UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ PEDRO MARTINEZ
LUCAS ASSIS
RÉGES OBERDERFER

Modelos de Linguagens de Programação - SchnorR Doidão: Galáxias em R

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2017

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho consiste na implementação de um problema em dois paradigmas diferentes: funcional e orientado a objeto. A linguagem a ser utilizada deve ser a mesma para ambos os paradigmas.

O objetivo deste trabalho consiste em fornecer aos alunos a oportunidade de estudar uma linguagem de programação moderna com características híbridas (i.e., multiparadigma). O trabalho permitirá aos alunos demonstrarem que aprenderam os princípios de programação relacionados com os diferentes paradigmas estudados ao longo do semestre, demonstrando, ainda, a capacidade de analisar e avaliar linguagens de programação, seguindo os critérios abordados em aula.

As implementações, tanto funcional quanto orientada a objeto, e relatório estão disponíveis em <https://github.com/lbassis/galaxias>.

1.1 Linguagem Escolhida: R

R é um projeto GNU (FREE SOFTWARE FOUNDATION, 2018) voltado a programação estatística e gráfica. Faz parte dele uma linguagem multiparadigma (vetorial, orientada a objetos, imperativa, funcional, procedural e reflexiva) com tipagem dinâmica, fundamentada por S e inspirada por Scheme.

A linguagem fornece uma grande variedade de ferramentas para estatística, como modelagem linear e não-linear, classificação, clustering, testes estatísticos. Um dos seus pontos fortes é a facilidade com que gráficos bem construídos e prontos para uso em publicações científicas podem ser gerados. Esses gráficos incluem formulas e símbolos matemáticos onde necessário.

Resumindo, R é um conjunto integrado de métodos para manipulação de dados, cálculos e display gráfico que inclui (THE R FOUNDATION, 2017):

- Uma série de operações sobre vetores, em particular matrizes;
- Uma extensa coleção de ferramentas intermediárias para análise de dados;
- Ferramentas de plot para análise de dados.
- Uma linguagem de programação bem desenvolvida, simples e eficaz que inclui condicionais, loops, funções recursivas definidas pelo usuário e funções de entrada e

saída.

1.2 Problema escolhido: Galáxias

O problema consiste em implementar um simulador de partículas, considerando forças físicas de repulsão e atração. Serão utilizadas leis gravitacionais para simular órbitas de estrelas e planetas. Além disso, uma interface será criada para que seja possível acompanhar o movimento simulado.

Para suportar a simulação de muitos corpos, será utilizado o algoritmo de Barnes-Hut (BARNES; HUT, 1986), que consiste em subdividir o espaço em quadrantes (no caso de um espaço bi-dimensional) recursivamente até atingir um dado limite de modo a simplificar o problema sem perdas consideráveis de precisão no modelo.

1.2.1 Problema de N-corpos

$$\theta = \frac{d}{r} \tag{1.1}$$

Uma determinada região do espaço contém N corpos, cada um com suas próprias propriedades como sua massa e com um campo de forças gravitacionais resultante agindo sobre si. A massa de cada corpo é uma característica física, e o campo resultante pode ser calculado a partir das posições dos corpos e de suas massas. Segundo a primeira lei de Newton, os corpos sofrem uma aceleração devido ao campo de forças gravitacionais resultante. Logo, a aceleração de um corpo e a sua posição dependem do campo de força resultante da interação entre todos os corpos.

Numa simulação desse problema, parte-se de um estado s em que as posições e as velocidades dos corpos são conhecidas. A partir de s, calculam-se as forças entre todos os corpos seguindo a equação 1.2, onde G é igual à constante gravitacional universal, m_1 é a massa do corpo 1, m_2 é a massa do corpo 2 e r é a distância entre os dois corpos. Visto que é necessário calcular o campo de forças resultante para cada um dos corpos, $\frac{1}{2}N * (N-1)$ forças são calculadas. Logo, a complexidade de calcular todas as forças é $O(N^2)$.

$$F = G\frac{m_1.m_2}{r^2} \tag{1.2}$$

Uma vez que as forças foram calculadas, as acelerações das partículas são obtidas seguindo a equação 1.3. Para o cálculo das novas posições e velocidades das partículas, assume-se um tempo Δt entre cada etapa da simulação. Também se asssume que Δt é suficientemente pequeno para que se possa fazer a aproximação que a aceleração é constante durante esse período de tempo. Utilizam-se, pois, as equações 1.4 e 1.5 do Movimento Uniformemente Variado (MUV) para atualizar as posições e as velocidades das partículas.

