Reference Manual

Generated by Doxygen 1.8.13

Contents

Chapter 1

Modular arbitrary-order ocean-atmosphere model: MAOOAM -- Fortran implementation

About

This software is provided as supplementary material with:

- De Cruz, L., Demaeyer, J. and Vannitsem, S.: The Modular Arbitrary-Order Ocean-Atmosphere Model: MAOOAM v1.0, Geosci. Model Dev., 9, 2793-2808, doi:10.5194/gmd-9-2793-2016, 2016.

Please cite this article if you use (a part of) this software for a publication.

The authors would appreciate it if you could also send a reprint of your paper to lesley.decruz@meteo.be, jonathan.demaeyer@meteo.be and svn@meteo.be.

Consult the MAOOAM code repository for updates, and our website for additional resources.

A pdf version of this manual is available here.

Installation

The program can be installed with Makefile. We provide configuration files for two compilers : gfortran and ifort.

By default, gfortran is selected. To select one or the other, simply modify the Makefile accordingly or pass the COMPILER flag to `make`.

To install, unpack the archive in a folder or clone with git:

```
git clone https://github.com/Climdyn/MAOOAM.git
cd MAOOAM/fortran
```

and run:

```
make
```

The command

```
make clean
```

removes the compiled files.

For Windows users, a minimalistic GNU development environment (including gfortran and make) is available at www.mingw.org .

Description of the files

The model tendencies are represented through a tensor class called AtmOcTensor (aotensor_def::atmoctensor) which includes all the coefficients. In the standard implementation using maooam.f90 , this tensor is computed once at the program initialization.

- maooam.f90 : Main program.
- model_def.f90 : Main model class module.
- aotensor_def.f90 : Tensor class AtmOcTensor module.
- inprod_analytic.f90 : Inner products class module.
- integrator_def.f90 : A module holding the model's integrator base class definition.
- rk2_integrator.f90 : A module which contains the Heun integrator class for the model equations.
- rk2_tl_integrator.f90 : Heun Tangent Linear (TL) model integrator class module.
- rk2_ad_integrator.f90 : Heun Adjoint (AD) model integrator class module.
- rk4_integrator.f90 : A module which contains the RK4 integrator class for the model equations.
- rk4_tl_integrator.f90 : RK4 Tangent Linear (TL) model integrators module.
- rk4_ad_integrator.f90 : Adjoint (AD) model integrators module.
- Makefile : The Makefile.
- params.f90 : The model parameters classes module.
- tl_ad_tensor.f90 : Tangent Linear (TL) and Adjoint (AD) model tensors class definition module.
- test_tl_ad.f90 : Tests for the Tangent Linear (TL) and Adjoint (AD) model versions.
- README.md : A read me file.
- LICENSE.txt : The license text of the program.
- util.f90 : A module with various useful functions.
- tensor_def.f90 : Main tensor class utility module.
- stat.f90 : A module implementing a statistics accumulator class.
- params.nml : A namelist to specify the model parameters.
- int_params.nml : A namelist to specify the integration parameters.
- modeselection.nml : A namelist to specify which spectral decomposition will be used.

Usage

The user first has to fill the params.nml and int_params.nml namelist files according to their needs. Indeed, model and integration parameters can be specified respectively in the params.nml and int_params.nml namelist files. Some examples related to already published article are available in the `params` folder.

The modeselection.nml namelist can then be filled :

- NBOC and NBATM specify the number of blocks that will be used in respectively the ocean and the atmosphere. Each block corresponds to a given x and y wavenumber.

- The OMS and AMS arrays are integer arrays which specify which wavenumbers of the spectral decomposition will be used in respectively the ocean and the atmosphere. Their shapes are OMS(NBOC,2) and AMS(NB↩ ATM,2).
- The first dimension specifies the number attributed by the user to the block and the second dimension specifies the x and the y wavenumbers.
- The VDDG model is given as a default example. It is described in:
  – Vannitsem, S., Demaeyer, J., De Cruz, L., and Ghil, M.: Low-frequency variability and heat transport in a loworder nonlinear coupled ocean-atmosphere model, Physica D: Nonlinear Phenomena, 309, 71-85, doi:10.1016/j.physd.2015.07.006, 2015.
- Note that the variables of the model are numbered according to the chosen order of the blocks.

Finally, the IC.nml file specifying the initial condition should be defined. To obtain an example of this configuration file corresponding to the model you have previously defined, simply delete the current IC.nml file (if it exists) and run the program :

```
./maooam
```

It will generate a new one and start with the 0 initial condition. If you want another initial condition, stop the program, fill the newly generated file and restart :

```
./maooam
```

It will generate two files :

- evol_field.dat : the recorded time evolution of the variables.
- mean_field.dat : the mean field (the climatology)

By default, the code uses the rk2_integrator class of integrator, which integrates the model with the Heun algorithm. However, by modifying the file maooam.f90, it is possible to use the rk4_integrator class which integrates the model with the fourth-order Runge-Kutta algorithm (RK4). It is also possible to write an user-defined integrator by subclassing the base class integrator_def::integrator.

The tangent linear and adjoint models of MAOOAM are provided in the tl_ad_tensor, with integrators provided in the rk2_tl_integrator, rk2_ad_integrator, rk4_tl_integrator and rk4_ad_integrator modules. It is documented here.

Implementation notes

As the system of differential equations is at most bilinear in y_j ($j = 1..n$), $\boldsymbol{y}$ being the array of variables, it can be expressed as a tensor contraction :

$$\frac{dy_i}{dt} = \sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k}\, y_k\, y_j$$

with $y_0 = 1$.

The tensor aotensor_def::aotensor is the tensor $\mathcal{T}$ that encodes the differential equations is composed so that:

- $\mathcal{T}_{i,j,k}$ contains the contribution of dy_i/dt proportional to $y_j\, y_k$.
- Furthermore, y_0 is always equal to 1, so that $\mathcal{T}_{i,0,0}$ is the constant contribution to dy_i/dt
- $\mathcal{T}_{i,j,0} + \mathcal{T}_{i,0,j}$ is the contribution to dy_i/dt which is linear in y_j.

The tensor aotensor_def::atmoctensor is composed as an upper triangular matrix (in the last two coordinates), and its computation uses the inner products defined in a inprod_analytic::innerproducts class.

The implementation is made using Fortran classes that are linked together. It turns the model into an instanciated object that can be reused, allowing the usage of several different model versions in the same program. See the page model_def::model for a sketch of how the various classes are linked together.

Final Remarks

The authors would like to thank Kris for help with the lua2fortran project. It has greatly reduced the amount of (error-prone) work.

No animals were harmed during the coding process.

Chapter 2

Modular arbitrary-order ocean-atmosphere model: The Tangent Linear and Adjoint model

Description :

The Tangent Linear and Adjoint model model are implemented in the same way as the nonlinear model, with a tensor storing the different terms. The Tangent Linear (TL) tensor $\mathcal{T}_{i,j,k}^{TD}$ is defined as:

$$\mathcal{T}_{i,j,k}^{TL} = \mathcal{T}_{i,k,j} + \mathcal{T}_{i,j,k}$$

while the Adjoint (AD) tensor $\mathcal{T}_{i,j,k}^{AD}$ is defined as:

$$\mathcal{T}_{i,j,k}^{AD} = \mathcal{T}_{j,k,i} + \mathcal{T}_{j,i,k}.$$

where $\mathcal{T}_{i,j,k}$ is the tensor of the nonlinear model.

These two tensors are used to compute the trajectories of the models, with the equations

$$\frac{d\delta y_i}{dt} = \sum_{j=1}^{ndim} \sum_{k=0}^{ndim} \mathcal{T}_{i,j,k}^{TL} \, y_k^* \, \delta y_j.$$

$$-\frac{d\delta y_i}{dt} = \sum_{j=1}^{ndim} \sum_{k=0}^{ndim} \mathcal{T}_{i,j,k}^{AD} \, y_k^* \, \delta y_j.$$

where $\boldsymbol{y}^*$ is the point where the Tangent model is defined (with $y_0^* = 1$).

Implementation :

The two tensors are implemented in the module tl_ad_tensor and must be initialized inside a given model_def↩
::model object with the method model_def::init_tl_model and model_def::init_ad_model. The tendencies are then given by the routine model_def::tl_tendencies and model_def::ad_tendencies. Integrators with the Heun method (RK2) or the 4th-order Runge-Kutta method are available with the classes rk2_tl_integrator, rk2_ad_integrator, rk4_tl_integrator and rk4_ad_integrator. An example on how to use it can be found in the test file test_tl_ad.f90

Chapter 3

Modules Index

3.1 Modules List

Here is a list of all documented modules with brief descriptions:

Chapter 4

Data Type Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Data Type Index

5.1 Data Types List

Here are the data types with brief descriptions:

Chapter 6

Module Documentation

6.1 aotensor_def Module Reference

The equation tensor $\mathcal{T}_{i,j,k}$ for the coupled ocean-atmosphere model with temperature which allows for an extensible set of modes in the ocean and in the atmosphere.

Data Types

- type atmoctensor

  Class to hold the tensor $\mathcal{T}_{i,j,k}$ representation of the tendencies.

Functions/Subroutines

- subroutine init_aotensor (aot, model_configuration, inprods)

  Subroutine to initialise the AtmOcTensor tensor.
- subroutine delete_aotensor (aot)

  Subroutine to clean a AtmOcTensor tensor.

6.1.1 Detailed Description

The equation tensor $\mathcal{T}_{i,j,k}$ for the coupled ocean-atmosphere model with temperature which allows for an extensible set of modes in the ocean and in the atmosphere.

Copyright

6.1.2 Function/Subroutine Documentation

6.1.2.1 delete_aotensor()

```
subroutine aotensor_def::delete_aotensor (
            class(atmoctensor), intent(inout) aot )  [private]
```

Subroutine to clean a AtmOcTensor tensor.

Parameters

`in,out`	*aot*	The AtmOcTensor tensor object to initialize.

Definition at line 277 of file aotensor_def.f90.

```
277     CLASS(atmoctensor), INTENT(INOUT) :: aot
278 
279     IF (allocated(aot%count_elems)) DEALLOCATE(aot%count_elems)
280 
281     CALL aot%tensor%clean
282     NULLIFY(aot%ndim)
283     NULLIFY(aot%natm)
284     NULLIFY(aot%noc)
285 
286     aot%initialized = .false.
287 
```

6.1.2.2 init_aotensor()

```
subroutine aotensor_def::init_aotensor (
            class(atmoctensor), intent(inout) aot,
            class(modelconfiguration), intent(in), target model_configuration,
            class(innerproducts), intent(in), target inprods ) [private]
```

Subroutine to initialise the AtmOcTensor tensor.

Parameters

`in,out`	*aot*	The AO tensor object to initialize.
`in`	*model_configuration*	A model configuration object to initialize the model tensor with.
`in`	*inprods*	A model inner products object to initialize the model with.

Definition at line 221 of file aotensor_def.f90.

```
221     CLASS(atmoctensor), INTENT(INOUT) :: aot
222     CLASS(modelconfiguration), INTENT(IN), TARGET :: model_configuration
223     CLASS(innerproducts), INTENT(IN), TARGET :: inprods
224 
225     INTEGER :: i
226     INTEGER :: allocstat
227 
228     IF (.NOT.model_configuration%initialized) THEN
229       print*, "Warning: Model configuration not initialized."
230       print*, "Aborting aotensor initialization."
231       RETURN
232     END IF
233 
234     IF (.NOT.inprods%initialized) THEN
235       print*, "Warning: Inner products not initialized."
236       print*, "Aborting aotensor initialization."
237       RETURN
238     END IF
239 
240     aot%ndim => model_configuration%modes%ndim
241     aot%natm => model_configuration%modes%natm
242     aot%noc => model_configuration%modes%noc
243 
244     ALLOCATE(aot%count_elems(aot%ndim), stat=allocstat)
245     IF (allocstat /= 0) THEN
246       print*, "*** init_aotensor: Problem with allocation! ***"
```

```
      stop "Exiting ..."
    END IF
    aot%count_elems=0

    CALL aot%tensor%init(aot%ndim)

    aot%operation => ao_add_count
    CALL aot%compute_tensor(model_configuration, inprods)

    DO i=1,aot%ndim
      ALLOCATE(aot%tensor%t(i)%elems(aot%count_elems(i)), stat=allocstat)
      IF (allocstat /= 0) THEN
        print*, "*** init_aotensor: Problem with allocation! ***"
      stop "Exiting ..."
    END IF

    END DO

    aot%operation => ao_coeff
    CALL aot%compute_tensor(model_configuration, inprods)

    CALL aot%tensor%simplify

    aot%initialized = .true.

```

6.2 inprod_analytic Module Reference

Inner products between the truncated set of basis functions for the ocean and atmosphere streamfunction fields.

Data Types

- type atmosphereinnerproducts

  Class holding the atmospheric inner products functions.
- type atmosphericwavenumber

  Atmospheric bloc specification object.
- type innerproducts

  Global class for the inner products. Contains also the modes informations.
- type oceanicwavenumber

  Oceanic bloc specification object.
- type oceaninnerproducts

  Class holding the oceanic inner products functions.

Functions/Subroutines

- real(kind=8) function calculate_a (self, i, j)

  Eigenvalues of the Laplacian (atmospheric)
- real(kind=8) function calculate_b (self, i, j, k)

  Streamfunction advection terms (atmospheric)
- real(kind=8) function calculate_c_atm (self, i, j)

  Beta term for the atmosphere.
- real(kind=8) function calculate_d (self, i, j)

  Forcing of the ocean on the atmosphere.
- real(kind=8) function calculate_g (self, i, j, k)

  Temperature advection terms (atmospheric)
- real(kind=8) function calculate_s (self, i, j)

  Forcing (thermal) of the ocean on the atmosphere.
- real(kind=8) function calculate_k (self, i, j)

Forcing of the atmosphere on the ocean.

