```
!
! To compile and run:
!   pgf95 001.f95 -o 001.exe
!   ./001.exe
!
!------------------------
! filename: src/001.f95
!
! finite difference method. 1st derivative. 1st order finite difference.
!
!------------------------
!

module constants
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
end module constants


program main
  use constants
  implicit none
  real(DP) :: d1f
  real(DP) :: exact_solution
  real(DP) :: x0 = 2.0_DP
  real(DP) :: dx = 0.01_DP

!  d1f = ???

  exact_solution =  1 + x0 + x0**2 + x0**3

  print *,'    f(x0) = ', f(x0)
  print *,' d1f at x0 = ', d1f
  print *,' solution = ', exact_solution
  print *,'   error = ', abs(exact_solution - d1f)

contains

  function f(x)
    real(DP), intent(in) :: x
    real(DP) :: f

    f = 1 + x + (x**2)/2.0_DP + (x**3)/3.0_DP + (x**4)/4.0_DP
  end function f
end program main
```

```
!
! To compile and run:
!   pgf95 002.f95 -o 002.exe && ./002.exe
!
!------------------------
!
! finite difference method. 2nd derivatve. 2nd order finite difference.
!
!------------------------
!

module constants
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
end module constants


program main
  use constants
  implicit none

  real(DP) :: d2f
  real(DP) :: exact_solution
  real(DP) :: x0 = 2.0_DP
  real(DP) :: dx = 0.01_DP

  d2f = ( f(x0+dx) - 2*f(x0) + f(x0-dx) ) / (dx*dx)

  exact_solution =  1 + 2*x0 + 3*(x0**2)

  print *,'    f(x0) = ', f(x0)
  print *,' d2f at x0 = ', d2f
  print *,'  solution = ', exact_solution
  print *,'     error = ', abs(exact_solution - d2f)

contains

  function f(x)
    real(DP), intent(in) :: x
    real(DP) :: f

    f = 1 + x + (x**2)/2.0_DP + (x**3)/3.0_DP + (x**4)/4.0_DP
  end function f
end program main
```

```
!
! To compile and run:
!   pgf95 -Mbounds 003.f95 && ./a.out
!
!------------------------
!
! to make x (1d) grid system.
!
!------------------------
!

module constants
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
end module constants

module ut
  use constants
  implicit none

contains

  subroutine ut__assert(condition, message)
    logical, intent(in) :: condition
    character(len=*), intent(in) :: message

    if ( .not.condition ) then
       stop message ! print out the message and die.
    end if
  end subroutine ut__assert
end module ut


module grid
  use constants
  implicit none
  private
  public :: grid__initialize,   &
            grid__dx,           &
            grid__nx

  integer  :: nx
  real(DP) :: dx

contains

  subroutine grid__initialize(xlength, nx_mesh)
    real(DP), intent(in) :: xlength
    integer,  intent(in) :: nx_mesh

    integer  :: i

    nx = nx_mesh
    dx = xlength / real(nx,DP)
  end subroutine grid__initialize

  function grid__nx()
    integer :: grid__nx
    grid__nx = nx
  end function grid__nx
```



```
  function grid__dx()
    real(DP) :: grid__dx
    grid__dx = dx
  end function grid__dx
end module grid


program main
  use constants
  use ut
  use grid
  implicit none

  integer :: i, nx
  real(DP) :: xlen = 10.0_DP ! meter

  print *, ' enter nx:'
  read(5,*) nx

  call ut__assert(nx>0, 'nx out of range')

  call grid__initialize(xlen,nx)

  print *,' nx = ', grid__nx()
  print *,' dx = ', grid__dx()
end program main
```

```
!
! To compile and run:
!   pgf95 -Mbounds 004.f95 && ./a.out
!
!------------------------
!
! to make a (x-y) 2d grid system.
!
!------------------------
!

module constants
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
end module constants

module ut
  use constants
  implicit none

contains

  subroutine ut__assert(condition, message)
    logical, intent(in) :: condition
    character(len=*), intent(in) :: message

    if ( .not.condition ) then
       stop message ! print out the message and die.
    end if
  end subroutine ut__assert
end module ut


module grid
  use constants
  implicit none
  private
  public :: grid__initialize,           &
            grid__d,                    &
            grid__n

  integer  :: nx, ny
  real(DP) :: dx, dy

contains

  subroutine grid__initialize(xlength, ylength, nx_mesh, ny_mesh)
    real(DP), intent(in) :: xlength, ylength
    integer,  intent(in) :: nx_mesh,   ny_mesh

    integer  :: i, j

    nx = nx_mesh
    ny = ny_mesh

    dx = xlength / real(nx,DP)
    dy = ylength / real(ny,DP)
  end subroutine grid__initialize

  function grid__n(which)
    character, intent(in) :: which
```



```
    integer :: grid__n

    select case (which)
    case ('x')
       grid__n = nx
    case ('y')
       grid__n = ny
    case default
       stop 'grid__n arg incorrect.'
