DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2019

Explicit Symplectic Integrators for Non-Separable Hamiltonians in Molecular Dynamics

ANNA LASSEN

JONAS CONNERYD

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

DEGREE PROJECT IN TEKNIK,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2019

Explicita symplektiska integratorer för icke-separabla Hamiltonianer i molekyldynamik

ANNA LASSEN

JONAS CONNERYD

KTH ROYAL INSTITUTE OF TECHNOLOGY
SKOLAN FÖR TEKNIKVETENSKAP

Theoretical Physics

Explicit Symplectic Integrators for Non-Separable Hamiltonians in Molecular Dynamics

Jonas Conneryd, Anna Lassen
conneryd@kth.se
alassen@kth.se
SA114X Degree Project in Engineering Physics, First Level
Department of Theoretical Physics
Royal Institute of Technology (KTH)
Supervisor: Anatoly Belonoshko

3 juni 2019

Abstract

In molecular dynamics, mathematical models of metallic systems should in general have the temperature of the system as a dependent variable [1]. In particular, the potential energy term of the Hamiltonian function of the interaction model should be dependent on temperature in addition to interparticular distances. This puts the Hamiltonian function on a form which is generally non-separable. Conventional explicit numerical methods which are symplectic when used to integrate the equations of motion of systems with separable Hamiltonians are not in general symplectic when used to integrate the equations of motion of systems with a non-separable Hamiltonian. Hence, an integrator which sustains symplecticity when used in a system with non-separable Hamiltonian is sought. A family of explicit integrators which are symplectic when integrating systems with a non-separable Hamiltonian are shown to exhibit similar or superior performance to the Velocity Verlet and fourth-order Runge-Kutta schemes, albeit with the drawback of numerical instability when used on a system where forces depend exponentially on the inverted interparticular distances. To the knowledge of the authors, this study is the first time this family of integrators is applied in the context of molecular dynamics. The results of this study provide a first indication that a comprehensive solution to the problem of integrating the equations of motion of a system with a non-separable Hamiltonian explicitly and symplectically is not provided by the considered family of integrators. However, further investigations into using this family of integrators in other molecular dynamics systems than those investigated here are needed to provide a more definitive conclusion.

Sammanfattning

Inom molekyldynamik bör modeller av metalliska system i allmänhet ha systemets temperatur som en beroende variabel [1]. I synnerhet bör termen i systemets Hamiltonian som representerar potentiell energi utöver det interpartikulära avståndet även vara beroende av temperatur. Detta temperaturberoende gör i allmänhet Hamiltonianen icke-separabel. Konventionella explicita numeriska metoder som är symplektiska då de används på system med separabel Hamiltonian är i allmänhet inte symplektiska då de används i system med icke-separabel Hamiltonian. På grund av detta eftersöks en integrator som behåller symplekticitet då den används i system med en icke-separabel Hamiltonian. En samling integratorer som är symplektiska även då de används på system med icke-separabel Hamiltonian visas prestera lika bra eller bättre än de konventionella Velocity Verlet- och fjärde ordningens Runge Kutta-integratorerna, med nackdelen att de undersökta integratorerna uppvisar numerisk instabilitet då de tillämpas på system där de interpartikulära krafterna beror exponentiellt på inverterade interpartikulära avstånd. Till författarnas kännedom är denna studie den första tillämpningen av de undersökta integratorerna inom molekyldynamik. Resultaten i denna studie ger en fingervisning om att de undersökta integratorerna inte ger en övergripande lösning på problemet att integrera rörelseekvationerna hos ett system med icke-separabel Hamiltonian. Emellertid behövs vidare undersökningar som använder den undersökta samlingen av integratorer i andra system i molekyldynamik än de som undersöks i denna studie för att ge en mer definitiv slutsats.

Acknowledgements

The authors would like to thank our advisor Anatoly Belonoshko for his continuing support and guidance during this project.

Contents

Chapter 1

Introduction

In a short expository paper [1], G. Ackland states that for systems involving metals, it is desirable to eliminate the explicit treatment of electronic degrees of freedom, since doing so yields computational advantages of around six orders of magnitude. Ackland states that according to Sommerfeld theory, there exists a temperature-dependent contribution to the free electron energy in a metal, and as a result, any potential that eliminates explicit treatment of electronic degrees of freedom should be temperature dependent. This naturally leads to the question of which numerical methods are appropriate for integrating a system in which the interparticular potential is temperature-dependent. Desirable properties for such an integrator are, among others, it being explicit and symplectic. However, temperature dependence in an interparticular potential puts it on a form such that many properties of conventional integrators, such as symplecticity, may be lost when applied to such a potential. This report considers integrators which sustain symplecticity of a certain form when used to integrate temperature-dependent potentials and their performance compared to conventional integrators.

Chapter 2

Background Material

2.1. Molecular dynamics

2.1.1. Introduction

Molecular dynamics (MD) is a computer simulation method that is used to study the dynamics of atoms and molecules under a given set of interactions between them. Mathematically (and in reality, numerically), this is done by solving Newton's equations of motion using the Hamiltonian formalism, usually with the particles being approximated as point-like.

2.1.2. Hamiltonian systems

Consider a system of N point-like particles and denote the position and momentum of the i:th particle by q_i and p_i respectively. Furthermore, let q denote the vector of respective positions of all particles in the system and let p denote the vector of respective momenta of all particles in the system. Then Newton's equations take the following form:

$$m\frac{\mathrm{d}^2 q_i}{\mathrm{d}t^2} = \mathbf{F}_i;\ i = 1, \ldots, N, \tag{2.1}$$

or alternatively

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = \mathbf{F}_i;\ i = 1, \ldots, N. \tag{2.2}$$

In the case where the $\mathbf{F}_i$ of (2.1) have no explicit time dependence, that is, $\mathbf{F}_i = \mathbf{F}_i(p, q)$, Newton's equations of motion are invariant under time translation, and so Noether's theorem implies that there should exist a corresponding quantity in the system that is conserved in time [2]. This quantity turns out to be the *Hamiltonian* $H(p, q)$, which for an isolated system (which is the only type of system that is considered in this report) corresponds to the total energy of the system [4]. Then

$$H = T + V, \tag{2.3}$$

where T denotes kinetic energy and V denotes potential energy. The equations of motion of a system can be obtained through the Hamiltonian via Hamilton's equations [4]

$$\dot{p} = -\frac{\partial H}{\partial q} \qquad \dot{q} = \frac{\partial H}{\partial p}. \tag{2.4}$$

For a conservative force it turns out that

$$\mathbf{F}_i = -\frac{\partial H}{\partial q_i} = -\frac{\partial V}{\partial q_i} \tag{2.5}$$

since kinetic energy is independent of position [4].

2.1.3. Separable and non-separable Hamiltonians

In many systems of interest to scientists, the Hamiltonian can be written in the form

$$H(p, q) = T(p) + V(q). \tag{2.6}$$

If this is the case, the Hamiltonian of the system is said to be *separable* [6]. If the potential of a system also depends on the momenta of the particles and therefore cannot be put on the form (2.6), the Hamiltonian is said to be *non-separable*.

2.1.4. Periodic boundary conditions

When simulating a material using Molecular dynamics, the particles are put in a closed "container" with volume V to keep parameters such as particle density constant. In this report, the container consists of a three-dimensional cube with sides L. Often, due to computational constraints, the number of atoms in the system is so small that surface effects on the container walls dominate the overall behavior of the system. To remove surface effects, the approach often taken is to impose *periodic boundary conditions* (pbc) on the boundaries of the container. The process is described by J.M. Haile in [4] as follows: The volume V is denoted as the *primary cell* and is imagined to be surrounded by exact replicas of itself, denoted *image cells*, all containing the same number of atoms in the same positions relative to the respective image cell reference frames. Thus, all particles in the primary cell have an *image* in every image cell. The process is illustrated in 2 dimensions in Figure 2.1.

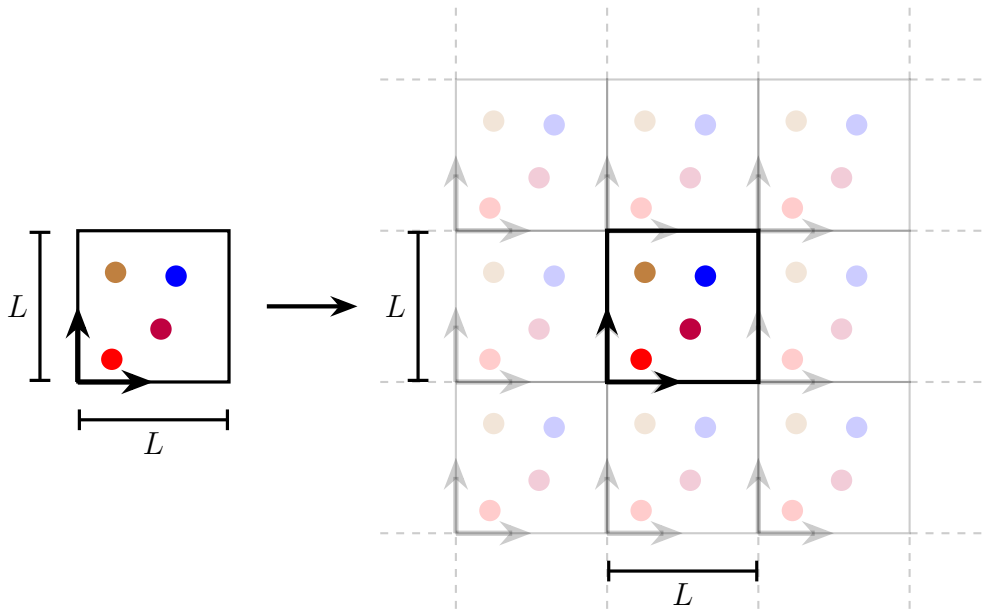

Figure 2.1: The primary cell is surrounded by image cells in which each image particle has the same position and momentum as in the original cell relative to each respective image cell's reference frame.

Since the momenta of the images are exactly the momenta of the particle in the primary cell, when a particle leaves the primary cell a new particle will enter it on the opposite boundary according to Figure 2.2.

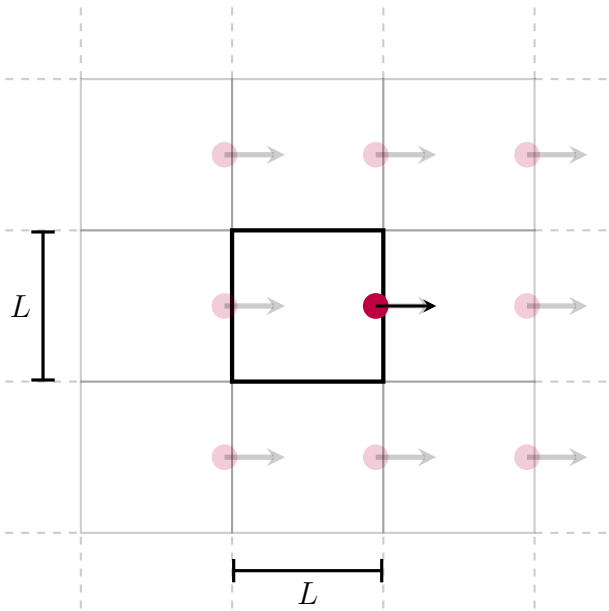

Figure 2.2: When a particle leaves the primary cell, its image will enter it on the opposite boundary.

The physical intuition behind pbc is assuming the volume V is a very small part of the whole material being investigated.

2.1.5. Conserved quantities

In [4], J.M. Haile outlines the quantities whose conservation is characteristic of a system of N particles with positions $q_i, i = 1, \dots N$ and momenta $p_i, i = 1, \dots, N$, and imposed periodic poundary conditions. The conserved quantities which are considered in this report are:

- Total energy, defined as
$$E = T + V.$$
- Total linear momentum, defined as
$$P = \sum_{i=1}^{N} p_i.$$
- Total angular momentum, defined as
$$L = \sum_{i=1}^{N} q_i \times p_i.$$

However, in reality, angular momentum is not a reliable metric of algorithm performance in molecular dynamics for reasons outlined in [11], and so only linear momentum and total energy are used as benchmarks for algorithm performance in this report.