- real(kind=8) function calculate_m (self, i, j)

  Forcing of the ocean fields on the ocean.
- real(kind=8) function calculate_n (self, i, j)

  Beta term for the ocean.
- real(kind=8) function calculate_o (self, i, j, k)

  Temperature advection term (passive scalar)
- real(kind=8) function calculate_c_oc (self, i, j, k)

  Streamfunction advection terms (oceanic)
- real(kind=8) function calculate_w (self, i, j)

  Short-wave radiative forcing of the ocean.
- subroutine init_inner_products (inner_products, model_config)

  Initialization routine for the inner products functions.
- subroutine delete_inner_products (inner_products)

  Routine to clean a inner products global object.

6.2.1 Detailed Description

Inner products between the truncated set of basis functions for the ocean and atmosphere streamfunction fields.

Remarks

These are partly calculated using the analytical expressions from Cehelsky, P., & Tung, K. K. : Theories of multiple equilibria and weather regimes-A critical reexamination. Part II: Baroclinic two-layer models. Journal of the atmospheric sciences, 44(21), 3282-3303, 1987.

Copyright

2015-2020 Lesley De Cruz & Jonathan Demaeyer. See LICENSE.txt for license information.

6.2.2 Function/Subroutine Documentation

6.2.2.1 calculate_a()

```
real(kind=8) function inprod_analytic::calculate_a (
              class(atmosphereinnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j ) [private]
```

Eigenvalues of the Laplacian (atmospheric)

$a_{i,j} = (F_i, \nabla^2 F_j)$.

Definition at line 179 of file inprod_analytic.f90.

```
179     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
180     INTEGER, INTENT(IN) :: i,j
181     TYPE(atmosphericwavenumber) :: ti
182 
183     calculate_a = 0.d0
184     IF (i==j) THEN
185       ti = self%inner_products%awavenum(i)
186       calculate_a = -(self%inner_products%model_config%physics%n**2) * ti%Nx**2 - ti%Ny**2
187     END IF
```

6.2.2.2 calculate_b()

```
real(kind=8) function inprod_analytic::calculate_b (
            class(atmosphereinnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j,
            integer, intent(in) k )  [private]
```

Streamfunction advection terms (atmospheric)

$$b_{i,j,k} = (F_i, J(F_j, \nabla^2 F_k)) .$$

Definition at line 194 of file inprod_analytic.f90.

```
194     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
195     INTEGER, INTENT(IN) :: i,j,k
196
197     calculate_b = self%a(k,k) * self%g(i,j,k)
198
```

6.2.2.3 calculate_c_atm()

```
real(kind=8) function inprod_analytic::calculate_c_atm (
            class(atmosphereinnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j )  [private]
```

Beta term for the atmosphere.

$$c_{i,j} = (F_i, \partial_x F_j) .$$

Definition at line 205 of file inprod_analytic.f90.

```
205     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
206     INTEGER, INTENT(IN) :: i,j
207     TYPE(atmosphericwavenumber) :: ti, tj
208
209     ti = self%inner_products%awavenum(i)
210     tj = self%inner_products%awavenum(j)
211     calculate_c_atm = 0.d0
212     IF ((ti%typ == "K") .AND. (tj%typ == "L")) THEN
213       calculate_c_atm = ti%M * delta(ti%M - tj%H) * delta(ti%P - tj%P)
214     ELSE IF ((ti%typ == "L") .AND. (tj%typ == "K")) THEN
215       ti = self%inner_products%awavenum(j)
216       tj = self%inner_products%awavenum(i)
217       calculate_c_atm = - ti%M * delta(ti%M - tj%H) * delta(ti%P - tj%P)
218     END IF
219     calculate_c_atm = self%inner_products%model_config%physics%n * calculate_c_atm
```

6.2.2.4 calculate_c_oc()

```
real(kind=8) function inprod_analytic::calculate_c_oc (
            class(oceaninnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j,
            integer, intent(in) k ) [private]
```

Streamfunction advection terms (oceanic)

$$C_{i,j,k} = (\eta_i, J(\eta_j, \nabla^2\eta_k)) \ .$$

Definition at line 438 of file inprod_analytic.f90.

```
438     CLASS(oceaninnerproducts), INTENT(IN) :: self
439     INTEGER, INTENT(IN) :: i,j,k
440
441     calculate_c_oc = self%M(k,k) * self%O(i,j,k)
442
```

6.2.2.5 calculate_d()

```
real(kind=8) function inprod_analytic::calculate_d (
            class(atmosphereinnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j ) [private]
```

Forcing of the ocean on the atmosphere.

$$d_{i,j} = (F_i, \nabla^2\eta_j) \ .$$

Definition at line 226 of file inprod_analytic.f90.

```
226     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
227     INTEGER, INTENT(IN) :: i,j
228
229     calculate_d=self%s(i,j) * self%inner_products%ocean%M(j,j)
230
```

6.2.2.6 calculate_g()

```
real(kind=8) function inprod_analytic::calculate_g (
              class(atmosphereinnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j,
              integer, intent(in) k ) [private]
```

Temperature advection terms (atmospheric)

$$g_{i,j,k} = (F_i, J(F_j, F_k)) \, .$$

Definition at line 237 of file inprod_analytic.f90.

```
237     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
238     INTEGER, INTENT(IN) :: i,j,k
239     TYPE(atmosphericwavenumber) :: ti,tj,tk
240     REAL(KIND=8) :: val,vb1, vb2, vs1, vs2, vs3, vs4
241     INTEGER, DIMENSION(3) :: a,b
242     INTEGER, DIMENSION(3,3) :: w
243     CHARACTER, DIMENSION(3) :: s
244     INTEGER :: par
245 
246     ti = self%inner_products%awavenum(i)
247     tj = self%inner_products%awavenum(j)
248     tk = self%inner_products%awavenum(k)
249 
250     a(1)=i
251     a(2)=j
252     a(3)=k
253 
254     val=0.d0
255 
256     IF ((ti%typ == "L") .AND. (tj%typ == "L") .AND. (tk%typ == "L")) THEN
257 
258       CALL piksrt(3,a,par)
259 
260       ti = self%inner_products%awavenum(a(1))
261       tj = self%inner_products%awavenum(a(2))
262       tk = self%inner_products%awavenum(a(3))
263 
264       vs3 = s3(tj%P,tk%P,tj%H,tk%H)
265       vs4 = s4(tj%P,tk%P,tj%H,tk%H)
266       val = vs3 * ((delta(tk%H - tj%H - ti%H) - delta(tk%H &
267         &- tj%H + ti%H)) * delta(tk%P + tj%P - ti%P) +&
268         & delta(tk%H + tj%H - ti%H) * (delta(tk%P - tj%P&
269         & + ti%P) - delta(tk%P - tj%P - ti%P))) + vs4 *&
270         & ((delta(tk%H + tj%H - ti%H) * delta(tk%P - tj&
271         &%P - ti%P)) + (delta(tk%H - tj%H + ti%H) -&
272         & delta(tk%H - tj%H - ti%H)) * (delta(tk%P - tj&
273         &%P - ti%P) - delta(tk%P - tj%P + ti%P)))
274     ELSE
275 
276       s(1)=ti%typ
277       s(2)=tj%typ
278       s(3)=tk%typ
279 
280       w(1,:)=isin("A",s)
281       w(2,:)=isin("K",s)
282       w(3,:)=isin("L",s)
283 
284       IF (any(w(1,:)/=0) .AND. any(w(2,:)/=0) .AND. any(w(3,:)/=0)) THEN
285         b=w(:,1)
286         ti = self%inner_products%awavenum(a(b(1)))
287         tj = self%inner_products%awavenum(a(b(2)))
288         tk = self%inner_products%awavenum(a(b(3)))
289         call piksrt(3,b,par)
290         vb1 = b1(ti%P,tj%P,tk%P)
291         vb2 = b2(ti%P,tj%P,tk%P)
292         val = -2 * sqrt(2.) / self%inner_products%model_config%physics%pi * tj%M * delta(tj%M - tk%H) *
     flambda(ti%P + tj%P + tk%P)
293         IF (val /= 0.d0) val = val * (vb1**2 / (vb1**2 - 1) - vb2**2 / (vb2**2 - 1))
294       ELSEIF ((w(2,2)/=0) .AND. (w(2,3)==0) .AND. any(w(3,:)/=0)) THEN
295         ti = self%inner_products%awavenum(a(w(2,1)))
296         tj = self%inner_products%awavenum(a(w(2,2)))
297         tk = self%inner_products%awavenum(a(w(3,1)))
298         b(1)=w(2,1)
299         b(2)=w(2,2)
300         b(3)=w(3,1)
```

```
301         call piksrt(3,b,par)
302         vs1 = s1(tj%P,tk%P,tj%M,tk%H)
303         vs2 = s2(tj%P,tk%P,tj%M,tk%H)
304         val = vs1 * (delta(ti%M - tk%H - tj%M) * delta(ti%P -&
305           & tk%P + tj%P) - delta(ti%M- tk%H - tj%M) *&
306           & delta(ti%P + tk%P - tj%P) + (delta(tk%H - tj%M&
307           & + ti%M) + delta(tk%H - tj%M - ti%M)) *&
308           & delta(tk%P + tj%P - ti%P)) + vs2 * (delta(ti%M&
309           & - tk%H - tj%M) * delta(ti%P - tk%P - tj%P) +&
310           & (delta(tk%H - tj%M - ti%M) + delta(ti%M + tk%H&
311           & - tj%M)) * (delta(ti%P - tk%P + tj%P) -&
312           & delta(tk%P - tj%P + ti%P)))
313       ENDIF
314     ENDIF
315     calculate_g=par*val*self%inner_products%model_config%physics%n
316
```

6.2.2.7 calculate_k()

```
real(kind=8) function inprod_analytic::calculate_k (
              class(oceaninnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j )  [private]
```

Forcing of the atmosphere on the ocean.

$$K_{i,j} = (\eta_i, \nabla^2 F_j) \ .$$

Definition at line 357 of file inprod_analytic.f90.

```
357     CLASS(oceaninnerproducts), INTENT(IN) :: self
358     INTEGER, INTENT(IN) :: i,j
359
360     calculate_k = self%inner_products%atmos%s(j,i) * self%inner_products%atmos%a(j,j)
```

6.2.2.8 calculate_m()

```
real(kind=8) function inprod_analytic::calculate_m (
              class(oceaninnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j )  [private]
```

Forcing of the ocean fields on the ocean.

$$M_{i,j} = (eta_i, \nabla^2 \eta_j) \ .$$

Definition at line 367 of file inprod_analytic.f90.

```
367     CLASS(oceaninnerproducts), INTENT(IN) :: self
368     INTEGER, INTENT(IN) :: i,j
369     TYPE(oceanicwavenumber) :: di
370
371     calculate_m=0.d0
372     IF (i==j) THEN
373       di = self%inner_products%owavenum(i)
374       calculate_m = -(self%inner_products%model_config%physics%n**2) * di%Nx**2 - di%Ny**2
375     END IF
```

6.2.2.9 calculate_n()

```
real(kind=8) function inprod_analytic::calculate_n (
            class(oceaninnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j ) [private]
```

Beta term for the ocean.

$N_{i,j} = (\eta_i, \partial_x \eta_j)$.

Definition at line 382 of file inprod_analytic.f90.

```
382     CLASS(oceaninnerproducts), INTENT(IN) :: self
383     INTEGER, INTENT(IN) :: i,j
384     TYPE(oceanicwavenumber) :: di,dj
385     REAL(KIND=8) :: val
386 
387     di = self%inner_products%owavenum(i)
388     dj = self%inner_products%owavenum(j)
389     calculate_n = 0.d0
390     IF (dj%H/=di%H) THEN
391       val = delta(di%P - dj%P) * flambda(di%H + dj%H)
392       calculate_n = val * (-2) * dj%H * di%H * self%inner_products%model_config%physics%n
393       calculate_n = calculate_n / ((dj%H**2 - di%H**2) * self%inner_products%model_config%physics%pi)
394     ENDIF
395 
```

6.2.2.10 calculate_o()

```
real(kind=8) function inprod_analytic::calculate_o (
            class(oceaninnerproducts), intent(in) self,
            integer, intent(in) i,
            integer, intent(in) j,
            integer, intent(in) k ) [private]
```

Temperature advection term (passive scalar)

$O_{i,j,k} = (\eta_i, J(\eta_j, \eta_k))$.

Definition at line 402 of file inprod_analytic.f90.

```
402     CLASS(oceaninnerproducts), INTENT(IN) :: self
403     INTEGER, INTENT(IN) :: i,j,k
404     TYPE(oceanicwavenumber) :: di,dj,dk
405     REAL(KIND=8) :: vs3,vs4,val
406     INTEGER, DIMENSION(3) :: a
407     INTEGER :: par
408 
409     val=0.d0
410 
411     a(1)=i
412     a(2)=j
413     a(3)=k
414 
415     CALL piksrt(3,a,par)
416 
417     di = self%inner_products%owavenum(a(1))
418     dj = self%inner_products%owavenum(a(2))
419     dk = self%inner_products%owavenum(a(3))
420 
421     vs3 = s3(dj%P,dk%P,dj%H,dk%H)
422     vs4 = s4(dj%P,dk%P,dj%H,dk%H)
423     val = vs3*((delta(dk%H - dj%H - di%H) - delta(dk%H - dj&
424       &%H + di%H)) * delta(dk%P + dj%P - di%P) + delta(dk&
425       &%H + dj%H - di%H) * (delta(dk%P - dj%P + di%P) -&
426       & delta(dk%P - dj%P - di%P))) + vs4 * ((delta(dk%H &
427       &+ dj%H - di%H) * delta(dk%P - dj%P - di%P)) +&
428       & (delta(dk%H - dj%H + di%H) - delta(dk%H - dj%H -&
429       & di%H)) * (delta(dk%P - dj%P - di%P) - delta(dk%P &
430       &- dj%P + di%P)))
431     calculate_o = par * val * self%inner_products%model_config%physics%n / 2
```

6.2.2.11 calculate_s()

```
real(kind=8) function inprod_analytic::calculate_s (
              class(atmosphereinnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j )  [private]
```

Forcing (thermal) of the ocean on the atmosphere.

