Fundamentos Computacionais de Simulações em Química

Leandro Martínez
leandro@iqm.unicamp.br
Última atualização: 17 de setembro 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 SHELL[1]. As linguagens compiladas não possuem essa forma interativa, 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 como uma linguagem interpretada e prática como Python, mas a geração de códigos rápidos como os de Fortran e C (ou quase). É uma linguagem que pode ser usada de forma interativa, e possui um SHELL, 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.

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

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 ao 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 é um curso de fundamentos de simulação. As simulações requerem uma linguagem de programação, e o 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 um substituto 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). Pode ser obtido 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 (neste caso, professor e monitores, se eles souberem).

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 1.0.2 (2018-11-08)
  _/ |\__'_|_|_|\__'_|  |  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.

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".

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
  pritnln("i is 2")
elseif i == 3
  pritnln("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.

Cuidado com a notação dos operadores "e" (`&&`) e "ou" (`||`), que são as duplas de caracteres. Os operadores simples (`&` e `|`) também funcionam mas tem 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
```

Copie este código e veja o que ele faz. Veja se a sintaxe faz sentido para você. Varie os números e observe os resultados. Aqui apareceu, pela primeira vez, o comando `println()`, que escreve algo na tela. Há dois comandos para isso, `print()` e `println()`, sendo que no segundo vai-se para a próxima linha depois de escrever o que se quer. Teste os dois nesse laço e veja a diferença.

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
```

Atividades

3. Teste o laço acima e veja o resultado. Substitua `break` por `continue` e veja o resultado.

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 coisa, 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.

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
```

Escrever um laço explícito para fazer toda operação componente a componente de vetores pode ser muito chato. Em Julia isso se resolve usando uma notação compacta que se chama "dot" (ponto). O nome vem do fato de que o produto interno de vetores, cuja notação é um ponto, consiste em multiplicar cada componente de um par de vetores. A notação aqui significa que a operação, seja qual for, será feita componente a componente. Assim, o laço acima pode ser reescrito nesta notação compacta um `@.` antes da conta, como

```
julia> z = similar(x);
julia> @. z = x + 1
```

Se quisermos o vetor que tem em cada componente o produto das componentes de cada um dos vetores, fazemos:

```
julia> x = [ 2, 3 ];
julia> y = [ 3, 4 ];
julia> @. x * y
2-element Array{Int64,1}:
  6
 12
```

Note que o resultado é o produto, componente a componente, dos dois vetores. O produto `x*y` vai resultar em um erro, porque não está bem definido.

Atividades

4. Escreva o laço explícito que faz o produto componente a componente dos vetores, e salva o resultado em um terceiro vetor.

As notações compactas são práticas, mas são totalmente supérfluas e muitas vezes atrapalham a leitura do código. Linguagens de baixo nível como C e Fortran têm poucos atalhos desse tipo. Você não vai programar mais e melhor por economizar toques no teclado. O uso dessas notações é bom quando o código fica mais legível com o seu uso, e essa é a única razão para que sejam usadas.

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. E 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, e em Julia, 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.

Atividades

5. Em Julia, você pode definir um número real de 32 bits usando a notação, por exemplo, `1.f0`, `1.f10`, etc. Repita o exemplo anterior para descobrir quantas ordens de grandeza diferentes são tratáveis com números reais de 32 bits.

O maior número inteiro que pode ser representado com 64 bits é 9223372036854775807. Novamente, o padrão é usar 64 bits. Você pode testar o tipo de número usando o comando `typeof()`. Por exemplo:

```
julia> typeof(1)
Int64
```

Atividades

6. Some 1 ao número acima, na mão, e use o comando `typeof()`. Qual o tipo indicado?

Você deve ter notado que o tipo de número mudou para um inteiro de 128 bits, para ser capaz de acomodar o número maior. Teste, agora o seguinte:

```
julia> i = 9223372036854775807
julia> i + 1
```

O resultado deve ser surpreendente. O que aconteceu? O resultado anterior, em que o número mudou de tipo, deve ter parecido muito mais razoável. No entanto, essa possibilidade faz o programa ser muito mais lento. Linguagens compiladas, que são eficientes, não permitem mudanças de tipo o tempo todo, e Julia não mudará o tipo da variável a não ser que você queira. O resultado que você viu agora é o mesmo que obteria em C, C++ ou Fortran.

É fundamental conhecer essas limitações na 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 consegue 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
```

Atividades

7. Defina duas funções de mesmo nome, que recebem tipos diferentes, e mostre como chamando a função com parâmetros diferentes uma o outra versão da função é acionada.

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
```

Atividades

8. Repita o exemplo acima, mas defina `x` usando `x = [ 2, 2 ]`. (a diferença é que não há um ponto depois de cada número 2). O que aconteceu? Explique.

9. A função `sum()` retorna a soma dos elementos de um vetor. Combine as informações que você já aprendeu com esta e calcule o produto interno de depois vetores.

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

10. 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
Int64(x) # Converts Float to Integer
```

Atividades

11. Teste as funções acima com diferentes tipos de argumentos

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")
```

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

12. 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:

[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> x = [ 1, 1 ]
julia> y = copy(x)
```

Atividades

13. 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

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

1.5.3 Variáveis globais e locais

Esta seção discute um aspecto técnico da linguagem que dificilmente você entenderá completamente agora. Não se preocupe. É necessário colocar aqui a questão para que, quando erros aparecerem, você tenha ouvido falar que a solução existe, e vá se habituando a ela. No entanto, é bem provável que considere esta seção desanimadora. Não se preocupe, vá adiante sem grandes preocupações, e um dia você volta a ela quando se enfrentar com os erros no código que estas questões podem gerar.

A principal mensagem objetiva desta parte é: Se você quiser modificar uma variável escalar dentro de um laço ou função, mas a variável foi criada fora, tem que explicitar que a variável é de fora, usando o comando `global`. Por exemplo,

```
julia> a = 1
julia> for i in 1:2
         global a
         a = 2
       end
julia> function f()
         global a
         a = 2
       end
```

Se não fizer isso, a modificação será feita em uma variável interna do laço ou função. Agora vamos para a explicação.

Como mencionado na introdução, Julia tem uma particularidade que é que funções são compiladas e transformadas em códigos rápidos sem que você perceba. Todos os laços também são compilados, para serem rápidos. Isto traz uma complicação no uso da interface interativa, que ilustramos aqui:

```
julia> x = 1
julia> for i in 1:2
         x = x + 1
       end
```

O esperado aqui seria que $\mathtt{x}$ saísse do laço valendo 3. No entanto, você verá um erro, dizendo que $\mathtt{x}$ não está definido.

Para entender o erro, vamos primeiro escrever o laço trocando o $\mathtt{x}$ do lado esquerdo por outro nome, no caso $\mathtt{y}$:

```
julia> x = 1
julia> for i in 1:2
         y = x + 1
       end
julia> y
```

O laço em si não tem nenhum erro. E podemos verificar que $\mathtt{y}$ assume os valores esperados se mandarmos escrevê-lo dentro do laço. No entanto, ao tentar ver o valor de $\mathtt{y}$ fora do laço, recebemo o erro de que não está definido.

