

A Brief Tutorial on GSI Infrastructures & Advanced Features

Ricardo Todling

Global Modeling and Assimilation Office

GSI Tutorial, DTC/NCAR, 21-23 August, 2012

This presentation is a brief guide to some of the basic infrastructure being added to GSI, namely:

- Introducing MetGuess_Bundle
- Interfacing to user-specific applications
- Remarks on adding new observing instruments

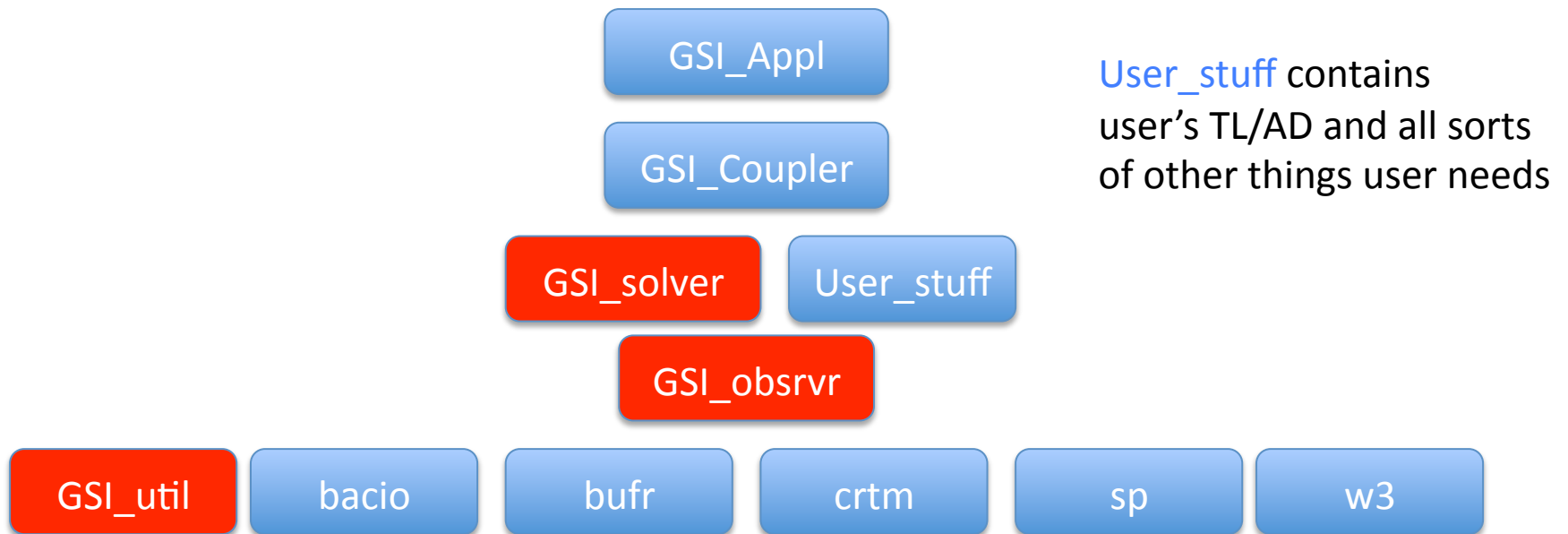
OUTLINE

- Introducing MetGuess_Bundle
- Interfacing to user-specific applications
 - General concept
 - Illustration 1: timing routines
 - Illustration 2: aerosols
 - Illustration 3: 4D-Var
 - Illustration 4: Hybrid Ensemble
- Adding new instruments (obs. operator)

GSI Infrastructure:

Split GSI into sub-libraries

- Schematic view of GSI & supporting libraries at GMAO



- Issues:
 - At present, GSI_solver and GSI_obsrvr cannot be separated
 - At present, GSI_util cannot sit parallel to supporting libs due to its reliance on some of those

Chem/MetGuess_Bundle

- Presently, ChemGuess_Bundle allows flexible input of Chem-related fields (tracers and aerosols) to GSI.
- A desirable similar flexibility to handle all of the other (meteorological) guess fields, motivates introduction of MetGuess_Bundle.
- Just as with ChemGuess, MetGuess_Bundle is controlled by a table named met_guess added to the anavinfo resource file. Examples are given below:

GMAO	met_guess::					met_guess::					Global NCEP
	lvar	level	crtm_use	desc	orig_name	lvar	level	crtm_use	desc	orig_name	
	cw	72	10	cloud_condensate	qctot	cw	64	10	cloud_condensate	qctot	
	ql	72	-1	Water	qltot	#ql	64	-1	Water	qltot	
	qi	72	-1	Ice	qitot	#qi	64	-1	Ice	qitot	
	#qr	72	10	Rain	qrtot	#qr	64	10	Rain	qrtot	
	#qs	72	10	Snow	qstot	#qs	64	10	Snow	qstot	
	#qg	72	10	Graupel	qg	#qg	64	10	Graupel	qg	
	#qh	72	10	Hail	qh	#qh	64	10	Hail	qh	
	#cf	72	2	cloud_frac4rad(fcld)	cloud	#cf	64	2	cloud_frac4rad(fcld)	cloud	
	::					::					

MetGuess_Bundle: Methods

- As with ChemGuess, MetGuess **does not handle parallelization**; e.g., fields are on subdomains. This means filling up this bundle must be done by the user after reading the guess and distributing it onto subdomains.
- Presently, the available **methods** in MetGuess are:

```
public :: gsi_metguess_create_grids  
public :: gsi_metguess_destroy_grids  
public :: gsi_metguess_init  
public :: gsi_metguess_get  
public :: gsi_metguess_final
```

- The trickiest of the Methods is the GET. It's easy to use but has multiple capability. Examples of the GET function are given in the **ProTex documentation** available in the source code. Here a couple of simple examples follow:

MetGuess_Bundle: Methods

- Examples of using the GET Method:
 - Say a routine wants to know whether or not the variable **cw** is in **MetGuess_Bundle**. This can be done simply with the call
call gsi_metguess_get ('var::cw', ivar, ier)
if **ivar** is greater than zero, the variable is present in the bundle.
 - Say a routine wants to get the number of all 3d cloud fields in the **MetGuess_Bundle**, this can be done by use the tag **clouds::3d**, as in:
call gsi_metguess_get ('clouds::3d', n, ier)
notice this uses the same interface as in the example above, but returns information about something else.
 - Say a routine wants the name of all 3d cloud-fields
call gsi_metguess_get ('clouds::3d', cld3dnames, ier)
now the returned variable **cld3dnames** is a character array with the names of all 3d-cloud-guess. Notice it is important to inquire before hand about the number of 3d-cloud fields available and to properly allocate space for the character arrays **cld3dnames**, and only then make the call above.
 - Other functionalities and inquire modes are available.

Chem/MetGuess_Bundle

Remarks and Work in Progress

- As ChemGuess_Bundle, MetGuess_Bundle is treated as a common block.
 - This means you **cannot**, for the time being, **instantiate** it.
- As ChemGuess_Bundle, MetGuess_Bundle is an **almost opaque** object.
 - This means only methods are available to the outside world, and the bundle itself (common block, for now)
- In the present (upcoming, June 2011) version of the GSI NCEP trunk, **only clouds** are being handled by MetGuess. That is, winds, temperature, specific humidity, and all other meteorological fields are still handled in guess_grids (as the ges_X variables). This will change in the near future.

Interfacing to user-specific applications

- Simplest possible paradigm: FORTRAN-77-like
 - No “use” statements allowed at interface level
 - No “ifdef’s” (preferred)
 - GSI Convention: package name stub_XXX.F90
 - User Convention: (suggested) name cplr_XXX.F90
- Current available GSI interfaces:
 - timermod.F90
 - set_crtm_aerosolmod.F90
 - gsi_4dcouplermod.F90
 - gsi_ensmod.F90
 - gsi_nstmod.F90
- Each of the interfaces is associated with a stub, respectively:
 - stub_timermod.F90
 - stub_set_crtm_aerosol.F90
 - stub_pertmod.F90
 - stub_ensmod.F90
 - stub_nstmod.F90

Interfacing to user-specific applications

Illustration I: **timermod**

- More often than not, timing routines are user and machine specific. **timermod** allows for the possibility of a user to supply its own timing mechanism.

