Fundamentos Computacionais de Simulações em Química

Leandro Martínez
leandro@iqm.unicamp.br
Última atualização: 14 de setembro de 2018

Sumário

1 Elementos básicos de programação

A linguagem de escolha do curso é Julia, nas suas versões mais recentes (isto é, a partir da versão 1.0). A escolha desta linguagem se deve à simplicidade de sua sintaxe e à eficiência numérica dos programas gerados. Julia é uma linguagem idealizada, desde o princípio, para a computação numérica, mas permite a interação fácil com linguagens muito populares, como Python, Fortran e C, que tem muitas bibliotecas prontas.

O site oficial da linguagem Julia é

http://julialang.org

A documentação apresentada no site é muito detalhada e completa, e pode ser usada como referência permanente. Entre nesse site, e instale o Julia no seu computador.

Há vários cursos, tutoriais e livros sobre Julia. Uma busca na internet por "Introdução a Julia", em qualquer idioma, permite o acesso a uma grande variedade de material gratuito. Este curso, no entanto, não é um curso de Julia. 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. Quando aprendemos a programar em uma linguagem, passar para qualquer outra é sempre bastante fácil.

1.1 Estrutura básica do programa

Antes de nada, tudo o que compuser o 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.

A estrutura mínima dos programas que vamos escrever é:

```
let Program
  # This is a commentary
end
```

Todas as linhas que começam com uma *cerquilha* (o hashtag), "#", são ignoradas, sendo chamadas de "comentários". Os comentários ajudam, em programas complexos, a entender o que está sendo feito. Nossos programas vão começar sempre com esse `let Program` e terminar com esse `end`. Isto não é obrigatório, mas muito mais adiante vamos explicar esta escolha. Não se preocupe com isto agora, entenda que isto apenas define que isso define onde o programa começa e termina.

Este código pode ser editado em qualquer editor de texto básico (como o Notepad, Vim, etc. Uma forma de trabalhar, no entanto, é usando um editor especializado, como o Juno:

`http://junolab.org/`

Instale este editor em seu computador também, se quiser.

Vai ser fundamental escrever o resultado dos programas "na tela". Em Julia o comando mais útil para isto é o `println` (de "imprimir em uma linha"):

```
let Program
  # This is a commentary
  println("test")
end
```

1.1.1 Loops

Uma das estruturas mais fundamentais em programação são os *loops*. Há diferentes maneiras de escrever um loop em Julia, sendo a mair comum a que usa o comando "para ... em ... ",

```
a = 0
for i in 1:3
  a = a + 1
end
```

Atividades

1. Você já deve ter verificado o que acontece com um número inteiro quando tenta-se ultrapassar o valor máximo representável. Use a estrutura do loop acima para escrever o inteiro máximo representável.

Loops podem também ser parados por testes feitos internamente. Por exemplo, a seguinte sintaxe é válida:

```
i = 0
do
  i = i + 1
  if ( i > 10 ) then
    exit
  end if
end do
```

O loop não possui nenhum teste, e só termina quando o o teste realizado internamente se satisfaz. Este tipo de saída de um loop vai ser muito útil nas simulações, em particular para a checagem de erros.

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

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

O programa abaixo faz este procedimento recursivamente, até que a concentração do reagente A se anule. Estude o programa com atenção:

```
let Sim1
  file = open("sim1.dat","w")
  nsteps = 1000 # Number of steps
  dt = 1.e-1 # Time-step
  k1 = 0.1e0 # Velocity constant
  CA = 10.e0 # Initial concentration
  time = 0.e0
  write(file, "# Time    CA \n")
  for i in 1:nsteps
    CA = CA - k1*CA*dt
    time = time + dt
    write(file, "$time $CA \n")
  end
  close(file)
end
```

[Clique para baixar o código]

Aqui criamos pela primeira vez um arquivo de saída para nosso programa. Ele se chama `sim1.dat`, e a ele foi atribuída a "unidade 10". Quando escrevemos o resultado, o `write` foi modificado para escrever o tempo e a concentração na "unidade 10", ou seja, nesse arquivo. Esse arquivo pode ser importado em qualquer programa de fazer gráficos, para vermos o que está acontecendo.

