interpolation schemes

We require that the interpolation scheme does not introduce numerical artifacts. Therefore, we make use of moment-conserving interpolation schemes. The n-th moment of a function is again given by

$$M_n = \int_\Omega x^n f(x) \mathrm{d}x. \tag{7.18}$$

For example, the 0-th moment is the total area under the function. We require that when interpolating from particles to a mesh, as many moments of the function are conserved as the operator discretization conserves. In addition, we require that interpolation results in a smooth function on the mesh nodes.

These conditions are satisfied by the Λ interpolation kernels [37]. The Λ interpolation kernel $\Lambda_{n,m}$ conserves moments up to order n and produces a function that is $\mathcal{C}^m$ smooth. Kernels with larger values of n conserve more moments, and kernels with a larger value of m result in smoother functions.

Using the $\Lambda_{n,m}$ interpolation, the values on the mesh nodes are obtained as follows: Consider a regular mesh where each mesh node is indexed by a triplet (i, j, k) corresponding to coordinates $(i\delta x, j\delta y, k\delta z)$ in a three-dimensional domain with coordinates (x, y, z). The δx, δy and δz are the mesh spacings in the x, y, and z directions, respectively. Assume that particles carry strength ω_p. Given ω_p carried by the particles p, the values of the strength at the mesh nodes are computed as

$$\begin{aligned} &\omega(i,j,k) = \sum_p W_p(i,j,k)\omega_p, \\ &i = 0,\ldots,N_{\mathrm{x}}-1, \quad j = 0,\ldots,N_{\mathrm{y}}-1, \quad k = 0,\ldots,N_{\mathrm{z}}-1, \end{aligned} \tag{7.19}$$

where N_{x}, N_{y}, and N_{z} are the number of mesh modes in the x, y, and z directions, respectively. The function W_p is the weight (a multiplicative factor) of the property carried by particle p with location $\vec{x}_p = (x_p, y_p, z_p)$, contributing to the value of the property at location (j, i, k) on the mesh node. The weight is given by

$$W_p(i,j,k) = \Lambda_{n,m}\left(\left|\frac{x_p - i\delta x}{\delta x}\right|\right)\Lambda_{n,m}\left(\left|\frac{y_p - j\delta y}{\delta y}\right|\right)\Lambda_{n,m}\left(\left|\frac{z_p - k\delta z}{\delta z}\right|\right), \tag{7.20}$$

where $\Lambda_{n,m}$ is the interpolation kernel that conserves moments up to order n and results in a function that is $\mathcal{C}^m$ smooth, i.e., m times continuously differentiable.

In the following we present the expression of a couple of $\Lambda_{n,m}$ kernels for different values of n and m. The $\Lambda_{2,1}$ kernel is given by

$$\Lambda_{2,1}(s) = \begin{cases} 1 - \frac{5}{2}|s|^2 + \frac{3}{2}|s|^3, & 0 \le |s| < 1, \\ 2 - 4|s| + \frac{5}{2}|s|^2 - \frac{1}{2}|s|^3, & 1 \le |s| < 2, \\ 0 & 2 \le |s|. \end{cases} \tag{7.21}$$

for the normalized distance $s = \frac{x_p - i\delta x}{\delta x}$ and analogous for the other coordinate directions.

The $\Lambda_{4,4}$ kernel is given by

$$\Lambda_{4,4}(s) = \begin{cases} 1 - \frac{5}{4}|s|^2 + \frac{1}{4}|s|^4 - \frac{100}{3}|s|^5 \\ +\frac{455}{4}|s|^6 - \frac{295}{2}|s|^7 + \frac{345}{4}|s|^8 - \frac{115}{6}|s|^9, & 0 \le |s| < 1, \\ \\ -199 + \frac{5485}{4}|s| - \frac{32975}{8}|s|^2 + \frac{28425}{4}|s|^3 \\ -\frac{61953}{8}|s|^4 + \frac{33175}{6}|s|^5 \\ -\frac{20685}{8}|s|^6 + \frac{3055}{4}|s|^7 - \frac{1035}{8}|s|^8 + \frac{115}{12}|s|^9, & 1 \le |s| < 2, \\ \\ 5913 - \frac{89235}{4}|s| + \frac{297585}{8}|s|^2 \\ -\frac{143895}{4}|s|^3 + \frac{177871}{8}|s|^4 - \frac{54641}{6}|s|^5 \\ +\frac{19775}{8}|s|^6 - \frac{1715}{4}|s|^7 + \frac{345}{8}|s|^8 - \frac{23}{12}|s|^9, & 2 \le |s| < 3, \\ \\ 0 & 3 \le |s|. \end{cases} \tag{7.22}$$

As seen from the expression of the above kernels, they have compact support. As a consequence, particle p only contributes to the values of the mesh nodes in its vicinity. Therefore, the time complexity for interpolating particle property to mesh nodes is $O(N)$. However, as the number of moments required to be conserved and the degree of smoothness increases, the support of the kernels increase.

