Introdução à Física Computacional: Projeto 1

Instituto de Física de São Carlos, Universidade de São Paulo

Solano E. S. Felício, n° USP 10288907

19 de março de 2018

Introdução

Este é um relatório entregue para avaliação da disciplina 7600017 – Introdução à Física Computacional. Cada seção é uma execução direta das tarefas do projeto 1, cujo objetivo é introduzir a linguagem de programação Fortran. A linguagem é comumente utilizada para computação científica e de alta performance. Usaremos no curso a versão Fortran 90.

Incluí os códigos-fonte (ou trechos) relevantes, assim como as tabelas de resultados, diretamente nas discussões dos problemas, destacados em fonte monoespaçada. Usei o compilador gFortran em sistemas GNU/Linux. Por simplicidade, entrada e saída de dados foram feitas por *pipes* e redirecionamentos em linha de comando, em vez de gerenciar arquivos diretamente no código Fortran. Nas tabelas de resultados, adicionei títulos para cada coluna, dividi uma coluna em duas menores (quando necessário) e eliminei espaços em branco para facilitar a visualização, mas nenhuma outra edição foi feita.

Problema 1 – Fatoriais e aproximação de Stirling

Um laço simples resolve a parte (a):

```
f = 1.0d0
do i=1,20
    f = f*dfloat(i)
    write(*,*)i,f
end do
```

#n	n!	n	n!
1	1.0000000000000000	11	39916800.000000000
2	2.0000000000000000	12	479001600.00000000
3	6.0000000000000000	13	6227020800.0000000
4	24.000000000000000	14	87178291200.000000
5	120.00000000000000	15	1307674368000.0000
6	720.00000000000000	16	20922789888000.000
7	5040.0000000000000	17	355687428096000.00
8	40320.000000000000	18	6402373705728000.0
9	362880.00000000000	19	1.2164510040883200E+017
10	3628800.0000000000	20	2.4329020081766400E+018

A partir do penúltimo fatorial, os dígitos finais não foram calculados (estavam além da precisão de 8 bytes usada). Embora aparentemente desvantajoso, usar floats em vez de inteiros é desejável. Veja que o número mínimo de bits necessários para armazenar $n!$ como um inteiro é $\lceil \log_2(n!) \rceil$, o que rapidamente se torna maior que 8 bytes = 64 bits, mesmo antes de $n = 30$ (veja a próxima tabela).

Na parte (b), calculamos os *logaritmos* dos fatoriais, o que requer adições em vez de multiplicações:

```
lf=0.0d0
do i=2,30
    lf = lf + log(dfloat(i))
    write(*,*) i,lf,ceiling(lf/log(2.d0))
end do
```

#n	log(n!)	ceil(log2(n!))
2	0.69314718055994529	1
3	1.7917594692280550	3
4	3.1780538303479453	5
5	4.7874917427820458	7
6	6.5792512120101012	10
7	8.5251613610654147	13
8	10.604602902745251	16
9	12.801827480081471	19
10	15.104412573075518	22
11	17.502307845873887	26
12	19.987214495661888	29
13	22.552163853123425	33
14	25.191221182738683	37
15	27.899271383840894	41
16	30.671860106080675	45

n	log(n!)	ceil(log2(n!))
17	33.505073450136891	49
18	36.395445208033053	53
19	39.339884187199495	57
20	42.335616460753485	62
21	45.380138898476908	66
22	48.471181351835227	70
23	51.606675567764377	75
24	54.784729398112326	80
25	58.003605222980525	84
26	61.261701761002008	89
27	64.557538627006338	94
28	67.889743137181540	98
29	71.257038967168015	103
30	74.658236348830172	108

Facilmente se verifica que os resultados das partes (a) e (b) são consistentes entre si e corretos. Observe também que, já para $n \geq 21$, a tabela acima mostra que $n!$ não pode ser armazenado como inteiro de 8 bytes, justificando o uso dos floats.