Isto ocorre pelo que chamamos o $escopo$ das variáveis. Quando criamos uma variável fora de qualquer laço ou função, esta variável é $global$, ou seja, pode ser usada em qualquer lugar. Por exemplo, dentro de uma função, mesmo não sendo parâmetro dela:

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

No entanto, a variável z, que foi criada dentro da função, é de *escopo local* da função, e não pode ser usada fora. Continuando o código anterior,

```
julia> f(2)
3
julia> z
ERROR: UndefVarError: z not defined
```

Ainda mais especificamente, mesmo que a variável tenha o mesmo nome da variável global, ela vai ser local da função e não modificará a variável global:

```
julia> x = 1
julia> function f()
         x = 2
       end
julia> f()
julia> x
1
```

Em Julia, os laços geram um escopo local da mesma forma que as funções. Isto porque se comportam igual que as funções (são compilados antes de serem executados) para que a eficiência seja máxima. Voltando ao primeiro laço, que é o que gera o erro ambíguo:

```
julia> x = 1
julia> for i in 1:2
         x = x + 1
       end
```

Se tentarmos fazer uma função parecida, que gere um escopo local, e pretendêssemos usar `x`, teríamos

```
julia> x = 1
julia> function f()
         x = x + 1
       end
julia> f()
```

Receberemos o mesmo erro ao tentar executar `f()`. Aqui o erro deve parecer mais natural, porque `x` não é enviado à função, portanto a conta `x = x + 1` de fato parece estar usando um número que não foi definido dentro da função nem passado como parâmetro. Para usar o `x` global dentro da função, teríamos que explicitar que é daquele `x` que estamos falando, usando

```
julia> x = 1
julia> function f()
```

```
        global x
        x = x + 1
      end
julia> f()
```

A resumo da história é que os laços, por se comportarem como funções, requerem a mesma atenção se forem usadas variáveis globais dentro deles, e isto funciona como esperado:

```
julia> x = 1
julia> for i in 1:2
        global x
        x = x + 1
      end
julia> x
3
```

Por fim, esta problemática só acontece se estivermos trabalhando no escopo global (isto é, fora das funções). Dentro de uma função este problema não existe: as variáveis que o laço usa são as variáveis do escopo da função, por padrão:

```
julia> function f()
        x = 1
        for i in 1:2
          x = x + 1
        end
        println(" x inside function = ",x)
      end
julia> f()
 x inside function = 3
```

No entanto, se dentro do laço for criada uma nova variável, ela continua sendo do escopo do laço, e não é visível nas outras partes da função:

```
julia> function f()
        for i in 1:2
           y = 1
        end
        println(y)
      end
f (generic function with 1 method)

julia> f()
ERROR: UndefVarError: y not defined
```

Mais especificamente, a variável definida dentro do laço é específica para cada volta específica do laço, como mostra o exemplo:

```
julia> function f()
        for i in 1:2
           println(" volta: ",i)
           if i == 1
             y = 1
           end
           println(" y = ",y)
        end
```

```
        end;
julia> f()
 volta: 1
 y = 1
 volta: 2
ERROR: UndefVarError: y not defined
```

Na segunda volta do laço, y não está definido. Se você quer que a variável esteja disponível em todas as voltas do laço, mesmo ela sendo atualizada em uma volta específica, é necessário criar ela fora do laço. Isso pode ser feito atribuindo um valor qualquer à variável, ou simplesmente declarando que ela existe e é local:

```
julia> function f()
         local y # or y = 0, for example
         for i in 1:2
           if i == 1
             y = i
           end
           println(" y = ",y)
         end
       end
f (generic function with 1 method)

julia> f()
 y = 1
 y = 1
```

Resumindo, sempre que receber uma mensagem de erro associada a variáveis não definidas, verifique se não é um exemplo como estes. O entendimento detalhado destas questões pode ser complicado, de forma que tenta-se evitar explicar isto no início, mas há algumas situações muito simples, como o primeiro laço desta seção, em que erros aparecem e são difíceis de entender.

Atividades

15. Crie um laço no escopo global que atualize o valor de uma variável criada também no escopo global. Para isso, você tem que usar o comando `global`, que explicita que qual a variável que está sendo modificada.

2 Primeiras simulações: cinética química

Aqui estudaremos a simulação de uma cinética de primeira ordem, irreversível, na forma

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B}$$

A equação diferencial que determina como a concentração de A varia no tempo é

$$-\frac{d[\mathrm{A}]}{dt} = k_1[\mathrm{A}]$$

Dada uma concentração inicial $[\mathrm{A}](0)$, podemos calcular uma aproximação da concentração em um tempo posterior, usando

$$[\mathrm{A}](\Delta t) = [\mathrm{A}](0) + \frac{d[\mathrm{A}]}{dt}\Delta t$$

ou seja,

$$[\mathrm{A}](\Delta t) = [\mathrm{A}](0) - k_1[\mathrm{A}](0)\Delta t$$

O resultado desta conta nos dá um novo valor de concentração, que permite que calculemos a concentração em um instante mais avançado no tempo. A fórmula geral deste processo é

$$[\mathrm{A}](t+\Delta t) = [\mathrm{A}](t) - k_1[\mathrm{A}](t)\Delta t \tag{1}$$

Nosso programa será executado, e faremos um gráfico dos resultados, usando os seguintes comandos:

```
include("./sim1.jl")
CA0 = 10. # Initial concentration
k1 = 0.1  # Velocity constant
time = 100. # Simulation time
t, c = sim1( CA0, k1, time )
plot(t,c,xlabel="time",ylabel="[A]")
```

No código `sim1.jl` temos a função que faz a simulação. Esta função executa o procedimento acima, a integração da equação de velocidade, até que o tempo desejado é atingido. Estude o código com atenção:

```
function sim1( CA0, k1, time )
  dt = 1.e-1 # Time-step
  nsteps = Int64(time/dt) # Number of steps
  t = Vector{Float64}(undef,nsteps)
  CA = Vector{Float64}(undef,nsteps)
  CA[1] = CA0
  t[1] = 0.
  for i in 2:nsteps
    CA[i] = CA[i-1] - k1*CA[i-1]*dt
    t[i] = t[i-1]+dt
  end
  return t, CA
end
```

[Clique para baixar o código]

Você pode variar as condições da simulação, alterando os parâmetros `CA`, `k1` e `time`. Podemos variar esses parâmetros simplesmente chamando a função com parâmetros diferntes, como em

```
t, c = sim1( 10. , 0.2, 100. )
plot(t,c,xlabel="time",ylabel="[A]")
```

Por fim, podemos salvar os resultados, seja salvando a figura gerada, ou criando um arquivo com os dados gerados que podem ser lidos em qualquer outro programa de gráficos, como o Origin:

```
savefig("./sim1.pdf")
file = open("sim1.dat","w")
i = 0
for x in t
  global i = i + 1
  println(file,t[i]," ",c[i])
end
close(file)
```

Atividades

16. Modifique o programa para que `dt` seja também um parâmetro da função.
17. Varie os valores de `CA`, `k1`, `time` e `dt` no programa, e observe o comportamento da concentração em função do tempo em cada caso.
18. Um dos resultados possíveis da execução do programa acima, é a obtenção de concentrações negativas para o reagente A. Naturalmente, isso se deve a um erro numérico. Adicione um "`if ... end`" ao programa que detecte esse erro, escreva uma mensagem de erro, e pare o programa.
19. Adicione ao programa o cálculo da concentração do produto B. Assuma que a concentração de B tem um valor inicial especifico.

Nós sabemos que a solução analítica da equação diferencial deste problema é simplesmente $[\mathrm{A}(t)] = [\mathrm{A}](0)e^{-k_1 t}$. Essa exponencial se escreve como `exp(-k1*t)`. Podemos fazer o gráfico desta função criando um vetor para os mesmos tempos, usando:

```
nsteps = length(t)
CA_exact = Vector{Float64}(undef,nsteps)
for i in 1:length(CA_exact)
  CA_exact[i] = CA0*exp(-k1*t[i])
end
plot!(t,CA_exact,label="Exact")
```

A exclamação em `plot!` indica que o gráfico vai ser acrescentado à figura anterior (em Julia usa-se a convenção de que funções que modificam os argumentos levam uma exclamação).

Atividades

20. Escreva o resultado da solução analítica em seguida de CA, no programa, e veja que diferença tem a solução analítica da solução numérica, em cada tempo, em função dos parâmetros `CA`, `k1` e `dt`.

O código que usamos para calcular a solução analítica pode ser escrito de forma mais compacta, por exemplo:

```
CA_exact = similar(t)
@. CA_exact = CA0*exp(-k1*t)
```

A primeira linha cria um vetor `CA_exact` do mesmo tipo do vetor `t`, isto é, um vetor de mesmo tamanho e do mesmo tipo de dados, no caso `Float64`. A segunda linha é uma notação de `Julia` que indica que a conta deve ser feita para cada componente dos vetores envolvidos, no caso os vetores `CA_exact` e `t`. Este código faz a mesma coisa que o laço anterior. A notação é muito prática para fazer operações com grandezas vetoriais (de mesma dimensão).