$$a = \frac{F}{m} \tag{1.3}$$

$$v_{prox} = v_{atual} + a * \Delta t \tag{1.4}$$

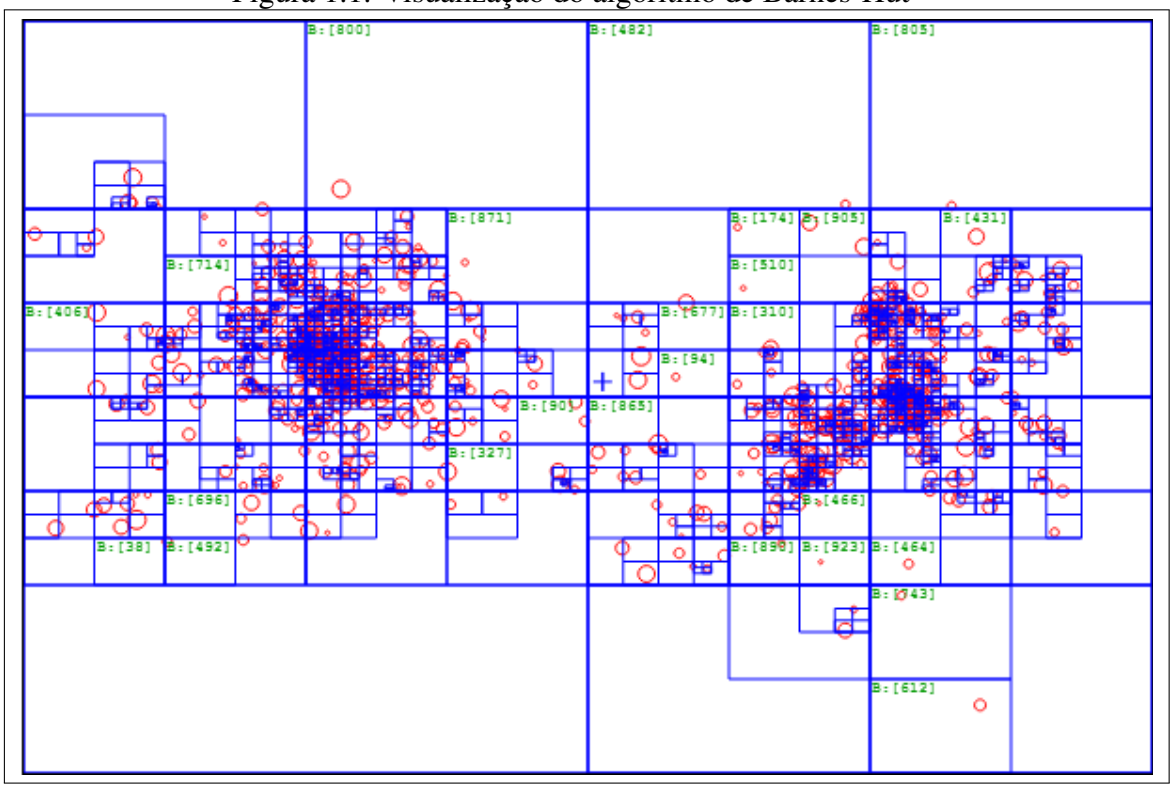
$$s_{prox} = s_{atual} + v_{atual} * \Delta t + \frac{a}{2} * (\Delta t)^2 \tag{1.5}$$

1.2.2 Algoritmo de Barnes-Hut

O algoritmo de Barnes-Hut resolve o problema de N-corpos numa complexidade inferior àquela apresentada na seção anterior. Em vez de calcular todas as forças, uma aproximação baseada em árvores que divide o espaço em quadrantes é utilizada. Esse algoritmo tem complexidade igual a $O(N log(N))$

O algoritmo de Barnes-Hut reduz o número de forças calculadas agrupando partículas. A aproximação utilizada é que um grupo de partículas distantes pode ser reduzido a uma única partícula de centro igual ao centro de massa do grupo. Dessa forma, só é necessário calcular as forças uma a uma para partículas muito próximas, enquanto que as distantes têm seu cálculo simplificado. A simplificação só é válida quando a distância da partícula ao grupo é grande em relação ao tamanho do quadrante do grupo. Para determinar se a razão entre o tamanho d do quadrante do grupo e a distância r é suficiente, utiliza-se o critério da equação (1.1). O algoritmo de Barnes-Hut aplica $\theta = 1$ subdividindo recursivamente o espaço em quadrantes até que apenas uma partícula por quadrante exista, como visto na Figura 1.1.

