1 n-Body Tree

Constructing the octree data structure has several steps. A broad outline of the algorithm would be -

a. Construct a function to split space into equal octants.

b. Construct a function to identify which octant a particle resides in.

c. Define a new data structure to store tree-node information.

d. Define a top-leaf (in other words, a parent node) in the form of the newly created Tree-node data type.

e. Construct a function which recursively fills in all the tree nodes by repeatedly splitting space until there are zero or one particles left in the node.

Once the tree structure has been created, computing any quantities of interest (such as the angle subtended by a node diagonal on a particle) would also involve recursively tracing the branches of the tree. I have created two main modules (`treeGenerator.jl` and `treeAttributes.jl`) which generate the desired tree and acquires desired attributes from the tree. I will quickly summarize each of their functions and more details about low-level implementations can be gathered by following the comments in my Julia code.

1.1 Code Walk-through

Following is the code-listing of `treeGenerator.jl`. A vital step is to define a new datastructure `TreeNode`. Each TreeNode has five properties - its spatial span, positions of the particles inside that spatial span, the center of mass of the particles inside the node, total mass of the node and a tree of daughter tree nodes. The next function definition in the module is to divide a 3D space of a defined size into equal octants. There are two types of spans which constitute the octant bounds - either the bound spans between minimum to mid-point (labelled as 1) or from mid-point to maximum (labelled as 2). Using an array of these span types, we can uniquely label each of the octants. The next function, uses particle's location and these label types to identify what octant of the given node would the particle reside in. Finally, we have the function which populates the entire tree recursively. The base cases are identified when there is one or no particle and recursion into further nodes stops there. In case of large data-sets, we also define a maximum recursion depth as an upper bound of the tree-layers.

```julia
### treeGenerator.jl Module ###
###############################
"For turning n-body position data into octrees"

# creating custom julia structure to store tree type data
struct TreeNode
    bounds # stores the spatial bounds of the node
    bodies # stores the particles in the node
    COM    # stores center of mass of the node
    mass   # stores the mass of the node (in solar masses)
    nodes  # stores the child nodes
end;

# function to divide the node into 8 equal octants
function canvas_split(bound)

    # unpacking bounds of the full region
    xMin, xMax, yMin, yMax, zMin, zMax = bound

    # Calculating midpoints of the total region
    xMid = (xMax + xMin)/2.0
    yMid = (yMax + yMin)/2.0
    zMid = (zMax + zMin)/2.0

    # Defining the new bounds of the 8 octants
    new_bounds = [(xMin, xMid, yMin, yMid, zMin, zMid), # label (1, 1, 1)
                  (xMid, xMax, yMin, yMid, zMin, zMid), # label (2, 1, 1)
                  (xMin, xMid, yMid, yMax, zMin, zMid), # label (1, 2, 1)
                  (xMin, xMid, yMin, yMid, zMid, zMax), # label (1, 1, 2)
                  (xMid, xMax, yMid, yMax, zMin, zMid), # label (2, 2, 1)
                  (xMid, xMax, yMin, yMid, zMid, zMax), # label (2, 1, 2)
                  (xMin, xMid, yMid, yMax, zMid, zMax), # label (1, 2, 2)
                  (xMid, xMax, yMid, yMax, zMid, zMax)  # label (2, 2, 2)
    ]

    return new_bounds # output is a list of 8 tuples
end;


# function to locate which octant a particle belongs to
function which_node(particle_location, nodes_boundaries)

    xMid = nodes_boundaries[1][2]   # extracting node boundary information
    yMid = nodes_boundaries[1][4]   # extracting node boundary information
    zMid = nodes_boundaries[1][6]   # extracting node boundary information

    x, y, z = particle_location  # unpacking particle locations

    xPass = x < xMid    # checking if x-coordinate passes the y-z plane
    yPass = y < yMid    # checking if y-coordinate passes the x-z plane
    zPass = z < zMid    # checking if z-coordinate passes the x-y plane

    identifier = (xPass, yPass, zPass) # creating an identifier to check which octant the particle
     is in

    if identifier == (true, true, true)
        n = 1
    elseif identifier == (false, true, true)
        n = 2
    elseif identifier == (true, false, true)
        n = 3
    elseif identifier == (true, true, false)
        n = 4
    elseif identifier == (false, false, true)
        n = 5
    elseif identifier == (false, true, false)
        n = 6
    elseif identifier == (true, false, false)
        n = 7
    else
        n = 8
    end

    return n
end;

# The following function builds the tree recursively
# Input parameters: parent node, particles in the parent node, boundaries of the parent node, max
    allowed recursion depth
# Output: a tree structure with the parent node as the root

function build_tree(particles, nodes_boundaries, depth)

    # Base case: if there is only one particle in the node or the depth of the tree is 0
    if length(particles) <= 1 || depth == 0
        return TreeNode(nodes_boundaries, particles, compute_COM(particles), length(particles),
    nodes_boundaries)
    end

    # keeping track of which particle belongs to which octant
    child = [which_node(particles[i], nodes_boundaries) for i in eachindex(particles)]

    # Recursively building the tree
    new_nodes = [build_tree(particles[child .== i], canvas_split(nodes_boundaries[i]), depth-1)
    for i in 1:8]

    return TreeNode(nodes_boundaries, particles, compute_COM(particles), length(particles),
    new_nodes)
end;
```

