Fundamentos Computacionais de Simulações em Química

Leandro Martínez
leandro@iqm.unicamp.br

Soluções de atividades selecionadas

Atividade 25

O loop do programa que propaga as concentrações usando a discretização das equações diferencias deve ser algo como:

```
for i in 1:nsteps
  CA[i] = CA[i-1] - k1*CA[i-1]*dt + km1*CB[i-1]*dt # Concentration of A
  CB[i] = CB[i-1] + k1*CA[i-1]*dt - km1*CB[i-1]*dt # Concentration of B
  error = ( CA + CB ) - ( CA0 + CB0 ) # Testing balance of mass
  time[i] = time[i] + dt
end
```

Dentro deste loop, deve ser adicionado um teste sobre o erro, que deve sair do loop e escrever uma mensagem de erro caso a diferença da soma das concentrações com relação às concentrações iniciais seja muito grande:

```
...
println(time," ",CA," ",CB," ",error)
if  error > 1.e-3
  println(" ERROR: Balance of mass failed. error = ", error)
  break
end
...
```

O `break` acima vai terminar o loop, e o programa termina depois disto. Poderíamos, também, sair da função diretamente, usando `return`. No exemplo, escolheu-se um erro de 10^{-3} como erro máximo tolerável. Naturalmente, o erro máximo tolerável depende do problema.

Atividade 30

No código abaixo, foram introduzidas duas variávies, `i` e `ntrial`, e o loop foi modificado para dar no máximo `ntrial` voltas. Além disso, removemos a parte do código que parava o programa no caso de aumento do valor da função. Compare com o programa `min1`.

```
function atividade21(x0,deltax,ntrial)
  x = x0
  xbest = x # Save best point
  fbest = x^2 # Best value up to now
  println(" Initial point: ",xbest," ",fbest)
  deltaf = -1.
  for i in 1:ntrial
    # 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(i," Current point: ",x," ",x^2," ",deltaf)
    # If the function decreased, save best point
    if deltaf < 0.
      xbest = x
      fbest = x^2
    end
  end
  return xbest, fbest
end
```

[Clique para baixar o código]

Atividade 33

```
function min2(x0,precision)
  xbest = x0 # Save best point
  fbest = xbest^2 # Best value up to now
  println(" Initial point: ",xbest," ",fbest)
  deltaf = -1.
  deltax = 0.1
  while deltaf < 0.
    # Move x in the descent direction, with step deltax
    dfdx = 2*xbest # Computing the derivative
    if abs(dfdx) < precision
      println(" Critical point found. ")
      println(" xbest = ", xbest, " fbest = ", fbest, " dfdx = ", dfdx )
      return xbest, fbest
    end
    xtrial = xbest - deltax * dfdx # Move x in the -f' direction
    # Compute function value at trial point
    ftrial = xtrial^2
    # If the function decreased, accept trial point and increase step
    if ftrial < fbest
      xbest = xtrial
      fbest = ftrial
      println(" Accepted: ", xbest," ",fbest," ",deltax," ",dfdx)
      deltax = deltax * 2
    else
      println(" Not accepted: ", xbest," ",fbest," ",deltax," ",dfdx)
      deltax = deltax / 2
    end
  end
end
```

[Clique para baixar o código]

Atividade 39

Nesta atividade as rotinas de cálculo da função e do gradiente são separadas da função principal. Quando você chama a função principal, o vetor de entrada x, é enviado apenas pelo seu *endereço na memória*. Isto é, o programa principal diz à função: "trabalhe com este vetor, está neste lugar na memória". Por isso,

para não modificar x, copiamos ele em novos vetores, que vão ser modificados dentro da função (p. ex. `xbest = copy(x)`. Note que `xbest = x` não é a mesma coisa, apenas diria que `xbest` é o vetor `x`. O final, definimos as duas funções e a precisão, chamamos a função é escrevemos o resultado. Todo o processo pode ser executado com

```
julia atividade39.jl
```