2.2. Symplectic integrators and Hamiltonian mechanics

2.2.1. Introduction

To simulate the time evolution of a system using Molecular Dynamics, Newton's equations of motion (2.1) are numerically integrated in time using an *integrator*, which is a numerical scheme that approximates the solution to an ordinary differential equation (ODE). Integrators can inhibit a range of properties, one of which is the topic of this report, namely the integrator being *symplectic*. An integrator is symplectic if it conserves a certain map in *phase space*, in a sense to be made precise in this section.

2.2.2. Phase space

The natural setting in which to study symplecticity is the *phase space* of the system of interest, and in particular the phase space of each particle in the system. In [4], J.M. Haile defines phase space as such: For a system of N particles, the phase space S is a $6N$-dimensional space which is composed of two parts: A $3N$-dimensional *configuration space* in which each of the $3N$ coordinate axes are components of the q_i of each particle

(defined as in section 2.1.1) and a $3N$-dimensional *momentum space* in which each of the $3N$ coordinate axes are components of the p_i of each particle (also defined as in section 2.1.1). A point in the phase space S therefore corresponds to a point in the configuration space along with a point in the momentum space, which together represent a possible state (combination of positions and momenta) of the system [4].

2.2.3. Symplectic transformations

Consider a system with N particles moving in d dimensions. To each particle we can assign a $2d$-dimensional space (a d-dimensional configuration space combined with a d-dimensional momentum space) with entries in $\mathbb{R}$. The theory needed to define symplecticity and subsequent proofs of theorems concerning symplectic maps in the resulting space $\mathbb{R}^{2d}$ are documented by by Hairer et al. in [8]. Hairer et al. initially consider two vectors

$$\xi = \begin{bmatrix} \xi_p \\ \xi_q \end{bmatrix} \in \mathbb{R}^{2d}, \qquad \eta = \begin{bmatrix} \eta_p \\ \eta_q \end{bmatrix} \in \mathbb{R}^{2d}$$

where ξ_q, η_q and ξ_p, η_p denote the position and momentum components of ξ and η, respectively. The *oriented area* $A_{\mathrm{Or},P}(\xi, \eta)$ of the parallelogram P spanned by ξ and η is introduced by Hairer et al. for the case $d = 1$ through the map

$$A_{\mathrm{Or},P} = \det \begin{bmatrix} \xi_p & \eta_p \\ \xi_q & \eta_q \end{bmatrix} = \xi_p \eta_q - \xi_q \eta_p. \tag{2.7}$$

For the general case (and consequently, the case of a higher-dimensional parallellogram), Hairer et al. generalize this map by considering the *sum of the oriented areas of the projections of P onto each plane spanned by* (q_i, p_i) which Hairer et al. define by the bilinear map

$$\omega(\xi, \eta) := \sum_{i=1}^{d} \det \begin{bmatrix} \xi_{p,i} & \eta_{p,i} \\ \xi_{q,i} & \eta_{q,i} \end{bmatrix} = \sum_{i=1}^{d} \xi_{p,i}\eta_{q,i} - \xi_{q,i}\eta_{p,i}. \tag{2.8}$$

Using matrix notation Hairer et al. write (2.8) as

$$\begin{aligned} \omega(\xi, \eta) &= \xi^T J \eta, \\ J &= \begin{bmatrix} 0 & I_d \\ -I_d & 0 \end{bmatrix}, \end{aligned} \tag{2.9}$$

where I_d denotes the d-dimensional identity matrix. Using (2.8) and (2.9) Hairer et al. define a linear symplectic map as follows:

Definition 1. *A linear map* $A : \mathbb{R}^{2d} \longrightarrow \mathbb{R}^{2d}$ *is* symplectic *if*

$$A^T J A = J,$$

or equivalently if

$$\omega(A\xi, A\eta) = \omega(\xi, \eta), \ \forall \xi, \eta \in \mathbb{R}^{2d}.$$

Hairer et al. remark that the case $d = 1$ where $\omega(\xi, \eta)$ is the oriented area of the parallelogram P spanned by ξ and η, symplectic maps can be identified as *area preserving* with respect to P and similarly that a symplectic map generally preserves the sums of the areas of the projected parallelograms considered in (2.8).

Hairer et al. define a symplectic non-linear map as follows and motivate the definition by noting that non-linear differential maps can be locally approximated as linear.
Definition 2. *A differentiable map $g : U \longrightarrow \mathbb{R}^{2d}$ (where $U \subset \mathbb{R}^{2d}$ is open) is* symplectic *if the Jacobian matrix is everywhere symplectic, that is, if*

$$g'(p, q)^T J g'(p, q) = J$$

or equivalently if

$$\omega(g'(p, q)\xi, g'(p, q)\eta) = \omega(\xi, \eta), \ \forall \xi, \eta \in \mathbb{R}^{2d}.$$

2.2.4. Properties of symplectic maps in Hamiltonian mechanics

A fact that will be used in the report is that compositions of symplectic maps are symplectic [6].

In [8], Hairer et al. use the theory and definitions introduced in the previous section to further develop the theory of symplectic maps and provide connections to Hamiltonian mechanics. Hairer et al. introduce the *time-t flow* of a Hamiltonian system by the following definition:
Definition 3. *Consider a solution set $(p(t), q(t))$ of Hamilton's equations (2.4) with initial values $p(0) = p_0, q(0) = q_0$. The* time-t flow *ϕ_t of a Hamiltonian system is defined as the map that advances the solution by time: VAD ÄR U???*

$$\begin{aligned} \phi_t : U &\longrightarrow \mathbb{R}^{2d} \\ (p(t_0), q(t_0)) &\xmapsto{\phi_t} (p(t_0 + t), q(t_0 + t)). \end{aligned} \tag{2.10}$$

The theory in this section will ultimately show that the defining property of Hamiltonian systems are that their flow maps are symplectic maps. To this end, Hairer et al. introduce the concept of a differential equation being *locally Hamiltonian* through
Definition 4. *Let $U \subset \mathbb{R}^{2d}$ be open. A differential equation*

$$\dot{y} = f(y)$$

defined in U is locally Hamiltonian *if for every $y_0 \in U$ there exists a neighbourhood where*

$$f(y) = J^{-1} \nabla H(y)$$

for some function H.

To show why *locally Hamiltonian* is a proper name for this property, Hairer et al. define $y = (p, q)$, that is, y is a coordinate in phase space, and show that Hamilton's equations (2.4) can be written in the more suggestive way

$$\dot{y} = J^{-1} \nabla H(y) \tag{2.11}$$

where J was defined in (2.9) and $\nabla H(y) = H'(y)^T$. The fact that flow maps are in general symplectic maps is established by Hairer et al. in the following theorem.
Theorem 1 (Poincaré)**.** Let $H(p, q)$ be a twice continuously differentiable function on an open set $U \subset \mathbb{R}^{2d}$. Then, for each fixed t, the time-t flow ϕ_t of $H(p, q)$ is a symplectic map wherever it is defined.

In the theorem below, Hairer et al. establish that Hamiltonian systems are fully characterized by their flow maps being symplectic:
Theorem 2. Let $U \subset \mathbb{R}^{2d}$ be open. Let $f : U \longrightarrow \mathbb{R}^{2d}$ be continuously differentiable. Then the differential equation

$$\dot{y} = f(y)$$

is locally Hamiltonian if and only if its time-t flow $\phi_t(y)$ is symplectic for all $y \in U$ and for all sufficiently small t.

The following theorem will give another important property of flow maps:
Theorem 3 (Liouville)**.** Let y be defined as in (2.11). The time-t flow ϕ_t of a Hamiltonian system is volume-preserving in phase space; that is, for every bounded, open set $\Omega \in \mathbb{R}^{2d}$ and every t for which $\phi_t(y)$ is defined and for all $y \in \Omega$ we have

$$\int_{\Omega} \mathrm{d}y = \int_{\phi_t(\Omega)} \mathrm{d}y.$$

(In fact, this is true of every symplectic map).

For a proof of this theorem, see [9].

2.2.5. Integrators and symplectic maps

For a solution set (p, q) to Hamilton's equations in a phase space S with initial values (p_0, q_0) and a corresponding time-t flow map ϕ_t, it is possible to view a numerical one-step method as a map

$$\psi_t : S \longrightarrow S$$
$$\psi_t^n\left((p_0, q_0)\right) = (p_n, q_n) \approx (p(nt), q(nt))$$

that approximates the flow map ϕ_t of the system. Here, ψ_t^n denotes n-fold function composition of ψ_t with itself. The virtue of this approach is that it is then possible to discuss integrators of a system in terms of properties of flow maps. This fact leads naturally to the following definition, taken essentially verbatim from [8]:
Definition 5. *A numerical one-step method is called* symplectic *if the map*

$$y_1 = \psi_t(y_0)$$

is symplectic whenever the method is applied to an infinitely differentiable Hamiltonian system.

Viewed this way, a symplectic integrator has the properties outlined in section 2.2.4.. In addition, it is possible to show that a symplectic integrator nearly conserves the numerical Hamiltonian of a system over long time intervals under suitable conditions [7].

2.2.6. Extended phase space

A central notion in the Tao paper (see section 2.3) is *extended phase space*, proposed by Pihajoki in [12] building on ideas introduced by Hellström and Mikkola in [10]. Letting the phase space of the system of interest be denoted S, Pihajoki introduces the extended phase space of S as $S^2 = S \times S$. Using the coordinates $(q, \tilde{q}, p, \tilde{p})$ ((q, x, p, y) in Tao's notation, see section 2.3.), Pihajoki notes that the symplectic form (see (2.9)) in S^2 takes the form

$$J_{S^2} = J \otimes I_2 = \begin{bmatrix} 0 & -I_{2n} \\ I_{2n} & 0 \end{bmatrix} \tag{2.12}$$

where $\otimes$ denotes the Kronecker product. Symplecticity in the extended phase space S^2 is then defined analogously to in S (see Definition 1 and Definition 2) with J_{S^2} in place of J. Many further notions and ideas introduced by Pihajoki in [12] are used by Tao in [16] and so will be covered in section 2.3.

2.2.7. Explicit and implicit integrators

A short discussion by T. Sauer on the difference between explicit and implicit integrators is found in [15]. Sauer introduces *explicit* integrators as an integrator for which it is possible to write

$$y_{i+1} = y_i + f(y_i, t_i) \tag{2.13}$$

for some function f, when integrating some function y in time. Explicit integrators are called so to distinguish them from *implicit* integrators, which according to Sauer are of the form

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1}). \tag{2.14}$$

Instead of a function evaluation, a step forward in time for an implicit integrator consists of numerically solving the system (2.14), which Sauer notes can be done by Newton's method or similar methods. Since solving (2.14) is numerically demanding compared to evaluating (2.13), explicit integrators are desired in Molecular Dynamics where the systems of interest are often large and computationally demanding.

2.2.8. Note on symplectic integrators for separable Hamiltonians

A means of constructing arbitrary-order explicit symplectic integrators for separable Hamiltonians is provided by Yoshida in [17]. The case of separable Hamiltonians is therefore not of primary interest in this report.