The second module involves functions to compute center of mass of a node, total mass of the node, angle subtended by a node on a particle etc. and methods to find these quantities recursively for all non-empty leafs.

```
# angles subtended on a particle by a node
function angle_subtended(particle_location, node)

    # unpacking particle location
```

```julia
    x, y, z = particle_location

    # unpacking node boundaries
    xMin, xMax, yMin, yMax, zMin, zMax = node.bounds[1][1], node.bounds[8][2], node.bounds[1][3], node.bounds[8][4], node.bounds[1][5], node.bounds[8][6]

    d = sqrt((xMax - xMin)^2 + (yMax - yMin)^2 + (zMax - zMin)^2)        # distance between the two corners of the node
    xMid, yMid, zMid = (xMin + xMax)/2, (yMin + yMax)/2, (zMin + zMax)/2 #  center of the node

    r = sqrt((x - xMid)^2 + (y - yMid)^2 + (z - zMid)^2) # distance between the center of the node and the particle

    # calculating the angle subtended by the node on the particle
    angle = 2*atan(d/r)

    return angle*180/pi # output is in degrees
end;


##############################
######## COM LIST ############
##############################

# Center of Mass computing function for equal mass particles
function compute_COM(particle_list)

    N = length(particle_list)      # total number of particles
    vecAdd = [0,0,0]               # initiating array to compute center of mass
    i = 1                          # initiating iterating index

    while i <= N
        vecAdd += particle_list[i] # adding each particle's contribution to the total
        i += 1
    end

    return vecAdd/N                 # dividing my total number of particles (note it only works with equal mass particles)
end;

# recursive function to get the centers of mass of all nodes
function get_COMs(tree, COM_list)
    # if node has only one particle storing particle position as COM of the node
    if length(tree.bodies) == 1
        push!(COM_list, tree.bodies[1])
    end

    push!(COM_list, tree.COM)                 # appending the center of mass position to the list
    for i in 1:8                              # iterating over the 8 octants
        if typeof(tree.nodes[i]) == TreeNode  # convoluted way of checking if the node is empty
            get_COMs(tree.nodes[i], COM_list) # recursive step
        end
    end
end;

# Recursively tracing the tree to find the centers of mass of the non-empty nodes
# Input: tree structure
# Output: list of center of mass locations for all "non-empty" nodes

function list_COMs(tree)

    COM_list = []   # intiating an empty list to store the centers of mass

    get_COMs(tree, COM_list)                          # updating list
    COM_list = [x for x in COM_list if !isnan(x[1])]  # removing the NaNs from the list

    return COM_list
end;

################################
######### MASS LIST ############
################################

# recursive function to get the masses of all nodes
function get_mass(tree, mass_list)

    # if node has only one particle, storing particle mass as mass of the node
    if length(tree.bodies) == 1
        push!(mass_list, tree.mass)
    end

    push!(mass_list, tree.mass)                  # appending the center of mass position to the
    list
    for i in 1:8                                # iterating over the 8 octants
        if typeof(tree.nodes[i]) == TreeNode    # convoluted way of checking if the node is empty
            get_mass(tree.nodes[i], mass_list) # recursive step
        end
    end
end;

# Recursively tracing the tree to find the centers of mass of the non-empty nodes
# Input: tree structure
# Output: list of masses of corresponding to the COMs generated by list_COM() function

function list_mass(tree)

    mass_list = []   # intiating an empty list to store the centers of mass

    get_mass(tree, mass_list)                         # updating list
    mass_list = [x for x in mass_list if x > 0]        # removing the NaNs from the list

    return mass_list
end;


################################
######### NODE LIST ############
################################

# recusrively finding angle subtended by each non-empty node on a particle
function get_nodes(particle, tree, theta_cutoff, outLists)

    # base case: angle_subtended is less than theta_cutoff
    if angle_subtended(particle, tree) <= theta_cutoff || length(tree.bodies) == 1
        return push!(outLists, tree.COM)
    end

    # recursively finding the centers of mass of the nodes to be considered
    for i in 1:8     # iterating through octants
        if typeof(tree.nodes[i]) == TreeNode
            push!(outLists, get_nodes(particle, tree.nodes[i], theta_cutoff, outLists))
        end
    end

    return outLists
end;

function list_nodes(p, cutoff, tree)

    consider_COM = []
    consider_COM = get_nodes(particles[p], tree, cutoff, consider_COM)

    # removing non-vector elements from the list
    consider_COM = [x for x in consider_COM if typeof(x) == Vector{Float64} && !isnan(x[1])];

    return consider_COM
end;
```