Atividades

2. Varie os valores de `CA`, `k1` e `dt` no programa acima, e observe o comportamento da concentração em função do tempo em cada caso. Se necessário, varie `nsteps` também.

3. 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 if`" ao programa que detecte esse erro, escreva uma mensagem de erro, e pare o programa.

4. Adicione ao programa o cálculo da concentração do produto B. Assuma que a concentração de B tem um valor inicial especifico. Escreva também a concentração de B no arquivo de saída.

5. Nós sabemos que a solução analítica da equação diferencial deste problema é simplesmente $[\mathrm{A}(t)] = [\mathrm{A}](0)e^{-k_1 t}$. Em Fortran, essa exponencial se escreve como `exp(-k1*t)`. 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`.

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

6. 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, $[\mathrm{A}] + [\mathrm{B}] = [\mathrm{A}]_0 + [\mathrm{B}]_0$. O programa usando o balanço de massa pode ser:

```
program sim2
  implicit none
  integer :: i, nsteps
  double precision :: CA, CB, CA0, CB0
  double precision :: dt, time, k1, km1
  open(10,file='sim2.dat')
```

```
  nsteps = 1000 ! Number of steps
  dt = 1.d-1 ! Time-step
  k1 = 0.1d0 ! Velocity constant
  km1 = 0.05d0 ! Velocity constant
  CA0 = 10.d0 ! Initial concentration of A
  CB0 =  0.d0 ! Initial concentration of B
  time = 0.d0
  CA = CA0
  CB = CB0
  do i = 1, nsteps
    CA = CA - k1*CA*dt + km1*CB*dt
    CB = CA0 + CB0 - CA
    time = time + dt
    write(10,*) time, CA, CB
  end do
end program sim2
```

[Clique para baixar o código]

Entenda bem o programa acima. Note que, agora, salvamos as concentrações iniciais em duas novas variáveis, `CA0` e `CB0`, porque são usadas o tempo todo no cálculo da concentração de B pelo balanço de massa. Um detalhe: no início do programa, adicionamos um comando `implicit none`. Este comando serve para que não esqueçamos de declarar nenhuma variável. Se uma variável, como `CA`, `time`, etc, for usada, mas seu tipo não tiver sido declarado, a compilação do programa vai acusar um erro. Na medida que o programa for ficando grande, isto vai ser importante. Portanto, a partir de agora, todos os nossos programas começarão com esse comando.

Atividades

7. 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.

8. 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.

9. 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

10. 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).

11. 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 a função em si. 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 mínimo valor. 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 simplesmente sobre. 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. O código abaixo implementa esta estratégia (`dabs(x)` é o módulo do número x):

```
program min1
  implicit none
  double precision :: x, dfdx, deltax, deltaf, xbest, fbest
  deltax = 0.1d0 ! Step size
  x = 14.1357d0 ! Initial guess
  fbest = x**2
  xbest = x ! Save best point
  write(*,*) ' Initial point: '
  write(*,*) x, x**2
  do
    ! Move x in the descent direction, with step deltax
    dfdx = 2.d0*x ! Computing the derivative
    x = x - deltax * dfdx/dabs(dfdx) ! 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
    else
      write(*,*) ' Function is increasing. '
      write(*,*) ' Best solution found: x = ', xbest
      exit
    end if
  end do
end program min1
```

[Clique para baixar o código]

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 Equação2). 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

12. Varie o valor de Δx no programa acima, e estude o que acontece. Qual a precisão das soluções atingidas. Que resultados inesperados podem acontecer? Por quê? O que ocorre se o passo Δx for muito grande?

13. Modifique o programa de tal forma que quando a função aumenta, em vez de parar, simplesmente continua-se, mas sem salvar o melhor ponto x. 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.

14. Faça um programa que minimize a função $x^2 + \text{sen}\,(10x)$. A sintaxe para o seno em dupla precisão é `dsin(x)`. Faça testes variando o tamanho de passo e os pontos iniciais, no intervalo $[-2, 2]$.

15. 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 no programa `min1`, e avalie as características do processo de minimização (número de iterações até o fim e precisão da solução).