We also see from Eq. 7.20 that the interpolation weights are Cartesian products. That is, the kernel does not need to be evaluated for all possible combinations of i, j, and k. Instead, 1D interpolation weights can be computed independently in each direction and the results multiplied. For a kernel with a support of $\rho = \text{supp}(\Lambda)$ mesh nodes in each direction, the complexity of interpolation in d dimensions therefore is $O(d\rho)$ and not $O(\rho^d)$, which renders these kernels very efficient.

The same interpolation schemes can also be used to interpolate back from the mesh to particles, if that is required. For this, the roles $p \leftrightarrow (i, j, k)$ are simply swapped in the above equations. The interpolation kernels remain the same and anyway only depend on the absolute value of the normalized distance between a particle and a mesh node. Interpolating from the mesh to particles may be useful in hybrid discrete-continuous simulations in order to, e.g., interpolate forces back to the particles that represent discrete model entities. In simulations of field-based models, mesh-to-particle interpolation is almost never used, because it is easier there to just make new particles at the mesh nodes. If the particles however represent discrete model entities, like atoms or animals, they cannot be reallocated.

Chapter 8

Fast Algorithms for Far-field Interactions

If particles do not only interact with their neighbors, but with every other particle, the number of interactions to be computed becomes $O(N^2)$, which renders the simulation prohibitively expensive. A variety of algorithms is available to compute fast approximate solutions.

8.1 Hybrid Particle-Mesh Methods

8.1.0.1 Mesh-Particle Interpolation

8.1.0.2 Field solvers on the mesh

8.1.1 Lennard-Jones molecular dynamics with electrostatics

8.2 Fast Multipole Methods

8.3 Inverting the System Matrix

Chapter 9

Boundary Conditions

Basic idea is to modify the particle properties near the boundary such that they also fulfill the boundary conditions. This modification can be given by "ghost particles", adjusting the kernels to become one-sided [35], artificial boundary-reaction terms [34] or by solving another equation on the boundary [33].

9.1 Ghost particles: the Method of images

9.2 Immersed boundary methods and Penalization

Chapter 10

Particle Methods for Surfaces

10.1 Particle Level-Set Surface Representation

10.2 Particle Methods for Item-based Models

Moving along the surface and projecting back

10.3 Particle Methods for Field-based Models

10.3.1 Embedding schemes

10.3.2 Moving local frames

Chapter 11

Adaptive-resolution Particle Methods

11.1 Self organization

11.2 Particle-Particle interpolation

Chapter 12

Particle Methods on Parallel Computers

Bibliography

[1] W. of Ockham, *Quaestiones et decisiones in quattuor libros Sententiarum Petri Lombardi*. ed. Lugd., 1495.

[2] D. Frenkel and B. Smit, *Understanding Molecular Simulation. From Algorithms to Applications*. Academic Press, 2002.

[3] J. J. Monaghan, "Smoothed particle hydrodynamics," *Annu. Rev. Astron. Astrophys.*, vol. 30, pp. 543–574, 1992.

[4] J. Cardinale, G. Paul, and I. F. Sbalzarini, "Discrete region competition for unknown numbers of connected regions," *IEEE Trans. Image Process.*, vol. 21, no. 8, pp. 3531–3545, 2012.

[5] J. H. Walther and I. F. Sbalzarini, "Large-scale parallel discrete element simulations of granular flow," *Engineering Computations*, vol. 26, no. 6, pp. 688–697, 2009.

[6] J. Barnes and P. Hut, "A hierarchical O(Nlog$N$) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[7] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Phys.*, vol. 73, pp. 325–348, 1987.

[8] L. Greengard and V. Rokhlin, "The rapid evaluation of potential fields in three dimensions," *Lect. Notes Math.*, vol. 1360, pp. 121–141, 1988.

[9] R. W. Hockney, "The potential calculation and some applications," *Methods Comput. Phys.*, vol. 9, pp. 136–210, 1970.

[10] U. Trottenberg, C. Oosterlee, and A. Schueller, *Multigrid*. San Diego: Academic Press, 2001.

[11] M. M. Hejlesen, J. T. Rasmussen, P. Chatelain, and J. H. Walther, "A high order solver for the unbounded Poisson equation," *J. Comput. Phys.*, vol. 252, no. 458–467, 2013.

[12] L. Verlet, "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," *Phys. Rev.*, vol. 159, no. 1, pp. 98–103, 1967.

[13] G. Sutmann and V. Stegailov, "Optimization of neighbor list techniques in liquid matter simulations," *J. Mol. Liq.*, vol. 125, pp. 197–203, 2006.