Running the code is straightforward. We start with particle locations stored in a clean CSV file (developed from the provided data). Then using the farthest distances of the given particles, we compute the total size of the space. We divide the total space into octants manually to initiate the top leaf. Then we provide the particle list and top leaf data to treeGenerator script which fills up all the branches.

```julia
include("treeGenerator.jl");
include("treeAttributes.jl");

using CSV # importing libraries to read the data

# reading position of equal mass n-bodies from a csv file
pos = CSV.read("nbody.csv", DataFrame, header=true);
particles = [[pos.X[i], pos.Y[i], pos.Z[i]] for i in 1:Integer(nrow(pos))]; # creating a list of
    particles positions

# size of the parent node
```

```
init_bound = (minimum(pos.X), maximum(pos.X), minimum(pos.Y), maximum(pos.Y), minimum(pos.Z),
    maximum(pos.Z));
first_div = canvas_split(init_bound)                                    # splitting the
    parent node into 8 equal octants
top_leaf = TreeNode(init_bound, particles, get_COM(particles), first_div);     # initiating the
    top_leaf (parent node)

# building the tree starting with maximum recursive depth of 200
tree = build_tree(particles, top_leaf.nodes, 200);
```

Finally, a fairly ugly function had to be defined to visualize node boundaries as it served as a useful check during the code development process. We will see the results of this visualization code (and others) in action in the next section.

```
# special funcion to plot octant division of a node
function plot_nodes(nodes_boundaries, color, alpha)
    for i in 1:8
        xMin, xMax, yMin, yMax, zMin, zMax = nodes_boundaries[i]
        plot!([xMin, xMax], [yMin, yMin], [zMin, zMin], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMax], [yMax, yMax], [zMin, zMin], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMax], [yMin, yMin], [zMax, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMax], [yMax, yMax], [zMax, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMin], [yMin, yMax], [zMin, zMin], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMin], [yMin, yMax], [zMax, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMax, xMax], [yMin, yMax], [zMin, zMin], color=color, alpha=alpha, linewidth=2)
        plot!([xMax, xMax], [yMin, yMax], [zMax, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMin], [yMin, yMin], [zMin, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMin, xMin], [yMax, yMax], [zMin, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMax, xMax], [yMin, yMin], [zMin, zMax], color=color, alpha=alpha, linewidth=2)
        plot!([xMax, xMax], [yMax, yMax], [zMin, zMax], color=color, alpha=alpha, linewidth=2)
    end
end;
```