$s_{i,j} = (F_i, \eta_j)$.

Definition at line 323 of file inprod_analytic.f90.

```
323     CLASS(atmosphereinnerproducts), INTENT(IN) :: self
324     INTEGER, INTENT(IN) :: i,j
325     TYPE(atmosphericwavenumber) :: ti
326     TYPE(oceanicwavenumber) :: dj
327     REAL(KIND=8) :: val
328
329     ti = self%inner_products%awavenum(i)
330     dj = self%inner_products%owavenum(j)
331     val=0.d0
332     IF (ti%typ == "A") THEN
333       val = flambda(dj%H) * flambda(dj%P + ti%P)
334       IF (val /= 0.d0) THEN
335         val = val*8*sqrt(2.)*dj%P/(self%inner_products%model_config%physics%pi**2 * (dj%P**2 - ti%P**2) * dj%H)
336       END IF
337     ELSEIF (ti%typ == "K") THEN
338       val = flambda(2 * ti%M + dj%H) * delta(dj%P - ti%P)
339       IF (val /= 0.d0) THEN
340         val = val*4*dj%H/(self%inner_products%model_config%physics%pi * (-4 * ti%M**2 + dj%H**2))
341       END IF
342     ELSEIF (ti%typ == "L") THEN
343       val = delta(dj%P - ti%P) * delta(2 * ti%H - dj%H)
344     END IF
345     calculate_s=val
346
```

6.2.2.12 calculate_w()

```
real(kind=8) function inprod_analytic::calculate_w (
              class(oceaninnerproducts), intent(in) self,
              integer, intent(in) i,
              integer, intent(in) j )  [private]
```

Short-wave radiative forcing of the ocean.

$W_{i,j} = (\eta_i, F_j)$.

Definition at line 449 of file inprod_analytic.f90.

```
449     CLASS(oceaninnerproducts), INTENT(IN) :: self
450     INTEGER, INTENT(IN) :: i,j
451
452     calculate_w = self%inner_products%atmos%s(j,i)
453
```

6.2.2.13 delete_inner_products()

```
subroutine inprod_analytic::delete_inner_products (
              class(innerproducts), intent(inout), target inner_products )  [private]
```

Routine to clean a inner products global object.

Parameters

in,out	*inner_products*	Inner products global object to initialize

Definition at line 558 of file inprod_analytic.f90.

```
    CLASS(innerproducts), INTENT(INOUT), TARGET :: inner_products

    IF (allocated(inner_products%owavenum)) DEALLOCATE(inner_products%owavenum)
    IF (allocated(inner_products%awavenum)) DEALLOCATE(inner_products%awavenum)

    inner_products%initialized = .false.

```

6.2.2.14 init_inner_products()

```
subroutine inprod_analytic::init_inner_products (
        class(innerproducts), intent(inout), target inner_products,
        class(modelconfiguration), intent(in), target model_config )  [private]
```

Initialization routine for the inner products functions.

Parameters

in,out	*inner_products*	Inner products global object to initialize
in	*model_config*	Global model configuration object to initialize the inner products with

Definition at line 466 of file inprod_analytic.f90.

```
    CLASS(innerproducts), INTENT(INOUT), TARGET :: inner_products
    CLASS(modelconfiguration), INTENT(IN), TARGET :: model_config

    TYPE(innerproducts), POINTER :: ips

    INTEGER :: i,j
    INTEGER :: allocstat

    ips => inner_products

    IF (.NOT.model_config%initialized) THEN
      print*, "Warning: Model configuration not initialized."
      print*, "Aborting inner products initialization."
      RETURN
    END IF

    ! Definition of the types and wave numbers tables

    IF (allocated(ips%owavenum)) DEALLOCATE(ips%owavenum)
    ALLOCATE(ips%owavenum(model_config%modes%noc), stat=allocstat)
    IF (allocstat /= 0) THEN
      print*, "*** init_inner_products: Problem with allocation! ***"
      stop "Exiting ..."
    END IF


    IF (allocated(ips%awavenum)) DEALLOCATE(ips%awavenum)
    ALLOCATE(ips%awavenum(model_config%modes%natm), stat=allocstat)
    IF (allocstat /= 0) THEN
      print*, "*** init_inner_products: Problem with allocation! ***"
      stop "Exiting ..."
    END IF

```

```
    j=0
    DO i=1,model_config%modes%nbatm
      IF (model_config%modes%ams(i,1)==1) THEN
        ips%awavenum(j+1)%typ='A'
        ips%awavenum(j+2)%typ='K'
        ips%awavenum(j+3)%typ='L'

        ips%awavenum(j+1)%P=model_config%modes%ams(i,2)
        ips%awavenum(j+2)%M=model_config%modes%ams(i,1)
        ips%awavenum(j+2)%P=model_config%modes%ams(i,2)
        ips%awavenum(j+3)%H=model_config%modes%ams(i,1)
        ips%awavenum(j+3)%P=model_config%modes%ams(i,2)

        ips%awavenum(j+1)%Ny=REAL(model_config%modes%ams(i,2))
        ips%awavenum(j+2)%Nx=REAL(model_config%modes%ams(i,1))
        ips%awavenum(j+2)%Ny=REAL(model_config%modes%ams(i,2))
        ips%awavenum(j+3)%Nx=REAL(model_config%modes%ams(i,1))
        ips%awavenum(j+3)%Ny=REAL(model_config%modes%ams(i,2))

        j=j+3
      ELSE
        ips%awavenum(j+1)%typ='K'
        ips%awavenum(j+2)%typ='L'

        ips%awavenum(j+1)%M=model_config%modes%ams(i,1)
        ips%awavenum(j+1)%P=model_config%modes%ams(i,2)
        ips%awavenum(j+2)%H=model_config%modes%ams(i,1)
        ips%awavenum(j+2)%P=model_config%modes%ams(i,2)

        ips%awavenum(j+1)%Nx=REAL(model_config%modes%ams(i,1))
        ips%awavenum(j+1)%Ny=REAL(model_config%modes%ams(i,2))
        ips%awavenum(j+2)%Nx=REAL(model_config%modes%ams(i,1))
        ips%awavenum(j+2)%Ny=REAL(model_config%modes%ams(i,2))

        j=j+2

      ENDIF
    ENDDO

    DO i=1,model_config%modes%noc
      ips%owavenum(i)%H=model_config%modes%oms(i,1)
      ips%owavenum(i)%P=model_config%modes%oms(i,2)

      ips%owavenum(i)%Nx=model_config%modes%oms(i,1)/2.d0
      ips%owavenum(i)%Ny=model_config%modes%oms(i,2)

    ENDDO

    inner_products%model_config => model_config
    inner_products%atmos%inner_products => inner_products
    inner_products%ocean%inner_products => inner_products

    inner_products%initialized = .true.

```

6.3 integrator_def Module Reference

Base class definition for the model's integrators.

Data Types

- interface clean_int

  Abstract interface for the procedure to clean the integrator objects.
- interface init_int

  Abstract interface for the procedures initializing the integrator objects.
- type integrator

  Base class to be subclassed to create a new integrator.
- interface step_int

  Abstract interface for the procedure to make the integrator compute a model's time step.

6.3.1 Detailed Description

Base class definition for the model's integrators.

Copyright

6.4 model_def Module Reference

Module to articulate the model classes and define a model version.

Data Types

- type model

  Class to hold the components of a model version.

Functions/Subroutines

- type(tensor) function jacobian (imodel, ystar)

  Compute the Jacobian of MAOOAM in point $\boldsymbol{y}^*$ *and return a tensor object.*
- real(kind=8) function, dimension(imodel%ndim, imodel%ndim) jacobian_mat (imodel, ystar)

  Compute the Jacobian of MAOOAM in point $\boldsymbol{y}^*$ *and return a tensor object.*
- subroutine tendencies (imodel, t, y, res)

  Routine computing the tendencies of the model.
- subroutine ad_tendencies (imodel, t, ystar, deltay, res)

  Tendencies for the AD model of MAOOAM in point $\boldsymbol{y}^*$ *for a perturbation* $\boldsymbol{\delta y}$*.*
- subroutine tl_tendencies (imodel, t, ystar, deltay, res)

  Tendencies for the TL model of MAOOAM in point $\boldsymbol{y}^*$ *for a perturbation* $\boldsymbol{\delta y}$*.*
- subroutine init_model (imodel, physics_nml, mode_nml, int_nml)

  Subroutine to initialize the model object from NML files.
- subroutine delete_model (imodel)

  Subroutine to clean a model object.
- subroutine init_tl_model (imodel)

  Subroutine to initialize the TL tendencies tensor of a model.
- subroutine init_ad_model (imodel)

  Subroutine to initialize the AD tendencies tensor of a model.
- real(kind=8) function, dimension(0:imodel%ndim) load_ic (imodel, filename)

  Subroutine to initialize the AD tendencies tensor of a model.

6.4.1 Detailed Description

Module to articulate the model classes and define a model version.

Copyright

6.4.2 Function/Subroutine Documentation

6.4.2.1 ad_tendencies()

```
subroutine model_def::ad_tendencies (
            class(model), intent(in) imodel,
            real(kind=8), intent(in) t,
            real(kind=8), dimension(0:imodel%ndim), intent(in) ystar,
            real(kind=8), dimension(0:imodel%ndim), intent(in) deltay,
            real(kind=8), dimension(0:imodel%ndim), intent(out) res )  [private]
```

Tendencies for the AD model of MAOOAM in point $\boldsymbol{y}^*$ for a perturbation $\boldsymbol{\delta y}$.

Parameters

in	*imodel*	Model to compute the AD tendencies of.
in	*t*	time
in	*ystar*	Vector $\boldsymbol{y}^*$ (current point in model's trajectory).
in	*deltay*	Vector $\boldsymbol{\delta y}$, i.e. the perturbation of the variables at time t.
out	*res*	Vector to store the tendencies.

Definition at line 99 of file model_def.f90.

```
99      CLASS(model), INTENT(IN) :: imodel
100      REAL(KIND=8), INTENT(IN) :: t
101      REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(IN) :: ystar,deltay
102      REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(OUT) :: res
103      CALL imodel%adtensor%tensor%sparse_mul3(deltay, ystar, res)
```

6.4.2.2 delete_model()

```
subroutine model_def::delete_model (
            class(model), intent(inout) imodel )  [private]
```

Subroutine to clean a model object.

Parameters

in,out	*imodel*	Model object to clean.

Definition at line 152 of file model_def.f90.

```
152      CLASS(model), INTENT(INOUT) :: imodel
153
154      NULLIFY(imodel%ndim)
155
156      CALL imodel%model_configuration%clean
```

```
    CALL imodel%inner_products%clean
    CALL imodel%aotensor%clean

    CALL imodel%tltensor%clean
    CALL imodel%adtensor%clean

    imodel%initialized = .false.

```

6.4.2.3 init_ad_model()

```
subroutine model_def::init_ad_model (
            class(model), intent(inout), target imodel )  [private]
```

Subroutine to initialize the AD tendencies tensor of a model.

Parameters

in,out	*imodel*	Model object to initialize.

Definition at line 186 of file model_def.f90.

```
186     CLASS(model), INTENT(INOUT), TARGET :: imodel
187
188     CALL imodel%adtensor%clean
189
190     IF (.NOT.imodel%initialized) THEN
191       print*, "*** init_ad_model: Trying to initialize AD model of an uninitialized model ! ***"
192       print*, "Please first initialize the model before trying to initialize the AD model."
193       print*, "Aborting operation."
194       RETURN
195     END IF
196     CALL imodel%adtensor%init(imodel%aotensor)
```

6.4.2.4 init_model()

```
subroutine model_def::init_model (
            class(model), intent(inout), target imodel,
            character(len=*), intent(in), optional physics_nml,
            character(len=*), intent(in), optional mode_nml,
            character(len=*), intent(in), optional int_nml )  [private]
```

Subroutine to initialize the model object from NML files.

Parameters

in,out	*imodel*	Model object to initialize.
in	*physics_nml*	Physical parameters namelist filename
in	*mode_nml*	Modes configuration namelist filename
in	*int_nml*	Numerical integration parameters namelist filename

Remarks

If no NML filenames are provided, it will assume that the standard filenames of the model NML have to be used (e.g. "params.nml", "modeselection.nml" and "int_params.nml").

Definition at line 127 of file model_def.f90.

```
127     CLASS(model), INTENT(INOUT), TARGET :: imodel
128     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: physics_nml
129     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: mode_nml
130     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: int_nml
131     LOGICAL :: ok
132 
133     CALL imodel%model_configuration%clean
134     CALL imodel%inner_products%clean
135     CALL imodel%aotensor%clean
136 
137     CALL imodel%model_configuration%init(physics_nml=physics_nml, mode_nml=mode_nml, int_nml=int_nml)
138     CALL imodel%inner_products%init(imodel%model_configuration)
139     CALL imodel%aotensor%init(imodel%model_configuration, imodel%inner_products)
140 
141     imodel%ndim => imodel%model_configuration%modes%ndim
142 
143     ok = imodel%model_configuration%initialized .AND. imodel%inner_products%initialized .AND. imodel%aotensor%initialized
144 
145     if (ok) imodel%initialized = .true.
146 
```

6.4.2.5 init_tl_model()

```
subroutine model_def::init_tl_model (
            class(model), intent(inout), target imodel )  [private]
```

Subroutine to initialize the TL tendencies tensor of a model.

Parameters

in,out	*imodel*	Model object to initialize.

Definition at line 170 of file model_def.f90.

```
170     CLASS(model), INTENT(INOUT), TARGET :: imodel
171 
172     CALL imodel%tltensor%clean
173 
174     IF (.NOT.imodel%initialized) THEN
175       print*, "*** init_tl_model: Trying to initialize TL model of an uninitialized model ! ***"
176       print*, "Please first initialize the model before trying to initialize the TL model."
177       print*, "Aborting operation."
178       RETURN
179     END IF
180     CALL imodel%tltensor%init(imodel%aotensor)
```

6.4.2.6 jacobian()

```
type(tensor) function model_def::jacobian (
            class(model), intent(in) imodel,
            real(kind=8), dimension(0:imodel%ndim), intent(in) ystar )
```

Compute the Jacobian of MAOOAM in point $\boldsymbol{y}^*$ and return a tensor object.

