DEPARTMENT OF COMPUTER SCIENCE

NEW BULGARIAN UNIVERSITY

Bachelor's Thesis in Informatics

Kalin Stoyanov

DEPARTMENT OF COMPUTER SCIENCE

NEW BULGARIAN UNIVERSITY

Bachelor's Thesis in Informatics

Gravitational N-Body Simulations with Apache Spark

Author:	Kalin Stoyanov
Supervisor:	Stoyan Mishev, PhD
Submission Date:	June 1, 2021

Contents

1. Introduction

1.1. The N-body problem

Predicting the orbits of observable planets bothered astronomers' minds for centuries. With Newton's discovery of the law of mutual attraction started the endeavour of the exact calculations of trajectories of stellar bodies. It proved to be an easy task for two bodies, a very hard task for three bodies and unsolved for a large number of bodies in the general case. However approximate analytic solutions such as the perturbation approach has given landmark results. The so-called N-body problem in classical gravity can be defined as:

> *Given the initial positions and velocities of a number of celestial bodies evaluate the forces acting on them at any subsequent moment in time.*

The increase in computing power in the last several decades has allowed brute-force ab-initio solutions which are dubbed as computer simulations. They allow accurate modeling of planetary systems, star clusters, and weak force interactions in general and provide interesting computational challenges leading to the creation of new technologies, albeit mostly specifically focused on the problem's solution. This problem is especially important in the light of the recent spur of interest in the search of dark energy and in particular in studying the redshift of approx. 10 million galaxies within the DESI experiment [DESI-Collaboration 2016] which is facilitated by gravitational models capable of propagating $\sim 10^{12}$ particles[1]. Below we summarize the main steps

[1] conf. the EXASKY project @ `https://www.exascaleproject.org/research-project/exasky/`

that we implemented in our simulation.

With the initial conditions m, r, and v being mass, position, and velocity at the initial time t_0, the first step is to use Newton's Law of Gravity to evaluate the gravitational forces acting on each body. The force per unit of mass acting on the i-th particle[2] in a system of N particles is defined as

$$F_i = -G \sum_{j=1;j\neq i}^{N} \frac{m_j(r_i - r_j)}{|r_i - r_j|^3}. \tag{1.1}$$

Considering that $F_i = m\ddot{r_i}$ this produces $6N$ first order differential equations with respect to the initial conditions that provide a solution for $r_i(t)$ over a time interval of $(-\infty, \infty)$. Analytical solutions for this exist only when $N = 2$ (i.e. the 2-body problem) (D. C. Heggie 2005), and simulations that use different methods are used to predict the evolution of systems of more particles. Since the energy of conservative systems remains constant (for simplicity's sake these are presumed to be closed systems with no external influence), it is typically used to estimate the accuracy of such simulations. The total kinetic (T) and potential (U) energy, and total energy E used for such checks are computed by

$$T = \frac{1}{2} \sum_{i=1}^{N} m_i v_i^2 \tag{1.2}$$

$$U = -\sum_{i=1}^{N} \sum_{j>i}^{N} \frac{G m_i m_j}{|r_i - r_j|} \tag{1.3}$$

$$E = T + U \tag{1.4}$$

as presented by Sverre J. Aarseth 2003 (with the outside energy W omitted as we are focusing on a closed system). There are also other qualities that should be conserved during a stellar evolution like the cluster's angular momentum or its center of mass

[2] "object", "body", and "particle" are used interchangeably

moving at a constant speed, etc., which are usually not used to gauge accuracy as evident by the various existing implementations of N-body scripts. (Sverre J. Aarseth 2003)

The problem is computationally challenging as a direct integration method would have a complexity of $O(n^2)$. Such a method provides the most accuracy at the cost of most computing time. A similar relationship is naturally exhibited by the particular method chosen for direct integration with higher order methods providing higher accuracy for larger time steps (Δt) than lower order ones, at the cost of each individual step taking longer to compute. There also exist approximation methods to solve the N-body problem, mostly reliant on the fact that far away bodies exert a minimal amount of force on each other in relation to ones close by (e.g. Tree code methods).

Historically, exploration of the N-body problem has led to the creation of both specialized software and hardware in order to facilitate more efficient and/or accessible solutions - for example Sverre Aarseth's NBODY scripts and the numerous iterations of the GRAPE specialized computer (Sverre J. Aarseth 2003). Also a trend has begun to emerge in which interested researchers aim to create comprehensive software suites that handle multiple aspects of stellar dynamics, which can include generation of sample cluster data, unit conversions, etc. in addition to the N-body simulations themselves. Examples of such projects include NEMO, Maya, and Starlab, and an inexhaustive list of these can be found on the NEMO website, Teuben 2019.

1.2. Apache Spark

Apache Spark is an engine for large scale data processing, designed to be run in parallel on a cluster. This allows it to leverage both large amounts of memory and processing power in order to handle huge workloads. While having a number of tools for different purposes, we will focus on Spark SQL, its module for data processing. A notable feature of Spark is its code optimization and ease of use - while providing high-level API-s for

Java, Scala, Python, and R, user code is processed into an optimized execution plan by SQL's Catalyst Optimizer, and lazily executed. This is largely beneficial as it improves performance in most cases, but on the other hand it introduces some difficulties (and sometimes arcane practices) in tuning a Spark application as opposed to running the user code directly. Another useful feature of Spark is that it provides an interactive shell that allows users to dynamically explore their data. All of these should make Apache Spark a natural fit for running N-body simulations due to their computational intensity, and potentially large volume of data (Apache Software Foundation 2020).

1.3. Goals

The main goal of this work is to run N-body simulations on Spark. A roof of concept for a stellar dynamics toolkit was created using the Python API for Spark (PySpark), along with some native python libraries[3]. The source code includes several integrators to solve the computational problem outlined above as well as means to calculate the energy of a system for diagnostics. It is also designed to be extensible so as to easily allow future expansion. Furthermore, any notable findings during our work on the Spark implementation are documented in order to facilitate the intrigued user in understanding the implemented algorithms. Finally our implementation of one of the algorithms is compared to implementations from other authors who used different frameworks. It is expected that Spark will underperform when N is a moderate number due to its overhead but it should overtake single and possibly multi-threaded implementations when N is large.

[3]The source code and data used are available at `https://github.com/kgskgs/stars-spark3d`

2. Code overview

2.1. High level overview

The code for the PySpark N-body simulation framework is split across the following files:

File	Description
`/src/main.py`	Entry point when ran with spark-submit, accepts parameters to control the simulation
`/src/simulation.py`	Holds the simulation conditions, runs the simulation and provides output
`/src/integrator_base.py`	Abstract class inherited by integrators, contains the force calculation
`/src/integrators.py`	Holds the currently available integrator classes
`/src/cluster.py`	Contains energy and other calculations applicable to a whole cluster
`/src/utils.py`	Miscellaneous functions used by the other files
`/src/schemas.py`	Holds the schemas for all dataFrames used throughout the project
`/src/test.py`	Tests the force calculations from cluster.py
`/src/generate_data.sh`*	Used to facilitate data generation with NEMO

Table 2.1.: Source code files

The notable code segments in these files will be discussed in this chapter, starting with the integrators. Then we move on describing the data format, usage and verification. The full API documentation is given in Appendix A.

2.2. Integrators

The simplest idea for numerical integration is implemented here - after the force acting on every particle is computed as in (1.1) at time t_0, it can be used to compute its future

*needs to be run from a Linux terminal with NEMO installed (https://teuben.github.io/nemo/)

position and velocity at time $t_0 + \Delta t$. This process is repeated until t reaches the desired time to end the simulation, or some other condition is satisfied. All the integrators currently implement a time step that is constant and uniform for all particles.

The `IntegratorBase` class in `integrator_base.py` realizes the summation in Eq. (1.1). It is an abstract class (created using Python's *ABC* module) to be inherited by the concrete implementations of integrators as the force evaluation is most likely to be common to all of them. The only abstract method that has to be implemented by children classes is `advance` - it is used by the `Simulation` class (discussed later on) when running the N-body simulation. It has to advance the positions and velocities of all particles by a single step, and must return a tuple containing a Spark DataFrame containing the new cluster data, and the time Δt that has passed since the start of the step. The latter is added to facilitate the addition of variable time step methods in the future. The concrete implementations of integrators are contained in `integrators.py`. Every instance has to be instantiated with parameters `dt` and `G` which control the time step and gravitational constant used for the force calculations.

The concrete implementation of (1.1) takes place in the `IntegratorBase.calc_F` and `IntegratorBase.calc_F_cartesian`. Firstly, in the former, the cluster data is joined with itself in a Cartesian join, and the joins between a particle and itself are filtered out. The force is then calculated between every two particles and the result is returned. This can be used to check that on a star per star basis, or as following the script it can be aggregated by the particle's id in order to get the total force acting on it.

2.2.1. Euler Integrator

`IntergratorEuler` implements the simplest method for numerical integration - forward Euler. With the force F_i evaluated for particle i at t_0, it obtains its position and velocity at time $t = t_0 + \Delta t$ via

$$v_i(t) = F_i \Delta t + v_i(t_0) \tag{2.1}$$

$$r_i(t) = \frac{1}{2}F_i\Delta t^2 + v_i(t_0)\Delta t + r_i(t_0) \tag{2.2}$$

(Sverre J. Aarseth 2003). These are implemented in the functions `step_v` and `step_r` respectively. The whole process outlined above for advancing all particles simultaneously for a singe time step is contained in this integrator's `advance` method. With it being a first order integrator, it naturally provides the least accuracy and the best performance as it requires only one force evaluation per step.

2.2.2. Improved Euler Integrator

The second class, `IntergratorEuler2` inherits `IntergratorEuler` and provides a second order integrator that yields more accuracy per step at the cost of evaluating the force twice in a single step. Firstly it obtains provisional position coordinates at time t via (2.2) using the force at t_0, which are then used in (1.1) to obtain the new force $F_i(t)$. In turn that is used to calculate the average force over the period Δt:

$$F_i = \frac{1}{2}[F_i(t) + F_i(t_0)] \tag{2.3}$$

(Hut and Makino 2007) This average force is used to calculate the final values for position and velocity at time t by (2.2) and (2.1).

2.2.3. Runge-Kutta Integrator

`IntegratorRungeKutta4` provides a fourth order Runge-Kutta integrator. Similarly to the improved Euler integrator, it calculates the force at four points during the interval Δt in order to arrive at a better approximation for F. Let us denote calculating equation (1.1) from position coordinates r as $F(r)$. The final force used for advancing a single step can be calculated by the weighted average

$$F = \frac{1}{6}[F_1 + 2F_2 + 2F_3 + F_4] \tag{2.4}$$

where F_1 is calculated directly from (1.1) with the initial positions ($F_1 = F(r_{t0})$) and the following

$$F_2 = F(r_2)$$

$$F_3 = F(r_3)$$

$$F_4 = F(r_4)$$

are calculated by using provisional coordinates r_k (particle index i omitted for clarity) which are computed modifying (2.2) and (2.1) so they accept coefficients c as per the fourth order Runge-Kutta matrix described in Roa et al. 2020

$$r_k = \frac{1}{2}F_{k-1}\Delta t^2 c_k^2 + v_{k-1}\Delta t c_k + r_{k-1} \tag{2.5}$$

$$v_k = F_{k-1}v_{k-1}\Delta t c_k + v_{k-1} \tag{2.6}$$

where $c = (\frac{1}{2}, \frac{1}{2}, 1)$.

2.2.4. Leapfrog Integrator

`IntegratorLeapfrog` contains a Verelt-leapfrog integrator, which is a second order integrator despite using just one force evaluation per step. It calculates the new positions and velocities at different times with the new velocities being evaluated at $\frac{1}{2}t$, and the positions - at t. In order to use a uniform integer time step v_t is computed by

$$v_i(t) = \frac{1}{2}(F_i(t_0) + F_i(t))\Delta t + v_i(t_0) \tag{2.7}$$

Before starting the simulation $F_i(t_0)$ is initialized from the given data. Afterwards every step is conducted as follows

1. Compute the new positions $r_i(t)$ via (2.2) using $F(t_0)$
2. Plug in $r_i(t)$ into (1.1) to get $F_i(t)$

3. Compute the new velocities via (2.7)

4. Set $F_i(t_0) = F_i(t)$ in preparation for the next step

This algorithm is described in Castle 2010. The class illustrates how an integrator that saves and reuses its state can be created (as opposed to the previous three that compute the steps independently of one another). The force $F_i(t_0)$ or `df_F_t0` is a class variable that is initialized as `None` in the constructor, and precomputed in the `advance` method if it does not exist already.

2.3. Simulation class

We now have the complete picture of how a N-body simulation can be ran. After getting the initial conditions, an integrator is chosen to coupled differential equations presented above, along with the value of Δt which will control the time step. The integrator's `advance` method is then called to advance the simulation step by step until the desired time is reached.

