Simulações de sistemas dinâmicos: da pandemia ao movimento molecular

Leandro Martínez

leandro@iqm.unicamp.br

Última atualização: 5 de novembro de 2020

Sumário

1 Elementos básicos de programação: a linguagem Julia

A linguagem de escolha do curso é Julia. A escolha desta linguagem se deve à simplicidade de sua sintaxe e à eficiência numérica dos programas gerados. Julia é uma linguagem relativamente nova, sua primeira versão "estável" foi lançada em 2018. Ela foi criada para unir as vantagens de linguagens de *alto nível*, práticas e fáceis, como Python, com a eficiência de linguagens de *baixo nível*, como C e Fortran.

A principal diferença destas linguagens é que as alto nível são *interpretadas* interativamente, enquanto as baixo nível são *compiladas* e depois executadas. Isto significa que em Julia e Python, por exemplo, existe uma forma interativa de interagir com a linguagem, chamada de REPL[1]. As linguagens compiladas não possuem essa forma interativa de uso, e você deve primeiro escrever o código para depois usar um *compilador*, que é um programa que transforma seu código em um *programa* propriamente dito, que o computador executa.

Para entender porque Julia foi criada, talvez seja necessário conhecer mais profundamente como funcionam as outras linguagens. Mas basicamente, o que acontece é que as linguagem com as quais o usuário interage diretamente costumam ser pouco eficientes. Isto acontece porque ao transformar cada comando isoladamente em um código a ser executado, o interpretador da linguagem não tem ideia do que será feito e, então, não pode otimizar a forma como o computador entende o código. Há muitas coisas que podem ser feitas para melhorar a forma como o computador entende o código: por exemplo, é possível escolher onde vão ficar cada um dos números na memória RAM para que eles sejam acessados de forma mais rápida. Além disso, os processadores (seu Intel Core i7 por exemplo) têm a capacidade de fazer várias contas ao mesmo tempo, se alguém (o compilador) sabe que contas você quer fazer, ele pode usar essa capacidade. Esses são apenas dois exemplos, mas os compiladores são hoje em dia muito bons em transformar códigos em programas eficientes. As linguagens de alto nível (como Python, R, Matlab, etc.) não permitem o uso destas vantagens, e não são boas para programar códigos que precisam ser rápidos. Por outro lado, as linguagens de baixo nível (C, C++, Fortran, etc.) geram códigos rápidos, mas não permitem essa interação prática e rápida que as linguagens interpretadas permitem. Por estas razões, programas em Python, Matlab, etc, usam muito funções pré-programadas, que foram feitas em linguagens de baixo nível (geralmente Fortran e C), para as partes que demandam mais tempo computacional. O programador acaba tendo que aprender duas linguagens, uma para fazer programas práticos e interativos, outra para fazer a parte que realmente demanda recursos computacionais. Julia surgiu com o objetivo de resolver esta complicação.

Desta forma, Julia tem uma estrutura um tanto particular, que permite tanto seu uso

[1] de *Read-Eval-Print-Loop*, que significa que o interpretador lê o comando, o avalia, escreve o resultado e volta ao estado inicial.

como uma linguagem interpretada e prática como Python, mas a geração de códigos rápidos como os de Fortran e C. É uma linguagem que pode ser usada de forma interativa, e possui um REPL, por exemplo, contas, como

```
julia> 1 + 1
2
julia>
```

No entanto, em Julia, quando você define uma função, por exemplo,

```
julia> f(x) = x^2 + 2
f (generic function with 1 method)
julia> f(2)
6
julia>
```

a função vai ser compilada a primeira vez que for executada, no caso quando pedimos para que seja calculado `f(2)`. A compilação é muito rápida, e você provavelmente não vai perceber ela acontecendo na maior parte dos casos. Isto quer dizer que foi gerado um programa que calcula o valor da função, um programa eficiente como um programa de baixo nível em Fortran ou C.

Para uma função simples como essa a velocidade não faz nenhuma diferença. Mas quando o programa faz coisas complicadas, custosas, a diferença é muito grande. Portanto, em Julia, o esperado é que tentemos programar tudo dentro de funções.

Julia foi criada, em princípio, para computação científica. Isto é, para fazer contas. Por isso, possui uma sintaxe bastante simples (muito similar às do Matlab e Fortran), e muitos recursos para lidar com vetores, matrizes, etc. Para a computação científica o importante é fazer contas.

Há vários cursos online e tutoriais sobre Julia. Uma busca na internet por "Julia language", permite o acesso a uma grande variedade de material gratuito. Este curso, no entanto, não é um curso de Julia somente. Este curso pretende introduzir os estudantes a alguns dos elementos fundamentais da programação numérica de alto desempenho. As simulações requerem uma linguagem de programação, e Julia é a linguagem de escolha porque sua sintaxe é simples e natural. Espera-se que o aluno aprenda a linguagem de programação com exemplos e resolvendo problemas (como eu aprendi). Nenhuma importância será dada a estruturas de linguagem e programação exceto no momento em que sejam requeridas. Ao final do curso, espera-se que o aluno adquira uma razoável familiaridade com a programação como conceito e com a linguagem, e possa buscar as ferramentas corretas para a resolução de seus próprios

problemas aplicados. Ainda, vale dizer que Julia pode ser considerado substituta de linguagens como Matlab, Fortran e Python, para a maior parte dos problemas.

1.1 Instalação e primeiros passos

O interpretador da linguagem é gratuito, pode ser instalado em qualquer plataforma (Linux, Windows, MacOS). Obtenha a versão mais recente (1.5 pelo menos), em:

https://julialang.org

Instale no seu computador, seguindo as instruções do site para seu sistema operacional. Vale notar que existe uma documentação extensa em

https://docs.julialang.org

que deverá ser consultada em caso de dúvida de uso de qualquer função específica. No entanto, esta documentação às vezes é muito técnica, e fica mais fácil encontrar a resposta no Google em um dos fóruns de discussão, ou perguntar para alguém que entenda.

Uma vez instalado o interpretador, você poderá usar o modo interativo, digitando (no terminal ou prompt de comando) "`julia`", abrindo o modo interativo na forma

```
   _       _ _(_)_     | Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   | Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  | Version 1.5.1 (2020-08-25)
 _/ |\__'_|_|_|\__'_|  | Official https://julialang.org/ release
|__/                   |

julia>
```

Os comandos são inseridos diretamente neste modo de comando, e as respostas são obtidas interativamente, como em

```
julia> x = 2
2
julia> y = x^3 + 6*x + 3
23
julia>
```

Quando você estiver desenvolvendo programas mais sofisticados, o papel deste modo interativo será menor. A maior parte do código será escrita em um arquivo de texto comum (desses que o Notepad do Windows lê). Por exemplo, se um código foi escrito no arquivo `program.jl`, ele pode ser executado usando

julia program.jl

A extensão "`.jl`" é normalmente usada para os programas em Julia.

Se você gosta de interfaces sofisticadas para programação (as chamadas IDEs), o Visual Studio Code tem suporte para Julia (https://github.com/julia-vscode/julia-vscode). Uma alternativa que instala o Julia mais uma série de pacotes, incluindo a IDE Juno, é a JuliaPro (https://juliacomputing.com/products/juliapro.html).

No entanto, eu não uso estas interfaces, portanto posso prestar pouca ajuda se quiserem usar. Eu uso um editor de textos (`vim`), com algumas extensões para Julia que podem ser obtidas em: https://github.com/JuliaEditorSupport/julia-vim. O uso ou não destas interfaces é uma questão de gosto, com apenas as ressalva de que elas podem requerer um computador mais potente para funcionarem de forma rápida e prática.

1.2 Os comandos essenciais

O aprendizado de qualquer linguagem de programação depende do conhecimento de sua sintaxe. Julia tem uma sintaxe bastante simples, ao menos nos comandos mais simples. Para a maior parte dos programas que vamos desenvolver, precisamos saber algumas poucas estruturas. Ao final do curso, vamos aprender como usar alguns dos milhares de pacotes que existem para resolver problemas específicos. O uso avançado da linguagem consiste nessas duas etapas: saber usar a linguagem e sua sintaxe básica, e aproveitar tudo o que já está feito.

No modo iterativo, entrar um comando imprime imediatamente uma resposta,

```
julia> x = 1
1
julia>
```

exceto se colocarmos um ponto-e-vírgula em frente ao comando:

```
julia> x = 1;
julia>
```

É possível escrever contas, definir funções, usando uma sintaxe totalmente natural, por exemplo:

```
julia> 3^2 + 1
10
julia> f(x,y) = x^2 + y^2
f (generic function with 1 method)
julia> a = 2; b = 3;
julia> f(a,b)
13
julia>
```

Note que aparece que a função tem "1 método". Se outra função tiver o mesmo nome, mas atuar sobre outro tipo de variáveis, ela sera "outro método" da mesma função. Por exemplo, se continuarmos a mesma seção com

```
julia> f(x,y,z) = x + y + z
f (generic function with 2 methods)
```

A função `f` agora tem "2 métodos". São duas funções diferentes, e uma ou outra é usada dependendo de como chamamos a função:

```
julia> f(2,3)
13
julia> f(1,1,1)
3
julia>
```

Esta capacidade de definir funções diferentes, com mesmo nome, é uma das características importantes do Julia, e se chama "despacho múltiplo". O despacho múltiplo, que permite a *especialização* das funções para cada tipo de variável, é uma das razões pelas quais os programas em Julia são eficientes. Falaremos disto em muitas ocasiões, mas podemos dar um exemplo agora:

```
julia> f(x) = 2*x
f (generic function with 1 method)

julia> f(1)
```

```
2

julia> f(1.0)
2.0
```

Note que ao chamar a função com o número inteiro `1`, recebemos como resposta o número inteiro `2`, enquanto que se chamamos a função com o número *real* `1.0` recebemos como resposta o número também *real* `2.0`. Isto acontece porque a função que foi gerada no momento da chamada é diferente em cada caso, e especializada para o *tipo de variável* (este é um termo técnico). Os detalhes da forma como a função foi codificada em um nível mais baixo podem ser vistos com o comando `@code_typed`:

```
julia> @code_typed f(1)
CodeInfo(
1 - %1 = Base.mul_int(2, x)::Int64
      return %1
) => Int64

julia> @code_typed f(1.0)
CodeInfo(
1 - %1 = Base.sitofp(Float64, 2)::Float64
    %2 = Base.mul_float(%1, x)::Float64
      return %2
) => Float64
```

Não se preocupe com os detalhes, mas note que quando a função foi chamada com o número inteiro, a multiplicação foi feita com a função `mul_int`, que é especial para esse tipo de variável. Quando foi chamada com o número real, a função chamada foi `mul_float`. Ou seja, a sua implementação única de `f(x)` foi *especializada* para cada *tipo* de variável no momento em que isso foi necessário.

