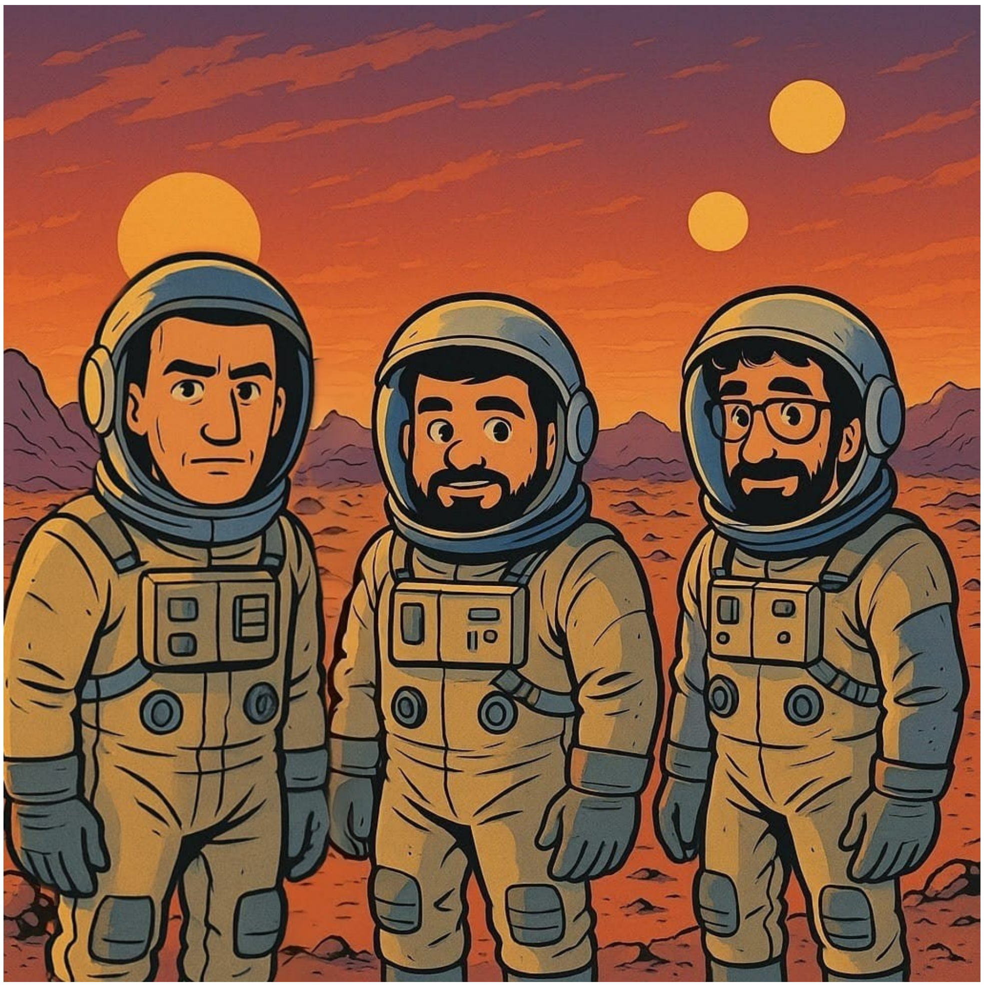
ECS529U Algorithms and Data Structures **Mini-Project**

Deadline: Wednesday 3 December (week 11) at 10am



The n-Body Problem

*Read the project description carefully, including **Essential Guidelines** on page 13.*

Mark allocation

There are three components for this project: the report, the code and the in-person test. The code and test have corresponding marks **C** and **T** out of 100.

All three components need to be completed in order for a mark to be allocated. In such a case, your final (individual) mark will be the average of **C** and **T**.

Your code mark **C** will be calculated automatically using a set of tests. We will also look at the code to ensure that essential guidelines are respected (see page 13), and deduct marks if they are not. The tests are going to be revealed only after all projects have been marked, although we will reveal a part of them before submission to assist you.

E. Kelmendi, H. Bandukara, N. Tzevelekos Autumn 2025

Summary

The n-body problem has been studied by philosophers, mathematicians and astronomers since antiquity. The problem, in modern terminology, is to calculate the trajectories of celestial bodies (e.g. planets, stars, etc.) in the presence of gravitational forces exerted to one another.

In n-body systems such as our solar system where celestial bodies of lesser masses orbit around a dominant body, this calculation can be done by analytical means using Kepler's laws of planetary motion. In its general form, though, the n-body problem cannot by solved analytically. Instead, the problem can be approximated by computational means, using simulations. At each given time step, given the positions and velocities of all bodies, we can calculate the gravitational forces exerted to each body and, using Newton's law of motion, derive its new position and velocity.

The project studies different aspects of this simulation approach. First, we look at side-problems that we can solve with the simulation algorithm, namely identifying the minimum distance between bodies during a simulation and solving the so called 3-body problem.[1] We then devise a data structure that allows us to perform the simulation step more efficiently.

In our initial approach, to find the gravitational forces on one of the n bodies, we calculate the force that each of the other n-1 bodies exerts on it. Thus, the simulation step has time complexity $\Theta(n^2)$. This is wasteful as some of those n-1 bodies are going to be far from the one we are examining. Given a group of such faraway bodies, their individual positions and masses are not crucial; instead, the whole group can be considered as a single body representing the centre of mass of the group. This approximation allows us to speed up the simulation step to $\Theta(n \log n)$ on average and is standard in modern implementations for the n-body problem.

This speed up is facilitated by a tree data structure which divides the space we are observing into smaller and smaller subspaces until each of the n bodies is included in a leaf of the tree. We call such tree a Gadget. In 2 dimensions, which is the setting for this project, a Gadget is a quad-tree, i.e. its internal (non-leaf) nodes have four children. Each node of the tree represents a rectangle in the observation space, which we call a Box:

- The root of the Gadget represents the whole observation space.
- If an internal node represents a Box b, its children represent each of the 4 equal boxes we get by dividing b in half horizontally and vertically. We call the children NE, NW, SW, SE according to their position in the North-South/East-West axes. Moreover, the Box b must contain at least two bodies – otherwise, the node should be a leaf.
- A leaf node can either be empty or contain exactly one body. For example, it is not possible to have a leaf with two bodies.

