some

Numerical Methods

with

Julia

programming language

CAMBarbosa, DSc

09/2019
http://www.camb.eng.br/nmj

Contents

Introduction

This tutorial presents the implementation (made by the author) in Julia[1] programming language) of some numerical methods applied in the course 'Computational Numerical Calculus' at UFMT[2]. These numerical methods are part of the syllabus for the course. In some cases, this programs version may differ from those presented in the course, as identified bugs are fixed and other enhancements are incorporated, aiming to improve use by other readers. However, it is important to emphasize that this programs version has the academic purpose.

Description

In order to facilitate understanding and application, for each programs the full source code are presented. With this approach, some small blocks of code (e.g. functions) may be present in more than one program. It is adopted the presentation sequence starting with simpler programs, having as reference the sequence of topics, adopted by the syllabus of the course where these programs are applied.

At the beginning of each program is presented: (**i**) name considered for the 'file.jl'; (**ii**) name of the numerical method, (**iii**) input parameters for the method call function; (**iv**) value returned as processing status and a brief message corresponding to each situation; (**v**) values returned or presented as a result of processing; (**vi**) reference (including sheet number) considered for the development of the main algorithm of the method.

The reference considered (in each method) describes the theoretical foundations and, in several cases, a basic pseudo code algorithm of the studied method. In the reference cases where there is a pseudo code, as well as in cases where there is no pseudo code, as well as the complementary functions required for the functional program presented, such algorithms and implementation in Julia[1] are developed by the author. The author notes that no search is done, seeking algorithms or programs for the topics studied. The approach followed by the author is that maintains the bibliographic reference adopted for the course and develops the algorithms and programs, appropriate to the theoretical aspects studied. In this sense, the author does not intend to make any kind of comparison between the algorithms and programs presented (in this tutorial) with existing ones, whether in mathematical libraries, numerical software or other sources.

Some issues should normally be noted in numerical processing, for example: (**i**) when a particular problem can be solved by more than one method, it may occur that the solution can be found with a certain method but not with

[1] Julia, release v.1.2.0;

[2] Universidade Federal de Mato Grosso (Federal University of Mato Grosso) / Campus Universitário do Araguaia (Araguaia University Campus) / Bacharelado em Engenharia Civil (Bachelor in Civil Engineering) / 2019-1;

another one; (**ii**) due to computer number representation, rounding, etc., there may be slight numerical variations in the solutions found by the methods used; (**iii**). each numerical method has its specificity, has processing limitations and, in certain cases, may be sensitive to the input parameters (i.e. when such parameters are not consistent with the method's operating conditions, the solution for the problem studied may not be found). Thus, for a better understanding of the operating conditions of each method, the author suggests that a reading be made in the indicated reference.

Comments in the code are usually in portuguese. Some programs call functions referring to other methods, in the 'usage example' presented at the end of each program, there is a sequence of operations to be performed for the application of a particular method. Also, in the 'usage example' `pth` is adopted as the path to the directory where the programs are located, which can be set with the command (e.g. linux):

```
julia> pth = "/home/user/mypath"
```

Licence

Considerations

The author thanks the students of the course[2] for their observations on the use of these programs. With these contributions and other issues noted by the author, several improvements are incorporated into the algorithms presented in this tutorial.

The author makes this tutorial available considering that the presented algorithms, as well as their implementation in Julia[1], may be useful for those interested in the study and development of computational numerical methods.

Roots second degree polynomial

```
# root2degree.jl / CAMBarbosa
# description:
# » computing roots of a 2 degree polynomial
# » input: coefficientes 'a', 'b', 'c'
# » output: status successful
#           roots 'x1','x2'
# reference:
# » public domain

using Printf # incluindo biblioteca

function Root2Degree(a,b,c)
# computing roots of a 2 degree polynomial
# x1,x2 = (-b +- (b^2 - 4ac)) / (2a)
  if a == 0 # nao é equação 2 grau; calcula raiz para equação da reta y = bx + c
    if b == 0 # nao é equação 1 grau; é apenas um ponto
      @printf("a = b = 0, point on the axis, y = %.3f\n",c)
    else
      @printf("a = 0, just a root, x = %.3f\n",-c / b)
    end
  else
    dt = b^2 - 4 * a * c # calcula 'dt'
    # avalia os casos|situacoes possiveis para 'dt'
    if dt == 0 # 'dt' igual a zero; raizes iguais
      @printf("equal roots, x1 = x2 = %.3f\n",-b / (2 * a))
    elseif 0 < dt # raizes diferentes
          ax = dt^0.5;  bx = 2 * a
```

```
          @printf("different roots, x1 = %7.3f\n",(-b + ax) / bx)
          @printf("                 x2 = %7.3f\n",(-b - ax) / bx)
        else # dt < 0, raizes conjugado complexo
          ax = 2 * a;  bx = -b / ax;  cx = abs(dt)^0.5 / ax
          @printf("complex roots, x1 = %7.3f%+.3fim\n",bx, cx)
          @printf("              x2 = %7.3f%+.3fim\n",bx,-cx)
        end
  end
end
```

usage example:

```
julia> include("$pth/root2degree.jl")
julia> Root2Degree(1,2,3)
       complex roots, x1 =  -1.000+1.414im
                      x2 =  -1.000-1.414im
```

Decimal machine numbers system

```
# nsystem.jl / CAMBarbosa
# description:
# » number representation in the system F(b,t,m,M)
# » input: amount of decimal digits of precision
#          minimum exponent, maximum exponent, number to be represented
# » output: status successful or error found
#           normalized and rounded number on cientific notation
# reference:
# » Franco,N.B.; 2006; Cálculo Numérico; Pearson Prentice Hall; s42

using Printf # incluindo biblioteca

function Normalize(td, # qtd digitos a direita do ponto
                   nm) # numero para ser normalizado e arredondado
  ep = 0 # define expoente da potencia de 10
  sg = nm < 0 ? -1 : 1 # obtem o sinal do numero
  nm = abs(nm) # obtem o modulo do numero
  if nm != 0 # verifica consistencia do numero fornecido
    if nm < 0.1 # desloca ponto para a direita
      while nm < 0.1 # normaliza o numero
        nm *= 10;   ep -= 1;
      end
    elseif 1 <= nm # desloca ponto para a esquerda
          while 1 <= nm # normaliza o numero
            nm /= 10;   ep += 1;
          end
        end
    nm = convert(Int,trunc(nm * 10.0^td + 0.5)) / 10.0^td  # desloca ponto para a direita td digitos
  end
  return sg*nm,ep # retorna numero e o expoente da potencia de 10
end

function System(bt, # beta
                td, # qtd digitos a direita do ponto
                en, # expoente mínimo
                ex, # expoente maximo
                nm) # numero para ser representado no sistema proposto
  # calcula a representacao do numero no sistema proposto F(beta,qtd_digitos,exp_minimo,exp_maximo)
  # retorna: rs,nm,ep
  #          rs == true ou false, numero 'nm' pode (ou nao) ser representado no sistema proposto
  #          quando rs == true, 'nm' = numero normalizado e arredondado conforme qtd de digitos do sistema
  #                             'ep' = expoente da potencia de 10 para ser multiplicada pelo numero
  rs = false;   nm,ep = Normalize(td,nm) # obtem o numero normalizado e o exponte da potencia de 10
  if -en <= ep <= ex # verifica se o expoente encontrado está dentro do intervalo permitido pelo sistema
```

```
    # calcula os valores significativos minimos e maximo que o sistema pode representar
    sn = 10^(-1) * (1 - 0.5 * 10.0^(-td)) # valor minimo
    sx = 1 - 0.5 * 10.0^(-td)             # valor maximo
    if sn < abs(nm) < sx  rs = true  end # verifica numero encontrado está dentro do intervalo permitido
  end
  return rs,nm,ep
end

function NSystem(td, # qtd digitos a direita do ponto
                 en, # expoente mínimo
                 ex, # expoente maximo
                 nm) # numero para ser representado no sistema proposto
  # verifica se o numero pode ser representado no sistema F(10,td,en,ex)
  rs,nm,ep = System(10,td,en,ex,nm)
  if rs == false # numero não pode ser representado no sistema proposto
    @printf("number can not be represented in the system!\n")
  else # numero pode ser representado no sistema proposto
    nm = string(nm,ep < 0 ? "e-" : "e+",abs(ep))
    @printf("number represented in the system: %s\n",nm)
  end
end
```

usage example:

```
julia> include("$pth/nsystem.jl")
julia> NSystem(3,5,5,123456.7)
       number cannot be represented in the system!
julia> NSystem(3,5,5,1234.56)
       number represented in the system: 0.123e+4
```