This process is handled by the `Simulation` class. When initialized it receives the particle data (`cluster` in the script), and the chosen integrator. Other notable parameters are `dt_out`, and `dt_diag` that control how often a snapshot of the simulation should be saved, and how often diagnostic energy calculations should be conducted. These two default to `None`, and if not provided, the respective operation will not be performed. The `run` method conducts the simulation by repeatedly calling the chosen integrator's `advance` method, and receiving the new cluster data, and time passed since the previous step until the target time, `ttarget`, is reached.

If `dt_out` is supplied, it also saves the current cluster snapshot every `dt_out` time units by calling the `snapshot` method. Similarly, every `dt_diag` time units, the `diag` method is called to perform energy diagnostics. It computes the total energy of the cluster at time t, and if no initial energy $E(t_0)$ or `E_initial` in the script, sets it to that

value. Afterwards computes the total energy error dE:

$$dE = \frac{E(t) - E(t_0)}{E(t_0)} \tag{2.8}$$

Finally it logs the current time t, current energy, and total energy error.

2.4. Cluster functions

The file `cluster.py` contains a collection of functions that provide calculations about the statistics of a whole cluster (or collection of particles). It's methods `calc_T` and `calc_U` implement equations (1.2) and (1.3). They are used in the energy diagnostic described in the previous section. It also provides the following methods that can be used to glean information about the following aspects of a cluster which are also presented as metrics that can be tracked according to Sverre J. Aarseth 2003:

- The center of mass R

$$R = \frac{1}{M} \sum_{i=1}^{N} m_i r_i$$

  where M is the total mass

- The half mass radius rh

- The cluster's angular momentum J

$$J = \sum_{i=1}^{N} r_i \times m_i v_i$$

2.5. Usage

With the outline of the simulation now complete, we can move on to how to use the scripts provided to conduct it. There are two major ways to utilize them - either by

Argument	Description	format	Default
`dt`	Δt for calculating steps	float	-
`target`	target time to reach in the simulation	float	-
`integrator`	integrator to use for running the simulation	`[eul1, eul2, rk4, vlf]`	-
`input`	path(s) to input data	string	-
`--dtout`	time interval between cluster snapshots	float	`None`
`--dtdiag`	time interval between diagnosting output	float	`None`
`--saveDiag`	if set diagnostics are saved to disk and not printed	float	-
`--addT`	if set t is added to every particle in the snapshots	float	-
`-l --limit`	limit the number of input rows to read from input	int	`None`
`-o --outputDir`	output path	string	`../output/`
`-f`	format to save output in	`[parquet, csv]`	`parquet`
`--comp`	compression to apply when saving output	string	`None`
`-G`	gravitational constant to use in the simulation	float	`1`

Table 2.2.: Arguments for main.py

running a self contained simulation by submitting `main.py` as a Spark application or by interactively running them in a PySpark shell.

Spark applications are submitted via the `spark-submit` command. Its general format is

`spark-submit [options] <app jar | python file | R file> [app arguments]`

While discussed in detail in the Spark documentation, Apache Software Foundation 2020, there are a few notable Spark options for this project. The resulting output is named from a combination of the `--name` option and the app id that is assigned automatically by Spark. If ran in a cluster the `--deploy-mode` can be set to `cluster`, and all python files other than `main.py` have to be supplied to the option `--py-files` (usually as an archive for convenience, or as a list of all the paths) so they are transmitted to all nodes, and added to the python path for the execution. If ran locally from the source directory, this last option can be omitted. Finally, `--conf spark.default.parallelism=X` can be used to adjust the number of partitions in general and `--conf spark.sql.shuffle.partitions=X` - the number of partitions for shuffle operations. The number of partitions the data is split in can have an impact on performance and is discussed in more detail later on.

The various parameters of the simulations are controlled by the arguments passed to

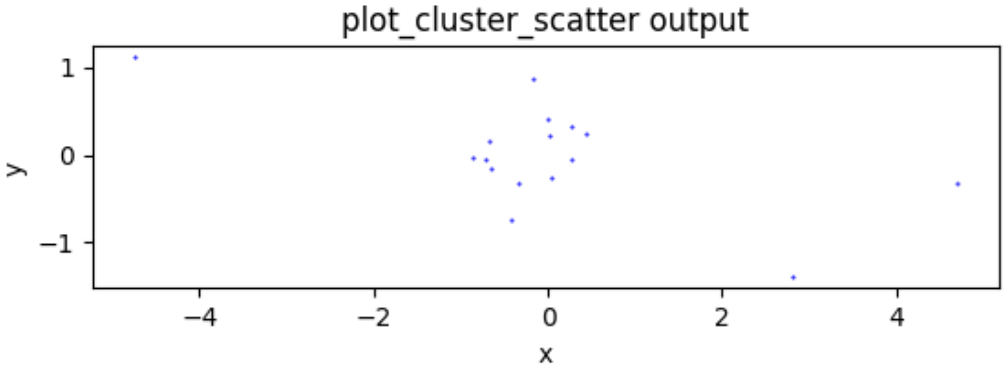
`main.py` are outlined in table 2.2. These are all implemented via Python's `argparse` module, and when the app is submitted they are used to create an instance of the `Simulation` class. That is then set to run until the target time is reached. If a user wants to add a new integrator, they have to update the dictionary `methods` with a key of type string, and value - an instance of the new integrator, and also add the key to the list of possible values for the `integrator` argument.

Alternatively, if the user wishes to use the tools provided without running a full simulation, they can do so via the PySpark shell, which combines the Spark framework with Python's interactive shell. This can be useful if one wishes to explore data, results, or do custom plots like the ones in presented in chapter 5. For example one can load some cluster data, calculate its energy, and visualize it with `matplotlib` via the `utils.plot_cluster_scatter` function. The example in 2.1 is ran from the source directory so the source files can be used directly. If the Pyspark shell is started from elsewhere, the files have to made available in the `PYTHONPATH` environment variable.

Listing 2.1: Shell usage example

```
PS D:\prog\proj\stars-spark3d\src> pyspark
[Spark welcome message]
>>> import utils
>>> import schemas
>>> import cluster
>>> data = utils.load_df("../data/plummer/plummer_16_s123.csv", schema=schemas.clust)
>>> cluster.calc_E(data)
-0.19140579990173162

>>> utils.plot_cluster_scatter(data, ['x','y'], title="plot_cluster_scatter_output")
>>> exit()
```

Figure 2.1.: Shell usage example plot result



3. Data and Code Verification

In this chapter we'll discuss the various data sets included in this project and how they are used to validate the implementations of the scripts. The data format used is either csv or parquet, with the output format controlled by the parameter discussed in 2.5. When reading a Dataframe via `utils.load_df` the format is detected from the filename. The main data for the cluster has to contain the following eight columns: particle id, position and velocity vectors, split in components, and mass, and that information is contained in `schemas.clust` so it can be easily reused (this file also contains several other schemas that serve mainly as information of what the different Dataframes used throughout the scripts contain). There are several different data sets included in the `data` directory, presented in table 3.1.

The data in the cluster and directory is in standard N-body units[1] where $G = M = -4E = 1$, and all particles have equal mass. It contains the evolution of a King model cluster from $t = 0$ to $t = 1800$, with snapshots at every 100 time units, as simulated by Aarseth's NBODY6 script. This data is used to validate the energy calculation from `cluster.py`. As that calculation is used to judge the accuracy of the simulations conducted it is very important that it itself is accurate. In order to verify this, the `test.py` script is used. It calculates the total energy for all 18 snapshots, and compares the results to the target of -0.25. As this data is itself the result of a simulation, the energy throughout the different snapshots is not constant, but the difference from the

[1] Since their introduction, see D. Heggie and Mathieu 1986, they have been established as the defacto standard when running N-Body simulations

path	description	source
/data/cluster/	simulation of a cluster with 64000 particles	generated with NBODY6, see Sverre J. Aarseth 2003 and Pasquato 2017
/data/plummer/	Plummer models of clusters with different numbers of particles	generated with NEMO, see Teuben 2019
/data/nbabel/	Plummer models	Castle 2010
/low_n/orbit_circular.csv	one particle orbiting another in a circular orbit	Hut and Makino 2007
/low_n/orbit_eliptical.csv	one particle orbiting another in an eliptical orbit	Hut and Makino 2007
/low_n/binary_eliptical.csv	two particles with elliptical orbits	Hut and Makino 2007
/low_n/3body_fig8.csv	three particles forming a figure-8	Roa et al. 2020

Table 3.1.: Data provided

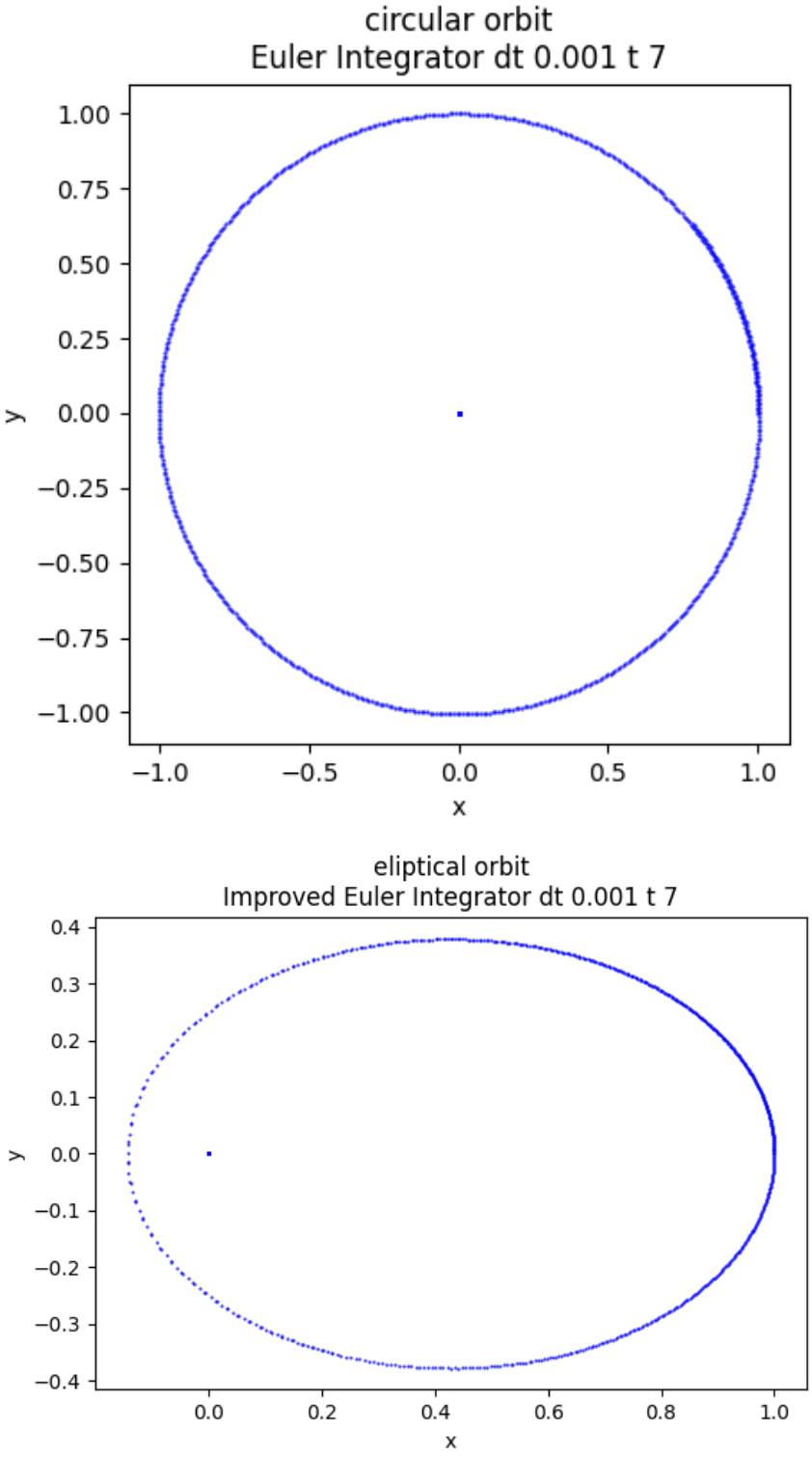
target varies from 2.316e−7 for c_000.csv to 3.292e−5 for the last snapshot, which shows that the energy calculations in this project is correct.