Funções também podem ser definidas de forma menos compacta, como

```
julia> function f(x,y)
         z = x^2 + y^2
         return z
       end
```

Aqui não há nenhuma vantagem. Mas as funções vão se tornar operações muito complicadas, impossíveis de escrever de forma compacta. A notação acima vai ser usada muito mais que a

notação compacta, na verdade. Você já deve ter percebido que torna-se muito inconveniente escrever coisas complicadas no modo interativo. Por isso a maior parte dos programas será feita em um arquivo de texto separado, e executado independentemente.

Há 4 coisas essenciais, além das funções, que precisamos aprender para fazer a maior parte dos programas: os condicionais, os laços, os vetores, e os tipos de variáveis.

1.2.1 Condicionais

Condicionais são usados para tomar decisões. Por exemplo,

```
i = 2
if i == 2
  println("i is 2")
elseif i == 3
  println("i is 3")
else
  println("i is not 2")
end
```

Copie este código, mude o valor de `i` e execute novamente o bloco de condicionais. Note que o igual, no condicional é duplo, ==. Isto é porque ele significa algo diferente do igual no resto da linguagem. Quando escrevemos `x = 2` estamos atribuindo o valor 2 à variável `x`. Quando escrevemos `x == 2` estamos querendo saber se `x` é 2. Veja só:

```
julia> x = 2
2
julia> x == 3
false
```

Os condicionais aceitam parênteses e outros operadores além da igualdade: "e" (`&&`), "ou" (`||`), "diferente" (`!=`), e outros. Aqui usamos os mais comuns:

```
julia> x = 2 ; y = 3 ;
julia> x == 2 && y == 3
true
julia> x == 2 && y == 4
false
julia> x == 2 || y == 4
true
```

```
julia> x == 2 && y != 4
true
```

Você pode, então combinar o condicional com essas operações, por exemplo,

```
julia> x = 2 ; y = 3 ;
julia> if x == 2 && y != 4
         println("Inside!")
       end
Inside!
```

Atividades

1. Os condicionais aceitam também maior (>), menor (<), maior-ou-igual (>=) e menor-ou-igual (<=). Teste estas operações.
2. Teste todos os condicionais em textos, por exemplo "a" > "b", "abc" < "cdef", etc.

Atenção

Cuidado com a notação dos operadores "e" (&&) e "ou" (||), que são as duplas de caracteres. Os operadores simples (& e |) também funcionam mas têm significado lógico distinto (se chamam operadores bit-a-bit, e nunca vamos usá-los).

1.2.2 Laços

Laços são estruturas que permitem a repetição de instruções, ou a atualização de variáveis, muitas vezes. Há duas principais formas de fazer laços em Julia, que usaremos aqui. Vejamos exemplos:

```
for i in 1:10
  println(2*i)
end
```

A segunda forma de laços que vamos usar são do tipo "while", como no exemplo:

```
while rand(1:100) < 80
```

```
  println("less than 80")
end
```

O comando `rand(1:100)` gera um número inteiro aleatório entre 1 e 100. Teste este comando independentemente. Rode o laço várias vezes (basta apertar a flecha para cima para repetir o comando).

Há dois comandos que controlam o laço por dentro: `break` e `continue`. `break` termina o laço, e `continue` continua para a próxima iteração a partir do ponto em que se está. Por exemplo:

```
for i in 1:10
  if i == 7
    break
  end
  println(i)
end
```

1.2.3 Vetores

Vetores são listas (de números na maior parte dos casos que nos interessam, mas podem ser outras coisas). Em Julia são definidos usando colchetes,

```
julia> x = [ 2, 3 ];
julia> y = [ 3, 4 ];
julia> x + y
2-element Array{Int64,1}:
 5
 7
```

Note que ao somar `x` e `y` obtemos o vetor `[5,7]`, que foi escrito na forma de coluna. A resposta forneceu também outras informações, em particular o `Int64`, que discutiremos mais adiante. O que importa aqui é notar que fizemos a soma de dois vetores, e obtivemos um vetor. Poderíamos ter salvo o vetor resultante em uma terceira variável,

```
julia> z = x + y
```

Podemos multiplicar um vetor por um número escalar,

```
julia> z = 2*x
2-element Array{Int64,1}:
 4
 6
```

No entanto, a soma de um vetor com um escalar não é bem definida:

```
julia> z = x + 1
ERROR: MethodError: no method matching +(::Array{Int64,1}, ::Int64)
```

Se o que queremos é somar o número 1 a *cada elemento* de x, temos que usar um laço,

```
julia> x = [ 2, 3 ]
julia> z = similar(x);
julia> for i in 1:2
         z[i] = x[i] + 1
       end
julia> z
2-element Array{Int64,1}:
 3
 4
```

Note que, neste caso, temos que definir o que é z *antes*, porque caso contrário teremos um erro quando tentarmos ocupar qualquer emento, z[i], desse vetor, dentro do laço. A definição de z se faz, aqui, criando um vetor similar ao vetor x. Similar significa duas coisas: A) tem a mesma dimensão, no caso 2 e B) é do mesmo tipo de variável, no caso de números inteiros. O conteúdo desse vetor não vai ter nenhum sentido quando ele for criado, discutiremos isso mais adiante, na próxima seção. Vetores podem ser definidos de várias formas. Veremos com mais detalhes adiante, junto com o entendimento do que são os tipos de variáveis em um computador.

Existem formas compactas de fazer a mesma coisa, que "escondem" o fato de que z está sendo gerado antes de ser preenchido. Por exemplo:

```
julia> z = x .+ 1
2-element Array{Int64,1}:
```

```
 3
 4
```

O que mudou aqui foi que adicionamos um ponto (`.`) antes do (`+`). A adição do ponto significa que a operação será feita para todos os elementos do vetor em questão. É uma sintaxe muito poderosa, que usaremos com alguma frequência. Por exemplo, se definimos qualquer função, podemos usar essa notação para calcular um vetor com o resultado dessa função aplicada sobre todos os elementos de um vetor de entrada:

```
julia> f(x) = x^2 + 2
f (generic function with 1 method)

julia> f.(x)
2-element Array{Int64,1}:
  6
 11
```

No entanto, o que quero chamar à atenção aqui é que estas notações compactas não são obrigatórias em Julia para a obtenção de códigos eficientes. De fato, usar o loop explícito ou estas notações convenientes é exatamente a mesma coisa (usaremos o pacote `BenchmarkTools` para medir os tempos):

```
julia> using BenchmarkTools

julia> @btime z = f.($x)
  26.906 ns (1 allocation: 96 bytes)

julia> @btime begin
                 z = similar($x)
                 for i in 1:length($x)
                   z[i] = f($x[i])
                 end
               end
  25.745 ns (1 allocation: 96 bytes)
```

Se você tem experiência em outras linguagens como Python, deve ter notado que o estamos fazendo aqui se assemelha à função `map`. Em Julia também temos esta função. A diferença é que usá-la aqui não é fundamental, porque ela não vai ser especialmente mais rápida que qualquer um dos códigos acima:

```
julia> @btime map($f,$x)
  28.941 ns (1 allocation: 96 bytes)
```

Nota

> Isto não significa, obviamente, que função `map` em Julia é ineficiente. Pelo contrário, significa que os loops explícitos o são, o que não é o caso de linguagens interpretadas como Python, R, TCL, etc. Em Julia você não é obrigado a usar funções prontas para obter um código rápido, e é portanto mais simples adaptar o código ao seu problema.

Algumas funções com vetores são muito usadas:

```
x = zeros(3) # Gera um vetor de 3 componentes, de números reais
y = copy(x) # Cria um novo vetor com o mesmo conteúdo de x
y = similar(x) # Cria um novo vetor do mesmo tipo de x
x = zeros(3,3) # Cria uma matriz de 3x3 de números reais
length(x) # Retorna o número de elementos de x
```

1.2.4 Tipos de variáveis

Os números, e qualquer outra coisa, em um computador, têm tamanhos. Isto é, ocupam um determinado espaço na memória. O seu computador tem uma quantidade de memória determinada e finita (8Gb, 2Tb, etc.). Isto significa que nela cabe uma quantidade definida de números. Mas nem todo número ocupa o mesmo espaço. A quantidade de espaço que ocupa cada número tem relação com a quantidade de informação que pode ser guardada nesse tipo de número.