Bisection (zero of function)

```
# bisection.jl / CAMBarbosa
# description:
# » bisection method, root of one variable non linear equation
# » input: interval [a,b] where the function is continuous
# » output: status successful or error found
#           iteration number, root obtained, estimated error, convergence order
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s48
# » Franco,N.B.; 2006; Cálculo Numérico; Pearson Prentice Hall; s73

using Printf

er = zeros(Float32,1,4)
function Convergence(k,ek)
# calcula|retorna a ordem de convergencia do método
  if k < 4  er[k] = ek
  else      ex = er;  er[1:3] = vcat(ex[2:3],ek) end
  if 2 < k  er[4] = log(er[3]/er[2]) / log(er[2]/er[1])  end
  return er[4]
end

function Bisection(a,b,fct)
# encontra zeros de funções não linear pelo método
  k,x,qt,em,ex = 0,0,1000,1.0e-6,abs(b - a) # erro maximo permitido
  ms = [ "Bisection, successful calculation"
         "Bisection, not converge" ]
  @printf("   k   x         ex           p\n")
  while ex > em && k < qt # enquanto erro permitido é maior que erro maximo
    x = (a + b) / 2.0 # determina ponto medio entre 'a' e 'b'
    if fct(a) * fct(x) < 0  b = x # ponto está a esquerda de 'x'
    else                    a = x  end # ponto está sobre ou a direita de 'x'
```

```
    ex = abs(b - a);  k += 1 # calcula delta e incrementa contador de iteraçoes
    @printf("%4d  %+8.6f  %11.9f  %.6f\n",k,x,ex,Convergence(k,ex))
  end
  if k < qt # calculo bem sucedido
    @printf("  iteration number: %d\n",k)
    @printf("  root obtained:    %+.6f\n",x)
    @printf("  estimated error:   %.9f\n",ex)
  end
  return ms[k < qt ? 1 : 2]
end
```

usage example:

```
julia> include("$pth/bisection.jl")
julia> fct(x) = x^3 + 4*x^2 - 10;
julia> Bisection(1,2,fct)
   k   x         ex           p
   1  +1.500000  0.500000000  0.000000
   2  +1.250000  0.250000000  0.000000
   3  +1.375000  0.125000000  1.000000
      ...
  18  +1.365231  0.000003815  1.000000
  19  +1.365229  0.000001907  1.000000
  20  +1.365230  0.000000954  1.000000
  iteration number: 20
  root obtained:    +1.365230
  estimated error:   0.000000954
"Bisection, successful calculation"
```

Newton (zero of function)

```
# newton.jl / CAMBarbosa
# description:
# » newton method, root of one variable non linear equation
# » input: initial approximation 'a', the function and its derivative
# » output: status successful or error found
#           iteration number, root obtained, estimated error, convergence order
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s67
# » Franco,N.B.; 2006; Cálculo Numérico; Pearson Prentice Hall; s73

using Printf

er = zeros(Float32,1,4)
function Convergence(k,ek)
# calcula|retorna a ordem de convergencia do método
  if k < 4  er[k] = ek
  else      ex = er;  er[1:3] = vcat(ex[2:3],ek) end
  if 2 < k  er[4] = log(er[3]/er[2]) / log(er[2]/er[1])  end
  return er[4]
end

function Newton(a,fct,drv)
  k,qt,em = 0,1000,1e-6;
  x0::Float32 = a;   x::Float32 = ex::Float32 = 1
  ms = [ "Newton, successful calculation"
         "Newton, not converge" ]
  @printf("   k   x         ex           p\n")
  while em < ex && k < qt
    x = x0 - fct(x0) / drv(x0) # calcula novo ponto
    ex = abs(x - x0) # calcula erro
    x0 = x;   k += 1 # atualiza valores
    @printf("%4d  %+8.6f  %11.9f  %.6f\n",k,x,ex,Convergence(k,ex))
  end
```

```
  if k < qt # calculo bem sucedido
    @printf("  iteration number: %d\n",k)
    @printf("  root obtained:    %+.6f\n",x)
    @printf("  estimated error:   %.9f\n",ex)
  end
  return ms[k < qt ? 1 : 2]
end
```

usage example:

```
julia> include("$pth/newton.jl")
julia> fct(x) = 4 * cos(x) - exp(x);
julia> drv(x) = -4 * sin(x) - exp(x);
julia> Newton(1.0,fct,drv)
   k   x         ex           p
   1  +0.908439  0.091561079  0.000000
   2  +0.904794  0.003644824  0.000000
   3  +0.904788  0.000005841  1.996501
   4  +0.904788  0.000000060  0.712381
  iteration number: 4
  root obtained:    +0.904788
  estimated error:   0.000000060
"Newton, successful calculation"
```

Secant (zero of function)

```
# secant.jl / CAMBarbosa
# description:
# » secant method, root of one variable non linear equation
# » input: interval [a,b] where the function is continuous
# » output: status successful or error found
#           iteration number, root obtained, estimated error, convergence order
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s71
# » Franco,N.B.; 2006; Cálculo Numérico; Pearson Prentice Hall; s73

using Printf

er = zeros(Float32,1,4)
function Convergence(k,ek)
# calcula|retorna a ordem de convergencia do método
  if k < 4  er[k] = ek
  else      ex = er;  er[1:3] = vcat(ex[2:3],ek) end
  if 2 < k
    er[4] = log(er[3]/er[2]) / log(er[2]/er[1])
  end
  return er[4]
end

function Secant(a,b,fct)
  k,x0,x1,qt,em = 0,a,b,1000,1e-6
  x  = ex = 1;   q0,q1 = fct(x0),fct(x1)
  ms = [ "Secant, successful calculation"
         "Secant, not converge" ]
  @printf "   k   x         ex           p\n"
  while em < ex && k < qt
    x = x1 - q1 * (x1 - x0) / (q1 - q0) # calcula novo ponto
    ex = abs(x - x1) # calcula erro
    x0 = x1;   q0 = q1;   x1 = x;   q1 = fct(x);   k += 1 # atualiza valores
    @printf "%4d  %+8.6f  %11.9f  %.6f\n" k x ex Convergence(k,ex)
  end
  if k < qt # calculo bem sucedido
    @printf "  iteration number: %d\n" k
```

```
    @printf "  root obtained:    %+.6f\n" x
    @printf "  estimated error:   %.9f\n" ex
  end
  return ms[k < qt ? 1 : 2]
end
```

usage example:

```
julia> include("$pth/secant.jl")
julia> fct(x) = x^3 + 4*x^2 - 10;
julia> Secant(1,2,fct)
   k   x         ex          p
   1  +1.263158  0.736842105  0.000000
   2  +1.338828  0.075669944  0.000000
   3  +1.366616  0.027788556  0.440141
   4  +1.365212  0.001404492  2.979714
   5  +1.365230  0.000018098  1.457849
   6  +1.365230  0.000000012  1.676098
  iteration number: 6
  root obtained:    +1.365230
  estimated error:   0.000000012
"Secant, successful calculation"
```