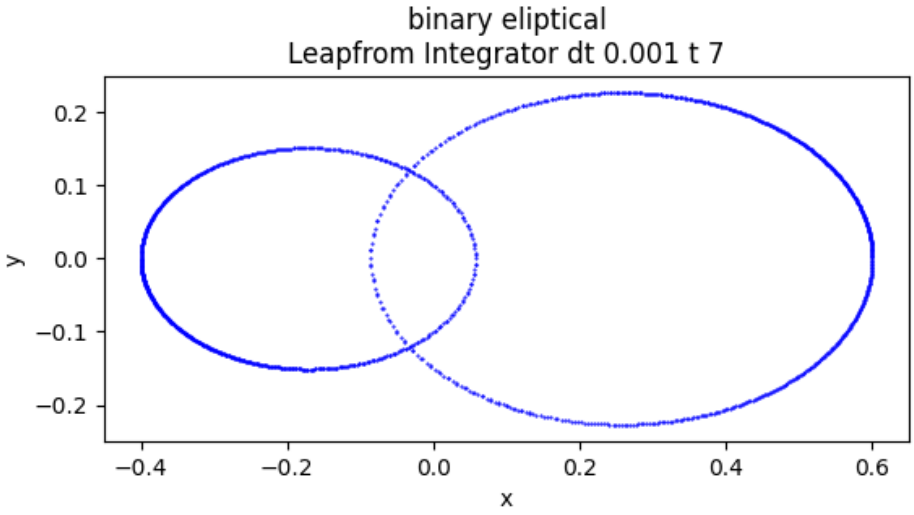
The data in the plummer folder also uses N-body units and has equal mass split across all particles. As the name implies it contains Plummer models, which are commonly used for N-body simulations and numerical methods comparison. (S. J. Aarseth, Henon, and Wielen 1974) The data sets are generate using NEMO, a stellar computation toolkit, using the `generate_data.sh` bash script in order to conveniently generate multiple sets, and format them to match the cluster schema used in this project. The filenames contain the number of particles and the seed used to generate it so it can be recreated if needed. This data is used when running the simulations discussed later on, and comparing the different integrators in terms of performance. The data in the nbabel folder also contains Plummer models. Due to the data impact on simulation accuracy discussed in 5.2 it is used when measuring the accuracy for simulations since this data set has already been used for such purposes, and can be judged more appropriate than the randomly generated data.

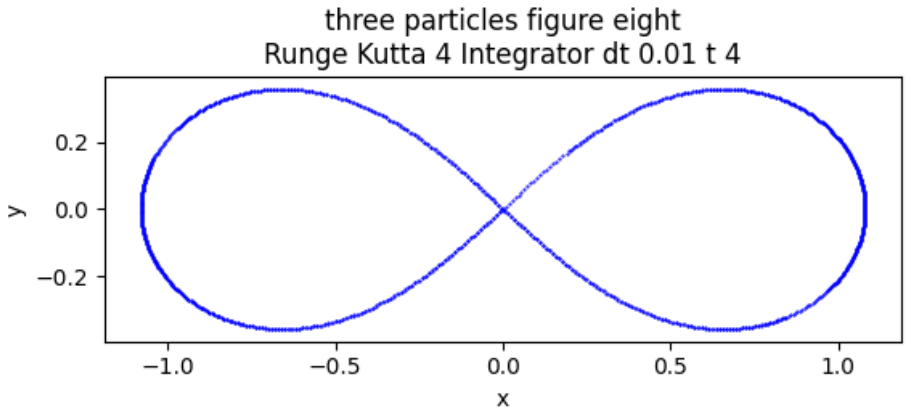
The other four sets of data located in the low_n folder contain particles with such initial conditions that their behaviour is known in advance. They don't follow the N-body units as the total energy is not always -0.25 or their masses don't always add up to 1, but they should still be run with $G = 1$. This data can be used to verify integrators used for the simulation by checking if the results they produce match the

predetermined pattern. All four data sets don't contain a z component to their position and velocity vectors so that they can be easily plotted in two dimensions. The below figure shows the results when using these data sets. Each individual image bears the name of the integrator used, as well as the value of Δt (`dt`), and final t reached during the simulation. All four integrators provide accurate results for these known problems, but it's worth to note that with too high values for Δt or if $G = 1$ is not used, they will not produce the correct images. If implementing more integrators one would have to make sure these values are tuned during testing in order not to misattribute incorrect figures to an incorrect implementation of the integrator.

Figure 3.1.: Results from running the four included integrators on the /data/low_n/data sets




binary eliptical
Leapfrom Integrator dt 0.001 t 7
0.2
0.1
0.0
−0.1
−0.2
y
−0.4
−0.2
0.0
0.2
0.4
0.6
x


three particles figure eight
Runge Kutta 4 Integrator dt 0.01 t 4
0.2
0.0
−0.2
y
−1.0
−0.5
0.0
0.5
1.0
x

4. Working with Apache Spark

The easily accessible high level API of Spark is accompanied with several peculiarities one of which is that the code executed on the cluster is pretty far removed from the user's input code. While the general idea of how a Spark job is conducted is fairly simple, its modifications and tuning is a more elaborate business as evidenced by the myriad of papers on this subject. In this chapter we will discuss details on how to use Spark to perform our task.

4.1. Spark Overview

Before moving on to the actual findings we'll first broadly outline how the Spark framework works, focusing on the parts that will provide the needed context. In essence it is a distributed computing framework, that can be ran in a variety of ways - on a standalone cluster, Kubernetes or Hadoop-YARN. This particular work has been run mostly on YARN, using Amazon EMR, which provides a streamlined way to create virtual clusters that come pre-installed with Spark.

When deployed in cluster mode, the Spark engine consists of one master node, and multiple worker nodes. A driver program is ran on the master node as well as a `SparkContext` object that contains resources allocated by whatever cluster manager is used. For the execution of each Spark application, the `SparkContext` acquires processes on the worker nodes called `Executors`, which have their own memory and computing cores, and which ultimately process the data. These are assigned individual

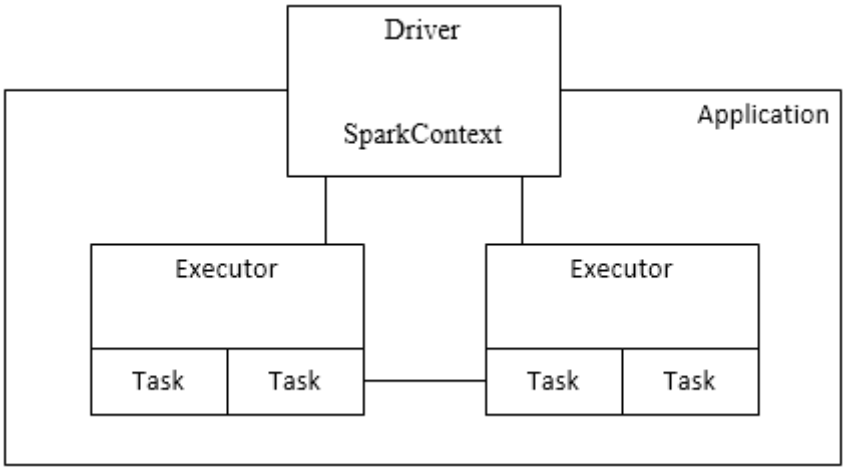


Figure 4.1.: Application high level organisation

`Tasks`, the smallest unit of work, to execute by the context.

The data operated on is contained in distributed structures that can be processed on in parallel, which can be Resilient Distributed Datasets (or `RDD`s) or DataFrames. They are both row-oriented and immutable, but a RDD is "just" a distributed collection of elements. Dataframes are built on top of RDDs and are organized into named and typed columns, with that information contained in the Dataframe's `schema`. They are conceptually equivalent to a relational database table. Spark uses the extra metadata that Dataframes provide to enact additional optimizations when working on them, and that is why this is the datatype used in our project; when data is discussed further on, one can assume it is contained in a Dataframe. The aforementioned tasks ran by the executors each operate on chunks of Dataframes called `partitions`. (Apache Software Foundation 2020)

The operations that can be performed on the data are split in two types - transformations and actions. An action involves the collection of results from the whole DataFrame - for example if it needs to be sent to the driver (`collect()` or `count()` functions) or saved to disk. Transformations, that alter the data as the name implies, and are preformed by the executors. It is notable that some transformations

require the data to redistributed between partitions and executors - a costly `shuffle` operation which involves disk IO. Examples of these include joins between two dataframes, `groupBy(key)` (a reduce function that can be followed by a number of others like `sum`), and anything similar that would require the data to be shared between partitions. As Spark uses lazy evaluation, the transformations on the data are only computed once an action is called. Throughout execution a record of the transformation that led to the creation of a Dataframe is kept, called its `lineage`. (Apache Software Foundation 2020)

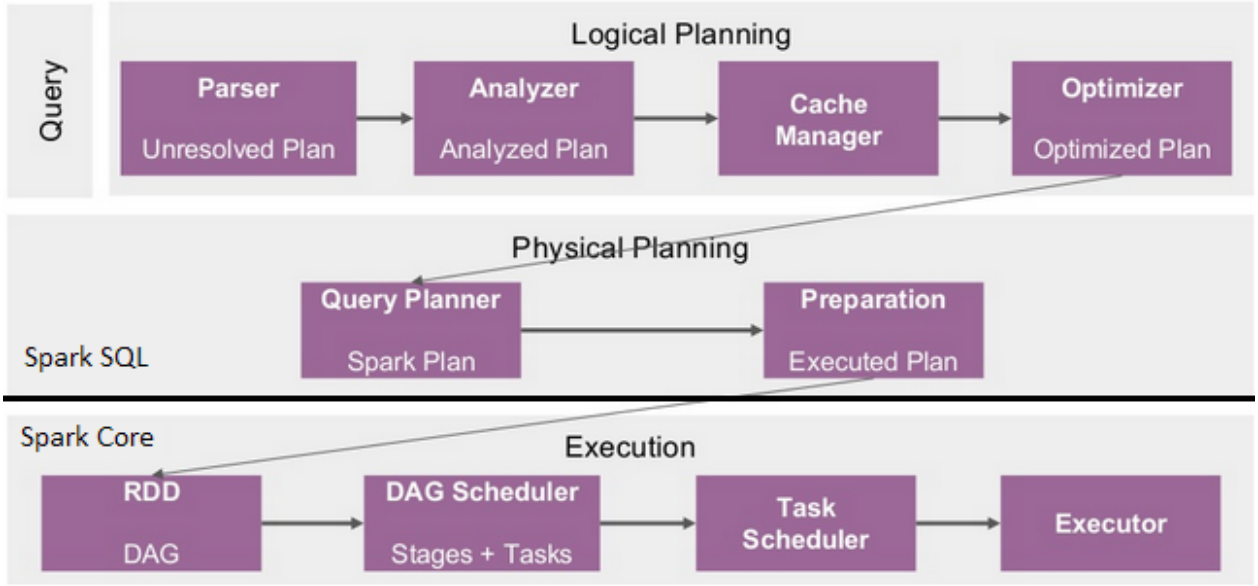


Figure 4.2.: Optimization and Execution Flow (Vrba 2019)

After an application has started running, it is split into `Jobs`, with each job corresponding to an action on a Dataframe. Their execution is done in two stages, and the first is governed by Spark's SQL module, where the jobs are organised into `Queries`. From the user code a Logical Plan is created, which is then analyzed and optimized by Spark's Catalyst optimizer. Finally, a Physical Plan is created which specifies how the Optimized Logical Plan can be executed on the cluster most efficiently, and is the bridge to the next stage. In the second stage, governed by Spark

Core and its DAG scheduler, the job is further split into `Stages` that represent it as a directed cyclic graph with the nodes being the data and the edges - the transformations to be applied on it. The boundaries between stages represent shuffle operations. Ultimately each stage is broken into tasks that are sent to executors that enact them on the underlying RDD. If we were working with a RDD only the second stage of this process would take place (Vrba 2019).

Spark provides a monitoring interface for both applications running in real time, and finished ones (i.e. the Spark History Server). There one can review both the environment - the Spark configuration used, Executors with their characteristics, etc. - and the aforementioned execution plans, jobs and stages. It is the primary tool for observing and tuning a Spark application.

4.2. Persistence

When solving the N-body problem, all algorithms follow an iterative cyclic pattern: the force or acceleration is calculated from the current state, and then the new state is calculated from that. This can be seen in the `simulation` and `integrator` scripts where the variables `df_clust` and `df_F` are continuously updated and reused. This pattern can lead to a major performance issue in Spark: with each iteration the lineage of the dataframes will grow, and include that of the previous ones it is based upon. This would lead to larger and larger query plans that take more and more time to optimize, and at each new iteration the previous ones would be recomputed. An example of the first four queries in the leapfrog integrator when it exhibits this behaviour is shown in figure 4.3. While each of these actions should take uniform time to complete, the execution time has doubled by the fourth iteration. Another good indicator of the issue is that each query takes more and more jobs to complete.