A visual representation of a Gadget built from 20 bodies is shown in Figure 1.[2]

We next look at the units we are going to use in this project and then proceed to the project questions. Essential guidelines are included on the last page (page 14).

1 This is taken from fiction and, technically, is a 4-body problem, with 3 stars and 1 planet. See https://en.wikipedia.org/wiki/The_Three-Body_Problem_(novel).

2 Adapted from Warren, M.S. and Salmon, J.K., 1994. A Fast Tree Code for Many-Body Problems. *Los Alamos Science*, (22), pp.88-97. https://sgp.fas.org/othergov/doe/lanl/pubs/00326635.pdf

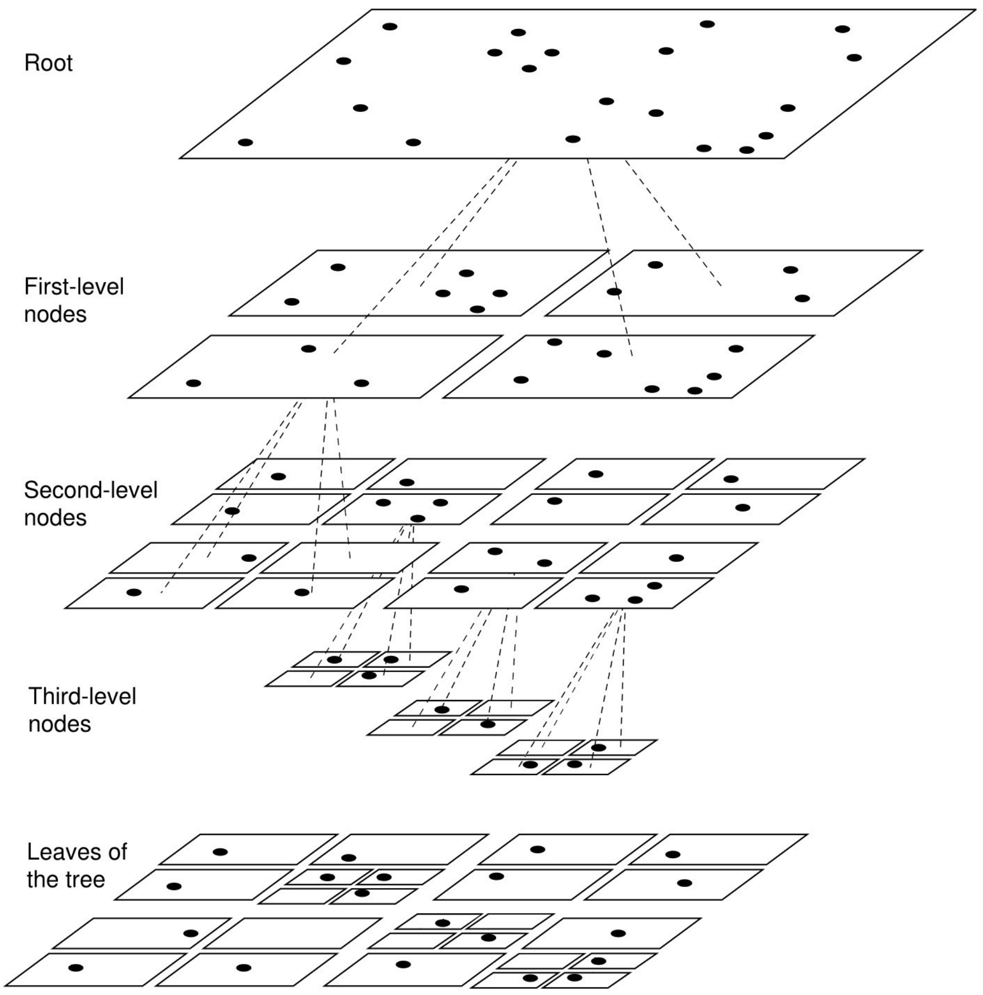


Figure 1. A gadget tree containing 20 bodies. The root node covers the whole observation space and is connected to four children nodes, each covering a quartile of the whole space. The edges of the tree connect each internal node to four children nodes obtained by cutting through its centre. The leaves are the nodes that contain at most one body and therefore do not need to be divided further. At the bottom is a “flat” representation of the gadget structure induced by the bodies.

Units

We work in **2D space**. As we study the movement of celestial bodies, we use these units:

- Time is measured in Astronomical Units (**AU**). 1 AU is equal to the distance between the Sun and the Earth.
- Mass is measured in Solar Masses (**M_{sun}**). 1 M_{sun} equals the mass of the Sun.
- Time in Years (**yr**). 1 yr is the time of one rotation of the Earth around the Sun.
- Luminosity is measured in Solar Luminosities (**L_{sun}**). 1 L_{sun} equals the luminosity of the Sun (approximately $3.8 \cdot 10^{26}$ watts).
- Velocity is measured in **AU/yr**.

Question 1 – Introduction to the n-Body Problem [30 marks]

You are given the implementation of two data structures:

1. `Body` is a data structure for representing celestial bodies. Each `Body` object `p` has:
   - coordinates `p.x` and `p.y` (so, its position is `(p.x,p.y)`)
   - velocity components `p.vx` and `p.vy` (so, its velocity is the vector `(p.vx,p.vy)`)
   - mass `p.m`.

   A body with 0 velocity is called *stationary*. Calling the `Body` constructor with three arguments construct a stationary body.

   Calling the function `p.next` with inputs an array `Bodies` of bodies and a time step `dt` calculates the position and velocity of `p` after time `dt` given its current position and the gravitational forces that each element in `Bodies` exerts on `p`. It then returns a copy of `p` with the new position and velocity.