2.3. The Tao paper

In [16], M. Tao presents a family of explicit integration methods for general non-separable Hamiltonians that are symplectic in extended phase space. Tao introduces (using ideas from [12]) the idea that for a given Hamiltonian $H(q, p)$, it is possible to introduce an extended phase space with the auxiliary variables (x, y) and consider an augmented

Hamiltonian

$$\bar{H}(q,p,x,y) := H(q,y) + H(x,p) + \omega\left(\frac{1}{2}\|q-x\|_2^2 + \frac{1}{2}\|p-y\|_2^2\right), \tag{2.15}$$

where $\|\cdot,\cdot\|_2^2$ denotes the square of the usual norm in R^n. $H(q,y)$ and $H(x,p)$ are copies of the original system, except that the auxiliary variables x, y are used instead of the usual q, p respectively, $\frac{1}{2}\|q-x\|_2^2 + \frac{1}{2}\|p-y\|_2^2$ is a constraint imposed on the system which further couples the extended and non-extended systems but with no physical significance, and w is a parameter that can be varied to increase or decrease the binding of the two copies of the initial system. For more compact notation, Tao defines

$$H_A := H(q,y), \qquad H_B := H(x,p), \qquad H_C := \frac{1}{2}\|q-x\|_2^2 + \frac{1}{2}\|p-y\|_2^2. \tag{2.16}$$

Using this notation, Tao writes the augmented Hamiltonian as

$$\bar{H}(q,p,x,y) = H_A + H_B + \omega H_C. \tag{2.17}$$

Tao notes that a feature of this construction is that Hamilton's equations in the original system and the extended Hamilton's equations using the augmented Hamiltonian, that is, the two initial value problems

$$\begin{aligned} \dot{Q} &= \partial_P H(Q,P); \qquad & Q(0) &= Q_0 \\ \dot{P} &= -\partial_Q H(Q,P); \qquad & P(0) &= P_0 \end{aligned} \tag{2.18}$$

and

$$\begin{aligned} \dot{q} &= \partial_p \bar{H}(q,p,x,y) &&= \partial_p H(x,p) + \omega(p-y); \qquad & q(0) &= Q_0 \\ \dot{p} &= -\partial_q \bar{H}(q,p,x,y) &&= -\partial_q H(q,y) - \omega(q-x); \qquad & p(0) &= P_0 \\ \dot{x} &= \partial_y \bar{H}(q,p,x,y) &&= \partial_y H(q,y) + \omega(y-p); \qquad & x(0) &= Q_0 \\ \dot{y} &= -\partial_x \bar{H}(q,p,x,y) &&= -\partial_x H(x,p) - \omega(x-q); \qquad & y(0) &= P_0, \end{aligned} \tag{2.19}$$

have the same solution in the sense that if $Q(t), P(t)$ solve the original system (2.18), putting

$$\begin{aligned} q(t) &= x(t) = Q(t), \\ p(t) &= y(t) = P(t) \end{aligned}$$

solves the augmented system (2.19) as well, and since $Q(t), P(t)$ are ODE:s, the existence and uniqueness theorem states that this solution is unique. Tao constructs the proposed integrator ϕ_l^δ, where l denotes the order of the numerical method, through a certain function composition of the respective exact time-δ flows of H_A, H_B and H_C. In this report, the second- and fourth-order version of ϕ_l^δ, that is, ϕ_2^δ, ϕ_4^δ are considered. Denote the exact time-δ flows of $H_A, H_B, \omega H_C$ as $\phi_{H_A}^\delta, \phi_{H_B}^\delta, \phi_{\omega H_C}^\delta$ respectively. These flow maps are symplectic in extended phase space. The exact expressions of each flow map are given

by Tao as

$$
\begin{aligned}
\phi^{\delta}_{H_A} &: \begin{bmatrix} q \\ p \\ x \\ y \end{bmatrix} \overset{\phi^{\delta}_{H_A}}{\longmapsto} \begin{bmatrix} q \\ p - \delta\partial_q H(q,y) \\ x + \delta\partial_y H(q,y) \\ y \end{bmatrix}, \\
\phi^{\delta}_{H_B} &: \begin{bmatrix} q \\ p \\ x \\ y \end{bmatrix} \overset{\phi^{\delta}_{H_B}}{\longmapsto} \begin{bmatrix} q + \delta\partial_p H(x,p) \\ p \\ x \\ y - \delta\partial_x H(x,p) \end{bmatrix}, \\
\phi^{\delta}_{\omega H_C} &: \begin{bmatrix} q \\ p \\ x \\ y \end{bmatrix} \overset{\phi^{\delta}_{\omega H_C}}{\longmapsto} \frac{1}{2} \begin{bmatrix} q + x + (q-x)\cos(2\omega\delta) + (p-y)\sin(2\omega\delta) \\ p + y + (p-y)\cos(2\omega\delta) - (q-x)\sin(2\omega\delta) \\ q + x - (q-x)\cos(2\omega\delta) - (p-y)\sin(2\omega\delta) \\ p + y - (p-y)\cos(2\omega\delta) + (q-x)\sin(2\omega\delta) \end{bmatrix}.
\end{aligned}
\tag{2.20}
$$

The second order integrator is then defined by Tao as

$$
\phi^{\delta}_{2} := \phi^{\delta/2}_{H_A} \circ \phi^{\delta/2}_{H_B} \circ \phi^{\delta}_{\omega H_C} \circ \phi^{\delta/2}_{H_B} \circ \phi^{\delta/2}_{H_A}, \tag{2.21}
$$

and the l-th order integrator as

$$
\phi^{\delta}_{l} := \phi^{\gamma_l \delta}_{l-2} \circ \phi^{(1-2\gamma_l)\delta}_{l-2} \circ \phi^{\gamma_l \delta}_{l-2} \tag{2.22}
$$

(referred to by Tao as "the triple jump"), where

$$
\gamma_l = \frac{1}{2 - 2^{1/(l+1)}}.
$$

Using the fact that compositions of symplectic maps are symplectic (as stated in subsection 2.2.4), Tao notes that ϕ^{δ}_{l} is symplectic in extended phase space. This integrator produces a discrete trajectory

$$
\begin{bmatrix} q_N \\ p_N \\ x_N \\ y_N \end{bmatrix} := \left(\phi^{\delta}_{l}\right)^N \begin{bmatrix} Q(0) \\ P(0) \\ Q(0) \\ P(0) \end{bmatrix}
$$

where $P(t), Q(t)$ are the exact solution of Hamilton's equations for the given Hamiltonian and initial conditions $P(0), Q(0)$ and where $q_N, x_N \approx Q(N\delta)$ and $p_N, y_N \approx P(N\delta)$. Tao provides some results on selecting ω if the system in question is integrable, but gives no fully systematic way of deciding appropriate ω for integrating a given Hamiltonian. Hence this problem is handled heuristically in this report.

Chapter 3

Investigation

3.1. Problem

The objective of this report was to gauge the performance of the proposed integrators outlined in section 2.3 when integrating two Molecular dynamical systems with non-separable Hamiltonians, and to compare their performance to those of the Velocity Verlet scheme [13] as well as the fourth-order Runge-Kutta scheme [14]. The comparison was made through examination of conservation of total energy and total linear momentum when using each of the integrators described above. A discussion of why these parameters are appropriate benchmarks for each of the integrators can be found in subsection 2.1.5.

3.2. Model and physics

3.2.1. Kinetic temperature

Using the results

$$\begin{aligned} PV &= \frac{1}{3}Nm\langle v^2\rangle, \\ PV &= Nk_BT \end{aligned} \tag{3.1}$$

from [3] (where P denotes pressure to distinguish from the momentum p) which are valid for an ideal gas, the *kinetic temperature* of a system of particles can be expressed as

$$T = \frac{1}{3Nk_B}\sum_i^N \frac{p_i^2}{2m_i} \tag{3.2}$$

This measure of temperature, renormalized to $k_B = 1$, was used in the temperature-dependent potentials. This is not entirely physically valid, since the particles in the considered system interacted via interparticular potentials and this measure of temperature is valid for ideal gases where there is no interparticular interaction. This measure of temperature mainly served as an effective means to make the potentials depend non-linearly on the momentum of the system in addition to positions, which made the Hamiltonians of each system non-separable and thus appropriate to stress-test the algorithms with.

3.2.2. Temperature-dependent spring potential

The first system considered in this report was a one-dimensional system of two particles with equal masses $m_1 = m_2 = m$ interacting via the spring-like potential

$$\begin{aligned} V(q_{12}, T) &= \frac{k(T)}{2}(q_1 - q_2 - x_0)^2, \\ k(T) &= k_0 \exp(-\beta T) \end{aligned} \tag{3.3}$$

where q_{12} denotes the interparticular distance, x_0 denotes the distance such that $V(x_0, T) = 0$, β is a parameter without physical significance, and the temperature T is defined as in (3.2). The (non-separable) Hamiltonian of the system was therefore

$$H(p, q) = \frac{p_1^2}{2m} + \frac{p_2^2}{2m} + \frac{k\left(T(p)\right)}{2}(q_1 - q_2 - x_0)^2, \tag{3.4}$$

By Hamilton's equations (2.4), the equation of motion for particle one in this system is

$$\begin{aligned} \dot{q}_1 &= \partial_{p_1} H(q, p) = \frac{p_1}{m}[1 - \frac{\beta k(p)}{2}(q_1 - q_2 - x_0)] \\ \dot{p}_1 &= -\partial_{q_1} H(q, p) = k(p)(q_1 - q_2 - x_0); \end{aligned} \tag{3.5}$$

3.2.3. Temperature-dependent Lennard-Jones

The second system considered in this report was a three-dimensional system consisting of N particles in a box with side length L and imposed periodic boundary conditions interacting through the interparticular Lennard-Jones potential

$$V_{LJ}(q_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{\|q_i - q_j\|} \right)^{12} - \left(\frac{\sigma}{\|q_i - q_j\|} \right)^{6} \right] \tag{3.6}$$

where q_{ij} denotes the distance between particles i and j, ε denotes the depth of the potential well, and σ denotes the interparticular distance such that $V_{LJ}(\sigma) = 0$ By summing over all contributions to the total potential energy of the system, The Lennard-Jones potential (3.6) gives rise to the Hamiltonian

$$H(p, q) = \sum_{i=1}^{N} \frac{p_i^2}{2m} + \sum_{1 \le i < j \le N} 4\varepsilon \left[\left(\frac{\sigma}{\|q_i - q_j\|} \right)^{12} - \left(\frac{\sigma}{\|q_i - q_j\|} \right)^{6} \right]. \tag{3.7}$$

To make (3.7) non-separable, a temperature dependence was introduced in ε through

$$\varepsilon \longrightarrow \varepsilon(T) := \varepsilon_0 \exp(-\beta T), \tag{3.8}$$

where β denotes a parameter with no physical significance. This substitution yielded the Hamiltonian

$$H(p, q) = \sum_{i=1}^{N} \frac{p_i^2}{2m} + \sum_{1 \le i < j \le N} 4\varepsilon(T) \left[\left(\frac{\sigma}{\|q_i - q_j\|} \right)^{12} - \left(\frac{\sigma}{\|q_i - q_j\|} \right)^{6} \right]. \tag{3.9}$$

β could then be varied to control the degree of "non-separability" of (3.9), since for small β, $\varepsilon(T) \approx \varepsilon_0$. By Hamilton's equations (2.4), the Hamiltonian (3.9) gives rise to the equations of motion

$$\begin{aligned} \dot{q}_i &= \partial_{p_i} H(q,p) = \frac{p_i}{m}\left[1 - \beta V(p,q)\right] \\ \dot{p}_i &= -\partial_{q_i} H(q,p) = 4\varepsilon(p) \sum_{j=1, j\neq i}^{N} \left(\frac{2\sigma^{12}}{\|q_i - q_j\|^{14}} - \frac{\sigma^6}{\|q_i - q_j\|^8} \right)(q_i - q_j). \end{aligned} \tag{3.10}$$

3.3. Simulation

3.3.1. Temperature-dependent spring constant

initially the particles were places at distance 1, both with the speed $\|v\| = 0.25$ directed away from each other. Simulations were preformed using the fourth-order Runge-Kutta scheme, Velocity Verlet scheme and the proposed integrator of orders 2 and 4, respectively. The simulations were performed using the parameters displayed in Table 3.1. The results of interest in this report were conservation of energy and conservation of momentum. In this report the error in total energy was calculated as

$$\Delta E = \frac{E(t) - E_0}{E_0} \tag{3.11}$$

where E denotes the total energy at time t and E_0 denotes $E_0 = E(0)$. The error in momentum Δp was calculated as

$$\Delta p = p(t) - p_0, \tag{3.12}$$

where p(t) denotes the momentum at time t and $p_0 = p(0)$.

