Fundamentos Computacionais de Simulações em Química

Leandro Martínez
leandro@iqm.unicamp.br

Soluções de atividades selecionadas

Atividade 6

```
program at6
  integer :: i, j
  i = 1
  do while( i > 0 )
    j = i
    i = i + 1
  end do
  write(*,*) ' Greatest integer = ', j
end program at6
```

Comentários: O programa vai naturalmente ser mais rápido se, em lugar de começar com `i=1`, começarmos com um valor próximo do maior inteiro. Aqui são definidas duas variáveis inteiras, `i` e `j`, sendo que `j` assume o valor de `i` *antes* de que este seja modificado. Assim, quando o `i` ultrapassar o maior inteiro representável, e se tornar negativo, o *loop* termina e `j` preserva o valor anterior à última modificação.

Atividade 15

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 18

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`.

```
program atividade18
  implicit none
  integer :: i, ntrial
  double precision :: x, dfdx, deltax, deltaf, xbest, fbest
  deltax = 1.1d0 ! Step size
  x = 10.d0 ! Initial guess
  fbest = x**2
  xbest = x ! Save best point
  write(*,*) ' Initial point: '
  write(*,*) x, x**2
  ntrial = 1000
  do i = 1, ntrial
    ! Move x in the descent direction, with step deltax
    dfdx = 2.d0*x ! Computing the derivative
    x = x - dfdx * deltax ! Move x in the -f' direction
    ! Test new point
    deltaf = x**2 - fbest
    ! Write current point
    write(*,*) x, x**2, deltaf
    ! If the function decreased, save best point
    if ( deltaf < 0 ) then
      xbest = x
      fbest = x**2
    end if
  end do
end program atividade18
```

[Clique para baixar o código]

Atividade 21

```
program min2
  implicit none
  double precision :: x, dfdx, deltax, f
  double precision :: xtrial, ftrial
  deltax = 0.1d0 ! Step size
  x = 10.d0 ! Initial guess
  f = x**2
  write(*,*) ' Initial point: '
  write(*,*) x, f
  write(*,*) ' x, f, deltax, dfdx : '
  do
    ! Move x in the descent direction, with step deltax
    dfdx = 2.d0*x ! Computing the derivative
    ! If the derivative is very small, stop
    if ( dabs(dfdx) < 1.d-10 ) then
      write(*,*) ' Critical point found. '
      write(*,*) ' x = ', x, ' f = ', f, ' dfdx = ', dfdx
      stop
    end if
    ! Computing trial point
```

```
    xtrial = x - 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 < f ) then
      x = xtrial
      f = ftrial
      deltax = deltax * 2.d0
      write(*,*) ' Accepted: ', x, f, deltax, dfdx
    else
      deltax = deltax / 2.d0
      write(*,*) ' Not accepted: ', xtrial, ftrial, deltax, dfdx
    end if
  end do
end program min2
```

[Clique para baixar o código]

Atividade 29

Nesta atividade as rotinas de cálculo da função e do gradiente são separadas do programa principal. Há alguns detalhes importantes: Os vetores são declarados, aqui, com dimensões *fixas* no programa principal, por exemplo, `x(1000)`. Nas subrotinas, declaramos os vetores usando, por exemplo, `x(n)`, sendo `n` um parâmetro de entrada da subrotina. Esta declaração, dentro da subrotina é, na verdade, apenas um lembrete de quantos elementos do vetor vão se usados. O que o programa passa para a subrotina é apenas o *endereço na memória* do vetor. Isto é, o programa principal diz à subrotina: "trabalhe com este vetor, está neste lugar na memória". O tamanho efetivo do vetor é aquele declarado no programa principal, e a declaração na subrotina é, na verdade, redundante. De fato, a mesma declaração `x(n)` poderia ser feita usando `x(1)`, ou mesmo `x(*)`. Note, também, que como critério de parada usamos o quadrado da norma do gradiente.

```
program ativ29
  implicit none
  integer :: i, n
  double precision :: x(1000), g(1000), deltax, f
  double precision :: xtrial(1000), ftrial, gnorm
  deltax = 0.1d0 ! Step size
  ! Number of variables
  n = 1000
  ! Initial guess
  do i = 1, n
    x = 10.d0
  end do
  call computef(n,x,f)
  write(*,*) ' f at initial point: ', f
  do
    ! Compute the gradient
    call computeg(n,x,g)
    ! If the derivative is very small, stop
    gnorm = 0.d0
    do i = 1, n
      gnorm = gnorm + g(i)**2
    end do
    if ( gnorm < 1.d-10 ) then
      write(*,*) ' Critical point found. '
      write(*,*) ' f = ', f, ' gnorm = ', gnorm
      stop
```

```
    end if
    ! Computing trial point
    do i = 1, n
      xtrial(i) = x(i) - deltax * g(i) ! Move x in the -f' direction
    end do
    ! Compute function value at trial point
    call computef(n,xtrial,ftrial)
    ! If the function decreased, accept trial point and increase step
    if ( ftrial < f ) then
      do i = 1, n
        x(i) = xtrial(i)
      end do
      f = ftrial
      deltax = deltax * 2.d0
      write(*,*) ' Accepted: ', f, deltax
    else
      deltax = deltax / 2.d0
      write(*,*) ' Not accepted: ', ftrial, deltax
    end if
  end do
end program ativ29
!
! Subroutine that computes the function value
!
subroutine computef(n,x,f)
  implicit none
  integer :: i, n
  double precision :: x(n), f
  f = 0.d0
  do i = 1, n
    f = f + x(i)**2
  end do
end subroutine computef
!
! Subroutine that computes the gradient
!
subroutine computeg(n,x,g)
  implicit none
  integer :: i, n
  double precision :: x(n), g(n)
  do i = 1 , n
    g(i) = 2.d0*x(i)
  end do
end subroutine computeg
```

[Clique para baixar o código]

Atividade 34

```
program randomsearch2
  implicit none
  integer :: i, ntrial
  double precision :: random, x(2), f, fbest, xbest(2)
  ! Test 10000 points
  ntrial = 10000
  call random_number(random)
  xbest(1) = -10.d0 + 20.d0*random
  call random_number(random)
  xbest(1) = -10.d0 + 20.d0*random
  call computef(xbest,fbest)
```

```
  do i = 1, ntrial
    call random_number(random)
    x(1) = xbest(1) + 1.d-3*(-1.d0 + 2.d0*random)
    call random_number(random)
    x(2) = xbest(2) + 1.d-3*(-1.d0 + 2.d0*random)
    call computef(x,f)
    if ( f < fbest ) then
      fbest = f
      xbest(1) = x(1)
      xbest(2) = x(2)
      write(*,*) i, ' New best point: ', x(1), x(2), ' f = ', f
    end if
  end do
  write(*,*) ' Best point found: ', xbest(1), xbest(2), ' f = ', fbest
end program randomsearch2
!
! Subroutine that computes the function value
!
subroutine computef(x,f)
  implicit none
  double precision :: x(2), f
  f = x(1)**2 + x(2)**2
end subroutine computef
```

[Clique para baixar o código]