High Level Classes in UPIC Framework

The purpose of the High Level Classes is to encapsulate large pieces of code that the programmer does not wish to modify. They also provide a template for constructing new high level classes. In general, the data structure for the high level classes includes the helper classes, the data itself, and various additional constants.
S_field2d class

This high level class provides support for scalar fields that have properties. Examples of such fields include charge density or potentials. This class encapsulates the information necessary for the fields to be able to perform various operations such as solving Poisson’s equation.

The data structure for this class is:

```fortran
  type s_field2d
    integer :: psolver, state, gcells, iunit, iol   ! states
    real :: sf0                                         ! initial value
    integer, dimension(3) :: n2                         
    type (ufield2d) :: rs, ks, r2s            ! the helper objects 
    type (nfield2d) :: ns                              ! "
    real, dimension(:, :, :) , pointer :: sf         ! the data
    complex, dimension(:, :, :) , pointer :: sft        ! "
    type (fft2d), pointer :: ft                      ! for fft
    type (fpois2d), pointer :: pt          ! for poisson solver
    type (pgraf2d), pointer :: cmap      ! for plotting
  end type s_field2d
```

By default, the object can be FFTed. This means that it can be in one of two states, in real space, stored in sf, where the y co-ordinate is distributed, or in kspace, stored in sft, where the x co-ordinate is distributed. Helper objects rs, ks and ns describe the data. The data member "state" indicates which state is the current one. A charge density is such an object. The member ft points to an FFT table needed to perform an FFT.
The default constructor creates an object with this property. Using such objects is very simple. For example:

```fortran
  type (s_field2d) :: qe               ! charge density
  call new_s_field2d(qe,spect,ftflag=.true.) ! constructor
  call ffrk(qe)                     ! fft to k space
```

Another property scalar field objects can have is that they can be the solution of a Poisson equation. A potential field is such an object. Different kinds of spectral field solvers can be implemented, and the flag psolver indicates which one is to be used. Currently, only periodic and conducting boundaries are implemented. The member pt points to a Poisson table needed to solve a Poisson's equation.

```fortran
  type (s_field2d) :: pot             ! potential
  call new_s_field2d(pot,spect,ax,ay,affp) ! constructor
  call poisson(pot,qe,we)          ! solve poisson's equation
  call fftkr(pot)                          ! fft to real space
```

Additional special constructors can create objects which contain only real space data or no data at all. These are typically used by fields which are not calculated self consistently, such as an external charge density.

Some poisson solvers require that the grid be doubled in size, and the data members r2s and n2 are used in that case. These are calculated automatically by the constructors. Even objects with no data have valid helper objects calculated.
subroutine fftrk_s(this)
! this subroutine performs fft of a high level scalar 2d field object
! from real space to fourier space
! this = s_field2d scalar field object
  implicit none
  type (s_field2d) :: this
  integer :: isign = -1
  real, dimension(2*size(this%sf,1),this%n2(2),this%n2(3)) :: sf2
  character(len=9), save :: sname = ':fftrk_s:'
  if (monitor==2) call werrfl(class//sname//' started')

! check for errors
  if (monitor > 0) then
    if (.not.associated(this%sf)) then
      erstr = ' invalid s_field2d object'
      S_FIELD2D_ERR = 2; EXCEPTION = EXCEPTION + 1
      call ehandler(EDEFAULT,class//sname//erstr); return
    endif
    if (this%state /= XLOCAL) then
      write (erstr,*) 'invalid state = ', this%state
      S_FIELD2D_ERR = 3; EXCEPTION = EXCEPTION + 1
      call ehandler(EDEFAULT,class//sname//erstr); return
    endif
  endif
  ! add up guard cells
  if (this%gcells==0) call add_guard2d_s(this)