   Finally, there is a function `p.squareDist` to calculate the square of the distance between `p` and another `Body` (we prefer calculating the square of the distance rather than the distance to avoid calculating square roots if not needed).
2. `Simulation` is for performing simulations. For each simulation object `s`:
   - `s.bodies` is an array of bodies the simulation is going to run on. Put otherwise, `s.bodies` is the starting configuration of the bodies of the simulation.
   - `s.total_time` is the total time the simulation will run for (default = 10 yr), and `s.dt` is the time step used in the simulation (default = 0.01 yr).

   The main function in a simulation `s` is `s.run`. This runs the simulation with the given time step and total time, using the `next` function of `Body`, and returns an array `ps` of configurations, where `ps[0]` is the starting configuration, `ps[1]` the configuration after one time step, and so on. The length of `ps` is `(s.total_time/s.dt)+1`, as the simulation takes `s.total_time/s.dt` time steps.

   Also, the function `s.show` runs the simulation and shows it as an animation.

You are asked to implement the following unimplemented functions.

1. `closestDistance(self)` in `Simulation`.

   The function should run the simulation (i.e. call `self.run` without arguments) and find the minimum distance between any two (distinct) bodies at any step in the simulation and return it. If the simulation contains less than two bodies, the function should return `None`.
   Hint: your function could store internally the min squared distance between two bodies and only at the end use a square root call to return the min distance.

   Examples

   Running a simulation with the following initially stationary bodies we can see that the minimum distance in the simulation is 0.015527720571708991 AU.

```
P = [None]*81
for y in range(9):
    for x in range(9):
        P[9*y+x] = Body(1,2*x,2*y)

sim = Simulation(P)
print(sim.closestDistance())
```

This calculation seems overly generous (~2,322,913 km): if we run `sim.show` then we can see in the animation that several bodies seem to collide (technically, there are no collisions in our simulation, only flyby's). But note here that our function does not calculate the closest distance in the full continuous trajectories of these bodies, but only the closest distance we can see in the calculated simulation steps!

In fact, if we run the simulation with the same initial configuration but a finer time step of 0.0001 yr then we get a closest distance of 0.00019088314702779433 AU.

```
sim = Simulation(P, total_time=0.5, dt=0.0001)
print(sim.closestDistance())
```

2. `threeBodyProblem(self, sunA, sunB, sunC, lA, lB, lC)` in `Body`.

   Inputs `sunA,sunB,sunC` are `Body` objects which we assume to be stars while `self` is a planet orbitting around them. Inputs `lA,lB,lC` are the luminosities of each of the three suns respectively. The function should run a simulation with initial configuration `[sunA,sunB,sunC,self]` and calculate the time (in yr) that `self` is going to be continuously habitable counting from the initial configuration. If the planet is habitable for the full simulation period then your function should return 10.01. We use two criteria for habitability of a planet:

   a) The planet should not overheat by getting too much heat from its suns (*no scorching*). For this criterion, the total luminosity reaching `self` should not be greater than the given threshold `scoeff`. If sun `s` has luminosity `l`, then the luminosity reaching `self` from `s` is given by:

      `luminosity(s,l) = l / (4*math.pi*self.squareDist(s))`

      The total luminosity reaching `self` is the sum of the luminosities reaching it from `sunA`, `sunB` and `sunC`.

   b) The planet should not freeze by getting too little heat from its suns (*no freezing*), i.e. the total luminosity reaching `self` should not be less than the given `fcoeff`.

   <u>Examples</u>

   Running a simulation with the following bodies we can see that planet `p` gets overly close to `sA` at time step 42 and gets scorched. The code should return 0.42.

```
sA = Body(10,-10,0,3.14,-5.44,)
sB = Body(10,0,0,3.24,5.44)
sC = Body(10,-5,8.660,-6.28,0)
p  = Body(3.0e-6,-10,6,-3,-15)

Bodies=[sA,sB,sC,p]
p.threeBodyProblem(sA,sB,sC,50,50,50)
```

   If we decrease the luminosities of the suns by 10% then the planet becomes frozen at the initial configuration. The code below should therefore return 0.0.

```
p.threeBodyProblem(sA,sB,sC,45,45,45)
```

Question 2 – Gadget [50 marks]

This question asks you to build a tree data structure to support faster simulations (which are studied in the next question). We call this data structure `Gadget`. A gadget object uses nodes from the class `GNode`, with each such node covering a rectangle in 2D space.

To represent rectangles we use the class `Box`. Each `Box` object `b` has:

- bottom-left and top-right corners `(b.x0,b.y0)` and `(b.x1,b.y1)` respectively
- centre `(b.mx,b.my)` (e.g. `b.mx = (b.x1-b.x0)/2`)

and largest side `b.maxSide`.There is also the function `b.isIn` that checks whether body `p` is within the space represented by `b`. On the other hand, `b.split4` returns 4 new `Box` objects representing the 4 rectangles obtained by splitting `b` at its centre. Finally, the class method `getBox` builds and returns the smallest box that encloses all elements of an input array `P` of bodies.

The constructor of class `GNode` is as follows:

```
def __init__(self, box):
   self.box = box
   self.COM = None            # this node’s COM
   self.nbodies = 0           # number of bodies contained in node
   self.p = None              # if this node is a leaf with a Body
   self.children = None       # children are: [NE, NW, SW, SE]
   self.updateCOM()
```

Each `GNode` object `node` can be either a leaf or an internal tree node. A leaf node can be empty or contain a single body; an internal node must have four children nodes and contain at least two bodies. If `node` is a leaf that contains a body `p`, then the latter is stored in `node.p`. If, on the other hand, `node` is an internal node then its four children are stored in the array `node.children`.

The space covered by a `GNode` object `node` is `node.box`. In this space there are a number of bodies (`node.nbodies`-many), each with its own mass and position. We can approximate these objects by calculating their *centre of mass* (COM). The COM of `node` is stored at `node.COM` and is a stationary body (i.e. a `Body` object with 0 velocity). For example:

- if `node` contains a single body `p` then its COM has: mass `p.m`, position `(p.x,p.y)`;
- if `node` contains four bodies of equal mass `m` positioned at the corners of `node.box` then its COM has mass `4*m` and position `(node.box.mx,node.box.my)`.