Números no computador podem ser "reais", inteiros, ou complexos. Não usaremos números complexos, portanto vamos discutir os números reais e inteiros aqui. Um número real pode ter infinitas casas decimais, como π. É impossível guardar o valor de π, portanto. Quando definimos π no computador, ele está salvo com uma determinada precisão. Essa precisão depende de que tipo de número usamos para salvar π. Se o número é inteiro, teremos $\pi = 3$. Se o número é real, a precisão depende de quanto espaço na memória queremos usar para guardar o número.

Há dois principais tipos de números reais nos computadores, os de 32 bits e os de 64 bits. Hoje em dia o padrão é trabalhar com números de 64 bits. Por isso, ao escrever

```
julia> x = zeros(2)
```

```
2-element Array{Float64,1}:
 0.0
 0.0
```

O interpretador diz que temos um vetor de 2 elementos de números do tipo `Float64`. `Float` é uma nomenclatura para número real, `64` é 64 bits.

Estes números tem uma precisão, que implica no número de algarismos que eles podem salvar. Veja, portanto, estes exemplos:

```
julia> 1.e16 + 1.
1.0e16
julia> 1.e15 + 1.
1.000000000000001e15
```

A soma de 1×10^{16} com 1 dá 1×10^{16}. Isto porque não há algarismos suficientes em um número real de 64 bits para armazenar 16 ordens de grandeza de diferença. No entanto, a conta é correta para uma grandeza a menos.

Nota

> Em Julia, você pode definir um número real de 32 bits usando a notação, por exemplo, `1.f0`, `1.f10`, etc.

É fundamental conhecer a representação dos números nos computadores. Se o seu programa lidar com números de ordens de grandeza muito diferentes, ou números muito grandes, próximos dos limites das representações, o programa tem que ser feito com cuidados especiais.

Além disso, Julia faz uma análise do código e, se conseguir determinar o tipo das variáveis ao longo de todo o código, é capaz de gerar um programa tão rápido como C ou Fortran. Desta forma, é interessante fazer os códigos de forma que os tipos de variáveis não mudem. Uma forma de colaborar com isso e, inclusive, ter mais controle sobre as funções, é *declarar* os tipos das variáveis. Isso faz sentido nas funções, que serão muito mais eficientes se trabalharem com tipos imutáveis: exemplo,

```
function f( x :: Float64 )
  return x^2 + x - 1.
end
```

A função está definida para receber exclusivamente um número real de 64 bits. É possível, como mencionado antes, definir outra função com o mesmo nome que recebe outro tipo de

variável:

```
function f( x :: Int64 )
  return x + 2
end
```

As definições dos tipos de variáveis, neste caso, não são necessárias porque, como vimos, o interpretador de Julia vai descobrir que tipo de função é necessária para uma função simples como esta e especializar a função adequadamente de forma automática. No entanto, há situações em que o compilador não consegue fazer isto é, nesses casos, informar adequadamente que tipo de variável está sendo usada é fundamental. Isso é particularmente relevante quando definirmos variáveis com tipos novos.

Uma das principais funções das declaração está na criação de vetores. Isto será feito o tempo todo. Criar um vetor de números inteiros é diferente de criar um vetor de números reais, ou de caracteres. Vimos que há várias maneiras de definir vetores, e todas elas permitem a especificação do tipo de variável que o vetor contém. Por exemplo,

```
x = Vector{Int64}(undef,2)
```

Este comando cria em `x` um vetor de números inteiros de 64 bits, de duas posições. O `undef` indica que não vamos nos preocupar agora com o conteúdo do vetor. Os números que estarão no vetor serão qualquer coisa, por exemplo,

```
julia> x = Vector{Int64}(undef,2)
2-element Array{Int64,1}:
 8583523454
          2
```

Esta é a forma mais rápida de criar um vetor. Podemos criar o vetor com zeros em todas as posições,

```
julia> x = zeros(Int64,2)
2-element Array{Int64,1}:
 0
 0
```

Esta opção tem uma notação mais compacta, mas é um pouco menos eficiente, caso não seja necessário zerar o vetor desde o início. "Criar" o vetor significa reservar, na memória RAM, o espaço em que vetor ficará alocado. Zerar todas as posições significa operar sobre essas posições, por isso toma mais tempo. A diferença de tempo pode ou não ser relevante no seu programa, dependendo de quantas vezes é feito. De todos modos, é bom saber que a criação do vetor nulo acima equivale ao seguinte código:

```
julia> x = Vector{Int64}(undef,2)
julia> for i in 1:2
         x[i] = 0
       end
```

As funções, então, podem receber vetores como argumentos, e devolver vetores como resultado:

```
julia> f( x :: Vector{Float64} ) = 2*x
f (generic function with 1 method)

julia> x = [ 2. , 2. ]
2-element Array{Float64,1}:
 2.0
 2.0

julia> f(x)
2-element Array{Float64,1}:
 4.0
 4.0
```

1.3 Estrutura básica de um programa

Antes de nada, tudo o que compuser os programas será escrito em inglês. Há duas razões para isto. A mais direta, e banal, é que no inglês não há acentos, e acentos são uma fonte de problemas em um programa. A segunda é que é um hábito saudável se acostumar a programar tudo pensando que, um dia, o programa será distribuído para outras pessoas, e o inglês é a língua para isso hoje em dia.

Como mencionado, há duas formas de programar em Julia: a forma interativa, e os códigos independentes. A forma interativa serve para testar coisas e obter resultados rápidos. Os programas escritos em arquivos são, por outro lado, onde a maior parte do código vai ser

escrito.

Se uma função foi, então, definida no arquivo `func.jl`, ela pode ser usada carregando este arquivo na parte interativa, usando o comando `include`:

```
julia> include("./func.jl")
```

Atividades

3. Crie um arquivo que contenha uma função $f(x)$. Carregue o arquivo como mencionado acima e calcule o valor da função em um ponto, no modo interativo.

Você pode incluir a execução da função no próprio arquivo, e executar o arquivo sem entrar na parte interativa. Por exemplo, o arquivo poderia ser

```
function test(x)
  y = x + 2
  return y
end
test(4)
```

Um arquivo, digamos `test.jl`, pode ser executado com `julia test.jl`.

Desta forma, a estrutura mínima de um programa dos que vamos fazer consiste na definição de uma função e sua execução:

```
function func()
  # This is a commentary
end
func()
```

Todas as linhas que começam com o numeral, "#", são ignoradas, sendo chamadas de "comentários". Os comentários ajudam, em programas complexos, a entender o que está sendo feito.

Os espaços de indentação não são obrigatórios em Julia (só em Python são), mas são fundamentais para que o código fique legível. Fica claro o que faz parte de cada função, de cada laço, de cada condicional, etc.

1.4 Funções intrínsecas e pacotes

Há muitas funções que, por serem usadas com frequência, já estão implementadas no Julia. Outras dependem de pacotes, que podem ser instalados e carregados facilmente. Algumas funções intrínsecas que usaremos são:

```
sin(x) # Sin of x
sum(x) # Sum of elements of vector x
abs(x) # Absolute value of x
length(x) # Number of elements of vector x
exp(x) # Exponential of x
```

Outras funções são carregadas de pacotes. Os pacotes podem conter funções muito sofisticadas, desde machine learning até geração de gráficos. Nós vamos usar desde o princípio o pacote de fazer gráficos. É necessário instalar este pacote, usando:

```
julia> using Pkg
julia> Pkg.add("Plots")
```

Ou, usando uma notação mais compacta, use o colchete (`]`):

```
julia> ]
(@v1.5) pkg> add Plots
```

Isto vai baixar o pacote e instalar no seu computador. Para fazer um gráfico, fazemos, por exemplo,

```
julia> using Plots
julia> x = [ 1, 2, 3 ]
julia> y = [ 1, 4, 9 ]
julia> plot(x,y)
```

O comando `using Plots` carrega o pacote. Na primeira vez que você usar este comando, o pacote será compilado, e vai demorar um pouco. Nas vezes seguintes será mais rápido. O pacote `Plots` gera gráficos de todo tipo, com muitas opções. Aprenderemos algumas delas quando for útil.

1.5 Detalhes técnicos da linguagem

Aqui vamos descrever alguns detalhes técnicos da linguagem Julia que precisamos mencionar antes de fazer coisas mais sofisticadas. Provavelmente você só vai entender a importância e a razão destes detalhes mais para frente, mas é melhor apresentar estas questões antes de que fiquemos presos em erros que não entendemos em programas mais complicados.

1.5.1 Cópia ou passagem por referência: escalares e vetores

Quando uma variável é enviada para uma função, como `a` em

```
julia> f(x) = 2*x
julia> a = 2
julia> f(a)
```

há duas possibilidades: `a` pode ser passado copiando seu valor, ou por referência. A forma natural de pensar é que a variável é copiada, de forma que a função não modifica o valor de `a`. Por exemplo, quando calculamos uma função qualquer $\sin(x)$, $\cos(x)$, nunca imaginamos que o valor de x pode mudar. Ou seja, se alguma conta é feita dentro da função com x, ela não deve mudar o valor de x. Por padrão, em Julia, escalares (números) são passados por cópia. Comprovamos isto no exemplo:

```
julia> function f(x)
         x = x + 1
         return x
       end
julia> x = 2;
julia> f(x)
3
julia> x
2
```

Note que a função `f(x)` recebe `x`, aparentemente muda o valor de `x` e retorna este valor. No entanto, o valor de `x` fora não foi modificado. Portanto, o `x` dentro da função não é a mesma variável que o `x` que a função recebeu. O valor foi copiado na passagem para a função em uma nova variável (ocupando outro lugar na memória RAM do seu computador).