Figura 1.1: Visualização do algoritmo de Barnes-Hut



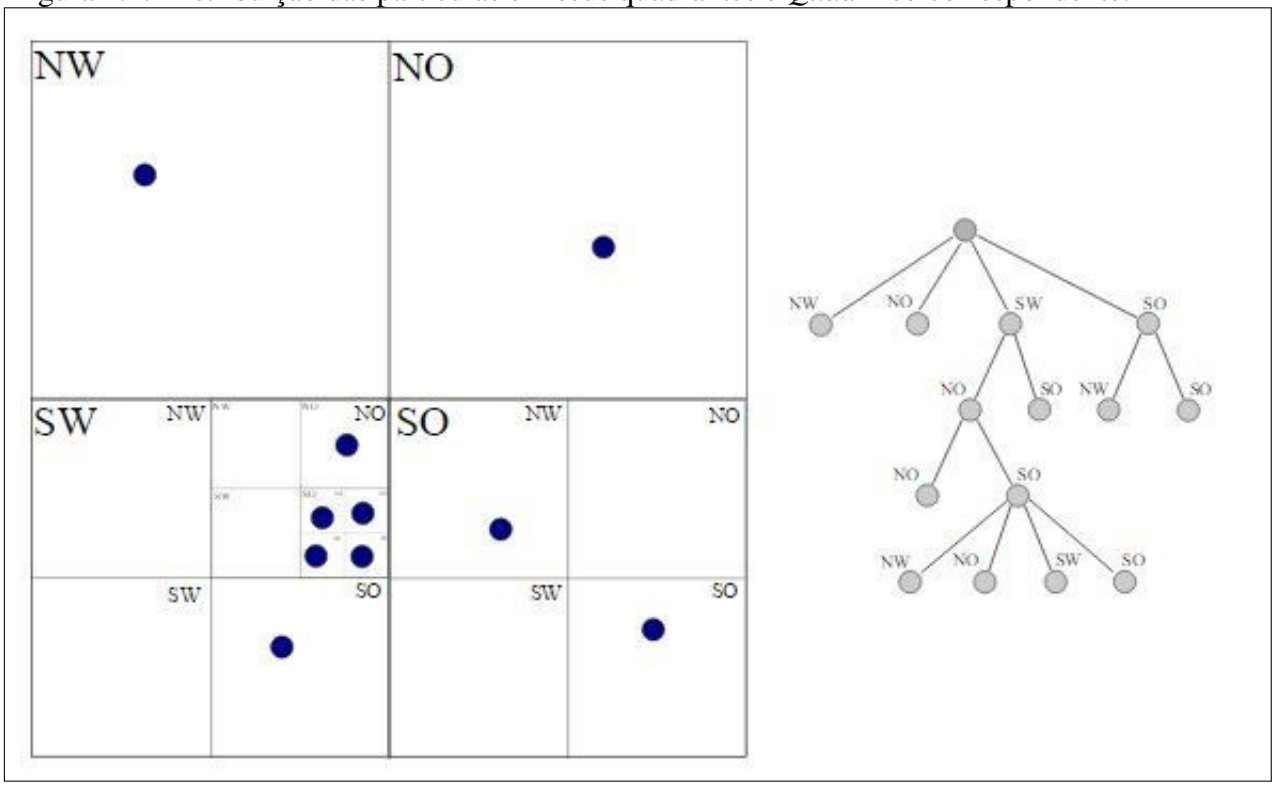
Fonte: (ANSARI, 2016)

2 IMPLEMENTAÇÃO

2.1 Implementação Física

O algoritmo recebe como entrada uma lista de partículas com suas posições e suas velocidades. Começa-se subdivindindo o espaço recursivamente em quadrantes até que haja no máximo uma partícula por quadrante. Ao mesmo tempo que os quadrantes são divididos, popula-se uma *QuadTree* hierarquicamente de modo que cada nodo folha seja uma partícula do espaço, como pode ser visto na Figura 2.1. Em um segundo momento, os centros de massa para cada quadrante são calculados. Após isso, calculam-se as forças resultantes para todas as partículas, reduzindo o número de forças calculadas quando um quadrante passa na regra da equação (1.1). Finalmente, as posições e velocidades das partículas são atualizadas considerando um passo de atualização que corresponde ao tempo percorrido entre uma etapa da simulação e a seguinte. A implementação em pseudocódigo pode ser vista no algoritmo 1.

Figura 2.1: Distribuição das partículas em seus quadrantes e *QuadTree* correspondente.



Fonte: (BERG, 2017)

Algorithm 1: Algoritmo de Barnes-Hut para simulação de N-corpos

1 function barnesHut $(particles, positions, velocities, masses)$;
Input : Uma lista de partículas, suas posições, velocidades e massas
Output: As posições e velocidades das partículas atualizadas
2 $quadTree = createQuadTree(particles, positions, velocities, masses)$;
3 $quadTree = computeMassDistribution(quadTree)$;
4 $quadTree = computeResultantForces(quadTree)$;
5 $newVelocities = updateVelocities(quadTree)$;
6 $newPositions = updatePositions(quadTree)$;
7 return $[newVelocities, newPositions]$;

2.2 Implementação Gráfica

Em uma abordagem inicial, a ideia foi utilizar diversos gráficos carregados em sequência no ambiente *Rstudio*, com um pequeno intervalo de espera entre cada geração. No entanto, a performance não foi satisfatória; apesar de, inicialmente e com um número pequeno de partículas, a animação ficasse fluída, com o passar do tempo o tempo entre as gerações é cada vez maior, tornando o algoritmo muito lento em pouco tempo.

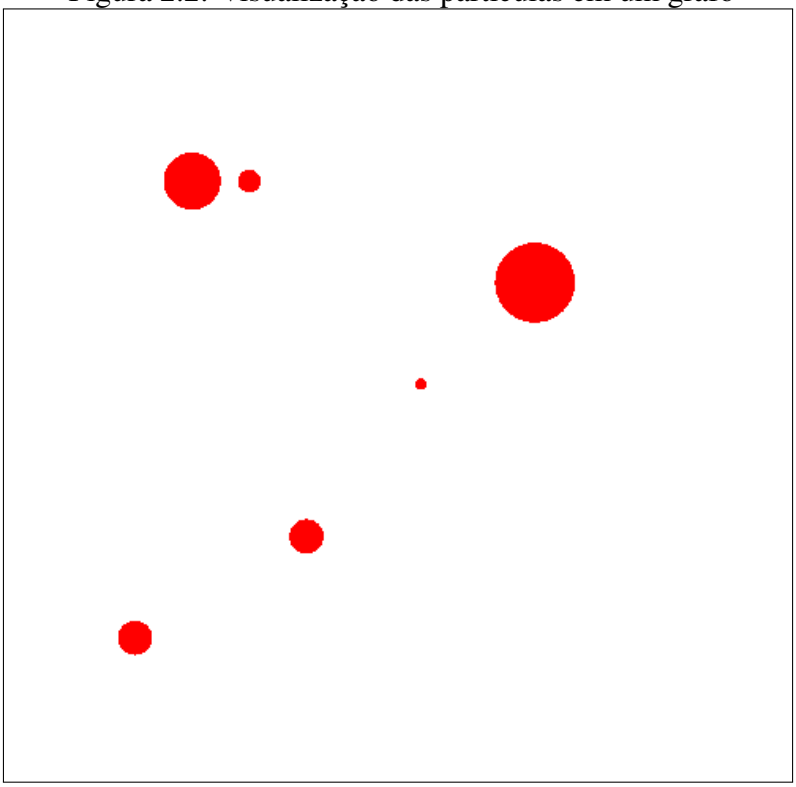
A partir daí, a ideia escolhida foi a de selecionar inicialmente um número de iterações; depois disso, são geradas todas as imagens necessárias para representar esse número e, por fim, as imagens são transformadas em um arquivo *.gif*. Dessa maneira, o algoritmo perde a capacidade de ser demonstrado em tempo real, já que a duração da simulação é definida antes da execução da mesma. No entanto, dessa maneira foi possível obter uma animação adequada ao esperado e, portanto, essa foi a solução escolhida.