[14] L. Devroye, *Non-uniform random variate generation*. Springer-Verlag New York, 1986.

[15] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 1958.

[16] D. T. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *J. Comput. Phys.*, vol. 22, pp. 403–434, 1976.

[17] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *J. Phys. Chem.*, vol. 81, no. 25, pp. 2340–2361, 1977.

[18] M. L. Hunt, "Discrete element simulations for granular material flows: effective thermal conductivity and self-diffusivity," *Int. J. Heat Mass Transfer*, vol. 40, no. 13, pp. 3059–3068, 1997.

[19] A. Daerr and S. Douady, "Two types of avalanche behaviour in granular media," *Nature*, vol. 399, pp. 241–243, 1999.

[20] P. G. de Gennes, "Granular matter: a tentative view," *Rev. Mod. Phys.*, vol. 71, no. 2, pp. 374–382, 1999.

[21] S. Douady, B. Andreotti, and A. Daerr, "On granular surface flow equations," *Eur. Phys. J. B*, vol. 11, pp. 131–142, 1999.

[22] S. Douady, B. Andreotti, A. Daerr, and P. Cladé, "From a grain to avalanches: on the physics of granular surface flows," *C. R. Physique*, vol. 3, pp. 177–186, 2002.

[23] S. Douady, A. Manning, P. Hersen, H. Elbelrhiti, S. Protière, A. Daerr, and B. Kabbachi, "Song of the dunes as a self-synchronized instrument," *Phys. Rev. Lett.*, vol. 97, p. 018002, 2006.

[24] L. E. Silbert, D. Ertaş, G. S. Grest, T. C. Halsey, D. Levine, and S. J. Plimpton, "Granular flow down an inclined plane: Bagnold scaling and rheology," *Phys. Rev. E*, vol. 64, p. 051302, 2001.

[25] L. E. Silbert, G. S. Grest, and S. J. Plimpton, "Boundary effects and self-organization in dense granular flows," *Phys. Fluids*, vol. 14, no. 8, pp. 2637–2646, 2002.

[26] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos, "PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems," *J. Comput. Phys.*, vol. 215, no. 2, pp. 566–588, 2006.

[27] O. Awile, O. Demirel, and I. F. Sbalzarini, "Toward an object-oriented core of the PPM library," in *Proc. ICNAAM, Numerical Analysis and Applied Mathematics, International Conference*, pp. 1313–1316, AIP, 2010.

[28] G.-H. Cottet and P. Koumoutsakos, *Vortex Methods – Theory and Practice*. New York: Cambridge University Press, 2000.

[29] P. Degond and S. Mas-Gallic, "The weighted particle method for convection-diffusion equations. Part 2: The anisotropic case," *Math. Comput.*, vol. 53, no. 188, pp. 509–525, 1989.

[30] P. Degond and S. Mas-Gallic, "The weighted particle method for convection-diffusion equations. Part 1: The case of an isotropic viscosity," *Math. Comput.*, vol. 53, no. 188, pp. 485–507, 1989.

[31] P. Degond and F.-J. Mustieles, "A deterministic approximation of diffusion equations using particles," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 2, pp. 293–310, 1990.

[32] J. D. Eldredge, A. Leonard, and T. Colonius, "A general deterministic treatment of derivatives in particle methods," *J. Comput. Phys.*, vol. 180, pp. 686–709, 2002.

[33] P. Koumoutsakos, A. Leonard, and F. Pépin, "Boundary conditions for viscous vortex methods," *J. Comput. Phys.*, vol. 113, pp. 52–61, 1994.

[34] N. Fiétier, O. Demirel, and I. F. Sbalzarini, "A meshless particle method for Poisson and diffusion problems with discontinuous coefficients and inhomogeneous boundary conditions," *SIAM J. Sci. Comput.*, vol. 35, no. 6, pp. A2469–A2493, 2013.

[35] B. Schrader, S. Reboux, and I. F. Sbalzarini, "Discretization correction of general integral PSE operators in particle methods," *J. Comput. Phys.*, vol. 229, pp. 4159–4182, 2010.

[36] B. Schrader, S. Reboux, and I. F. Sbalzarini, "Choosing the best kernel: performance models for diffusion operators in particle methods," *SIAM J. Sci. Comput.*, vol. 34, no. 3, pp. A1607–A1634, 2012.

[37] G.-H. Cottet, J.-M. Etancelin, F. Pérignon, and C. Picard, "High order semi-Lagrangian particle methods for transport equations: numerical analysis and implementation issues," *ESAIM: Mathematical Modelling and Numerical Analysis*, vol. 48, no. 4, pp. 1029–1060, 2014.

Index