Calling the function `node.updateCom` sets the COM of `node` as follows:

- if `node` is a leaf, then either its COM is at its centre and has 0 mass (if `node` is empty) or it is calculated as we saw above (if `node` contains a single body `p`),
- if `node` is an internal node then its COM is calculated by taking the centre of mass of the COM’s of the four children of `node`.

`Gadget`’s are trees built of `GNode`’s. Each `Gadget` object `g` contains a root node `g.root` and a size `g.size`. The size records the number of bodies stored in the gadget (this is not the same as the number of nodes). Calling `g.getBodies` returns an array of all the bodies in `g`, while `g.plot` draws the gadget in 2D. You are asked to implement the following main functions in this class, for adding and removing bodies in a gadget.

1. `add(self, p)` adds body `p` in `self`.

   The function should traverse the nodes in `self` to find the correct node for `p`, i.e. the unique leaf node that encloses `p` in its box. The traversal is done similarly to BST traversal. Suppose we are looking at `node` which encloses `p` in its box:

   - If `node` is an internal node then we locate the child to insert `p` in:
     - if `node.child[0].isIn(p)` then we add `p` in the NE child
     - else if `node.child[1].isIn(p)` then we add `p` in the NW child
     - else if `node.child[2].isIn(p)` then we add `p` in the SW child
     - else we add `p` in the SE child.

     Note the above rules determine how bodies positioned on middle-lines of `node` are added. E.g. if `p` is at `(node.box.mx,node.box.my)` then it goes to NE.
   - If `node` is a leaf then:
     - if `node` is not empty, i.e. there is a body `p0` stored in `node.p`, then we make `node` an internal `node` and add both `p0` and `p` in it (as explained above),
     - if `node` is empty (i.e. `node.p` is `None`) then we add `p` in `node` (set `node.p=p`).

   Note that `add` should also update the size of the gadget and correctly update the fields of all the nodes it affects.

2. `remove(self, p)` removes body `p` from `self`.

   The function should traverse the nodes in `self` as described above. If `p` is not in `self`, then `remove` should return without doing anything. Otherwise, it should remove `p` from the leaf node where it is stored.

   Removing `p` from a leaf node may lead to its parent node becoming *a false internal*, i.e. an internal node containing only one body. Your function should clean up the tree by recursively changing all false internal nodes to leaves once `p` is removed.

   Note that `remove` should also update the size of the gadget and correctly update the fields of all the nodes it affects.

<u>Examples</u>

In the following code we use the `Gadget.plot` function to view the state of our gadget. We first create a gadget `g` with four stationary bodies positioned at the corners of a 100x100 square centred at the origin (0,0).

```
P = [None]*4
P[0] = Body(1,50,50)
P[1] = Body(1,-50,50)
P[2] = Body(1,-50,-50)
P[3] = Body(1,50,-50)
g = Gadget(Box.getBox(P))
for p in P: g.add(p)
print(g)
g.plot()
```

```
0:Box(-50,-50,50,50),4
├─ 1:Box(0.0,0.0,50,50),1
├─ 2:Box(-50,0.0,0.0,50),1
├─ 3:Box(-50,-50,0.0,0.0),1
└─ 4:Box(0.0,-50,50,0.0),1
0  -> (4, 0.0, 0.0)
1  -> (1, 50, 50)
2  -> (1, -50, 50)
3  -> (1, -50, -50)
4  -> (1, 50, -50)
```