O passo que faz o algoritmo de otimização mais robusto é o passo que varia de forma inteligente de acordo com o que acontece com a função. 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 < f ) then
  deltax = deltax * 2.d0
  f = ftrial
  x = xtrial
else
  deltax = deltax / 2.d0
end if
...
```

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 dessa função nesse ponto teste. Dependendo de como varia a função, tomamos uma decisão sobre a atualização do ponto corrente x 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.

Atividades

16. Modifique o programa do exercício 15, 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.

17. Compare o programa da atividade anterior com o programa da atividade 15 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 nessa direção,

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

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

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 genéria 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

18. Modifique o programa da atividade 21 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 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, um vetor é declarado usando

```
double precision :: x(2)
```

e, em lugar de usar duas variáveis, `x` e `y`, usaremos as componentes do vetor `x(1)` e `x(2)`.

Atividades

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

4 Subrotinas e 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, foram inventadas as *subrotinas* e as *funções*.

As funções você já conhece. Quando você usa a notação, por exemplo `dabs(x)`, que calcula o módulo de x, você "chamou" a função módulo. Essa função está escrita em algum lugar, e deve ser alguma coisa assim:

```
function dabs(x)
  implicit none
  double precision :: x
  if ( x > 0 ) then
    dabs = x
  else
    dabs = -1.d0*x
  end if
end function dabs
```

Entenda este exemplo. O valor de x é um *parâmetro de entrada* da função. A função *retorna* o valor do módulo de x em `dabs`. Esta função poderia estar escrita no mesmo arquivo em que está o seu programa, depois do fim do programa.

Atividades

22. Faça um programa que, dado um valor de uma variável x, chame uma função que multiplique o valor dessa variável por cinco.
23. Faça um programa que calcule a soma do quadrado dos elementos de um vetor. O vetor pode ser passado como parâmetro da função de uma vez só. Cuidado com as declarações das variáveis, elas devem ser declaradas dentro e fora das das funções, da mesma forma.

O próximo passo é aproveitar esta estrutura de funções e subrotinas para estruturar o nosso programa de minimização. Vamos criar agora subrotinas que calculam o valor da função e o gradiente, em cada ponto testado. A subrotina que calcula a função pode ser (poderia ser uma função também):

```
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
```

Os parâmetros `n` e `x` são *parâmetros de entrada* da subrotina, e `f` é o parâmetro de saída (isto não tem a ver com a a ordem dos parâmetros, é apenas uma observação lógica). A subrotina que calcula o gradiente é:

```
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
```

Estas duas subrotinas podem ser colocadas antes ou depois do programa principal, no mesmo arquivo, ou podem ser colocadas em outros arquivos. No programa principal, em lugar de calcular explicitamente a função ou o gradiente, usa-se

```
call computef(n,x,f}
```

e

```
call computeg(n,x,g)
```

Nos nossos programas calculamos um ponto teste, que chamamos `xtrial`. As subrotinas podem ser *chamadas* usando este vetor, por exemplo,

```
call computef(n,xtrial,f)
```

Ou seja, não é necessário programar explicitamente nenhuma outra vez a função que está sendo minimizada. Basta chamar a subrotina adequada. Note que, com isto, modificar o programa para minimizar uma função diferente passa a ser, essencialmente, mudar a subrotina, e programa principal pode ser sempre o mesmo. Se a subrotina for colocada em um arquivo diferente do programa principal, é necessário compilar tudo junto. Na linha de comando, isto é bastante simples, por exemplo:

```
gfortran -o minimize min2.f90 computef.f90 computeg.f90
```

onde `minimize` é o nome do executável que será gerado, e os arquivos que seguem contém o programa principal e cada uma das subrotinas (no Windows é necessário usar o Prompt de comando e colocar o compilador no $PATH$, peça ajuda se precisar). Note que, se quiser criar um programa para minimizar outra função, basta criar as subrotinas em outros aquivos, com outros nomes, e compilar de acordo.

Atividades

24. Modifique o programa do exercício 26 introduzindo subrotinas para o cálculo da função e do gradiente.

25. Crie subrotinas distintas para o cálculo da função e do gradiente de outras funções, por exemplo, $(x_1 - 1)^4 + (x_2 - 3)^2 + (x_3 + 7)^4$, ou $x_1^4 \text{sen } (x) + x_2^2$. Ou qualquer outra que desejar.

5 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 otimizar a função. Sem saber a derivada, não sabemos para onde a função aumenta ou diminui, portanto, os novos pontos testados serão sempre menos "racionais" 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 usá-las, devem ser usadas.

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

5.1 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 Fortran, o gerador de números aleatórios se usa da seguinte maneira:

```
double precision :: random
call random_number(random)
```

O número real `random` 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

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

27. Faça um programa que gere milhares de valores aleatórios, e faça um gráfico destes valores, para verificar, visualmente, a natureza de sua aleatoriedade.

5.2 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 that computes the function value
#
function computef(x)
  f = x[1]^2 + x[2]^2
  return f
end
#
# Program randomsearch
#
let RandomSearch
  # Test 10000 points
  x = [ 0. , 0. ]
  xbest = x
  ntrial = 10000
  fbest = computef(x)
  for i in 1:ntrial
    x[1] = -10.e0 + 20.e0*rand(Float64)
    x[2] = -10.e0 + 20.e0*rand(Float64)
    f = computef(x)
    if f < fbest
      fbest = f
      xbest = x
      println( i, " New best point: ", x, " f = ", f )
    end
  end
  println(" Best point found: ", xbest, " f = ", fbest)
end
```

[Clique para baixar o código]

Note as seguintes características do programa: 1) Serão testados `ntrial` pontos (x, y). 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

28. 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:

```
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)
```

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.

Atividades

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

30. Varie o tamanho da perturbação e observe a precisão da solução obtida. Compare com a precisão do método totalmente aleatório.

31. A perturbação até aqui teve tamanho constante. Será que você consegue fazer algo melhor que isso?

5.3 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

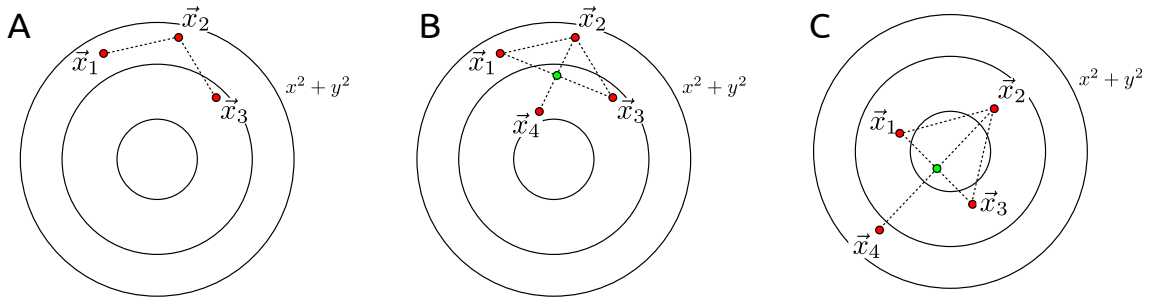

Figura 1: Representação do método simplex de minimização sem derivadas. (A) Pontos iniciais. (B) Ponto $\vec{x}_4$ gerado na direção de descida sugerida pelos pontos anteriores. (C) Ponto $\vec{x}_4$ é insatisfatório.

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 passamo 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 na forma de subrotinas sempre que for conveniente.