3.3.2. Temperature-dependent Lennard-Jones potential

To save computing resources, the truncated Lennard-Jones potential

$$V(q_{ij}) = \begin{cases} 4\varepsilon \left[\left(\frac{\sigma}{\|q_i - q_j\|} \right)^{12} - \left(\frac{\sigma}{\|q_i - q_j\|} \right)^{6} \right], & \|q_i - q_j\| \leqslant r_c \\ 0, & \|q_i - q_j\| > r_c \end{cases} \tag{3.13}$$

dt	ω	β	k_0	x_0	m
0.01	7	0.5	2	1	2

Table 3.1: Parameters used in the temperature-dependent Lennard-Jones simulation.

which can be found in [5], where r_c is a cutoff distance often set to 2.5σ (so also in this report), was used in the simulation with the modification

$$\varepsilon = \varepsilon\left(T\right) = \varepsilon_0 \exp\left(-\beta T\right)$$

(see subsection 3.2.3) to yield

$$V(q_{ij}, T) = \begin{cases} 4\varepsilon_0 \exp\left(-\beta T\right) \left[\left(\frac{\sigma}{\|q_i - q_j\|} \right)^{12} - \left(\frac{\sigma}{\|q_i - q_j\|} \right)^{6} \right], & \|q_i - q_j\| \leqslant r_c \\ 0, & \|q_i - q_j\| > r_c \end{cases} \tag{3.14}$$

The N particles were placed in a three-dimensional box with side length L with imposed periodic boundary conditions. The initial positions and velocities of the particles were randomly generated (using a seeded random number generator to be able to reproduce results) and then distributed as to not place any particles too close to another and to fix the center of mass of the system by setting the total momentum of the system to 0. This was done using Algorithm 1 and Algorithm 2. The system was integrated a predetermined number of steps in time using the Velocity Verlet scheme [13], the fourth-order Runge-Kutta scheme [14], and the proposed integrator of order 2, respectively.

The proposed integrator of order 4 was not used since it is infeasibly slow for systems larger than a few particles; one step of the fourth-order Tao scheme is a composition of three steps of the second-order Tao scheme (see section 2.3), which in turn is a composition of five flow maps, each of which require one force evaluation, so one step of the fourth-order Tao scheme for one particle requires 15 force evaluations (compared to one force evaluation per step for the Velocity Verlet scheme and five force evaluations per step for the second-order Tao scheme). Since force evaluations are by far the most computationally expensive part of each integration step for large systems, the fourth-order Tao scheme could not be used for systems larger than a few particles.

The result of the simulations were plotted and compared on the basis of conservation of total energy and total linear momentum. To plot the energy error, the expression

$$\Delta E(t) = \frac{E(t) - E_0}{E_0}, \tag{3.15}$$

where $E(t)$ denotes the total energy of the system at time t and E_0 denotes the initial total energy of the system, was used as a measure of the energy error. To plot the error in momentum, the expressions

$$\Delta p_i(t) = p_i(t) - p_{i0}, \tag{3.16}$$

where $p_i(t)$ denotes the i:th component of the total momentum at time t and p_{i0} denotes the i:th component of the initial momentum, was used as a measure of the momentum error. The difference $p_i(t) - p_{i0}$ was not divided by p_{i0} analogously to in (3.15) since p_{i0} is initially set to 0, as stated above.

The parameters used in the simulation are compiled in Table 3.2.

β	N	dt	ω	Box side length	ε_0	σ	No. time steps
0.01	10	0.005	20	6σ	1	1	10 000

Table 3.2: Parameters used in the temperature-dependent Lennard-Jones simulation.

Algorithm 1: Initialize velocities

```
for i := 1 to N do
|   i:th velocity := Random components in range -1 to 1 in all directions
end
Total velocity := 0 ;
for i := 1 to N do
|   Total velocity += i:th velocity
end
for i := 1 to N do
|   i:th velocity -= (Total velocity)/N
end
```

Algorithm 2: Initialize positions

```
All Positions := Empty list ;
while # elements in All Positions < N do
|   New position := random components in range 0 to L in all directions;
|   forall Position ∈ All Positions do
|   |   if distance between Position and New Position < σ then
|   |   |   Re-randomize New Position
|   |   end
|   |   else
|   |   |   Add New Position to All Positions
|   |   end
|   end
end
```

3.4. Results

3.4.1. Temperature-dependent spring

Figure 3.1 shows the time evolution of the potential, kinetic and total energy in one simulation in the system described in subsection 3.2.2 when integrating with fourth order Runge-Kutta, Velocity Verlet and the proposed Tao schemes of orders two and four, respectively. Figure 3.2 compares the error in total energy and in total momentum.

3.4.2. Temperature-dependent Lennard-Jones potential

Figure 3.3 shows the time evolution of the total, potential and kinetic energies of the system described in subsection 3.3.2 when integrated using the Velocity Verlet scheme, fourth-order Runge Kutta scheme, and the second order Tao scheme, respectively. Figure 3.4 shows the errors in total energy and all three total linear momentum components of the system.

3.4.3. General takeaways

There was considerable difficulty involved in finding appropriate parameters for dt and ω when using the Tao scheme. Other time steps and values for ω than those presented in Table 3.2 would often result in numerical problems with particles moving too closely to each other during the Lennard-Jones simulation, with unphysically large interparticular forces as a result. Longer simulation times meant a larger probability of this problem surfacing, and so the longest recorded simulation time in which this problem did not occur was the time presented in Table 3.2. The Lennard-Jones simulation using the Tao scheme of order 2 took a considerably longer time to run than either of the Velocity-Verlet or fourth-order Runge-Kutta scheme-based simulations.

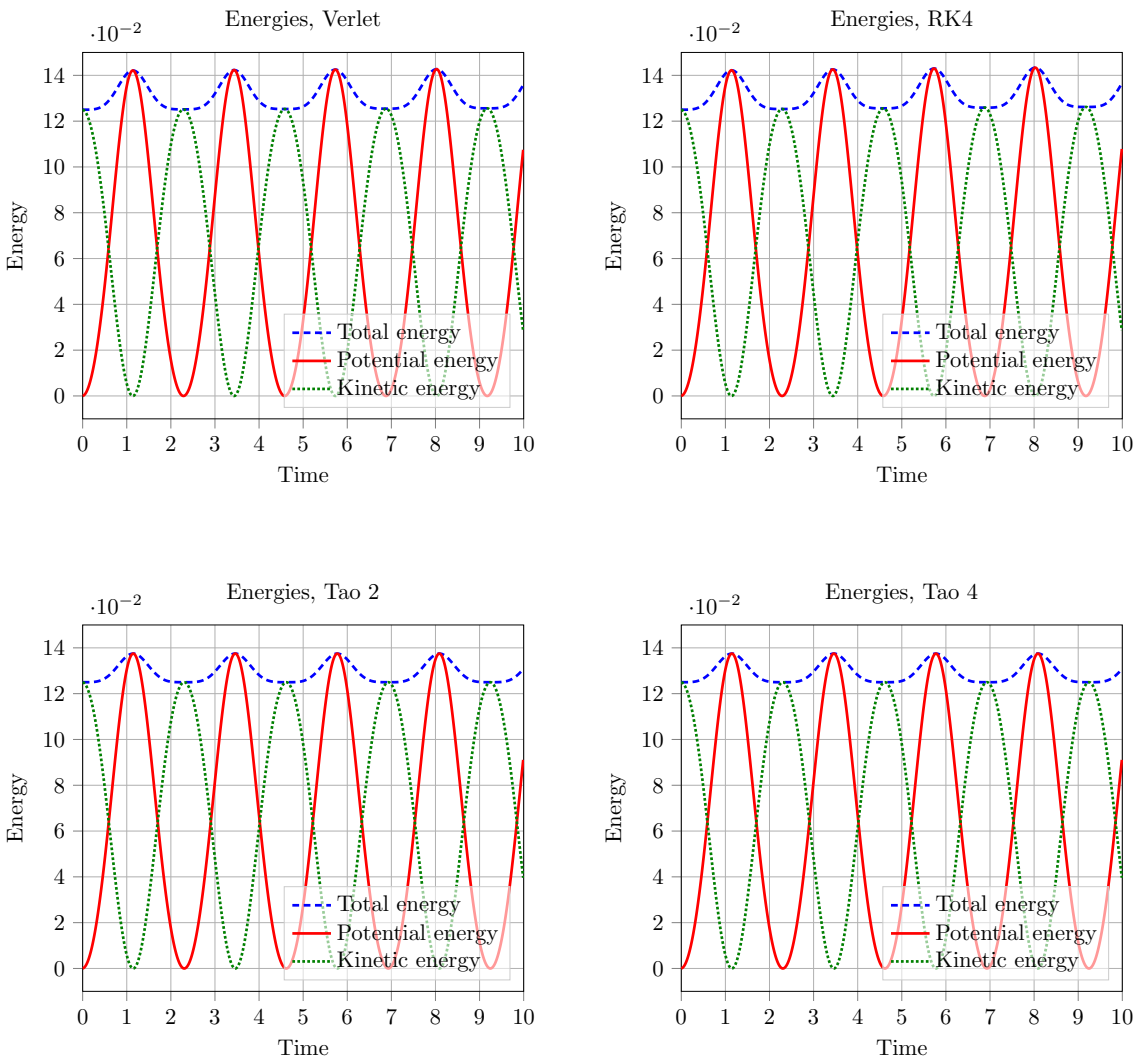

Figure 3.1: Time evolution of the total, potential and kinetic energies of the system described in subsection 3.3.2 when integrated using the Velocity Verlet scheme, fourth-order Runge Kutta scheme, and the second order Tao scheme, respectively. Parameters as in Table 3.1.