Actual interface: **timermod.F90**

```

public timer_ini
public timer_fnl
public timer_pri

interface timer_ini
  subroutine timer_init_ (str)
    implicit none
    character(len=*),intent(in ) ::
  end subroutine timer_init_
end interface

interface timer_fnl
  subroutine timer_final_ (str)
    implicit none
    character(len=*),intent(in ) :: str
  end subroutine timer_final_
end interface

interface timer_pri
  subroutine timer_pri_ (lu)
    use kinds, only : i_kind
    implicit none
    integer(i_kind),intent(in ) :: lu
  end subroutine timer_pri_
end interface

```

This is a
module;
this what
routines in
GSI call

Stub routines: **stub_timermod.F90**

```

subroutine timer_init_ (str)
!$$$ subprogram documentation block
!
! subprogram: timer_init_ initialize procedure
!
! prgmmr: todling org: gmao
!
! abstract: initializes timer
!
! program history log:
! 2007-10-01 todling
!
! input argument list:
! str - string designation for process to be t
!
! output argument list:
!
! attributes:
! language: f90
! machine:
!
!$$$ end documentation block

implicit none
character(len=*),intent(in ) :: str
end subroutine timer_init_

```

This is a
NOT
module;
ONLY
called by
timermod

Similar for other two routines, i.e., they are empty subroutines that do nothing

Interfacing to user-specific applications

Illustration I: **timermod**

- If a user wants to specify its own timings, it should provide a Coupler for the timing routines, as in **cplr_timermod** below:

User-specific routines: **cplr_timermod.F90**

```
subroutine timer_init_ (str)
  use m_zeit, only: zeit_ci ! A GMAO module for timing
  implicit none
  character(len=*),intent(in ) :: str
  call zeit_ci(str) ! start GMAO timing for str
end subroutine timer_init_

subroutine timer_final_ (str)
  use m_zeit, only: zeit_co ! A GMAO module for timing
  implicit none
  character(len=*),intent(in ) :: str
  call zeit_co(str) ! stop GMAO timing for str
end subroutine timer_final_

subroutine timer_pri_ (lu)
  use kinds, only : i_kind
  use mpimod, only : mype
  use m_zeit, only : zeit_flush ! A GMAO module for timing
  implicit none
  integer(i_kind),intent(in ) :: lu
  if(mype==0) call zeit_flush(lu,subname_at_end=.true.)
end subroutine timer_pri_
:
```

These are user
functions GSI
knows nothing
of

Prologues stripped off for display only. 10

Interfacing to user-specific applications

Illustration I: **timermod**

- The implications of adding user-specific functions/routines to GSI are the following:
 - The Make procedure in the **GSI directory** can no-longer create the GSI executable.
 - The Make procedure in the GSI directory **must** instead **create a GSI library**.
 - The corresponding dummy stub must be removed from the GSI library before the executable is created. This is easily accomplished by the flags of the archiving command. For example, in Linux, to remove the **stub_timermod.o** object file that would be in the GSI library (called it **libgsi.a** for the time being), one can simply add the following line to the Makefile that creates the executable:
 - **ar -d libgsi.a stub_timermod.o**
 - The Make procedure creating the executable can then load the GSI library, together with the user-library containing the Coupler, that is, in the example above **cplr_timermod.o**, and whatever else is needed, plus the main program from GSI (**gsimain.F90**).
 - This means the **gsimain.F90** should be placed outside of GSI. For the time being, the GSI directory could still keep a copy of this program, but only for reference.

Interfacing to user-specific applications

Illustration II: `set_crtm_aerosolmod`

- Aerosols can be brought into GSI via the ChemBundle. For example, to bring the 15 GOCART aerosols GMAO sets the `chem_guess` table as

```
chem_guess::
#   GOCART Aerosols
#   ----- Dust -----
du001  72  1  10  dust      DU001
du002  72  1  10  dust      DU002
du003  72  1  10  dust      DU003
du004  72  1  10  dust      DU004
du005  72  1  10  dust      DU005
#   ----- Sea-salt -----
ss001  72  1  10  ssam      SS001
ss002  72  1  10  sscm1     SS002
ss003  72  1  10  sscm2     SS003
ss004  72  1  10  sscm3     SS004
ss005  72  1  10  sea_salt  SS005
#   ----- Sulfates -----
so4    72  1  10  sulfate   SO4
#   ----- Carbonaceous (main) -----
bcphobic 72  1  10  dry_black_carbon BCPHOBIC
bcphilic  72  1  10  wet_black_carbon BCPHILIC
ocphobic  72  1  10  dry_organic_carbon OCPHOBIC
ocphilic  72  1  10  wet_organic_carbon OCPHILIC
::
```