É relativamente fácil lidar com o aumento da complexidade do mecanismo reacional em uma simulação de cinética química. A reação reversível,

$$\mathrm{A} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} \mathrm{B}$$

tem duas constantes de velocidade, e as equações diferenciais que regem o comportamento das concentrações de A e B são, agora, dependentes destas duas velocidades. Ou seja, agora temos duas equações diferenciais que regem a evolução temporal das concentrações:

$$-\frac{d[\mathrm{A}]}{dt} = k_1[\mathrm{A}] - k_{-1}[\mathrm{B}]$$

$$\frac{d[\mathrm{B}]}{dt} = k_1[\mathrm{A}] - k_{-1}[\mathrm{B}]$$

Atividades

21. Escreva a *discretização* de cada uma destas equações, seguindo o exemplo da Equação 1.

Há duas maneiras de programar a evolução temporal das concentrações das espécies. Uma delas propaga as concentrações usando as discretizações na forma da Equação 1 para ambas as espécies. A outra é usar um balanço de massa, já que sabemos que existe uma relação entre as concentrações. Neste caso, $[\mathrm{A}] + [\mathrm{B}] = [\mathrm{A}]_0 + [\mathrm{B}]_0$. A função que usa o balanço de massa pode ser:

```
function sim2( CA0, CB0, k1, km1, time)
  dt = 1.e-1 # Time-step
  nsteps = Int64(time/dt) # Number of steps
  t = Vector{Float64}(undef,nsteps)
  CA = Vector{Float64}(undef,nsteps)
  CB = Vector{Float64}(undef,nsteps)
  CA[1] = CA0
  CB[1] = CB0
  t[1] = 0.
  for i in 2:nsteps
    CA[i] = CA[i-1] - k1*CA[i-1]*dt + km1*CB[i-1]*dt
    CB[i] = CA0 + CB0 - CA[i]
    t[i] = t[i-1] + dt
  end
  return t, CA, CB
end
```

[Clique para baixar o código]

Entenda bem o programa acima. Note que, agora, temos que fornecer à função duas concentrações iniciais, `CA0` e `CB0`, e a função retorna três vetores.

Atividades

22. Faça um gráfico que contenha as concentrações dos dois reagentes em função do tempo, usando o comando `plot!`.

23. O programa acima pode ser modificado para calcular a concentração de B usando a discretização da equação diferencial correspondente em vez do balanço de massa. Faça isso.

24. Com esta modificação, é possível testar a precisão da propagação das concentrações já que, em princípio, deveriam satisfazer sempre o balanço de massa. Se não satisfazem, isto quer dizer que há algo que não está indo bem. Calcule o erro no balanço de massa ao longo da execução do programa, e escreva este erro. Estude como este erro varia em função do passo de tempo e das constantes de velocidade.

25. Você terá percebido que sempre há um erro associado à propagação das concentrações. Modifique seu programa adicionando um teste (`if...`) que detecte quando o erro for grande demais.

Não há nada fundamentalmente diferente do que fizemos até aqui para a simulação da cinética de nenhum sistema químico. Basta escrever e discretizar as equações diferenciais correspondentes ao mecanismo reacional proposto, e programar a integração numérica das equações.

Atividades

26. O mecanismo-modelo mais conhecido de catálise enzimática é o mecanismo de Michaellis-Menten,

$$\mathrm{E} + \mathrm{S} \underset{k_f}{\overset{k_r}{\rightleftharpoons}} \mathrm{ES} \xrightarrow{k_{cat}} \mathrm{E} + \mathrm{P}$$

onde E e S são a enzima e o substrato, ES é o complexo enzima-substrato, e P é o produto da reação. Faça um programa que simule esta cinética enzimática. Varie as concentrações dos reagentes para encontrar as condições nas quais a aproximação do estado estacionário é razoável (isto é, que a concentração do complexo é aproximadamente constante ao longo da reação).

27. A decomposição da camada de ozônio pelos clorofluorcarbonos acontece através de uma reação em cadeia. A reação se inicia pela decomposição do clorofluorcarbono formando cloro radicalar, o que acontece sob radiação ultravioleta,

$$\mathrm{CFCl_3} \rightarrow \mathrm{Cl}^{\bullet} + {}^{\bullet}\mathrm{CFCL_2}$$

O cloro radicalar decompõe o ozônio de forma catalítica, pelas reações

$$\mathrm{Cl}^{\bullet} + \mathrm{O_3} \rightarrow \mathrm{ClO} + \mathrm{O_2}$$

$$\mathrm{ClO} + \mathrm{O_3} \rightarrow \mathrm{Cl}^{\bullet} + 2\mathrm{O_2}$$

Faça um programa que simule este mecanismo reacional.

3 Otimização com derivadas

Quase todo problema real envolve a maximização ou minimização de alguma coisa. Falemos de Química. Quando sintetizamos um novo composto, tentamos maximizar o rendimento. Quando criamos um novo material, tentamos melhorar algumas de suas características que nos interessam, como, por exemplo, a eficiência de uma célula solar, ou a resistência de um polímero. Quando fazemos um ajuste de uma curva experimental, estamos encontrando qual a equação (linear ou não) que melhor se ajusta às observações. Na química teórica, permanentemente estamos minimizando energias para obter estruturas químicas melhores e, com sorte, mais representativas das estruturas reais.

A otimização, do ponto de vista da computação, se divide em duas grandes áreas: otimização com o uso de derivadas, e otimização sem o uso de derivadas. As derivadas de uma função indicam para onde esta função cresce, portanto consistem em uma informação importantíssima, se disponível, na resolução de um problema de otimização. As derivadas de uma função são conhecidas, no entanto, somente quando conhecemos, explicitamente, a função, e quando é derivável. Quando não conhecemos a função, temos que otimizar sem derivadas. Por exemplo, se queremos minimizar a função $f(x) = x^2$, podemos usar a informação sobre sua derivada, $df/dx = 2x$. No entanto, se queremos maximizar a resistência de um compósito variando sua composição, não temos a derivada, porque não conhecemos a *função* que nos dá a resistência como função da composição. Vamos aprender, agora, os conceitos básicos da minimização com e sem derivadas, e veremos que muitos problemas reais podem ser modelados usando uma ou outra técnica.

3.1 Minimizando com derivadas

Vamos construir um programa, o mais simples possível, que procure usar a informação da derivada de uma função para encontrar seu valor mínimo. Tomemos a função

$$f(x) = x^2,$$

cuja derivada é

$$\frac{df}{dx} = 2x$$

Nós sabemos que o mínimo desta função está em $x = 0$. O que nos interessa aqui é como um tal problema é abordado do ponto de vista numérico.

A derivada indica para que lado a função cresce, portanto "menos-a-derivada" indica para que lado ela diminui. O programa mais simples que tenta obter o mínimo da função é aquele que simplesmente testa a função em vários pontos, na direção indicada pela derivada, e para se a função aumenta. Testar a função na direção indicada pela derivada significa usar uma aproximação da função. Por exemplo, a aproximação de Taylor de primeira ordem,

$$f(x_1) \approx f(x_0) + f'(x_0)(x_1 - x_0) \tag{2}$$

onde $f'(x_0)$ é a derivada de $f(x)$ calculada no ponto x_0. Portanto, o processo de busca do mínimo da função vai consistir em testar pontos x_1 diferentes, na direção em que a derivada indica que o mínimo deve estar. A direção da derivada, no caso unidimensional, é apenas seu sinal, isto é, se ela indica se a função cresce na direção de x mais positivo, ou negativo. Portanto, o passo em x é dado na direção

$$-\frac{f'(x_0)}{|f'(x_0)|}$$

que pode valer $+1$ ou -1. Este caso é muito simples, mas se a função tivesse mais dimensões, a direção seria dada por $-\nabla f/|\nabla f|$, e seria um vetor unitário na direção da derivada. Faremos isso mais tarde. A função abaixo recebe um chute inicial, `x0`, e procura o mínimo da função usando essa estratégia (`abs(x)` é o módulo do número x):

```
function min1(x0)
  x = x0
  deltax = 0.1 # Step size
  xbest = x # Save best point
  fbest = x^2 # Best value up to now
  println(" Initial point: ",xbest," ",fbest)
  deltaf = -1.
  while deltaf < 0.
    # Move x in the descent direction, with step deltax
    dfdx = 2*x # Computing the derivative
    x = x - deltax * dfdx/abs(dfdx) # Move x in the -f' direction
    # Test new point
    deltaf = x^2 - fbest
    # Write current point
    println("Current point: ",x," ",x^2," ",deltaf)
    # If the function decreased, save best point
    if deltaf < 0.
      xbest = x
      fbest = x^2
    else
      println(" Function is increasing. ")
      println(" Best solution found: x = ", xbest, " f(x) = ",fbest)
      return xbest, fbest
    end
  end
end
```

[Clique para baixar o código]

Este código será executado usando, por exemplo,

```
include("./min1.jl")
xbest, fbest = min1(5.3)
```

onde `5.3` é o chute inicial.