Parameters

in	*imodel*	Model to return the Jacobian of.
in	*ystar*	Vector $\boldsymbol{y}^*$ at which the jacobian should be evaluated.

Returns

Jacobian in tensor form (table of tuples {i,j,0,value}).

Definition at line 59 of file model_def.f90.

```
CLASS(model), INTENT(IN) :: imodel
REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(IN) :: ystar
TYPE(tensor) :: jacobian
CALL jacobian%init(imodel%ndim)
CALL imodel%aotensor%tensor%jsparse_mul(ystar,jacobian)
```

6.4.2.7 jacobian_mat()

```
real(kind=8) function, dimension(imodel%ndim,imodel%ndim) model_def::jacobian_mat (
            class(model), intent(in) imodel,
            real(kind=8), dimension(0:imodel%ndim), intent(in) ystar )  [private]
```

Compute the Jacobian of MAOOAM in point $\boldsymbol{y}^*$ and return a tensor object.

Parameters

in	*imodel*	Model to return the Jacobian of.
in	*ystar*	Vector $\boldsymbol{y}^*$ at which the jacobian should be evaluated.

Returns

Jacobian in matrix form.

Definition at line 71 of file model_def.f90.

```
CLASS(model), INTENT(IN) :: imodel
REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(IN) :: ystar
REAL(KIND=8), DIMENSION(imodel%ndim,imodel%ndim) :: jacobian_mat
CALL imodel%aotensor%tensor%jsparse_mul_mat(ystar,jacobian_mat)
```

6.4.2.8 load_ic()

```
real(kind=8) function, dimension(0:imodel%ndim) model_def::load_ic (
            class(model), intent(in), target imodel,
            character(len=*), intent(in), optional filename )  [private]
```

Subroutine to initialize the AD tendencies tensor of a model.

Parameters

in	*imodel*	Model object for wich to load the initial condition.
in	*filename*	Filename of the initial condition NML file.

Returns

A vector with the initial condition.

Definition at line 204 of file model_def.f90.

```
204     CLASS(model), INTENT(IN), TARGET :: imodel
205     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: filename
206     REAL(KIND=8), DIMENSION(0:imodel%ndim) :: ic
207
208     INTEGER :: i,allocstat,j
209     INTEGER, POINTER :: ndim, natm, noc
210     CHARACTER(len=20) :: fm
211     REAL(KIND=8) :: size_of_random_noise
212     INTEGER, DIMENSION(:), ALLOCATABLE :: seed
213     CHARACTER(LEN=4) :: init_type
214     LOGICAL :: exists, std
215     namelist /iclist/ ic
216     namelist /rand/ init_type,size_of_random_noise,seed
217
218     fm(1:6)='(F3.1)'
219
220     IF (.NOT.imodel%initialized) THEN
221       print*, 'Model not yet initialized, impossible to load any initial condition!'
222       RETURN
223     END IF
224
225     CALL random_seed(size=j)
226
227     ndim => imodel%model_configuration%modes%ndim
228     natm => imodel%model_configuration%modes%natm
229     noc => imodel%model_configuration%modes%noc
230
231     ALLOCATE(seed(j), stat=allocstat)
232     IF (allocstat /= 0) THEN
233       print*, "*** load_ic: Problem with allocation! ***"
234       stop "Exiting ..."
235     END IF
236
237
238     IF (present(filename)) THEN
239       INQUIRE(file=filename,exist=exists)
240       std = .false.
241     ELSE
242       print*, "Warning: IC filename not provided."
243       print*, "Trying to load the standard file IC.nml instead ..."
244       INQUIRE(file='./IC.nml',exist=exists)
245       std = .true.
246     END IF
247
248     IF (exists) THEN
249       IF (std) THEN
250         OPEN(8, file="IC.nml", status='OLD', recl=80, delim='APOSTROPHE')
251       ELSE
252         OPEN(8, file=filename, status='OLD', recl=80, delim='APOSTROPHE')
253       END IF
254       READ(8,nml=iclist)
255       READ(8,nml=rand)
256       CLOSE(8)
257       SELECT CASE (init_type)
258         CASE ('seed')
259           CALL random_seed(put=seed)
260           CALL random_number(ic)
261           ic=2*(ic-0.5)
262           ic=ic*size_of_random_noise*10.d0
263           ic(0)=1.0d0
264           WRITE(6,*) "*** Namelist file written. Starting with 'seeded' random initial condition !***"
265         CASE ('rand')
266           CALL init_random_seed()
267           CALL random_seed(get=seed)
268           CALL random_number(ic)
269           ic=2*(ic-0.5)
270           ic=ic*size_of_random_noise*10.d0
```

```
      ic(0)=1.0d0
      WRITE(6,*) "*** Namelist file written. Starting with random initial condition !***"
    CASE ('zero')
      CALL init_random_seed()
      CALL random_seed(get=seed)
      ic=0
      ic(0)=1.0d0
      WRITE(6,*) "*** Namelist file written. Starting with initial condition in IC.nml !***"
    CASE ('read')
      CALL init_random_seed()
      CALL random_seed(get=seed)
      ic(0)=1.0d0
      ! except IC(0), nothing has to be done IC has already the right values
      WRITE(6,*) "*** Namelist file written. Starting with initial condition in IC.nml !***"
  END SELECT
ELSE
  CALL init_random_seed()
  CALL random_seed(get=seed)
  ic=0
  ic(0)=1.0d0
  init_type="zero"
  size_of_random_noise=0.d0
  WRITE(6,*) "*** Namelist file written. Starting with 0 as initial condition !***"
END IF
IF (std) THEN
  OPEN(8, file="IC.nml", status='REPLACE')
ELSE
  OPEN(8, file=filename, status='REPLACE')
END IF
WRITE(8,'(a)') "!------------------------------------------------------------------------------!"
WRITE(8,'(a)') "! Namelist file :                                                              !"
WRITE(8,'(a)') "! Initial condition.                                                           !"
WRITE(8,'(a)') "!------------------------------------------------------------------------------!"
WRITE(8,*) ""
WRITE(8,'(a)') "&ICLIST"
WRITE(8,*) " ! psi variables"
DO i=1,natm
  WRITE(8,*) " IC("//trim(str(i))//") = ",ic(i),"   ! typ= "&
    &//imodel%inner_products%awavenum(i)%typ//", Nx= "//trim(rstr(imodel%inner_products%awavenum(i)&
    &%Nx,fm))//", Ny= "//trim(rstr(imodel%inner_products%awavenum(i)%Ny,fm))
END DO
WRITE(8,*) " ! theta variables"
DO i=1,natm
  WRITE(8,*) " IC("//trim(str(i+natm))//") = ",ic(i+natm),"   ! typ= "&
    &//imodel%inner_products%awavenum(i)%typ//", Nx= "//trim(rstr(imodel%inner_products%awavenum(i)&
    &%Nx,fm))//", Ny= "//trim(rstr(imodel%inner_products%awavenum(i)%Ny,fm))
END DO

WRITE(8,*) " ! A variables"
DO i=1,noc
  WRITE(8,*) " IC("//trim(str(i+2*natm))//") = ",ic(i+2*natm),"   ! Nx&
        &= "          //trim(rstr(imodel%inner_products%owavenum(i)%Nx,fm))//", Ny= "&
    &//trim(rstr(imodel%inner_products%owavenum(i)%Ny,fm))
END DO
WRITE(8,*) " ! T variables"
DO i=1,noc
  WRITE(8,*) " IC("//trim(str(i+noc+2*natm))//") = ",ic(i+2*natm+noc),"   &
        &! Nx= "          //trim(rstr(imodel%inner_products%owavenum(i)%Nx,fm))//", Ny= "&
    &//trim(rstr(imodel%inner_products%owavenum(i)%Ny,fm))
END DO

WRITE(8,'(a)') "&END"
WRITE(8,*) ""
WRITE(8,'(a)') "!------------------------------------------------------------------------------!"
WRITE(8,'(a)') "! Initialisation type.                                                         !"
WRITE(8,'(a)') "!------------------------------------------------------------------------------!"
WRITE(8,'(a)') "! type = 'read': use IC above (will generate a new seed);"
WRITE(8,'(a)') "!        'rand': random state (will generate a new seed);"
WRITE(8,'(a)') "!        'zero': zero IC (will generate a new seed);"
WRITE(8,'(a)') "!        'seed': use the seed below (generate the same IC)"
WRITE(8,*) ""
WRITE(8,'(a)') "&RAND"
WRITE(8,'(a)') "  init_type= '"//init_type//"'"
WRITE(8,'(a,d15.7)') "  size_of_random_noise = ",size_of_random_noise
DO i=1,j
  WRITE(8,*) " seed("//trim(str(i))//") = ",seed(i)
END DO
WRITE(8,'(a)') "&END"
WRITE(8,*) ""
CLOSE(8)
```

6.4.2.9 tendencies()

```
subroutine model_def::tendencies (
            class(model), intent(in) imodel,
            real(kind=8), intent(in) t,
            real(kind=8), dimension(0:imodel%ndim), intent(in) y,
            real(kind=8), dimension(0:imodel%ndim), intent(out) res )  [private]
```

Routine computing the tendencies of the model.

Parameters

in	*imodel*	Model to compute the tendencies of.
in	*t*	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
in	*y*	Point at which the tendencies have to be computed.
out	*res*	Vector to store the tendencies.

Remarks

Note that it is NOT safe to pass `y` as a result buffer, as this operation does multiple passes.

Definition at line 85 of file model_def.f90.

```
85     CLASS(model), INTENT(IN) :: imodel
86     REAL(KIND=8), INTENT(IN) :: t
87     REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(IN) :: y
88     REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(OUT) :: res
89     CALL imodel%aotensor%tensor%sparse_mul3(y, y, res)
```

6.4.2.10 tl_tendencies()

```
subroutine model_def::tl_tendencies (
            class(model), intent(in) imodel,
            real(kind=8), intent(in) t,
            real(kind=8), dimension(0:imodel%ndim), intent(in) ystar,
            real(kind=8), dimension(0:imodel%ndim), intent(in) deltay,
            real(kind=8), dimension(0:imodel%ndim), intent(out) res )  [private]
```

Tendencies for the TL model of MAOOAM in point $\boldsymbol{y}^*$ for a perturbation $\delta\boldsymbol{y}$.

Parameters

in	*imodel*	Model to compute the TL tendencies of.
in	*t*	time
in	*ystar*	Vector $\boldsymbol{y}^*$ (current point in model's trajectory).
in	*deltay*	Vector $\delta\boldsymbol{y}$, i.e. the perturbation of the variables at time t.
out	*res*	Vector to store the tendencies.

Definition at line 113 of file model_def.f90.

```
113     CLASS(model), INTENT(IN) :: imodel
114     REAL(KIND=8), INTENT(IN) :: t
115     REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(IN) :: ystar,deltay
116     REAL(KIND=8), DIMENSION(0:imodel%ndim), INTENT(OUT) :: res
117     CALL imodel%tltensor%tensor%sparse_mul3(deltay,ystar,res)
```

6.5 params Module Reference

The model parameters module.

Data Types

- type integrationparameters

  The subclass containing the integration parameters.
- type modelconfiguration

  The general class holding the model configuration.
- type modesconfiguration

  The subclass containing the modes parameters.
- type physicsconfiguration

  The subclass containing the physical parameters of the model.

Functions/Subroutines

- subroutine init_model_config (model_config, physics_nml, mode_nml, int_nml)

  Subroutine to initialize the model configuration with NML files. Reads the physical parameters and mode selection from the namelist.
- subroutine clean_model_config (model_config)

  Subroutine to clean the model configuraion object.

6.5.1 Detailed Description

The model parameters module.

Copyright

2015-2020 Lesley De Cruz & Jonathan Demaeyer. See LICENSE.txt for license information.

6.5.2 Function/Subroutine Documentation

6.5.2.1 clean_model_config()

```
subroutine params::clean_model_config (
            class(modelconfiguration), intent(inout) model_config )
```

Subroutine to clean the model configuraion object.

Parameters

in,out	*model_config*	Model configuration object

Definition at line 159 of file params.f90.

```
159     CLASS(modelconfiguration), INTENT(INOUT) :: model_config
160
161     CALL model_config%modes%clean
162     model_config%initialized = .false.
163     model_config%integration%initialized = .false.
164     model_config%physics%initialized = .false.
165
```

6.5.2.2 init_model_config()

```
subroutine params::init_model_config (
            class(modelconfiguration), intent(inout) model_config,
            character(len=*), intent(in), optional physics_nml,
            character(len=*), intent(in), optional mode_nml,
            character(len=*), intent(in), optional int_nml )
```

Subroutine to initialize the model configuration with NML files. Reads the physical parameters and mode selection from the namelist.

Parameters

in,out	*model_config*	Model configuration object
in	*physics_nml*	Physical parameters namelist filename
in	*mode_nml*	Modes configuration namelist filename
in	*int_nml*	Numerical integration parameters namelist filename

Remarks

If no NML filenames are provided, it will assume that the standard filenames of the model NML have to be used (e.g. "params.nml", "modeselection.nml" and "int_params.nml").

Definition at line 127 of file params.f90.

```
127     CLASS(modelconfiguration), INTENT(INOUT) :: model_config
128     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: physics_nml
129     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: mode_nml
130     CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: int_nml
131
132     IF (present(physics_nml)) THEN
133       CALL model_config%physics%init(physics_nml)
134     ELSE
135       CALL model_config%physics%init
136     END IF
137
138     IF (present(mode_nml)) THEN
139       CALL model_config%modes%init(mode_nml)
140     ELSE
141       CALL model_config%modes%init
142     END IF
143
```

```
    IF (present(int_nml)) THEN
      CALL model_config%integration%init(int_nml)
    ELSE
      CALL model_config%integration%init
    END IF

    IF ((model_config%physics%initialized).AND.(model_config%modes%initialized).AND.(model_config
%integration%initialized)) THEN
      model_config%initialized = .true.
    END IF

```

6.6 rk2_ad_integrator Module Reference

Adjoint (AD) model versions of MAOOAM. Second-order Runge-Kutta (RK2) integrators module.