These are
the
internal
GSI names

In ChemBundle
a value of 10
means aerosol.

These are
the names
of the
fields in
file

- The settings above allow aerosols to be passed to CRTM, but the GSI Jacobians do not take into account the sensitivity of fields to the aerosols – only the radiance feel the aerosols, but not the conventional fields. It's very simple to have the Jacobian augmented to take such sensitivities into account.

Interfacing to user-specific applications

Illustration II: `set_crtm_aerosolmod`

- When having aerosols passed to CRTM one thing necessary is specification of the particle sizes. This is done via a model-specific Mie calculation that requires the environment relative humidity and aerosol type. This is where the **aerosol interface** comes into play.

Actual interface: `set_CRTM_aerosolmod.F90`

```
module set_crtm_aerosolmod
implicit none

private
public Set_CRTM_Aerosol

interface Set_CRTM_Aerosol
subroutine Set_CRTM_Aerosol_ ( km, na, aero_name, aero_conc, rh, aerosol)

use kinds, only: i_kind,r_kind
use mpimod, only: mype
use CRTM_Aerosol_Define, only: CRTM_Aerosol_type

implicit none

integer(i_kind) , intent(in)      :: km                ! number of levels
integer(i_kind) , intent(in)      :: na                ! number of aerosols
character(len=*) , intent(in)     :: aero_name(na)      ! [na]   GOCART aerosol names: du0001, etc.
real(r_kind),    intent(in)       :: aero_conc(km,na)   ! [km,na] aerosol concentration (Kg/m2)
real(r_kind),    intent(in)       :: rh(km)             ! [km]   relative humidity [0,1]

type(CRTM_Aerosol_type), intent(inout) :: aerosol(na)! [na]   CRTM Aerosol object

end subroutine Set_CRTM_Aerosol_
end interface

end module set_crtm_aerosolmod
```

This provides
a general
interface

Stub routines: `stub_set_crtm_aerosol.F90`

```
subroutine Set_CRTM_Aerosol ( km, na, aero_name, aero_conc, rh, aerosol)

! USES:

use kinds, only: i_kind,r_kind
use mpimod, only: mype
use CRTM_Aerosol_Define, only: CRTM_Aerosol_type

implicit none

! !ARGUMENTS:

integer(i_kind) , intent(in)      :: km                ! number of levels
integer(i_kind) , intent(in)      :: na                ! number of aerosols
character(len=*) , intent(in)     :: aero_name(na)      ! [na]   GOCART aerosol names: du0001, etc.
real(r_kind),    intent(in)       :: aero_conc(km,na)   ! [km,na] aerosol concentration (Kg/m2)
real(r_kind),    intent(in)       :: rh(km)             ! [km]   relative humidity [0,1]

type(CRTM_Aerosol_type), intent(inout) :: aerosol(na)! [na]   CRTM Aerosol object

if(mytype==0) then
  print*, 'Stub Set_CRTM_Aerosol: Call to CRTM_Aerosol object'
  print*, 'Stub Set_CRTM_Aerosol: Call to CRTM_Aerosol object'
endif

end subroutine Set_CRTM_Aerosol
```

This
doesn't do
anything.

Prologues stripped off for display only.