Na parte (c), a nova variável real `st` carrega a aproximação de Stirling, enquanto `pi` é o valor de $\pi = 4 \arctan 1$. O código completo é então

```
      program stirling
      implicit none

      integer i
      real*8 lf,st,pi

      parameter(pi = 4*atan(1.0d0))
      lf = 0.0d0
      do i=2,30
          lf = lf + log(dfloat(i))
          st = i*log(dfloat(i)) - i + log(2*pi*i)/2
          write(*,*) i, lf, st, (lf-st)/lf
      end do
      end program stirling
```

#n	log(n!)	stirling	erro relativo
2	0.69314718055994529	0.65180648460453594	5.9642017041767491E-002
3	1.7917594692280550	1.7640815435430568	1.5447344445693184E-002
4	3.1780538303479453	3.1572631582441804	6.5419508962466849E-003
5	4.7874917427820458	4.7708470515922246	3.4767038950857614E-003
6	6.5792512120101012	6.5653750831870310	2.1090741751492960E-003
7	8.5251613610654147	8.5132646511195222	1.3954820843890348E-003
8	10.604602902745251	10.594191637483277	9.8176851669556312E-004
9	12.801827480081471	12.792572017898756	7.2297976184380495E-004
10	15.104412573075518	15.096082009642156	5.5153177212670661E-004
11	17.502307845873887	17.494734170385932	4.3272439010034666E-004
12	19.987214495661888	19.980271655554681	3.4736406659938851E-004
13	22.552163853123425	22.545754858935421	2.8418533271324276E-004
14	25.191221182738683	25.185269812625918	2.3624778130418178E-004
15	27.899271383840894	27.893716650288930	1.9909959208402092E-004
16	30.671860106080675	30.666652450161063	1.6978611344735332E-004
17	33.505073450136891	33.500172054188461	1.4628817202040308E-004
18	36.395445208033053	36.390816054283718	1.2719046910608648E-004
19	39.339884187199495	39.335498626950255	1.1147872800975242E-004
20	42.335616460753485	42.331450141061481	9.8411693044942686E-005

21	45.380138898476908	45.376170944258263	8.7438124143291159E-005
22	48.471181351835227	48.467393733766791	7.8141649590576936E-005
23	51.606675567764377	51.603052607539681	7.0203325148098907E-005
24	54.784729398112326	54.781257376729350	6.3375714749727078E-005
25	58.003605222980525	58.000272067343786	5.7464628688598669E-005
26	61.261701761002008	61.258496790773954	5.2316049602376045E-005
27	64.557538627006338	64.554452348323736	4.7806634952946398E-005
28	67.889743137181540	67.886767073197987	4.3836724754421305E-005
29	71.257038967168015	71.254165517805660	4.0325130036325496E-005
30	74.658236348830172	74.655458673900412	3.7205204215939583E-005

Observe que o valor de `st` em cada iteração não depende do seu valor anterior. Esta é a vantagem da aproximação de Stirling: permite calcular o fatorial de números grandes sem recursão ou laços. Por exemplo, num único cálculo, sabemos que $\log(1000000!) \approx 12815518{,}38$. Conforme previsto (pois a aproximação tende assintoticamente ao valor real), o erro relativo diminuiu conforme aumentou n, tornando-se menor que 0,01% já em $n = 20$.

Problema 2 – Série de Taylor para o cosseno

Através de um laço (`do while`), podemos calcular sucessivos termos da série e o resultado cumulativo, até que o incremento se torne menor que a precisão desejada. Na i-ésima iteração, calculamos o termo de ordem $2i$ na série, armazenado na variável `inc`. A variável auxiliar `fac` é o fatorial do denominador. Em cada iteração, adiciona-se ao resultado `res` o incremento calculado na iteração anterior. O laço é interrompido se o número de iterações supera `imax` ou o último termo calculado é menor que `eps`. Todo o processo é repetido para cada real `x` dado numa linha da entrada.

```
        program taylor
        implicit none

        ! Calcular o cosseno de x real pela série de Taylor
        ! cos x = 1 - x^2/2! + x^4/4! - x^6/6! + ...

        integer i,imax
        real*8 x,inc,res,fac,eps

        parameter (imax = 50)    ! limite de iterações
        parameter (eps = 1.d-6) ! precisão satisfatória

        do
            read(*,*,end=20) x

            i = 1
            fac = 1.d0
            res = 0.d0
            inc = 1.d0
            do while (abs(inc) >= eps .and. i <= imax)
                res = res + inc
                fac = fac*dfloat(2*i)*dfloat(2*i-1)
                inc = (-1)**i * x**(2*i)/(fac)
                i = i+1
            end do
            res = res + inc

            write(*,*) x, res, abs(inc), i-1
        end do

20      end program taylor
```