    end select
  end function grid__n

  function grid__d(which)
    character, intent(in) :: which
    real(DP) :: grid__d

    select case (which)
    case ('x')
       grid__d = dx
    case ('y')
       grid__d = dy
    case default
       stop 'grid__d arg incorrect.'
    end select
  end function grid__d
end module grid


program main
  use constants
  use ut
  use grid
  implicit none

  integer :: i, j, nx, ny
  real(DP) :: xlen = 10.0_DP ! meter
  real(DP) :: ylen =  8.0_DP ! meter

  print *, ' enter nx:'
  read(5,*) nx
  print *, ' enter ny:'
  read(5,*) ny

  call ut__assert(nx>0 .and. ny>0, 'nx,ny out of range.')

  call grid__initialize(xlen,ylen,nx,ny)

  print *,' nx = ', grid__n('x')
  print *,' dx = ', grid__d('x')
  print *,' ny = ', grid__n('y')
  print *,' dy = ', grid__d('y')
end program main
```

```
!
! To compile and run:
!   pgf95 -Mbounds 005.f95 && ./a.out
!
!------------------------
!
! to make temperature and heat source fields.
!
!------------------------
!

module constants
  implicit none
  integer,  parameter :: SP = kind(1.0)
  integer,  parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), parameter :: PI = 3.1415926535897932384626433832795028_DP
  real(DP), parameter :: TWOPI = PI*2
end module constants

module ut
  use constants
  implicit none

contains

  subroutine ut__assert(condition, message)
    logical, intent(in) :: condition
    character(len=*), intent(in) :: message

    if ( .not.condition ) then
       stop message ! print out the message and die.
    end if
  end subroutine ut__assert
end module ut


module grid
  use constants
  implicit none
  private
  public :: grid__initialize, grid__d, grid__n

  integer  :: nx, ny
  real(DP) :: dx, dy

contains

  subroutine grid__initialize(xlength, ylength, nx_mesh, ny_mesh)
    real(DP), intent(in) :: xlength, ylength
    integer,  intent(in) :: nx_mesh, ny_mesh
    integer  :: i, j
    nx = nx_mesh
    ny = ny_mesh
    dx = xlength / real(nx,DP)
    dy = ylength / real(ny,DP)
  end subroutine grid__initialize

  function grid__n(which)
    character, intent(in) :: which
    integer :: grid__n

    select case (which)
```



```
    case ('x')
       grid__n = nx
    case ('y')
       grid__n = ny
    case default
       stop 'grid__n arg incorrect.'
    end select
  end function grid__n

  function grid__d(which)
    character, intent(in) :: which
    real(DP) :: grid__d

    select case (which)
    case ('x')
       grid__d = dx
    case ('y')
       grid__d = dy
    case default
       stop 'grid__d arg incorrect.'
    end select
  end function grid__d
end module grid


module field
  use constants
  use ut
  use grid
  implicit none
  private
  public :: field__initialize,  &
            field__statistics
  public :: tmp, src

  real(DP), dimension(:,:), allocatable :: tmp, src

contains

  subroutine boundary_condition
    integer :: nx, ny
    nx = grid__n('x')
    ny = grid__n('y')
    !
    !   +--------(D)-------+
    !   |                  |
    !   |                  |
    !   |                  |
    !  (A)                (B)
    !   |                  |
    !   |                  |
    !   |                  |
    !   |                  |
    !   +--------(C)-------+
    !