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

```
program simplex
  implicit none
  integer :: i, j, iter, niter
  double precision :: random
  double precision :: x(3,2), f(3), ftemp, xtemp(2)
  double precision :: xav(2)
  double precision :: xtrial(2), ftrial
  double precision :: convcrit
  ! Generate initial points
  do i = 1, 3
    call random_number(random)
    x(i,1) = -10.d0 + 20.d0*random
    call random_number(random)
    x(i,2) = -10.d0 + 20.d0*random
    ! Necessary because x in computef is of the form x(2):
    xtemp(1) = x(i,1)
    xtemp(2) = x(i,2)
    call computef(xtemp,f(i))
  end do
  write(*,*) ' Initial points: '
  do i = 1, 3
    write(*,*) x(i,1), x(i,2), f(i)
  end do
  ! Convergence criterium desired
  convcrit = 1.d-10
  ! Maximum number of iterations
  niter = 10000
  do iter = 1, niter
    write(*,*) ' ------ ITERATION: ', iter
    ! Order the points from best to worst
    do i = 1, 3
      j = i
      do while( j > 1 .and. f(j-1) > f(j) )
        ftemp = f(j-1)
        f(j-1) = f(j)
        f(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 do
  end do
  ! Check convergence
  if ( f(3) - f(2) < convcrit .and. f(3) - f(1) < convcrit ) then
    write(*,*) ' Precision reached. '
    write(*,*) ' Best point found: ', x(1,1), x(1,2), ' f = ', f(1)
    stop
  end if
  ! Compute averge of best points
  xav(1) = 0.5d0*(x(1,1) + x(2,1))
  xav(2) = 0.5d0*(x(1,2) + x(2,2))
  ! Compute trial point
  xtrial(1) = x(3,1) + 2.d0*(xav(1)-x(3,1))
  xtrial(2) = x(3,2) + 2.d0*(xav(2)-x(3,2))
  call computef(xtrial,ftrial)
  ! If ftrial is better than f(3), replace point 3 with trial point
  if ( ftrial < f(3) ) then
    f(3) = ftrial
    x(3,1) = xtrial(1)
    x(3,2) = xtrial(2)
    write(*,*) ' Accepted point: ', x(3,1), x(3,2), ' f = ', f(3)
  else
    write(*,*) ' Function increased. Trying line search. '
    ! Try up to 10 different points in the
    ! direction x(3)+gamma*(xtrial-x(3))
    do j = 1, 10
      call random_number(random)
      xtemp(1) = x(3,1) + random*(xtrial(1)-x(3,1))
      xtemp(2) = x(3,2) + random*(xtrial(2)-x(3,2))
      call computef(xtemp,ftemp)
      if ( ftemp < f(3) ) then
        f(3) = ftemp
        x(3,1) = xtemp(1)
        x(3,2) = xtemp(2)
        write(*,*) '   Line search succeeded at trial ', j
        write(*,*) '   New point: ', x(3,1), x(3,2), ' f = ', f(3)
        exit
      end if
    end do
    ! If the line search didn't find a better point, stop
    if ( ftemp > f(3) ) then
      write(*,*) ' End of search. '
      write(*,*) ' Best point found: ', x(1,1), x(1,2), ' f = ', f(1)
      stop
```

```
      end if
    end if
  end do
  write(*,*) ' Maximum number of trials reached. '
  write(*,*) ' Best point found: ', x(1,1), x(1,2), ' f = ', f(1)
end program simplex
!
! Compute 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]

Atividades

32. Modifique o programa para minimizar a função $x^2 + \text{sen}\ (y)$.
33. Modifique o programa para minimizar a função $x^2 + y^2 + z^2$. Cuidado com as dimensões dos vetores, inicializações, etc!
34. 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 programa de tal forma que, desde o início, os pontos gerados estejam próximos, e observe o comportamento.
35. 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

36. 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.
37. Separe o algoritmo de ordenação do resto do programa, na forma de uma subrotina.

6 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 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,

$$[\mathrm{A}] = \frac{k_1 - k_{-1}e^{-(k_1+k_{-1})t}}{k_1 + k_{-1}}[\mathrm{A}_0]$$

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.

6.1 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

38. 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.

6.2 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, [A]. A simulação do item anterior gerou uma séria de valores de para [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, ([A](1),[A](2),....), em que [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:

```
f = 0.d0
do i = 1, N
  f = f + (aexp(i)-asim(i))**2
end do
```

onde `aexp` é um vetor que contém os dados experimentais, e `asim` é o resultado da simulação. Ou seja, para cada uma dos `N` 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

39. 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.

6.3 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

40. Transforme o seu programa, acima, em uma subrotina, que se chame `computef`.

41. Substitua a subrotina que calcula a função no programa Simplex da Seção 5.3. Cuidado com os nomes das variáveis, porque as constantes de velocidade tomarão o lugar do vetor `x`.

42. 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.

6.4 Refinamentos do programa

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 subrotina `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 subrotina `computef`, que é *chamada* muitas vezes. A leitura deve estar no programa principal. A leitura de dados é alguma coisa da forma