Interfacing to user-specific applications

Illustration II: `set_crtm_aerosolmod`

- A user wanting to exercise the aerosol capability should provide it's own coupler. In the case of GMAO, the coupler looks something like this:

Actual GMAO coupler: `cplr_set_CRTM_aerosolmod.F90`

```

subroutine Set_CRTM_Aerosol ( km, na, aero_name, aero_conc, rh

! USES:

use kinds, only: i_kind,r_kind
use CRTM_Aerosol_Define, only: CRTM_Aerosol_type
use crt_m_aerosol, only: SetAerosol

implicit none

! !ARGUMENTS:

integer(i_kind) , intent(in)      :: km           ! numbe
integer(i_kind) , intent(in)      :: na           ! numbe
character(len=*) , intent(in)     :: aero_name(na) ! [na]
real(r_kind),    intent(in)       :: aero_conc(km,na) ! [km,r
real(r_kind),    intent(in)       :: rh(km)        ! [km]

type(CRTM_Aerosol_type), intent(inout) :: aerosol(na)! [na]

call setAerosol (aero_name, aero_conc, rh, aerosol)

end subroutine Set_CRTM_Aerosol

```

This is
where the
work starts
to happen

f77-like Coupler
calls user-
specific f90
routine

```

module crt_m_aerosol

! !USES:

use kinds, only: i_kind,r_kind
use CRTM_Aerosol_Define, only: CRTM_Aerosol_type
use CRTM_Aerosol_Define, only: DUST_AEROSOL, SEASALT_SSAM_AEROSOL, &
SEASALT_SSCM1_AEROSOL, SEASALT_SSCM2_AEROSOL, &
SEASALT_SSCM3_AEROSOL, SEASALT_SSCM3_AEROSOL, &
BLACK_CARBON_AEROSOL, ORGANIC_CARBON_AEROSOL, &
SULFATE_AEROSOL

use Chem_RegistryMod, only: Chem_Registry, Chem_RegistryCreate, Chem_RegistryDestroy
use Chem_MieMod,       only: Chem_Mie, Chem_MieCreate, Chem_MieDestroy, &
Chem_MieQueryIdx, Chem_MieQuery

use m_chars,           only: lowercase
use m_die,             only: die

implicit none

! !PUBLIC METHODS:

public setAerosol

! !PUBLIC DATA MEMBERS:
! !-----
type(Chem_Registry), pointer :: aerReg => null() ! Aerosol Registry
type(Chem_Mie),      pointer :: Mie => null()   ! Mie tables

! !REVISION HISTORY:
!
! 23feb2011 da Silva Initial version.
!
!EOP

```

Prologues stripped off for display only.

Interfacing to user-specific applications

Illustration II: **set_crtm_aerosolmod**

- As with timermod when we wanted **to replace the stub** with the real thing, we need to remove the stub from the GSI library and add our own (user-specific) library containing, in this case, the Coupler with the aerosol-specific calculations.
- This must **follow** the same mechanism through the **Makefiles as discussed** in Illustration I.

Interfacing to user-specific applications

Illustration III: **gsi_4dcouplermod**

- This provides the coupling mechanism to user-specific TL and AD models (will eventually be renamed **gsi_pertmod.F90**)
- The companion stub file is **stub_pertmod.F90**, that, as with other stubs, must be removed from the GSI library to allow the user to specify it's own coupler.
- This interface is more complex than those of previous illustrations. Only a sketchy illustration follows.

Methods in **gsi_4dcouplermod.F90**

Actual interface:

```
interface GSI_4dCoupler_init_traj
  subroutine pertmod_initialize_ (idmodel,rc)
  use kinds, only: i_kind
  implicit none
  logical,optional,intent(in):: idmodel
  integer(i_kind),optional,intent(out):: rc
end subroutine pertmod_initialize_
end interface
```

Trajectory
initialization

Run
ADM

```
interface GSI_4dCoupler_model_ad
  subroutine pertmod_ADrun_ (xini,xobs,iymd,ihms,ndt,rc)
  use kinds, only: i_kind
  use gsi_bundlemod, only: gsi_bundle
  implicit none
  type(gsi_bundle),intent(inout):: xini ! inout: adjoint increment perturbation
  type(gsi_bundle), pointer:: xobs ! input: adjoint perturbation state
  integer(i_kind),intent(in):: iymd ! starting date (YYYYMMDD) of the adjoint perturbation state
  integer(i_kind),intent(in):: ihms ! starting time (HHMMSS) of the adjoint perturbation state
  integer(i_kind),intent(in):: ndt ! Number of time steps to integrate TLM for
  integer(i_kind),optional,intent(out):: rc ! return status code
end subroutine pertmod_ADrun_
end interface
```

```
!
! !PUBLIC MEMBER FUNCTIONS:
!
public GSI_4dCoupler_parallel_init
public GSI_4dCoupler_setServices
public GSI_4dCoupler_init_traj
public GSI_4dCoupler_init_model_tl
public GSI_4dCoupler_model_tl
public GSI_4dCoupler_final_model_tl
public GSI_4dCoupler_init_model_ad
public GSI_4dCoupler_model_ad
public GSI_4dCoupler_final_model_ad
public GSI_4dCoupler_grtests
public GSI_4dCoupler_getpert
public GSI_4dCoupler_putpert
public GSI_4dCoupler_final_traj
```