LU factorization (lower upper)

```
# lufactoriztion.jl / CAMBarbosa
# description:
# » computing lu factorization method
# » input: coefficient matrix 'A'
# » output: status successful or error found
#           lower upper triangular matrix
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s400

function LUFactorization(A)
# cria matrizes L (trinagular inferior) e U (triangular superior)
# A[i,j],   1<=i<=n  1<=j<=n
  L,U,rs,er,n = 0,0,0,1.0e-6,size(A)[1]
  ms = [ "LUFactorization, successful calculation"          # rs = 0
         "LUFactorization, matrix inconsistent dimensions"  # rs = 1
         "LUFactorization, division by zero|small number" ] # rs = 2
  if size(A)[2] != n # verifica consistencia 'n,n' da matriz
    rs = 1 # inconsistencia na dimensao da matriz
  else
    L = zeros(Float32,n,n);  U = zeros(Float32,n,n)
    A = [convert(Float32,A[i,j]) for i = 1:n, j = 1:n]
    L[1,1] = 1;  U[1,1] = A[1,1] # inicializa primeiro elementos de 'L U'
    if abs(L[1,1] * U[1,1]) < er
      rs = 2 # diagonal com elemento zero
    else
      for j = 2:n # percorre linhas de 'L' e colunas de 'U'
        U[1,j] = A[1,j] / L[1,1] # primeira linha de 'U'
        L[j,1] = A[j,1] / U[1,1] # primeira coluna de 'L'
      end
      for i = 2:n-1 # percorre linhas de 'U' e colunas de 'L'
        L[i,i] = 1;  U[i,i] = A[i,i] # inicializa elemento da diagonal principal de 'L U'
        for k = 1:i-1 # soma produto das colunas de L[i,1..i-1] pelas linhas de U[1..i-1,i]
          U[i,i] -= L[i,k] * U[k,i]
        end
        for j = i+1:n # percorre linhas de 'L' e colunas de 'U'
          L[j,i] = A[j,i] # inicializa elemento de 'L'
          U[i,j] = A[i,j] # inicializa elemento de 'U'
          for k = 1:i-1 # define elemento de 'U'
            L[j,i] -= L[j,k] * U[k,i]
```

```
            U[i,j] -= L[i,k] * U[k,j]
          end
          L[j,i] /= U[i,i] # elemento da diagonal de 'U' não é zero
          U[i,j] /= L[i,i] # elemento da diagonal de 'L' não é zero
        end
      end
      L[n,n] = 1;  U[n,n] = A[n,n] # inicializa elemento da diagonal principal de 'L U'
      for k = 1:n-1 # soma produto das colunas de L[n,1..n-1] pelas linhas de U[1..n-1,n]
        U[n,n] -= L[n,k] * U[k,n]
      end
    end
  end
  return rs,L,U,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/lufactorization.jl")
julia> A = [1 1 0 3; 2 1 -1 1; 3 -1 -1 2; -1 2 3 -1];
julia> LUFactorization(A)
       (0, Float32[1.0 0.0 0.0 0.0; 2.0  1.0  0.0  0.0; 3.0 4.0 1.0  0.0; 1.0 -3.0 0.0   1.0],
           Float32[1.0 1.0 0.0 3.0; 0.0 -1.0 -1.0 -5.0; 0.0 0.0 3.0 13.0; 0.0  0.0 0.0 -13.0],
        "LUFactorization, successful calculation")
       det(A) = det(L) * det(U) = 1.0 * (1.0 * (-1.0) * 3.0 * (-13.0)) = 39.0
```

LU solve (system Ax = b)

```
# lusolve.jl / CAMBarbosa
# description:
# » computing linear systems of equations, lu factorization method
# » input: lower upper triangular matrix and result vector 'b'
# » output: status successful or error found
#           'x' solution vector
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s400

function LUSolve(b,L,U)
# resolve sistema de equações lineares usando 'L U' » Ax = LUx = b
  rs,n = 0,size(L)[1]
  x,y = Matrix{Float32}(undef,1,n),Matrix{Float32}(undef,1,n) # define vetor para resultados
  ms = [ "LUSolve, successful calculation"           # rs = 0
         "LUSolve, matrix inconsistent dimensions" ] # rs = 1
  if size(U)[1] != n || size(U)[2] != n ||
     size(L)[2] != n || size(b)[ndims(b) == 1 ? 1 : 2] != n # verifica consistencia matrizes
    rs = 1 # inconsistencia na dimensão da matriz
  else
    b = transpose([convert(Float32,b[i]) for i = 1:n]) # convert vetor 'b' to real
    # processa Ly = b com 'forward substitution'
    y[1] = b[1] / L[1,1] # inicializa primeiro elemento de 'y'
    for i = 2:n # percorre as linhas de 'L'
      y[i] = b[i] # inicializa elemento do vetor 'y'
      for j = 1:i-1 # percorre as colunas de 'L'
        y[i] -= L[i,j] * y[j]
      end
      y[i] /= L[i,i]
    end
    # processa Ux = Y com 'backward substitution'
    x[n] = y[n] / U[n,n] # inicializa ultimo elemento de 'x'
    for i = n-1:-1:1 # percorre as linhas de 'U'
      x[i] = y[i] # inicializa elemento do vetor 'x'
      for j = i+1:n # percorre as colunas de 'U'
        x[i] -= U[i,j] * x[j]
      end
```

```
      x[i] /= U[i,i]
    end
  end
  return rs,x,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/lufactorization.jl")
julia> include("$pth/lusolve.jl")
julia> b = [9 9 12];  A = [2 -1 3; 4 2 1; -6 -1 2];
julia> lufactorization(A)
       (0, float32[1.0 0.0 0.0; 2.0 1.0 0.0; -3.0 -1.0 1.0],
           float32[2.0 -1.0 3.0; 0.0 4.0 -5.0; 0.0 0.0 6.0], "lufactorization, successful calculation")
julia> L = [1.0 0.0 0.0; 2.0 1.0 0.0; -3.0 -1.0 1.0];  U = [2.0 -1.0 3.0; 0.0 4.0 -5.0; 0.0 0.0 6.0];
julia> lusolve(b,L,U)
       (0, float32[-1.0 4.0 5.0], "lusolve, successful calculation")
```

Gaussian elimination (system Ax = b)

```
# gelimination.jl / CAMBarbosa
# description:
# » computing linear systems of equations, gaussian elimination with backward substitution method
# » input: coefficient matrix 'A' and result vector 'b'
# » output: status successful or error found
#           matrix extended initial and final
#           'x' solution vector
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s358

using Printf

function Show(n,m,A,tx)
# apresenta matriz A
  println("matrix: $tx")
  for i = 1:n
    for j = 1:m
      @printf("  %10.6f",A[i,j])
    end
    @printf("\n")
  end
end

function Multiplicity(n,A)
# verifica se uma linha|coluna é multipla de outra
  rs,rd = false,1.0e+6 # arredondamento
  for i = 1:n-1 # multiplicidade entre linhas
    for k = i+1:n # percorre as linhas abaixo da linha corrente
      j::Int8,mt::Int8 = 2,0 # flag para nao multiplicidade em linha|coluna
      fl::Float32 = trunc(Int64,A[k,1]*rd) / trunc(Int64,A[i,1]*rd) # fator elementos primeira linha
      fc::Float32 = trunc(Int64,A[1,k]*rd) / trunc(Int64,A[1,i]*rd) # fator elementos primeira coluna
      while j < n+1 # interrompe quando faor linha&coluna sao diferentes
        mt |= (fl != trunc(Int64,A[k,j]*rd) / trunc(Int64,A[i,j]*rd))      # diferente fator linha
        mt |= (fc != trunc(Int64,A[j,k]*rd) / trunc(Int64,A[j,i]*rd)) << 1 # diferente fator coluna
        if mt == 0b11 # encontro linha|coluna com fator diferente
          break;
        end
        j += 1;
      end
      if j == n+1  rs = true;  @goto(RTN)  end # linha|coluna 'k' é multipla da linha|coluna 'i'
    end
  end
  @label(RTN)
```

```
  return rs
end

function Pivot(j,n,M)
# define o elemento M[j,j] pivot da matriz
  rs,er = 0,1.0e-6
  # inicia pivotamento de linha
  p = j # linha corrente
  for i = j+1:n # procura linha com elemento nao nulo na coluna 'j'
    if abs(M[p,j]) < abs(M[i,j])  p = i  end # obtem a linha com maior valor
  end
  if p != j # troca linha, encontrou linha com elemento com maior valor
    ax = M[p,1:n+1];  M[p,1:n+1] = M[j,1:n+1];  M[j,1:n+1] = ax[1:n+1]; # troca elementos linha 'p|j'
  end
  if abs(M[j,j]) < er # pivotamento de linha nao foi bem sucedido
    # inicia pivotamento de coluna
    p = j # coluna corrente
    for i = j+1:n # procura coluna com elemento nao nulo na linha 'j'
      if abs(M[j,p]) < abs(M[j,i])  p = i  end # obtem a coluna com maior valor
    end
    if p != j # troca coluna, encontrou coluna com elemento com maior valor
      ax = M[1:n,p];  M[1:n,p] = M[1:n,j];  M[1:n,j] = ax[1:n,1]; # troca elementos coluna 'p|j'
    elseif abs(M[j,j]) < er  rs = 4  end # pivotamento de coluna nao foi bem sucedido
  end
  return rs,M
end

function Elimination(j,n,M)
  # zera coeficiente nas posicoes 'i,j'
  rs,er = 0,1.0e-6
  if abs(M[j,j]) < er # verifica ocorrência divisao por zero
    rs = 3 # divisao por zero
  else
    for i = j+1:n
      lf::Float32 = M[i,j] / M[j,j] # calcula fator de linearidade
      for k = j:n+1 # multiplica fator pelos elementos da linha 'i'
        M[i,k] -= lf * M[j,k]
      end
    end
  end
  return rs,M
end

function BEvaluation(n,M,x)
# processa avaliação reverssa do vetor 'x'
  rs,er = 0,1.0e-6
  if abs(M[n,n]) < er # verifica ocorrência de zero no ultimo elemento da diagonal
    rs = 3 # divisao por zero
  else
    x[n] = M[n,n+1] / M[n,n] # inicia solucao com 'backward substitution'
    for i = n-1:-1:1
      x[i] = M[i,n+1]
      for j = i+1:n
        x[i] -= M[i,j] * x[j]
      end
      x[i] /= M[i,i] # Pivot garante que elemento da diagonal nao é zero
      x[i]  = round(x[i],digits=6) # arredonda valor para mesma qtd digitos que 'er'
    end
  end
  return rs,x
end

function GESolve(b,A)
# gaussian elimination with backward substitution
```

```
# b[j], A[i,j],   1<=i<=n  1<=j<=n;  retorna: (rs [x] msg)
  rs,n = 0,size(A)[1]
  ms = [ "GESolve, successful calculation"          # rs = 0
         "GESolve, line|column multiplicity"        # rs = 1
         "GESolve, matrix inconsistent dimensions"  # rs = 2
         "GESolve, division by zero|small number"   # rs = 3
         "GESolve, singular matrix" ]               # rs = 4
  if size(A)[2] != n || size(b)[ndims(b)] != n # verifica consistencia 'n,n' da matriz
    rs = 2 # inconsistencia na dimensao da matriz
  elseif Multiplicity(n,A) == true
       rs = 1 # infinitas solucoes
     else
       x = Matrix{Float32}(undef,1,n)
       A = [convert(Float32,j < n+1 ? A[i,j] : b[i]) for i = 1:n, j = 1:n+1]
       Show(n,n+1,A,"A extended initial")
       for j = 1:n-1 # percorre as diagonas 'j,j:1..n-1'
         rs,A = Pivot(j,n,A)
         if rs == 0  rs,A = Elimination(j,n,A)  end
         if rs != 0  break  end
       end
       rs,x = BEvaluation(n,A,x)
     end
  Show(n,n+1,A,"A extended final") # apresenta matriz resultante
  return (0 < rs < 3) ? (rs,ms[rs+1]) : (rs,transpose(x),ms[rs+1])
end
```