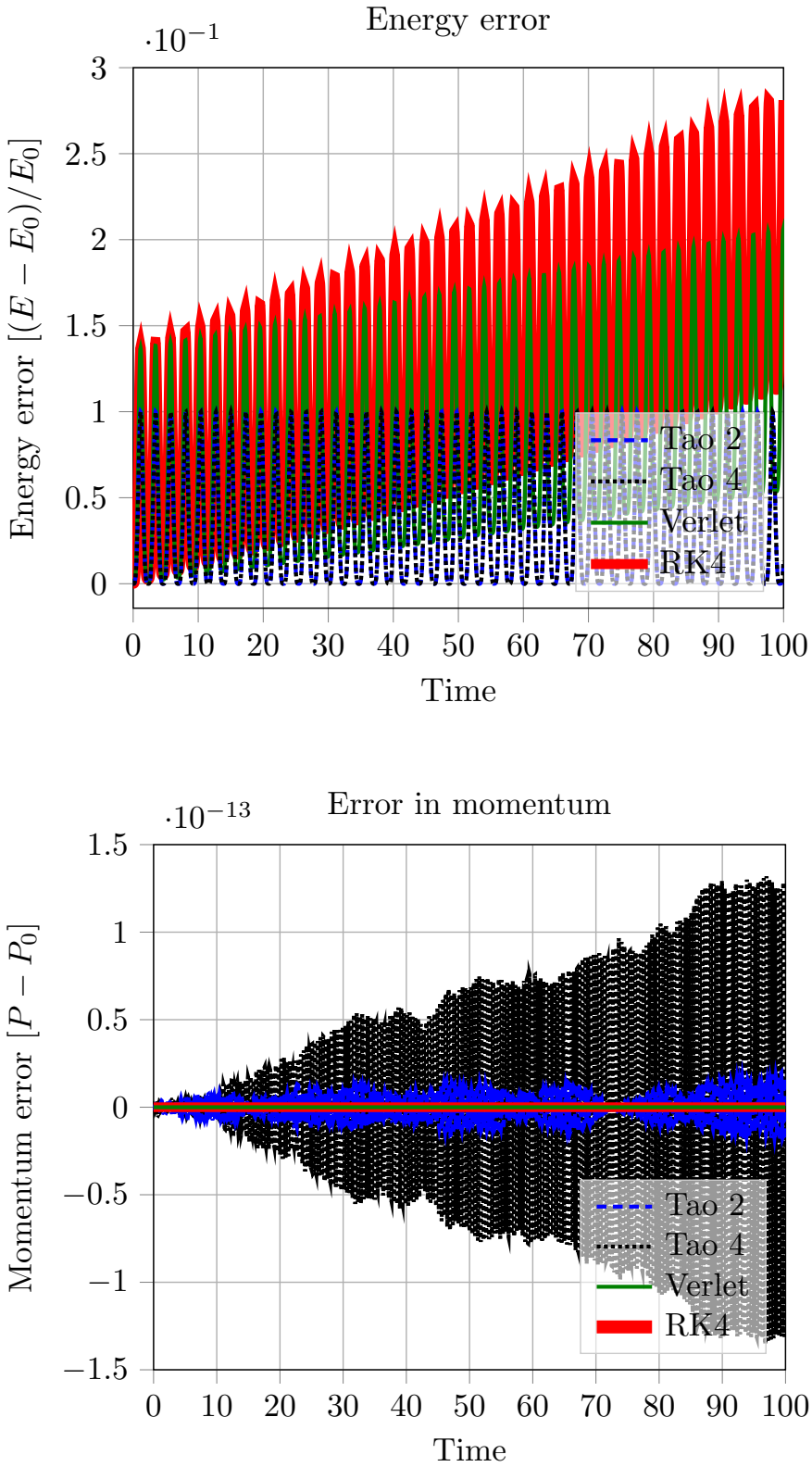

Figure 3.2: Error in total energy and in total momentum of the system simulated in subsection 3.3.1. Parameters as in Table 3.1.

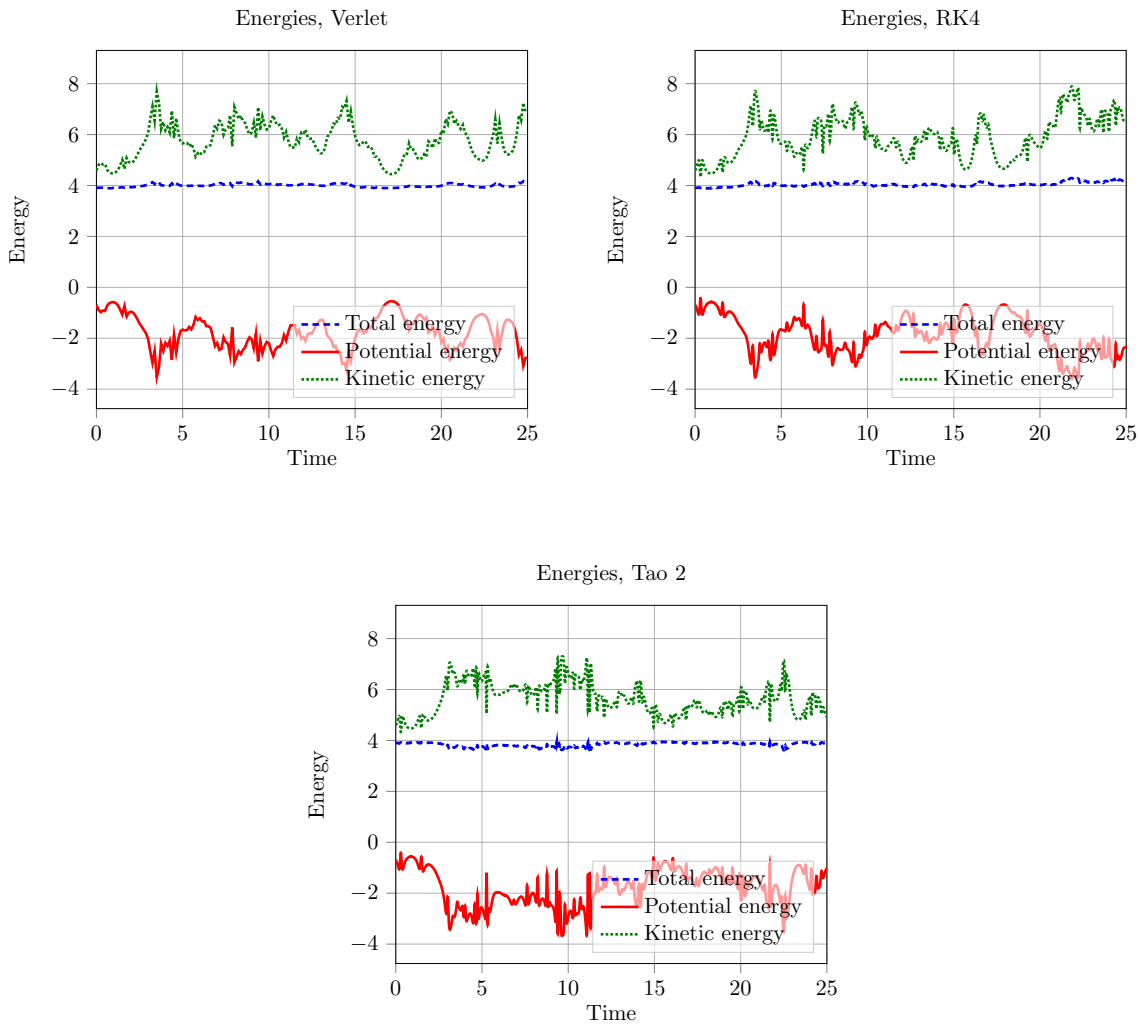

Figure 3.3: Time evolution of the total, potential and kinetic energies of the system described in subsection 3.3.2 when integrated using the Velocity Verlet scheme, fourth-order Runge Kutta scheme, and the second order Tao scheme, respectively. Parameters as in Table 3.2.

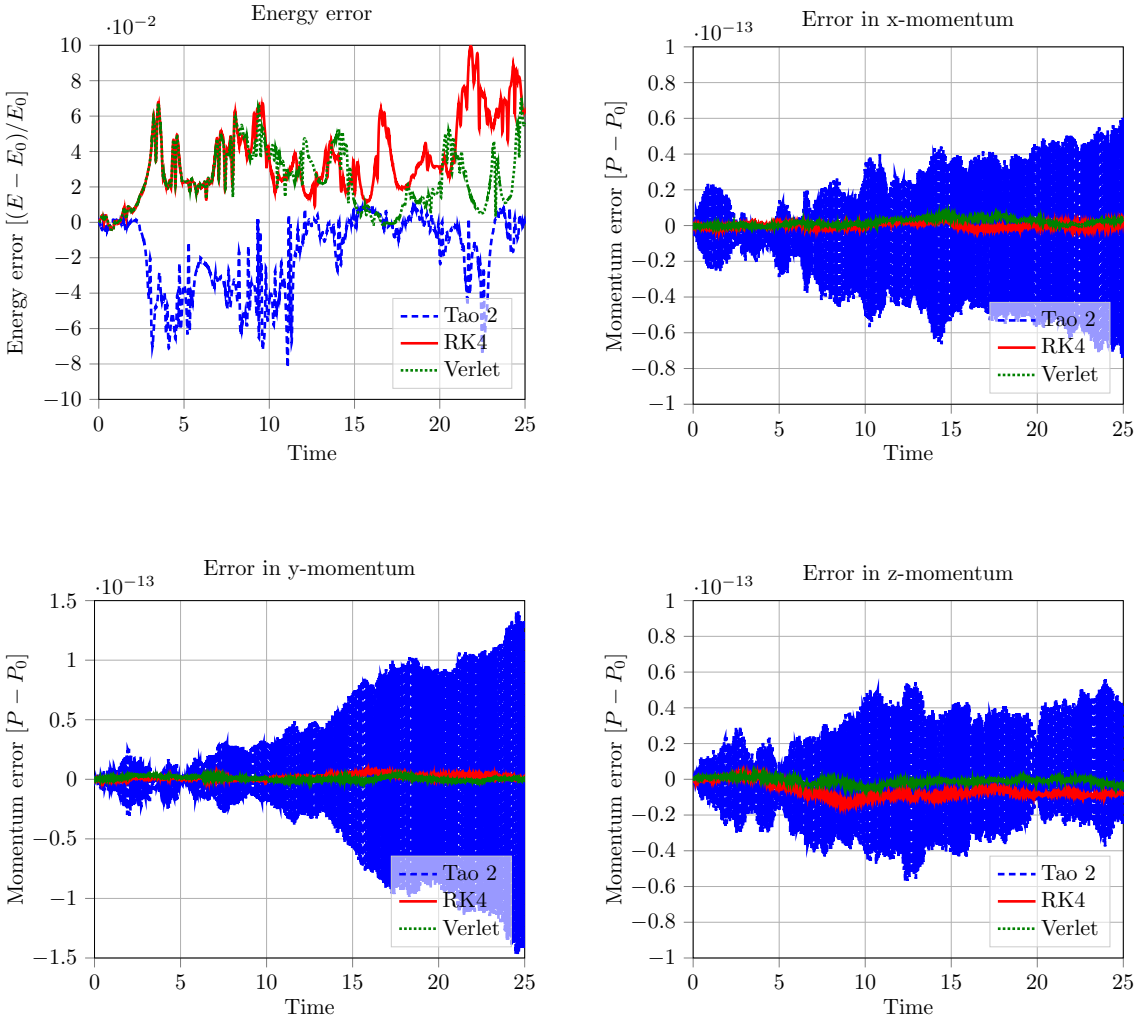

Figure 3.4: The errors in total energy and all three total linear momentum components of the system described in subsection 3.3.2. Parameters as in Table 3.2.

3.5. Discussion

3.5.1. Temperature-dependent spring potential

Conservaton of momentum

As seen in Figure 3.2, the Tao integrator did not conserve momentum as well as the Velocity Verlet and fourth-order Runge-Kutta schemes. The momentum error of the second-order Tao scheme oscillated around the true value of 0, while the momentum error of the fourth-order Tao scheme seemed to grow linearly with time. The momentum errors of both the Velocity Verlet and Runge-Kutta schemes were both constantly 0, which is likely to be an effect of the investigated system being small and symmetric.

Energy conservation

As noted in Figure 3.2, the Tao schemes of orders 2 and 4 preserved total energy during the whole simulation with the error oscillating arount the true energy while the energy error seemed to grow linearly in time when using both the Velocity Verlet and Runge-Kutta schemes. The errors of both the Velocity Verlet and Runge-Kutta schemes were also oscillatory in nature.

3.5.2. Temperature-dependent Lennard-Jones potential

Conservaton of momentum

As seen in Figure 3.4, the Tao integrator did not conserve momentum to the same degree as the Velocity Verlet and fourth-order Runge-Kutta schemes. The momentum error of the Tao scheme of order 2 seemed to oscillate around the true value of 0, with total momentum being approximately conserved in the mean. The magnitude of the oscillatory amplitude was small but seemed to grow with time. The reason for this behaviour is not immediately apparent, but the problem could possibly stem from the reciprocal actions of the extended and original systems.

Energy conservation

Again from Figure 3.4, the Tao scheme preserved total energy better than both the fourth-order Runge-Kutta and Velocity Verlet schemes. The difference was small but noticeable.

3.5.3. Selecting the parameter ω

The fact that there is no systematic way of establishing a proper value for the parameter ω when integrating a given system in time using the Tao scheme is a major drawback of using the Tao scheme. Many different values for ω had to be tried, with wildly varying quality of results, to establish the results in Figure 3.3 and Figure 3.1, which was time consuming since every simulation took a significant amount of time to run.