The way to resolve this is by using Spark's checkpoint system on the dataframes subject to such an algorithm. It comes in the form of either `dataframe =`

ID	Description	Submitted	Duration	Job IDs
3	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2021/04/11 20:00:21	25 s	[54][55][56][57][58][59][60][61][62][63][64][65][66][67] [68][69][70][71][72][73][74][75][76][77][78][79][80][81]
2	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2021/04/11 20:00:00	18 s	[31][32][33][34][35][36][37][38][39][40][41][42][43][44] [45][46][47][48][49][50][51][52][53]
1	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2021/04/11 19:59:46	13 s	[13][14][15][16][17][18][19][20][21][22][23][24][25][26] [27][28][29][30]
0	localCheckpoint at NativeMethodAccessorImpl.java:0 +details	2021/04/11 19:59:33	12 s	[0][1][2][3][4][5][6][7][8][9][10][11][12]

Figure 4.3.: SQL queries slowdown

`dataframe.checkpoint()` or `dataframe = dataframe.localCheckpoint()` with the former saving the Dataframe to memory to disk, and the latter - in executor memory. (Apache Software Foundation 2020) These both truncate the logical plan of the Dataframe and thus prevent it from growing. Local checkpoints are faster but consider unsafe when compared to their counterparts. Checkpoints are also eager by default, and trigger an action on the checkpointed Dataframe. In the `Simulation` class' `advance` function, the particle data is checkpointed at the end of every iteration. The `IntegratorLeapfrog` class needs and additional checkpoint on the force Dataframe, as it is reused between iterations separately from the cluster data (fig. 4.3 resulted from running that integrator with this second checkpoint omitted).

Another property of checkpointing is that it allows the Dataframe to be easily reused once computed. This functionality is shared with the more popular `cache` and `persist` methods, that save the Dataframe in memory, in the same way as `localCheckpoint` does, but do not truncate its lineage. When checkpointing is not required, caching can be used to speed up execution instead. That is the case in `IntergratorEuler2` and `IntegratorRungeKutta4` where the caching the intermediate force dataframes helps speed up the process. It should be noted that both checkpointing and caching have overhead, and if used inappropriately they can slow an application down rather than speeding it up. When implementing new integrators or algorithms of similar nature,

it can be beneficial to run small amounts of iterations with and without them, and to compare the resulting plans and execution times in Spark's monitoring interface.

4.3. Physical Plan Optimization

Regardless of the optimization that takes place in Spark, the user code has a profound impact on what is actually executed on the cluster and the efficiency of the application. This is more pronounced than when using pure python where different code used to achieve the same effect can be only marginally different in terms of speed. This is the result of a combination of Spark's lazy execution and physical plan creation which can cause minor differences in the user code to produce radically different physical plans.

We'll illustrate that with an example of two different versions of the Euler integrator's `advance` function. In the first version, 4.1, the process of advancing the particles' positions and velocities is split in two with `step_r` and `step_v` functions that use the current force from `df_F` to update them. Afterwards they are joined, and the resulting Dataframe has its columns reordered so it matches the original cluster schema (`schemas.clust`). The version of the script actually used, 4.2, uses a single step function for both position and velocity instead. It is not immediately obvious that one would prefer the second version over the first - after all one may want to get the results for just one or the other, and what's more in `integrators.IntegratorEuler2` we need just the updated positions to get the force at the end of the time step so the separate functions are easy to reuse once this integrator is inherited.

Listing 4.1: Initial version of integrators.InegratorEuler.advance

```
def advance(self, df_clust):
    df_F = self.calc_F(df_clust)
    df_v, df_r = self.step_v(df_clust, df_F), self.step_r(
        df_clust, df_F)
```

```
    df_clust = df_r.join(df_v, "id")
    # bring order back to schema
    df_clust = df_clust.select('id', 'x', 'y', 'z',
                               'vx', 'vy', 'vz', 'm')

    return (df_clust, self.dt)
```

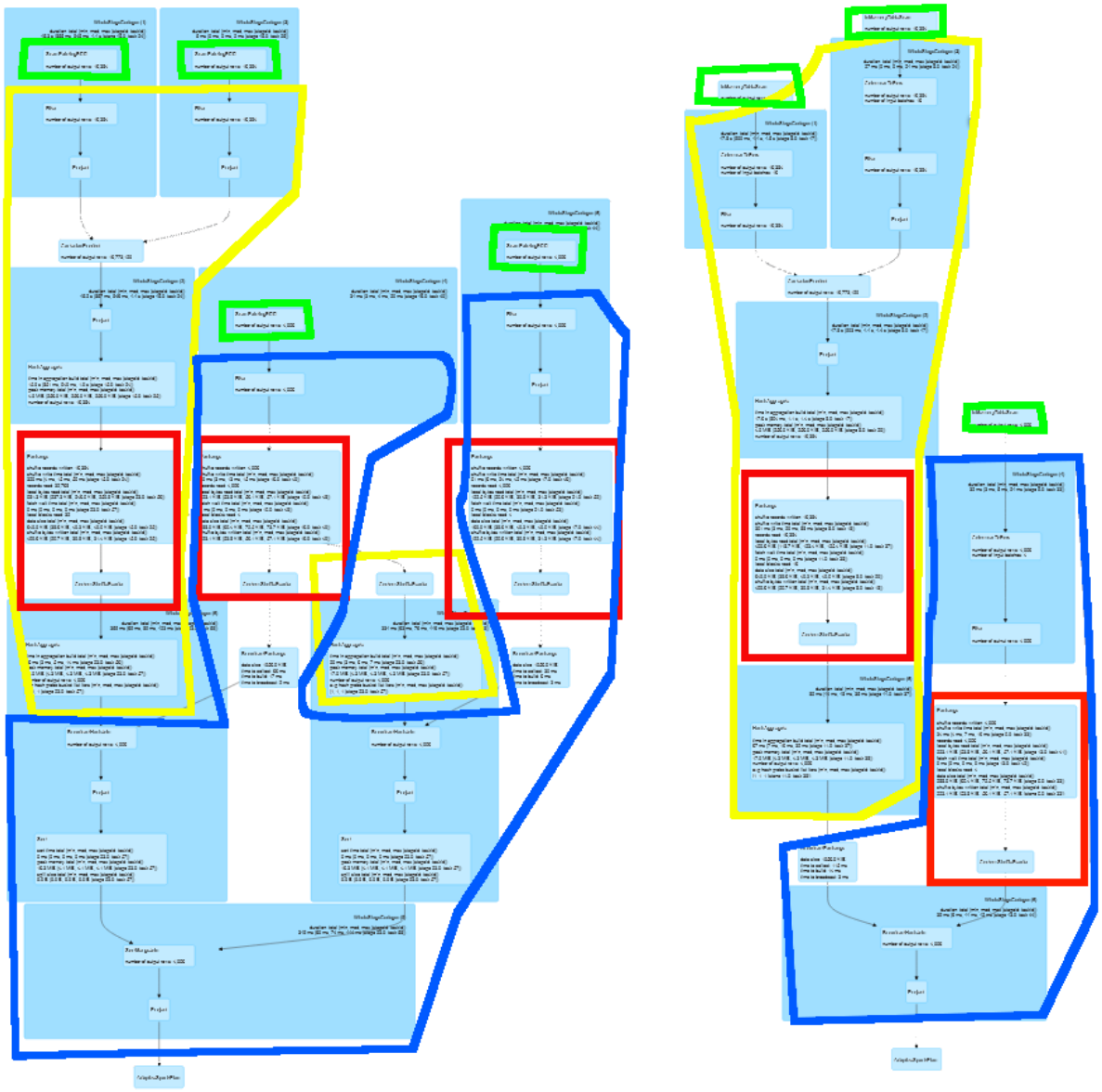
Listing 4.2: Final version of integrators.InegratorEuler.advance

```
def advance(self, df_clust):
    df_clust = df_clust.cache()
    df_F = self.calc_F(df_clust)

    df_clust = self.step(df_clust, df_F)

    return (df_clust, self.dt)
```

The obvious advantage that the second version of the function has it that saves one join operation, but the main improvement can be seen when comparing the physical plans that they both produce. As stated before the monitoring interface is invaluable in reviewing the physical plans, and while it provides a text tree, it also gives a visual representation which can often be more intuitive to follow. Both physical plans for one iteration are displayed in figure 4.4, shrunk down to size, and with the following areas highlighted:

- green is reading the cluster data (they are always the leaves of the graph)
- yellow is computing the force (per (1.1))
- blue is computing the new positions and velocities (per (2.2) and (2.1))
- in red are the expensive shuffle operations that take place in the previous two, these are denoted by an "Exchange" in the query plan

Figure 4.4.: Phyiscal plan produced by 4.1 (left) versus the one created by 4.2 (right)



While it is pretty obvious that the second one is more efficient lets break down what happens in them. In the left plan, produced by 4.1, since we are computing the speeds and velocities separately, the cluster data is read twice at the beginning of the blue branches. It than has to be joined to the Force Dataframe which triggers a shuffle operation in each branch. The final part of the force calculation is also repeated twice -

the final step for computing the force acting on each particle is an aggregation operation which is typically executed in three stages - first the data is aggregated within the different partitions, then the results are exchanged between partitions (hence the shuffle operation in the yellow block) and finally the exchanged data is aggregated as well. This very last step of the aggregation is repeated twice at the two final yellow blocks. In contrast, the two branches are ”merged” together when the positions and velocities are updated in the same function, in 4.2. This cuts the number of operations performed nearly in half.

This shows the immense effect changes to the user script have on the action plan. In the majority of other languages these two versions of the same script would have been a lot closer in terms of efficiency. Same as with the previous section about persistence, it illustrate how reviewing the monitoring interface is an integral part of creating a spark application. We can also draw the conclusion that simpler and more composite code is handled better by Spark, although it is hard to generalize this to every application.

4.4. Partitioning

Another aspect of a Spark application that needs to be tuned is the number of partitions that the data is split in. If the data is split in too few partitions, then Spark's parallelism will not be fully utilised. On the other hand if it is split in too many partitions, execution will slow down as a lot of small parts will need to be transmitted between executors. A commonly given guideline for the number of partitions should be two to three times the number of executor cores available, but that can only apply when working with a very large amount of data. In practice the best results can be achieved when tuning the partitions to each data set and/or algorithm individually.

The data in our project is partitioned by the id of each particle when it is read in `main.py` so the number of partitions is controlled by the spark configuration via the `spark.default.parallelism` and `spark.sql.shuffle.partitions` attributes. The

latter of these controls the number of partitions for shuffle operations which is set separately from the default. After a shuffle operation, the number of partitions for a Dataframe will become double the shuffle partitions value, and in older versions of Spark it would be advisable to bring them back to a smaller value using either the `repartition` or `coalesce` methods. As of Spark 3.0, however, there exists an automatic system to adjust the partitions number after a shuffle, and so these are not present in our integrator scripts. Instead they should be run with the spark configuration options `spark.sql.adaptive.coalescePartitions.enabled` and `spark.sql.adaptive.enabled` set to `true` (Apache Software Foundation 2020).

To determine the optimal number of partitions for the integrators a smaller number of iterations were ran with different numbers of particles on a cluster with 25 executor cores. In the following tables the total runtime of each application in seconds is divided by the number of force evaluations (the most computationally expensive part in every algorithm), and thus can also provide some comparison between the speed of the different integrators. In order to find the optimal number of partitions we are starting from the best number for the previous amount of particles tested, and increasing it until the time taken starts rising again. The best times are highlighted and the corresponding number of partitions is used when running simulations with that number of particles.

Table 4.1.: Average runtime in seconds per single force evaluation depending on different number of partitions and particles

partitions \ particles	512	1024	2048	4096	8192	16384	32768
Leapfrog Integrator							
1	1.015						
2	0.911	1.096					
4	0.950	0.999	1.150	1.647	3.905	12.703	
8	0.964	1.025	1.071	1.532	3.202	10.121	42.314
16		1.140	1.228	1.584	3.125	9.118	35.882
24		1.242	1.372	1.703	3.299	9.088	34.687
32						9.613	35.485
Euler Integrator							
1	0.742						
2	0.641	0.713					
4	0.655	0.707	0.954	1.944	6.036	12.433	
8	0.717	0.765	0.941	1.865	5.610	20.251	40.237
16		0.928	1.130	1.630	3.679	13.998	50.591
24		1.197	1.490	2.058	2.951	9.090	34.437
32						11.307	41.868
Improved Euler Integrator							
1	0.648						
2	0.506	0.650					
4	0.548	0.585	0.738	1.315	3.333	12.153	
8	0.569	0.587	0.712	1.163	3.151	10.465	38.396
16		0.749	0.812	1.189	2.749	8.687	33.063
24		0.919	0.992	1.394	2.790	8.405	32.448
32						9.294	35.457
Runge-Kutta Integrator							
1	0.772						
2	0.656	0.773					
4	0.666	0.711	0.809	1.418	3.248	11.700	
8	0.698	0.732	0.820	1.241	2.938	9.885	36.599
16		0.900	0.953	1.337	2.801	8.623	32.920
24		1.160	1.225	1.557	2.959	8.614	31.479
32						9.530	33.197

As seen in the above table, while the general pattern is the same, the algorithm does have some impact on the optimal number of partition regardless of the fact that the force evaluation (inherited from the `InegratorBase` class) is the same. Another observation is that the optimal times per force evaluation for the different integrators, as well as the rate that the time increases for the different number of particles is similar. This indicates that at the very least no integrator is implemented more inefficiently than any other one.