```
open(10,file='sim2.dat')
do i = 1, N
  read(10,*) aexp(i)
end do
close(10)
```

A leitura preenche o vetor `aexp`, que contém as concentrações do reagente A para cada instante experimental. Esse mesmo vetor deve ficar disponível dentro da subrotina `computef`, sem precisar ser novamente preenchido todas as vezes que esta subrotina for chamada. Para isso, servem as estruturas dos *módulos*. Os módulos são usados da seguinte forma:

1. Antes do programa principal, define-se um módulo que contém as variáveis de interesse. por exemplo:

```
module reaction
  double precision :: aexp(1000)
end module reaction
```

2. No programa principal, antes da declaração das variáveis, e antes mesmo do `implicit none`, declaramos que vamos *usar* esse módulo:

```
program simplex
  use reaction
  implicit none
  ...
```

As variáveis que foram definidas no módulo $não$ podem ser novamente declaradas no programa principal.

3. Por fim, dentro da subrotina `computef`, que vai precisar desses dados, também declaramos que vamos usar as variáveis definidas no módulo:

```
subroutine computef(x,f)
  use reaction
  implicit none
  ...
```

A leitura dos dados, agora, pode ser feita no programa principal, em qualquer lugar, antes da primeira chamada à subrotina `computef`. Naturalmente, em algum lugar onde não se repita. Esta leitura vai preencher o vetor `aexp`, que ficará disponível para todas as rotinas do programa que usam o módulo `reaction`. O módulo pode conter outros dados da reação, por exemplo, as concentrações iniciais e o passo de tempo, que são necessários para fazer a simulação e é natural que sejam parâmetros definidos pelo usuário no programa principal (ou mesmo lidos de algum arquivo).

Atividades

43. Modifique o seu programa da atividade anterior de tal forma que todos os parâmetros que são definidos apenas uma vez e são necessários para o cálculo da função sejam colocados em um módulo, que é compartilhado pelo programa principal e pela subrotina.

O programa Simplex, 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]

6.5 Usando uma subrotina pronta

O último grande passo em termos de fundamento de programação que devemos dar é o uso de uma subrotina ou função que foi escrita por outra pessoa. Neste caso, vamos usar uma subrotina que implementa o método Simplex com todo o cuidado, e com variações que o fazem mais eficientes. Há milhares de subrotinas de cálculo numérico programadas em Fortran que podem ser utilizadas gratuitamente. Uma busca na internet permite encontrar muitas coisas boas. A subrotina que vamos usar foi encontrada no google mesmo, buscando por "simplex fortran 90". Todo algoritmo relativamente sofisticado já foi implementado por alguém.

A código que vamos usar está no arquivo `NelderMeadMinimizer.f90`, e foi escrito originalmente por David E. Shaw. Por curiosidade, D. E. Shaw era um cientista da computação da Universidade de Columbia, que abandonou a carreira acadêmica para se dedicar a ganhar dinheiro na bolsa de valores usando métodos de otimização de riscos. Ficou bilionário e entediado, e criou uma instituição de pesquisa dedicada à bioquímica computacional, na qual desenvolve computadores especificamente desenhados para simulações de dinâmica molecular (D. E. Shaw Research).

A subrotina que vamos usar está disponível aqui:

http://www.nist.gov/pml/div684/grp03/upload/NelderMeadMinimizer.f90

Para usar uma subrotina precisamos entender o que ela requer como parâmetros de *entrada* e *saída*. Neste caso, você vai notar que o arquivo possui três subrotinas, chamadas `minimize_2d`, `minimize_1d` e `minim`.

A subrotina principal, em que o algoritmo simplex está efetivamente implementado, é a `minim`. Leia com cuidado os comentários o arquivo. Como você vai notar, a subrotina `minim` possui muitos parâmetros, alguns dos quais não temos muita certeza do que são, sem conhecer o algoritmo implementado com maiores detalhes. Por causa desta complexidade, o programador fornece as outras duas subrotinas, `minimize_2d` e `minimize_1d`, que simplificam o uso da subrotina principal em problemas específicos.

Aqui vamos nos concentrar na subrotina `minimize_2d`. Como o nome sugere, está pensada para a minimização de funções de duas variáveis. Nosso problema da seção 6 é um problema de duas variáveis (as duas constantes de velocidade). Vamos, então, usar esta subrotina para resolver aquele problema.