Data Types

- type rk2adintegrator

  Class for the Heun (RK2) AD integrator object.

Functions/Subroutines

- subroutine ad_step (integr, y, ystar, t, res)

  Routine to perform an integration step (Heun algorithm) of the adjoint model. The incremented time is returned.

6.6.1 Detailed Description

Adjoint (AD) model versions of MAOOAM. Second-order Runge-Kutta (RK2) integrators module.

Copyright

Remarks

This module actually contains the Heun algorithm routines.

6.6.2 Function/Subroutine Documentation

6.6.2.1 ad_step()

```
subroutine rk2_ad_integrator::ad_step (
            class(rk2adintegrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), dimension(0:integr%ndim), intent(in) ystar,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )
```

Routine to perform an integration step (Heun algorithm) of the adjoint model. The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*ystar*	Evaluating the adjoint model at the point y^*.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 49 of file rk2_ad_integrator.f90.

```
49      CLASS(rk2adintegrator), INTENT(INOUT) :: integr
50      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y,ystar
51      REAL(KIND=8), INTENT(INOUT) :: t
52      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res
53
54      CALL integr%pmodel%ad_tendencies(t,ystar,y,integr%buf_f0)
55      integr%buf_y1 = y+integr%dt*integr%buf_f0
56      CALL integr%pmodel%ad_tendencies(t+integr%dt,ystar,integr%buf_y1,integr%buf_f1)
57      res=y+0.5*(integr%buf_f0+integr%buf_f1)*integr%dt
58      t=t+integr%dt
```

6.7 rk2_integrator Module Reference

Module containing the second-order Runge-Kutta (RK2) integration routines.

Data Types

- type rk2integrator

  Class for the Heun (RK2) integrator object.

Functions/Subroutines

- subroutine init (integr, imodel)

  Routine to initialise the integration buffers.
- subroutine step (integr, y, t, res)

  Routine to perform an integration step (Heun algorithm). The incremented time is returned.
- subroutine clean (integr)

  Routine to clean the integrator.

6.7.1 Detailed Description

Module containing the second-order Runge-Kutta (RK2) integration routines.

Copyright

Remarks

This module actually contains the Heun algorithm routines.

6.7.2 Function/Subroutine Documentation

6.7.2.1 clean()

```
subroutine rk2_integrator::clean (
            class(rk2integrator), intent(inout) integr )  [private]
```

Routine to clean the integrator.

Parameters

in,out	*integr*	Integrator object to clean.

Definition at line 92 of file rk2_integrator.f90.

```
92       CLASS(rk2integrator), INTENT(INOUT) :: integr
93
94       IF (allocated(integr%buf_y1)) DEALLOCATE(integr%buf_y1)
95       IF (allocated(integr%buf_f1)) DEALLOCATE(integr%buf_f1)
96       IF (allocated(integr%buf_f0)) DEALLOCATE(integr%buf_f0)
97
```

6.7.2.2 init()

```
subroutine rk2_integrator::init (
            class(rk2integrator), intent(inout) integr,
            class(model), intent(in), target imodel )
```

Routine to initialise the integration buffers.

Parameters

in,out	*integr*	Integrator object to initialize.
in	*imodel*	Model object to initialize the integrator with.

Definition at line 41 of file rk2_integrator.f90.

```
41     CLASS(rk2integrator), INTENT(INOUT) :: integr
42     CLASS(model), INTENT(IN), TARGET :: imodel
43     INTEGER :: allocstat
44     IF (.NOT.imodel%initialized) THEN
45       print*, 'Model not yet initialized, impossible to associate an integrator to an empty model!'
46       RETURN
47     END IF
48
49     integr%pmodel => imodel
50     integr%dt => imodel%model_configuration%integration%dt
51     integr%ndim => imodel%model_configuration%modes%ndim
52
53     ALLOCATE(integr%buf_y1(0:integr%ndim) ,stat=allocstat)
54     IF (allocstat /= 0) THEN
```

```
      print*, "*** rk2integrator%init: Problem with allocation! ***"
      stop "Exiting ..."
    END IF
    ALLOCATE(integr%buf_f0(0:integr%ndim) ,stat=allocstat)
    IF (allocstat /= 0) THEN
      print*, "*** rk2integrator%init: Problem with allocation! ***"
      stop "Exiting ..."
    END IF
    ALLOCATE(integr%buf_f1(0:integr%ndim) ,stat=allocstat)
    IF (allocstat /= 0) THEN
      print*, "*** rk2integrator%init: Problem with allocation! ***"
      stop "Exiting ..."
    END IF

```

6.7.2.3 step()

```
subroutine rk2_integrator::step (
            class(rk2integrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )  [private]
```

Routine to perform an integration step (Heun algorithm). The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 77 of file rk2_integrator.f90.

```
77    CLASS(rk2integrator), INTENT(INOUT) :: integr
78    REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y
79    REAL(KIND=8), INTENT(INOUT) :: t
80    REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res
81
82    CALL integr%pmodel%tendencies(t,y,integr%buf_f0)
83    integr%buf_y1 = y+integr%dt*integr%buf_f0
84    CALL integr%pmodel%tendencies(t+integr%dt,integr%buf_y1,integr%buf_f1)
85    res=y+0.5*(integr%buf_f0+integr%buf_f1)*integr%dt
86    t=t+integr%dt
```

6.8 rk2_tl_integrator Module Reference

Tangent Linear (TL) model versions of MAOOAM. Second-order Runge-Kutta (RK2) integrators module.

Data Types

- type rk2tlintegrator

    Class for the Heun (RK2) TL integrator object.

Functions/Subroutines

- subroutine tl_step (integr, y, ystar, t, res)

  Routine to perform an integration step (Heun algorithm) of the tangent linear model. The incremented time is returned.

6.8.1 Detailed Description

Tangent Linear (TL) model versions of MAOOAM. Second-order Runge-Kutta (RK2) integrators module.

Copyright

Remarks

This module actually contains the Heun algorithm routines.

6.8.2 Function/Subroutine Documentation

6.8.2.1 tl_step()

```
subroutine rk2_tl_integrator::tl_step (
            class(rk2tlintegrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), dimension(0:integr%ndim), intent(in) ystar,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )
```

Routine to perform an integration step (Heun algorithm) of the tangent linear model. The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*ystar*	Evaluating the adjoint model at the point y^*.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 49 of file rk2_tl_integrator.f90.

```
49     CLASS(rk2tlintegrator), INTENT(INOUT) :: integr
50     REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y,ystar
51     REAL(KIND=8), INTENT(INOUT) :: t
52     REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res
53
54     CALL integr%pmodel%tl_tendencies(t,ystar,y,integr%buf_f0)
55     integr%buf_y1 = y+integr%dt*integr%buf_f0
```

```
    CALL integr%pmodel%tl_tendencies(t+integr%dt,ystar,integr%buf_y1,integr%buf_f1)
    res=y+0.5*(integr%buf_f0+integr%buf_f1)*integr%dt
    t=t+integr%dt
```

6.9 rk4_ad_integrator Module Reference

Adjoint (AD) model versions of MAOOAM. Fourth-order Runge-Kutta (RK4) integrators module.

Data Types

- type rk4adintegrator

    Class for the fourth-order Runge-Kutta (RK4) AD integrator object.

Functions/Subroutines

- subroutine ad_step (integr, y, ystar, t, res)

    Routine to perform an integration step (RK4 algorithm) of the adjoint model. The incremented time is returned.

6.9.1 Detailed Description

Adjoint (AD) model versions of MAOOAM. Fourth-order Runge-Kutta (RK4) integrators module.

Copyright

6.9.2 Function/Subroutine Documentation

6.9.2.1 ad_step()

```
subroutine rk4_ad_integrator::ad_step (
            class(rk4adintegrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), dimension(0:integr%ndim), intent(in) ystar,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )
```

Routine to perform an integration step (RK4 algorithm) of the adjoint model. The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*ystar*	Evaluating the adjoint model at the point y^*.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 47 of file rk4_ad_integrator.f90.

```
CLASS(rk4adintegrator), INTENT(INOUT) :: integr
REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y,ystar
REAL(KIND=8), INTENT(INOUT) :: t
REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res

CALL integr%pmodel%ad_tendencies(t,ystar,y,integr%buf_kA)
integr%buf_y1 = y+0.5*integr%dt*integr%buf_kA
CALL integr%pmodel%ad_tendencies(t+0.5*integr%dt,ystar,integr%buf_y1,integr%buf_kB)
integr%buf_y1 = y+0.5*integr%dt*integr%buf_kB
integr%buf_kA = integr%buf_kA+2*integr%buf_kB
CALL integr%pmodel%ad_tendencies(t+0.5*integr%dt,ystar,integr%buf_y1,integr%buf_kB)
integr%buf_y1 = y+0.5*integr%dt*integr%buf_kB
integr%buf_kA = integr%buf_kA+2*integr%buf_kB
CALL integr%pmodel%ad_tendencies(t+integr%dt,ystar,integr%buf_y1,integr%buf_kB)
integr%buf_kA = integr%buf_kA+integr%buf_kB
res=y+integr%buf_kA*integr%dt/6
t=t+integr%dt
```

6.10 rk4_integrator Module Reference

Module containing the fourth-order Runge-Kutta (RK4) integration routines.

Data Types

- type rk4integrator

  Class for the fourth-order Runge-Kutta (RK4) integrator object.

Functions/Subroutines

- subroutine init (integr, imodel)

  Routine to initialise the integration buffers.
- subroutine step (integr, y, t, res)

  Routine to perform an integration step (RK4 algorithm). The incremented time is returned.
- subroutine clean (integr)

  Routine to clean the integrator.

6.10.1 Detailed Description

Module containing the fourth-order Runge-Kutta (RK4) integration routines.

Copyright

6.10.2 Function/Subroutine Documentation

6.10.2.1 clean()

```
subroutine rk4_integrator::clean (
            class(rk4integrator), intent(inout) integr )  [private]
```

Routine to clean the integrator.

Parameters

in,out	*integr*	Integrator object to clean.

Definition at line 101 of file rk4_integrator.f90.

```
101       CLASS(rk4integrator), INTENT(INOUT) :: integr
102
103       IF (allocated(integr%buf_y1)) DEALLOCATE(integr%buf_y1)
104       IF (allocated(integr%buf_kA)) DEALLOCATE(integr%buf_kA)
105       IF (allocated(integr%buf_kB)) DEALLOCATE(integr%buf_kB)
106
```

6.10.2.2 init()

```
subroutine rk4_integrator::init (
            class(rk4integrator), intent(inout) integr,
            class(model), intent(in), target imodel )
```

Routine to initialise the integration buffers.

Parameters

in,out	*integr*	Integrator object to initialize.
in	*imodel*	Model object to initialize the integrator with.

Definition at line 39 of file rk4_integrator.f90.

```
39      CLASS(rk4integrator), INTENT(INOUT) :: integr
40      CLASS(model), INTENT(IN), TARGET :: imodel
41      INTEGER :: allocstat
42      IF (.NOT.imodel%initialized) THEN
43        print*, 'Model not yet initialized, impossible to associate an integrator to an empty model!'
44        RETURN
45      END IF
46
47      integr%pmodel => imodel
48      integr%dt => imodel%model_configuration%integration%dt
49      integr%ndim => imodel%model_configuration%modes%ndim
50
51      ALLOCATE(integr%buf_y1(0:integr%ndim) ,stat=allocstat)
52      IF (allocstat /= 0) THEN
53        print*, "*** rk4integrator%init: Problem with allocation! ***"
54        stop "Exiting ..."
55      END IF
56      ALLOCATE(integr%buf_kA(0:integr%ndim) ,stat=allocstat)
57      IF (allocstat /= 0) THEN
58        print*, "*** rk4integrator%init: Problem with allocation! ***"
59        stop "Exiting ..."
60      END IF
61      ALLOCATE(integr%buf_kB(0:integr%ndim) ,stat=allocstat)
62      IF (allocstat /= 0) THEN
63        print*, "*** rk4integrator%init: Problem with allocation! ***"
64        stop "Exiting ..."
65      END IF
66
```

6.10.2.3 step()

```
subroutine rk4_integrator::step (
            class(rk4integrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )  [private]
```

Routine to perform an integration step (RK4 algorithm). The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 75 of file rk4_integrator.f90.

```
75      CLASS(rk4integrator), INTENT(INOUT) :: integr
76      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y
77      REAL(KIND=8), INTENT(INOUT) :: t
78      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res
79
80      CALL integr%pmodel%tendencies(t,y,integr%buf_kA)
81      integr%buf_y1 = y + 0.5*integr%dt*integr%buf_kA
82
83      CALL integr%pmodel%tendencies(t+0.5*integr%dt,integr%buf_y1,integr%buf_kB)
84      integr%buf_y1 = y + 0.5*integr%dt*integr%buf_kB
85      integr%buf_kA = integr%buf_kA + 2*integr%buf_kB
86
87      CALL integr%pmodel%tendencies(t+0.5*integr%dt,integr%buf_y1,integr%buf_kB)
88      integr%buf_y1 = y + integr%dt*integr%buf_kB
89      integr%buf_kA = integr%buf_kA + 2*integr%buf_kB
90
91      CALL integr%pmodel%tendencies(t+integr%dt,integr%buf_y1,integr%buf_kB)
92      integr%buf_kA = integr%buf_kA + integr%buf_kB
93
94      t=t+integr%dt
95      res=y+integr%buf_kA*integr%dt/6
```

6.11 rk4_tl_integrator Module Reference

Tangent Linear (TL) model versions of MAOOAM. Fourth-order Runge-Kutta (RK4) integrators module.

