```fortran
program landau_damping

  ! program solves dg/dt + v * d/dx (g + phi)
  ! phi = int dv exp(-v**2) * g

  implicit none

  !-------------- input parameters ---------------------!

  ! Te/Ti
  real, parameter :: tau = 1.0

  ! xgrid parameters
  ! nx is total number of x grid points
  integer, parameter :: nx = 50
  real, parameter :: Lx = 1.0

  ! vgrid parameters
  ! nvp is number of positive parallel velocities sampled
  integer, parameter :: nvp = 500
  real, parameter :: Lv = 3.0

  ! simple (explicit, grid-based) time grid parameters
  integer, parameter :: nstep_simple = 1
  real, parameter :: dt_simple = 0.001
  integer, parameter :: nwrite_simple = 5

  ! advanced (implicit, spectral) time grid parameters
  integer, parameter :: nstep_adv = 500
  real, parameter :: dt_adv = 0.01
  integer, parameter :: nwrite_adv = 5

  ! k-space parameters
  ! k = 2*pi*kint
  integer, parameter :: kint = 1
  integer, parameter :: nk = 1

  ! Hermite (m-space) parameters
  ! number of Hermite polynomials
  integer, parameter :: nm = 100
  ! coefficient for artificial hyperviscosity of form -nu_m * m**4
  real, parameter :: nu_m = 0.00001

  !-----------------------------------------------------!
  ! grid spacing in x and v
  real :: dx, dv

  ! total number of parallel velocities sampled
  integer :: nv

  ! keeps track of time variable
  real :: time

  ! the constant pi
  real :: pi

  ! the imaginary unit
  complex :: zi = (0.,1.)

  ! wavenumber of mode in Fourier approach
  real :: k

  ! g(x,v)
  real, dimension (:,:), allocatable :: gxv
  ! phi(x)
  real, dimension (:), allocatable :: phi
  ! grid in x
  real, dimension (:), allocatable :: xgrid
  ! grid in v
  real, dimension (:), allocatable :: vgrid

  ! gk(m)
  complex, dimension (:), allocatable :: gkm
  real, dimension (:), allocatable :: gkmr, gkmi

  ! phi(k)
  complex, dimension (:), allocatable :: phik

  ! arrays needed to fill tridiagnoal advance matrix in spectral
approach
  real, dimension (:), allocatable :: aa, bb, cc

  integer :: phi_unit = 101, gxv_unit = 102, phi2_unit = 103, phik2_unit
= 104, gkm_unit = 105

  ! define pi for later use
  pi = 2.*acos(0.)
  k = 2.*pi*kint
```

```fortran
  call init_io

  ! allocates and populates xgrid and computes dx
  call get_xgrid
  ! allocates and populates vgrid and computes dv
  call get_vgrid

  ! allocates and initializes g(x,v)
  call init_gxv
  ! allocates and initializes phi(x)
  call init_phi
  ! explicit in time, grid-based in x and v solver
  call simple_solve

  ! allocates and initializes g(k,m)
  call init_gkm
  ! allocates and initializes phi(k)
  call init_phik
  ! implicit in time, spectral in x and v solver
  call advanced_solve

  call finish_io

  deallocate (xgrid, vgrid)
  deallocate (gxv, gkm, gkmr, gkmi)
  deallocate (phi, phik)
  deallocate (aa, bb, cc)

contains

  subroutine init_io

    implicit none

    open (phi_unit, file='landau_damping.phi', status='replace')
    open (phi2_unit, file='landau_damping.phi2', status='replace')
    open (phik2_unit, file='landau_damping.phik2', status='replace')
    open (gxv_unit, file='landau_damping.gxv', status='replace')
    open (gkm_unit, file='landau_damping.gkm', status='replace')

  end subroutine init_io

  subroutine finish_io
    implicit none

    close (phi_unit)
    close (phi2_unit)
    close (phik2_unit)
    close (gxv_unit)
    close (gkm_unit)

  end subroutine finish_io

  subroutine get_xgrid

    implicit none

    integer :: i

    if (.not.allocated(xgrid)) allocate (xgrid(nx))

    dx = Lx/(nx-1)
    do i = 1, nx
       xgrid(i) = (i-1)*dx
    end do