Interfacing to user-specific applications

Illustration III: **gsi_4dcouplerm**od

- Both **GMAO** and **NCEP** have interfaced their TLM/ADM to GSI. The former has interfaced two different models, the most recent one being fully ESMF-capable; the latter has interfaced a perturbation model based on integrating the tendencies originally available in GSI.
- As illustration we show some of the NCEP perturbation model interface. This is composed mainly of two components:
 - **cplr_pertmod**: An f77-like coupler providing a replacement of **stub_pertmod**
 - **ncep_permod**: A f90 module providing the entry point to the perturbation model, and its TL and AD counterparts.
 - For now, a specific feature of the perturbation model implementation is that the observer must “run the non-linear model” (that is the perturbation model). Though quite unusual, the interface is general to easily accommodate this case.

Interfacing to user-specific applications

Illustration III: **gsi_4dcouplermod**

Actual interface: **cplr_pertmod.F90**

```
subroutine pertmod_initialize_(idmodel,rc)
```

Typically, this initializes the trajectory

```
use kinds, only: i_kind
use mpimod, only: mype
use nonlinmod, only: ncep_model_nl_init
use nonlinmod, only: ncep_model_nl
use mpeu_util, only: tell,perr,die
use obsmod, only: lobserver
implicit none
```

Relies on nonlinmod.F90, a module driving the pert model

```
logical,optional,intent(in):: idmodel
integer(i_kind),optional,intent(out):: rc    ! return status code
```

```
!~~~~~
character(len=*),parameter :: myname_=MYNAME//'::pertmod_initialize_'
logical:: idmodel_
integer(i_kind):: ier
```

```
if(present(rc)) rc=0
idmodel_=.true.
if(present(idmodel)) idmodel_=idmodel
call ncep_model_nl_init(ier)
  if(ier/=0) then
    call perr(myname_,'pertmod_initialize()', rc =',ier)
    if(.not.present(rc)) call die(myname_)
    rc=ier
    return
  endif
if(idmodel_) return
```

```
! For now, the observer runs the non-linear trajectory model
if(.not.lobserver) return ! _rt must be another param to control NL call
call ncep_model_nl
```

```
end subroutine pertmod_initialize_
```

In this case, it actually runs the perturbation to generate the trajectory – in the conventional case, the trajectory would simply be read-in

Prologues stripped off for display only.

Interfacing to user-specific applications

Illustration III: **gsi_4dcouplermod**

Actual interface: **cplr_pertmod.F90**

```
subroutine pertmod_TLrun_(p_xini,xobs,iymd,ihms,ntstep,rc)
```

Typically, this runs the TLM

```
use kinds, only: i_kind
use gsi_bundlemod, only: gsi_bundle
use ncep_pertmod, only: ncep_4dmodel_tl
use mpeu_util, only: tell,perr,die
implicit none
```