usage example:

```
julia> include("$pth/gelimination.jl")
julia> b = [16 26 -19 -34];  A = [6 -2 2 4; 12 -8 6 10; 3 -13 9 3; -6 4 1 -18];
julia> GESolve(b,A)
       matrix: A extended initial
           6.000000   -2.000000    2.000000    4.000000   16.000000
          12.000000   -8.000000    6.000000   10.000000   26.000000
           3.000000  -13.000000    9.000000    3.000000  -19.000000
          -6.000000    4.000000    1.000000  -18.000000  -34.000000
       matrix: A extended final
          12.000000   -8.000000    6.000000   10.000000   26.000000
           0.000000  -11.000000    7.500000    0.500000  -25.500000
           0.000000    0.000000    4.000000  -13.000000  -21.000000
           0.000000    0.000000    0.000000    0.272727    0.272727
       (0, Float32[3.000001; 0.999997; -2.000004; 0.9999987], "GESolve, successful calculation")
```

Matrix inverse (system Ax = b)

```
# minverse.jl / CAMBarbosa
# description:
# » computing the inverse matrix, gaussian elimination with lower upper elimination method
# » input: coefficient matrix 'A'
# » output: status successful or error found
#           matrix extended initial and final
#           inverse matrix
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s386

using Printf

function Show(n,m,A,tx)
# apresenta matriz A
  println("matrix: $tx")
  for i = 1:n
    for j = 1:m
      @printf("  %10.6f",A[i,j])
    end
```

```
    @printf("\n")
  end
end

function DNormalize(j,m,M)
# divide elementos da linha pelos elementos da diagonal
  lf::Float32 = M[j,j]
  for i = 1:m # coloca 1 na diagonal principal
      M[j,i] /= lf
  end
  return M
end

function Multiplicity(n,A)
# verifica se uma linha|coluna é multipla de outra
  rs,rd = false,1.0e+6 # arredondamento
  for i = 1:n-1 # multiplicidade entre linhas
    for k = i+1:n # percorre as linhas abaixo da linha corrente
      j::Int8,mt::Int8 = 2,0 # flag para não multiplicidade em linha|coluna
      fl::Float32 = trunc(Int64,A[k,1]*rd) / trunc(Int64,A[i,1]*rd) # fator elementos primeira linha
      fc::Float32 = trunc(Int64,A[1,k]*rd) / trunc(Int64,A[1,i]*rd) # fator elementos primeira coluna
      while j < n+1 # interrompe quando faor linha&coluna sao diferentes
        mt |= (fl != trunc(Int64,A[k,j]*rd) / trunc(Int64,A[i,j]*rd))     # diferente fator linha
        mt |= (fc != trunc(Int64,A[j,k]*rd) / trunc(Int64,A[j,i]*rd)) << 1 # diferente fator coluna
        if mt == 0b11  break  end # encontro linha|coluna com fator diferente
        j += 1;
      end
      if j == n+1  rs = true;  @goto(RTN)  end # linha|coluna 'k' é multipla da linha|coluna 'i'
    end
  end
  @label(RTN)
  return rs
end

function Pivot(n,m,M)
# define o elemento M[j,j] pivot da matriz
  rs,er = 0,1.0e-6
  for j = 1:n-1 # pivotamento de linha
    p = j # linha corrente
    for i = j+1:n # procura linha com elemento nao nulo na coluna 'j'
      if abs(M[p,j]) < abs(M[i,j])  p = i  end# obtem a linha com maior valor
    end
    if p != j # troca linha, encontrou linha com elemento com maior valor
      ax = M[p,1:m];  M[p,1:m] = M[j,1:m];  M[j,1:m] = ax[1:m]; # troca elementos linha 'p|j'
    end
  end
  for j = 1:n-1 # pivotamento de coluna
    if abs(M[j,j]) < er # pivotamento de linha nao foi bem sucedido
      # inicia pivotamento de coluna
      p = j # coluna corrente
      for i = j+1:n # procura coluna com elemento nao nulo na linha 'j'
        if abs(M[j,p]) < abs(M[j,i])  p = i  end # obtem a linha com maior valor
      end
      if p != j # troca coluna, encontrou coluna com elemento com maior valor
        ax = M[1:n,p];  M[1:n,p] = M[1:n,j];  M[1:n,j] = ax[1:n,1]; # troca elementos coluna 'p|j'
      elseif abs(M[j,j]) < er  rs = 4;  @goto(RTN)  end # pivotamento de coluna não foi bem sucedido
    end
  end
  @label(RTN)
  return rs,M
end

function LElimination(j,n,m,M)
# lower, zera coeficiente nas posicoes 'i,j'
```

```
  rs,er = 0,1.0e-6
  if abs(M[j,j]) < er # verifica ocorrência divisão por zero
    rs = 3 # divisao por zero
  else
    for i = j+1:n
      lf::Float32 = M[i,j] / M[j,j] # calcula fator de linearidade
      for k = j:m # multiplica fator pelos elementos da linha 'i'
        M[i,k] -= lf * M[j,k]
      end
    end
  end
  return rs,M
end

function UElimination(j,n,m,M)
# upper, zera coeficiente nas posicoes 'i,j'
  rs,er = 0,1.0e-6
  if abs(M[j,j]) < er # verifica ocorrência divisão por zero
    rs = 3 # divisao por zero
  else
    for i = j-1:-1:1
      lf::Float32 = M[i,j] / M[j,j] # calcula fator de linearidade
      for k = j:m # multiplica fator pelos elementos da linha 'i'
        M[i,k] -= lf * M[j,k]
      end
    end
  end
  return rs,M
end

function MInverse(A)
# matrix inverse, backward|forward substitution
# b[j], A[i,j],   1<=i<=n  1<=j<=n;  retorna: rs M msg
  rs,n,m = 0,size(A)[1],2*size(A)[1]
  ms = [ "MInverse, successful calculation"          # rs = 0
         "MInverse, inconsistent dimensions"         # rs = 1
         "MInverse, line|column multiplicity"        # rs = 2
         "MInverse, division by zero|small number"   # rs = 3
         "MInverse, singular matrix" ]               # rs = 4
  M = Matrix{Float32}(undef,n,m)
  if size(A)[2] != n || n < 2 # verifica consistencia 'n,n' da matriz
    rs = 1 # inconsistencia na dimensão da matriz
  elseif Multiplicity(n,A) == true  rs = 2 # infinitas solucoes
      else
        M[1:n,1:n] = [convert(Float32,A[i,j]) for i = 1:n, j = 1:n] # converte para real
        M[1:n,n+1:m] = [i != j ? 0.0 : 1.0 for i = 1:n, j = 1:n] # acrescenta matriz identidade a direita
        Show(n,m,M,"M initial")
        rs,M = Pivot(n,m,M)
        if rs == 0
          for j = 1:n-1 # percorre as diagonais 'j,j:1..n-1'
            rs,M = LElimination(j,n,m,M)
            if rs != 0  @goto(RTN)  end
          end
          for j = n:-1:2 # percorre as diagonais 'j,j:n..2'
            rs,M = UElimination(j,n,m,M)
            if rs != 0  @goto(RTN)  end
            M = DNormalize(j,m,M) # normaliza linha corrente
          end
          M = DNormalize(1,m,M) # normaliza primeira linha
          Show(n,m,M,"M final") # apresenta matriz resultante
        end
      end
  @label(RTN)
  return (0 < rs < 3) ? (rs,ms[rs+1]) : (rs,M[1:n,n+1:m],ms[rs+1])
```

```
148 end
```