A geração das imagens é feita a partir de um laço. Na implementação funcional, por exemplo, a função *draw_circle*, mostrada no algoritmo 2, desenha uma partícula na tela. O laço itera por todas as partículas existentes, gerando um gráfico com todas elas desenhadas num dado instante de tempo. Cada um desses gráficos é salvo para que, finalizado o número de iterações definido pelo usuário, uma função da biblioteca *ImageMagick* transforme o conjunto de imagens em um arquivo *.gif* animado.

Algorithm 2: Algoritmo que desenha uma partícula

1 function draw_particle $(particle)$;
Input: Uma partícula encapsulada em uma estrutura de nodo
2 $draw\_circle(qnode\_x(node), qnode\_y(node), qnode\_mass(node), col =$ "red", $nv = 1000, border = NA, lty = 1, lwd = 1)$;

Figura 2.2: Visualização das partículas em um grafo



2.3 Implementação no Paradigma Funcional

Nessa sessão, será realizada uma análise específica sobre a implementação do algoritmo de Barnes-Hut e sua visualização gráfica utilizando conceitos do paradigma funcional. Segue abaixo a descrição e demonstração de algumas técnicas tipicamente funcionais utilizadas na elaboração deste trabalho.

2.3.1 Elementos imutáveis

A utilização de elementos imutáveis é recorrente na solução desenvolvida. Como exemplo, pode-se citar a função de atualização da posição de uma partícula. A cada iteração do laço, uma nova partícula é criada (e, consequentemente, uma nova lista de partículas) para que esta ocupe a nova posição calculada com a velocidade também atualizada. A função que cria a nova partícula é mostrada no algoritmo 3.

Algorithm 3: Algoritmo de atualização de posição

```
1 particle_setPositionAndVelocity (particle, point, velocity);
  Input  : Uma partícula e dois novos pontos representando sua nova posição e
           nova velocidade
  Output: Uma nova partícula criada com os parâmetros atualizados
2 particle_setPositionAndVelocity <- function(particle, point, velocity);
3 new_particle(point, qnode_mass(particle), velocity,;
4 new_point(qnode_fx(particle), qnode_fy(particle)), qnode_size(particle));
```

2.3.2 Funções não nomeadas

Funções não nomeadas são utilizadas frequentemente para que elas possam ser passadas como se fossem um elemento de primeira ordem da implementação. Neste trabalho, por exemplo, foram utilizadas para definir a função da velocidade em função do tempo e da distância, demonstrada no algoritmo 4 que é aplicada posteriormente nas componentes horizontal e vertical da velocidade de uma partícula.

Algorithm 4: Trecho do algoritmo do cálculo da nova posição

```
1 updateVelocityAndPosition (node, deltaT);
  Input  : Uma partícula encapsulada em um nodo e um intervalo de tempo a
           ser considerado
  Output: Uma nova partícula criada com os parâmetros atualizados
2 calcVelocityComponent <- function(v,a,t)(v + a * t);
3 newVx <- calcVelocityComponent(vx, ax, deltaT);
4 newVy <- calcVelocityComponent(vy, ay, deltaT);
```

2.3.3 Currying

O *Currying* é uma técnica que transforma uma função de duas entradas em uma função com uma entrada que retorna outra função de uma entrada. Nesse trabalho, essa implementação foi utilizada para acessar as colunas dos *data.frames* criados - os *qnodes*. O algoritmo 5 mostra a implementação da função "original".

A partir daí, torna-se fácil construir as funções específicas de cada seletor, que utilizam como argumento apenas o *qnode* a ser acessado. Utilizando a função *Curry* do pacote *functional*, temos a seguinte implementação para obter o parâmetro *point_x*, por exemplo.