Usando os números 0,2, 1, 3, 5 e -8 para teste:

```
#          x                      cos(x)                       erro           número de termos
0.20000000000000001     0.98006657777777773     8.8888888888888935E-008         3
 1.0000000000000000     0.54030230379188704     2.7557319223985888E-007         5
 3.0000000000000000    -0.98999249800615452     6.0512008015618833E-008         9
 5.0000000000000000     0.28366218903935225     9.6067045396335028E-008        12
-8.0000000000000000    -0.14550001746791133     3.0109797634333939E-007        16
```

Problema 3 – Valores médios e desvio padrão

Nesse programa, `x` é um array de tamanho `n`, `med` = $\langle x \rangle$, `medquad` = $\langle x^2 \rangle$, `dp1` = $\sqrt{\sum_{i=1}^{n}(x_i - \langle x \rangle)^2/n}$ e `dp2` = $\sqrt{\langle x^2 \rangle - \langle x \rangle^2}$. Como `dp1` e `dp2` são apenas o desvio padrão calculado de maneiras diferentes, seus valores devem ser equivalentes.

A entrada consiste de `n`, seguido dos valores de `x`, um por linha. No primeiro laço, são lidos os valores e calculadas as médias. É necessário um segundo laço para calcular `dp1`, pois este requer o conhecimento de $\langle x \rangle$. `dp2` é calculado em seguida, mas não requer o segundo laço.

```fortran
program media
implicit none

integer i, n
real*8 med, medquad, dp1, dp2
real*8, dimension(:), allocatable :: x

read(*,*) n     ! ler tamanho da lista
allocate(x(n)) ! alocar memória

med = 0.d0
medquad = 0.d0
dp1 = 0.d0
dp2 = 0.d0

do i=1,n
    read(*,*) x(i)
    med = med + x(i)
    medquad = medquad + x(i)**2
end do
med = med/dfloat(n)
medquad = medquad/dfloat(n)

do i=1,n
    dp1 = dp1 + (x(i)-med)**2
end do
dp1 = sqrt(dp1/n)
dp2 = sqrt(medquad - med**2)

write(*,*) n, med, dp1, dp2

end program media
```

Teste com `n` $= 12$, `x` $= (1, 2, \ldots, 12)$:

```
#n          med                         dp1                         dp2
12    6.5000000000000000          3.4520525295346629          3.4520525295346629
```

Note a equivalência de `dp1` e `dp2`.
Indo além das especificações, o programa pode ser usado também com valores x_i negativos.

Problema 4 – Organizar uma lista

Primeiro, lemos o tamanho `n` da lista `nums` de números reais e a quantidade `m` de números que queremos da lista ordenada. Um laço lê as entradas de `nums`, uma por linha. Um outro laço percorre a lista e armazena os valores mínimo `xmin` e máximo `xmax`, assim como o índice `k` do valor mínimo.

No último (e principal) laço, a lista é percorrida `m` vezes. Em cada iteração, o índice do menor valor na lista é armazenado em `k`, e o valor em `xmin`. Escrevemos o valor `xmin` na tela, e então alteramos `nums(k)` para `xmax`, de modo que na próxima iteração obteremos o próximo dos menores números na lista. Por fim, fazemos `xmin = xmax`, o que garante que a busca pelo menor valor será repetida na próxima iteração.