O código acima possui muitas das características essenciais de qualquer programa de simulação, ou otimização. O programa se inicia atribuindo valores inciais para as variáveis (x) e os parâmetros que serão usados. Neste caso, o método envolve um único parâmetro, Δx, que é o tamanho do "passo" que vai ser dado na direção da derivada decrescente (a diferença $x_1 - x_0$ da Equação 2). Dentro do loop calcula-se a derivada, e modifica-se o valor de x na direção da derivada, com passo Δx, i. e., $x = x - f'(x)\Delta x$. Testamos, em seguida, se a função aumentou o diminuiu. Se diminuiu, o novo x é salvo como o melhor x até o momento. Se aumentou, decretamos que o programa terminou.

Atividades

28. Modifique a função para que Δx seja também um parâmetro da função.

29. Varie o valor de Δx no programa acima, e estude o que acontece. Qual a precisão das soluções atingidas? O que ocorre se o passo Δx for muito grande?

30. Modifique o programa de tal forma que quando a função aumenta, em vez de parar, simplesmente continua-se, mas sem salvar o ponto de maior valor como melhor ponto. Estabeleça um limite de número de voltas no loop, porque agora o programa poderá ficar rodando para sempre. Observe o que acontece com a variação do passo Δx, em particular para passos grandes.

31. Faça um programa que minimize a função $x^2 + \text{sen}\,(10x)$. Faça testes variando o tamanho de passo e os pontos iniciais, no intervalo $[-2, 2]$.

32. Uma alternativa razoável para o passo Δx é que ele seja proporcional à derivada. Ou seja, se a derivada é grande, dá-se um passo grande, se é pequena, dá-se um passo pequeno em x. Para isso, basta eliminar a normalização da direção da derivada. Faça isso na função `min1`, e avalie as características do processo de minimização (número de iterações até o fim e precisão da solução).

Variar de forma inteligente o passo de acordo com o que acontece com a função torna os algoritmos de otimização muito mais eficientes e robustos. A maior parte dos métodos de otimização usam alguma coisa similar ao que vamos descrever agora.

A ideia consiste em mover as variáveis na direção desejada (neste caso, na direção contrária à derivada) e, antes de *aceitar* o novo ponto, *testar* se a função aumento ou diminuiu. Se a função diminuiu, que é o que queremos, a direção em que andamos é boa e, talvez, possamos aumentar o passo. Além disso, aceitamos o novo ponto. Se a função aumentou, o que não queremos, não aceitamos o novo ponto, e reduzimos o passo para ficarmos mais próximos do ponto atual. Estas ideias estão associadas ao fato de que estamos nos movendo nas variáveis usando uma aproximação de Taylor. A aproximação de Taylor é boa próxima do ponto *corrente*. Se o passo for suficientemente pequeno, a função tem que diminuir se a derivada assim o diz. Se o passo for grande demais a função pode aumentar, porque a aproximação é ruim longe do ponto em que foi feita.

O programa, então, tem que ser modificado, para introduzir este passo de tamanho variável. Fundamentalmente, temos que introduzir alguns testes, e um critério para a variação do passo Δx:

```
...
if ftrial < fbest
  deltax = deltax * 2
  fbest = ftrial
  xbest = xtrial
else
  deltax = deltax / 2
end
...
```

O ponto é que, agora, antes de modificar efetivamente a variável x, vamos modificar outra variável, x_{trial}, e testar o que acontece com o valor da função nesse ponto de teste. Dependendo de como varia a função,

tomamos uma decisão sobre a atualização do melhor ponto e outra sobre o tamanho do passo na iteração seguinte. Note que, agora, o valor da função sempre vai diminuir em pontos $aceitados$, portanto temos que definir um critério para parar o programa. Um critério razoável é, por exemplo, o valor da derivada. Se ela for muito pequena, podemos considerar que chegamos em um ponto crítico da função. A precisão com que desejamos a solução, ou seja, quão pequena tem que ser a derivada para que consideremos que o ponto é uma solução (um ponto crítico) é um parâmetro que deve ser introduzido também.

Atividades

33. Modifique o programa do exercício 32, introduzindo as modificações discutidas nesta parte. Várias modificações devem ser feitas no programa, e é importante entender o que está se fazendo antes de modificar o programa. A solução está na lista de soluções. Mas tente bastante antes de olhar lá. Vale a pena.

34. Compare o programa da atividade anterior com o programa da atividade 32 quanto à eficiência em encontrar uma solução. A eficiência, neste caso, é composta pelo número de $iterações$, e pelo valor final da função.

3.2 Funções de múltiplas variáveis

Agora vamos modificar o programa para trabalhar com funções de mais de uma variável. Neste caso, a direção de aumento da função é dada pelo gradiente da função, um vetor. Tomemos a função $f(x, y) = x^2 + y^2$, por exemplo. Seu gradiente é o vetor

$$\nabla f = \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

Portanto, mover o ponto (x, y) na direção de decréscimo da função significa mover cada uma das suas variáveis nas direções dadas pelas derivadas parciais,

$$x_{\text{trial}} = x - \frac{\partial f}{\partial x}(x)\Delta x$$

$$y_{\text{trial}} = y - \frac{\partial f}{\partial y}(y)\Delta y$$

Em geral, usamos o mesmo passo básico, Δs para as duas variáveis, e podemos escrever as equações acima na forma

$$\begin{bmatrix} x \\ y \end{bmatrix}_{\text{trial}} = \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \end{bmatrix} \Delta s.$$

Esta mesma equação pode ser escrita de forma genérica e sucinta usando notação vetorial,

$$\vec{x}_{\text{trial}} = \vec{x} - \nabla f \Delta s$$

O programa pode minimizar a função $x^2 + y^2$ usando o mesmo procedimento que foi usado para as funções de uma variável. Calcula-se um $\vec{x}_{\text{trial}}$, testa-se se a função diminui. Se a função diminui, aceita-se $\vec{x}_{\text{trial}}$ como o novo $\vec{x}$ e aumenta-se o passo Δs. Em caso contrário, diminui-se Δs.

Atividades

35. Modifique o programa da atividade 33 para minimizar a função $x^2 + y^2$. Agora, em lugar da derivada, temos um gradiente. Em lugar do módulo da derivada como critério de parada, temos a norma (módulo) do gradiente.

Usando a notação vetorial, definimos que a função é dependente dos elementos do vetor $\vec{x}$, que chamaremos x_1 e x_2. No programa, o vetor é declarado usando

```
x = Vector{Float64}(undef,2)
```

e, em lugar de usar duas variáveis, `x` e `y`, usaremos as componentes do vetor `x[1]` e `x[2]`. O mesmo vale para a derivada. O resto do programa é quase idêntico ao anterior.

Atividades

36. Modifique o programa anterior para usar a notação vetorial, em lugar de definir uma variável diferente para cada variável do problema.
37. Modifique o programa anterior para minimizar a função $x^2 + y^2 + z^2$.
38. Modifique o programa anterior para minimizar a função $x_1^2 + x_2^2 + \cdots + x_{1000}^2$.

3.3 Decompondo o programa em funções

Os nossos programas estão ficando complicados, e você já deve ter percebido que estamos começando a repetir muitas coisas. Para aproveitar os programas feitos antes, e evitar erros cada vez que algo novo vai ser feito, vamos introduzir o uso de funções de forma mais abrangente e inteligente, e isso vai organizar muito o código.