Façamos a mesma coisa, agora, com um vetor:

```
julia> function f(x :: Vector)
```

```
          x[1] = x[1] + 1
          return x
        end
```

A função agora recebe um vetor, e modifica a primeira componente desse vetor, somando 1. Vamos ver o que acontece quando usamos esta função, da mesma forma que no caso anterior:

```
julia> x = [ 1, 1 ]
2-element Array{Int64,1}:
 1
 1

julia> f(x)
2-element Array{Int64,1}:
 2
 1

julia> x
2-element Array{Int64,1}:
 2
 1
```

Note que, agora, o vetor `x` de fora mudou! Ou seja, quando modificamos a primeira componente do vetor dentro da função, mudamos o mesmo vetor que foi enviado de fora. Isto é diferente do caso anterior. Agora o vetor foi passado para a função *por referência*. Isto é, o que foi enviado para a função não foi uma cópia do vetor, mas a sua posição na memória RAM do computador. Ao modificar o vetor dentro da função, modificamos o vetor original. Se não quisermos que isso aconteça, temos que copiar explicitamente o vetor, dentro da função, usando, por exemplo,

```
julia> function f(x :: Vector)
          y = copy(x)
          y[1] = y[1] + 1
          return y
        end
```

Atividades

4. Verifique que, nesta última versão, o resultado da função é o mesmo, mas o vetor externo a ela não foi modificado.

A maior parte das linguagens se comporta de forma similar ao Julia neste aspecto. A razão para isso é que os vetores (e matrizes) podem ser enormes. Pode acontecer de um vetor no seu programa usar quase toda a memória disponível no seu computador. Portanto, copiar os vetores para evitar que a função os modifique seria muito ruim. Assim, por padrão, as funções modificam os vetores originais, e cópias são feitas só se o programador quer explicitamente isto. Copiar escalares, como antes, raramente é um problema, de forma que o padrão é copiá-los.

1.5.2 Definição ou assignação de vetores

Essa diferença no tratamento de vetores tem uma consequência importante para a programação, que deve ser considerada com cuidado quando acontecer. Quando escrevemos

```
julia> x = [ 1, 1 ]
julia> y = x
```

o vetor `y` não é uma cópia do vetor `x`, ele é o mesmo vetor (a igualdade foi apenas uma indicação por referência). Isso tem uma consequência que pode levar a erros importantes nos programas, que é que modificar os elementos de `y` modifica `x`. Continuando o código anterior:

```
julia> y[1] = 2
julia> x
2-element Array{Int64,1}:
 2
 1
```

O primeiro elemento de `x` mudou! Notem como isso pode gerar confusão[2]. Se queremos criar um vetor `y` para ser modificado sem modificar `x`, temos que copiar explicitamente:

```
julia> x = [ 1, 1 ]
```

[2]Se achou isto horrível, concordo com você. Mas saiba que em Python é a mesma coisa. Por essas e outras que Fortran continua existindo. Mas é o preço que se paga por ter uma linguagem que pode ser usada interativamente.

```
julia> y = copy(x)
```

Atividades

5. Mostre que, com a cópia explícita, a mudança das componentes de y não modifica as componentes de x.

Esta sutileza pode gerar outra confusão e ser uma fonte de erros: Suponha que criamos um vetor x, em seguida *outro* vetor y, e agora queremos que y tenha a mesma informação que x. Poderíamos tentar o seguinte:

```
julia> y = [ 2, 2 ]
julia> x = [ 1, 2 ]
julia> y = x
```

A terceira linha, y = x, não vai fazer com que y tenha a mesma informação que x, ela vai fazer y *ser o mesmo vetor que* x, caindo no problema anterior, de que modificar y vai modificar x também.

No entanto, se copiarmos as componentes uma a uma, isso não acontece, porque as componentes são escalares, e o valor é copiado de uma variável na outra:

```
julia> y = [ 2, 2 ]
julia> x = [ 1, 1 ]
julia> y[1] = x[1]; y[2] = x[2];
julia> y[1] = 3
julia> x
2-element Array{Int64,1}:
 1
 1
```

Desta forma, se queremos copiar o conteúdo de um vetor em outro vetor, a temos que usar um laço que copie componente a componente:

```
julia> x = [ 1, 2, 3, 4, 5 ]
julia> y = Vector{Int64}(undef,5)
julia> for i in 1:5
         y[i] = x[i]
       end
```

Poderíamos ter usado a notação compacta

```
julia> @. y = x
```

para o laço, alternativamente.

Atividades

6. Verifique que usando o laço o vetor `x` não muda se os valores da componentes de `y` forem modificados.

1.5.3 Variáveis globais

Quando definimos uma variável no REPL, ou fora de qualquer função, definimos uma variável global. Elas podem ser muito úteis, por exemplo:

```
julia> pi = 3.14
3.14

julia> area(r) = pi * r^2
area (generic function with 1 method)
```

O cálculo da área do círculo requer o sudo do valor de π, que foi definido globalmente. No entanto, nosso programa poderia continuar assim:

```
julia> pi = "example of irrational number"
```

onde variamos completamente o tipo de variável que é π. Aqui temos um caso típico de "instabilidade de tipos". A função area não tem como saber de antemão que tipo de variável é π e, então, ela tem que assumir que ele pode ser qualquer coisa. Nenhuma especialização é possível e o código fica muito lento. Vejamos:

```
julia> pi = 3.14
3.14

julia> @btime area(5)
  19.071 ns (1 allocation: 16 bytes)
78.5
```

```
julia> @code_typed area(5)
CodeInfo(
1 - %1 = Base.mul_int(r, r)::Int64
    %2 = (Main.pi * %1)::Any
         return %2
) => Any
```

Note que existe um `Any` no código da função obtido com `@code_typed`, que indica que a variável π não tem tipo definido.

Vamos agora garantir o tipo de variável que é π, transformando-a em uma constante dentro da função:

```
julia> area(r,pi) = pi * r^2

julia> @btime area(5,$pi)
  0.024 ns (0 allocations: 0 bytes)

julia> @code_typed area(5,$pi)
CodeInfo(
1 - %1 = Base.mul_int(r, r)::Int64
    %2 = Base.sitofp(Float64, %1)::Float64
    %3 = Base.mul_float(pi, %2)::Float64
         return %3
) => Float64
```

O que fizemos foi, em vez de simplesmente usar a variável π definida globalmente, passamos ela como argumento para a função que calcula a área. Agora, a função é especializada para o tipo de variável que π é ao entrar (um número real). Isto não podia ser feito antes porque a o π que estava sendo acessado era global. Agora, o π dentro da função não pode mudar de tipo porque, justamente, seu valor foi copiado na chamada da função. Mesmo que mudemos o o valor de π fora da função, a função não vai ser alterada.

Note as seguintes diferenças: Primeiro, o código ficou quase 800 vezes mais rápido, e não faz nenhuma alocação de memória. Segundo, não temos mais o tipo `Any` mencionado em nenhum lugar no código de baixo nível. Só temos números.

O mesmo resultado, em termos de eficiência, pode ser obtido transformando π em uma constante:

```
julia> const pi = 3.14
```

```
3.14

julia> @btime area(5)
  0.025 ns (0 allocations: 0 bytes)
```

Esta pode ser uma opção interessante quando os valores são efetivamente constantes (como o valor de π). No entanto, neste caso o "nome" `pi` fica reservado:

```
julia> pi = "example of irrational number"
ERROR: invalid redefinition of constant pi
```

Portanto, exceto nos casos em que os números (ou outras variáveis) não mudam mesmo *nunca*, sempre devemos passar elas como parâmetros (ou argumentos) para as funções. Outra discussão sobre os efeitos na eficiência do código do uso de variáveis globais pode ser lida aqui [LINK].

2 Simulação de partículas

2.1 Métodos ingênuos para o cálculo da energia potencial

Para entender como algoritmos podem acelerar um programa, primeiro precisamos conhecer as razões pelas quais um programa é lento. Os métodos que vamos estudar envolvem conceitos gerais aplicáveis a muitos problemas diferentes, mas vamos focar aqui em sua aplicação na simulação da dinâmica de sistemas de muitas partículas. O primeiro que vamos fazer é programar o método *ingênuo* de simulação, que vai ser simples e, também, lento.

2.1.1 Potencial de interação

Vamos usar aqui um potencial de Lennard-Jones. Potenciais de Lennard-Jones são os potenciais mais usados na representação de interações dispersivas (forças de Van der Waals) em simulações de sistemas moleculares. o potencial de Lennard-Jones tem a forma:

$$U(r, \varepsilon, \sigma) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]$$

onde r é a distância entre elas, e ε e σ são duas constantes relacionadas com a profundidade do mínimo de energia e com o raio de Van der Waals das partículas, respectivamente.

Atividades

7. Defina uma função que calcule o potencial U dados r, ε e σ.
8. Crie um vetor de distâncias, `R = [r1,...,rN]` e calcule a energia potencial para cada distância, dados ε e σ. Faça um gráfico.
9. Entenda o efeito da variação de ε fazendo novos gráficos com ε variável.
10. Faça o mesmo para σ.