Relies on ncep_pertmod.F90, a module driving the TL/AD pert model

```
type(gsi_bundle), pointer:: p_xini      ! input: increment perturbation prop
type(gsi_bundle),intent(inout):: xobs    ! inout: TL perturbation state
integer(i_kind),intent(in):: iymd        ! starting date (YYYYMMDD) of the perturbation
integer(i_kind),intent(in):: ihms        ! starting time (HHMMSS) of the perturbation
integer(i_kind),intent(in):: ntstep      ! Number of time steps to integrate TLM for
integer(i_kind),optional,intent(out):: rc ! return status code
```

```
!! t := (nymdi,nhmsi); n:=ntstep; xi:=xini; yo:=xobs
!! e(t) = A(t)*xi(t)
!! z(t+n) = M(t+n,t)*[z(t)+e(t)]
!! yo(t+n) = G(t+n)*z(t)
```

```
!~~~~~
character(len=*),parameter :: myname_='MYNAME//':='pertmod_TLrun_'
integer(i_kind):: ier
```

```
if(present(rc)) rc=0
call ncep_4dmodel_tl(p_xini,xobs,iymd,ihms,ntstep,ier)
if(ier/=0) then
  call perr(myname_, 'pertmod_TLrun(), rc =',ier)
  if(.not.present(rc)) call die(myname_)
  rc=ier
  return
endif
```

```
end subroutine pertmod_TLrun_
```

Procedure from ncep_pertmod.F90 that actually integrates TLM

NOTE: ncep_pertmod.F90, and nonlinmod.F90 do not live inside GSI – they are part of the so-called NCEP_Coupler library. This also includes various other codes specific to the perturbation model.

Prologues stripped off for display only.

Interfacing to user-specific applications

Illustration III: **gsi_4dpertmod**

Actual interface: **cplr_pertmod.F90**

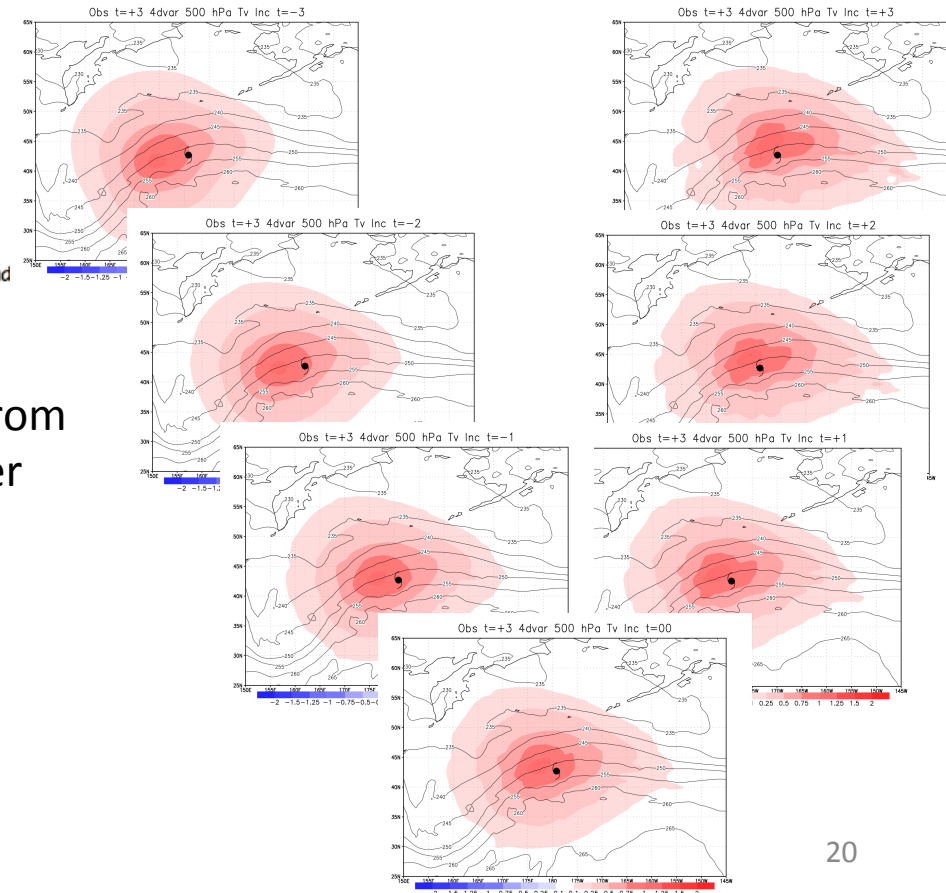
The handy put method: **GSI_4dCoupler_putpert** allows for writing of the Increment as it evolves within the assimilation window, see routine **view_st**

```
! write out perturbation vector|
mydate = ibdate
do ii=1,nobs_bins
  nynd = 10000*mydate(1)+mydate(2)*100+mydate(3)
  nhms = 10000*mydate(4)
  ! iwrtinc ...

  if(mytype==0) then
    write(6,'(2a,i8.8,2x,i6.6)')trim(myname_),': start writing state on ', nynd
  endif
  call gsi_4dcoupler_putpert (sval(ii),nynd,nhms,'t1m',filename)

  ! increment mydate ...
  fha(:)=0.0; ida=0; jda=0
  fha(2)=nhr_obsbin! relative time interval in hours
  ida(1)=mydate(1) ! year
  ida(2)=mydate(2) ! month
  ida(3)=mydate(3) ! day
  ida(4)=0 ! time zone
  ida(5)=mydate(4) ! hour
  ! Move date-time forward by nhr_assimilation hours
  call w3movdat(fha,ida,jda)
  mydate(1)=jda(1)
  mydate(2)=jda(2)
  mydate(3)=jda(3)
  mydate(4)=jda(5)
enddo
```