1.2 Results

Once the tree is generated, we can use some specially defined plotting functions to visualizing the data in three dimensions. For example, the following code snippet chooses the first particle from the particle list and computes the angle subtended on the particle by a node generated on the third level of the tree. Then, it proceeds to plot the node boundaries to illustrate division into octants, positions of all n-bodies and the angle created between desired node diagonal and chosen particle. Using this functionality, the list (added in Appendix 1) of all 312 nodes with non-zero masses and their center of mass coordinates were enumerated .

```
# illustrative visualization of subtended angle calculation for a single particle and single node
p, n = 1, 1 #particle index, node index

node = tree.nodes[5].nodes[3].nodes[n]
xMin, xMax, yMin, yMax, zMin, zMax = node.bounds[1][1], node.bounds[8][2], node.bounds[1][3], node
    .bounds[8][4], node.bounds[1][5], node.bounds[8][6];

angle = round(angle_subtended(particles[p], node))

scatter(pos.X, pos.Y, pos.Z, legend = false, markersize=0.6, alpha=0.8, label="All Particles");

# plotting subtended angle
scatter!((particles[p][1], particles[p][2], particles[p][3]), color="red", legend = false,
    markersize=2) # plotting the particle
plot!([xMin, xMax], [yMin, yMax], [zMin, zMax], color="blue", linewidth=3) # plotting the node
    diagonal

# plotting lines connecting the particle to edges of the node diagonal
plot!([particles[p][1], xMax], [particles[p][2], yMax], [particles[p][3], zMax], color="blue",
    linewidth=3) # plotting the subtended angle
plot!([particles[p][1], xMin], [particles[p][2], yMin], [particles[p][3], zMin], color="blue",
    linewidth=3) # plotting the subtended angle


# plotting the node boundaries
plot_nodes(tree.bounds, "red", 0.4);
plot_nodes(tree.nodes[5].bounds, "red", 0.2);
plot_nodes(tree.nodes[5].nodes[3].bounds, "black", 0.3);
display(plot!(dpi=200, size=(900, 800), camera=(15, 25, 250), background_color="white", title="
    Canvas Spliting (node depth = 3) and Subtended Angle ~ $angle",
             xlabel="x", ylabel="y", zlabel="z"))
```

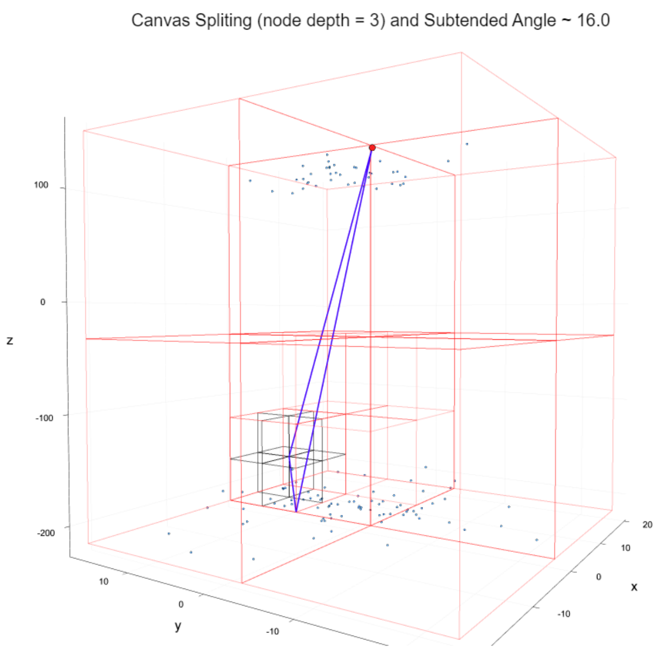

Figure 1: Plotting the 16^o angle (bold lines) subtended by a third layer leaf of the tree on the first particle (bold spot). Other particles from the list plotted as smaller dots and the node boundaries are plotted as light lines.