! transform function to fourier space
  if (this%psolver==PERIODIC_2D) then
    call fft(this%ft,this%rs,this%sf,this%ks,this%sft,isign)
  else if (this%psolver==DIRICHLET_2D) then
    call dblsin(this%rs,this%sf,this%r2s,sf2)
    call fft(this%ft,this%r2s,sf2,this%ks,this%sft,isign)
  else
    erstr = ' invalid s_field2d object'
    S_FIELD2D_ERR = 2; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
  this%state = YLOCAL
  if (monitor==2) call werrfl(class//sname//' complete')
end subroutine fftrk_s
Note the implementation of the high level class uses the middle layer to perform the work:

```fortran
subroutine pois_s(this,that,we)
! this subroutine calculates potential in fourier space
! this = output s_field2d scalar field object (potential)
! that = input s_field2d scalar field object (charge density)
implicit none
type (s_field2d) :: this, that
real :: we
! local data
character(len=8), save :: sname = ':pois_s:'
if (monitor==2) call werrfl(class//sname//' started')
! check for errors
if (monitor > 0) then
  if (.not.associated(this%sf)) then
    erstr = ' invalid s_field2d object'
    S_FIELD2D_ERR = 2; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
  if ((this%psolver < 1) .or. (this%psolver > 2)) then
    erstr = ' invalid s_field2d object'
    S_FIELD2D_ERR = 2; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
  if (that%state /= YLOCAL) then
    write (erstr,*) 'invalid state = ', that%state
    S_FIELD2D_ERR = 3; EXCEPTION = EXCEPTION + 1
    call ehandler(EDEFAULT,class//sname//erstr); return
  endif
endif
call pois(this%pt,that%sft,this%sft,we)
this%state = YLOCAL
if (monitor==2) call werrfl(class//sname//' complete')
end subroutine pois_s
```
Other properties can be added to scalar fields after the object has been created. For example, one can add the property that an object can write its data to a disk.

```fortran
    call add_writef(qe)  ! add property to object
    call writef(qe,fname,it)  ! write to file
```

This is a parallel write which will collect data from multiple processors onto a single node. The data member iunit and iol encapsulate the Fortran unit number and length of data.

To summarize, the data structure for this class encapsulates the following information:

- `psolver` = type of poisson solver
- `state` = (0,1) = current data layout is (XLOCAL,YLOCAL)
- `gcells` = (0,1) = (no,yes) guard cell processing complete
- `iunit` = fortran unit number for writing output
- `iol` = length of record for unformatted writes
- `sf0` = initial value for real space array
- `n2` = dimension of temporary doubled-size real space array
- `rs, ks, r2s` = helper objects for uniformly partitioned arrays
- `ns` = helper object for non-uniformly partitioned arrays
- `sf` = data array in real space
- `sft` = complex data array in fourier space
- `ft` = helper object for performing ffts
- `pt` = helper object for poisson solver
- `cmap` = helper object for graphics
Constructors for creating high level 2d scalar objects:

subroutine new_s_field2d(this,spect,ftflag)
subroutine new_s_field2d(this,spect,ax,ay,affp)
subroutine new_s_field2d_null(this,spect)

subroutine del_s_field2d(this)          ! destructor

The main idea here is that a user of this class does not need to know the details of what is inside the s_field2d type. The user only has to know about the constructors and the methods.

Add Properties to enable writing to disk and display:

subroutine add_writef(this)     ! write to disk
subroutine add_display(this,label,style,scale,clip,ncolors,ncontours,label2)

Accessor function to obtain area occupied by plasma. Used to normalize force to yield a plasma frequency of unity:

function get_area_s(this)
**Initialization functions** to assign values to sfield2d object:

- subroutine init_sfield(this,qi0) \(\text{! this = qi0}\)
- subroutine init_sfield2d_a(this,f,gcells) \(\text{! this = f}\)
- subroutine reset_sfield(this) \(\text{! this = qi0}\)

Main idea here is to make sure guard cells are handled properly.

**Guard cell functions** for adding and replicating data into guard cells:

- subroutine add_guards(this)
- subroutine copy_guards(this)

These functions call the middle layer to perform the work.

**General functions:**

- subroutine addto_sfield(this,that) \(\text{! this = this + that}\)
- subroutine fftrk(this) \(\text{! fft real to k space}\)
- subroutine fftkr(this) \(\text{! fft k space to real}\)
- subroutine poisson(this,that,we) \(\text{! poisson’s equation}\)
- subroutine writef(this,iunit,fname,it) \(\text{! write to disk}\)
- subroutine display(this,irc,scale,clip,ncolors, ncontours,label2) \(\text{! display on screen}\)
- subroutine printout(this) \(\text{! print out data members}\)
V_field2d class

This high level class provides support for vector fields that have properties. Examples of such fields include current density or electric fields. This class is very similar to the scalar field class in functionality.