Nota

Os gráficos podem ser feitos com o pacote `Plots`, que usaremos bastante. Um exemplo simples de seu uso é:

```
using Plots
x = rand(10); y = rand(10)
plot(x,y,xlabel="x",ylabel="y",label="dados y")
# adicionar uma nova curva no mesmo gráfico (note o !):
z = rand(10)
plot!(x,z,label="dados z")
# salvar em PDF
savefig("plot.pdf")
```

Agora vamos calcular o potencial de interação para um conjunto de 10 pontos no plano (todas as nossas simulações vão ser bidimensionais). Criemos posições aleatórias para 10 pontos, com coordenadas entre 0 e 10:

```
p = [ 10*rand(2) for i in 1:10 ]
```

(o gerador de números aleatórios está gerando, aqui um vetor de 2 números aleatórios, que serão as coordenadas de cada um dos 10 átomos). O resultado `p` é um vetor de 10 posições, onde em cada posição temos um vetor de 2 posições, que correspondem às coordenadas de cada partícula.

Então, o que queremos é fazer uma função que calcule o potencial de interação total do sistema formado por essas partículas.

Atividades

11. Escolha valores para ε e σ tais que o mínimo de energia esteja próximo de 1.0, e some, para todos os pares de partículas (sem repetições! - quantos pares são?) o potencial U calculado. Defina uma função para isto.

Para medir a velocidade da implementação do cálculo da energia potencial, vamos usar o pacote `BenchmarkTools`. Assumindo que a função que calcula os potenciais se chama `potential`, e recebe como parâmetros ε, σ e o vetor de posições p (esta é uma alternativa de como programar esta função, a sua pode estar diferente), faremos o seguinte:

```
] add BenchmarkTools
using BenchmarkTools
@btime potential($eps,$sig,$p)
```

É importante colocar o sifrão $ em cada variável, pelo seguinte: o comando `@btime` vai executar várias vezes a função, para medir bem o tempo. Para que os dados sejam sempre os mesmos, faz-se uma cópia deles na entrada da função em cada execução, e o sifrão tem esse significado.

O resultado vai se assemelhar a algo como:

```
julia> @btime potential($eps,$sig,$p)
  32.128 ns (1 allocation: 16 bytes)
120.90322388919448
```

Onde temos o tempo, a quantidade de memória alocada pela função, e o resultado final. Nos próximo exercício, você vai tentar acelerar sua função. Uma ferramenta útil neste processo, é a `@test`, que permitirá testar se suas modificações não introduziram erros no código. O fluxo de trabalho fica assim:

```
using BenchmarkTools
using Test
function f1(...) # primeira tentativa
   ...
end
function f2(...) # segunda tentativa
   ...
end
@test isapprox(f1(...),f2(...))
```

```
@btime f1(...)
@btime f2(...)
```

Caso as modificações feitas em `f2` alterem o resultado da função, o teste vai retornar um erro. Se estiver tudo certo, segue-se para a avaliação de desempenho com `@btime`.

Atividades

12. Procure modificar sua função para acelerar sua execução. Veja com cuidado que contas são feitas. É possível evitar a repetição de contas?

13. Sua função, em princípio, não deve alocar quase nada de memória, já que ela faz uma conta com dados de entrada e devolve um único número. Se sua função está alocando memória, procure modificá-la para evitar isso.

14. Por quê temos que usar $aproximadamente$ para testar se o resultado está consistente, e não simplesmente $igual$?

Dica

Julia aceita caracteres `Unicode`. Se você tem um editor configurado adequadamente, poderá escrever, por exemplo `\approx` e apertar `TAB`, para obter o símbolo ≈. Com isso, o teste pode ser escrito como

```
@test f1(...) ≈ f2(...) ≈ f3(...)
```

Como estamos começando, colocaremos aqui um exemplo do que poderia ser esperado para as atividades acima. Note que evito aqui calcular as potências de ε e σ dentro do loop, já que o resultado é o mesmo para todos os pares de pontos. O loop é feito sem nenhuma repetição de pares, e evitando o cálculo da interação dos pontos com eles mesmos. Dentro da função que calcula o potencial para um par de pontos, também tomamos o cuidado de não calcular as potencias grandes mais de uma vez (quanto isto acrescenta em desempenho é variável, e depende de quais otimizações o compilador faz para você. Algumas destas mudanças seriam automaticamente feitas pelo compilador e, então, defini-las explicitamente pode não fazer diferença em casos particulares).

```
using BenchmarkTools

function pot(x,y,eps4,sig6,sig12)
```

```
  r2 = (x[1]-y[1])^2 + (x[2]-y[2])^2
  r6 = r2^3
  r12 = r6^2
  return eps4*(sig12/r12 - sig6/r6)
end

function total_pot(p,eps,sig)
  eps4 = 4*eps
  sig6 = sig^6
  sig12 = sig6^2
  total_pot = 0.
  for i in 1:length(p)-1
    for j in i+1:length(p)
      total_pot = total_pot + pot(p[i],p[j],eps4,sig6,sig12)
    end
  end
  return total_pot
end

eps = 4.
sig = 1.
p = [ 10*rand(2) for i in 1:10 ]
@btime total_pot(p,eps,sig)
```

[Clique aqui para baixar o código]

2.1.2 Condições periódicas de contorno

Este código gera as posições de 100 partículas dentro de um quadrado de lado 10, e faz o gráfico correspondente, mostrado na Figura 1:

```
using Plots
p = [ 10*rand(2) for i in 1:100 ]
x = [ p[i][1] for i in 1:100 ]
y = [ p[i][2] for i in 1:100 ]
scatter(x,y,
        xlabel="x",ylabel="y",label="",
        xlim=[-15,25],ylim=[-15,25],
        size=(400,400),framestyle=:box)
savefig("pbc1.pdf")
```

[Clique aqui para baixar o código]

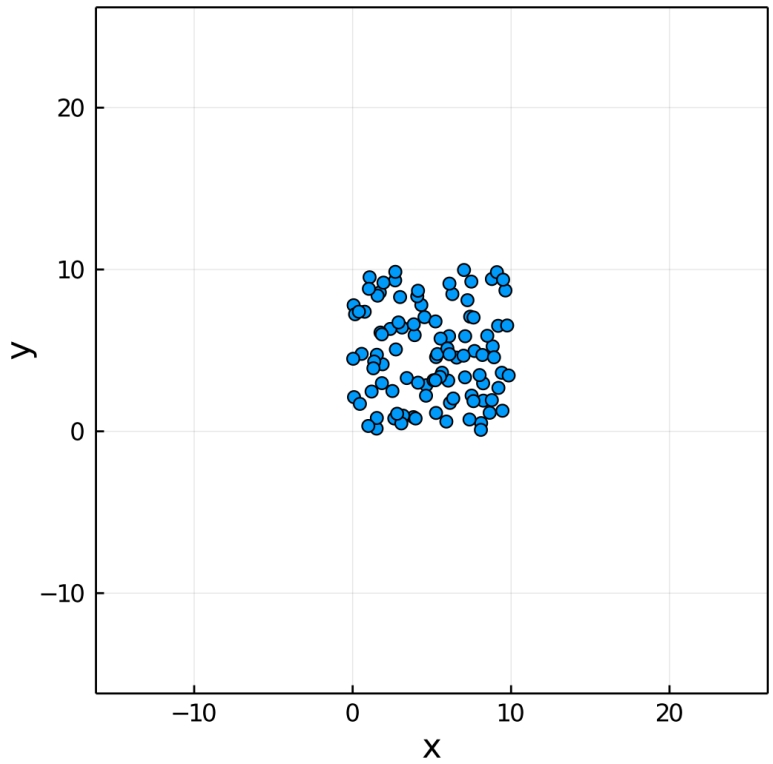

Figura 1: Posições, no plano, de pontos gerados aleatoriamente, com coordenadas entre 0 e 10. Os pontos nas extremidades não são equivalentes aos pontos no centro da distribuição.

O gráfico criado mostra, propositalmente, as regiões vazias em torno dos pontos. Se estas são as posições de partículas em um sistema molecular, é claro que as fronteiras são muito artificiais. Ou este sistema é apenas um pequeno aglomerado de partículas, ou as fronteiras do sistema estão muito mal representadas, já que as partículas próximas às fronteiras não interagem da mesma forma com o restante das partículas que as partículas no centro.

Para que um sistema como este possa representar melhor um sistema molecular (um líquido, ou um gás, por exemplo), que do ponto de vista das interações microscópicas é essencialmente infinito, introduzimos condições periódicas de contorno. As posições das partículas são repetidas, infinitamente, em todas as direções, como mostra a Figura 2. Em torno de cada ponto, agora, podemos encontrar um sistema essencialmente homogêneo. A "cópia" de cada outro ponto que está na mais próxima de cada ponto é chamada de *imagem mínima*. As interações de curto alcance são calculadas entre cada partícula e a imagem mínima das outras partículas em relação a ela. Nesta figura, mostramos todos os pontos que estão na imagem periódica mínima de um dos pontos, em vermelho.

Assim, todas as partículas estão inseridas em um sistema infinito, que pode representar muito melhor um sistema em fase condensada.