ou, de dentro do REPL,

```
include("./atividade39.jl")
```

O código é:

```
function min(x,f,g,precision)
  xbest = copy(x) # Vector to save the best point
  fbest = f(xbest) # Best value up to now
  println(" Initial point: ",xbest," ",fbest)
  xtrial = copy(x)   # Vector of same dimension
  gbest = g(xtrial)  # Compute gradient at initial point
  gnorm = sqrt(gbest[1]^2+gbest[2]^2+gbest[3]^2)
  deltas = 0.1
  while gnorm > precision
    # Move x in the descent direction, with step deltas
    for i in 1:3
      xtrial[i] = xbest[i] - deltas * gbest[i] # Move x in the -f' direction
    end
    # Compute function value at trial point
    ftrial = f(xtrial)
    # If the function decreased, accept trial point and increase step
    if ftrial < fbest
      # Update best point (do NOT use "xbest = xtrial"!)
      for i in 1:3
        xbest[i] = xtrial[i]
      end
      fbest = ftrial
      # Update gradient
      gbest = g(xbest)
      gnorm = sqrt(gbest[1]^2+gbest[2]^2+gbest[3]^2)
      # Print progress and udpate deltas
      println(" Accepted: ", xbest," ",fbest," ",deltas," ",gbest)
      deltas = deltas * 2
    else
      println(" Not accepted: ", xtrial," ",ftrial," ",deltas,xbest)
      deltas = deltas / 2
    end
  end
  println(" Critical point found. ")
  println(" xbest = ", xbest, " fbest = ", fbest, " g = ", gbest )
  return xbest, fbest
end

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

x = [ 5.0, 7.0, -3.0 ]
precision = 1.e-8

xmin, fmin = min(x,f,g,precision)
println("xmin = ", xmin," fmin = ", fmin)
```

[Clique para baixar o código]

Atividade 43

```
# Function to be minimized
f(x::Vector{Float64}) = x[1]^2 + x[2]^2
# Minimizer by random search
function randomsearch2(f,ntrial,x0 :: Vector{Float64})
  x = copy(x0)
  xbest = copy(x0)
  fbest = f(xbest)
  for i in 1:ntrial
    x[1] = xbest[1] + 1.e-3*(-1. + 2. * rand())
    x[2] = xbest[2] + 1.e-3*(-1. + 2. * 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]

Atividades 64 e 65

```
#
# Use the simplex minimizer
#
include("./simplex-struct.jl")

#
# Compute the function value (performs a simulation to do so)
#
function computef(x,data)

  # The constants are given in x
  k1 = x[1]
  km1 = x[2]

  # Number of data points
  ndata = length(data.Aexp)

  # Vector that will contain the simulation data
  Asim = similar(data.Aexp)

  # Initial concentrations
  Asim[1] = data.CA0
  CB = data.CB0

  # Simulate the reaction using the parameters given
  for i in 2:ndata
    Asim[i] = Asim[i-1] - k1*Asim[i-1]*dt + km1*CB*dt
```

```
    CB = CA0 + CB0 - Asim[i]
  end

  # Compute the deviation relative to experimental data
  computef = 0.
  for i in 1:ndata
    computef = computef + ( Aexp[i] - Asim[i] )^2
  end

  return computef

end

# Read experimental data
using DelimitedFiles
data = readdlm("./julia/kineticmodel.dat")
t = data[:,1]
Aexp = data[:,2]
dt = t[2] - t[1] # Time-step
println(" Time-step found: ", dt)

# Initial guess for rate constants, for simplex
# The "experimental" constants in sim2 were k1=0.8 and km1=0.3
k1 = 2.
km1 = 1.
x0 = [ Vector{Float64}(undef,2) for i in 1:3 ]
for i in 1:3
  x0[i][1] = k1 + 0.1*(rand() - 0.5)
  x0[i][2] = km1 + 0.1*(rand() - 0.5)
end

# Initial experimental concentrations
CA0 = 10.
CB0 = 0.

# Define structure
struct Data
  Aexp :: Vector{Float64}
  CA0 :: Float64
  CB0 :: Float64
  dt :: Float64
end
data = Data(Aexp,CA0,CB0,dt)

# Call optimizer
niter = 1000
simplex(computef,x0,niter,data)
```

[Clique para baixar o código]

```
#
# Simplex minimization
#
function simplex(f,x0,niter,data)
  # 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],data)
  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,data)
    # 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,data)
        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]