3.5.4. Choice of other parameters

The choices of the parameters outlined in Table 3.1 and Table 3.2 were largely based on heuristics and trial and error. It is entirely possible for other parameter choices to give differing results from those obtained in section 3.4. Variation of ingoing parameters in the simulations of subsection 3.3.2 and subsection 3.3.1 is a possible topic for further study.

3.5.5. Sensitivity to step size

In general, the Tao scheme demanded a smaller step size than the Velocity Verlet and RK4 schemes (regardless of the choice of the parameter ω) to not run into numerical problems arising from two particles being too close to each other (with unreasonably large interparticular forces as a result) during the Lennard-Jones simulation. This issue could possibly be resolved by using a variable step size during the simulation, but this was not attempted in this report.

3.5.6. Cumbersome implementation

The numerical implementation of the Tao scheme was very cumbersome compared to the implementation of the Velocity Verlet or RK4 schemes, which increases the risk of human errors when designing a Molecular dynamical simulation based on the Tao scheme. This issue is of course only relevant when building a simulation from scratch, and would be resolved by inclusion of the Tao scheme in Molecular dynamical software packages.

3.5.7. Numerical speed

The Tao schemes were considerably slower than the Velocity Verlet and fourth-order Runge-Kutta schemes for reasons outlined in subsection 3.3.2. This issue was more apparent when simulating the larger system of particles of subsection 3.3.2.

Chapter 4

Summary and Conclusions

4.1. Temperature-dependent spring potential

The errors in momentum of the Tao schemes of orders 2 and 4 seemed to oscillate around the true value of 0 during the simulation, with the amplitude of oscillation bounded in the case of the second-order Tao scheme and linearly increasing with time in the case of the fourth-order Tao scheme. The Tao schemes of orders 2 and 4 seemed to conserve total energy in the mean, while the Velocity Verlet and fourth-order Runge-Kutta schemes seemed to not conserve total energy over the course of the simulation. The Tao schemes of orders 2 and 4 were cumbersome to implement, but could be worthwile if conservation of total energy in the system is highly desirable.

4.2. Temperature-dependent Lennard-Jones potential

The error in momentum of the Tao scheme of order 2 oscillated to a higher degree than both the Velocity Verlet and fourth-order Runge-Kutta schemes, but total momentum seemed to be conserved in the mean. The Tao scheme of order 2 performed slightly better in conserving energy than the Velocity Verlet and fourth-order Runge-Kutta schemes. Again, the Tao scheme was cumbersome to implement and required a great deal of heuristics when selecting a value for the parameter ω in addition to small step sizes to not run into numerical issues during the simulation. Furthermore, it was very slow when integrating large systems since it requires multiple force evaluations per particle and integration step (see subsection 3.3.2). In summary, the second-order Tao scheme is not an appropriate choice of integrator when integrating systems containing forces whose magnitude, like that of the forces arising from an interparticular Lennard-Jones potential, grows exceedingly large if interparticular distance is small. However, the strong performance of the Tao schemes of order 2 and 4 in the temperature-dependent spring potential simulation gives reason to believe that there are Molecular dynamical systems in which the Tao schemes are sound choices of integrator.

Bibliography

[1] G J Ackland. "Temperature Dependence in Interatomic Potentials and an Improved Potential for Ti". In: *Journal of Physics: Conference Series* 402 (Dec. 20, 2012), p. 012001. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/402/1/012001. URL: http://stacks.iop.org/1742-6596/402/i=1/a=012001?key=crossref.576c8d9f03f16c2e11fae11b43e9f008 (visited on 03/26/2019).

[2] Mattias Blennow. "Lagrangian Mechanics". In: *Mathematical Methods for Physics and Engineering*. Boca Raton: CRC Press, Nov. 21, 2017, pp. 615–631. ISBN: 978-1-138-05690-9.

[3] Stephen Blundell and Katherine M. Blundell. "The Ideal Gas Law". In: *Concepts in Thermal Physics*. 2nd ed. OCLC: ocn430497029. Oxford ; New York: Oxford University Press, 2010. ISBN: 978-0-19-956209-1 978-0-19-956210-7.

[4] J.M. Haile. "Fundamentals". In: *Molecular Dynamics Simulation : Elementary Methods*. New York: Wiley, 1992, pp. 38–103. ISBN: 0-471-81966-2.

[5] J.M. Haile. "Soft Spheres". In: *Molecular Dynamics Simulation : Elementary Methods*. New York: Wiley, 1992, pp. 188–191. ISBN: 0-471-81966-2.

[6] E. Hairer, Christian Lubich, and Gerhard Wanner. "II.5 Splitting Methods". In: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Springer Series in Computational Mathematics 31. OCLC: ocm69223213. Berlin ; New York: Springer, 2006, p. 48. ISBN: 978-3-540-30663-4.

[7] E. Hairer, Christian Lubich, and Gerhard Wanner. "IX.8 Long-Time Energy Conservation". In: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Springer Series in Computational Mathematics 31. OCLC: ocm69223213. Berlin ; New York: Springer, 2006, p. 367. ISBN: 978-3-540-30663-4.

[8] E. Hairer, Christian Lubich, and Gerhard Wanner. "VI.2 Symplectic Transformations". In: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Springer Series in Computational Mathematics 31. OCLC: ocm69223213. Berlin ; New York: Springer, 2006, pp. 182–187. ISBN: 978-3-540-30663-4.

[9] E. Hairer, Christian Lubich, and Gerhard Wanner. "VI.9 Volume Preservation". In: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd ed. Springer Series in Computational Mathematics 31. OCLC: ocm69223213. Berlin ; New York: Springer, 2006, pp. 227–233. ISBN: 978-3-540-30663-4.

[10] Christian Hellström and Seppo Mikkola. "Explicit Algorithmic Regularization in the Few-Body Problem for Velocity-Dependent Perturbations". In: *Celestial Mechanics and Dynamical Astronomy* 106.2 (Feb. 2010), pp. 143–156. ISSN: 0923-2958, 1572-9478. DOI: 10.1007/s10569-009-9248-8. URL: http://link.springer.com/10.1007/s10569-009-9248-8 (visited on 04/13/2019).

[11] V.A. Kuzkin. "On Angular Momentum Balance for Particle Systems with Periodic Boundary Conditions". In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 95.11 (Nov. 2015), pp. 1290–1295. ISSN: 00442267. DOI: 10.1002/zamm.201400045. URL: http://doi.wiley.com/10.1002/zamm.201400045 (visited on 04/24/2019).

[12] Pauli Pihajoki. "Explicit Methods in Extended Phase Space for Inseparable Hamiltonian Problems". In: *Celestial Mechanics and Dynamical Astronomy* 121.3 (Mar. 2015), pp. 211–231. ISSN: 0923-2958, 1572-9478. DOI: 10.1007/s10569-014-9597-9. URL: http://link.springer.com/10.1007/s10569-014-9597-9 (visited on 04/04/2019).

[13] William C. Swope et al. "A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters". In: *The Journal of Chemical Physics* 76.1 (Jan. 1982), pp. 637–649. ISSN: 0021-9606, 1089-7690. DOI: 10.1063/1.442716. URL: http://aip.scitation.org/doi/10.1063/1.442716 (visited on 04/28/2019).

[14] Tim Sauer. "6.4 Runge-Kutta Methods and Applications". In: *Numerical Analysis*. 2nd ed. OCLC: ocn725295545. Boston: Pearson, 2012, p. 316. ISBN: 978-0-321-78367-7.

[15] Tim Sauer. "6.6. Implicit Methods and Stiff Equations". In: *Numerical Analysis*. 2nd ed. OCLC: ocn725295545. Boston: Pearson, 2012. ISBN: 978-0-321-78367-7.

[16] Molei Tao. "Explicit Symplectic Approximation of Nonseparable Hamiltonians: Algorithm and Long Time Performance". In: *Physical Review E* 94.4 (Oct. 10, 2016). ISSN: 2470-0045, 2470-0053. DOI: 10.1103/PhysRevE.94.043303. URL: https://link.aps.org/doi/10.1103/PhysRevE.94.043303 (visited on 02/22/2019).

[17] Haruo Yoshida. "Construction of Higher Order Symplectic Integrators". In: *Physics Letters A* 150.5-7 (Nov. 1990), pp. 262–268. ISSN: 03759601. DOI: 10.1016/0375-9601(90)90092-3. URL: http://linkinghub.elsevier.com/retrieve/pii/0375960190900923 (visited on 02/27/2019).