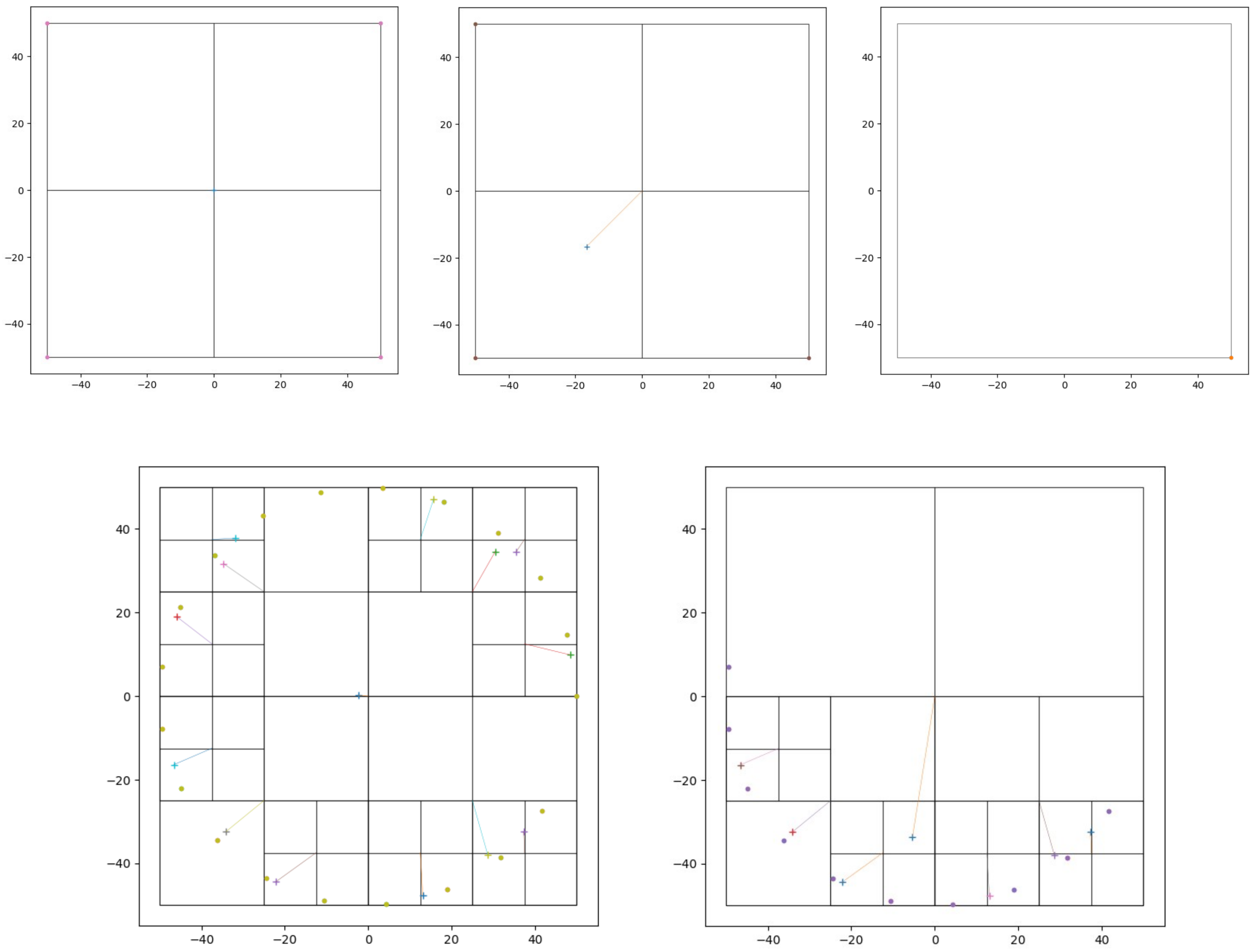


Figure 2. Example Gadget plots. Nodes are shown as rectangles, objects shown as dots. At internal nodes, the COM is shown with a cross, and a line connecting it to the centre of the node.

The printout of g is shown on the right above. Observe that we print gadgets as trees and, for each of their nodes, we print an integer identifier, followed by the node's box and nbodies. Following the tree printout, we also show the COM of each node. In this case, the root of the tree is node 0, and its children nodes 1-4. In the plot of g (Figure 2, top left), the root node covers the outside rectangle, and its four children cover the inside ones. The plot function shows the bodies included leaf nodes as pink dots. In internal nodes, it shows that COM with a cross. Here, the COM of the root is at the origin and not clearly visible.

Next, we remove the body at the top-right corner (50,50), and print and plot.

```
g.remove(P[0])
print(g)
g.plot()
```

```
0:Box(-50,-50,50,50),3
 ├─ 1:Box(0.0,0.0,50,50),0
 ├─ 2:Box(-50,0.0,0.0,50),1
 ├─ 3:Box(-50,-50,0.0,0.0),1
 └─ 4:Box(0.0,-50,50,0.0),1

0  -> (3, -16.666666666666668, -16.666666666666668)
1  -> (0, 25.0, 25.0)
2  -> (1, -50, 50)
3  -> (1, -50, -50)
4  -> (1, 50, -50)
```

In the printout and the plot (Figure 2, top middle) we can see that the top-right body is gone and the top-right node is an empty leaf with COM (0,25,25). A consequence of this removal is that the COM of the root node has moved towards the bottom-left corner.

We now remove all bodies but the bottom-right one, and print and plot (Figure 2, top right).

```
g.remove(P[1]); g.remove(P[2])
print(g)
g.plot()
```

```
0:Box(-50,-50,50,50),1
0  -> (1, 50, -50)
```

For our next example, we fill in a gadget with 20 moving bodies of varying mass spread on a circle of radius 50 AU around the origin. We subsequently remove the first 10 of them.

```
P = [None]*20
for i in range(20):
    x  = 50.0 * math.cos(i * 0.3)
    y  = 50.0 * math.sin(i * 0.3)
    vx =  5.0 * math.cos(i * 0.7)
    vy =  5.0 * math.sin(i * 0.7)
    m  = 1.0 + (i % 5)            # masses: 1..5
    P[i] = Body(m, x, y, vx, vy)

g = Gadget(Box.getBox(P))
for p in P: g.add(p)
print(g); g.plot()

for i in range(10): g.remove(P[i])
print(g); g.plot()
```