Para calcular as imagens mínimas, você vai precisar usar uma função que calcula o *resto* da

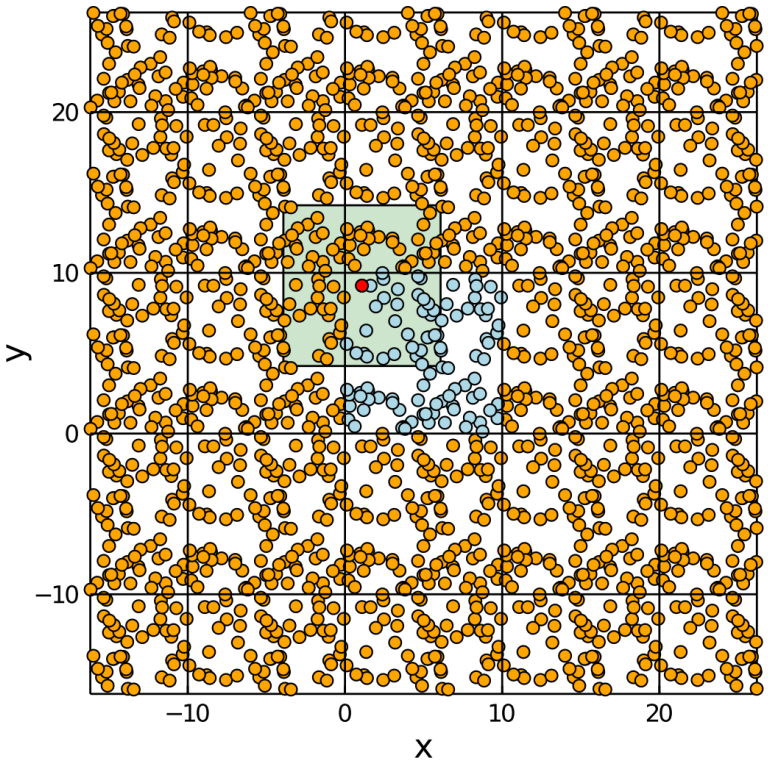

Figura 2: Representação pictórica das condições periódicas de contorno, gerada com o código `[pbc2.jl]` . As posições das partículas são repetidas periodicamente, deslocadas por distâncias iguais aos comprimentos dos lados da cela unitária.

divisão de dois números. Essa função se chama `mod()`, ou simplesmente `%`, tal que `5%2=1`, ou `mod(5,2)=1`.

Atividades

15. Faça um programa que gere dois pontos bidimensionais com coordenadas que variam entre 0 e 100. Faça, em seguida, um programa que calcula a distância entre os dois pontos de acordo com sua imagem mínima, assumindo que o sistema periódico tem lados de 10. em ambas as direções.

16. Converta este programa em uma função, que recebe as coordenadas de dois pontos, e retorna a distância de imagem mínima entre eles.

17. Gere, agora, 100 pontos com coordenadas entre 0 e 100. Faça um programa que calcule a máxima distância entre os pontos, de acordo com suas imagens mínimas, em um sistema periódico quadrado de lado 10. Verifique que o resultado faz sentido.

18. Calcule, agora, o potencial de interação dos 100 pontos usando sua função de energia potencial e as distâncias de imagem mínima.

2.1.3 Raio de corte (*cutoff*)

Usemos agora $\sigma = 0.5$ e $\varepsilon = 2$. A função potencial, neste caso, possui o perfil mostrado na Figura 3.

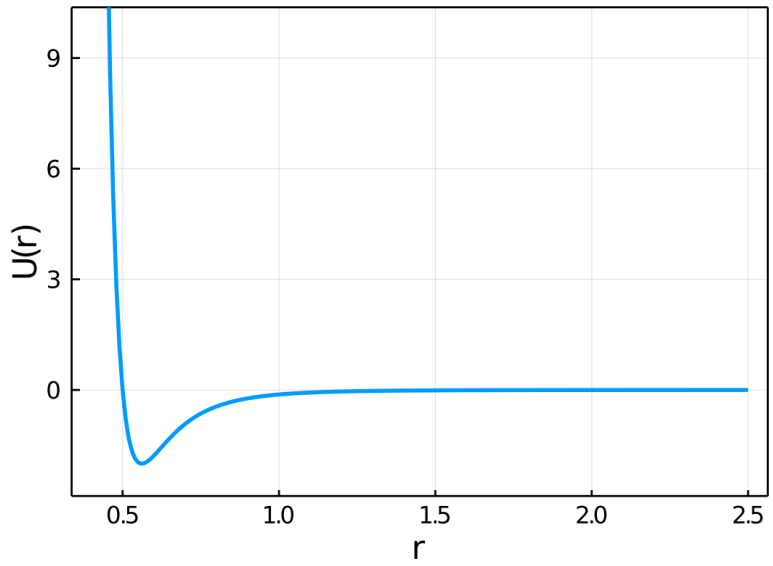

Figura 3: Potencial de Lennard-Jones entre um par de partículas, usando $\varepsilon = 2.0$ e $\sigma = 0.5$, obtido com o código `[cutoff1.jl]` . Note que após $r = 2.0$ o potencial vale quase que exatamente zero.

O potencial vale quase exatamente zero a partir de $r = 2.0$, ou algum valor próximo. Desta forma, não é necessário calcular a função de energia potencial se a distância entre as partículas é maior que 2.0. Do ponto de vista de simulações de partículas, uma das estratégias mais importantes para acelerar cálculos é evitar calcular interações desnecessariamente.

No algoritmo ingênuo, o cálculo das interações entre todos os pares de N partículas envolve a soma dos potenciais de interações de $N(N-1)/2$ interações. Podemos tentar melhorar este desempenho, evitando calcular o potencial de Lennard-Jones sempre que a distância entre as partículas for maior que esse radio de corte.

Atividades

19. Faça um desenho de uma matriz com toos os pares de N partículas, indique todos os pares não-repetidos e os pares de partículas iguais, e mostre que o número de interações resultantes é $N(N-1)/2$.

20. Faça um programa que calcule a energia potencial total de um conjunto de 100 pontos, usando $\varepsilon = 2.0$ e $\sigma = 0.5$, com pontos gerados no intervalo $[0, 10]$.

21. Crie uma nova função, na qual a distância é calculada antes de que o potencial seja avaliado. Crie uma nova função que calcula a energia potencial total, mas que

não calcule o potencial se a distância entre os pontos é maior que 2.0. Verifique que os resultados são iguais com boa precisão, e meça o tempo de execução.

O resultado do programa do Atividade 21 deve ter mostrado que evitar o cálculo das interações desnecessariamente leva a um ganho considerável de tempo. A extensão do ganho vai ser dependente dos detalhes da implementação. No meu caso (`[cutoff2.jl]`), a aceleração do cálculo da função foi de aproximadamente ≈ 2.5 vezes.

Atividades

22. Estude o código `[cutoff2.jl]` . Compare com sua implementação do mesmo problema, que é o resultado do Atividade 21. Note a introdução de algumas notações úteis, como o número de elementos do vetor `x` como sendo `length(x)`.

23. Faça todas as modificações no seu programa da Atividade 21 (ou no código `[cutoff2.jl]`) que julgar pertinentes e que evitem fazer contas desnecessárias. Verifique quanto que cada uma das modificações melhora o tempo de execução.

O cálculo das interações está sendo feito, ainda, de forma muito pouco eficiente. Da forma como está proposta a solução da Atividade 21, ainda estamos calculando todas as distâncias entre os pontos. O que queremos é evitar completamente o cálculo mesmo das distâncias entre os pontos. Mais do que isso, queremos evitar mesmo fazer um *loop* sobre todos os pares de átomos. Para isso, usaremos o método das *células ligadas*.

2.2 Método das células ligadas

O método das células ligadas permite evitar completamente o cálculo de grande parte das interações e, mais que isso, evitar inclusive percorrer os pares de partículas que estão distantes umas das outras.

2.2.1 Conceitos gerais

A ideia central do método de células ligadas é a classificação das partículas em regiões do espaço *antes* do cálculo das distâncias. As distâncias são calculadas apenas para partículas em regiões vizinhas, evitando os pares de partículas em regiões que distam mais do que o raio de corte. A Figura 4 ilustra a partição do espaço em regiões em um sistema com raio de corte de 2 (unidades de distância, u. a.).

Na Figura 4 o espaço está particionado em quadrados de lado igual ao raio de corte. Portanto, as partículas que estão na célula central (em verde), com certeza estão a uma distância maior que o radio de corte das partículas que estão fora da região amarela. As interações entre estas partículas não devem ser calculadas.

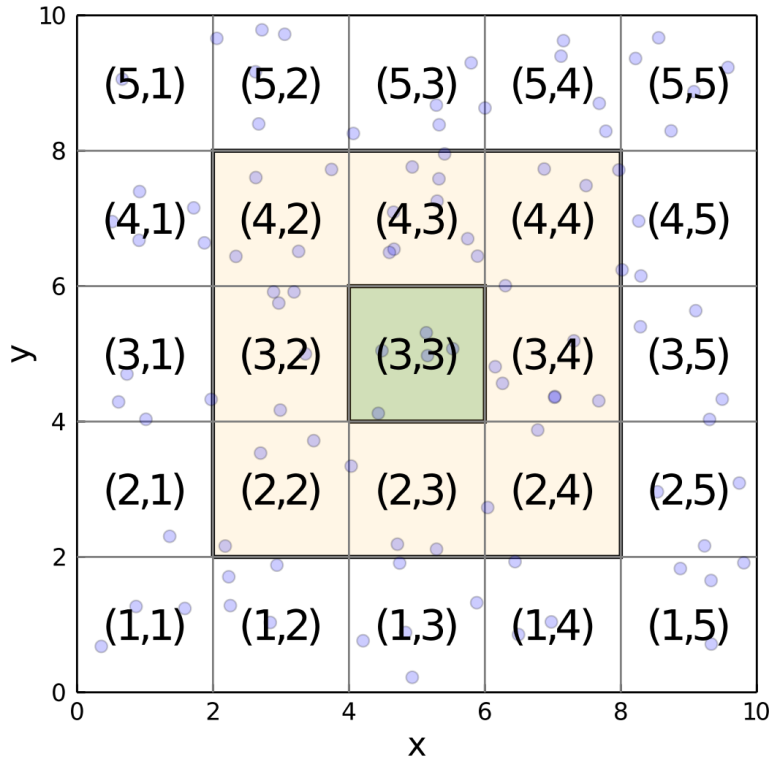

Figura 4: Representação do método das células ligadas. As partículas na célula central, $(3, 3)$, estão necessariamente a uma distância maior que o raio de corte (aqui 2.0 u. d.) das partículas fora da região amarela. Figura gerada com `[lcells.jl]` .