  end subroutine get_xgrid

  subroutine get_vgrid

    implicit none

    integer :: i

    if (.not.allocated(vgrid)) allocate (vgrid(-nvp:nvp))

    nv = 2*nvp+1
    dv = Lv/nvp
    do i = -nvp, nvp
       vgrid(i) = i*dv
    end do

  end subroutine get_vgrid

  subroutine init_gxv

    implicit none
    if (.not.allocated(gxv)) allocate (gxv(nx,-nvp:nvp))

    gxv = spread(cos(2.*pi*xgrid),2,nv)

  end subroutine init_gxv

  subroutine init_gkm

    implicit none

    if (.not.allocated(gkm)) then
       allocate (gkm(0:nm-1))
       allocate (gkmr(nm))
       allocate (gkmi(nm))
    end if

    gkm(0) = 0.5
    gkm(1:) = 0.0

  end subroutine init_gkm

  subroutine init_phi

    implicit none

    if (.not.allocated(phi)) allocate (phi(nx))

    call get_phi_grid

  end subroutine init_phi

  subroutine init_phik

    implicit none

    integer :: i

    if (.not.allocated(phik)) allocate (phik(nk))

    phik = gkm(0)
    if (nk == 1) phi = 2.*real(phik(1))*cos(k*xgrid)

  end subroutine init_phik

  subroutine get_phi_grid
```

```
  implicit none

  integer :: i

  do i = 1, nx
     call integrate_v (gxv(i,:),phi(i))
  end do
  phi = tau*phi

end subroutine get_phi_grid

subroutine integrate_v (f, tot)

  implicit none

  real, dimension (-nvp:), intent (in) :: f
  real, intent (out) :: tot

  integer :: i

  tot = 0.
  do i = -nvp, nvp
     tot = tot + dv*exp(-vgrid(i)**2)*f(i)/sqrt(pi)
  end do

end subroutine integrate_v

! simple_solve solves Landau damping problem
! with explicit time-stepping (forward Euler)
! upwind differencing in x
! simple grid-based integration in v
subroutine simple_solve

  implicit none

  integer :: istep

  time = 0.0
  call write_simple (0)

  do istep = 1, nstep_simple
     call update_gxv
     call get_phi_grid
     time = time + dt_simple
     call write_simple (istep)
  end do

end subroutine simple_solve

subroutine update_gxv

  implicit none

  integer :: i

  ! special case of v = 0 -- dg/dt = 0
  ! so no need to do anything

  ! v < 0
  do i = 1, nx-1
     gxv(i,:-1) = gxv(i,:-1) + dt_simple*vgrid(:-1)/dx &
          * (gxv(i,:-1) - gxv(i+1,:-1) &
          + phi(i) - phi(i+1))
  end do
  ! need a boundary condition on x
  ! use periodicity
  gxv(nx,:-1) = gxv(1,:-1)

  ! v > 0
  do i = nx, 2, -1
     gxv(i,1:) = gxv(i,1:) + dt_simple*vgrid(1:)/dx &
          * (gxv(i-1,1:) - gxv(i,1:) &
          + phi(i-1) - phi(i))
  end do
  ! need a boundary condition on x
  ! use periodicity
  gxv(1,1:) = gxv(nx,1:)

end subroutine update_gxv

subroutine write_simple (istep)

  implicit none

  integer, intent (in) :: istep

  integer :: i, j

  if (mod(istep,nwrite_simple)==0) then
```

```fortran
       do j = -nvp, nvp
          do i = 1, nx
             write (gxv_unit,*) time, xgrid(i), vgrid(j), gxv(i,j)
          end do
          write (gxv_unit,*)
       end do
       write (gxv_unit,*)
       write (gxv_unit,*)

       do i = 1, nx
          write (phi_unit,*) time, xgrid(i), phi(i)
       end do
       write (phi_unit,*)
    end if

    write (phi2_unit,*) time, dx*sum(phi(:nx-1)**2)

    write (*,*) 'time = ', time, '|phi|**2 = ', dx*sum(phi**2)

  end subroutine write_simple

  subroutine advanced_solve

    implicit none

    integer :: istep

    time = 0.0
    call write_advanced (0)

    call init_advance_matrix

    do istep = 1, nstep_adv
       call update_gkm
       time = time + dt_adv
       call write_advanced (istep)
    end do

  end subroutine advanced_solve

  subroutine init_advance_matrix

    implicit none

    integer :: i, sgn
    if (.not. allocated(aa)) then
       allocate (aa(nm)) ; aa = 0.