usage example:

```
julia> include("$pth/minverse.jl")
julia> b = [9 9 12]';  A = [2 -1 3; 4 2 1; -6 -1 2];
julia> MInverse(A)
       matrix: M initial
           2.000000   -1.000000    3.000000    1.000000    0.000000    0.000000
           4.000000    2.000000    1.000000    0.000000    1.000000    0.000000
          -6.000000   -1.000000    2.000000    0.000000    0.000000    1.000000
       matrix: M final
           1.000000   -0.000000   -0.000000    0.104167   -0.020833   -0.145833
           0.000000    1.000000    0.000000   -0.291667    0.458333    0.208333
           0.000000    0.000000    1.000000    0.166667    0.166667    0.166667
       (0, Float32[ 0.104166664 -0.020833328 -0.14583333;
                   -0.2916667    0.4583333    0.20833331;
                    0.16666666   0.16666667   0.16666667], "MInverse, successful calculation")
julia> A_1 = [ 0.104166664 -0.020833328 -0.14583333;
              -0.2916667    0.4583333    0.20833331;
               0.16666666   0.16666667   0.16666667];
julia> x = A_1 * b  # solution system Ax=b
         -0.9999999360000003
          3.99999912
          5.000000009999999
```

Jacobi iterative (system Ax = b)

```
  1 # jiterative.jl / CAMBarbosa
  2 # description:
  3 # » computing linear systems of equations, jacobi iterative method
  4 # » input: coefficient matrix 'A', result vector 'b' and optional guess to 'x' vector
  5 # » output: status successful or error found
  6 #           iteration number, 'x' iteration vector, iteration error
  7 #           iteration number, 'x' solution vector
  8 # reference:
  9 # » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s450
 10
 11 using Printf
 12
 13 function Show(k,n,nx,x)
 14   @printf("x(%03d): [",k)
 15   for i = 1:n
 16     @printf("%11.6f",x[i])
 17   end
 18   @printf(" ]   er: %.6f\n",nx)
 19 end
 20
 21 function JISolve(b,A,c = 0)
 22 # iterative jacobi for solve linear systema Ax=bn
 23 # b[j], A[i,j],   1<=i<=n  1<=j<=n;  opcional fornecer vetor z[j] para estimativa inicial
 24 # retorna, 0,x: calculo bem sucedido, vetor com o resultado de 'Ax = b'
 25 #          1|2|3: erro
 26   k,qt,em,nx,rs,n = 0,1000,1.0e-6,1.0,0,size(A)[1]
 27   x = Matrix{Float32}(undef,1,n) # vetor com resultados
 28   ms = ["jISolve, successful"              # rs = 0
 29         "jISolve, error dimensions matrix"  # rs = 1
 30         "jISolve, error singular matrix"    # rs = 2
 31         "jISolve, non convergence" ]        # rs = 3
 32   if ndims(c) == 0   c = [1.0 for i = 1:n]  end # quando vetor 'c' não é fornecido
 33   if size(A)[2] != n || size(b)[ndims(b)] != n || size(c)[ndims(c)] != n # verifica consistencia matriz
 34     rs = 1 # inconsistencia na dimensao da matriz
 35   else
 36     y = [convert(Float32,b[i]) for i = 1:n] # vetor com resultados
 37     z = [convert(Float32,c[i]) for i = 1:n] # vetor com estimativa iniciais para 'x'
```

```
    M = [convert(Float32,A[i,j]) for i = 1:n, j = 1:n] # matriz com os coeficientes
    for i = 1:n
      if abs(A[i,i]) < em  rs = 2;  @goto(RTN)  end # encontro zero na diagonal
    end
    while em < nx && k < qt # erro maximo menor que norma do vetor 'x' e não atingiu qtd iterações
      for i = 1:n # percorre as linhas da matriz
        x[i] = 0.0
        for j = 1:n # percorre as colunas da linha 'i'
          if j != i # não é elemento da diagonal principal
            x[i] += M[i,j] * z[j] # subtrai produto dos outros termos da linha 'i'
          end
        end
        x[i] = (y[i] - x[i]) / M[i,i]
      end
      k,nx = k+1,0.0
      for i = 1:n
        if nx < abs(x[i] - z[i]) # norma, maior diferença entre iteração corrente e anterior
          nx = abs(x[i] - z[i])
        end
        z[i] = x[i] # salva valor da iteração corrente para comparação na iteração seguinte
      end
      Show(k,n,nx,x) # apresenta o vetor da iteração 'k'
    end
  end
  if k == qt  rs = 3  end
  @label(RTN)
  return rs,k,x,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/jiterative.jl")
julia> b = [0 2 1];  A = [5 2 1; 2 4 1; 2 2 4];
julia> JISolve(b,A)
       x(001): [  -0.600000  -0.250000  -0.750000 ]   er: 1.750000
       x(002): [   0.250000   0.987500   0.675000 ]   er: 1.425000
       x(003): [  -0.530000   0.206250  -0.368750 ]   er: 1.043750
       ...
       x(049): [  -0.259260   0.611111   0.074073 ]   er: 0.000002
       x(050): [  -0.259259   0.611112   0.074075 ]   er: 0.000001
       x(051): [  -0.259260   0.611111   0.074074 ]   er: 0.000001
       (0, 51, Float32[-0.25925952 0.6111108 0.07407369], "jISolve, successful")
```

Gauss-Seidel iterative (system Ax = b)

```
# gsiterative.jl / CAMBarbosa
# description:
# » computing linear systems of equations, gauss-seidel iterative method
# » input: coefficient matrix 'A', result vector 'b' and optional guess to 'x' vector
# » output: status successful or error found
#           iteration number, 'x' iteration vector, iteration error
#           iteration number, 'x' solution vector
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s454

using Printf

function Show(k,n,nx,x)
  @printf("x(%03d): [",k)
  for i = 1:n
    @printf("%11.6f",x[i])
  end
  @printf(" ]   er: %.6f\n",nx)
end
```

```
20
21  function GSISolve(b,A,c = 0)
22  # iterative gauss-seidl for solve linear systema Ax=bn
23  # b[j], A[i,j],   1<=i<=n  1<=j<=n;  opcional fornecer vetor c[j] para estimativa inicial
24  # retorna, 0,x: calculo bem sucedido, vetor com o resultado de 'Ax = b'
25  #          1|2|3: erro
26    k,qt,em,nx,rs,n = 0,1000,1.0e-6,1.0,0,size(A)[1]
27    x = Matrix{Float32}(undef,1,n) # vetor com resultados
28    ms = ["GSISolve, successful"              # rs = 0
29          "GSISolve, error dimensions matrix"  # rs = 1
30          "GSISolve, error singular matrix"    # rs = 2
31          "GSISolve, non convergence" ]        # rs = 3
32    if ndims(c) == 0   c = [1.0 for i = 1:n]  end # quando vetor 'c' não é fornecido
33    if size(A)[2] != n || size(b)[ndims(b)] != n || size(c)[ndims(c)] != n # verifica consistencia  matriz
34      rs = 1 # inconsistencia na dimensao da matriz
35    else
36      y = [convert(Float32,b[i]) for i = 1:n] # vetor com resultados
37      z = [convert(Float32,c[i]) for i = 1:n] # vetor com estimativa iniciais para 'x'
38      M = [convert(Float32,A[i,j]) for i = 1:n, j = 1:n] # matriz com os coeficientes
39      for i = 1:n
40        if abs(A[i,i]) < em  rs = 2;  @goto(RTN)  end # encontro zero na diagonal
41      end
42      while em < nx && k < qt # erro maximo menor que norma do vetor 'x' e não atingiu qtd iteracoes
43        for i = 1:n # percorre as linhas da matriz
44          x[i] = 0.0
45          for j = 1:i-1 # percorre as colunas da linha 'i'
46            x[i] += M[i,j] * x[j] # subtrai produto dos outros termos da linha 'i'
47          end
48          for j = i+1:n # percorre as colunas da linha 'i'
49            x[i] += M[i,j] * z[j] # subtrai produto dos outros termos da linha 'i'
50          end
51          x[i] = (y[i] - x[i]) / M[i,i]
52        end
53        k,nx = k+1,0.0
54        for i = 1:n
55          if nx < abs(x[i] - z[i]) # norma, maior diferenca entre iteração corrente e anterior
56            nx = abs(x[i] - z[i])
57          end
58          z[i] = x[i] # salva valor da iteração corrente para comparação na iteração seguinte
59        end
60        Show(k,n,nx,x) # apresenta o vetor da iteração 'k'
61      end
62    end
63    if k == qt  rs = 3  end
64    @label(RTN)
65    return rs,k,x,ms[rs+1]
66  end
```