The data structure for this class is:

```fortran
  type v_field2d
    integer :: psolver, state, gcells
    integer, dimension(3) :: n2
    type (ufield2d) :: rs, ks, r2s
    type (nfield2d) :: ns
    real, dimension(:,:,:,:), pointer :: vf
    complex, dimension(:,:,:,:), pointer :: vft
    type (fft2d), pointer :: ft
    type (fpois2d), pointer :: pt
  end type v_field2d
```

- `psolver` = type of poisson solver
- `state` = (0,1) = current data layout is (XLOCAL,YLOCAL)
- `gcells` = (0,1) = (no,yes) guard cell processing complete
- `n2` = dimension of temporary doubled-size real space array
- `rs, ks, r2s` = helper objects for uniformly partitioned arrays
- `ns` = helper object for non-uniformly partitioned arrays
- `vf` = data array in real space
- `vft` = complex data array in fourier space
- `ft` = helper object for performing ffts
- `pt` = helper object for poisson solver
Constructors for creating high level 2d vector objects:

subroutine new_v_field2d(this,spect,ndim,fftflag)
subroutine new_v_field2d(this,spect,ndim,ax,ay,affp)
subroutine new_v_field2d_null(this,spect)

subroutine del_v_field2d(this) ! destructor

Initialization functions to assign values to vfield2d object:

subroutine init_vfield(this,xj0,yj0,zj0) ! this = (xj0,yj0,zj0)
subroutine init_bfield(this,omx,omy,omz)
subroutine reset_vfield(this) ! this = (xj0,yj0,zj0)

Guard cell functions for adding and replicating data into guard cells:

subroutine add_guardv(this)
subroutine copy_guardv(this)

These functions call the middle layer to perform the work.

General functions:

subroutine ffrtk(this) ! fft real to k space
subroutine fftkr(this) ! fft k space to real
subroutine poisson(this,that,we) ! poisson’s equation
subroutine vpot(this,that,we) ! vector potential
subroutine magfield(this,that,we) ! magnetic field
subroutine printout(this) ! print out data members
Species2d class

This high level class contains particle co-ordinates as well as a particle helper object which describes properties of a particle. There are three main methods in the class, depositing charge and current and pushing particles. The helper object determines what algorithms are used in each case.

The data structure for this class is:

```fortran
  type species2d
    integer :: npmax
    type (part2d) :: pd
    real, dimension(:,:,m), pointer :: part
    type (pgraf2d), pointer :: cmap
  end type species2d
```

which encapsulates the following information:

- npmax = maximum number of particles in each partition
- pd = part2d descriptor of particle data
- part(:,:,m) = particle coordinates in partition m
- cmap = helper object for graphics

Here is a typical use of a push routine:

```fortran
  call push(electrons,efield,bfield,it ime,kinetic_energy)
```
Constructors for 2d high level species object:

subroutine new_species2d(this,pd,bfield,backg,beam)
subroutine new_species2d(this,nvp,mshare,backg,beam)

subroutine del_species2d(this)  ! destructor

Add Properties for displaying phase space:

subroutine add_display(this,nx,ny,label,scale)
! this subroutine adds a descriptor for 2d parallel graphics

Accessor function for obtaining information about total number of particles in simulation. Used to normalize force to yield a plasma frequency of unity:

function get_num(this)

General functions for depositing charge and current, pushing particles, displaying phase space:

subroutine qdeposit(this,qfield)
subroutine jdeposit(this,qfield)
subroutine push(this,efield,bfield,itime,ek)
subroutine grasp(this,itime,iyp,ixp,ierr,scale)
subroutine printout(this)
Here is the implementation of the high level push:

    subroutine ihpgbpush2(this,efield,bfield,itime,ek)
    ! this subroutine updates particle co-ordinates, occasionally sorts them
    ! this = species2d particle object
    ! efield = v_field2d vector field object for electric field
    ! bfield = v_field2d vector field object for magnetic field
    ! itime = current time step
    ! ek = kinetic energy / mass
    implicit none
    type (species2d) :: this
    type (v_field2d) :: efield, bfield
    integer, intent(in) :: itime
    real :: ek
    ! local data
    character(len=12), save :: sname = ':ihpgbpush2:'
    if (monitor==2) call werrfl(class//sname//' started')