A classificação das partículas em regiões do espaço permite o cálculo acelerado das interações porque pode ser feita em tempo linear no número de partículas. Dadas as posições `[x,y]` de uma das partículas, as coordenadas da célula em que esta partícula se encontra podem ser calculadas usando

```
ijcell = [ trunc(Int64,x/cutoff) + 1, trunc(Int64,y/cutoff) + 1 ]
```

Atividades

24. Estude e entenda o que faz a função `trunc`.

25. Gere coordendadas de 10 partículas como indicado em [LINK]. Calcule as coordenadas da célula de cada partícula, notando que foram geradas em uma caixa de lado 12, e usando raio de corte 3.0. Faça um laço para isso, e entenda que o custo computacional desta operação é proporcional ao número de partículas.

2.2.2 Implementação

O método consiste, então, em anotar quais são as partículas que correspondem a cada região do espaço e, e seguida, fazer um laço sobre as partículas dentro do qual apenas as distâncias entre partículas de regiões vizinhas são calculadas. Note, no entanto, que precisamos criar uma lista para cada região do espaço, contendo a lista das partículas que estão nessa região. Aparentemente, então, precisamos criar uma matriz de $N_c \times N_c$ posições, sendo N_c o número de caixas em cada direção do espaço, e para cada posição uma lista de M partículas, contendo quais as partículas que estão em cada caixa. Este é o método $ingênuo$ de armazenar esta informação. Melhoraremos esta estratégia mais adiante.

Atividades

26. Caso ainda não tenha feito isto, transforme o cálculo do exercício anterior em uma função que tenha como parâmetro de entrada as coordenadas das partículas, e como saída uma lista de a que caixa pertence cada partícula.

27. Faça, agora, uma função que receba como entrada a lista da célula de cada partícula e devolva duas listas: 1) Quantas partículas estão em cada célula. 2) Quais são as partículas de cada célula. O resultado esperado, usando a saída do exercício 25, está descrito em [LINK].

Duas alternativas para a geração de partículas por célula são o uso da função `push!`, que acrescenta elementos em um vetor, ou a pré-alocação do vetor, com dimensões suficientes para conter todas as partículas de cada célula. Essencialmente estas as opções poder ser vistas como variações dos seguintes códigos:

Usando `push!`:

```
list = [ Int[] for i in 1:nc, j in 1:nc ]
for i in 1:N
  icell = ... # first cell coordinate of particle
  jcell = ... # second cell coordinate of particle
  push!(list[icell,jcell],i)
end
```

onde `c` é o número de células em cada dimensão, `N` é o número de partículas, e `icell` e `jcell` são as coordenadas na matriz de células da partícula `i`.

Pré-alocando a matriz, com 3 dimensões, com capacidade de armazenar todas as partículas em cada célula:

```
nlist = zeros(Int64,nc,nc)
list = zeros(Int64,nc,nc,n)
for i in 1:N
  icell = ...
  jcell = ...
  nlist[icell,jcell] = nlist[icell,jcell] + 1
  list[icell,jcell,nlist[icell,jcell]] = i
end
```

Neste caso, é necessário o vetor `nlist`, contendo o número de partículas por célula, para que saibamos quantas posições foram ocupadas em `list`, e qual a próxima posição a ser ocupada.

A opção usando `push!` é, evidentemente, mais econômica em termos de uso de memória final, já que a matriz do segundo exemplo precisa ter uma dimensão sobre-estimada. No entanto, a adição de novos elementos em vetores não é eficiente, porque requer a realocação dos vetores a cada adição.

Se você testar as duas opções, verá uma diferença de tempo desta natureza:

```
Pre-alocating:
  159.148 ns (2 allocations: 1.53 KiB)
Using push:
  599.443 ns (23 allocations: 1.73 KiB)
```

Note que o número de alocações feitas usando `push!` foi maior que pré-alocando, mesmo com a alocação do máximo espaço possível que a matriz de células poderia ter.

Atividades

28. Formule duas versões do programa que calcula a lista de partículas por célula, uma usando `push!` e outra pré-alocando a matriz. Verifique a diferença de tempo das duas abordagens.

Por fim, uma alternativa ainda melhor é pré-alocar as matrizes *fora* da rotina que faz as contas. Se esta matriz vai ter que ser modificada e atualizada muitas vezes, é melhor alocá-la uma única vez, e passar a matriz também como parâmetro para a função que calcula as listas, de forma simplificada, como no código a seguir:

```
nlist = zeros(Int64,nc,nc)
list = zeros(Int64,nc,nc,n)
function celllist!(p,nlist,list)
```

```
  for i in 1:length(p) # loop over particles
    icell = ...
    jcell = ...
    nlist[icell,jcell] = nlist[icell,jcell] + 1
    list[icell,jcell,nlist[icell,jcell]] = i
  end
  return nlist, list
end
```

Usamos aqui a convenção de `Julia` em que a função, por modificar as listas `nlist` e `list`, recebe uma exclamação `!` no nome.

O *benchmark* desta opção resulta em:

```
Allocating outside:
  80.000 ns (0 allocations: 0 bytes)
```

Note que a função é significativamente mais rápida que qualquer das outras alternativas e, como esperado, não aloca mais memória. A pré-alocação de vetores auxiliares é uma parte importante do desenvolvimento de códigos eficientes. Fazer o benchmark desta função requer alguns cuidados, porque a função modifica os argumentos, de forma que execuções sucessivas das função não são idênticas. Veja em [LINK] alguns comentários a respeito.

2.2.3 Calculando a energia

Neste ponto devemos ter em mãos uma uma função que calcula em que célula está cada partícula e, da seção anterior, alguma função que calcula o potencial de interação entre duas partículas a partir de suas posições (calculando a distância). Agora, vamos aproveitar estas informações para fazer uma função que calcula o potencial de interação entre as partículas sem que seja necessário fazer um laço sobre todos os pares de partículas diferentes.

Como ilustrado na Figura 5, as céluas adjacentes a cada célula de coordenadas (i, j) podem ser obtidas simplesmente pela subtração ou adição dos índices. O laço de cálculo de interações é feito sobre todas as partículas e, para cada partícula, sobre as partículas que estão na mesma célula ou em células adjacentes:

```
for ip in 1:N # Number of particles
  icell = ... # Index i of cell of particle ip
  jcell = ... # Index j of cell of particle ip
  for i in icell-1:icell+1
    for j in jcell-1:jcell+1
```

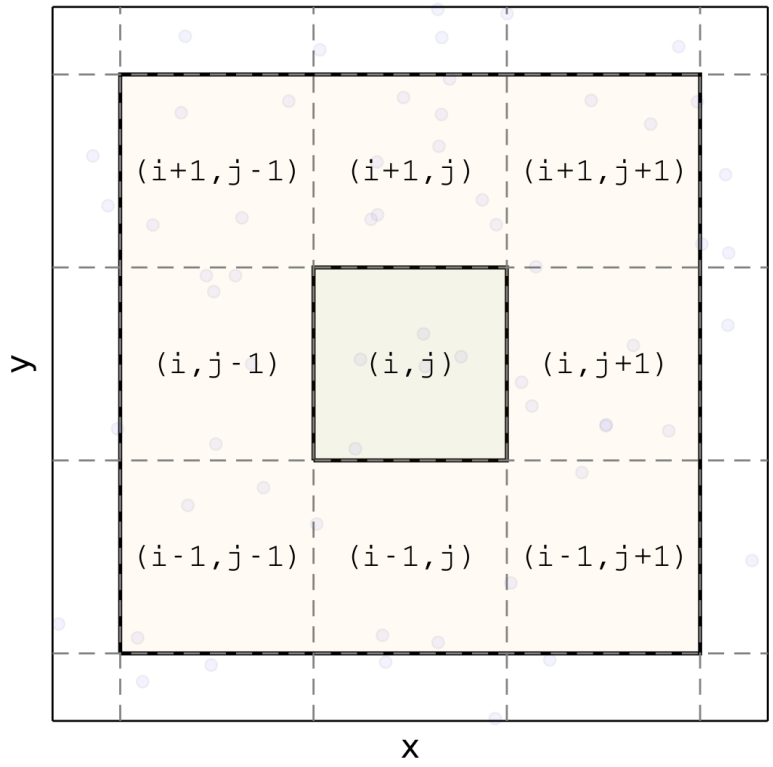

Figura 5: No método das células ligadas, somente as distâncias entre partículas de células adjacentes precisam ser calculadas. As células adjacentes podem ser mapeadas diretamente por seus índices. Figura gerada com `[lcells2.jl]` .

```
      for jp in 1:Nc # Particles of cell (i,j)
        if jp > ip # Avoid repetition
          # Compute interaction between ip and jp
        end
      end
    end
  end
end
```

Como exemplo, em lugar de calcular a energia total, vamos simplesmente calcular menor distância entre partículas. No código a seguir, assumimos que A lista `cellparticles` contém a lista de partículas de cada célula.