No programa da atividade 31, todas as vezes que você decidir calcular a função (que no caso é $f(x, y, z) = x^2 + y^2 + z^2$, em lugar de escrever explicitamente a conta, chame uma nova função, por exemplo,

```
fbest = f(x)
```

lembrando que x é um vetor. Precisamos definir o que é `f(x)`, *antes* de chamar nosso minimizador. Podemos fazer isso de duas formas. Em notação bem compacta, temos

```
f(x :: Vector{Float64}) = x[1]^2 + x[2]^2 + x[3]^2
```

(a declaração explícita do tipo de variável que é `x`, um vetor, não é obrigatória, mas evita que cometamos erros, como o de chamar a função com o tipo de variável errado). Também podemos escrever a função na mesma notação, menos compacta, que estamos usando a maior parte das vezes:

```
function f( x :: Vector{Float64} )
  return x[1]^2 + x[2]^2 + x[3]^2
end
```

Agora, faça o mesmo para o gradiente. Ou seja, em todos os lugares em que você calcula o gradiente, chame uma função, $g(x)$, que retorna o gradiente (um vetor). Feito isto, você vai poder usar a sua função de minimização, chamemos ela de `min` para minimizar qualquer função que tenha sido definida pela função `f` e gradiente `g`. A ideia é poder usar o minimizador assim, por exemplo:

```
f(x :: Vector{Float64}) = x[1]^2 + x[2]^2 + x[3]^2   # Function
g(x :: Vector{Float64}) = [ 2*x[1], 2*x[2], 2*x[3] ] # Gradient
x = [ 5.0, 7.0, -3.0 ] # First guess
precision = 1.d-8
xmin, fmin = min(x,f,g,precision)
```

Atividades

39. Modifique o programa da atividade 37 de acordo com as ideias desta seção, para poder minimizar qualquer função de três variáveis como no exemplo acima.

4 Minimização sem derivadas

A minimização sem derivadas é, conceitualmente, mais simples que a minimização com derivadas. A vantagem é que, naturalmente, nem sempre sabemos calcular a derivada da função que desejamos otimizar, ou a derivada pode nem mesmo existir. Portanto, a otimização sem derivadas é sempre uma maneira de caminhar sobre a função, avaliando diferentes pontos, procurando o máximo ou mínimo da função. Sem saber a derivada, não sabemos para onde a função aumenta ou diminui, portanto, os novos pontos testados serão sempre menos inteligentes que quando conhecemos as derivadas. Naturalmente, isto é muito pior. Não saber para onde a função aumenta é uma grande desvantagem e, portanto, sempre que for possível usar derivadas, elas devem ser usadas.

O método de otimização sem derivadas mais simples consiste em testar, aleatoriamente, novos pontos, e ficar com o melhor.

4.1 Gerador de números aleatórios

Nos métodos de otimização sem derivadas, e em muitas outras situações, vamos precisar gerar números aleatórios. Todas as linguagens possuem alguma função que gera números que *parecem* aleatórios. Em Julia, o gerador de números aleatórios se usa da seguinte maneira:

```
julia> rand()
```

O número real aleatório terá um valor aleatório, no intervalo entre 0 e 1. Uma segunda chamada, em seguida, da mesma subrotina, vai gerar *outro* número aleatório entre 0 e 1.

Atividades

40. Faça um laço que gere vários números aleatórios em sequência, e observe os valores gerados.

41. Você pode gerar muitos números aleatórios usando `rand(100)`, por exemplo. Faça um programa que gere milhares de valores aleatórios, e faça um gráfico destes valores, para verificar, visualmente, a natureza de sua aleatoriedade.

4.2 Minimizando $x^2 + y^2$

Vamos fazer alguns programas para minimizar a função $x^2 + y^2$, sem usar derivadas, com diferentes graus de sofisticação. O primeiro programa é muito simples, e consiste em testar pontos aleatoriamente. O fundamental

aqui é *preservar o melhor valor*. Não há escolha de como parar o programa a não ser por excesso de tempo. Isto é, a busca pelo minimizador para quando cansamos de fazer a busca.

```
# Function to be minimized
f(x::Vector{Float64}) = x[1]^2 + x[2]^2
# Minimizer by random search
function randomsearch(f,ntrial)
  fbest = 1.e30
  x = Vector{Float64}(undef,2)
  xbest = Vector{Float64}(undef,2)
  for i in 1:ntrial
    x[1] = -10. + 20. * rand()
    x[2] = -10. + 20. * rand()
    fx = f(x)
    if fx < fbest
      fbest = fx
      xbest[1] = x[1]
      xbest[2] = x[2]
      println(i," New best point: ", x," f(x) = ", fx)
    end
  end
  println(" Best point found: ",xbest," f = ", fbest)
end
```

[Clique para baixar o código]

Este código pode ser executado com:

```
julia> include("./randomsearch.jl")
julia> randomsearch(f,1000)
```

Note as seguintes características do programa: 1) Serão testados `ntrial` pontos (x, y). O número de testes é definido pelo usuário, quando chama a função `randomsearch()`. 2) Estes pontos são gerados de tal forma que x e y ficam no intervalo $[-10, +10]$, porque o número aleatório `random` é um número entre 0 e 1. 3) Salva-se sempre o melhor ponto. 4) O valor da função no melhor ponto foi inicializado como um número muito grande 10^{30}. Poderíamos ter gerado um primeiro ponto antes do loop e calculado o valor da função nesse ponto também.

Atividades

42. Teste diferentes valores de `ntrial` e observe a precisão do resultado. Tente observar quanto tem que aumentar `ntrial` para melhorar em uma ordem de grandeza a precisão da solução.

Agora vamos sofisticar um pouco a nossa estratégia. Em lugar de gerar um ponto aleatório em cada tentativa, vamos *perturbar* o melhor ponto. Fazemos isso da seguinte forma, por exemplo:

```
x[1] = xbest[1] + 1.e-3*(-1.e0 + 2.e0 * rand())
x[2] = xbest[2] + 1.e-3*(-1.e0 + 2.e0 * rand())
```

Neste caso, o novo ponto, `x` é gerado de tal forma que cada uma de suas componentes está no intervalo $\pm 10^{-3}$ em torno do melhor ponto. Note que, neste caso, como cada ponto teste é gerado como uma perturbação do melhor ponto, o melhor ponto `xbest` precisa ser inicializado antes do loop. Caso isto não seja feito, `xbest`

será automaticamente inicializado como $(0, 0)$ o que, neste caso, é uma trapaça, porque esta é a solução. É razoável, portanto, que o chute inicial seja passado também, como um vetor de duas componentes, para a a função.

Atividades

43. Modifique o programa anterior para usar a nova estratégia.

44. Varie o tamanho da perturbação e observe a precisão da solução obtida. Compare com a precisão do método totalmente aleatório, quando muitas tentativas são feitas (umas cem mil).

45. A perturbação até aqui teve tamanho constante. Será que você consegue fazer algo melhor que isso? Dica: pense no que fazer quando você está longe ou perto da solução. Como adivinhar se estamos perto ou longe?

4.3 O método Simplex

Há uma variedade de métodos que não usam derivadas explicitas, mas que não dependem tanto de variáveis aleatórias. Um dos métodos mais comuns é o *Simplex*, ou *Nelder-Mead* (há mais de um algoritmo que se chama Simplex, e isto pode causar confusão). Aqui faremos uma implementação simples do método. Pouco a pouco, vamos ver que os métodos podem ser muito mais sofisticados, mas que não é nosso propósito implementá-los. Mais tarde buscaremos as subrotinas prontas, feitas por outra pessoa. Ainda assim, implementar casos simples uma vez na vida nos ajuda a entender o que fazem e o que precisam estas rotinas mais sofisticadas.

O método Simplex se baseia no conhecimento adquirido sobre a função depois de avaliações em diversos pontos. Ilustramos isso na figura abaixo. A figura mostra as curvas de nível da função $x^2 + y^2$, e admitimos que calculamos o valor da função em três pontos, possivelmente aleatórios, $\vec{x}_1$, $\vec{x}_2$ e $\vec{x}_3$. De acordo com a

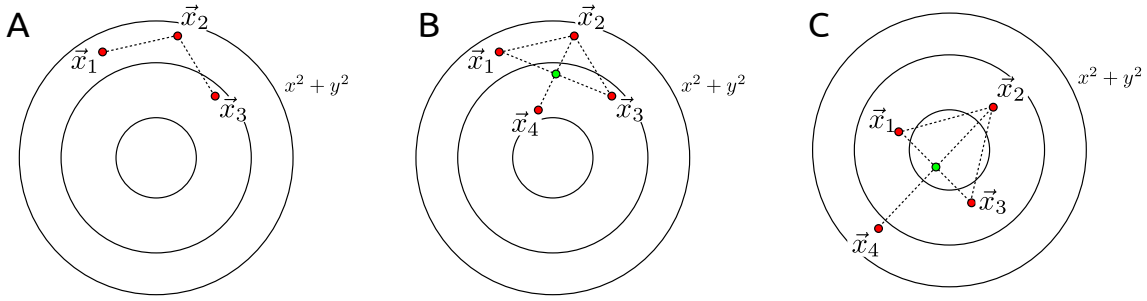

Figura 1: Representação do método simplex de minimização sem derivadas. (A) Pontos iniciais. (B) Ponto $\vec{x}_4$ gerado na direção de descida sugerida pelos pontos anteriores. (C) Ponto $\vec{x}_4$ é insatisfatório.