4.5. Other Considerations

In this final section about Spark, we'll discuss some miscellaneous findings starting with some more configuration options. Serialization plays an important role as data that is to be transferred between executors is serialized. As such the configuration for `spark.serializer` is set to `org.apache.spark.serializer.KryoSerializer`, since it is faster than the default Java serializer (Apache Software Foundation 2020). Secondly, when running the simulations, sometimes it was observed that the garbage collection takes a lot of time (GC times are also present in the monitoring interface). These improved when using the G1 garbage collector, and reducing the heap space occupancy that triggers garbage collection. This is done in the configuration by setting extra Java options via `spark.executor.extraJavaOptions` and `spark.driver.extraJavaOptions` to include `-XX:+UseG1GC` and `-XX:InitiatingHeapOccupancyPercent=35`. The tuning of these parameters is suggested in Oracle 2016.

Another important note is about using Spark user defined functions, or UDF-s. Spark provides the option for users to define custom functions for transforming each row of a Dataframe. While these might sound tempting to use, they should be avoided and if possible to use build-in Spark transformations instead. The issue with them is that they are treated as black boxes and not optimized by Spark's optimizer which is crucial for

performance.

Finally, one might wonder why the vectors for position and velocity are split into components in the data. Indeed, Spark does provides a vector data structure as a part of its machine learning library. However in practice that is less efficient to use than splitting them into components and transforming them with Spark SQL.

5. Running Simulations and Results

With the scripts validated and appropriate number of partitions chosen we move on to try and run some simulations on varying numbers of particles.

5.1. Environment

The simulations presented in this chapter were all ran on AWS' Elastic Map Reduce service (EMR). It allows easy deployment of Spark applications on a virtual cluster. The instances used are two c5.4xlarge models, which are optimized for computing, with 32GB memory and 16 cores each, with frequency of up to 3.4GHz. As it is recommended to leave one core per instance to run its OS (an Amazon version of Linux) and system processes, and the Spark driver also has to be allocated on one of them, we have 25 cores for executors, 5 cores for the driver and 2 left for the OS. Out of these we are allocating 5 executors with 5 cores each. All cluster settings are available as a json file (that can be loaded directly in EMR when creating a cluster) at `/emr/cluster_soft_settings_32g_16c.json`. When creating an EMR cluster, we also have to tell it to install the required third party python modules for all instances of the cluster. This is done by setting a bootstrap action for the cluster, which is also located in the emr folder at `bootsrap_cluster.sh`.

5.2. Simulations

The simulations that were ran on the above Spark environment consisted of using the included integrators to go from $t = 0$ to $t = 1$, with $\Delta t = 0.001$ for the different data sets, and save a snapshot at the final time. Every 0.1 time units, or 100 iterations, energy calculations were performed to track the accuracy of the simulations by logging the total energy error as per (2.8). The data used for the 512 to 16384 runs is from the nbabel (`/data/nbabel/`) data set, the data used for the 32768 (for the Euler and Leapfrog integrators) is from the NEMO (`/data/plummer/`) one. Firstly we'll present the results of that diagnostic for the different numbers of particles:

Figure 5.1.: Energy errors for the Euler integrator

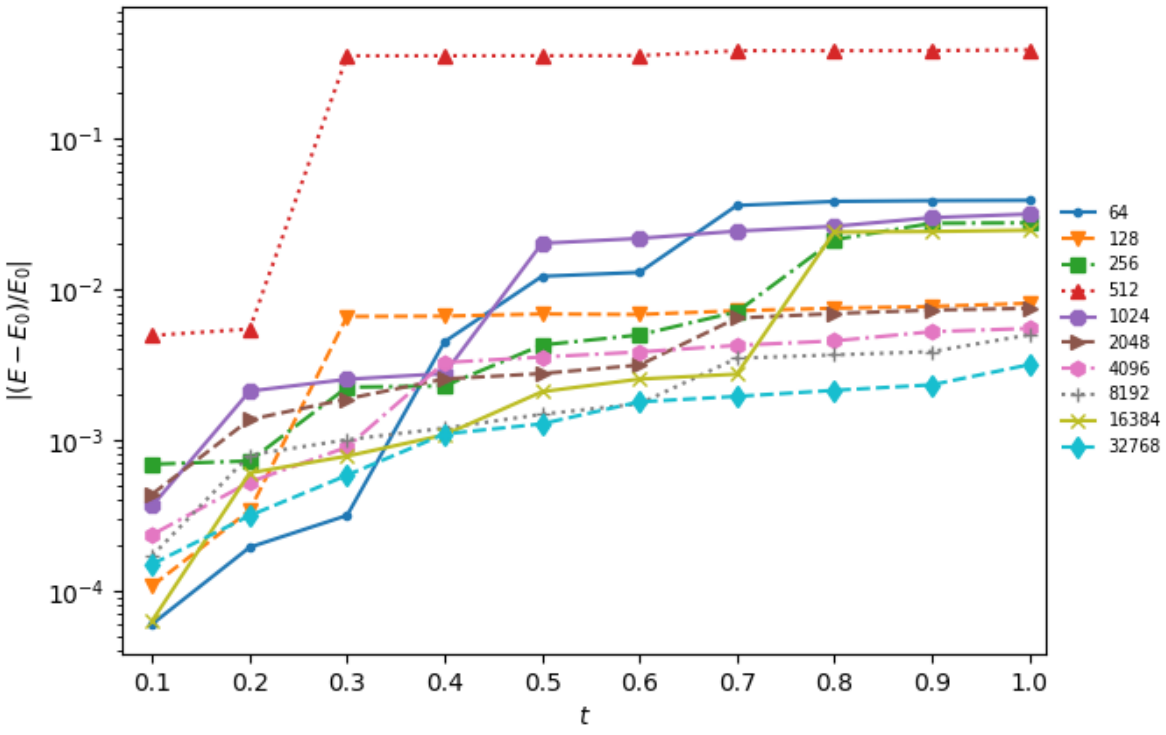


Figure 5.2.: Energy errors for the improved Euler integrator

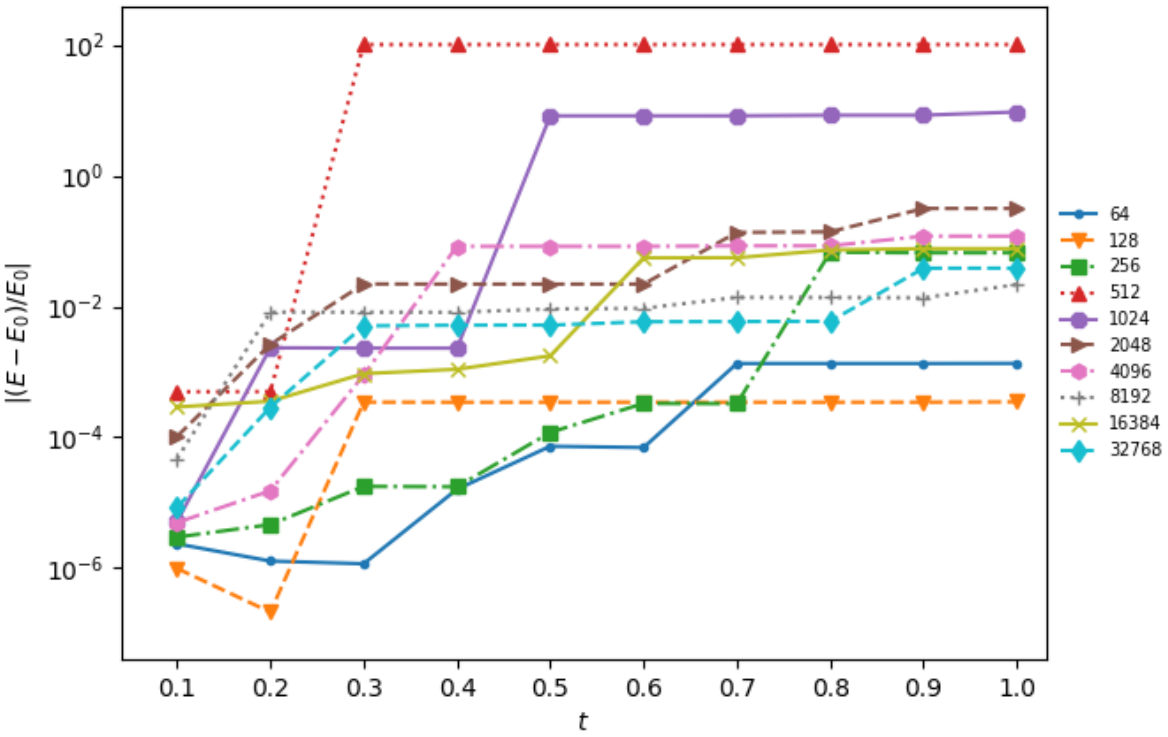


Figure 5.3.: Energy errors for the Leapfrog integrator

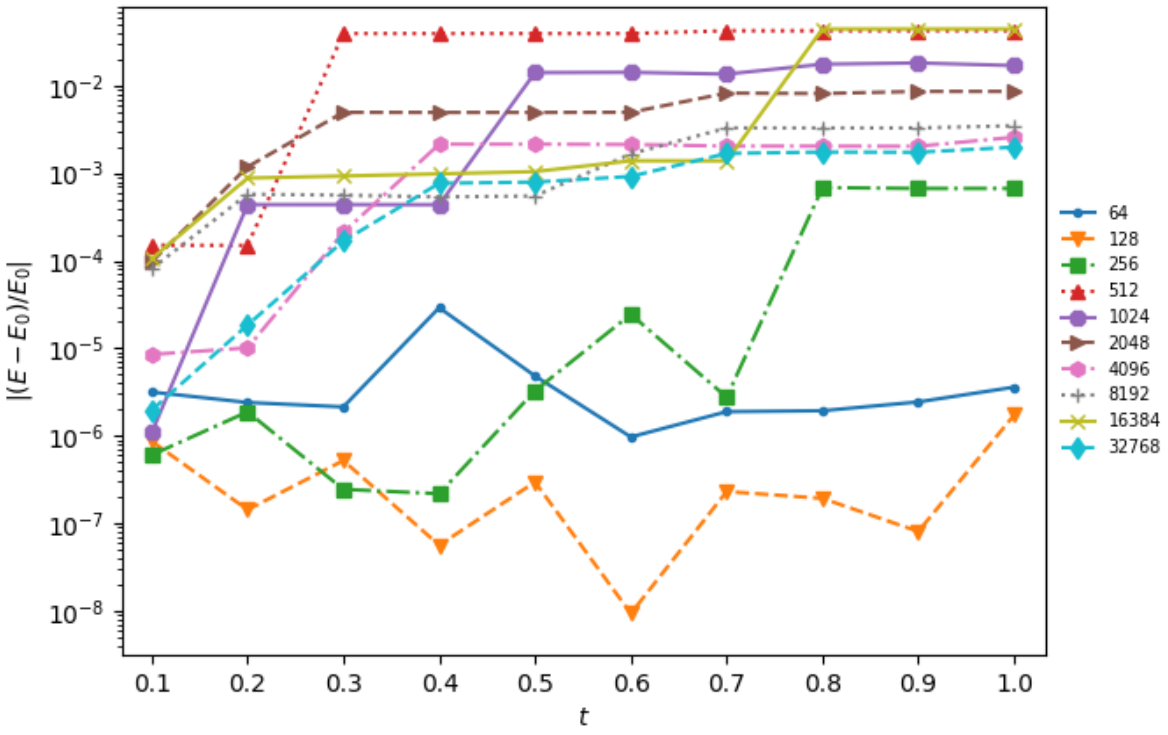
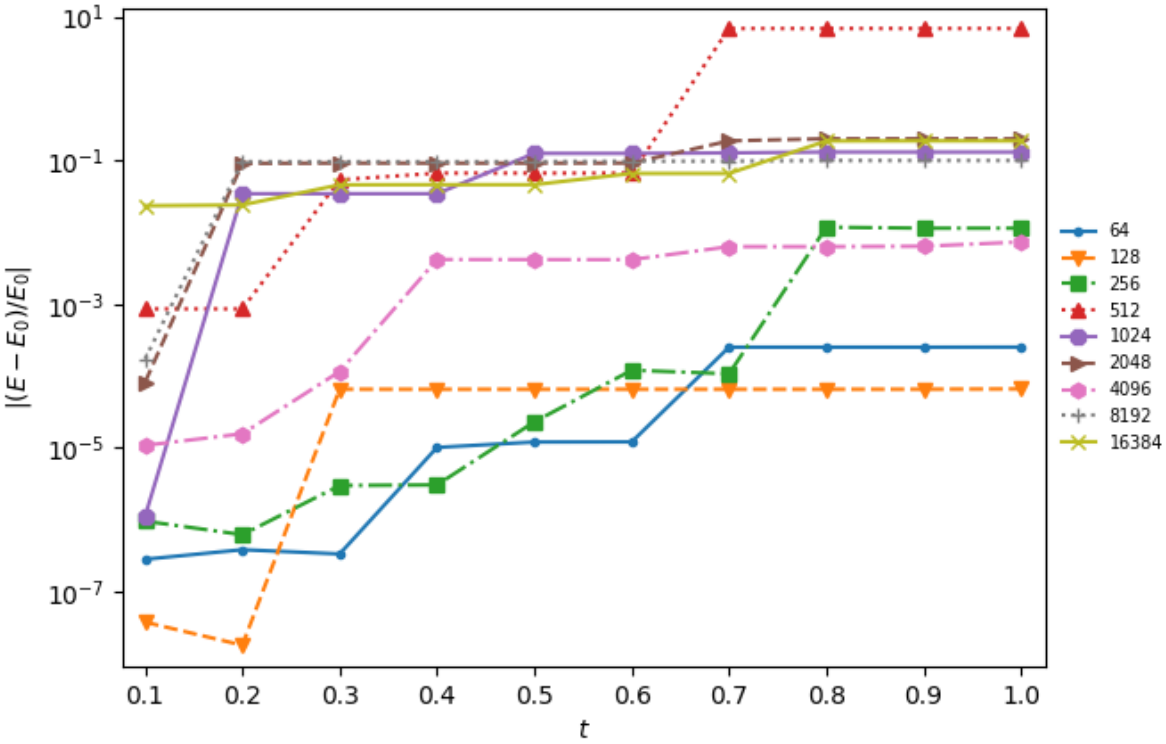
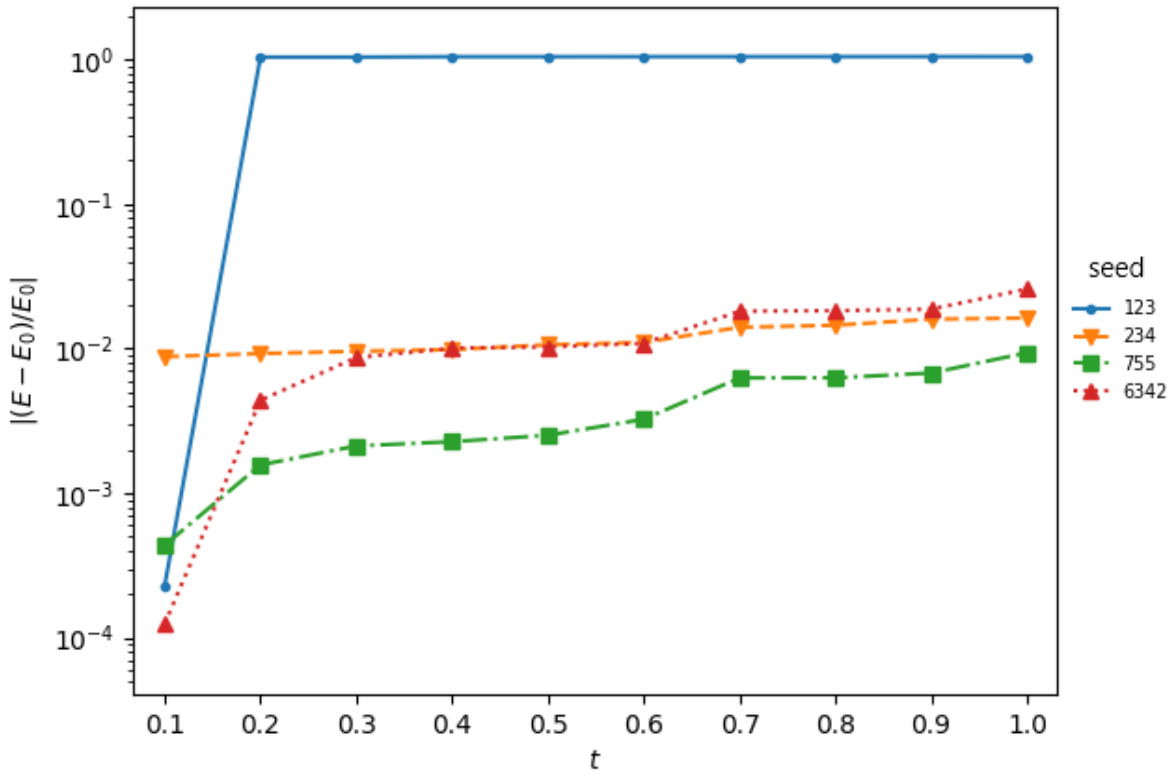


Figure 5.4.: Energy errors for the Runge Kutta integrator



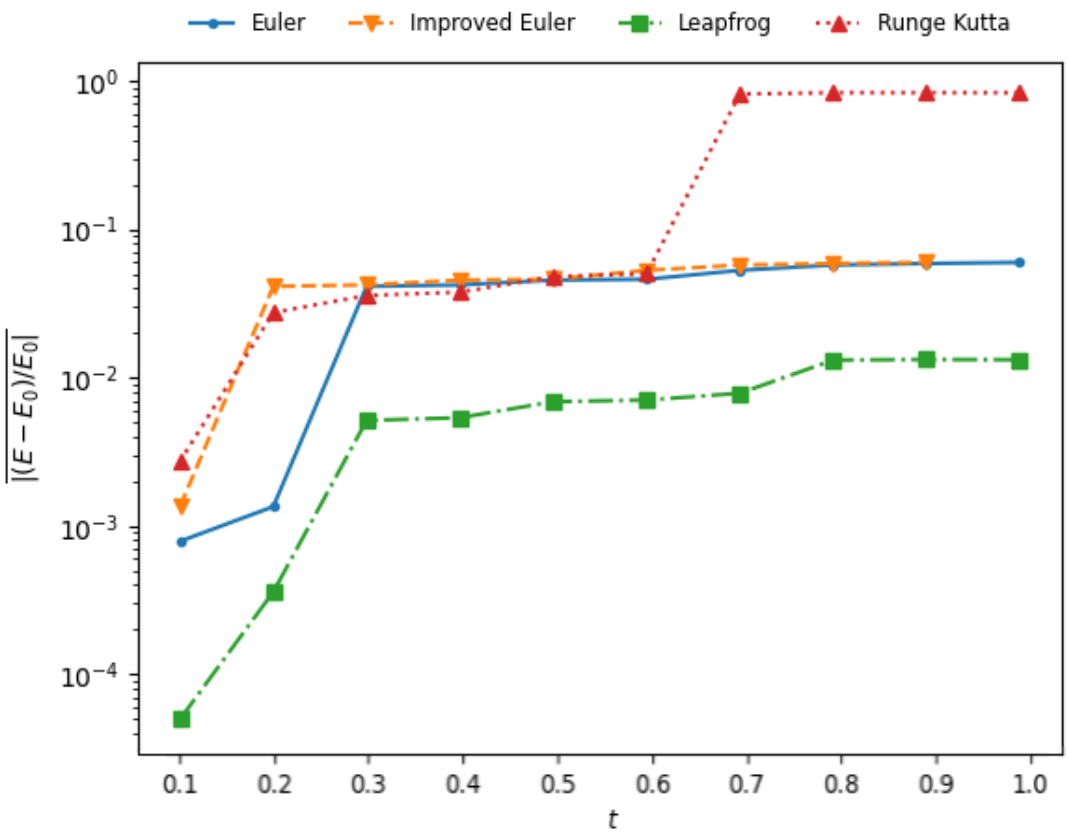
The first thing that jumps out from these figures is that for some of the data sets there is a serious degradation of accuracy in the span of just 100 iterations, for example for the one with 512 particles. What this means is that for the evolution of that particular data set, there exist events (like closer encounters for example) that the integrator cannot handle with the value of Δt used. Going back to the example of the 512 data set, if we compare several randomly generate by NEMO with using different seeds, we can see that the overall results are more in line with the overall accuracy displayed:

Figure 5.5.: Energy errors for 512 particles, generated by NEMO with different seeds, Euler integrator



Data sets with higher numbers of particles are more stable in this regard, as interactions between individual particles have a lesser effect on the entire system. The improved Euler and Runge-Kutta integrators exasperate this types of errors, while the Leapfrog one mitigates them the most. To get a clearer picture of the change in energy errors, we can also look at the average error for the different numbers of particles.

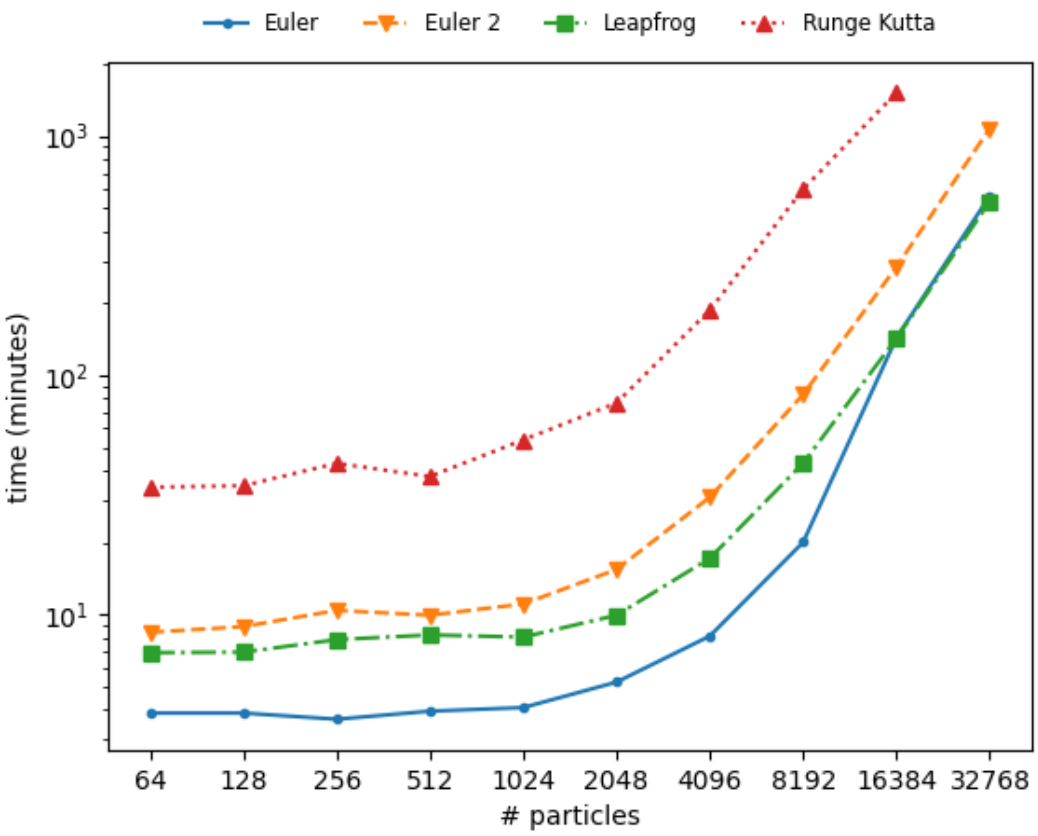
Figure 5.6.: Average energy errors over time



The other behaviour we are interested in is how the number of particles impacts the execution time for the different integrators. As expected, the execution time is growing exponentially with the increase in input data size. Out of the lower order integrators, the leapfrog integrator seems the most favorable as it is second order while being about twice as fast as the improved Euler one due to having half the force evaluations. As the particle number increases it also starts to equal the performance of the first order Euler integrator. The fourth order Runge-Kutta integrator is 4.5 to 10 times slower than Leapfrog.

Ultimately the Leapfrog integrator comes out on top in terms of both accuracy and performance in this test scenario, mostly due to it handling the problematic data better.