Data Types

- type rk4tlintegrator

    Class for the fourth-order Runge-Kutta (RK4) TL integrator object.

Functions/Subroutines

- subroutine tl_step (integr, y, ystar, t, res)

    Routine to perform an integration step (RK4 algorithm) of the tangent linear model. The incremented time is returned.

6.11.1 Detailed Description

Tangent Linear (TL) model versions of MAOOAM. Fourth-order Runge-Kutta (RK4) integrators module.

Copyright

2020 Lesley De Cruz, Jonathan Demaeyer & Sebastian Schubert. See LICENSE.txt for license information.

6.11.2 Function/Subroutine Documentation

6.11.2.1 tl_step()

```
subroutine rk4_tl_integrator::tl_step (
            class(rk4tlintegrator), intent(inout) integr,
            real(kind=8), dimension(0:integr%ndim), intent(in) y,
            real(kind=8), dimension(0:integr%ndim), intent(in) ystar,
            real(kind=8), intent(inout) t,
            real(kind=8), dimension(0:integr%ndim), intent(out) res )
```

Routine to perform an integration step (RK4 algorithm) of the tangent linear model. The incremented time is returned.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*ystar*	Evaluating the adjoint model at the point y^*.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 47 of file rk4_tl_integrator.f90.

```
47      CLASS(rk4tlintegrator), INTENT(INOUT) :: integr
48      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(IN) :: y,ystar
49      REAL(KIND=8), INTENT(INOUT) :: t
50      REAL(KIND=8), DIMENSION(0:integr%ndim), INTENT(OUT) :: res
51
52      CALL integr%pmodel%tl_tendencies(t,ystar,y,integr%buf_kA)
53      integr%buf_y1 = y+0.5*integr%dt*integr%buf_kA
54      CALL integr%pmodel%tl_tendencies(t+0.5*integr%dt,ystar,integr%buf_y1,integr%buf_kB)
55      integr%buf_y1 = y+0.5*integr%dt*integr%buf_kB
56      integr%buf_kA = integr%buf_kA+2*integr%buf_kB
57      CALL integr%pmodel%tl_tendencies(t+0.5*integr%dt,ystar,integr%buf_y1,integr%buf_kB)
58      integr%buf_y1 = y+0.5*integr%dt*integr%buf_kB
59      integr%buf_kA = integr%buf_kA+2*integr%buf_kB
60      CALL integr%pmodel%tl_tendencies(t+integr%dt,ystar,integr%buf_y1,integr%buf_kB)
61      integr%buf_kA = integr%buf_kA+integr%buf_kB
62      res=y+integr%buf_kA*integr%dt/6
63      t=t+integr%dt
```

6.12 stat Module Reference

Statistics accumulators.

Data Types

- type statacccumulator

    Statistics accumulator objects class.

Functions/Subroutines

- subroutine init_stat (istat, ndim)

    Initialize the accumulators.
- subroutine acc (istat, x)

    Accumulate one state.
- real(kind=8) function, dimension(size(istat%m)) mean (istat)

    Function returning the mean.
- real(kind=8) function, dimension(size(istat%m)) var (istat)

    Function returning the variance.
- integer function iter (istat)

    Function returning the number of data accumulated.
- subroutine reset (istat)

    Routine resetting the accumulator.
- subroutine clean (istat)

    Routine to clean the accumulator.

6.12.1 Detailed Description

Statistics accumulators.

Copyright

6.12.2 Function/Subroutine Documentation

6.12.2.1 acc()

```
subroutine stat::acc (
            class(stataccumulator), intent(inout) istat,
            real(kind=8), dimension(:), intent(in) x )  [private]
```

Accumulate one state.

Parameters

in,out	*istat*	Statistical accumulator to initialize
in	*x*	State to accumulate

Definition at line 69 of file stat.f90.

```
      CLASS(stataccumulator), INTENT(INOUT) :: istat
      REAL(KIND=8), DIMENSION(:), INTENT(IN) :: x
      istat%i=istat%i+1
      istat%mprev=istat%m+(x-istat%m)/istat%i
      istat%mtmp=istat%mprev
      istat%mprev=istat%m
      istat%m=istat%mtmp
      istat%v=istat%v+(x-istat%mprev)*(x-istat%m)
```

6.12.2.2 clean()

```
subroutine stat::clean (
              class(stataccumulator), intent(inout) istat )  [private]
```

Routine to clean the accumulator.

Parameters

in,out	*istat*	Statistical accumulator to clean

Definition at line 119 of file stat.f90.

```
      CLASS(stataccumulator), INTENT(INOUT) :: istat

      IF (allocated(istat%m)) DEALLOCATE(istat%m)
      IF (allocated(istat%mprev)) DEALLOCATE(istat%mprev)
      IF (allocated(istat%v)) DEALLOCATE(istat%v)
      IF (allocated(istat%mtmp)) DEALLOCATE(istat%mtmp)

```

6.12.2.3 init_stat()

```
subroutine stat::init_stat (
              class(stataccumulator), intent(inout) istat,
              integer, intent(in) ndim )
```

Initialize the accumulators.

Parameters

in,out	*istat*	Statistical accumulator to initialize
in	*ndim*	Dimension of the state space to accumulate statistics for.

Definition at line 44 of file stat.f90.

```
      CLASS(stataccumulator), INTENT(INOUT) :: istat
      INTEGER, INTENT(in) :: ndim
```

```
INTEGER :: allocstat

ALLOCATE(istat%m(ndim), istat%mprev(ndim), stat=allocstat)
IF (allocstat /= 0) THEN
  print*, "*** init_stat: Problem with allocation! ***"
  stop "Exiting ..."
END IF
ALLOCATE(istat%v(ndim), istat%mtmp(ndim), stat=allocstat)
IF (allocstat /= 0) THEN
  print*, "*** init_stat: Problem with allocation! ***"
  stop "Exiting ..."
END IF
istat%m=0.d0
istat%mprev=0.d0
istat%v=0.d0
istat%mtmp=0.d0
```

6.12.2.4 iter()

```
integer function stat::iter (
              class(stataccumulator), intent(in) istat )  [private]
```

Function returning the number of data accumulated.

Parameters

in,out	*istat*	Statistical accumulator to initialize

Returns

The number of the accumulated states

Definition at line 101 of file stat.f90.

```
CLASS(stataccumulator), INTENT(IN) :: istat
INTEGER :: iter
iter=istat%i
```

6.12.2.5 mean()

```
real(kind=8) function, dimension(size(istat%m)) stat::mean (
              class(stataccumulator), intent(in) istat )  [private]
```

Function returning the mean.

Parameters

in,out	*istat*	Statistical accumulator to initialize

Returns

The mean of the accumulated states

Definition at line 83 of file stat.f90.

```
83        CLASS(stataccumulator), INTENT(IN) :: istat
84        REAL(KIND=8), DIMENSION(size(istat%m)) :: mean
85        mean=istat%m
```

6.12.2.6 reset()

```
subroutine stat::reset (
            class(stataccumulator), intent(inout) istat )  [private]
```

Routine resetting the accumulator.

Parameters

in,out	*istat*	Statistical accumulator to initialize

Definition at line 109 of file stat.f90.

```
109        CLASS(stataccumulator), INTENT(INOUT) :: istat
110        istat%m=0.d0
111        istat%mprev=0.d0
112        istat%v=0.d0
113        istat%i=0
```

6.12.2.7 var()

```
real(kind=8) function, dimension(size(istat%m)) stat::var (
            class(stataccumulator), intent(in) istat )  [private]
```

Function returning the variance.

Parameters

in,out	*istat*	Statistical accumulator to initialize

Returns

The variance of the accumulated states

Definition at line 92 of file stat.f90.

```
92        CLASS(stataccumulator), INTENT(IN) :: istat
93        REAL(KIND=8), DIMENSION(size(istat%m)) :: var
94        var=istat%v/(istat%i-1)
```

6.13 tensor_def Module Reference

Tensor utility module. Contains class to represent sparse tensors.

Data Types

- type coolist

  Coordinate list. Type used to represent the sparse tensor.
- type coolistelem

  Coordinate list element type. Elementary elements of the sparse tensors.
- type tensor

  General class to represent a sparse tensor.

Functions/Subroutines

- logical function test_alloc (mtensor)

  Function to test if the tensor is allocated.
- logical function empty (mtensor)

  Function to test if the tensor is empty.
- subroutine clean (mtensor)

  Routine to clean (deallocate) a tensor.
- subroutine init (mtensor, ndim)

  Routine to initialize a tensor.
- integer function tensor_size (mtensor)
- subroutine copy (src, dst)

  Routine to copy a tensor into another one.
- subroutine from_mat (src, dst)

  Routine to convert a matrix to a tensor, using only the fist two indices of the rank-3 tensor.
- subroutine sparse_mul3 (mtensor, arr_j, arr_k, res)

  Sparse multiplication of a tensor with two vectors: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k}\, a_j\, b_k$.
- subroutine simplify (mtensor)

  Routine to simplify a coolist (sparse tensor). For each index i, it upper triangularize the matrix

  $$\mathcal{T}_{i,j,k} \qquad 0 \leq j,k \leq ndim.$$

  .
- subroutine jsparse_mul (mtensor, arr_j, jtensor)

  Sparse multiplication of two tensors to determine the Jacobian:

  $$J_{i,j} = \sum_{k=0}^{ndim} \left(\mathcal{T}_{i,j,k} + \mathcal{T}_{i,k,j} \right)\, a_k.$$

  It's implemented slightly differently: for every $\mathcal{T}_{i,j,k}$, we add to $J_{i,j}$ as follows:

  $$J_{i,j} = J_{i,j} + \mathcal{T}_{i,j,k}\, a_k J_{i,k} = J_{i,k} + \mathcal{T}_{i,j,k}\, a_j$$

  This version return a sparse tensor.
- subroutine jsparse_mul_mat (mtensor, arr_j, jmatrix)

Sparse multiplication of two tensors to determine the Jacobian:

$$J_{i,j} = \sum_{k=0}^{ndim} \left(\mathcal{T}_{i,j,k} + \mathcal{T}_{i,k,j} \right) a_k .$$

It's implemented slightly differently: for every $\mathcal{T}_{i,j,k}$, we add to $J_{i,j}$ as follows:

$$J_{i,j} = J_{i,j} + \mathcal{T}_{i,j,k}\, a_k J_{i,k} = J_{i,k} + \mathcal{T}_{i,j,k}\, a_j$$

This version return a matrix.

- subroutine sparse_mul2 (mtensor, arr_j, res)

  Sparse multiplication of a 2d sparse tensor with a vector: $\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k}\, a_j .$
- subroutine add_elem (mtensor, i, j, k, v)

  Subroutine to add element to a coolist.
- subroutine add_from_tensor (src, dst)

  Routine to add the entries of a rank-3 tensor to another one.
- subroutine print_tensor (mtensor, s)

  Routine to print a rank-3 tensor.
- subroutine write_tensor_to_file (mtensor, s)

  Write a rank-3 tensor coolist to a file.
- subroutine load_tensor_from_file (mtensor, s)

  Load a rank-3 tensor coolist from a file definition.

6.13.1 Detailed Description

Tensor utility module. Contains class to represent sparse tensors.

Copyright

2015-2020 Lesley De Cruz & Jonathan Demaeyer. See LICENSE.txt for license information.

6.13.2 Function/Subroutine Documentation

6.13.2.1 add_elem()

```
subroutine tensor_def::add_elem (
            class(tensor), intent(inout) mtensor,
            integer, intent(in) i,
            integer, intent(in) j,
            integer, intent(in) k,
            real(kind=8), intent(in) v ) [private]
```

Subroutine to add element to a coolist.

Parameters

in,out	*mtensor*	A tensor to add the element to.
in	*i*	tensor i index
in	*j*	tensor j index
in	*k*	tensor k index
in	*v*	value to add

Definition at line 377 of file tensor_def.f90.

```
    CLASS(tensor), INTENT(INOUT) :: mtensor
    INTEGER, INTENT(IN) :: i,j,k
    REAL(KIND=8), INTENT(IN) :: v
    INTEGER :: n
    IF (abs(v) .ge. real_eps) THEN
       n=(mtensor%t(i)%nelems)+1
       mtensor%t(i)%elems(n)%j=j
       mtensor%t(i)%elems(n)%k=k
       mtensor%t(i)%elems(n)%v=v
       mtensor%t(i)%nelems=n
    END IF
```

6.13.2.2 add_from_tensor()

```
subroutine tensor_def::add_from_tensor (
            class(tensor), intent(in) src,
            class(tensor), intent(inout) dst )  [private]
```

Routine to add the entries of a rank-3 tensor to another one.