We obtain the printouts below (next page) and the plots shown in Figure 2 (bottom).

```
0:Box(-50,-50,50,50),20
├─ 1:Box(0.0,0.0,50,50),6
│  ├─ 2:Box(25.0,25.0,50,50),2
│  │  ├─ 3:Box(37.5,37.5,50,50),0
│  │  ├─ 4:Box(25.0,37.5,37.5,50),1
│  │  ├─ 5:Box(25.0,25.0,37.5,37.5),0
│  │  └─ 6:Box(37.5,25.0,50,37.5),1
│  ├─ 7:Box(0.0,25.0,25.0,50),2
│  │  ├─ 8:Box(12.5,37.5,25.0,50),1
│  │  ├─ 9:Box(0.0,37.5,12.5,50),1
│  │  ├─ 10:Box(0.0,25.0,12.5,37.5),0
│  │  └─ 11:Box(12.5,25.0,25.0,37.5),0
│  ├─ 12:Box(0.0,0.0,25.0,25.0),0
│  └─ 13:Box(25.0,0.0,50,25.0),2
│     ├─ 14:Box(37.5,12.5,50,25.0),1
│     ├─ 15:Box(25.0,12.5,37.5,25.0),0
│     ├─ 16:Box(25.0,0.0,37.5,12.5),0
│     └─ 17:Box(37.5,0.0,50,12.5),1
├─ 18:Box(-50,0.0,0.0,50),5
│  ├─ 19:Box(-25.0,25.0,0.0,50),1
│  ├─ 20:Box(-50,25.0,-25.0,50),2
│  │  ├─ 21:Box(-37.5,37.5,-25.0,50),1
│  │  ├─ 22:Box(-50,37.5,-37.5,50),0
│  │  ├─ 23:Box(-50,25.0,-37.5,37.5),0
│  │  └─ 24:Box(-37.5,25.0,-25.0,37.5),1
│  ├─ 25:Box(-50,0.0,-25.0,25.0),2
│  │  ├─ 26:Box(-37.5,12.5,-25.0,25.0),0
│  │  ├─ 27:Box(-50,12.5,-37.5,25.0),1
│  │  ├─ 28:Box(-50,0.0,-37.5,12.5),1
│  │  └─ 29:Box(-37.5,0.0,-25.0,12.5),0
│  └─ 30:Box(-25.0,0.0,0.0,25.0),0
├─ 31:Box(-50,-50,0.0,0.0),5
│  ├─ 32:Box(-25.0,-25.0,0.0,0.0),0
│  ├─ 33:Box(-50,-25.0,-25.0,0.0),2
│  │  ├─ 34:Box(-37.5,-12.5,-25.0,0.0),0
│  │  ├─ 35:Box(-50,-12.5,-37.5,0.0),1
│  │  ├─ 36:Box(-50,-25.0,-37.5,-12.5),1
│  │  └─ 37:Box(-37.5,-25.0,-25.0,-12.5),0
│  ├─ 38:Box(-50,-50,-25.0,-25.0),1
│  └─ 39:Box(-25.0,-50,0.0,-25.0),2
│     ├─ 40:Box(-12.5,-37.5,0.0,-25.0),0
│     ├─ 41:Box(-25.0,-37.5,-12.5,-25.0),0
│     ├─ 42:Box(-25.0,-50,-12.5,-37.5),1
│     └─ 43:Box(-12.5,-50,0.0,-37.5),1
└─ 44:Box(0.0,-50,50,0.0),4
   ├─ 45:Box(25.0,-25.0,50,0.0),0
   ├─ 46:Box(0.0,-25.0,25.0,0.0),0
   ├─ 47:Box(0.0,-50,25.0,-25.0),2
   │  ├─ 48:Box(12.5,-37.5,25.0,-25.0),0
   │  ├─ 49:Box(0.0,-37.5,12.5,-25.0),0
   │  ├─ 50:Box(0.0,-50,12.5,-37.5),1
   │  └─ 51:Box(12.5,-50,25.0,-37.5),1
   └─ 52:Box(25.0,-50,50,-25.0),2
      ├─ 53:Box(37.5,-37.5,50,-25.0),1
      ├─ 54:Box(25.0,-37.5,37.5,-25.0),0
      ├─ 55:Box(25.0,-50,37.5,-37.5),1
      └─ 56:Box(37.5,-50,50,-37.5),0
0  -> (60.0, -2.4025184922444547, 0.20781343257294413)
1  -> (16.0, 30.486392719106174, 34.61239340143095)
2  -> (7.0, 35.44604798436923, 34.48025041925028)
3  -> (0, 43.75, 43.75)
4  -> (4.0, 31.080498413533224, 39.166345481374165)
5  -> (0, 31.25, 31.25)
6  -> (3.0, 41.26678074548391, 28.23212366975177)
7  -> (6.0, 15.687716450425595, 47.147420137001546)
8  -> (5.0, 18.117887723833682, 46.60195429836131)
9  -> (1.0, 3.5368600833851453, 49.874749330202725)
10 -> (0, 6.25, 31.25)
11 -> (0, 18.75, 31.25)
12 -> (0, 12.5, 12.5)
13 -> (3.0, 48.51121630418686, 9.850673555377986)
14 -> (2.0, 47.7668244562803, 14.776010333066978)
15 -> (0, 31.25, 18.75)
16 -> (0, 31.25, 6.25)
17 -> (1.0, 50.0, 0.0)
18 -> (15.0, -34.76290190678826, 31.72398497712211)
19 -> (2.0, -11.360104734654344, 48.69238154390976)
20 -> (7.0, -31.886522685461102, 37.79629158679446)
21 -> (3.0, -25.24230522999288, 43.160468332443685)
22 -> (0, -43.75, 43.75)
23 -> (0, -43.75, 31.25)
24 -> (4.0, -36.86968577706227, 33.77315902755755)
25 -> (6.0, -45.91961005571458, 18.983495076908486)
26 -> (0, -31.25, 18.75)
27 -> (5.0, -45.20360710085305, 21.36899401169151)
28 -> (1.0, -49.49962483002227, 7.0560004029933605)
29 -> (0, -31.25, 6.25)
30 -> (0, -12.5, 12.5)
31 -> (15.0, -34.103546372516696, -32.43175444879996)
32 -> (0, -12.5, -12.5)
33 -> (5.0, -46.65234788820171, -16.430527181710524)
34 -> (0, -31.25, -6.25)
35 -> (2.0, -49.37398849544324, -7.887284707162411)
36 -> (3.0, -44.83792081670736, -22.126022164742604)
37 -> (0, -31.25, -18.75)
38 -> (4.0, -36.29661521000701, -34.38830795919869)
39 -> (6.0, -22.18416588445231, -44.461741497775314)
40 -> (0, -6.25, -31.25)
41 -> (0, -18.75, -31.25)
42 -> (5.0, -24.513041067034973, -43.57878862067941)
43 -> (1.0, -10.539789971538985, -48.87650588325485)
44 -> (14.0, 28.647380796372026, -37.90806759952506)
45 -> (0, 37.5, -12.5)
46 -> (0, 12.5, -12.5)
47 -> (5.0, 13.089311950178336, -47.697732646548786)
48 -> (0, 18.75, -31.25)
49 -> (0, 6.25, -31.25)
50 -> (2.0, 4.37494917197232, -49.808230441792034)
51 -> (3.0, 18.898887135649012, -46.29073411638662)
52 -> (9.0, 37.29075237759074, -32.46936479562299)
53 -> (5.0, 41.73563924195799, -27.53427712988188)
54 -> (0, 31.25, -31.25)
55 -> (4.0, 31.734643797131696, -38.63822437779938)
56 -> (0, 43.75, -43.75)
```

```
0:Box(-50,-50,50,50),10
├─ 1:Box(0.0,0.0,50,50),0
├─ 2:Box(-50,0.0,0.0,50),1
├─ 3:Box(-50,-50,0.0,0.0),5
│  ├─ 4:Box(-25.0,-25.0,0.0,0.0),0
│  ├─ 5:Box(-50,-25.0,-25.0,0.0),2
│  │  ├─ 6:Box(-37.5,-12.5,-25.0,0.0),0
│  │  ├─ 7:Box(-50,-12.5,-37.5,0.0),1
│  │  ├─ 8:Box(-50,-25.0,-37.5,-12.5),1
│  │  └─ 9:Box(-37.5,-25.0,-25.0,-12.5),0
│  ├─ 10:Box(-50,-50,-25.0,-25.0),1
│  └─ 11:Box(-25.0,-50,0.0,-25.0),2
│     ├─ 12:Box(-12.5,-37.5,0.0,-25.0),0
│     ├─ 13:Box(-25.0,-37.5,-12.5,-25.0),0
│     ├─ 14:Box(-25.0,-50,-12.5,-37.5),1
│     └─ 15:Box(-12.5,-50,0.0,-37.5),1
└─ 16:Box(0.0,-50,50,0.0),4
   ├─ 17:Box(25.0,-25.0,50,0.0),0
   ├─ 18:Box(0.0,-25.0,25.0,0.0),0
   ├─ 19:Box(0.0,-50,25.0,-25.0),2
   │  ├─ 20:Box(12.5,-37.5,25.0,-25.0),0
   │  ├─ 21:Box(0.0,-37.5,12.5,-25.0),0
   │  ├─ 22:Box(0.0,-50,12.5,-37.5),1
   │  └─ 23:Box(12.5,-50,25.0,-37.5),1
   └─ 24:Box(25.0,-50,50,-25.0),2
      ├─ 25:Box(37.5,-37.5,50,-25.0),1
      ├─ 26:Box(25.0,-37.5,37.5,-25.0),0
      ├─ 27:Box(25.0,-50,37.5,-37.5),1
      └─ 28:Box(37.5,-50,50,-37.5),0
0  -> (30.0, -5.332982975618814, -33.67110875741189)
1  -> (0, 25.0, 25.0)
2  -> (1.0, -49.49962483002227, 7.0560004029933605)
3  -> (15.0, -34.103546372516696, -32.43175444879996)
4  -> (0, -12.5, -12.5)
5  -> (5.0, -46.65234788820171, -16.430527181710524)
6  -> (0, -31.25, -6.25)
7  -> (2.0, -49.37398849544324, -7.887284707162411)
8  -> (3.0, -44.83792081670736, -22.126022164742604)
9  -> (0, -31.25, -18.75)
10 -> (4.0, -36.29661521000701, -34.38830795919869)
11 -> (6.0, -22.18416588445231, -44.461741497775314)
12 -> (0, -6.25, -31.25)
13 -> (0, -18.75, -31.25)
14 -> (5.0, -24.513041067034973, -43.57878862067941)
15 -> (1.0, -10.539789971538985, -48.87650588325485)
16 -> (14.0, 28.647380796372026, -37.90806759952506)
17 -> (0, 37.5, -12.5)
18 -> (0, 12.5, -12.5)
19 -> (5.0, 13.089311950178336, -47.697732646548786)
20 -> (0, 18.75, -31.25)
21 -> (0, 6.25, -31.25)
22 -> (2.0, 4.37494917197232, -49.808230441792034)
23 -> (3.0, 18.898887135649012, -46.29073411638662)
24 -> (9.0, 37.29075237759074, -32.46936479562299)
25 -> (5.0, 41.73563924195799, -27.53427712988188)
26 -> (0, 31.25, -31.25)
27 -> (4.0, 31.734643797131696, -38.63822437779938)
28 -> (0, 43.75, -43.75)
```