The PUT from
the coupler



Prologues stripped off for display only.

Interfacing to user-specific applications

Illustration IV: **gsi_enscouplermod**

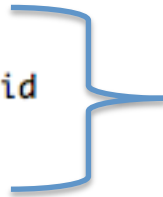
- At the moment this interface applies only to the Global option in GSI. In the future, a general interface will accommodate the regional option as well.
- The idea here is to allow users to inject their ensemble members in to GSI and its hybrid-ensemble component. For this, only a grid definition and a reader-like interface are needed. The bulk of what happens in these can be fully hidden from GSI; e.g., the GMAO interface does the read of the ensemble member through is ESMF-compliant reader.
- As illustration we show the general interface and a little detail of how the GMAO-coupling takes place. In GSI, the interfaces are defined through:
 - **gsi_enscouplermod**: An f90 interfacing defining available methods (see next page); and providing a replacement of **stub_ensmod** providing the f77 interface to allow GSI to build without the user-specific routines.
 - **cplr_ensmod**: is the set of programs defined by the user, that replace **stub_ensmod** at compilation (library build time).

Interfacing to user-specific applications

Illustration IV: **gsi_enscouplerm**od

Actual interface: **cplr_ensmod.F90**

```
!  
! !PUBLIC MEMBER FUNCTIONS:  
!  
public GSI_EnsCoupler_localization_grid  
public GSI_EnsCoupler_get_user_ens  
public GSI_EnsCoupler_put_gsi_ens
```



The only methods currently needed are:

- a grid definition
- a get – to retrieve user's members
- a put to allow writing of ensemble perturbations

NOTE: none of these are yet general enough, in particular, the get is tied up to the variables currently participating in the hybrid covariance. Some time in the near future we'll make this general so the bundle user to feed the necessary fields for hybrid can carry whatever the users desires (e.g., aerosol members, or CO, etc).

General Remarks: Adding new observing instruments

- We all know adding new instruments involves changes to the following files/procedures:
 - obsmod
 - read_obs
 - setuprhsall (and addition of corresponding new setupNEW)
 - intjo, stpjo
 - addition of intNEW and stpNEW

General Remarks: Adding new observing instruments

- What many may not know is that adding new obs-instruments also requires changes to:
 - read_obsdiags/write_obsdiags
 - setupyobs
 - obs_sensitivity
 - m_rhs (possibly, when stats involved)
- Invariably when these are not changed the basic mode of running GSI may work, but hardly any of the advanced features will.

General Remarks: Others

- Use general **intrinsic math functions**, instead of specific (only) functions, that is:
 - Sqrt() rather than Dsqrt()
 - Abs() rather than Dabs()
 - Etc
- Bundle supports both single and double precision. It is important to specify the bundle KIND when creating a bundle, as in for example:

```
write(bname,'(a)') 'State Vector'
```

```
call GSI_BundleCreate(yst,grid,bname,ierror, &  
  names2d=svars2d,names3d=svars3d,edges=edges, &  
  bundle_kind=r_kind)
```

Closing Remarks

- Please know MetGuess and ChemGuess are still under development.
- Comments and concerns are always welcome.