We can run the exact same code while choosing some other particle to see what angle is subtended by the same node on the particle. By comparing the results of Fig. 2 and Fig. 1, we can clearly see how the angle subtended on a particle closer to a particle node is much greater than the angle subtended on a particle farther away.

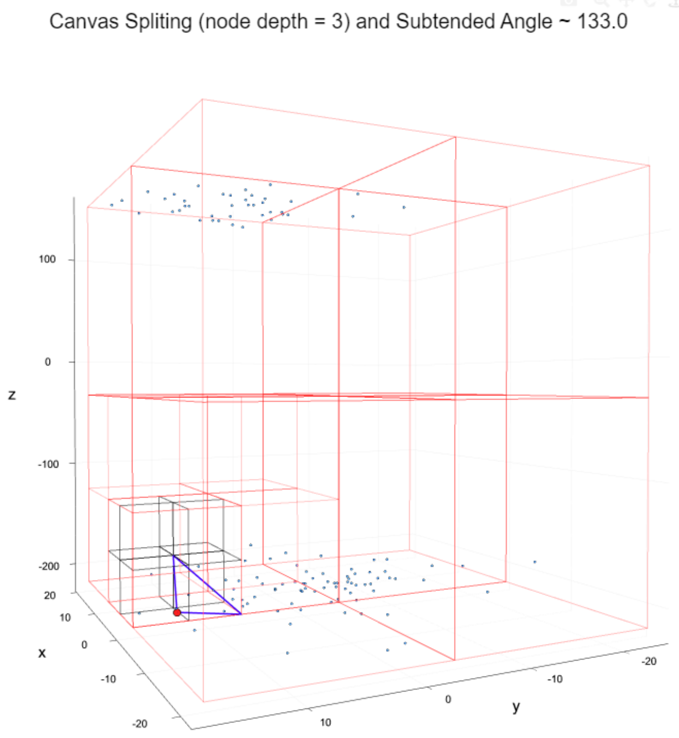

Figure 2: Plotting the 113^o angle (bold lines) subtended by a third layer leaf of the tree on the 50th particle (bold spot). Other particles from the list plotted as smaller dots and the node boundaries are plotted as light lines.

We can set a cut-off angle such that only nodes subtending an angle smaller than the cutoff would be listed for further computations. This is achieved by calling the `list_nodes(p, cutoff, tree)` function which takes the particle index of interest, cutoff angle and the data tree as its input. **Though I noticed the particles are broadly situated on two parallel planes, with $\theta_o = 1^o$, `list_nodes()` output is 120 entries long (119 nodes to consider + the particle's self node).** However, Fig. 3 shows how the number of nodes to be considered during force calculations falls as we increase the cutoff angle for a random selection of a few particles.