usage example:

```
julia> include("$pth/gsiterative.jl")
julia> b = [16 26 -19 -34];  A = [6 -2 2 4; 12 -8 6 10; 3 -13 9 3; -6 4 1 -18];
julia> GSISolve(b,A)
       x(001): [   2.000000   1.750000  -0.583333   1.578704 ]   er: 1.583333
       x(002): [   2.391975   1.873843  -0.728009   1.467528 ]   er: 0.391975
       x(003): [   2.555598   1.871801  -0.748440   1.411399 ]   er: 0.163623
       ...
       x(390): [   2.999979   1.000046  -1.999934   1.000021 ]   er: 0.000002
       x(391): [   2.999979   1.000044  -1.999936   1.000020 ]   er: 0.000002
       x(392): [   2.999980   1.000044  -1.999937   1.000020 ]   er: 0.000001
       (0, 392, Float32[2.99998 1.0000439 -1.9999368 1.0000199], "GSISolve, successful")
```

Polynomial interpolating Lagrange formula

```
# plagrange.jl / CAMBarbosa
# description:
# » polynomial interpolating formula with lagrange method
# » input: interpolation point 'a', 'x' 'y' array of coordenates points
# » output: status successful or error found
#          y = f(x) for given value of 'x'
# reference:
# » Ray,S.S.; 2016; Numerical Analysis with Algorithms and Programming; CRC Press; s91

function PILagrange(a,x,y)
# lagrange polynomial interpolation, retorna a interpolação y=f(a),
# para o vetor de pares de pontos y=f(x)
  rs,sm,er,n = 0,0.0,1.0e-6,size(x)[ndims(x)]
  ms = ["PILagrange, successful"                      # rs == 0
        "PILagrange, error dimensions vector"          # rs == 1
        "PILagrange, division by zero|small number" ]  # rs == 2
  if size(y)[ndims(y)] != n # verifica consistencia 'n' ddos vetores
    rs = 1
  else
    a = convert(Float32,a)
    x = [convert(Float32,x[i]) for i = 1:n]
    y = [convert(Float32,y[i]) for i = 1:n]
    for i = 1:n
      pl = 1.0
      for j = 1:n
        if j != i
          dn = x[i]-x[j]
          if abs(dn) < er  rs = 2;  @goto(RTN)  end
          pl *= (a-x[j]) / dn
        end
      end
      sm += pl * y[i]
    end
  end
  @label(RTN)
  return rs,sm,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/plagrange.jl")
julia> x = [0 0.1 0.2 0.3 0.4 0.5];  y = [1 1.3499 1.8221 2.4596 3.3201 4.4817];
julia> PILagrange(0.25,x,y)
       (0, 2.1169924981054327, "PILagrange, successful")
```

Polynomial interpolating Newton formula

```
# pnewton.jl / CAMBarbosa
# description:
# » polynomial interpolating formula with newton method
# » input: interpolation point 'x', 'xx' 'yy' array of coordenates points
# » output: status successful or error found
#          y = f(x) for given value of 'x'
# reference:
# » Cheney,W., Kincaid,D.; 2008; Numerical Mathematics Computing, 6e; Thomson Brooks/Cole; s128

function PINewton(x,xx,yy)
# newton polynomial interpolation, retorna a interpolação y=f(x),
# para o vetor de pares de pontos y=f(x)
  rs,pl,er,n = 0,0.0,1.0e-6,size(xx)[ndims(xx)]
```

```
  ms = ["PINewton, successful"                      # rs == 0
        "PINewton, error dimensions vector"         # rs == 1
        "PINewton, division by zero|small number" ]  # rs == 2
  if size(yy)[ndims(yy)] != n # verifica consistencia 'n' ddos vetores
    rs = 1
  else
    x = convert(Float32,x)
    xx = [convert(Float32,xx[i]) for i = 1:n]
    yy = [convert(Float32,yy[i]) for i = 1:n]
    cf = [yy[i] for i = 1:n]
    for j = 1:n
      for i = n:-1:j+1
        ax = (xx[i] - xx[i-j])
        if abs(ax) < er  rs = 2;  @goto(RTN)  end
        cf[i] = (cf[i] - cf[i-1]) / ax
      end
    end
    pl = cf[n]
    for i = n-1:-1:1
      pl = pl * (x-xx[i]) + cf[i]
    end
  end
  @label(RTN)
  return rs,pl,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/pnewton.jl")
julia> x = [0 0.1 0.2 0.3 0.4 0.5];  y = [1 1.3499 1.8221 2.4596 3.3201 4.4817];
julia> PINewton(0.25,x,y)
       (0, 2.1169925f0, "PINewton, successful")
```

Polynomial least squares

```
# pleastsquares.jl / CAMBarbosa
# description:
# » polynomial approximation with least squares method
# » input: polynomial degree 'n', 'x' 'y' array of coordenates points
# » output: status successful or error found
#           polynomial 'n+1' coefficients
# reference:
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s501

function PLeastSquares(n,x,y)
# polynomial least squares, calcula os coeficientes para o polinomio de grau 'n'
# cuja curva é a melhor aproximacao dos pares de pontos nos vetores 'x' e 'y'
  rs,m = 0,size(x)[ndims(x)];  c = Matrix{Float32}(undef,n+1,1)
  ms = ["PLeastSquares, successful"
        "PLeastSquares, vector inconsistent dimensions"
        "PLeastSquares, polynomial degree out of range"
        "PLeastSquares, GSISolve error" ]
  function SXE(v,e) # sum x^e
    sm = 0.0;  for i = 1:m  sm += v == false ? x[i]^e : y[i]*x[i]^e  end
    return sm
  end
  if size(y)[ndims(y)] != m # verifica consistencia 'n' ddos vetores
    rs = 1
  elseif !(0 < n < 6 && n < m-1) # verifica grau do polinomio
        rs = 2
      else
        x = [convert(Float32,x[i]) for i = 1:m]
        y = [convert(Float32,y[i]) for i = 1:m]
```

```
        A = Matrix{Float32}(undef,n+1,n+1);  b = Array{Float32}(undef,n+1)
        for i = 1:n+1
          for j = 1:n+1
            A[i,j] = SXE(false,i+j-2)
            if j == 1  b[i] = SXE(true,i+j-2)  end
          end
        end
        rs,k,c,mx = GSISolve(b,A)
        if rs != 0  rs = 3  end
      end
  return rs,c,ms[rs+1]
end
```

usage example:

```
julia> include("$pth/gsiterative.jl")
julia> include("$pth/pleastsquares.jl")
julia> x = [0 0.1 0.2 0.3 0.4 0.5];  y = [1 1.3499 1.8221 2.4596 3.3201 4.4817];
julia> PLeastSquares(2,x,y)
       x(001): [   2.063900   2.700600   2.212541 ]   er: 1.700600
       x(002): [   1.527600   3.667196   3.003972 ]   er: 0.966596
       x(003): [   1.213404   4.200328   3.543841 ]   er: 0.539869
       ...
       x(384): [   1.028408   1.834327  10.020837 ]   er: 0.000002
       x(385): [   1.028408   1.834326  10.020839 ]   er: 0.000002
       x(386): [   1.028408   1.834325  10.020840 ]   er: 0.000001
       (0, Float32[1.0284082 1.8343253 10.02084], "PLeastSquares, successful")
       => p(x) = 1.0284082 + 1.8343253*x + 10.02084*x^2
```

Integration trapezoid

```
# itrapezoid.jl / CAMBarbosa
# description: integration with trapezoid rule method
# » input: lower upper limits of integral [a,b] and integrand function
# » output: status successful
#           value of integral
# reference:
# » Cheney,W., Kincaid,D.; 2008; Numerical Mathematics Computing, 6e; Thomson Brooks/Cole; s190

function ITrapezoid(a,b,fct)
# calcula a integral da função 'fct(x)' no intervalo '[a,b]'
  rs,n = 0,50;   h::Float32,sm::Float32 = (b - a)/n,(fct(a) + fct(b))/2.0
  ms = ["ITrapezoid, successful"]   # rs == 0
  for i = 1:n-1
    sm += fct(a + i*h)
  end
  sm *= h
  return rs,sm,ms[1]
end
```

usage example:

```
julia> include("$pth/itrapezoid.jl")
julia> fct(x) = exp(x) * cos(x)
julia> ITrapezoid(0,1.2,fct)
       (0, 1.6486361f0, "ITrapezoid, successful")
```