    ! push particles
    call push(this%pd,this%part,efield%ns,efield%vf,bfield%vf,ek)
    ! move particles into appropriate spatial regions
    call pmove(this%pd,this%part,efield%ns)

    ! sort particles
    if (this%pd%sortime > 0) then
      if (mod(itime,this%pd%sortime)==0) then
        call sortp(this%pd,this%part,efield%ns)
      endif
    endif
    if (monitor==2) call werrfl(class//sname//' complete')
end subroutine ihpgbpush2
Let us examine what a simplified main code looks like with high level objects. First declare the variables:

```fortran
program main

  integer :: mshare = 0
  integer :: np, nx, ny, nxh, nloop, nvp, ipbc
  integer :: ou, idproc, id0, itime, it, ierr
  real :: zero = 0.0, qbme, affp, qi0, we, wke
  double precision :: dtime
  type (spect2d) :: space, lspace
  type (s_field2d) :: q, pot
  type (v_field2d) :: fxy, bxy, cu, axy
  type (fdist2d) :: ebackg, ebeam
  type (part2d) :: epart
  type (species2d) :: electrons
  real, dimension(3) :: wtot
  real, dimension(2) :: time
  double precision, dimension(3) :: msg
  character(len=12) :: label, potname

```

Then initialize all the variables (create a plasma):

call init_parallel(idproc,id0,nvp)
call readnml2('pinput2')
call init_pgraf(nplot,idpal)
nx = 2**indx; ny = 2**indy; nxh = nx/2
call fcomp(nvp,nxh,ny)
nloop = tend / dt + .0001

call new_spect2d(space,indx,indy,nvp,mshare,psolve,inorder)
call pwtimer(time,dtime,-1)
itime = 0

call new_s_field2d(q,space)
call new_fdist2d(ebackg,vtx,vty,zero,zero,npx,npy)
call new_fdist2d(ebeam,vtdx,vtdy,vdx,vyd,npxb,npyb)
call new_v_field2d_null(bxy,space)

ipbc = get_ipbc(space)
qbme = qme
call new_part2d(epart,ipbc,emf,qme,qbme,dt,sortime,popt,dopt)
call new_species2d(electrons,epart,bxy,ebackg,ebeam)
affp = get_area(q)/real(get_num(electrons))
call new_v_field2d(fxy,space,2,ax,ay,affp)
qi0 = -qme*real(get_num(electrons))/get_area(q)
call init_sfield(q,qi0)
call qdeposit(electrons,q)
call add_guards(q)
call pwtimer(time,dtime)
Here is what the main loop looks like:

! *** start main iteration loop ***
!
500 if (nloop <= itime) go to 2000
    write (label,991) itime
    call fftrk(q)
    call poisson(fxy,q,we)
    call fftkr(fxy)
    wke = 0.
    call push(electrons,fxy,bxy,itime,wke)
    call reset_sfield(q)
    call qdeposit(electrons,q)
    call add_guards(q)
    itime = itime + 1
    go to 500
2000 continue
!
*** end main iteration loop ***
!

When finished, deallocate the variables (delete the plasma):

    call pwtimer(time,dtime)
    call end_pgraf()
    call end_parallel()
    call del_species2d(electrons)
    call del_part2d(epart)
    call del_v_field2d(bxy)
    call del_v_field2d(fxy)
    call del_s_field2d(q)
    call pr_alloc()
Plasma2d_class

This class encapsulates the entire code into one super class. One merely puts all the initial variables into a type:

type plasma2d
  integer :: ou, idproc, id0, nvp, mshare, nloop
  type (spect2d) :: space, lspace
  type (s_field2d) :: q, pot
  type (v_field2d) :: fxy, bxy
  type (fdist2d) :: ebackg, ebeam
  type (part2d) :: epart
  type (species2d) :: electrons
end type plasma2d