Algorithm 5: Algoritmo seletor de colunas do dataframe node

```
1 qnode_data (data, node);
  Input  : A descrição de uma coluna do dataframe e uma partícula
           encapsulada em um nodo
  Output: O valor associado à coluna selecionada no nodo fornecido
2 if(is.data.frame(node));
3 node[data][[1]];
4 elseif(!qnode_empty(node));
5 node[[1]][data][[1]];
6 else0;
```

Algorithm 6: Algoritmo seletor da coluna x do dataframe node

```
1 qnode_x (node);
  Input  : Uma partícula encapsulada em um nodo
  Output: O valor da coluna x no nodo fornecido
2 Curry(qnode_data, data = "point_x");
```

2.3.4 Funções de ordem superior próprias

Funções de ordem superior aceitam que outras funções sejam passadas como parâmetro. Como exemplo, temos a implementação da funçao *best_point*, conforme visto na função 7. Para determinar os limites do gráfico e o centro geométrico da simulação, são computados o ponto mais à esquerda/mais acima e a maior distância em uma das dimensões até outro ponto pertencente à simulação. Tendo a função de alta ordem *best_point* e usando as funções primitivas min e max, é possível atualizar os limites *top_left* e *bottom_right* (este usado para atualizar a maior distância entre os pontos) recursivamente.

Algorithm 7: Algoritmo que encontra os extremos dos quadrantes

```
1 best_point (p, q, fun);
  Input  : Um ponto, um quadrante e uma função para determinar o ponto
           mais apropriado
  Output: O ponto mais extremo do quadrante
2 new_point(fun(q["x"][[1]], p["x"][[1]]), fun(q["y"][[1]], p["y"][[1]]));;
```

2.3.5 Funções de ordem superior prontas

Um dos exemplos a ser citado de funções de ordem superior já implementadas pela linguagem é a função *lapply*. Ela funciona como um funtor, mapeando uma função

que recebe um elemento e generaliza-na para que ela receba uma lista desses elementos, aplicando a função inicial a todos eles. No algoritmo 2, por exemplo, desenha-se uma partícula. A partir dele, pode-se construir o algoritmo 8, generalizando sua implementação para trabalhar com uma lista de partículas em vez de um único elemento.

Algorithm 8: Algoritmo que desenha todas as partículas do instante atual

```
1 draw_particles (particles);
  Input: Uma lista de partículas encapsulada em nodos
2 invisible(lapply(particles, draw_particle));
```

2.3.6 Funções como elemento de primeira ordem

Em linguagens funcionais, é possível que funções sejam passadas como parâmetros de outras funções, funcionando como elementos de primeira ordem. Essa técnica foi utilizada no *loop* que desenha as partículas para indicar qual a função que atualizaria suas posições, conforme mostrado no algoritmo 9.

Algorithm 9: Laço que desenha os quadros da animação

```
1  drawing_ loop (particles, iteractions, name, updatePosAndVel);
   Input: Uma lista de partículas encapsulada em nodos, o número de iterações
          do laço, o nome dos arquivos temporários e uma função que devolve
          uma lista de nodes
2  if(iteractions == 0);
3  system("convert -delay 10 *.jpg result.gif");
4  file.remove(list.files(pattern = ".jpg"));
5  else;
6  name <- paste(name,"a");
7  filename <- paste(name,".jpg");
8  png(filename = filename);
9  plot.new();
10 frame();
11 draw_particles(normalize_masses(particles));
12 dev.off();
13 particles <- updatePosAndVel(qList_toParticles(particles));
14 drawing_loop(particles, iteractions - 1, name, updatePosAndVel);
```

2.3.7 Funções com pattern matching

Pattern matching é uma técnica que procura padrões dentro de sequências de valores, muito utilizada, por exemplo, em processamento de linguagens. Como esse trabalho é uma implementação que simula fenômenos físicos, o grupo não encontrou uma aplicação interessante para esse tipo de algoritmo, portanto não há nenhum exemplo a ser mostrado.

2.3.8 Recursão

A recursão foi utilizada frequentemente como iterador para que todos os elementos de uma lista fossem devidamente processados. Um possível exemplo é o algoritmo 9 mostrado na sessão anterior. Ao final de cada iteração, diminui-se em uma unidade o número de iterações restantes e a função *drawing_loop* é chamada novamente.

2.4 Implementação no Paradigma Orientado a Objetos

2.4.1 Classes

A construção base da orientação a objetos foi obviamente utilizada na elaboração deste trabalho. Foram utilizadas classes para representar pontos, quadrantes, partículas, nodos da árvore, pilhas e o universo como um todo.

A linguagem R fornece diversas implementações de classes (WICKHAM, 2017) com níveis de detalhamento diferentes. Essa implementação utiliza a *RefClass*, que conta com objetos mutáveis e envio de mensagens entre objetos.

Abaixo, o algoritmo 10 mostra a implementação da classe *Point* usando *RefClass*.

2.4.2 Encapsulamento

RefClasses, apesar de permitirem a declaração de métodos que pertencem a classes, o que é uma forma de encapsulamento, não permitem que sejam especificados os níveis de acesso dos elementos de cada classe. Portanto, todos os atributos e métodos das classes utilizadas nessa implementação são públicos. Apesar disso, todos os atributos contam com seus *setters* e *getters*, simulando uma implementação com atributos privados.