    tmp(     0,0:ny) = 0.0_DP  ! (A)
    tmp(    nx,0:ny) = 0.0_DP  ! (B)
    tmp(1:nx-1,   0) = 0.0_DP  ! (C)
    tmp(1:nx-1,  ny) = 0.0_DP  ! (D)
  end subroutine boundary_condition

  subroutine field__initialize
```

```
    integer  :: nx, ny, i, j
    nx = grid__n('x')
    ny = grid__n('y')
    allocate(tmp(0:nx,0:ny)) ! temperature field
    allocate(src(0:nx,0:ny)) ! heat source field
    tmp(:,:) = 0.0_DP
    src(:,:) = 0.0_DP
    call iSet_src
    call iSet_tmp
    call boundary_condition

  contains

    subroutine iSet_src
      src(:,:) = 0.1_DP  ! constant heat source
    end subroutine iSet_src

    subroutine iSet_tmp
      integer  :: i, j
      real(DP) :: dx, dy, x, y, xlen, ylen
      dx = grid__d('x')
      dy = grid__d('y')
      xlen = dx*nx  ! here we suppose xpos(0) = xmin = 0.0
      ylen = dy*ny  ! here we suppose ypos(0) = ymin = 0.0
      do j = 0 , ny
         y = dy*j
         do i = 0 , nx
            x = dx*i
            tmp(i,j) = sin(PI*x/xlen)*sin(PI*y/ylen)
         end do
      end do
    end subroutine iSet_tmp
  end subroutine field__initialize

  subroutine field__statistics
    integer :: total_grid_points
    total_grid_points = grid__n('x') * grid__n('y')
    print *,' temp  max: ', maxval(tmp(:,:))
    print *,' temp  min: ', minval(tmp(:,:))
    print *,' temp mean: ', sum(tmp(:,:)) / total_grid_points
  end subroutine field__statistics
end module field


program main
  use constants
  use ut
  use grid
  use field
  implicit none

  integer  :: i, j, nx, ny
  real(DP) :: xlen = 10.0_DP ! meter
  real(DP) :: ylen =  8.0_DP ! meter

  print *, ' enter nx:'
  read(5,*) nx
  print *, ' enter ny:'
  read(5,*) ny
  call ut__assert(nx>0 .and. ny>0, 'nx/ny out of range.')

  call grid__initialize(xlen,ylen,nx,ny)
```



```
  call field__initialize
  call field__statistics
end program main
```

```
!
! To compile and run:
!   pgf95 -Mbounds 006.f95 && ./a.out
! to check the array bounds, or just
!   pgf95 006.f95 && ./a.out
!
!------------------------
!
! time development of the temperature field.
!
!------------------------
!

module constants
  implicit none
  integer,  parameter :: SP = kind(1.0)
  integer,  parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), parameter :: PI = 3.141592653589793238462643383279502_DP
  real(DP), parameter :: TWOPI = PI*2
  real(DP), parameter :: K_THERM = 1.0_DP  ! thermal diffusivity
end module constants

module ut
  use constants
  implicit none

contains

  subroutine ut__assert(condition, message)
    logical, intent(in) :: condition
    character(len=*), intent(in) :: message

    if ( .not.condition ) then
       stop message ! print out the message and die.
    end if
  end subroutine ut__assert
end module ut


module grid
  use constants
  implicit none
  private
  public :: grid__initialize, grid__d, grid__n

  integer  :: nx, ny
  real(DP) :: dx, dy

contains

  subroutine grid__initialize(xlength, ylength, nx_mesh, ny_mesh)
    real(DP), intent(in) :: xlength, ylength
    integer,  intent(in) :: nx_mesh, ny_mesh
    integer  :: i, j
    nx = nx_mesh
    ny = ny_mesh
    dx = xlength / real(nx,DP)
    dy = ylength / real(ny,DP)
  end subroutine grid__initialize

  function grid__n(which)
    character, intent(in) :: which
```



```
    integer :: grid__n

    select case (which)
    case ('x')
       grid__n = nx
    case ('y')
       grid__n = ny
    case default
       stop 'grid__n arg incorrect.'
    end select
  end function grid__n

  function grid__d(which)
    character, intent(in) :: which
    real(DP) :: grid__d

    select case (which)
    case ('x')
       grid__d = dx
    case ('y')
       grid__d = dy
    case default
       stop 'grid__d arg incorrect.'
    end select
  end function grid__d
end module grid


module field
  use constants
  use ut
  use grid
  implicit none
  private
  public :: field__initialize,  &
            field__statistics,  &
            field__time_advance
  public :: tmp, src

  real(DP), dimension(:,:), allocatable :: tmp, src

contains

  subroutine boundary_condition
    integer :: nx, ny
    nx = grid__n('x')
    ny = grid__n('y')
    !
    !   +--------(D)-------+
    !   |                  |
    !   |                  |
    !   |                  |
    !  (A)                (B)
    !   |                  |
    !   |                  |
    !   |                  |
    !   |                  |
    !   +--------(C)-------+
    !