Integration Simpson 1/3

```
# iotsimpson.jl / CAMBarbosa
# description: integration with one third simpson rule method
# » input: lower upper limits of integral [a,b] and integrand function
# » output: status successful
```

```
#           value of integral
# reference:
# » Ray,S.S.; 2016; Numerical Analysis with Algorithms and Programming; CRC Press; s200

function IOTSimpson(a,b,fct)
# calcula a integral da função 'fct(x)' no intervalo '[a,b]'
  rs,n,as,bs = 0,50,0.0,0.0 # 'n' multiplo de 2
  h::Float32 = (b-a)/n;  f = Array{Float32}(undef,n+1);
  ms = ["IOTSimpson, successful"]   # rs == 0
  for i = 0:n  f[i+1] = fct(a + i*h)  end
  for i = 1:n-1
    if i % 2 == 0  as += f[i+1]
    else           bs += f[i+1]  end
  end
  it = h * (f[1] + f[n+1] + 2 * as + 4 * bs) / 3.0
  return rs,it,ms[1]
end
```

usage example:

```
julia> include("$pth/iotsimpson.jl")
julia> fct(x) = exp(x) * cos(x)
julia> IOTSimpson(0,1.2,fct)
       (0, 1.6487744131354252, "IOTSimpson, successful")
```

Integration Simpson 3/8

```
# itesimpson.jl / CAMBarbosa
# description: integration with three eighth simpson rule method
# » input: lower upper limits of integral [a,b] and integrand function
# » output: status successful
#           value of integral
# reference:
# » Ray,S.S.; 2016; Numerical Analysis with Algorithms and Programming; CRC Press; s208

function ITESimpson(a,b,fct)
# calcula a integral da função 'fct(x)' no intervalo '[a,b]'
  rs,n,as,bs = 0,51,0.0,0.0 # 'n' multiplo de 3
  h::Float32 = (b-a)/n;  f = Array{Float32}(undef,n+1);
  ms = ["ITESimpson, successful"]   # rs == 0
  for i = 0:n  f[i+1] = fct(a + i*h)  end
  for i = 1:n-1
    if i % 3 == 0  as += f[i+1]
    else           bs += f[i+1]  end
  end
  it = 3*h * (f[1] + f[n+1] + 2 * as + 3 * bs) / 8.0
  return rs,it,ms[1]
end
```

usage example:

```
julia> include("$pth/itesimpson.jl")
julia> fct(x) = exp(x) * cos(x)
julia> ITESimpson(0,1.2,fct)
       (0, 1.6487744839956293, "ITESimpson, successful")
```

Derivative (Richardson extrapolation)

```
# derivative.jl / CAMBarbosa
# description:
# » computing the function deivative with with richardson extrapolation method
# » input: function, 'x' position
```

```
# » output: status successful or error found
#          the derivative two dimensional triangular array
# reference:
# » Cheney,W., Kincaid,D.; 2008; Numerical Mathematics Computing, 6e; Thomson Brooks/Cole; s165

using Printf

function Show(h,n,D)
# apresenta matriz A
  for i = 1:n
    if i != 1  h /= 2.0  end
    @printf("%.4f  ",h)
    for j = 1:i
      @printf("%8.6f  ",D[i][j])
    end
    if 1 < i  @printf("er:%.6f",abs(D[i][i] - D[i][i-1]))  end
    @printf("\n")
  end
  @printf("\n")
end

function Derivative(x,fct)
# calcula a derivada da função 'fct(x)' para o valor de 'x'
  rs,k,i,em,er,D = 0,1.0,0,1.0e-6,1.0,Array{Any}(undef,1)
  h,ms = k,["Derivative, successful"     # rs == 0
            "Derivative, not converge" ] # rs == 1
  while em < er && em < h
    i += 1
    if 1 < i  D = vcat(D,Array{Any}(undef,1))  end
    D[i] = Array{Float32}(undef,1,i)
    D[i][1] = (fct(x+h) - fct(x-h)) / (2.0 * h)
    for j = 2:i
      D[i][j] = D[i][j-1] + (D[i][j-1] - D[i-1][j-1]) / (4^j - 1)
    end
    h /= 2.0
    if 1 < i  er = abs(D[i][i] - D[i][i-1])  end
  end
  if !(em < h)  rs = 1
  else          Show(k,i,D)  end
  return rs,D[i][i],ms[rs+1]
end
```

usage example:

```
julia> include("$pth/derivative.jl")
julia> fct(x) = exp(-x) * sin(x);
julia> Derivative(0.6,fct)
       1.0000  0.391377
       0.5000  0.206324  0.193987  er:0.012337
       0.2500  0.158945  0.155786  0.155180  er:0.000606
       0.1250  0.147043  0.146250  0.146099  0.146063  er:0.000036
       0.0625  0.144065  0.143866  0.143828  0.143819  0.143817  er:0.000002
       0.0313  0.143320  0.143270  0.143261  0.143258  0.143258  0.143258  er:0.000000
       (0, 0.1432577f0, "Derivative, successful")
```