Question 3 – FastSimulation [20 marks]

In this question you need to use the `Gadget` class to implement a faster simulation algorithm. You need to implement this in a new class called `FastSimulation` that is a subclass of `Simulation`. In `FastSimulation` we override the `run` function as follows:

```
def run(self, test=None):
    pss = [None]*(self.timesteps+1)
    pss[0] = self.bodies
    for t in range(self.timesteps):
        g = Gadget.fromBodies(pss[t])
        A = pss[t][:]
        for i in range(len(A)):
            new_ps = FastSimulation.getBodies(g,A[i])
            A[i] = A[i].next(new_ps,self.dt)
            if test is not None: test[i] = new_ps
        pss[t+1] = A
    return pss
```

The function starts by creating an array of arrays `pss` where it will store the simulation: the initial configuration will be stored as an array in `pss[0]`, the next one in `pss[1]`, and so on. Hence, `pss[0]` is set to `self.bodies`.

We next loop over the time steps of the simulation and, at each step `t`:

- we create a new `Gadget` `g` containing the bodies of configuration `pss[t]`;
- we then make a copy of configuration `pss[t]` and store it in array `A`;
- and for each body `A[i]` we calculate its new position and velocity by applying `A[i].next` not to `pss[t]` (i.e. the full current configuration) but, rather, to its approximation `new_ps` that we obtain by using `FastSimulation.getBodies` (this function uses the gadget `g` as explained in earlier on, see more on it below);
- the `test` array, if present, is used for testing (you can ignore it for now);
- we finally store in `pss[t+1]` the bodies with their new positions and velocities.

Your task is to implement the following (static) function in `FastSimulation`:

```
getBodies(g, p, shouldOpen=BarnesHut)
```

The function takes as inputs a gadget `g` and a reference body `p`, as well as an *opening criterion* `shouldOpen`, and returns an array of bodies `new_ps`. The function performs a depth-first traversal of `g` and at each internal `node` it uses `shouldOpen` to decide whether `node` should be *opened* or not. `shouldOpen` is a function taking `node` and `p` as inputs and returning:

- `True` if `node` needs to be opened, i.e. if the traversal should continue by examining the children of `node` to find more bodies, and
- `False` if `node` should not be opened, i.e. the traversal should not look at the children of `node`. Instead `node` should be treated as a single stationary body. In this case, we should add to `new_ps` the COM of `node`.

Otherwise, at each leaf `node` we add to `new_ps` the body stored in `node` (if there is any). Thus, at the end of the traversal the array `new_ps` will contain some real bodies and some approximations of groups of bodies due to internal nodes that remained unopened. Depending on how permissive `shouldOpen` is, `new_ps` could be much smaller than `self.bodies`. Our default opening criterion is due to Barnes and Hut,[3] and is tunable by a parameter `theta` (the lower its value, the less permissive the criterion).