Figure 5.7.: Time taken for the different data sets



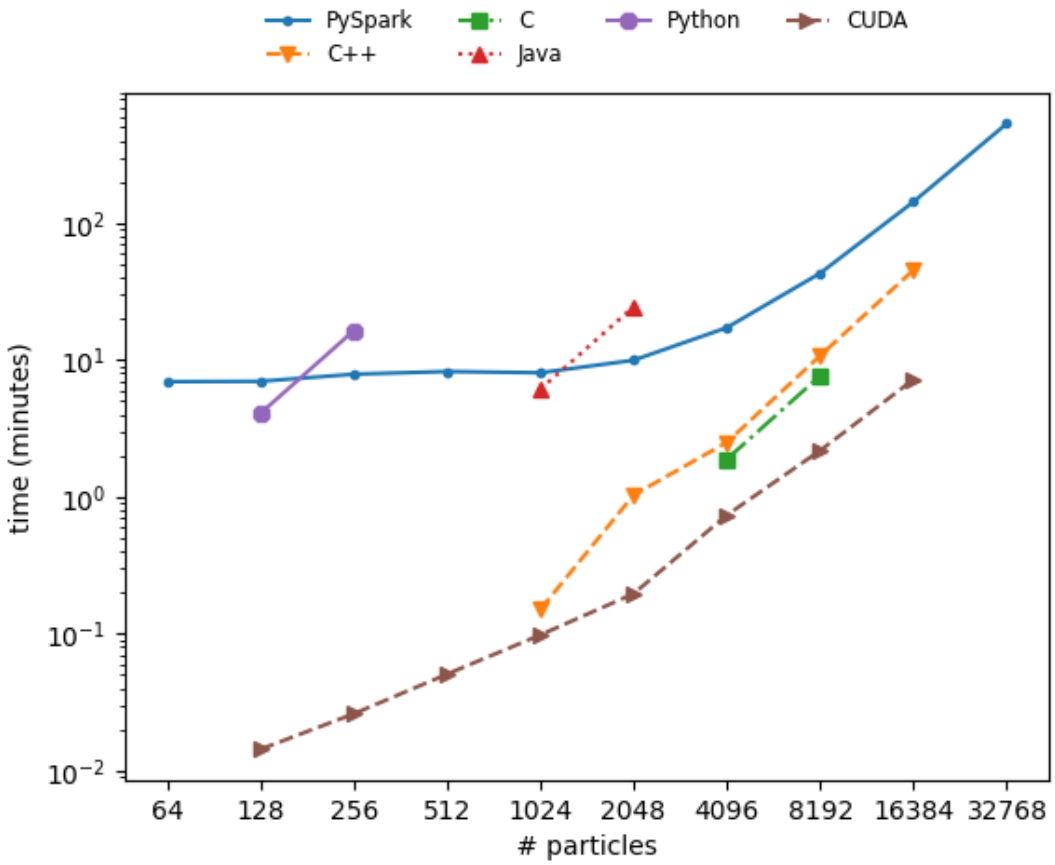
This is most likely due to it being a predictor-corrector method, while the others are explicit ones. Similarly the Hermite method, another predictor-corrector is used by Aarseth for his first two direct NBody scripts. A possible explanation for this is that the leapfrog integrator is more stable meaning that the energy errors do not accumulate over time (McMillan n.d.). Another thing we can note from 5.6 is that the four datasets take nearly the same time to process despite the increasing number of particles. This indicates that Spark's setup overhead can strongly influence the performance for lower number of particles, that introduces a minimum execution time which always needs to pass in order to complete the simulation.

5.3. Comparison to Other Languages

In order to compare Spark's performance to other programming languages, we will compare the performance of our Leapfrog integrator to implementations of the same algorithm collected by members of Leiden university's Computational Astrophysics mailinglist on the NBabel website. It's assumed that since we are comparing the same algorithm, the difference in performance will be indicative of how Spark compares to that programming language for running N-body simulations as a whole. The scripts were ran on Ubuntu 18.04 with AMD Ryzen 5 2500U (8 threads, boost clock up to 3.6GHz), and 8GB of RAM.

Starting with higher level languages, it is natural to compare our PySpark code to

Figure 5.8.: Leapfrog integrator time taken for 1000 iterations for different languages



Python. The NBabel python v2 code was used for this, which is based on the numpy module, the de facto standard for computation in Python. The python implementation takes more than twice as long to simulate just 256 particles as the Spark one. Thus Spark can safely be deemed more efficient than pure python without further testing. Java fares a bit better than pure python, with it being able to simulate the 1000 iterations for a data set of 1024 particles for 6 minutes and 2 seconds, which is 2 minutes faster than the Spark implementation. However, just doubling that size makes it take 24 minutes which is nearly three times slower then our implementation.

When using lower level languages, the results are quite different. The C++ code, when compiled with the default options, performs worse than the Spark implementation. On the other hand, When compiled with `-O3` optimization flag (the NBabel website specifies `-O4`, but according to gcc's documentation O3 is the maximum optimization level, see GCC team 2021), it outperforms it across the board. However, when we attempt to run this C++ code for the thirty two thousand particle data set, the process is killed within seconds because it runs out of memory. This does not happen when running the Spark code on the same machine (albeit than it being a lot slower then when ran on the cluster). the The C code performs better than the C++ one for 4096 and 8192 particles, when compiled with the same optimization, though not by a large margin. It also gets killed when attempting to run for 32768. What's more, it fails the energy calculation for 16384 by returning `nan` for the energy diagnostic. While it's beyond our scope to determine why this happens, it is most likely an issue with the implementation, as the C++ and Spark codes handle this data correctly. Finally, when it comes to multiprocessing Spark definitively loses out against CUDA[1] as far as speed concerned. This is mostly to be expected, as Spark's target is to handle large amounts of data, and thus its focus is more on stability.

[1]The CUDA code was not ran for this project, the results presented are from the NBabel website

6. Conclusion

This work features a foundation for stellar dynamics tool suite for Apache Spark, that contains some basic direct integration methods, energy analysis tools, and general utilities for working with Spark Dataframes. It is extensible so as to allow addition of new integrators. During the creation of this set of tools, some interesting observations were made for how to better tackle working with iterative algorithms in Spark, in particular how to truncate its action plans via checkpoining, which would be an integral part of any such algorithm. While Spark is not as fast as other multiprocessing frameworks, due its overhead which facilitates handling terabytes of data, it is reliable and robust. Future avenues of actions would be to try and implement some indirect simulation methods, and more importantly find ways to further improve performance. Due to the great computational performance of the CUDA code for N-Body simulations, it may be advisable to explore the addition of GPU acceleration, which is available for Spark with Nvidia's RAPIDS accelerator.

Bibliography

Aarseth, S. J., M. Henon, and R. Wielen (Dec. 1974). "A Comparison of Numerical Methods for the Study of Star Cluster Dynamics". In: 37.1, pp. 183–187.

Aarseth, Sverre J. (2003). *Gravitational N-Body Simulations*. Cambridge University Press.

Apache Software Foundation (2020). *Apache Spark Documentation*. Version 3.0.1. URL: https://spark.apache.org/docs/3.0.1/.

Castle - Stellar Dynamics and Computational Astrophysics (2010). Leiden University. URL: http://castle.strw.leidenuniv.nl/.

DESI-Collaboration (2016). *The DESI Experiment Part I: Science,Targeting, and Survey Design*. arXiv: 1611.00036 [astro-ph.IM].

GCC team (2021). *GCC online documentation*. Options That Control Optimization. Free Software Foundation, Inc. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

Heggie, D.C. and R.D. Mathieu (1986). "STANDARDISED UNITS AND TIME SCALES". In: *Lecture Notes in Physics*. Ed. by W. Beiglböck. Vol. 267. Institute for Advanced Study. Springer-Verlag, pp. 233–237.

Heggie, Douglas C. (2005). "The Classical Gravitational N-Body Problem". In: arXiv: astro-ph/0503600v2. URL: https://arxiv.org/abs/astro-ph/0503600v2.

Hut, Piet and Jun Makino (2007). "Moving Stars Around". URL: http://www.artcompsci.org/kali/pub/msa/volume1.pdf.

McMillan, Steve (n.d.). *The Leapfrog Integrator*. URL: http://www.physics.drexel.edu/~steve/Courses/Comp_Phys/Integrators/leapfrog/.

Oracle (2016). *Communications WebRTC Session Controller System Administrator's Guide*. Version 7.2. Tuning JVM Garbage Collection for Production Deployments. URL: https://docs.oracle.com/cd/E69505_01/doc.72/e69506/cnf_jvmgc.htm#WSEAD414.

Pasquato, Mario (2017). *Star Cluster Simulations*. URL: https://www.kaggle.com/mariopasquato/star-cluster-simulations.

Roa, Javier et al. (2020). *Moving Planets Around*. MIT Press.

Teuben, P.J (2019). *NEMO - A Stellar Dynamics Toolbox*. URL: https://teuben.github.io/nemo/.

Vrba, David (2019). *Physical Plans in Spark SQL*. URL: https://www.youtube.com/watch?v=99fYi2mopbs.

Appendix A - Script API Document

A.1 cluster module

`cluster.calc_E(`*df_clust*, *G=1*, *W=0*`)`

Calculate the total energy of the cluster in our case there is no external energy (W)

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **G** (*float, optional*) – gravitational constant to use, defaults to 1
- **W** (*float, optional*) – external energy, defaults to 0

Returns E

Return type float

`cluster.calc_J(`*df_clust*`)`

Calculate the total angular momentum of the cluster:

```
    N
J = Σ r_i × m_i * v_i
   i=1
```

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns J, broken into components

Return type {[type]}

`cluster.calc_T(`*df_clust*, *G=1*`)`

Calculate the total kinetic energy of the cluster:

```
        N
T = 1/2*Σ m_i*v_i^2
       i=1
```

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **G** (*float, optional*) – gravitational constant to use, defaults to 1

Returns T

Return type float

`cluster.calc_U(`*df_clust*, *G=1*`)`

Calculate the total potential energy of the cluster:

```
  N   N   G*m_i*m_j
- Σ   Σ  -----------
 i=1 j>i |r_i - r_j|
```

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **G** (*float, optional*) – gravitational constant to use, defaults to 1

Returns U

Return type float

`cluster.calc_cm(`*df_clust*`)`

Calcuate the center of mass of the cluster:

```
          N
R = 1/M *Σ m_i*r_i
        i=1
```

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns center of mass

Return type pyspark.sql.types.Row

`cluster.calc_rh(`*df_clust*, *cm*`)`

Calculate the half-mass radius of the cluster

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **cm** (*iterable ([x, y, z])*) – cluster center of mass

Returns half-mass radius

Return type float

A.2 integrator_base module

`class integrator_base.IntegratorBase(dt, G=1)`

Bases: `abc.ABC`

abstract class, contains the force calculation that is used by all integrators

Parameters

- **dt** (*float*) – time step for the simulation
- **G** (*float, optional*) – gravitational constant to use, defaults to 1

`abstract advance(df_clust)`

Advance by a step

Has to return the same value across all implemetations as it's used in Simulation

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns the new positions and velocities of the particles of the cluster, and time passed

Return type tuple (pyspark.sql.DataFrame, with schema schemas.clust, float)

`calc_F(df_clust)`

Calculate the force per unit mass acting on every particle:

```
          N    m_j*(r_i - r_j)
F = -G * Σ   ------------------
        i!=j   |r_i - r_j|^3
```

r - position m - mass G - Gravitational Force Constant

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns force per unit of mass

Return type pyspark.sql.DataFrame, with schema schemas.F_id

`calc_F_cartesian(df_clust)`

The pairwise calculations to be used for calculating F can be used to check which particle(s) contribute the most to the effective force acting on a single one

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns the forces acting between every two particles

Return type pyspark.sql.DataFrame, with schema schemas.F_cartesian

A.3 integrators module

class integrators.**IntegratorLeapfrog**(*dt*, *G=1*)

Bases: *integrator_base.IntegratorBase*

Second order leapfrog integrator

advance(*df_clust*)

advance by a step

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data

Returns the new positions and velocities of the particles of the cluster, and time passed

Return type tuple (pyspark.sql.DataFrame, with schema schemas.clust, float)

step_r(*df_clust*)

Calculate r

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns the new position of each particle

Return type pyspark.sql.DataFrame, with schema schemas.r_id

step_v(*df_clust*, *df_F_t*)

Calculate v based on the average of two forces:

```
v_i(t) = 1/2(F_i(t0) + F_i(t))* t + v_i(t0)
```

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **df_F_t** (*pyspark.sql.DataFrame, with schema schemas.F_id*) – force per unit of mass acting on each particle

Returns the new position of each particle

Return type pyspark.sql.DataFrame, with schema schemas.r_id

class integrators.**IntegratorRungeKutta4**(*dt*, *G=1*)

Bases: *integrator_base.IntegratorBase*

Fourth order Runge-Kutta integrator. Splits the interval t in four stages.

advance(*df_clust*)

advance by a step:

```
x_n+1= x_n +  t/6 * (k1 + 2*k2 + 2*k3 + k4)
k1 = f(t_n, x_n);
k2 = f(t_n +  t/2, x_n + k1* t/2);
k3 = f(t_n +  t/2, x_n + k2* t/2);
k4 = f(t_n +  t, x_n + k3* t)
```

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data

Returns the new positions and velocities of the particles of the cluster, and time passed