Figura 1A, o valor da função em $\vec{x}_1$ é menor que em $\vec{x}_2$, e o valor da função em $\vec{x}_3$ é o menor de todos. $\vec{x}_2$ é o pior dos três pontos. Ou seja, a função parece diminuir nas direções $\vec{d}_{21} = \vec{x}_1 - \vec{x}_2$ e $\vec{d}_{23} = \vec{x}_3 - \vec{x}_2$.

Agora vamos calcular a função em um novo ponto $\vec{x}_4$. Onde deve ser escolhido esse novo ponto? Ou seja, para onde devemos *andar* sobre a superfície da função, em função do conhecimento adquirido na avaliação dos pontos anteriores?

O método simplex consiste em calcular o ponto médio da posição dos $N-1$ melhores pontos (neste caso, $N = 3$), e espelhando o pior ponto em relação a esse ponto médio. Este procedimento está ilustrado na Figura 1B. O novo ponto, na Figura 1B, é o ponto $\vec{x}_4$. Formalmente, portanto, a implementação do simplex em duas variáveis consiste em:

1. Ordenar os pontos de melhor a pior valor de função. Digamos, para exemplificar, que a ordem é, como na Figura 1A, $f(\vec{x}_2) > f(\vec{x}_1) > f(\vec{x}_3)$.
2. Calcular $\vec{x}_{\text{médio}} = \frac{1}{2}(\vec{x}_1 + \vec{x}_3)$
3. Espelhar $\vec{x}_2$ em relação a $\vec{x}_{\text{médio}}$. Ou seja, calcular $\vec{x}_4 = \vec{x}_2 + 2(\vec{x}_{\text{médio}} - \vec{x}_2)$
4. Voltar ao passo 1.

Este novo ponto $\vec{x}_4$ está representado na Figura 1B. Podemos ter sorte e a função efetivamente diminuir nesse novo ponto em relação a algum dos pontos anteriores (como acontece no desenho, onde ele é o melhor de todos). Neste caso, o novo ponto é aceito, substituindo o pior dos três pontos anteriores, e recomeça-se o procedimento.

No entanto, como mostra a Figura 1C, o novo ponto gerado pode ser pior que todos os anteriores. Não faz sentido então substituir nenhum dos pontos. Neste caso, parece ser que passamos por cima de um vale (na Figura 1C é efetivamente assim). Por isso, sugere-se fazer uma "busca linear" ao longo da linha definida pelos pontos $\vec{x}_2$ e $\vec{x}_4$. Por exemplo, testando pontos da forma

$$\vec{x}_5 = \vec{x}_2 + \gamma(\vec{x}_4 - \vec{x}_2)$$

onde $0 < \gamma < 1$. No nosso exemplo, vamos fazer uma busca pouco sofisticada. Vamos simplesmente gerar *até* 10 pontos aleatórios, nessa linha, e se conseguirmos um ponto melhor que $\vec{x}_2$, aceitamos esse ponto e voltamos ao começo. Se não conseguirmos, decretamos todo o processo como terminado. Além disso, vamos parar se a diferença de função entre os três pontos em uma iteração é menor que uma precisão desejada.

O código deste programa vai ser o maior que colocaremos aqui, diretamente, no tutorial. Tente entender o código em função da descrição acima, e tire todas as suas dúvidas com o professor. As próximas etapas consistirão em substituir as funções neste código por coisas mais interessantes. Em seguida, pararemos de reinventar a roda, porque há pessoas que escreveram códigos melhores que os nossos, e nós podemos utilizar esses códigos sempre que for conveniente.

O código do método simplex, como descrito acima, é:

```
#
# Simplex minimization
#
function simplex(f,x0,niter)
  # Initial point: x0 contains 3 vectors of dimension 2
  x = copy(x0)
  # Initial function values
  fx = zeros(3)
  for i in 1:3
    fx[i] = f(x[i])
  end
  xtemp = zeros(2)
  ftemp = 0.
  xav = zeros(2)
  xtrial = zeros(2)
  println(" Initial points: ")
  for i in 1:3
    println(x[i]," ",fx[i])
  end
  # Convergence criterium desired
  convcrit = 1.e-10
  # Main interation
  for iter in 1:niter
    println(" ------ ITERATION: ", iter)
    # Order the points from best to worst
    for i in 1:3
```

```
      j = i
      while j > 1 && fx[j-1] > fx[j]
        ftemp = fx[j-1]
        fx[j-1] = fx[j]
        fx[j] = ftemp
        xtemp[1] = x[j-1][1]
        xtemp[2] = x[j-1][2]
        x[j-1][1] = x[j][1]
        x[j-1][2] = x[j][2]
        x[j][1] = xtemp[1]
        x[j][2] = xtemp[2]
        j = j - 1
      end
    end
    # Check convergence
    if (fx[3]-fx[2] < convcrit) && (fx[3]-fx[1] < convcrit)
      println(" Precision reached. ")
      println(" Best point found: ", x[1], " f = ", fx[1])
      return x[1], fx[1]
    end
    # Compute averge of best points
    @. xav = 0.5*(x[1]+x[2])
    # Compute trial point
    @. xtrial = x[3] + 2*(xav-x[3])
    ftrial = f(xtrial)
    # If ftrial is better than fx[3], replace point 3 with trial point
    if ftrial < fx[3]
      fx[3] = ftrial
      @. x[3] = xtrial
      println(" Accepted point: ", x[3]," f = ", fx[3])
    else
      println(" Function increased. Trying line search. ")
      # Try up to 10 different points in the
      # direction x[3]+gamma*(xtrial-x[3])
      for j in 1:10
        @. xtemp = x[3] + rand() * (xtrial - x[3])
        ftemp = f(xtemp)
        if ftemp < fx[3]
          fx[3] = ftemp
          @. x[3] = xtemp
          println("   Line search succeeded at trial ", j)
          println("   New point: ", x[3], " f = ", fx[3])
          break # exits from line search loop
        end
      end
      # If the line search didn't find a better point, stop
      if ftemp > fx[3]
        println(" End of search. ")
        println(" Best point found: ", x[1], " f = ", fx[1])
        return x[1], fx[1]
      end
    end
  end
  println(" Maximum number of trials reached. ")
  println(" Best point found: ", x[1], " f = ", fx[1])
  return x[1], fx[1]
end
```

[Clique para baixar o código]

Este código pode ser executado com:

```
include("./simplex.jl")

# Function to be minimized
```

```
f(x::Vector{Float64}) = x[1]^2 + x[2]^2

# Initial point 3 vectors of dimension 2
x0 = [ Vector{Float64}(undef,2) for i in 1:3 ]
for i in 1:3
  x0[i][1] = -10. + 20. * rand()
  x0[i][2] = -10. + 20. * rand()
end

# Minimize
niter = 1000
simplex(f,x0,niter)
```

[Clique para baixar o código]

Atividades

46. Modifique o programa para minimizar a função $x^2 + \text{sen}\ (y)$.

47. Modifique o programa para minimizar a função $x^2 + y^2 + z^2$. Cuidado com as dimensões dos vetores, inicializações, etc!

48. Na forma como foi descrito e implementado, o novo ponto testado está sempre a uma distância dos pontos anteriores semelhante às distâncias entre eles. Você pode ter notado isso na lentidão em que o método converge quando a função está próxima da solução. Modifique o ponto inicial de tal forma que, desde o início, os três vetores gerados estejam próximos, e observe o comportamento.

49. A observação anterior mostra que, se os pontos são próximos, o caminhar sobre a função pode ser muito lento. Nada nos impede de tentar algo mais ousado. Pense o que poderia ser modificado no método para admitir um caminhar mais rápido sobre a superfície da função. Tente implementar suas ideias.

Atividades

50. Este método depende da ordenação dos pontos de melhor a pior, do ponto de vista do valor da função. O método implementado aqui se chama "método da inserção", e é um dos mais simples. Entenda o que o método faz.