Parameters

in	*src*	Tensor to add
in,out	*dst*	Destination tensor

Definition at line 417 of file tensor_def.f90.

```
    CLASS(tensor), INTENT(IN) :: src
    CLASS(tensor), INTENT(INOUT) :: dst
    TYPE(coolistelem), DIMENSION(:), ALLOCATABLE :: celems
    INTEGER :: i,j,n,allocstat

    DO i=1,dst%ndim()
       IF (src%t(i)%nelems/=0) THEN
          IF (dst%t(i)%nelems==0) THEN
             IF (allocated(dst%t(i)%elems)) THEN
                DEALLOCATE(dst%t(i)%elems, stat=allocstat)
                IF (allocstat /= 0) THEN
                  print*, "*** tensor%add_from_tensor: Problem with allocation! ***"
                  stop "Exiting ..."
                END IF
             END IF
             ALLOCATE(dst%t(i)%elems(src%t(i)%nelems), stat=allocstat)
             IF (allocstat /= 0) THEN
                  print*, "*** tensor%add_from_tensor: Problem with allocation! ***"
                  stop "Exiting ..."
             END IF
             n=0
          ELSE
             n=dst%t(i)%nelems
             ALLOCATE(celems(n), stat=allocstat)
             DO j=1,n
                celems(j)%j=dst%t(i)%elems(j)%j
                celems(j)%k=dst%t(i)%elems(j)%k
                celems(j)%v=dst%t(i)%elems(j)%v
             ENDDO
             IF (allocated(dst%t(i)%elems)) DEALLOCATE(dst%t(i)%elems, stat=allocstat)
             ALLOCATE(dst%t(i)%elems(src%t(i)%nelems+n), stat=allocstat)
             IF (allocstat /= 0) THEN
                  print*, "*** tensor%add_from_tensor: Problem with allocation! ***"
                  stop "Exiting ..."
             END IF
             DO j=1,n
                dst%t(i)%elems(j)%j=celems(j)%j
                dst%t(i)%elems(j)%k=celems(j)%k
```

```
                dst%t(i)%elems(j)%v=celems(j)%v
             ENDDO
             IF (allocated(celems)) DEALLOCATE(celems, stat=allocstat)
          ENDIF
          DO j=1,src%t(i)%nelems
             dst%t(i)%elems(n+j)%j=src%t(i)%elems(j)%j
             dst%t(i)%elems(n+j)%k=src%t(i)%elems(j)%k
             dst%t(i)%elems(n+j)%v=src%t(i)%elems(j)%v
          ENDDO
          dst%t(i)%nelems=src%t(i)%nelems+n
       ENDIF
    ENDDO
```

6.13.2.3 clean()

```
subroutine tensor_def::clean (
            class(tensor), intent(inout) mtensor )  [private]
```

Routine to clean (deallocate) a tensor.

Parameters

in,out	*mtensor*	The tensor to clean.

Definition at line 92 of file tensor_def.f90.

```
    CLASS(tensor), INTENT(INOUT) :: mtensor

    IF (mtensor%allocated()) DEALLOCATE(mtensor%t)
```

6.13.2.4 copy()

```
subroutine tensor_def::copy (
            class(tensor), intent(in) src,
            class(tensor), intent(out) dst )  [private]
```

Routine to copy a tensor into another one.

Parameters

in	*src*	Source tensor.
out	*dst*	Destination tensor.

Warning

The destination tensor will be reinitialized, erasing all previous content! Use with care...

Definition at line 130 of file tensor_def.f90.

```
    CLASS(tensor), INTENT(IN) :: src
    CLASS(tensor), INTENT(OUT) :: dst
    INTEGER :: i,j,allocstat

    CALL dst%init(src%ndim())
    DO i=1,src%ndim()
      ALLOCATE(dst%t(i)%elems(src%t(i)%nelems), stat=allocstat)
      IF (allocstat /= 0) THEN
        print*, "*** tensor%copy: Problem with allocation! ***"
        stop "Exiting ..."
      END IF
      DO j=1,src%t(i)%nelems
        dst%t(i)%elems(j)%j=src%t(i)%elems(j)%j
        dst%t(i)%elems(j)%k=src%t(i)%elems(j)%k
        dst%t(i)%elems(j)%v=src%t(i)%elems(j)%v
      ENDDO
      dst%t(i)%nelems=src%t(i)%nelems
    ENDDO
```

6.13.2.5 empty()

```
logical function tensor_def::empty (
            class(tensor) mtensor )  [private]
```

Function to test if the tensor is empty.

Parameters

mtensor	The tensor to test.

Returns

A boolean indicating if the tensor is empty.

Definition at line 75 of file tensor_def.f90.

```
75    CLASS(tensor) :: mtensor
76    LOGICAL :: empty
77    INTEGER :: i
78
79    empty = .true.
80
81    IF (.NOT. mtensor%allocated()) RETURN
82
83    DO i=1,mtensor%ndim()
84      IF (mtensor%t(i)%nelems/=0) empty = .false.
85    END DO
86
```

6.13.2.6 from_mat()

```
subroutine tensor_def::from_mat (
            real(kind=8), dimension(:,:), intent(in) src,
            class(tensor), intent(inout) dst )  [private]
```

Routine to convert a matrix to a tensor, using only the fist two indices of the rank-3 tensor.

Parameters

in	*src*	Source matrix
out	*dst*	Destination tensor.

Warning

The destination tensor will be reinitialized, erasing all previous content! Use with care...

Definition at line 155 of file tensor_def.f90.

```
155     CLASS(tensor), INTENT(INOUT) :: dst
156     REAL(KIND=8), DIMENSION(:,:), INTENT(IN) :: src
157     INTEGER :: i,j,n,allocstat
158     INTEGER :: ndim
159     INTEGER, DIMENSION(2) :: sh
160
161     sh = shape(src)
162     ndim = sh(1)
163     CALL dst%init(ndim)
164
165     DO i=1,ndim
166       n=0
167       DO j=1,ndim
168         IF (abs(src(i,j))>real_eps) n=n+1
169       ENDDO
170       ALLOCATE(dst%t(i)%elems(n), stat=allocstat)
171       IF (allocstat /= 0) THEN
172         print*, "*** tensor%from_mat: Problem with allocation! ***"
173         stop "Exiting ..."
174       END IF
175       n=0
176       DO j=1,ndim
177         IF (abs(src(i,j))>real_eps) THEN
178           n=n+1
179           dst%t(i)%elems(n)%j=j
180           dst%t(i)%elems(n)%k=0
181           dst%t(i)%elems(n)%v=src(i,j)
182         ENDIF
183       ENDDO
184       dst%t(i)%nelems=n
185     ENDDO
```

6.13.2.7 init()

```
subroutine tensor_def::init (
            class(tensor), intent(inout) mtensor,
            integer, intent(in) ndim )  [private]
```

Routine to initialize a tensor.

Parameters

in,out	*mtensor*	The tensor to clean.
in	*ndim*	The first dimension of the tensor.

Definition at line 102 of file tensor_def.f90.

```
102     CLASS(tensor), INTENT(INOUT) :: mtensor
```

```
    INTEGER, INTENT(IN) :: ndim
    INTEGER :: allocstat

    CALL mtensor%clean
    ALLOCATE(mtensor%t(ndim), stat=allocstat)
    IF (allocstat /= 0) THEN
        print*, "*** tensor%init: Problem with allocation! ***"
        stop "Exiting ..."
    END IF

```

6.13.2.8 jsparse_mul()

```
subroutine tensor_def::jsparse_mul (
            class(tensor), intent(in) mtensor,
            real(kind=8), dimension(0:size(mtensor%t)), intent(in) arr_j,
            type(tensor), intent(inout) jtensor )  [private]
```

Sparse multiplication of two tensors to determine the Jacobian:

$$J_{i,j} = \sum_{k=0}^{ndim} \left(\mathcal{T}_{i,j,k} + \mathcal{T}_{i,k,j}\right) a_k.$$

It's implemented slightly differently: for every $\mathcal{T}_{i,j,k}$, we add to $J_{i,j}$ as follows:

$$J_{i,j} = J_{i,j} + \mathcal{T}_{i,j,k}\, a_k J_{i,k} = J_{i,k} + \mathcal{T}_{i,j,k}\, a_j$$

This version return a sparse tensor.

Parameters

in	*mtensor*	A sparse tensor of which index 2 or 3 will be contracted.
in	*arr_j*	The vector $\boldsymbol{a}$ to be contracted with.
out	*jtensor*	A sparse tensor to store the result of the contraction

Warning

The output jtensor will be reinitialized, erasing all previous content! Use with care...

Definition at line 288 of file tensor_def.f90.

```
288     CLASS(tensor), INTENT(IN) :: mtensor
289     TYPE(tensor), INTENT(INOUT):: jtensor
290     REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(IN)  :: arr_j
291     REAL(KIND=8) :: v
292     INTEGER :: i,j,k,n,nj,allocstat
293     CALL jtensor%init(mtensor%ndim())
294     DO i=1,mtensor%ndim()
295        nj=2*jtensor%t(i)%nelems
296        ALLOCATE(jtensor%t(i)%elems(nj), stat=allocstat)
297        IF (allocstat /= 0) THEN
298          print*, "*** tensor%jsparse_mul: Problem with allocation! ***"
299          stop "Exiting ..."
300        END IF
301        nj=0
302        DO n=1,mtensor%t(i)%nelems
303           j=mtensor%t(i)%elems(n)%j
304           k=mtensor%t(i)%elems(n)%k
305           v=mtensor%t(i)%elems(n)%v
306           IF (j /=0) THEN
```

```
            nj=nj+1
            jtensor%t(i)%elems(nj)%j=j
            jtensor%t(i)%elems(nj)%k=0
            jtensor%t(i)%elems(nj)%v=v*arr_j(k)
         END IF

         IF (k /=0) THEN
            nj=nj+1
            jtensor%t(i)%elems(nj)%j=k
            jtensor%t(i)%elems(nj)%k=0
            jtensor%t(i)%elems(nj)%v=v*arr_j(j)
         END IF
      END DO
      jtensor%t(i)%nelems=nj
   END DO
```

6.13.2.9 jsparse_mul_mat()

```
subroutine tensor_def::jsparse_mul_mat (
            class(tensor), intent(in) mtensor,
            real(kind=8), dimension(0:size(mtensor%t)), intent(in) arr_j,
            real(kind=8), dimension(size(mtensor%t),size(mtensor%t)), intent(out) jmatrix )
[private]
```

Sparse multiplication of two tensors to determine the Jacobian:

$$J_{i,j} = \sum_{k=0}^{ndim} \left(\mathcal{T}_{i,j,k} + \mathcal{T}_{i,k,j} \right) a_k .$$

It's implemented slightly differently: for every $\mathcal{T}_{i,j,k}$, we add to $J_{i,j}$ as follows:

$$J_{i,j} = J_{i,j} + \mathcal{T}_{i,j,k} \, a_k J_{i,k} = J_{i,k} + \mathcal{T}_{i,j,k} \, a_j$$

This version return a matrix.

Parameters

in	*mtensor*	A sparse tensor of which index 2 or 3 will be contracted.
in	*arr_j*	The vector $\boldsymbol{a}$ to be contracted with.
out	*jmatrix*	A matrix to store the result of the contraction.

Definition at line 333 of file tensor_def.f90.

```
333    CLASS(tensor), INTENT(IN) :: mtensor
334    REAL(KIND=8), DIMENSION(size(mtensor%t),size(mtensor%t)), INTENT(OUT):: jmatrix
335    REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(IN)  :: arr_j
336    REAL(KIND=8) :: v
337    INTEGER :: i,j,k,n
338    jmatrix=0.d0
339    DO i=1,mtensor%ndim()
340       DO n=1,mtensor%t(i)%nelems
341          j=mtensor%t(i)%elems(n)%j
342          k=mtensor%t(i)%elems(n)%k
343          v=mtensor%t(i)%elems(n)%v
344          IF (j /=0) jmatrix(i,j)=jmatrix(i,j)+v*arr_j(k)
345          IF (k /=0) jmatrix(i,k)=jmatrix(i,k)+v*arr_j(j)
346       END DO
347    END DO
```

6.13.2.10 load_tensor_from_file()

```
subroutine tensor_def::load_tensor_from_file (
            class(tensor), intent(inout) mtensor,
            character (len=*), intent(in) s )  [private]
```

Load a rank-3 tensor coolist from a file definition.

Parameters

in,out	*mtensor*	The tensor to load to.
in	*s*	Filename of the tensor definition file.

Remarks

The destination tensor have to be an empty tensor, i.e. with unallocated list of elements and nelems set to 0.

Definition at line 522 of file tensor_def.f90.

```
522     CLASS(tensor), INTENT(INOUT) :: mtensor
523     CHARACTER (LEN=*), INTENT(IN) :: s
524     INTEGER :: i,ir,j,k,n,allocstat, ndim
525     REAL(KIND=8) :: v
526     OPEN(30,file=s,status='old')
527     READ(30, *) ndim
528     ALLOCATE(mtensor%t(ndim), stat=allocstat)
529     IF (allocstat /= 0) THEN
530       print*, "*** tensor%load_tensor_from_file: Problem with allocation! ***"
531       stop "Exiting ..."
532     END IF
533     DO i=1,ndim
534        READ(30,*) ir,n
535        IF (n /= 0) THEN
536           ALLOCATE(mtensor%t(i)%elems(n), stat=allocstat)
537           IF (allocstat /= 0) THEN
538             print*, "*** tensor%load_tensor_from_file: Problem with allocation! ***"
539             stop "Exiting ..."
540           END IF
541           mtensor%t(i)%nelems=n
542        ENDIF
543        DO n=1,mtensor%t(i)%nelems
544           READ(30,*) ir,j,k,v
545           mtensor%t(i)%elems(n)%j=j
546           mtensor%t(i)%elems(n)%k=k
547           mtensor%t(i)%elems(n)%v=v
548        ENDDO
549     END DO
550     CLOSE(30)
```

6.13.2.11 print_tensor()

```
subroutine tensor_def::print_tensor (
            class(tensor), intent(in) mtensor,
            character(len=*), intent(in), optional, target s )  [private]
```

Routine to print a rank-3 tensor.

Parameters

in	*mtensor*	Tensor to print.
in	*s*	String to put before tensor entries. Default to "t".

Definition at line 474 of file tensor_def.f90.

```
474     CLASS(tensor), INTENT(IN) :: mtensor
475     CHARACTER(LEN=*), INTENT(IN), TARGET, OPTIONAL :: s
476
477     CHARACTER, TARGET :: sr = "t"
478     CHARACTER, POINTER :: r
479     INTEGER :: i,n,j,k
480     IF (present(s)) THEN
481        r => s
482     ELSE
483        r => sr
484     END IF
485     DO i=1,mtensor%ndim()
486        DO n=1,mtensor%t(i)%nelems
487           j=mtensor%t(i)%elems(n)%j
488           k=mtensor%t(i)%elems(n)%k
489           IF( abs(mtensor%t(i)%elems(n)%v) .GE. real_eps) THEN
490              write(*,"(A,ES12.5)") r//"["//trim(str(i))//"]["//trim(str(j)) &
491                   &//"]["//trim(str(k))//"] = ",mtensor%t(i)%elems(n)%v
492           END IF
493        END DO
494     END DO
```

6.13.2.12 simplify()

```
subroutine tensor_def::simplify (
              class(tensor), intent(inout) mtensor )   [private]
```

Routine to simplify a coolist (sparse tensor). For each index i, it upper triangularize the matrix

$$\mathcal{T}_{i,j,k} \qquad 0 \leq j,k \leq ndim.$$

.