Eigenvalue and eigenvectors (basic QR decomposition, Gram-Schmidt orthonormalization)

```
# eigenvv.jl / CAMBarbosa
# description:
# » computing eigenvalues of a matrix with basic QR decomposition method
# » computing eigenvectors of a matrix for each eigenvaluei, with iterative gauss-seidel method
# » input: a square matrix, the matrix must be non-singular and have no row or column multiplicity
# » output: status successful or error found
#          set of eigenvalues and corresponding eigenvectors
```

```
# reference:
# » Bronson,R., Costa,G.B., Saccoman,J.T.; 2014;
#     Linear Algebra, Algorithms, Applications and Techniques, 3th edition; Elsevier; s351
# » Burden,R.L., Faires,J.D.; 2011; Numerical Analysis, 9e; Cengage Learning; s454

using Printf

struct Evl  cp::Bool;  vl::Any  end # complex:false|true, eigenvalue
struct Egn  vl::Evl;   vt::Any  end # eigenvalue, eigenvector
ger = 1.0e-6

function EShow(n,E)
# apresenta os autovalores
  function PTF(c,vl,tx)
    if c == false # numero real
      @printf("%s%.6f",tx,vl)
    else # numero complexo
      @printf("%s%.6f%+.6fim",tx,real(vl),imag(vl))
    end
  end
  @printf("%s","EigenValues | EigenVectors\n")
  for i = 1:n
    @printf("%4d:",i)
    PTF(E[i].vl.cp,E[i].vl.vl,"  ")
    @printf("%s"," [")
    for j = 1:n  PTF(E[i].vl.cp,E[i].vt[j],j == 1 ? " " : "  ")  end
    @printf("%s"," ]'\n")
  end
end

function SDRoots(px)
# retorna as raizes do polinomio do segundo grau
  qv,rs = 1,Array{Evl}(undef,2)
  if abs(px[1]) < ger # nao é equacao 2 grau; calcula raiz para equacao da reta y = bx + c
    rs = [Evl(false,abs(px[2]) < ger ? 0.0 : px[3]/px[2])]
  else
    qv,xx = 2,2.0 * px[1];  dt = px[2]^2 - 4.0 * px[1] * px[3] # calcula 'dt'
    if abs(dt) < ger # 'dt' igual px[1] zero; raizes iguais
      yy = -px[2] / xx;  rs = [Evl(false,yy)  Evl(false,yy)]
    elseif 0.0 < dt # 'dt' maior que zero; raizes diferentes
          rs =[Evl(false,(-px[2] + dt^0.5) / xx)  Evl(false,(-px[2] - dt^0.5) / xx)]
        else # 'dt' menor que zero; raizes conjugado complexo
          yy = -px[2] / xx;  zz = abs(dt)^0.5 / xx
          rs = [Evl(true,complex(yy,zz))  Evl(true,complex(yy,-zz))]
        end
  end
  return qv,rs
end

function Multiplicity(n,A)
# verifica se uma linha|coluna é multipla de outra
  rs,rd = false,1.0e+6 # arredondamento
  for i = 1:n-1 # multiplicidade entre linhas
    for k = i+1:n # percorre as linhas abaixo da linha corrente
      j::Int8,mt::Int8 = 2,0 # flag para nao multiplicidade em linha|coluna
      fl::Float32 = trunc(Int64,A[k,1]*rd) / trunc(Int64,A[i,1]*rd) # fator elementos primeira linha
      fc::Float32 = trunc(Int64,A[1,k]*rd) / trunc(Int64,A[1,i]*rd) # fator elementos primeira coluna
      while j < n+1 # interrompe quando faor linha&coluna sao diferentes
        mt |= (fl != trunc(Int64,A[k,j]*rd) / trunc(Int64,A[i,j]*rd))      # diferente fator linha
        mt |= (fc != trunc(Int64,A[j,k]*rd) / trunc(Int64,A[j,i]*rd)) << 1 # diferente fator coluna
        if mt == 0b11  break  end # encontro linha|coluna com fator diferente
        j += 1;
      end
      if j == n+1  rs = true;  @goto(RTN)  end # linha|coluna 'k' é multipla da linha|coluna 'i'
```

```
    end
  end
  @label(RTN)
  return rs
end

function Converge(n,A)
# verifica convergencia na ultima e penultima linha de A
# retorna: 0: nao converge, 1:uma raiz, 2:duas raizes
  rs = 0
  if n == 1  rs = 1
  elseif n == 2  rs = 2
    else # 1 < n
      for i = n:-1:n-1 # verifica convergencia nas duas ultimas linhas
        j = 1
        while j < i
          if ger < abs(A[i,j])  break  end # encontrou valor maior que zero
          j += 1
        end
        if i == n # ultima linha
          if j == i || j+1 == i  rs = 1+(i-j)  end # qtd encontrada na ultima linha
        else # i == n-1, penultima linha
          if j < i # parou antes da coluna 'i'
            if rs == 2    rs = 0  end # desmarca, não tem bloco de 4 elementos
          elseif rs == 1  rs = 2  end # j == i, redefine para b loco de 4 elementos
        end
      end
    end
  return rs
end

function QRfactorize(n,A)
# A = Q*R (Q:orthogonal matrix, R:upper triangular matrix, Q^(-1) = transpose(Q))
  rs,Q,R = 0,zeros(Float32,n,n),zeros(Float32,n,n)
  function CIP(n,L,R) # column inner product
    cp = 0.0
    for i = 1:n  cp += L[i] * R[i]  end
    return cp
  end
  for i = 1:n
    for j = i:n
      if i == j
        R[i,j]   = (CIP(n,A[1:n,j],A[1:n,j]))^0.5 # (A[1:n,j] .* A[1:n,j])^0.5
        if abs(R[i,j]) < ger  rs = 3;  @goto(CTN)  end
        Q[1:n,i] = A[1:n,j] / R[i,j]
      else
        R[i,j]   = CIP(n,A[1:n,j],Q[1:n,i]) # A[1:n,j] .* Q[1:n,i]
        A[1:n,j] = A[1:n,j] - R[i,j] * Q[1:n,i]
      end
    end
  end
  @label(CTN)
  return rs,Q,R
end

function QRiterative(n,A)
# determina as raizes da equacao característica da matriz
  k,lx,rs,qt,lb = 0,0,0,1000,Array{Evl}(undef,n)
  function SDR(nr)
    if nr == 1
      lb[n] = Evl(false,A[n,n]);  n -= 1
    else # nr == 2
      qv,lx = SDRoots([1.0
                       -(A[n-1,n-1]+A[n,n])
```

```
                        A[n-1,n-1]*A[n,n]-A[n-1,n]*A[n,n-1]])
      if qv == 1 # encontrou uma raiz
        lb[n] = lx[1];  n -= 1
      else for i = 1:2 # encontrou 2 raizes real|complexa
            lb[n] = lx[i];  n -= 1
          end
      end
    end
  end
  while k < qt && 0 < n && rs == 0
    if 0 < n < 3  SDR(n)
    else
      rs,Q,R = QRfactorize(n,A)
      A      = R * Q
      rx     = Converge(n,A)
      if 0 < rx < 3  SDR(rx)  end
    end
    k += 1
  end
  return k == qt ? 4 : rs,lb
end

function GSiSolve(c,A)
# gauss-seidl iterative for solve linear systema Ax=b; b[j], A[i,j],   1<=i<=n  1<=j<=n
# retorna, 0,x: calculo bem sucedido, vetor com o resultado de 'Ax = b'
#          1|5: erro
  k,qt,nx,rs,n = 0,1000,1.0,0,size(A)[1]
  bb = [c == false ? 0.0 : 0.0+0.0im for i = 1:n]
  cc = [c == false ? 1.0 : 1.0+0.0im for i = 1:n] # estimativa inicial
  xx = Array{c == false ? Float32 : Complex}(undef,n)
  if size(A)[2] != n || size(bb)[ndims(bb)] != n # verifica consistencia 'n,n' da matriz
    rs = 1 # inconsistencia na dimensao da matriz
  else
    for i = 1:n
      if abs(A[i,i]) < ger  rs = 3;  @goto(RTN)  end # encontro zero na diagonal
    end
    while ger < nx && k < qt # erro maximo menor que norma do vetor 'x' e nao atingiu qtd iteracoes
      for i = 1:n # percorre as linhas da matriz
        xx[i] = c == false ? 0.0 : 0.0+0.0im
        for j = 1:i-1  xx[i] += A[i,j] * xx[j]  end # subtrai produto dos outros termos da linha 'i'
        for j = i+1:n  xx[i] += A[i,j] * cc[j]  end # subtrai produto dos outros termos da linha 'i'
        xx[i] = (bb[i] - xx[i]) / A[i,i]
      end
      k,nx = k+1,0.0
      for i = 1:n
        # calcula norma, obtem a maior diferenca entre iteracao corrente e anterior
        if nx < abs(xx[i] - cc[i])  nx = abs(xx[i] - cc[i])  end
        cc[i] = xx[i] # salva valor da iteracao corrente para comparacao na iteracoa seguinte
      end
    end
  end
  if k == qt  rs = 5  end
  @label(RTN)
  return rs,xx
end

function EigenVV(A)
# encontra autovalores da matriz 'A' pelo método QR básico
# encontra autovetors para cada auto valor usando métoto gaus-seidel iterativo
# 'A' deve ser matriz quadrada com linhas|colunas linearmente independentes
# quando processamento bem sucedido, apresenta os autovalores|autovetores da matrix
  rs,n = 0,size(A)[1];  lb = Array{Evl}(undef,n);  E = Array{Egn}(undef,n)
  ms = [ "EigenVV, successful calculation"          # rs = 0
         "EigenVV, matrix inconsistent dimensions"  # rs = 1
```

```
          "EigenVV, matrix line|column multiplicity"  # rs = 2
          "EigenVV, division by zero|small number"    # rs = 3
          "EigenVV, eigenvalues not converge"         # rs = 4
          "EigenVV, eigenvectors not converge" ]      # rs = 5
   if size(A)[2] != n # verifica consistencia 'n,n' da matriz
     rs = 1 # inconsistencia na dimensao da matriz
   elseif Multiplicity(n,A) == true
         rs = 2 # infinitas solucoes
       else
         A     = [convert(Float32,A[i,j]) for i = 1:n, j = 1:n] # converte para real
         rs,lb = QRiterative(n,A[1:n,1:n])
         if rs == 0
           I = [i != j ? 0.0 : 1.0 for i = 1:n, j = 1:n]  # matrix identidade
           for j = 1:n
             rs,X = GSiSolve(lb[j].cp,A - lb[j].vl * I) # determina autovetor 'X[j]'
             if rs != 0  @goto(RTN)  end
             E[j] = Egn(lb[j],X) # salva altovalor|autovetor
           end
         end
       end
   @label(RTN)
   if rs == 0  EShow(n,E)  end # processamento bem sucedido
   return rs,ms[rs+1]
 end
```

usage example:

```
julia> include("$pth/eigenvv.jl")
julia> A = [5 4 2; 4 5 2; 2 2 2];
julia> EigenVV(A)
  EigenValues | EigenVectors
     1:  10.000000 [ 1.333333  1.333333  0.666667 ]'
     2:  1.000000 [ -1.500000  1.000000  1.000000 ]'
     3:  1.000000 [ -1.500000  1.000000  1.000000 ]'
  (0, "EigenVV, successful calculation")
julia> A = [1 3 -7; -3 4 1; 2 -5 3];
julia> EigenVV(A)
  EigenValues | EigenVectors
     1:  3.989323-5.560120im [ -0.415817-0.598289im  -0.334273+0.315356im  0.509535+0.060365im ]'
     2:  3.989323+5.560120im [ -0.415817+0.598289im  -0.334273-0.315356im  0.509535-0.060365im ]'
     3:  0.021354 [ 7.386867  4.793925  3.087272 ]'
  (0, "EigenVV, successful calculation")
```