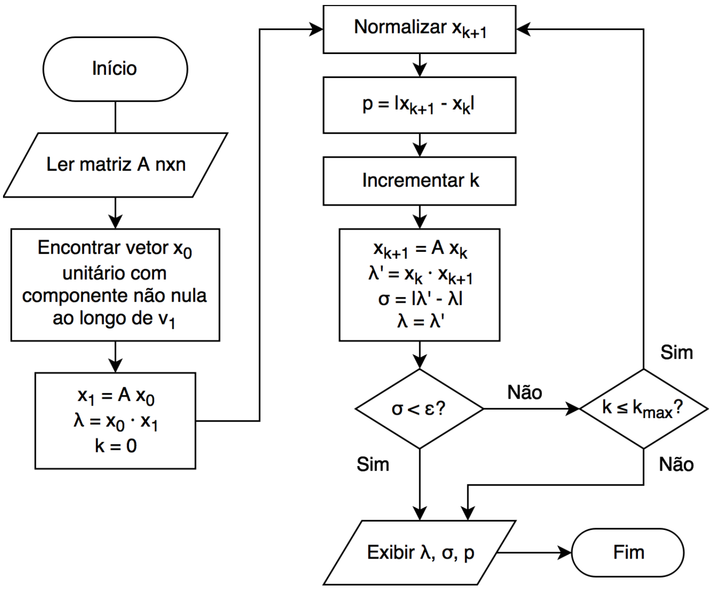
```fortran
program lista
implicit none

real*8, dimension(:), allocatable :: nums
real*8 xmin, xmax
integer n, m, i, j, k

read(*,*) n,m
allocate(nums(n))

do i=1,n
    read(*,*) nums(i)
end do

! Encontrar mínimo e máximo valores da lista
xmin = nums(1)
xmax = nums(1)
do i=1,n
    if (nums(i) <= xmin) then
        xmin = nums(i)
        k = i
    end if
    if (nums(i) >= xmax) then
        xmax = nums(i)
    end if
end do

! Percorrer a lista m vezes. Em cada iteração, fazer k = índice do
! menor valor encontrado. Imprimir tal valor, e alterá-lo para xmax.
do j=1,m
    do i=1,n
        if(nums(i) < xmin) then
            xmin = nums(i)
            k = i
        end if
    end do
    write(*,*) nums(k)
    nums(k) = xmax
    xmin = xmax
end do

end program lista
```

Teste com `n` $= 10$, `m` $= 5$, `nums` $= (0, 122, 89, 56, 23, 45, 26, 75, 59, 12)$:

```
0.0000000000000000
12.000000000000000
```

```
23.000000000000000
26.000000000000000
45.000000000000000
```

Problema 5 – Método da potência para o cálculo do autovalor/autovetor dominante



O diagrama explica o algoritmo implementado abaixo. No código, `y` representa x_{k+1}, `x` representa x_k, `prec_x` representa p e `lambda2` representa λ'. Note que σ é a precisão do autovalor, enquanto p é a precisão do autovetor (norma da diferença entre os vetores das duas últimas iterações normalizados). Em caso de autovalor negativo, fazemos $p = \|x_{k+1} + x_k\|$ em vez de $\|x_{k+1} - x_k\|$. A razão é que x_k e x_{k+1}, normalizados, ficam então em pontos praticamente antipodais da n-esfera unitária.