51. Separe o algoritmo de ordenação do resto do programa, na forma de uma função que ordene os vetores e os valores das funções.

52. Há métodos de ordenação que já vêm implementados em Julia, e são chamados pelo comando `sort`. Use este comando para ordenar um vetor de três elementos.

53. No simplex, a ordenação é um pouco mais complicada, porque você está ordenando os pontos na ordem do valor da função que eles assumem. Isso pode ser feito de diferentes formas, por exemplo:

```
sort!(x0,by=x->f(x))
```

    Ordene seu ponto inicial de acordo com o valor da função em cada ponto desta forma.

54. O problema do método acima é que ele exige calcular a função em todos os pontos para fazer o ordenamento, mas no nosso programa já calculamos estes valores antes. Para evitar isso, precisamos usar uma estratégia mais complicada, que consiste em determinar a ordem e, em seguida reordenar o vetor:

```
order = sort([1,2,3],by=i->fx[i])
x = x[order]
fx = fx[order]
```

    Teste este método, e entenda o que ele faz.

5 Aplicando a otimização a um problema "real"

Nas primeiras partes deste tutorial, aprendemos a fazer uma simulação de uma cinética química. Estas simulações, foi discutido, podem ser úteis para validar propostas cinéticas, por comparação com resultados experimentais. Por exemplo, seja a nossa reação

$$\mathrm{A} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} \mathrm{B}.$$

A representação acima é uma proposta para o mecanismo dessa reação, que deve ocorrer em uma etapa em ambos os sentidos. Suponha que essa reação foi estudada experimentalmente, o que significa obter, ao longo do tempo, as concentrações de A e B. Neste caso, ao obter uma concentração, obtemos as duas, por balanço de massa. Queremos, então, verificar se o mecanismo proposto é razoável e determinar as constantes de velocidade. Em um caso simples como este, em que a fórmula analítica da concentração em função do tempo

é conhecida,

$$C_A(t) = \frac{k_1 - k_{-1}e^{-(k_1+k_{-1})t}}{k_1 + k_{-1}} C_A(0)$$

basta ajustar essa fórmula aos dados experimentais. Isto pode ser feito com qualquer programa de ajustes não-lineares. Se o ajuste for bom, acreditamos que o mecanismo deve ser correto e, entre os parâmetros do ajuste, teremos as constantes de velocidade.

Se, por outro lado, a reação é um pouco mais complicada (como no exemplo da ozônio atmosférico, que ilustramos antes), não há uma solução analítica para a cinética reacional. Desta forma, só é possível comparar os dados experimentais com a proposta cinética pela realização de simulações. Uma vez feita uma simulação, podemos comparar as concentrações medidas em cada momento do tempo, com as concentrações previstas pelas simulações nos mesmos tempos. Assim, da mesma forma como se faz com um ajuste, comparamos a validade do mecanismo proposto.

No entanto, para fazer a simulação, precisamos das constantes de velocidade. Em princípio podemos não saber quanto valem. Aliás, este é o caso mais interessante, em que é necessário validar o mecanismo e, ao mesmo tempo, determinar estas constantes. Neste caso, portanto, é necessário:

1. Fazer várias simulações com constantes de velocidade distintas.
2. Comparar o resultado (concentrações em função do tempo) em cada simulação, com o dado experimental.
3. Verificar se há um conjunto de constantes de velocidade que explica bem o resultado experimental, dando suporte ao mecanismo proposto.

Naturalmente, a forma inteligente de fazer essas várias simulações, com diferentes constantes de velocidade, não é testar qualquer coisa. A forma inteligente é usar um *método de otimização* para *variar as constantes de velocidade* de forma a *minimizar a discrepância* entre o resultado da simulação e o resultado experimental.

O que vamos fazer aqui é transformar aquele programa Simplex, que minimiza uma função sem graça, em um programa que minimiza a discrepância entre um resultado de uma simulação e um dado "experimental", de tal forma a obter as constantes de velocidades corretas.

5.1 Resultado experimental

O "resultado experimental", neste caso, será o resultado de uma simulação feita com seu programa que simula a reação unimolecular reversível (programa `sim2` da Seção 2). Escolha concentrações iniciais para os compostos A e B e constantes de velocidade, e faça uma simulação. Estas concentrações iniciais você vai usar como parâmetros de entrada no programa que faremos a seguir, porque é um parâmetro controlado pelo pesquisador. As constantes de velocidade nós vamos *determinar*, como se fossem parâmetros desconhecidos.

Atividades

55. Escolha um conjunto de parâmetros (concentrações iniciais e constantes de velocidade), e execute o programa `sim2` da Seção 2. Guarde o arquivo de saída, que contém as concentrações em função do tempo.

5.2 Comparação com a simulação

Das simulações acima, você vai ter uma lista de concentrações para diferentes tempos. Como nossa reação, aqui, é simples, vamos nos concentrar na concentração de um dos reagentes, C_A. A simulação do item anterior gerou uma série de valores de para C_A, para diferentes instantes do tempo. Esqueça que esses

valores vieram de uma simulação, e imagine que foram obtidos experimentalmente. De agora em diante, chamaremos esses valores de concentrações experimentais. As concentrações experimentais formam uma lista, ($C_A[1]$,$C_A[2]$,....), em que $C_A[N]$ é a concentração de A no N-ésimo instante de tempo (a diferença de tempo entre duas concentrações consecutivas depende do passo de tempo da sua simulação acima).

Agora, vamos fazer uma simulação da mesma reação (lembre-se, os dados que temos são *experimentais*), e comparar com os dados experimentais. Para isso, vamos definir uma função de comparação, que é a soma do quadrado das diferenças entre os dados experimentais e os simulados:

```
function f(Asim,Aexp)
  f = 0.d0
  for i in 1:length(Aexp)
    f = f + (Aexp[i]-Asim[i])^2
  end
  return f
end
```

onde `Aexp` é um vetor que contém os dados experimentais, e `Asim` é o resultado da simulação. Ou seja, para cada um dos `length(Aexp)` instantes de tempo em que há medidas experimentais, calculamos o quadrado diferença da previsão da simulação com o dado experimental. Somamos todas essas diferenças para avaliar quão parecidos são os dois conjuntos de dados.

Atividades

56. Para ler os dados de um arquivo, a forma mais prática é usar o pacote `DelimitedFiles`, da seguinte forma:

    ```
    using DelimitedFiles
    data = readdlm("file.dat")
    x = data[:,1]
    y = data[:,2]
    ```

    Aqui, data é uma matriz que tem tantas colunas quantas houver no arquivo. O comando `data[:,1]` retorna todas as linhas da primeira coluna de `data`. Tente ler um arquivo de dados usando este método, e veja o conteúdo dos vetores `x` e `y` resultantes. Certifique-se que é o que você esperava.

57. Faça um programa que leia os resultados "experimentais", faça uma simulação com outro conjunto de constantes de velocidades (mesmas concentrações iniciais), e calcule a função acima. Reporte a similaridade entre o resultado da simulação e o resultado experimental. Faça um teste usando o conjunto de constantes de velocidade *corretos*, para validar seu programa.

5.3 Descobrindo as constantes de velocidade

O programa que você fez na seção anterior deve ler os resultados experimentais, fazer uma simulação, e calcular a similaridade dos dois resultados. Ou seja, dado um conjunto de constantes de velocidade, avalia a qualidade da sobreposição dos dados experimentais com a simulação. Esta qualidade de sobreposição será a nossa *função objetivo*. Ou seja, voltando aos nosso métodos de otimização, tudo que envolve este último programa será objeto da subrotina que calcula a função.

Atividades

58. Transforme o seu programa, acima, em uma função, que se chame `computef`.

59. Substitua a subrotina que calcula a função no programa Simplex da Seção 4.3. Cuidado com os nomes das variáveis, porque as constantes de velocidade tomarão o lugar do vetor `x`.

60. Seu programa consegue descobrir as constantes de velocidade corretas, partindo de chutes aleatórios?

Note que, agora, a *avaliação da função envolve uma simulação*. Isto é bastante sofisticado.