Return type tuple (pyspark.sql.DataFrame, with schema schemas.clust, float)

step(*df_clust*, *df_F*, *mod=1*)

Calculate new positions & velocities from F with a modifier added to t

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data
- **df_F** (*pyspark.sql.DataFrame, with schema schemas.F_id*) – force per unit of mass acting on each particle
- **mod** (*double, optional*) – modifier to apply to t, defaults to 1

Returns cluster data after force is applied

Return type pyspark.sql.DataFrame, with schema schemas.clust

class integrators.**IntergratorEuler**(*dt*, *G=1*)

Bases: *integrator_base.IntegratorBase*

First order integration method that advances all particles at a constant time step (dt). Calculates effective force per unit mass for each particle with O(n^2) complexity.

advance(*df_clust*)

Advance by a step

Parameters **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass

Returns the new positions and velocities of the particles of the cluster, and time passed

Return type tuple (pyspark.sql.DataFrame, with schema schemas.clust, float)

step(*df_clust*, *df_F*)

combination of setp_r and setp_v for more efficient computation

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **df_F** (*pyspark.sql.DataFrame, with schema schemas.F_id*) – force per unit of mass acting on each particle

Returns the new velocity of each particle

Return type pyspark.sql.DataFrame, with schema schemas.clust

step_r(*df_clust*, *df_F*)

Calculate r for a single timestep t, dt = t - t_0:

```
r_i(t) = 1/2*F_i* t^2 + v_i(t0)* t + r_i(t0)
```

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **df_F** (*pyspark.sql.DataFrame, with schema schemas.F_id*) – force per unit of mass acting on each particle

Returns the new position of each particle

Return type pyspark.sql.DataFrame, with schema schemas.r_id

step_v(*df_clust*, *df_F*)

Calculate v for a single timestep t, dt = t - t_0:

```
v_i(t) = F_i* t + v_i(t0)
```

Parameters

- **df_clust** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position, velocity, and mass
- **df_F** (*pyspark.sql.DataFrame, with schema schemas.F_id*) – force per unit of mass acting on each particle

Returns the new velocity of each particle

Return type pyspark.sql.DataFrame, with schema schemas.v_id

class integrators.**IntergratorEuler2**(*dt*, *G=1*)

Bases: *integrators.IntergratorEuler*

Improved Euler integrator - 2nd order. After calculating F(1) for the current position, it calculates provisional coordiantes r(1) (this is like the original so far). It then uses r(1) to calculate the force F(2), and uses the average of F(1) and F(2) in the final calculation of coordinates and velocities for the step

advance(*df_clust*)

Advance by a step;

Parameters **insteps** (*int*) – number of steps to advance

Returns the new positions and velocities of the particles of the cluster, and time passed

Return type tuple (pyspark.sql.DataFrame, with schema schemas.clust, float)

A.4 simulation module

class simulation.**Simulation**(*cluster*, *integrator*, *ttarget*, *save_params*, *t=0*, *add_t_snap=False*, *dt_out=None*, *dt_diag=None*, *saveDiag=False*)

Bases: object

Set the simulation conditions

Parameters

- **cluster** (*pyspark.sql.DataFrame, with schema schemas.clust*) – cluster data - position and velocity [broken into componenets], and mass
- **integrator** (**integrator_base.Integrator**) – Integration method to use
- **ttarget** (*int*) – target time to reach when running the simulation
- **save_params** (*utils.SaveOptions / dict*) – parameters to pass to utils.save_df when saving output
- **t** (*int, optional*) – timestamp of the current cluster data, defaults to 0
- **add_t_snap** (*bool, optional*) – if true add timestamp to each particle on output, defaults to False
- **dt_out** (*int, optional*) – time interval between cluster snapshots, not saved if omitted
- **dt_diag** (*int, optional*) – time interval between energy outputs, not saved if omitted
- **saveDiag** (*bool, optional*) – if true save diagnostic to disk instead of printing to standard output, defaults to False

diag()

Save diagnostic information about the cluster energy

run()

Run the simulation with the chosen method until the target time is reached

Raises ValueError if the target time is already reached

snapshot()

Save a snapshot of the cluster

A.5 utils module

class utils.**NpAccumulatorParam**

Bases: pyspark.accumulators.AccumulatorParam

spark acumulator param for a numpy array

addInPlace(*v1*, *v2*)

Add two values of the accumulator's data type, returning a new value; for efficiency, can also update *value1* in place and return it.

`zero`(*initialValue*)
Provide a "zero value" for the type, compatible in dimensions with the provided *value* (e.g., a zero vector)

`class utils.SaveOptions`(*pth*, *fformat='parquet'*, *compression='gzip'*, ***kwargs*)
Bases: `dict`

Configuration for the utils.save_df method that can be easily reused:

```
save_df(df, fname, **SaveOptions)
```

Parameters

- **pth** (*str*) – path to the folder the file(s) is in
- **fformat** (*str, optional*) – format to save in, defaults to "parquet"
- **compression** (*str, optional*) – compression to use, defaults to "gzip"
- ****kwargs** – additional arguments to pass to save

`utils.clean_str`(*string*)
Clean a string from everything except word characters, replace spaces with '_'

Parameters **string** (*str*) – string to clean

Returns result

Return type str

`utils.df_add_index`(*df*, *order_col*)
Add an index column to a dataframe

Parameters

- **df** (*pyspark.sql.DataFrame*) – dataframe to use
- **order_col** (*str*) – column to order by

Returns resulting dataframe

Return type pyspark.sql.DataFrame

`utils.df_agg_sum`(*df*, *aggCol*, **sumCols*)
Dataframe - aggregate by column and sum

groups all the rows that have the same value for aggCol, and sums their sumCols

Parameters

- **df** (*pyspark.sql.DataFrame*) – dataframe to use
- **aggCol** (*str*) – column to aggrate on
- ***sumCols** – columns to sum

Returns resulting dataframe

Return type pyspark.sql.DataFrame

`utils.df_collectLimit`(*df*, *limit*, **cols*, *sortCol=None*)
Collect from a dataframe up to a limit

Parameters

- **df** (*pyspark.sql.DataFrame*) – dataframe to collect
- **limit** (*int*) – maximum number of rows to collect
- ***cols** – columns to collect
- **sortCol** (*str*) – column to sort by (so you always get the same rows if needed)

Returns collected columns of df if col specified or all the columns

Return type list

`utils.df_compare(`*df*, *df_other*, *idCol*`)`

Comapre two dataframes with the same schema and ids by taking the absolute difference of each row

Parameters

- **df** (*pyspark.sql.DataFrame*) – first dataframe to use
- **df_other** (*pyspark.sql.DataFrame*) – second dataframe to use
- **idCol** (*str*) – column containing the matching ids

Returns resulting dataframe

Return type pyspark.sql.DataFrame

`utils.df_elementwise(`*df*, *df_other*, *idCol*, *op*, **cols*, *renameOutput=False*`)`

Join two dataframes with the same schema by id, and perform an elementwise operation on the selected columns

Parameters

- **df** (*pyspark.sql.DataFrame*) – first dataframe to use
- **df_other** (*pyspark.sql.DataFrame*) – second dataframe to use
- **idCol** (*str*) – column containing the matching ids
- **op** (*str {"+", "-", "*", "/"}*) – operation to perform
- ***cols** – columns to perform the operation on
- **renameOutput** (*bool, optional*) – if True, add the opeartion performed to the resulting columns' names, defaults to False

Returns resulting dataframe

Return type pyspark.sql.DataFrame

`utils.df_join_rename(`*df*, *df_other*, *idCol*`)`

Join two dataframes with the same columns and rename the second one's

Parameters

- **df** (*pyspark.sql.DataFrame*) – first dataframe to use
- **df_other** (*pyspark.sql.DataFrame*) – second dataframe to use
- **idCol** (*str*) – column containing the matching ids

`utils.df_x_cartesian(`*df*, *ffilter=None*`)`

Get the cartesian product of a dataframe with itself

Parameters

- **df** (*pyspark.sql.DataFrame*) – dataframe to use
- **ffilter** (*str*) – SQL string to filter the final product by

Returns resulting dataframe

Return type pyspark.sql.DataFrame

`utils.load_df`(*floc*, *schema=None*, *header='true'*, *limit=None*, *part=None*, ***kwargs*)

Wrapper for pyspark.sql.SparkSession.load - read dataframe from parquet or csv

Parameters

- **floc** (*str*) – path to the file(s) - load accepts wildcards
- **schema** (*pyspark.sql.types.StructType, optional*) – schema to use for the datframe, defaults to None
- **header** (*str {"true"/"false"}, optional*) – does the input data have a header, defaults to "true"
- **limit** (*int, optional*) – if set reads only the first {limit} rows, defaults to None
- **part** (*str - repartition with the default number of partitions by this column name *int - repartition into this number number of partitions, optional) – if set repartitions the datafarame by this parameter (pyspark.sql.DataFrame.repartition), defaults to None
- ****kwargs** – additional arguments to pass to load

Returns the resulting dataframe

Return type pyspark.sql.DataFrame

Raises ValueError if the file (floc) extension is not 'csv' or 'parquet'

`utils.mse`(*df*, *df_target*, *idCol*, *rmse=False*)

Get the mean squared error or root mean squared error between two dataframes with the same schema

Parameters

- **df** (*pyspark.sql.DataFrame*) – first dataframe to use
- **df_target** (*pyspark.sql.DataFrame*) – second dataframe to use
- **idCol** (*str*) – column containing the matching ids
- **rmse** (*bool, optional*) – get the root mean squared error instead, defaults to False

Returns [r]mse

Return type float

`utils.plot_cluster_scatter`(*df_clust*, *axes=['x', 'y']*, *title='Plot'*, *stack=False*, *fout=None*, *eqAspect=True*)

Plot a scatter of the data provided, 2d or 3d based on the axes paramater

Parameters

- **df_clust** (*pyspark.sql.DataFrame*) – dataframe containing the cluster
- **axes** (*list, optional*) – axes to plot
- **title** (*str, optional*) – title of the plot, defaults to "Plot"
- **stack** (*bool, optional*) – don't show/save the plot, for drawing continuously on the same plot
- **fout** (*str, optional*) – save the plot to a file if provided, defaults to None
- **eqAspect** (*bool, optional*) – should it use set_aspect('equal')

utils.**plot_des**(*dir*, *target='plummer'*, *fformat='parquet'*, *times=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]*, *ltitle='# particles'*)

Takes a directory containing a number of completed applications, each of which contains diagnostic outputs, and plots their total enegry errors together against time

it's assumed that the children directories are contain a number after the target string parameter, that will be used for labels

Parameters

- **dir** (*str*) – target directory
- **target** (*str, optional}*) – string preceding the number in child directory names, defaults to "plummer"
- **fformat** (*str, optional*) – format of the diagnostic dataframes, defaults to "parquet"
- **times** (*list, optional*) – times to use for the x axis, defaults to [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

utils.**plot_histogram**(*df*, *col*, *title='Plot'*, *fout=None*)

Plot a single dataframe column as a histogram

Parameters

- **df_clust** – dataframe to use
- **col** (*str*) – column to plot the histogram for
- **title** (*str, optional*) – title of the plot, defaults to "Plot"
- **fout** (*str, optional*) – save the plot to a file if provided, defaults to None

utils.**save_df**(*df*, *fname*, *pth*, *fformat='parquet'*, *compression='gzip'*, ***kwargs*)

Wrapper for pyspark.sql.DataFrame.save - save a dataframe

Parameters

- **df** (*pyspark.sql.DataFrame*) – dataframe to save
- **fname** (*str*) – filename
- **pth** (*str*) – path to the folder the file(s) is in
- ****kwargs** – additional arguments to pass to save
- **fformat** (*str, optional*) – format to save in, defaults to "parquet"

- **compression** (*str, optional*) – compression to use, defaults to "gzip"

Returns path to the saved dataframe

Return type str

utils.**simple_error**(*df*, *df_target*, *idCol*)

Get the absolute differences between all elements of two dataframes with the same schema, and sum them

note: since inner join is used elements not peresent in one of the dataframes are ignored

Parameters

- **df** (*pyspark.sql.DataFrame*) – first dataframe to use
- **df_target** (*pyspark.sql.DataFrame*) – second dataframe to use
- **idCol** (*str*) – column containing the matching ids

Returns total difference

Return type float

A.6 schemas module

schemas.**E_test_res**

Output schema for test.py

schemas.**F_cartesian**

Schema including two ids and force, used in integrator_base

schemas.**F_id**

Schema including id and force, used in integrator_base

schemas.**clust**

Default input schema - cluster data

schemas.**clust_t**

cluster data including, poistion, velocity, mass and time

schemas.**diag**

Diagnostic output schema

schemas.**r_id**

Schema including the id and position, used in integrators

schemas.**v_id**

Schema including the id and velocity, used in integrators