A função `prod` dá o produto interno usual de vetores. Isso possibilita o cálculo da norma induzida $\|x\| = \sqrt{x \cdot x}$ e do autovalor aproximado $\lambda = x_k \cdot x_{k+1}$. O palpite usado para x_0 foi um vetor com todas as componentes iguais; isso basta para as matrizes cujos autovalores foram pedidos. No caso geral, seria melhor usar componentes aleatórias, para garantir que x_0 tem componente não nula ao longo do autovetor dominante.

```fortran
program autovalor
implicit none

real*8, dimension(:,:), allocatable :: A
real*8, dimension(:), allocatable :: x,y ! xk, xk+1
real*8 sigma, eps, lambda, lambda2, prec_x, prod
integer i,j,k,n,kmax

parameter (eps = 1.d-8) ! tolerância do erro do autovalor
parameter (kmax = 60)   ! número máximo de iterações

read(*,*) n
allocate(A(n,n))
allocate(x(n))
allocate(y(n))

! Ler matriz
do i=1,n
    read(*,*) ( A(i,j), j=1,n )
```

```
end do

! Chutar vetor x0 = (1,1,...,1)/sqrt(n)
do i=1,n
    x(i) = 1/sqrt(float(n))
end do

! x1 = A x0
call multmv(y,A,x,n)

lambda = prod(x,y,n)
k = 0

!!!!!! LOOP !!!!!!
sigma = 10 ! qualquer coisa maior que eps para começar
do while(k <= kmax .and. sigma >= eps)
    ! Norma induzida pelo produto interno
    y = y/sqrt(prod(y,y,n))
    prec_x = sqrt(prod(x-y,x-y,n))

    k = k+1
    x = y

    call multmv(y,A,x,n)
    lambda2 = prod(x,y,n)
    sigma = abs(lambda2 - lambda)
    lambda = lambda2
end do

write(*,*) lambda, sigma, prec_x

end program autovalor

! Produto interno
function prod(x,y,n)
    real*8 prod
    real*8 x(n), y(n)
    integer i
    prod=0
    do i=1,n
        prod = prod + x(i)*y(i)
    end do
    return
end function

! Multiplica matriz A por vetor x e põe o resultado em y
subroutine multmv(y,A,x,n)
    real*8 A(n,n), y(n), x(n), soma
    do i=1,n
        soma = 0
        do j=1,n
            soma = soma + A(i,j)*x(j)
        end do
        y(i) = soma
    end do
    return
end subroutine
```

As entradas (matrizes hermitianas, e portanto com todos os autovalores reais)

```
3

2  8 10
8  4  5
10 5  7
```

```
4

10  -2   3   2
-2  10  -3   4
 3  -3   6   3
 2   4   3   6
```

```
5

-10   2  -3  -2  -1
  2 -10   3  -4  -2
 -3   3  -6  -3  -3
 -2  -4  -3  -6  -4
 -1  -2  -3  -4  -13
```

produziram, respectivamente (da esquerda para a direita), as saídas

```
#    autovalor                erro do autovalor        erro do autovetor
 19.884236025986148       4.2546304257484735E-009   1.8299148007582348E-005
 14.199731035074299       9.9279855447775844E-009   1.9493055558913181E-005
-17.764516401340160       9.3337675366456097E-009   1.7110762819114821E-005
```

A precisão dos autovetores foi sempre menor que a dos autovalores. Os autovalores calculados pelo método iterativo concordam com os valores calculados (com precisão arbitrária) pelas bibliotecas Python SymPy e mpmath dentro de 10^{-7}, com o oitavo dígito após o separador decimal divergindo. O erro dos autovalores foi, portanto, ligeiramente subestimado pelo algoritmo.

Apêndice: tratamento de dados

Nos exercícios em que se pediu escrever os resultados num arquivo, usei redirecionamentos no *shell* do GNU/Linux. Por exemplo, para enviar a saída do programa `stirling` para o arquivo `stirling.dat`, basta fazer

```
$ ./stirling > stirling.dat
```

O comando `cat` lê arquivos e os escreve na saída padrão. Digamos que há uma matriz armazenada no arquivo `autovalor1.in`:

```
$ cat autovalor1.in
3

2  8 10
8  4  5
10 5  7
```

O `cat` pode ser usado para direcionar arquivos para a entrada de outros programas, através de um *pipe*. Por exemplo, para enviar a matriz armazenada para o programa `autovalor`:

```
$ cat autovalor1.in | ./autovalor
  19.884236025986148        4.2546304257484735E-009   1.8299148007582348E-005
```

Esses dois processos podem ser combinados:

```
$ cat autovalor1.in | ./autovalor > autovalor.dat
```

Isto justifica a falta do processamento de arquivos (parte das tarefas do projeto) nos códigos-fonte dos programas. Os requisitos foram cumpridos, mas de outra maneira.