Algorithm 10: Definição da classe Point

```
1  Point <- setRefClass("Point",
2  fields = list(
3  x = "numeric",
4  y = "numeric"
5  ),
6  methods = list(
7  initialize = function(.Object,x=0,y=0) {
8  .self$x = x$
9  .self$y = y$
10 },
11 get_x = function() return(.self$x),
12 get_y = function() return(.self$y),
13 sum_op = function(p) {
14 Point(x = .self$x + p$x, y = .self$y + p$y)
15 },
16 sub_op = function(p) {
17 Point(x = .self$x - p$x, y = .self$y - p$y)
18 },
19 quadrant_i = function(quad_center) {
20 if (.self$x <= quad_center$x) {
21 if(.self$y <= quad_center$y) 1 else 2
22 } else {
23 if(.self$y <= quad_center$y) 3 else 4
24 }
25 },
26 subquadrant = function(quad) {
27 quad$quad_sub(.self$quadrant_i(quad$quad_center()))
28 },
29 set_x = function(x) .self$x = x,
30 set_y = function(y) .self$y = y
31 )
32 )
```

A *R6Class*, outra possível implementação da linguagem, conta com a opção de níveis de acesso (INTRODUCTION..., n.d.). No entanto, dessa forma não haveria a designação dos tipos de cada atributo da classe e portanto o grupo preferiu utilizar uma *RefClasses*.

2.4.3 Construtores

Conforme pode ser visto no algoritmo 10 na função *initialize*, é possível especificar os construtores das classes criadas. O parâmetro *.Object*, presente em todos os construtores, é o objeto protótipo da classe (INITIALIZE-METHODS..., n.d.).

2.4.4 Destrutores

A única classe em que se faz útil um destrutor é a *Qnode*, que conta com quatro listas de filhos. Portanto, o método 11, parte da implementação da classe, remove os elementos dessas listas assim que o *garbage collector* é invocado na execução do programa (isso pode ser feito forçadamente através do comando *gc()*).

Algorithm 11: Destrutor da classe Qnode

```
1 finalize = function() {
2 set_first_child(list())
3 set_second_child(list())
4 set_third_child(list())
5 set_fourth_child(list())
6 print("objeto finalizado")
7 }
```

2.4.5 Espaço de Nomes

Os espaços de nomes presentes nessa implementação são apenas as classes implementadas. No entanto, não há métodos com nomes iguais em classes diferentes, e por isso o resultado não seria diferente caso houvesse um único espaço de nomes.

2.4.6 Herança

O conceito de herança foi utilizado nas classes *Qnode* e *Particle*, sendo ambas especializações da classe *Point*. Como as duas classes especializadas compartilham informação de posição, em vez de incluir esses parâmetros, optou-se por fazê-las herdarem-nos da classe *Point*, que contém apenas essas informações.

2.4.7 Polimorfismo por Inclusão

2.4.8 Polimorfismo Paramétrico

O polimorfismo paramétrico foi utilizado na classe *Stack*. Na criação do objeto, é definida a classe cujos objetos serão incluídos na pilha. A linguagem em si não fornece mecanismos nativos para a verificação dos objetos incluídos, mas a implementação da função *push* se encarrega disso. Caso o usuário tente empilhar um objeto de uma classe diferente, uma exceção é acionada, conforme o algoritmo 1.

Algorithm 12: Função *push* da pilha que garante que os elementos empilhados são do tipo estabelecido na criação da pilha

```
1 push = function(e) { if (class(e) != type) { stop("Argument type should be
    ",type,"but received argument has type ",class(e)) } else { .self$stack = c(e,
    .self&stack) } }
```

2.4.9 Polimorfismo por Sobrecarga

A sobrecarga é utilizada para reescrever um método para alguma classe em especial. Isso foi utilizado na classe *Point* para que fosse possível utilizar o operador de soma entre dois objetos dessa classe, somando suas componentes horizontais e verticais, resultando num novo ponto, conforme mostra o algoritmo 13.

Algorithm 13: Sobrecarga da operação de soma

```
1 setMethod("+", c("Point", "Point"), function(e1, e2)
2 Point(x = e1$get_x() + e2$get_x(), y = e1$get_y() + e2$get_y())
3 )
```

2.4.10 Delegates

Delegates consistem em utilizar elementos para selecionar qual implementação dentre uma gama de opções será utilizada. A função *compute_single_force*, mostrada no algoritmo 14, possui um delegate que decide qual será a função de distância aplicada aos pontos computados. Nessa aplicação a única opção utilizada é a distância euclidiana, mas o grupo optou por implementar a construção mesmo assim para ilustrar sua utilização na linguagem.