Ou seja, `cellparticles[i,j][1] ... cellparticles[i,j][N]` é a lista de `N` partículas da célula (i, j), gerada com uma opção do tipo `push!` da seção anterior. Especificamente, se considerarmos o resultado da atividade 27, temos:

```
julia> cellparticles
4×4 Array{Array{Any,1},2}:
```

```
[6, 10]   [5]   []   []
[]        []    []   [4]
[3, 9]    []    []   [1, 7, 8]
[]        [2]   []   []
```

Evidentemente este método envolve uma série de operações adicionais em relação a um método ingênuo para calcular a distância mínima entre as partículas. É necessário, primeiro, classificar as partículas nas células. Em seguida, é necessário fazer um laço sobre as células vizinhas, e então um laço para todas as partículas de cada célula. Estas operações substituem, no entanto, um laço duplo sobre todos os pares de partículas do sistema, e vão ser compensadoras se o número de partículas for grande e o raio de corte for bem menor que o tamanho total do sistema. Neste caso, evita-se comparar as posições de muitos pares de partículas.

A estrutura de dados acima serve como entrada para a função abaixo, que calcula a mínima distância entre todos os pontos:

```
#
# Computes the minimum distance between a pair of
# particles using the linked cell method
#
function mindist(p,cellparticles,cutoff)
  np = length(p) # Number of particles
  nc = size(cellparticles,1) # Dimension of the grid
  d = +Inf
  for ip in 1:np
    icell = trunc(Int64,p[ip][1]/cutoff)+1
    jcell = trunc(Int64,p[ip][2]/cutoff)+1
    for i in icell-1:icell+1
      if ( i < 1 || i > nc ) continue end # Border
      for j in jcell-1:jcell+1
        if ( j < 1 || j > nc ) continue end # Border
        # Loop over the particles of this cell
        for jp in cellparticles[i,j]
          if jp > ip # Skip repeated
            # Compute distance and keep minimum
            d = min(d,sqrt( (p[ip][1]-p[jp][1])^2 +
                            (p[ip][2]-p[jp][2])^2) )
          end
        end
      end
    end
  end
```

```
  return d
end
```

[Clique aqui para baixar o código]

A execução deste código, para calcular a menor distância entre pares de 10 mil partículas, está detalhada em `[run-mindist.jl]`. Aqui, comparamos o tempo de execução com a o tempo de execução de um método ingênuo para calcular a menor distância, e calculamos também o custo da classificação das partículas em suas regiões. O resultado é:

```
Time for particle classification:
 778.665 μs (16374 allocations: 1.13 MiB)
Using mindist:
 1.872 ms (0 allocations: 0 bytes)
Using mindist_naive:
 246.290 ms (0 allocations: 0 bytes)
```

Note que a soma do tempo de classificação com o tempo de `mindist` é cerca de 90 vezes menor que o tempo usado pelo algoritmo ingênuo. (Aqui uma nota técnica a respeito de um detalhe de implementação que não é irrelevante e deve ser levada em consideração na programação de alto desempenho: [LINK]).

Atividades

29. Modifique o código `[run-mindist.jl]` (ou use sua implementação, o que é ainda melhor), de forma que as funções em vez de calcular a menor distância entre as partículas, calculem o potencial de Lennard-Jones entre elas, com $\varepsilon = 5$ e $\sigma = 2$. Se você gerou os pontos exatamente da mesma forma que no exemplo, o resultado esperado deve ser $U \approx 1.1 \times 10^{20}$ (e é muito alto, porque certamente há partículas quase sobrepostas). O método de células ligadas deve ser da ordem de 80 vezes mais rápido, neste exemplo.

2.2.4 Condições periódicas de contorno

Para simular partículas em um ambiente realista precisamos de condições periódicas de contorno, que agora tem que ser implementadas usando células ligadas. A Figura 6 exemplifica como este problema deve ser abordado.

Na Figura 6 centramos nossa atenção na célula $(1, 1)$, que fica em uma extremidade da caixa de imagem mínima definida entre as coordenadas $(x, y) = (0, 0)$ e $(x, y) = (6, 6)$. As

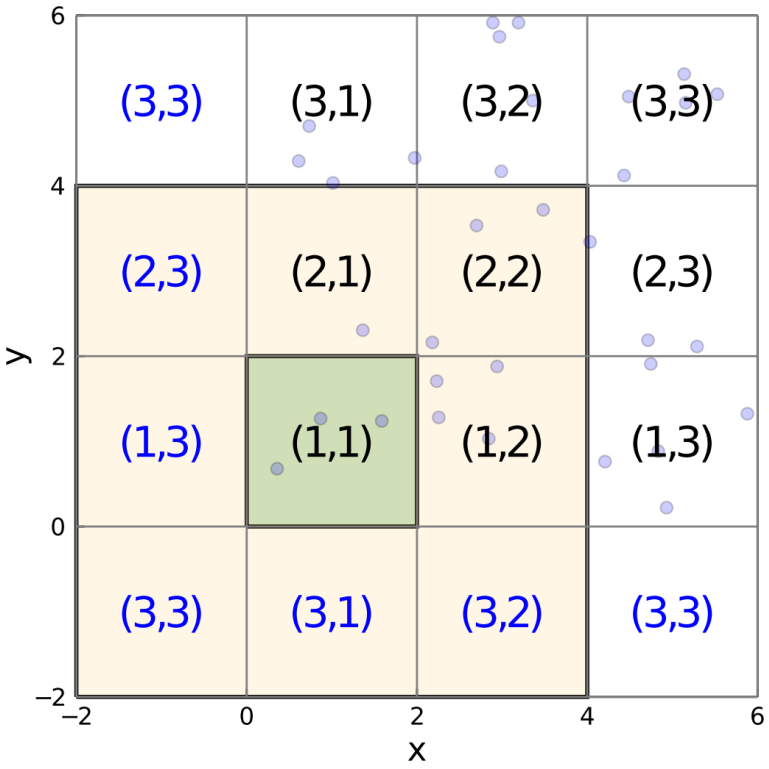

Figura 6: Projeção das células ligadas no sistema periódico. O sistema tem lado de tamanho 6 e as células tem lado 2. Figura gerada com `[lcells3.jl]` .

partículas da célula $(1, 1)$ devem interagir com as partículas da região alaranjada (incluindo a célula verde). Em azul temos os índices das células correspondentes, considerando a periodicidade do sistema.

No código `[mindist.jl]` , descrito na seção anterior, tomamos o cuidado de não percorrer células inexistentes no laço sobre as células vizinhas de cada outra célula. Para isto, introduzimos dois condicionais,

```
...
  if ( i < 1 || i > nc ) continue end # Border
...
    if ( j < 1 || j > nc ) continue end # Border
```

que garantiam que índices iguais a zero ou maiores que `nc` não eram percorridos. Isto era importante porque a lista de células não possui nada nesses índices e, portanto, percorrer esses índices na lista `cellparticles` resultaria em erro.

Agora, em lugar de evitar simplesmente estes índices vamos, nas mesmas condições de excepcionais, recalcular os índices para que a célula considerada seja a correspondente periódica.

Para isto, vamos introduzir uma função que recalcula as coordenadas da célula, levando em considerações possíveis periodicidades. Ou seja, se considerarmos o exemplo da Figura 6, queremos uma função que receba por exemplo as coordenadas da célula $(-1, -1)$, correspondente à diagonal inferior da célula $(1, 1)$ e devolva as coordenadas da célula que efetivamente deve ser considerada, $(3, 3)$.

Para testarmos uma implementação de tal função, vamos construir o seguinte teste:

```
function check_wrap(nc)
  list = Matrix{Int}(undef,0,4)
  for i in 1:nc[1]
    for j in 1:nc[2]
      for ic in i-1:i+1
        for jc in j-1:j+1
           iw, jw = wrap_cell(nc,ic,jc)
           list = vcat(list,[ic jc iw jw])
        end
      end
    end
  end
  return list
end
nc = [3,3]
list = check_wrap(nc)
println(list)
```

O que estamos fazendo neste código é percorrer todas as células "reais" da Figura 6, que tem índices de $(1, 1)$ a $(3, 3)$, (índices de 1 até `nc` em cada direção). Em seguida, nos dois laços internos (em `ic` e `jc`) estamos percorrendo todas as células vizinhas a cada cada célula.

A função `wrap_cell` é a função que você vai desenvolver, e o que ela faz é retornar os índices `iw` e `jw` que devem ser usados para calcular as interações das partículas da célula (i, j), corrigidos pelas condições periódicas de contorno.

O resultado está sendo salvo em uma matriz de 4 colunas, a qual poderemos ver com detalhes para confirmar que a função `wrap_cell` está fazendo o trabalho correto. Aqui aprendemos duas notações que podem ser úteis: A inicialização de uma matriz de dimensão definida, mas vazia (nenhuma linha, 4 colunas):

```
list = Matrix{Int}(undef,0,4)
```

e a adição de linhas a esta matriz, usando o comando `vcat` (de *vertical-cat*, em referência

ao comando `cat` do linux):

```
list = vcat(list,[ic jc iw jw])
```

Note que os índices não estão separados por vírgulas, o que significa que são quatro números na mesma *linha*.

A função `check_wrap` retorna a matriz completa. Por exemplo, teremos:

```
julia> list
81×4 Array{Int64,2}:
 0  0  3  3
 0  1  3  1
...
```

que indica que a célula de índices $(0, 0)$ foi mapeada à célula $(3, 3)$, e a célula de índices $(0, 1)$ foi mapeada à célula de índices $(3, 1)$. Volte à Figura 6 e verifique se isto faz sentido.

Atividades

30. Escreva a função `wrap_cell`, que recebe como input os índices de uma célula e o número de células em cada direção, e retorna a célula mapeada de forma correta para seus índices "reais". O resultado da sua função deve corresponder, se introduzida no código acima, à lista apresentada em [LINK].