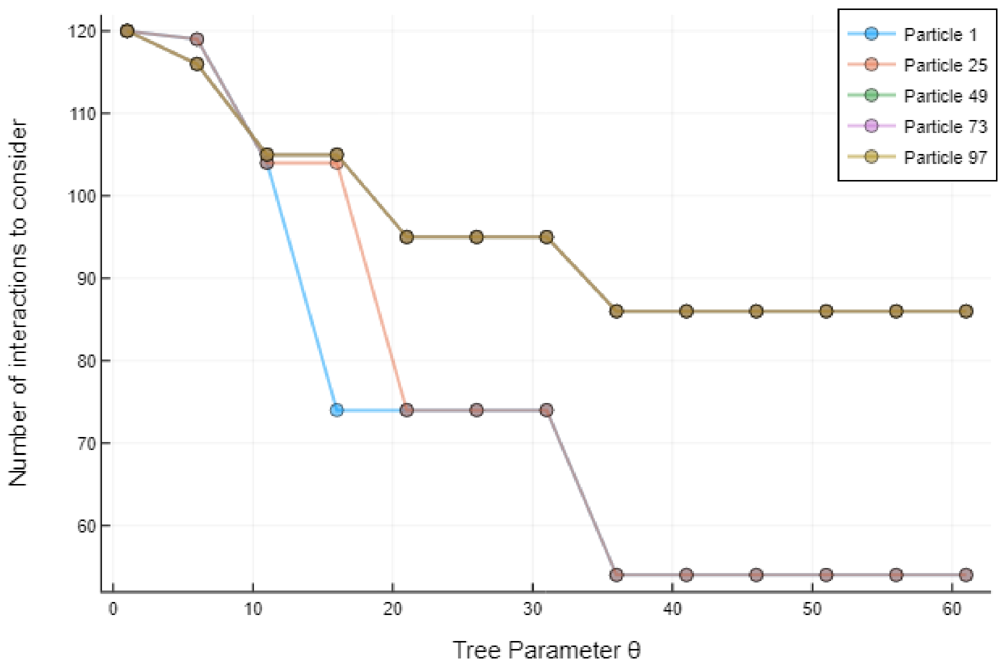

Figure 3: Relation between cutoff angle and number of nodes to consider

Now that the novel aspects of writing an octree data structure and recursively parsing through it have been developed, I look forward to developing a complete Barnes-Hut n-body solver in the coming weeks.

Appendix: Node COM Locations and Masses

To generate the desired list of center of mass locations for each non-empty node and their respective masses, we generate the tree and call the following functions

```
COM_list = list_COMs(tree);  # list of centers of mass of all non-empty nodes
mass_list = list_mass(tree); # list of masses of all non-empty nodes
out = [(COM_list[i], mass_list[i]) for i in eachindex(mass_list)] # array to store

using DelimitedFiles          # required for writing ouput files
writedlm("out.txt", out)      # writes to a text file ([COM location, mass] for each node)
```

The data is too big to be meaningfully visualized in a 2D black and white non-interactive medium. Therefore, Fig. 4 displays center of mass locations along with node masses for the first two layers of the tree (top leaf and the first splitting of space into octants) to demonstrate the output matches the intuition.

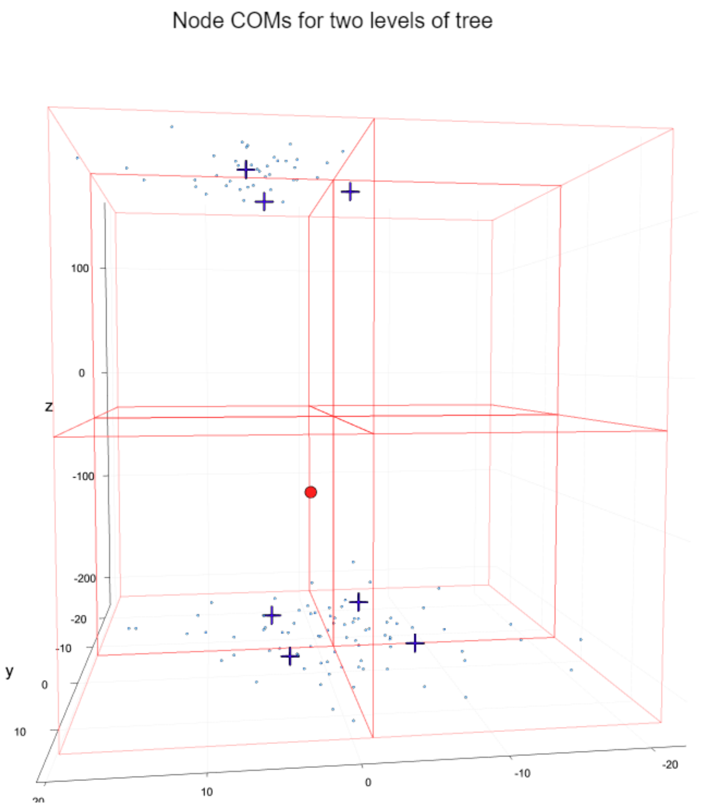

Figure 4: Node Center of masses positions. The big circle corresponds to the COM of the total n-body system and the crosses represent the nodes of the first layer of the octree. Smallest dots correspond to the equal mass particles in space.

This generates 312 entries corresponding to 312 non-empty leafs at various levels of the tree.