5.4 Refinamentos do programa

5.4.1 Evite ao máximo ler e escrever no disco

Se você não foi mais esperto do que o previsto, seu programa da seção anterior deve estar lendo os dados experimentais todas as vezes que chama a função `computef`. Ler o arquivo do disco rígido todas as vezes que a função é calculada é muito ruim. Provavelmente seu programa demora mais tempo fazendo isso que calculando outras coisas. Nesta seção, vamos mostrar como fazer isso melhor, e aproveitar para introduzir outras estruturas de programação que são fundamentais em programas mais complexos.

Nós queremos que o programa leia só uma vez os dados experimentais. Portanto, essa leitura não pode estar dentro da função `computef`, que é *chamada* muitas vezes. A leitura deve estar no programa principal. A leitura de dados é alguma coisa da forma

```
using DelimitedFiles
data = readdlm("sim2.dat")
t = data[:,1]
Aexp = data[:,2]
```

A leitura preenche o vetor `Aexp`, que contém as concentrações do reagente A para cada instante experimental. Esse mesmo vetor deve ser criado antes de todo o processo, ficando disponível como variável global para a função `computef`, sem precisar ser novamente preenchido todas as vezes que esta função for chamada.

Atividades

61. Modifique o seu programa da atividade anterior de tal forma que a leitura dos dados seja feita uma única vez.

O programa com todas as modificações desta seção, está disponível abaixo. Tente fazer tudo antes de ver o resultado. Seu programa não vai ficar igual a este, e você vai errar várias vezes antes de chegar a algo que funciona. Isso é normal, e faz parte do aprendizado, e não só do aprendizado, da própria natureza da programação, mesmo para quem já tem experiência. A disponibilidade do código serve principalmente para tirar dúvidas, e aprimorar soluções.

[Clique para baixar o código]

5.4.2 Evitando variáveis globais e organizando os dados

No programa anterior usamos variáveis globais (`Aexp`, `CA0`, `CB0` e `dt` para passar as informações para a função `computef`. Isto não é o ideal, porque o compilador não tem como saber os tipos das variáveis globais

prejudicando muito a eficiência do código. Há, portanto, outras maneiras de passar a informação para a função que não depende de variáveis globais.

A forma mais simples é passar as variáveis como parâmetros. Isto é, em vez de definir

```
function computef(x)
```

definimos a função de tal forma que ela recebe tudo o que precisa como parâmetros, neste caso

```
function computef(x,Aexp,CA0,CB0,dt)
```

O chato de fazer isto é que você terá que modificar o código do minimizador simplex para 1) passar essas variáveis a mais para ele como parâmetros e 2) modificar todas as chamadas de `f(x)` dentro do simplex para a forma que recebe os parâmetros.

Atividades

62. Modifique a função e o programa simplex para passar todos os dados como parâmetros. Teste. O resultado deve ser o mesmo e, talvez, seja possível perceber que ficou mais rápido.

Naturalmente, existem formas melhores de fazer isso: as estruturas de dados e os despachos múltiplos. Estruturas de dados são elas mesmas variáveis, mas que contém vários dados juntos. São definidas assim:

```
struct Data
  A :: Int64
  B :: Vector{Floa64}
end
```

É uma convenção que o nome da estrutura seja com maiúscula (mas não é obrigatório). Dentro dela temos a lista das variáveis e seus tipos. Uma estrutura serve, justamente, para que agrupemos os dados que precisamos. No nosso problema, teremos:

```
struct Data
  Aexp :: Vector{Float64}
  CA0 :: Float64
  CB0 :: Float64
  dt :: Float64
end
```

Uma vez definida a estrutura, você pode criar uma variável que contenha todos os dados da estrutura, usando

```
data = Data(Aexp,CA0,CB0,dt)
```

O usuário pode modificar a estrutura, então, sem modificar nem a função nem o simplex. Para se referir aos dados da estrutura, usa-se a seguinte notação (no caso em que a variável se chama `data`):

```
data.CA0
```

```
data.CB0
data.Aexp[1]
data.dt
...
```

Atividades

63. Defina uma estrutura que tenha o nome (tipo `String`), idade e altura de uma pessoa, inicialize-a com seus dados, e teste o acesso às informações que ela contém.

A grande vantagem disto é que, agora, em vez de passar cada dado como parâmetro para a função `computef` e para o simplex, você passa a estrutura inteira:

```
function computef(x,data)
```

Atividades

64. Modifique o programa anterior para receber, em vez de todos os dados separados em cada chamada, receba apenas uma variável chamada `data`.

Atividades

65. Defina uma estrutura com seus dados, inicialize uma variável com os dados, e execute o programa com a nova forma de passar os dados.

Este tipo de organização vai ser muito comum na linguagem, quando você for usar programas feitos por outras pessoas. No entanto, há um refinamento adicional, bastante característico da linguagem Julia, que faz o trabalho ainda mais simples, aproveitando a capacidade da linguagem de definir funções com mesmo nome e métodos diferentes.

Retomemos o programa simplex anterior, que não recebia os dados como parâmetros, e no qual as chamadas à função eram apenas da forma `f(x)`. Imagine que não queremos modificar mais este programa, mas nossa função requer os parâmetros. Podemos resolver este problema definindo uma nova função, `f(x)`, que consista em chamar a outra função, mas com os parâmetros:

```
f(x) = f(x,Aexp,CA0,CB0,dt)
```

ou

```
f(x) = f(x,data)
```

Com qualquer uma destas alternativas, as chamadas dentro do programa simplex vão chamar a função só com um parâmetro, que por sua vez chama a função com vários parâmetros. O programa sabe qual versão da função chamar justamente pelo número e tipo dos parâmetros.

Atividades

66. Modifique o programa da atividade 61 usando esta forma de passar os parâmetros. Lembre-se de eliminar as variáveis globais.

Justamente porque o reconhecimento de qual método usar é definido pelo tipo de parâmetro da chamada, é possível definir explicitamente qual o tipo de cada parâmetro. Por exemplo:

```
f( x :: Vector{Float64}, data :: Data ) = x + data.CA0 + 2
f( x :: Vector{Float64} ) = f(x,data)
```

Assim, quando `f` for chamada apenas com um vetor de números reais, a segunda versão da função vai ser chamada. Se `f` for chamada com um vetor mais uma variável do tipo `Data`, a primeira versão vai ser usada. Note, também, que a estrutura `Data` é um tipo de variável como qualquer outro neste contexto.

Atividades

67. Verifique o que acontece se você chama a função com dados que não correspondem a nenhum dos conjuntos de tipos dos métodos que foram definidos. A definição obsessiva dos tipos é muito importante, em geral, para evitar erros na programação e no uso das funções.

5.5 Usando uma subrotina pronta

O último grande passo em termos de fundamento de programação que devemos dar é o uso de uma função que foi escrita por outra pessoa. Neste caso, vamos usar uma função que implementa o método Simplex com todo o cuidado, e com variações que o fazem mais eficientes. Há milhares de funções de cálculo numérico programadas por outras pessoas que podem ser utilizadas gratuitamente. A maior parte destas funções foi programada originalmente em Fortran ou C, mas as linguagens de alto nível possuem pacotes que implementam interfaces relativamente fáceis de usar. O pacote de otimização que implementa o método Simplex (Nelder-Mead) para Julia se chama `Optim`. É fácil descobrir isso simplesmente procurando "Nelder-Mead Julia"no google. A estratégia é válida para outros problemas, quase todo algoritmo sofisticado já foi implementado por alguém e está disponível em algum pacote. O trabalho passa a ser entender exatamente como usar a função.

Para instalar o pacote `Optim`, use os mesmos comandos que usamos para instalar o `Plots`:

```
using Pkg
Pkg.add("Optim")
using Optim
```

A documentação do Nelder-Mead para Julia do pacote `Optim` está disponível em

http://julianlsolvers.github.io/Optim.jl/v0.9.3/algo/nelder_mead/

Frequentemente, como é o caso, a documentação não é muito simples, e poderia ser melhor. Mas o uso da função `optimize` disponibilizada pelo pacote é, de fato, muito simples, basta definir a função, um chute inicial, e definir o método de otimização:

```
using Optim
f(x) = x[1]^2 + (x[2] - 3.)^2
```

```
x0 = [ 8., 17. ]
optimize(f, x0, NelderMead())
```

Atividades

68. Crie diferentes funções, com diferentes números de variáveis, e teste o método acima.

69. Use o método para obter as constantes de velocidade da reação química que estudamos, substituindo o seu simplex por esta implementação.