Examples

We create 3n bodies with varying masses and velocities and position them on 3 circles of radii 50, 100 and 150 AU. We build a `FastSimulation` object `sim` for them and set the total simulation time to 0.01 yr, i.e. only one simulation step will take place. We run `sim.run` with a test array `new_ps` so that, for each body in `sim.bodies`, the result returned by `FastSimulation.getBodies` is stored in `new_ps`. We then print each of the three elements of `new_ps` along with its length. We run this experiment for n=1 and n=3. We can see in the printouts that e.g. even for n=1 we managed to approximate two bodies.

```
def testOnThreeCircles(n):
    P = [None]*3*n
    for i in range(n):
        x  = 50.0 * math.cos(i * 0.3); y  = 50.0 * math.sin(i * 0.3)
        vx = 50.0 * math.cos(i * 0.7); vy = 50.0 * math.sin(i * 0.7)
        m  = 1.0 + (i % 5)            # masses: 1..5
        P[3*i] = Body(m, x, y, 3*vx, 3*vy)
        P[3*i+1] = Body(m, 2*x, 2*y, 2*vx, 2*vy)
        P[3*i+2] = Body(m, 3*x, 3*y, vx, vy)
    sim = FastSimulation(P, total_time=0.01)
    new_ps = [None]*len(P)
    sim.run(test=new_ps)
    print(f"Experiment with n={n}, new_ps entries:")
    for ps in new_ps: print(len(ps),ps)

TestOnThreeCircles(1); TestOnThreeCircles(3)
```

```
Experiment with n=1, new_ps entries:
2 [Body(1.0,50.0,0.0,150.0,0.0), Body(2.0,125.0,0.0,0,0)]
3 [Body(1.0,50.0,0.0,150.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(1.0,150.0,0.0,50.0,0.0)]
3 [Body(1.0,50.0,0.0,150.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(1.0,150.0,0.0,50.0,0.0)]
Experiment with n=3, new_ps entries:
8 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(1.0,50.0,0.0,150.0,0.0),
Body(2.0,47.7668244562803,14.776010333066978,114.72632809267327,96.63265308565366),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602), Body(5.0,131.6003946894074,68.54903500523355,0,0)]
8 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(3.0,48.51121630418686,9.850673555377986,0,0),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)]
6 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(8.0,57.55016112176669,21.669054125957146,0,0),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)]
7 [Body(2.0,125.0,0.0,0,0), Body(1.0,50.0,0.0,150.0,0.0), Body(2.0,47.7668244562803,14.776010333066978,114.72632809267327,96.63265308565366),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602), Body(5.0,131.6003946894074,68.54903500523355,0,0)]
8 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(3.0,48.51121630418686,9.850673555377986,0,0),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)]
6 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(8.0,57.55016112176669,21.669054125957146,0,0),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)]
7 [Body(2.0,125.0,0.0,0,0), Body(1.0,50.0,0.0,150.0,0.0), Body(2.0,47.7668244562803,14.776010333066978,114.72632809267327,96.63265308565366),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602), Body(5.0,131.6003946894074,68.54903500523355,0,0)]
8 [Body(1.0,150.0,0.0,50.0,0.0), Body(1.0,100.0,0.0,100.0,0.0), Body(3.0,48.51121630418686,9.850673555377986,0,0),
Body(3.0,41.26678074548391,28.23212366975177,25.49507143503616,147.81745949826905),
Body(2.0,95.5336489125606,29.552020666133956,76.48421872844885,64.4217687237691),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)]
5 [Body(2.0,125.0,0.0,0,0), Body(8.0,57.55016112176669,21.669054125957146,0,0),
Body(3.0,82.53356149096783,56.46424733950354,16.996714290024105,98.54497299884602),
Body(2.0,143.3004733688409,44.328030999200934,38.242109364224426,32.21088436188455),
Body(3.0,123.80034223645174,84.6963710092553,8.498357145012053,49.27248649942301)].,
```

Essential Guidelines

What to submit

You should submit **exactly both** of the following files:

1. A **PDF file `report.pdf`** with a report, in which you should include:
   a) A section describing each group member's contribution. There should be one paragraph for each member and the whole section should cover about 1 page.
   b) Your code implementations, leaving out the code for `Box`, `GNode` and `Stack`.

   In order to be able to check submissions for plagiarism:
   - The report should be written electronically.
   - Do not embed the code in the report as an image; rather, copy-and-paste it in your document as text.
2. A **Jupyter file** called **`nBodyProblem.ipynb`** containing your code:
   - the file should contain the full classes `Body`, `Simulation`, `Gadget` and `FastSimulation`, as well as `Box`, `Gnode` and `Stack` (these three are given).

   We will mark your code automatically, so make sure that:

   ◦ your code is correctly indented and without syntax errors, and that it can by imported without errors
   ◦ you do not change any of the functions already implemented,
   ◦ you do not include any module imports apart from the ones we have included
   ◦ you do not include any code you used for testing, nor any debugging messages
   ◦ put all your code in one cell (i.e. a single "box" of code) with all classes inside it.

Code Specifications

Unless stated otherwise, **you are not allowed to use** built-in Python functions. Moreover, and unless specified otherwise, no built-in data structures can be used apart from arrays, tuples and strings. In particular, you cannot use built-in list operations for appending an element to a list. **You can use** substring/subarray-creating constructs like `A[lo:hi]`. You can use helper functions of your own.

We have provided you with an implementation of a `Stack` data structure, which you can use if you need to. You can use data structures that we saw in the module, **but only for auxiliary purposes** and not for replacing the functions required by the asked data structures. Please ask if not sure.

Avoid Plagiarism

- This is a group assignment and you should **only collaborate with students in your group**. No help from AI is allowed (e.g. do not use chatGPT).
- Showing your solutions to other groups is an examination offence.
- You can use material from the web, so long as you **clearly reference your sources** in your report. However, what will get marked is your own contribution, so if you simply copy code from the web you will get few or no marks