Encapsulates the initialization into a single constructor:

  subroutine new_plasma2d(this)                   ! create a plasma
  subroutine del_plasma2d(this)                   ! delete a plasma

Create a method to advance the plasma one time step:

  subroutine update_plasma2d(this,itime,done)     ! run one time step

The entire code then looks like:

  use plasma2d_class
  integer :: itime = 0, done = 0
  type (plasma2d) :: plasma

  call new_plasma2d(plasma)
  do while (done==0)
    call update_plasma(plasma,itime,done)
  enddo

  call del_plasma2d(plasma)
What’s the advantage? By encapsulating the entire PIC code, one can easily **incorporate** a PIC code into another code, for example, a fluid code.

A fluid code can create a PIC plasma by merely:

```fortran
use plasma2d_class
  type (plasma2d) :: plasma
  call new_plasma2d(plasma)  ! we now have a PIC plasma
```

It can update it by **sending a message** to the PIC code (e.g., calling update_plasma). One can write new **accessor methods** such as `get_the_potential` or `get_the_velocity_distribution` to enable other codes to get information from the PIC code, such as:

```fortran
  potential = get_potential(plasma)
```

All without knowing anything about what is inside. Other advantages: One can create several plasmas with different properties, compare them.
Suppose one wanted to call the PIC code from another language, e.g., from C or Java. We can’t use things like:

```
use plasma2d_class
type (plasma2d) :: plasma
```

because C doesn’t know about Fortran derived types.

But we could hide the entire plasma type inside the PIC code and only give C an integer into an array of plasmas, which are all hidden inside the PIC code. For example in the plasma2d class we include:

```fortran

! type plasma2d_list
!   type (plasma2d), pointer :: p
! end type plasma2d_list
!
! hidden internal array of pointers to plasma objects
!   integer, parameter :: mplasma = 4
!   integer, save :: nplasma = 0
!   type (plasma2d_list), dimension(mplasma), save :: plasmas
!
```

We can then let C ask for a plasma, and the PIC code will allocate one from its internal array of plasma pointers and return to C a Fortran index into that array.
The C code can then run the Fortran90 PIC code:

```c
int itime = 0, done = 0;
int plasma;

NEW_2DPLASMA(&plasma);
while (done==0) {
    UPDATE_2DPLASMA(&plasma,&itime,&done);
}
DEL_2DPLASMA(&plasma);
```

Why would you want to do this? One reason is to provide a user interface to the Fortran code. For example, it is generally easier to program event handling in C. At the end of each iteration of the while loop, the C code could check whether the user had selected a menu item, and send a message to the Fortran code to do something, such as turn on phase space plots.

Since Java can call C code, we can call this from Java, which could give our Fortran code a user interface on the web.
Here is the Fortran function which will allocate a plasma from its internal array of plasma pointers and return to C a Fortran index into that array:

```fortran
subroutine NEW_2DPLASMA(plasma_id)
! external interface for 2d plasma constructor
use perrors_class
use plasma2d_class
implicit none
integer :: plasma_id
! local data
integer :: i
character(len=14), save :: sname = ':NEW_2DPLASMA:'
plasma_id = 0
! nullify pointers on first
if (nplasma==0) then
  do i = 1, mplasma
    nullify(plasmas(i)%p)
  enddo
endif
! find unallocated plasma
  do i = 1, mplasma
    if (.not.associated(plasmas(i)%p)) then
      plasma_id = i
      exit
    endif
  enddo
! check for too many requests
if (plasma_id==0) then
  write (erstr,*) 'too many plasmas requested, max = ', mplasma
  PLASMA2D_ERR = 3; EXCEPTION = EXCEPTION + 1
  call ehandler(EDEFAULT,'plasma2d'//sname//erstr); return
endif
! create new plasma
allocate(plasmas(plasma_id)%p)
call new_plasma2d(plasmas(plasma_id)%p)
nplasma = nplasma + 1
end subroutine NEW_2DPLASMA
```
I hope these UPIC Framework examples illustrate how the techniques of object-oriented programming can be useful for:

1. Improving program safety
2. Constructing more complex scientific programs
3. Writing codes involving multiple authors
4. Coupling multiple independent codes together