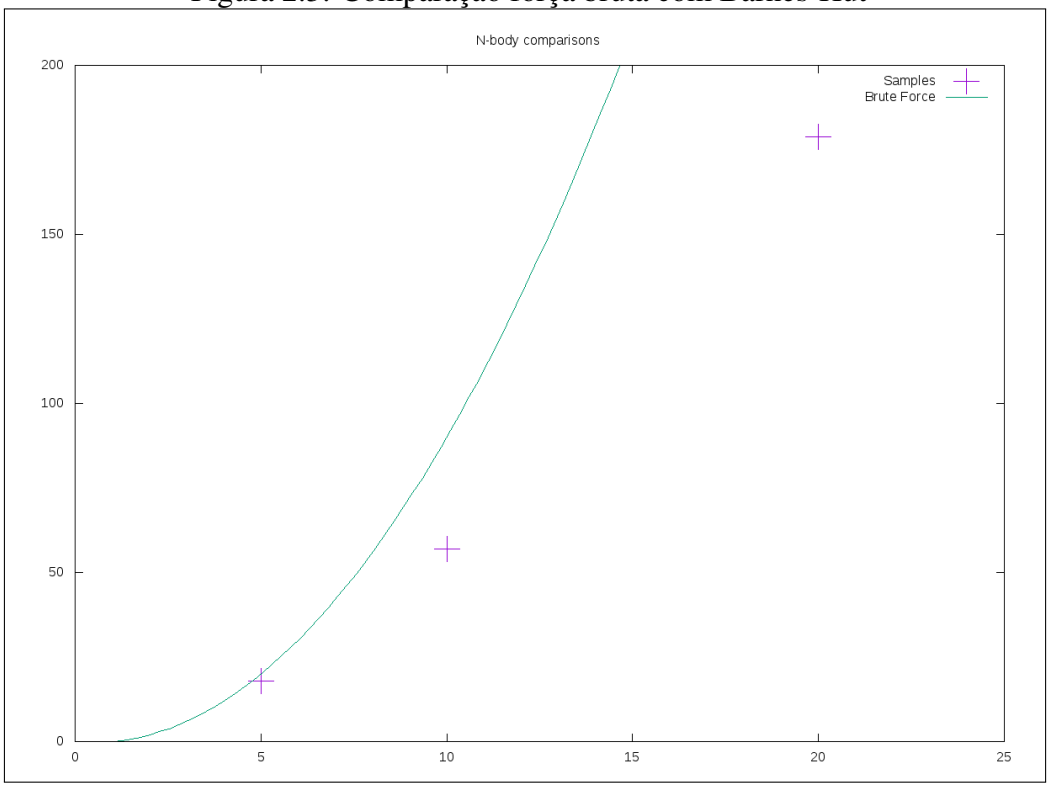
Algorithm 14: Método que faz uso do delegate

```
1  compute_single_force = function(node, distance_delegate) {
2  G = 6.67408*(10^(-11))
3  m1 = .self$get_mass()
4  m2 = node$get_mass()
5  d = distance_delegate(node)
6  if (d > 0) {
7  dx = .self$get_x() - node$get_x()
8  dy = .self$get_y() - node$get_y()
9  f = (G*m1*m2)/d^2
10 fx = f*dx/d
11 fy = f*dy/d
12 return(Point$new(x=fx,y=fy))
13 }
14 }
```

2.4.11 Análise do algoritmo

O grupo teve dificuldade em encontrar valores de variáveis adequados para realizar as simulações. Portanto, não foi gerado nenhum caso de teste grande o suficiente para comprovar a eficiência do algoritmo em relação à uma solução em força bruta. Por isso, foi elaborado o gráfico 2.3 a partir do caso de teste elaborado, comparando a quantidade de comparações do algoritmo de força bruta com a solução implementada. O eixo X representa o número de partículas, enquanto o eixo Y representa o número de comparações.

Figura 2.3: Comparação força bruta com Barnes-Hut



2.4.12 Análise da linguagem

A seguir são atribuídas notas às diferentes características da linguagem R considerando as duas implementações realizadas. As notas foram dadas em um intervalo [0,5].

A seguir, temos as análises que levaram a cada uma das notas mostradas na tabela.

2.4.12.1 Simplicidade

O operador <- vai contra a simplicidade da linguagem, já que ele implementa a atribuição, assim como o operador =. Além disso, existem diversas implementações de classes, o que dificulta na elaboração do projeto do algoritmo.

2.4.12.2 Ortogonalidade

Diversas funções do programa alternam entre data.frames e listas, e a linguagem oferece suporte a isso naturalmente.

Tabela 2.1: Características da linguagem R

Característica	*Nota [0, 5]*
Simplicidade	3.0
Ortogonalidade	4.5
Expressividade	3.0
Adequabilidade e variedade de estruturas de controle	3.5
Mecanismos de definição de tipos	4.0
Suporte a abstração de dados	3.5
Suporte a abstração de processos	4.5
Modelo de tipos	3.5
Portabilidade	3.0
Reusabilidade	4.0
Suporte e documentação	3.0
Tamanho de código	4.0
Generalidade	3.0
Eficiência	4.0
Custo	2.5

Fonte: Autor

2.4.12.3 Expressividade

A expressividade é adequada no geral, porém no acesso à listas, a necessidade de usar dois colchetes para recuperar um elemento torna o código menos expressivo do que ele poderia ser. Como alternativa, temos as funções *head* e *tail*, porém as duas também contam com um fator incomum: a necessidade de especificar quantos elementos serão recuperados da lista.

2.4.12.4 Adequabilidade e variedade de estruturas de controle

As estruturas de controle existem, mas a implementação funcional utiliza recursão em seu lugar e na implementação orientada a objeto foi necessário implementar uma pilha. Apesar disso, alguns laços iterativos foram úteis para depuração do código.

2.4.12.5 Mecanismos de definição de tipos

O tipo data.frame, utilizado para representar tabelas, funciona como uma definição de estrutura e atendeu bem ao desejado no trabalho.

2.4.12.6 Suporte a abstração de dados

A linguagem fornece a estrutura data.frame, que equivale a uma tabela em que cada coluna pode ter um valor de diferentes tipos. No entanto, não é criada uma estrutura propriamente dita, com sua própria semântica. Já na orientação a objetos, as opções de implementação de classes, apesar de diminuirem a simplicidade, fornecem mais opções de abstração de dados.

2.4.12.7 Suporte a abstração de processos

Implementa funções recursivas e inclusive funções como elementos de primeira ordem, tornando o suporte a abstração de processos excelente.