6.5.1 Subrotina `minimize_2d`

A subrotina `minimize_2d` é bem mais simples que a `minim`. Tem apenas cinco parâmetros: `X0, FCN, X, F, Info`. Todos estes parâmetros são fáceis de entender:

• `X0`: é, obviamente, a primeira aproximação das variáveis (o ponto inicial). Você vai ter que fornecer este ponto inicial. Note, na subrotina `minimize_2d`, o que é feito com `X0`.

• `FCN`: como o nome sugere, tem relação com a função. Note que está declarado de uma forma estranha, como `EXTERNAL FCN`. Isso significa que `FCN` não é um vetor ou um número. `FCN` é uma subrotina ou uma função. Você vai ter que fornecer para a subrotina `minimize_2d` o nome de uma subrotina que calcula a função. Isto porque, dentro do método simplex, esta subrotina vai ser usada. Portanto, a rotina que você vai fornecer precisa ser consistente (nos parâmetros de entrada e saída) com a subrotina que o método espera.

Atividades

44. Identifique como o parâmetro `FCN` é passado até a subrotina principal, `minim` (ele muda de nome, inclusive). Em seguida, leia os comentários da subrotina `minim` para entender como tem que ser esta subrotina ou função, do ponto de vista de seus parâmetros de entrada e saída.

45. Retome o programa da seção 6, e veja se sua subrotina de cálculo da função é consistente com o que é necessário para este novo programa. Se não for, ajuste os parâmetros de entrada e saída para que seja (lembre-se, os nomes das variáveis não são importantes, o que importa é o tipo e a ordem em que são passados).

• `X`: Este vetor vai conter a solução encontrada pelo método simplex. Note como ele está declarado em `minimize_2d`, e como ele é passado entre as outras subrotinas.

• F: como o nome sugere, este será o valor da função, ao final da otimização. Confira isso seguindo as passagens entre as subrotinas, e os comentários da rotina `minim`.

• `Info`: Muitas subrotinas possuem uma variável como esta, que indicará se alguma coisa não funcionou como deveria. Note que é um número inteiro, e veja nos comentários de `minim` o significado de cada um dos valores de retorno.

6.5.2 Preparando nosso programa para usar `minimize_2d`

O nosso programa, agora, vai ser bastante mais simples que os anteriores, porque toda a parte complicada está implementada na subrotina que pegamos pronta. Vamos precisar apenas:

1. Declarar corretamente todas as variáveis que serão usadas, de forma consistente com as subrotinas que vamos usar.
2. Adaptar a subrotina que calcula a função, se necessário. O que inclui ler os parâmetros do modelo, da mesma forma que fizemos anteriormente.
3. Gerar um ponto inicial para as variáveis.
4. Chamar a adequadamente a subrotina `minimize_2d` com e verificar qual foi o resultado.

A primeira tarefa consiste, então, em declarar corretamente as variáveis, de acordo com o que a subrotina `minimize_2d` requer. São alguns inteiros e vetores de dupla precisão (`double precision` e `REAL*8` são sinônimos). Além disso, como a subrotina que você vai fornecer para calcular a função vai, também, ser um parâmetro, ela precisa ser declarada, como foi feito na rotina `minimize_2d`. Seu programa vai precisar ter uma declaração do tipo

```
external computef
```

onde `computef` é o nome da subrotina que você programou para calcular a função.

Atividades

46. Copie seu programa se seção 6 em um novo arquivo. Remova tudo o que consistia no método simplex propriamente dito, com seu método iterativo. Coloque a chamada à subrotina minimize_2d, a ajuste todas as declarações de variáveis.

O seu código, que chamarei aqui `modelab.f90` pode ser compilado juntamente com a rotina que você pegou na internet, usando:

```
gfortran -o modelab modelab.f90 NelderMeadMininimizer.f90
```

Tente fazer seu programa funcionar. Com um detalhe: dentro da subrotina `minimize_2d` há um parâmetro chamado `IPrint`, que por padrão está definido como -1. Altere este valor, por exemplo para $+1$, você faz com que mais coisas sejam escritas pela subrotina `minim` no processo iterativo. Isto é interessante para ver o que está acontecendo.

A solução para toda esta atividade está disponível aqui:

[Clique para baixar o código]