Chapter 5

Appendix 1: Python code used in the simulations

5.1. Temperature-dependent spring simulation

```
import matplotlib.pyplot as plt
import numpy as np
import copy
from matplotlib2tikz import save as tikz_save
from collections import OrderedDict
from cycler import cycler

linestyles = OrderedDict(
    [('solid',                (0, ())),
     ('loosely dotted',       (0, (1, 10))),
     ('dotted',               (0, (1, 5))),
     ('densely dotted',       (0, (1, 1))),

     ('loosely dashed',       (0, (5, 10))),
     ('dashed',               (0, (5, 5))),
     ('densely dashed',       (0, (5, 1))),

     ('loosely dashdotted',   (0, (3, 10, 1, 10))),
     ('dashdotted',           (0, (3, 5, 1, 5))),
     ('densely dashdotted',   (0, (3, 1, 1, 1))),

     ('loosely dashdotdotted', (0, (3, 10, 1, 10, 1, 10))),
     ('dashdotdotted',         (0, (3, 5, 1, 5, 1, 5))),
     ('densely dashdotdotted', (0, (3, 1, 1, 1, 1, 1)))])


plt.rc('axes', prop_cycle=(cycler('color', [ 'b', 'r', 'g', 'k'
   ]) +
```

```
                                cycler('linestyle', [linestyles['
                                   densely dotted'], '-', '-',  '-'])
                                   ) + cycler('lw', [3, 3, 1.5, 3]))

class Body:
    """Particle with mass, position and velocity"""
    def __init__(self, name, mass, pos, vel):
        self.name = name      #Name
        self.mass = mass      #Mass
        self.pos = pos        #Position
        self.vel = vel        #Velosity

        #Extended system:
        self.pos2 = pos
        self.vel2 = vel

class Spring:
    """Spring with equilibrium and spring constant"""
    def __init__(self, equilibrium, springConst, separable,
       modelConst):
        self.L0 = equilibrium
        self._b = modelConst
        self._k0 = springConst
        self._sep = separable

    def k(self, syst, extended = False):
        """Returns the spring constant"""
        if self._sep:
            return self._k0    #Spring konstant for a separable
               system
        if not extended:
            p1 = syst[0].vel * syst[0].mass
            p2 = syst[1].vel * syst[1].mass
        elif extended:
            p1 = syst[0].vel2 * syst[0].mass
            p2 = syst[1].vel2 * syst[1].mass
        return self._k0 * np.exp( - self._b * ( p1**2 + p2**2 )
           )

def kineticEnergy(syst):
    """Calculates and returns the kinetic energy of the system
       """
    kinen = 0
    for particle in syst:
        kinen += particle.vel**2 * particle.mass / 2
```

```
    return kinen

def potentialEnergy(syst, spring):
    """Returns the potential of the system"""
    L0 = spring.L0                          #Equilibrium for the
        spring potential
    x = syst[0].pos - syst[1].pos - L0      #
    return spring.k(syst) * x**2 / 2

def vel_nonsep(body, syst, spring, extended = False):
    """Calculates and returns dH/dq for the system"""
    L0 = spring.L0    #equilibrium of the spring
    m = body.mass     #mass

    if not extended:
        #Om vi uppdaterar q vill vi ha += dH(x,p)/dp
        v1 = syst[0].vel
        v2 = syst[1].vel
        q1 = syst[0].pos2
        q2 = syst[1].pos2
        k = spring.k(syst)

    elif extended:
        #Om vi uppdaterar x vill vi ha += dH(q,y)/dp
        v1 = syst[0].vel2
        v2 = syst[1].vel2
        q1 = syst[0].pos
        q2 = syst[1].pos
        k = spring.k(syst, extended = True)

    if body.name == 'part1':
        return v1 * ( 1 - spring._b * k * (q1 - q2 - L0) ** 2 /
            2 ) #SPRING

    if body.name == 'part2':
        return v2 * ( 1 - spring._b * k * (q1 - q2 - L0) ** 2 /
            2) #SPRING

def forceOn(body, syst, spring, extendedMomentum = False,
    extendedPosition = False):
    """Returns the force on a particle"""
    L0 = spring.L0
    if not extendedMomentum:
        k = spring.k(syst)
    elif extendedMomentum:
```

```
        k = spring.k(syst, extended = True)
    if not extendedPosition:
        #Om vi uppdaterar p vill vi ha += dH(q,y)/dq
        if body.name == 'part1':
            x = - ( body.pos - syst[1].pos - L0 )
        elif body.name == 'part2':
            x = syst[0].pos - body.pos - L0
        return k * x
    elif extendedPosition:
        #Om vi uppdaterar y vill vi ha += dH(x,p)/dx
        if body.name == 'part1':
            x = - ( body.pos2 - syst[1].pos2 - L0 )
        if body.name == 'part2':
            x = syst[0].pos2 - body.pos2 - L0
        return k * x


def RK4(syst, spring, dt):
    """ """
    initPositions = [body.pos for body in syst]

    a_1s = [forceOn(body, syst, spring) / body.mass * dt for
       body in syst]
    b_1s = [ body.vel * dt for body in syst]

    for i in range(len(syst)):
        syst[i].pos = initPositions[i] + b_1s[i] * 1/2

    a_2s = [ ]
    a_3s = [ ]
    a_4s = [ ]

    b_2s = [ ]
    b_3s = [ ]
    b_4s = [ ]

    counter = 0
    for body in syst:
        a_2s.append(forceOn(body, syst, spring) / body.mass * dt
           )
        b_2s.append((body.vel + a_1s[counter] / 2) * dt)
        counter += 1

    for j in range(len(syst)):
        syst[j].pos = initPositions[j] + b_2s[j] * 1/2
```

```
    counter = 0
    for body in syst:
        a_3s.append(forceOn(body, syst, spring)/ body.mass * dt)
        b_3s.append((body.vel + a_2s[counter] / 2) * dt)
        counter += 1

    for j in range(len(syst)):
        syst[j].pos = initPositions[j] + b_3s[j]

    counter = 0
    for body in syst:
        a_4s.append(forceOn(body, syst, spring) / body.mass * dt
            )
        b_4s.append((body.vel + a_3s[counter]) * dt)
        counter += 1

    for j in range(len(syst)):
        syst[j].vel = syst[j].vel + (a_1s[j] + 2*a_2s[j] + 2*
            a_3s[j] + a_4s[j]) / 6
        syst[j].pos = initPositions[j] + (b_1s[j] + 2*b_2s[j] +
            2*b_3s[j] + b_4s[j]) / 6

def verlet(syst, spring, dt):
    """One step for the system using velocity verlet"""
    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel += ( forceOn(body, systCopy, spring) / body.
            mass ) * (dt / 2)
        body.pos += body.vel * dt
    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel += ( forceOn(body, systCopy, spring) / body.
            mass ) * (dt / 2)

def tao(syst, spring, dt):
    """One step for the system using Molei Taos method"""

    omega = 7  #Parameter in molei taos method

    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel += ( forceOn(body, systCopy, spring,
            extendedMomentum = True) / body.mass ) * ( dt / 2 )
```

```
        body.pos2 += vel_nonsep(body, systCopy, spring, extended
            = True) * dt / 2

    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel2 += ( forceOn(body, systCopy, spring,
            extendedPosition = True) / body.mass ) * ( dt / 2 )
        body.pos += vel_nonsep(body, systCopy, spring) * dt / 2

    for body in syst:
        q = body.pos                        #position in "reel" system
        p = body.vel  * body.mass           #momentum in "reel" system
        x = body.pos2                       #position in "cloned"
            system
        y = body.vel2 * body.mass           #momentum in "cloned"
            system

        body.pos = ( q + x + np.cos( 2 * omega * dt ) * ( q - x
            ) + np.sin( 2 * omega * dt ) * ( p - y ) ) / 2
        body.vel = ( p + y + np.cos( 2 * omega * dt ) * ( p - y
            ) - np.sin( 2 * omega * dt ) * ( q - x ) ) / ( 2 *
            body.mass )
        body.pos2 = ( q + x - np.cos( 2 * omega * dt ) * ( q - x
             ) - np.sin( 2 * omega * dt ) * ( p - y ) ) / 2
        body.vel2 = ( p + y - np.cos( 2 * omega * dt ) * ( p - y
             ) + np.sin( 2 * omega * dt ) * ( q - x ) ) / ( 2 *
            body.mass )

    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel2 += ( forceOn(body, systCopy, spring,
            extendedPosition = True) / body.mass ) * ( dt / 2 )
        body.pos += vel_nonsep(body, systCopy, spring) * dt / 2
    systCopy = copy.deepcopy(syst)
    for body in syst:
        body.vel += ( forceOn(body, systCopy, spring,
            extendedMomentum = True) / body.mass ) * ( dt / 2 )
        body.pos2 += vel_nonsep(body, systCopy, spring, extended
            = True) * dt / 2


def tao4(syst, spring, dt):
    """"""
    gamma_4 = 1/(2 - 2**(1/5))
    tao(syst, spring, dt * gamma_4)
```

```
    tao(syst, spring, dt * (1-2*gamma_4))
    tao(syst, spring, dt * (gamma_4))


def plotEnergy( dt, numsteps, separable = False, Integrator = '
   Tao', plot = 'Energy'):
    """Peforms one simulation and plots energy"""
    #Our particles
    system = [Body('part1', 2.0, 3.0, 0.25), Body('part2', 2.0,
       2.0, -0.25)]
    #Our spring
    spring = Spring(1, 2, separable, 0.5)      #Spring(self,
       equilibrium, springConstant, separable, modelKonstant)
    #The initial energy
    realenergy = potentialEnergy(system, spring) + kineticEnergy
       (system)
    #Distance betwen the particles
    dist = [ ]
    #Velocity of particle 1????
    vel = [ ]
    #Energies
    poten = [ ]
    kinen = [ ]
    toten = [ ]
    #Error of total energy
    enerror = [ ]
    #Time
    time = [ ]

    for i in range(numsteps):
        if Integrator == 'Runge-Kutta 4':
                RK4(system, spring, dt )
        elif Integrator == 'Verlet':
            verlet(system, spring, dt )
        elif Integrator == 'Tao':
            tao(system, spring, dt )
        elif Integrator == 'Tao 4':
            tao4(system, spring, dt)

        #Append to vectors for plots
        dist.append( system[0].pos - system[1].pos)
        vel.append( system[0].vel )
        poten.append( potentialEnergy(system, spring) )
        kinen.append( kineticEnergy(system) )
```

```
        toten.append( potentialEnergy(system, spring) +
            kineticEnergy(system) )
        enerror.append( (potentialEnergy(system, spring) +
            kineticEnergy(system) - realenergy) / realenergy)
        time.append( i*dt )

    if plot == 'Energy':
        plt.plot(time, toten, label = 'Total energy')
        plt.plot(time, poten, label = 'Potential energy')
        plt.plot(time, kinen, label = 'Kinetic energy')
        plt.grid(b = True)
        plt.ylabel('Energy')

    if plot == 'Total energy':
        plt.plot(time, toten, label = 'Total energy')
        plt.ylabel('Total energy')
        plt.grid(b = True)

    if plot == 'Energy error':
        plt.plot(time, enerror, label = 'Energy error')
        plt.ylabel('Energy error')
        plt.grid(b = True)

    plt.title('Energy error ' + Integrator)
    plt.xlabel('Time')
    plt.legend(loc = 'best', prop = {'size':10})
    tikz_save(plot + '-' + str(Integrator) + '-' + str(dt) + 'dt
        -' + str(dt*numsteps) + 't-' + str(0.1) + 'beta-'+ str(
        spring._b) + 't.tex')
    plt.show()

def plotError( dt, numsteps, separable = False, plot = 'Energy
    error'):
    """Peforms one simulation and plots the error of *energy or
        momentum"""
    for Integrator in ['Verlet', 'Runge-Kutta 4', 'Tao', 'Tao 4'
        ]:
        #Our particles
        system = [Body('part1', 2.0, 3.0, 0.25), Body('part2',
            2.0, 2.0, -0.25)]
        #Our spring
        spring = Spring(1, 2, separable, 0.5)      #Spring(self,
            equilibrium, springConstant, separable, modelKonstant
            )
        #The initial energy
```

```
        realenergy = potentialEnergy(system, spring) +
            kineticEnergy(system)
        #Error of total energy
        enerror = [ ]
        #Momentum energy
        momErr = [ ]
        #Time
        time = [ ]

        for i in range(numsteps):
            if Integrator == 'Runge-Kutta 4':
                RK4(system, spring, dt )
            elif Integrator == 'Verlet':
                verlet(system, spring, dt )
            elif Integrator == 'Tao':
                tao(system, spring, dt )
            elif Integrator == 'Tao 4':
                tao4(system, spring, dt)

            #Append to vectors for plots
            enerror.append( (potentialEnergy(system, spring) +
                kineticEnergy(system) - realenergy) / realenergy)
            momErr.append( system[0].vel * system[0].mass +
                system[1].vel * system[1].mass )
            time.append( i*dt )

        if plot == 'Energy error':
            plt.plot(time, enerror, label = Integrator)
        if plot == 'Momentum error':
            plt.plot(time, momErr, label = Integrator)

    plt.title(plot)
    plt.grid(b = True)
    plt.xlabel('Time')
    plt.ylabel(plot)
    plt.legend(loc = 'best', prop = {'size':10})
    tikz_save('SpringMomentumError' + '-' + str(dt) + 'dt-' +
        str(spring._b) + 'beta-' + 't.tex')
    plt.show()


#plotEnergy( 0.01, 1000, separable = False, Integrator = 'Tao
    4', plot = 'Energy') #plotEnergy( dt, numsteps, separable =
    False, Integrator = 'Tao', plot = 'Energy')
```

```
#plotError( 0.01, 10000, separable = False, plot = 'Energy error
    ') #plotError( dt, numsteps, separable = False, plot = '
    Energy error')
```