       allocate (bb(nm)) ; bb = 1.
       allocate (cc(nm)) ; cc = 0.
    end if

    do i = 3, nm-1
       sgn = (-1)**(i-1)
       aa(i) = sgn*0.5*k*dt_adv
       cc(i) = sgn*k*dt_adv*i
    end do

    cc(1) = k*dt_adv

    aa(2) = -0.5*k*dt_adv*(1.0 + tau)
    cc(2) = -2.*k*dt_adv

    aa(nm) = (-1)**(nm-1)*0.5*k*dt_adv

    do i  = 1, nm
       bb(i) = bb(i) + nu_m*(i-1)**4
    end do

  end subroutine init_advance_matrix

  subroutine update_gkm

    implicit none

    integer :: i

    do i = 0, nm-1
       if (mod(i,2)==0) then
          gkmr(i+1) = real(gkm(i))
          gkmi(i+1) = aimag(gkm(i))
       else
          gkmr(i+1) = -aimag(gkm(i))
          gkmi(i+1) = real(gkm(i))
       end if
    end do
    call tridag (aa, bb, cc, gkmr)
    call tridag (aa, bb, cc, gkmi)
    do i = 0, nm-1
       if (mod(i,2)==0) then
```

```
          gkm(i) = gkmr(i+1) + zi*gkmi(i+1)
       else
          gkm(i) = gkmi(i+1) - zi*gkmr(i+1)
       end if
    end do



  end subroutine update_gkm

  subroutine write_advanced (istep)

    implicit none

    integer, intent (in) :: istep

    integer :: i

    if (mod(istep,nwrite_adv)==0) then
       do i = 0, nm-1
          write (gkm_unit,*) time, i, real(gkm(i)), aimag(gkm(i))
       end do
       write (gkm_unit,*)
    end if

    write (phik2_unit,*) time, 2.*real(conjg(gkm(0))*gkm(0))*tau**2

    write (*,*) 'time = ', time, '|phik|**2 = ',
real(conjg(gkm(0))*gkm(0))*tau

  end subroutine write_advanced

  ! solves system Ax = b for x (which is returned as sol)
  ! inputs are aa, bb, and cc (the elements to the left, center, and
right
  ! of diagonal in tridiagonal matrix A)
  ! and sol=b as the rhs of the linear system Ax = b
  subroutine tridag (aa, bb, cc, sol)

    implicit none

    real, dimension (:), intent (in) :: aa, bb, cc
    real, dimension (:), intent (in out) :: sol

    integer :: ix, npts
```

```fortran
    real :: bet

    real, dimension (:), allocatable :: gam

    npts = size(aa)
    allocate (gam(npts))

    bet = bb(1)
    sol(1) = sol(1)/bet

    do ix = 2, npts
       gam(ix) = cc(ix-1)/bet
       bet = bb(ix) - aa(ix)*gam(ix)
       if (bet == 0.0) write (*,*) 'tridiagonal solve failed'
       sol(ix) = (sol(ix)-aa(ix)*sol(ix-1))/bet
    end do

    do ix = npts-1, 1, -1
       sol(ix) = sol(ix) - gam(ix+1)*sol(ix+1)
    end do

    deallocate (gam)

  end subroutine tridag

end program landau_damping
```