2.4.12.8 Modelo de tipos

Sendo uma linguagem dinamicamente tipada, o modelo de tipos deixa a desejar. Em alguns casos, houveram problemas de difícil identificação que, com um sistema de tipos mais robusto, seriam facilmente detectáveis. No entanto, na implementação orientada a objeto foram utilizados recursos para checagem de tipos, como no algoritmo 1.

2.4.12.9 Portabilidade

A linguagem oferece diversos pacotes, mas todos os utilizados precisaram ser instalados posteriormente.

2.4.12.10 Reusabilidade

A utilização de funções como elementos de primeira ordem faz com que a linguagem possua uma ótima reusabilidade; já as múltiplas implementações de classes impactam negativamente nesse aspecto.

2.4.12.11 Suporte e documentação

Especialmente para este trabalho, que não utiliza a linguagem em um contexto em que ela se destaca, foi difícil encontrar referências para várias funcionalidades. No entanto, isso talvez esteja mais relacionado com a natureza da aplicação desenvolvida do que com a linguagem em si.

2.4.12.12 Tamanho de código

Nada especialmente diferente do comum. A linguagem não tem o código muito longo nem notoriamente curto.

2.4.12.13 Generalidade

Apesar de ter sido usada satisfatoriamente nesse trabalho, a linguagem fornece muito mais recursos para algoritmos estatísticos e similares do que qualquer outra implementação.

2.4.12.14 Eficiência

Para que essa característica seja analisada apropriadamente, seria necessário implementar o mesmo algoritmo em outro paradigma/linguagem, portanto fica difícil avaliar sem ao menos concluir a implementação orientada a objeto. No entanto, pode-se observar que a maior parte do tempo que o algoritmo leva é na criação dos arquivos de imagem, indicando que as funções próprias são eficientes.

2.4.12.15 Custo

A curva de aprendizado é relativamente íngreme; além disso, o Rstudio eventualmente apresenta problemas em sua instalação e na instalação de seus pacotes. Por fim, para uma simulação física como a aqui apresentada, as estruturas nativas da linguagem e do ambiente não facilitam, portanto conclui-se que nesse caso o custo é alto.

2.5 Implementação funcional x Implementação orientada a objetos

A implementação funcional, em geral, foi mais adequada para a elaboração desse trabalho, possivelmente devido ao seu aspecto matemático. A utilização de recursão para iterar entre os elementos pareceu mais natural, até porque na versão orientada a objetos foi necessário uma nova classe para realizar as iterações.

No entanto, a abstração de dados na orientação a objetos foi muito melhor. Enquanto na funcional haviam *dataframes*, que nada mais eram que tabelas se comportando como estruturas, na OO haviam classes propriamente ditas. A existência dos métodos específicos de cada classe deixou o código muito mais legível e claro.

Com isso, o grupo conclui que, apesar da implementação funcional ter sido melhor, a verdadeira melhor opção seria uma combinação entre os dois paradigmas que fizesse uso dos melhores aspectos de cada um deles.

3 CONCLUSÃO

Ao final deste trabalho, foi possível formar uma opinião sobre várias das construções da programação funcional que o grupo nunca havia implementado. No entanto, se tratando de uma simulação física, também foi confirmado que R não é a linguagem mais adequada para construir esse tipo de programa. O algoritmo implementado demonstra uma possibilidade de simular diversas partículas com uma complexidade reduzida. Contudo, devido à dificuldade para gerar casos de teste, só foram criadas simulações com um número pequeno de partículas, não aproveitando o potencial total do algoritmo. Por esse motivo, o grupo optou por elaborar o gráfico que demonstra a eficiência do algoritmo em relação à uma implementação em força bruta.

Houve algumas dificuldades iniciais para definir as estruturas da implementação, já que foi necessário encapsular as partículas juntamente com os subquadrantes.

Por fim, um problema que foi detectado no final da implementação foi que o grupo não implementou colisões. Dessa maneira, assim que duas partículas ficam muito próximas uma da outra, as suas acelerações crescem até que elas sejam "arremessadas"com uma velocidade irreal. No entanto, até a conclusão da implementação orientada a objeto, acredita-se que esse problema terá sido tratado.

REFERÊNCIAS

ANSARI, S. **Barnes-Hut N-body Simulation in HTML/Javascript**. 2016. Available from Internet: <https://github.com/Elucidation/Barnes-Hut-Tree-N-body-Implementation-in-HTML-Js>.

BARNES, J.; HUT, P. A hierarchical o(n log n) force-calculation algorithm. **Nature**, Nature Publishing Group SN -, v. 324, p. 446 EP –, Dec 1986. Available from Internet: <http://dx.doi.org/10.1038/324446a0>.

BERG, I. **The Barnes-Hut Galaxy Simulator**. 2017. Available from Internet: <http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator>.

FREE SOFTWARE FOUNDATION. **GNU Operating System**. 2018. Available from Internet: <https://www.gnu.org/>.

INITIALIZE-METHODS: Methods to Initialize New Objects from a Class. n.d. Available from Internet: <https://rdrr.io/r/methods/initialize-methods.html>.

INTRODUCTION to R6 classes. n.d. Available from Internet: <https://cran.r-project.org/web/packages/R6/vignettes/Introduction.html>.

THE R FOUNDATION. **R: What is R?** 2017. Available from Internet: <https://www.r-project.org/about.html>.

WICKHAM, H. **OO field guide**. 2017. Available from Internet: <http://adv-r.had.co.nz/OO-essentials.html>.