    tmp(     0,0:ny) = 0.0_DP  ! (A)
    tmp(    nx,0:ny) = 0.0_DP  ! (B)
    tmp(1:nx-1,    0) = 0.0_DP  ! (C)
```

```
    tmp(1:nx-1,  ny) = 0.0_DP  ! (D)
  end subroutine boundary_condition

  subroutine field__initialize
    integer  :: nx, ny, i, j
    nx = grid__n('x')
    ny = grid__n('y')
    allocate(tmp(0:nx,0:ny)) ! temperature field
    allocate(src(0:nx,0:ny)) ! heat source field
    tmp(:,:) = 0.0_DP
    src(:,:) = 0.0_DP
    call iSet_src
    call iSet_tmp
    call boundary_condition

  contains

    subroutine iSet_src
      src(:,:) = 0.0_DP  ! constant heat source
    end subroutine iSet_src

    subroutine iSet_tmp
      integer  :: i, j
      real(DP) :: dx, dy, x, y, xlen, ylen
      dx = grid__d('x')
      dy = grid__d('y')
      xlen = dx*nx  ! here we suppose xpos(0) = xmin = 0.0
      ylen = dy*ny  ! here we suppose ypos(0) = ymin = 0.0
      do j = 0 , ny
         y = dy*j
         do i = 0 , nx
            x = dx*i
            tmp(i,j) = sin(PI*x/xlen)*sin(PI*y/ylen)
         end do
      end do
    end subroutine iSet_tmp
  end subroutine field__initialize

  recursive subroutine field__statistics(which)
    character(len=*), intent(in) :: which
    integer :: total_grid_points
    total_grid_points = grid__n('x') * grid__n('y')
    select case (which)
    case ('max')
       print *,' temp max: ', maxval(tmp(:,:))
    case ('min')
       print *,' temp min: ', minval(tmp(:,:))
    case ('mean')
       print *,' temp mean: ', sum(tmp(:,:)) / total_grid_points
    case default
       call field__statistics('max')
       call field__statistics('min')
       call field__statistics('mean')
    end select
  end subroutine field__statistics

  subroutine field__time_advance(dt,debug)
    real(DP), intent(in) :: dt
    character(len=*), intent(in), optional :: debug
    integer  :: i, j
    real(DP) :: dx, dy
    integer, save  :: nx, ny
```



```
    real(DP), save :: alpha_x, alpha_y, beta
    real(DP) :: beta2
    logical :: first_time = .true.  ! automatically 'save'd.
    real(DP), dimension(:,:), allocatable, save :: tmp0

    if ( first_time ) then
       nx = grid__n('x')
       ny = grid__n('y')
       dx = grid__d('x')
       dy = grid__d('y')
       alpha_x = K_THERM / (dx*dx)
       alpha_y = K_THERM / (dy*dy)
       beta = 2 * (alpha_x + alpha_y)
       allocate(tmp0(0:nx,0:ny))
       first_time = .false.
    end if

!----<origian form>-----------------------------------------------------------
!   tmp_new(i,j) = tmp(i,j)                                                 &
!                 + ( alpha_x * ( tmp(i+1,j) - 2*tmp(i,j) + tmp(i-1,j) )  &
!                   + alpha_y * ( tmp(i,j+1) - 2*tmp(i,j) + tmp(i,j-1) )  &
!                   + src(i,j) ) * dt
!----</origian form>----------------------------------------------------------

    beta2 = 1.0_DP - beta*dt

    if ( present(debug) ) call iDebug

    tmp0(:,:) = tmp(:,:)     ! copy

    do j = 1 , ny-1
       do i = 1 , nx-1
          tmp(i,j) = ( alpha_x * ( tmp0(i+1,j) + tmp0(i-1,j) )    &
                     + alpha_y * ( tmp0(i,j+1) + tmp0(i,j-1) )    &
                     + src(i,j)                                    &
                     ) * dt                                        &
                     + beta2 * tmp0(i,j)
       end do
    end do

  contains

    subroutine iDebug
      select case (debug)
      case ('beta2')
         print *, '[debug] beta2 = ', beta2
      case default
         print *, '[debug]     dt = ',       dt
         print *, '[debug] alpha_x = ', alpha_x
         print *, '[debug] alpha_y = ', alpha_y
         print *, '[debug]  beta2 = ',   beta2
      end select
    end subroutine iDebug
  end subroutine field__time_advance
end module field


program main
  use constants
  use ut
  use grid
  use field
```

```
  implicit none

  integer  :: i, j, nx, ny
  integer  :: loop, loop_max = 100000
  real(DP) :: xlen = 1.0_DP ! meter
  real(DP) :: ylen = 1.0_DP ! meter

  real(DP) :: dt_critical, dt, dx, dy

  print *, ' enter nx:'
  read(5,*) nx
  print *, ' enter ny:'
  read(5,*) ny
  call ut__assert(nx>0 .and. ny>0, 'nx/ny out of range.')

  call grid__initialize(xlen,ylen,nx,ny)
  dx = grid__d('x')
  dy = grid__d('y')
  call field__initialize
  call field__statistics('all')

  dt_critical = 0.5_DP / ( K_THERM * ( 1/(dx*dx) + 1/(dy*dy) ) )

  dt = dt_critical*0.8_DP

  do loop = 1 , loop_max
     call field__time_advance(dt)
     if ( mod(loop,100)==0 ) call field__statistics('max')
  end do
end program main
```