5.2. Lennard-Jones simulation

```
import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import animation
import random as rnd
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D as p3
from matplotlib import rc
from matplotlib2tikz import save as tikz_save
from IPython.display import HTML
import random as rnd
import copy
from numba import jit

class Body:
    """Particle with mass, position, velocity and acceleration
       """
    def __init__(self, name, mass, pos, vel):
        self.name = name        #Name
        self.mass = mass        #Mass
        self.pos = pos          #Position vector
        self.vel = vel          #Velosity vector
        #Extended system
        self.pos2 = pos
        self.vel2 = vel

class System:
    """Particles in a box with periodic boundery conditions
       interaction via the Lennard-Jones potential"""
    def __init__(self, bodies, boxSize):
        self.bodies = bodies        #List of all particles
        self.boxSize = boxSize      #The size of the box
        #
        self._beta = 0.1

    @jit
    def epsilon(self, extended = False):
        """Returns epsilon of T"""
```

```
        epsilon0 = 0.5  #Epsilon at T = 0
        T = 0           #Temperature
        for body in self.bodies:
            if not extended:
                momentum = body.vel * body.mass
            elif extended:
                momentum = body.vel2 * body.mass
            T += np.sqrt( np.sum( momentum * momentum ) ) / ( 2
               * body.mass * len(self.bodies) )
        return epsilon0 * np.exp( - self._beta * T )

    def vel_nonsep(self, body, extended = False):
        """Returnerar hamiltonianen deriverad med avsende på
           partikelns rörelsemängd dvs dH/dp"""
        """Returns the Hamiltonian derived with respect to
           momentum"""
        if not extended: #We want dH(x, p)/dp
            return body.vel * ( 1 - self._beta * self.
               potentialEnergy(extendedPosition = True,
               extendedMomentum = False) )
        elif extended:   #We want dH(q, y)/dy
            return body.vel2 * ( 1 - self._beta * self.
               potentialEnergy(extendedPosition = False,
               extendedMomentum = True) )

    @jit
    def potentialEnergy(self, extendedPosition = False,
       extendedMomentum = False):
        """Returns the Lennard-Jones potential of the system"""
        poten = 0
        for i in range(len(self.bodies)):
            for j in range(i):
                #
                if not extendedPosition:
                    distVec = shortestDist( self.bodies[i].pos,
                       self.bodies[j].pos, self.boxSize )
                if extendedPosition:
                    distVec = shortestDist( self.bodies[i].pos2,
                       self.bodies[j].pos2, self.boxSize )
                normDist = np.sqrt( np.sum( distVec*distVec ) )
                #
                if not extendedMomentum:
                    epsilon = self.epsilon()
                if extendedMomentum:
                    epsilon = self.epsilon(extended = True)
```

```
                #
                if normDist < 2.5:
                    poten = poten + 4 * epsilon * ( normDist
                        **(-12) - normDist**(-6) )  #Epsilon = 1,
                         sigma = 1
        return poten

    @jit
    def kineticEnergy(self):
        """Calculates and returns the kinetic energy of the
           system"""
        kinen = 0
        for elem in self.bodies:
            velocitySq = np.sum( elem.vel * elem.vel )
            kinen += velocitySq * elem.mass / 2
        return kinen

@jit
def shortestDist(position1, position2, boxSize):
    """Calculates and returns the shortest distance vector
       between two particles"""
    distVec = np.array([1,1,1])*boxSize
    normDist = np.sqrt(np.sum(distVec*distVec))
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            for k in [-1, 0, 1]:
                newpos2 = position2 + boxSize * np.array([i, j,
                   k])
                newdistVec = position1 - newpos2
                newnormDist = np.sqrt(np.sum(newdistVec*
                   newdistVec))
                if newnormDist < normDist:
                    distVec = newdistVec
                    normDist = np.sqrt(np.sum(newdistVec*
                       newdistVec))
    return distVec

@jit
def forceOn(body, syst, extended = False):
    """Returns the force on a particle"""
    force = np.array([0, 0, 0])
    for elem in syst.bodies:
        if body.name != elem.name:
            if not extended:
                #We want dH(q, y)/dq dvs
```

```
                distVec = shortestDist(body.pos, elem.pos, syst
                    .boxSize)
                epsilon = syst.epsilon( extended = True )
            elif extended:
                #We want dH(x, p)/dx
                distVec = shortestDist(body.pos2, elem.pos2,
                    syst.boxSize)
                epsilon = syst.epsilon()
            normDist = np.sqrt(np.sum( distVec*distVec ))
            if normDist < syst.boxSize / 2: #Cut off
                force = force + 24 * epsilon * ( 2 * normDist
                    **(-14) - normDist**(-8) ) * distVec  #Sigma
                    = 1
    return force

def initCond(boxSize, numElem):
    """Restuns an instans of the Syst method with initial
        conditions"""
    #Hastigheterna, ser till att masscentrum inte rör sig
    np.random.seed(0)
    velocities = [ 2*np.random.random_sample(size=3) - 1 for i
        in range(numElem) ]
    totalvel = np.array([0,0,0])
    for vel in velocities:
        totalvel = totalvel + vel
    velocities = [ velocities[i] - totalvel / numElem for i in
        range(numElem) ]
    #Positions, ser till att partiklarna inte börjar för nära
        varandra,
    positions = [ ]
    while len(positions) < numElem:
        isgood = True
        newpos = boxSize * np.random.random_sample(size=3)
        for pos in positions:
            distVec = shortestDist(pos, newpos, boxSize)
            dist = np.sqrt( np.sum( distVec * distVec ) )
            if dist < 1:
                isgood = False
                break
        if isgood:
            positions.append(newpos)
    bodies = [Body( 'Particle' + str(i), 4, positions[i],
        velocities[i] ) for i in range(numElem)]
```

```
    #bodies = [Body('part1', 1, np.array([2, 5, 0]), np.array
        ([3, 0, 0])), Body('part2', 1, np.array([8, 5, 0]), np.
        array([-3, 0, 0]))]
    #Our system
    return System(bodies, boxSize)

def verlet(syst, dt):
    """One step for using verlet"""
    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel = body.vel + ( forceOn(body, systCopy) / body.
            mass ) * (dt / 2)
        body.pos = body.pos + body.vel * dt
    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel = body.vel + ( forceOn(body, systCopy) / body.
            mass ) * (dt / 2)

def tao(syst, dt, i):
    """One step for one particle using Molei Taos method"""

    omega = 1 #Binding parameter in Molei Taos method

    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel = body.vel + ( forceOn(body, syst, extended =
            False) / body.mass ) * ( dt / 2 )
        body.pos2 = body.pos2 + syst.vel_nonsep( body, extended
            = True ) * dt / 2
    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel2 = body.vel2 + ( forceOn(body, syst, extended =
             True) / body.mass ) * ( dt / 2 )
        body.pos = body.pos + syst.vel_nonsep( body, extended =
            False ) * dt / 2

    for body in syst.bodies:
        q = body.pos
        p = body.vel * body.mass
        x = body.pos2
        y = body.vel2 * body.mass

        body.pos  = ( q + x + np.cos( 2 * omega * dt ) * ( q - x
             ) + np.sin( 2 * omega * dt ) * ( p - y ) ) / 2
```

```
        body.vel  = ( ( p + y + np.cos( 2 * omega * dt ) * ( p -
            y ) - np.sin( 2 * omega * dt ) * ( q - x ) ) / 2 ) /
            body.mass
        body.pos2 = ( q + x - np.cos( 2 * omega * dt ) * ( q - x
            ) - np.sin( 2 * omega * dt ) * ( p - y ) ) / 2
        body.vel2 = ( ( p + y - np.cos( 2 * omega * dt ) * ( p -
            y ) + np.sin( 2 * omega * dt ) * ( q - x ) ) / 2 ) /
            body.mass

    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel2 = body.vel2 + ( forceOn(body, syst, extended =
            True) / body.mass ) * ( dt / 2 )
        body.pos = body.pos + syst.vel_nonsep(body, extended =
           False) * dt / 2
    systCopy = copy.deepcopy(syst)
    for body in syst.bodies:
        body.vel = body.vel + ( forceOn(body, syst, extended =
           False) / body.mass ) * ( dt / 2 )
        body.pos2 = body.pos2 + syst.vel_nonsep(body, extended =
            True)  * dt / 2


def main(boxSize, numElem, dt, endtime):
    """Performs one simulation"""
    #Timestep
    numsteps = int(endtime/dt)
    #Our system
    system = initCond(boxSize, numElem)
    #Initial energy
    initEn = system.potentialEnergy() + system.kineticEnergy()
    #Stores the x, y, & z positions for all particles
    x = [ [ ] for body in system.bodies ]
    y = [ [ ] for body in system.bodies ]
    z = [ [ ] for body in system.bodies ]
    #Stores the energies of the system
    poten = [ ]
    kinen = [ ]
    toten = [ ]
    ###THIS CREATES FIGURES###
    #Creates the figure for 3D animation
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(121, projection='3d')
    ax.set_xlim(0, boxSize)
    ax.set_ylim(0, boxSize)
```

```
ax.set_zlim(0, boxSize)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
#Create figure for energy plot
energy = fig.add_subplot(122, label = "Energy")
energy.set_xlim(0, endtime)
if 1.5 * initEn < 2.5:
    energy.set_ylim( - 0.4 * initEn, 2.5 )
else:
    energy.set_ylim( - 0.4 * initEn, 1.5 * initEn )
energy.set_xlabel('Time')
energy.set_ylabel('Energy')
##THIS STORES SOMETHING###
#Stores time
time = [ ]
#For the 3D plot
lines = [ ax.plot([],[],[], '-', alpha=0.5, c = 'r')[0] for
   i in range(numElem) ]
pts = [ ax.plot([],[],[], 'o', c = 'r')[0] for i in range(
   numElem) ]
#For the energy plot
energies = [ energy.plot([], [])[0] for i in range(3) ] #
   energies = [potential energy, kinetic energy, total
   energy]

def init():
    """skapar första bilden"""
    for line, pt in zip(lines, pts):
        line.set_data([], [])
        line.set_3d_properties([])
        pt.set_data([], [])
        pt.set_3d_properties([])
    for energy in energies:
        energy.set_data([], [])
    return lines + pts + energies

def animate(i):
     for body in system.bodies:
         #Moves the particles one step forward
         RK4step_sepLJ(body, system, dt)
         #tao_sep(body, system, dt)
     for body in system.bodies:
         #Moves particles back in the box
         body.pos = body.pos % boxSize
```

```
            body.pos2 = body.pos2 % boxSize
        for j in range(len(system.bodies)):
            #Append positions to positionvector
            x[j].append(system.bodies[j].pos[0])
            y[j].append(system.bodies[j].pos[1])
            z[j].append(system.bodies[j].pos[2])
        for j in range(len(system.bodies)):
            #Plots all previous positions as lines
            lines[j].set_data(x[j], y[j])
            lines[j].set_3d_properties(z[j])
            #plots current positions as dots
            pts[j].set_data(x[j][-1], y[j][-1])
            pts[j].set_3d_properties(z[j][-1])
        #Appends time and energies
        poten.append( system.potentialEnergy() )
        kinen.append( system.kineticEnergy() )
        toten.append( system.potentialEnergy() + system.
            kineticEnergy() )
        time.append( i * dt )
        #Plots the energies
        energies[0].set_data(time, poten)
        energies[1].set_data(time, kinen)
        energies[2].set_data(time, toten)
        return lines + pts + energies

    anim = animation.FuncAnimation(fig, animate, init_func =
        init,
                                   frames = numsteps, interval
                                       =50, blit=True, repeat=
                                       False)
    plt.show()

def plotEnergy(boxSize, numElem, dt, endtime, integrator = "Tao"
    ):
    """Plots energy"""
    #Number of steps
    numsteps = int(endtime/dt)
    #Our system
    system = initCond(boxSize, numElem)
    #Stores the potential, kinetic and toatal energy
    poten = [ ]
    kinen = [ ]
    toten = [ ]
    #Stores time
    time = [ ]
```

```
#Initial energy, not used at the moment
initEn = system.potentialEnergy() + system.kineticEnergy()
for i in range(numsteps):
    if i % int(0.1/dt) == 0:
        print(i*dt)
    if integrator == "Verlet":
        #Moves all particles forward one timestep using
            verlocity Verlet
        verlet(system, dt)
    elif integrator == "Tao":
        #Moves all particles forward one timestep using Tao
        tao(system, dt, i)
    for body in system.bodies:
        #Moves all particles back in the box
        body.pos = body.pos % boxSize
        body.pos2 = body.pos2 % boxSize
    #Append energies and time
    poten.append( system.potentialEnergy() )
    kinen.append( system.kineticEnergy() )
    toten.append( system.potentialEnergy() + system.
        kineticEnergy() )
    time.append( i * dt )
##Need for plots
plt.plot(time, toten, label = 'Total energy', linewidth = 2)
plt.plot(time, poten, label = 'Potential energy', linewidth
    = 2)
plt.plot(time, kinen, label = 'Kinetic energy', linewidth =
    2)
plt.legend(loc = 'best', prop = {'size':12})
plt.show()
```