Parameters

in,out	*mtensor*	A sparse tensor which will be simplified.

Definition at line 214 of file tensor_def.f90.

```
214     CLASS(tensor), INTENT(INOUT) :: mtensor
215     INTEGER :: i,j,k
216     INTEGER :: li,lii,liii,n
217     DO i= 1,mtensor%ndim()
218       n=mtensor%t(i)%nelems
219       DO li=n,2,-1
220         j=mtensor%t(i)%elems(li)%j
221         k=mtensor%t(i)%elems(li)%k
222         DO lii=li-1,1,-1
223           IF (((j==mtensor%t(i)%elems(lii)%j).AND.(k==mtensor%t(i)&
224             &%elems(lii)%k)).OR.((j==mtensor%t(i)%elems(lii)%k).AND.(k==mtensor%t(i)%elems(lii)%j))) THEN
225             ! Found another entry with the same i,j,k: merge both into
226             ! the one listed first (of those two).
227             mtensor%t(i)%elems(lii)%v=mtensor%t(i)%elems(lii)%v+mtensor%t(i)%elems(li)%v
228             IF (j>k) THEN
229               mtensor%t(i)%elems(lii)%j=mtensor%t(i)%elems(li)%k
230               mtensor%t(i)%elems(lii)%k=mtensor%t(i)%elems(li)%j
231             ENDIF
232
233             ! Shift the rest of the items one place down.
234             DO liii=li+1,n
235               mtensor%t(i)%elems(liii-1)%j=mtensor%t(i)%elems(liii)%j
236               mtensor%t(i)%elems(liii-1)%k=mtensor%t(i)%elems(liii)%k
237               mtensor%t(i)%elems(liii-1)%v=mtensor%t(i)%elems(liii)%v
238             END DO
239             mtensor%t(i)%nelems=mtensor%t(i)%nelems-1
```

```
            ! Here we should stop because the li no longer points to the
            ! original i,j,k element
            EXIT
          ENDIF
        ENDDO
      ENDDO
      n=mtensor%t(i)%nelems
      li=1
      DO WHILE (li<=mtensor%t(i)%nelems)
        ! Clear new "almost" zero entries and shift rest of the items one place down.
        ! Make sure not to skip any entries while shifting!
        DO WHILE (abs(mtensor%t(i)%elems(li)%v) < real_eps)
          DO liii=li+1,n
            mtensor%t(i)%elems(liii-1)%j=mtensor%t(i)%elems(liii)%j
            mtensor%t(i)%elems(liii-1)%k=mtensor%t(i)%elems(liii)%k
            mtensor%t(i)%elems(liii-1)%v=mtensor%t(i)%elems(liii)%v
          ENDDO
          mtensor%t(i)%nelems=mtensor%t(i)%nelems-1
          if (li > mtensor%t(i)%nelems) THEN
            EXIT
          ENDIF
        ENDDO
        li=li+1
      ENDDO

      n=mtensor%t(i)%nelems
      DO li=1,n
        ! Upper triangularize
        j=mtensor%t(i)%elems(li)%j
        k=mtensor%t(i)%elems(li)%k
        IF (j>k) THEN
          mtensor%t(i)%elems(li)%j=k
          mtensor%t(i)%elems(li)%k=j
        ENDIF
      ENDDO
    ENDDO
```

6.13.2.13 sparse_mul2()

```
subroutine tensor_def::sparse_mul2 (
            class(tensor), intent(in) mtensor,
            real(kind=8), dimension(0:size(mtensor%t)), intent(in) arr_j,
            real(kind=8), dimension(0:size(mtensor%t)), intent(out) res )  [private]
```

Sparse multiplication of a 2d sparse tensor with a vector: $\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k}\, a_j$.

Parameters

in	*mtensor*	A sparse tensor of which index 2 will be contracted.
in	*arr_j*	The vector $\boldsymbol{a}$ to be contracted with.
out	*res*	vector (buffer) to store the result of the contraction

Remarks

Note that it is NOT safe to pass `arr_j` as a result buffer, as this operation does multiple passes.

Definition at line 357 of file tensor_def.f90.

```
357     CLASS(tensor), INTENT(IN) :: mtensor
358     REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(IN)  :: arr_j
359     REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(OUT) :: res
```

```
INTEGER :: i,j,n
res=0.d0
DO i=1,mtensor%ndim()
  DO n=1,mtensor%t(i)%nelems
     j=mtensor%t(i)%elems(n)%j
     res(i) = res(i) + mtensor%t(i)%elems(n)%v * arr_j(j)
  END DO
END DO
```

6.13.2.14 sparse_mul3()

```
subroutine tensor_def::sparse_mul3 (
            class(tensor), intent(in) mtensor,
            real(kind=8), dimension(0:size(mtensor%t)), intent(in) arr_j,
            real(kind=8), dimension(0:size(mtensor%t)), intent(in) arr_k,
            real(kind=8), dimension(0:size(mtensor%t)), intent(out) res )  [private]
```

Sparse multiplication of a tensor with two vectors: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} \, a_j \, b_k$.

Parameters

in	*mtensor*	A sparse tensor of which index 2 and 3 will be contracted.
in	*arr_j*	The vector $\boldsymbol{a}$ to be contracted with index 2 of the tensor.
in	*arr_k*	The vector $\boldsymbol{b}$ to be contracted with index 3 of the tensor.
out	*res*	Vector to store the result of the contraction.

Remarks

Note that it is NOT safe to pass `arr_j` or `arr_k` as a result buffer, as this operation does multiple passes. However, passsing the same vector as `arr_j` and `arr_k` is safe.

Definition at line 196 of file tensor_def.f90.

```
CLASS(tensor), INTENT(IN) :: mtensor
REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(IN)  :: arr_j, arr_k
REAL(KIND=8), DIMENSION(0:size(mtensor%t)), INTENT(OUT) :: res
INTEGER :: i,j,k,n
res=0.d0
DO i=1,mtensor%ndim()
  DO n=1,mtensor%t(i)%nelems
    j=mtensor%t(i)%elems(n)%j
    k=mtensor%t(i)%elems(n)%k
    res(i) = res(i) + mtensor%t(i)%elems(n)%v * arr_j(j)*arr_k(k)
  END DO
END DO
```

6.13.2.15 tensor_size()

```
integer function tensor_def::tensor_size (
            class(tensor) mtensor )  [private]
```

Parameters

mtensor	The tensor to return the size of.

Returns

The size of the tensor

Definition at line 119 of file tensor_def.f90.

```
119     CLASS(tensor) :: mtensor
120     INTEGER :: ndim
121     ndim = size(mtensor%t)
```

6.13.2.16 test_alloc()

```
logical function tensor_def::test_alloc (
            class(tensor) mtensor )
```

Function to test if the tensor is allocated.

Parameters

mtensor	The tensor to test.

Returns

A boolean indicating if the tensor is allocated.

Definition at line 64 of file tensor_def.f90.

```
64     CLASS(tensor) :: mtensor
65     LOGICAL :: test_alloc
66
67     test_alloc = allocated(mtensor%t)
68
```

6.13.2.17 write_tensor_to_file()

```
subroutine tensor_def::write_tensor_to_file (
            class(tensor), intent(in) mtensor,
            character (len=*), intent(in) s )  [private]
```

Write a rank-3 tensor coolist to a file.

Parameters

	mtensor	The tensor to write
in	*s*	Filename

Definition at line 501 of file tensor_def.f90.

```
501     CLASS(tensor), INTENT(IN) :: mtensor
502     CHARACTER (LEN=*), INTENT(IN) :: s
503     INTEGER :: i,j,k,n
504     OPEN(30,file=s)
505     WRITE(30,*) mtensor%ndim()
506     DO i=1,mtensor%ndim()
507        WRITE(30,*) i,mtensor%t(i)%nelems
508        DO n=1,mtensor%t(i)%nelems
509           j=mtensor%t(i)%elems(n)%j
510           k=mtensor%t(i)%elems(n)%k
511           WRITE(30,*) i,j,k,mtensor%t(i)%elems(n)%v
512        END DO
513     END DO
514     CLOSE(30)
```

6.14 tl_ad_tensor Module Reference

Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Tensors definition module.

Data Types

- type adtensor

  Tensor representation of the Adjoint tendencies.
- type tltensor

  Tensor representation of the Tangent Linear tendencies.

Functions/Subroutines

- subroutine delete_tensor (tens)

  Subroutine to clean a TL tensor.
- subroutine init_tltensor (tens, aot)

  Subroutine to initialise the TL tensor.
- subroutine init_adtensor (tens, aot)

  Subroutine to initialise the AD tensor.

6.14.1 Detailed Description

Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Tensors definition module.

Copyright

6.14.2 Function/Subroutine Documentation

6.14.2.1 delete_tensor()

```
subroutine tl_ad_tensor::delete_tensor (
            class(tltensor), intent(inout) tens )
```

Subroutine to clean a TL tensor.

Parameters

in,out	*tens*	The tensor to clean.

Definition at line 64 of file tl_ad_tensor.f90.

```
CLASS(tltensor), INTENT(INOUT) :: tens

IF (allocated(tens%count_elems)) DEALLOCATE(tens%count_elems)

CALL tens%tensor%clean

tens%initialized = .false.

```

6.14.2.2 init_adtensor()

```
subroutine tl_ad_tensor::init_adtensor (
            class(adtensor), intent(inout) tens,
            class(atmoctensor), intent(in), target aot )  [private]
```

Subroutine to initialise the AD tensor.

Parameters

in,out	*tens*	The tensor to clean.
in	*aot*	A Atmosphere-Ocean tensor to initialize the AD tensor with.

Definition at line 204 of file tl_ad_tensor.f90.

```
CLASS(adtensor), INTENT(INOUT) :: tens
CLASS(atmoctensor), INTENT(IN), TARGET :: aot

INTEGER :: i
INTEGER, POINTER :: ndim
INTEGER :: allocstat

IF (.NOT.aot%initialized) THEN
  print*, 'Provided AO tensor not yet initialized, impossible to initialize AD tensor!'
  RETURN
END IF

ndim => aot%ndim

ALLOCATE(tens%count_elems(ndim), stat=allocstat)
IF (allocstat /= 0) THEN
    print*, "*** init_adtensor: Problem with allocation! ***"
    stop "Exiting ..."
END IF
tens%count_elems=0

CALL tens%tensor%init(ndim)

tens%ad_operation => ad_add_count
CALL tens%compute_tensor(aot)

DO i=1,ndim
  ALLOCATE(tens%tensor%t(i)%elems(tens%count_elems(i)), stat=allocstat)
  IF (allocstat /= 0) THEN
    print*, "*** init_adtensor: Problem with allocation! ***"
    stop "Exiting ..."
  END IF
END DO
```

```
    tens%ad_operation => ad_coeff
    CALL tens%compute_tensor(aot)

    CALL tens%tensor%simplify

    tens%initialized = .true.

```

6.14.2.3 init_tltensor()

subroutine tl_ad_tensor::init_tltensor (
class(tltensor), intent(inout) *tens*,
class(atmoctensor), intent(in), target *aot*) [private]

Subroutine to initialise the TL tensor.

Parameters

in,out	*tens*	The tensor to clean.
in	*aot*	A Atmosphere-Ocean tensor to initialize the TL tensor with.

Definition at line 85 of file tl_ad_tensor.f90.

```
   CLASS(tltensor), INTENT(INOUT) :: tens
   CLASS(atmoctensor), INTENT(IN), TARGET :: aot

   INTEGER :: i
   INTEGER, POINTER :: ndim
   INTEGER :: allocstat

   IF (.NOT.aot%initialized) THEN
     print*, 'Provided AO tensor not yet initialized, impossible to initialize TL tensor!'
     RETURN
   END IF

   ndim => aot%ndim

   ALLOCATE(tens%count_elems(ndim), stat=allocstat)
    IF (allocstat /= 0) THEN
        print*, "*** init_tltensor: Problem with allocation! ***"
        stop "Exiting ..."
    END IF
    tens%count_elems=0

    CALL tens%tensor%init(ndim)

    tens%tl_operation => tl_add_count
    CALL tens%compute_tensor(aot)

    DO i=1,ndim
      ALLOCATE(tens%tensor%t(i)%elems(tens%count_elems(i)), stat=allocstat)
      IF (allocstat /= 0) THEN
        print*, "*** init_tltensor: Problem with allocation! ***"
        stop "Exiting ..."
      END IF
    END DO

    tens%tl_operation => tl_coeff
    CALL tens%compute_tensor(aot)

    CALL tens%tensor%simplify

    tens%initialized = .true.

```

6.15 util Module Reference

Utility module.

Functions/Subroutines

- character(len=20) function, public str (k)

  Convert an integer to string.
- character(len=40) function, public rstr (x, fm)

  Convert a real to string with a given format.
- integer function, dimension(size(s)), public isin (c, s)

  Determine if a character is in a string and where.
- subroutine, public init_random_seed ()

  Random generator initialization routine.
- subroutine, public piksrt (k, arr, par)

  Simple card player sorting function.
- subroutine, public init_one (A)

  Initialize a square matrix A as a unit matrix.

6.15.1 Detailed Description

Utility module.

Copyright

6.15.2 Function/Subroutine Documentation

6.15.2.1 isin()

```
integer function, dimension(size(s)), public util::isin (
            character, intent(in) c,
            character, dimension(:), intent(in) s )
```

Determine if a character is in a string and where.

Remarks

: return positions in a vector if found and 0 vector if not found

Definition at line 45 of file util.f90.

```
45      CHARACTER, INTENT(IN) :: c
46      CHARACTER, DIMENSION(:), INTENT(IN) :: s
47      INTEGER, DIMENSION(size(s)) :: isin
48      INTEGER :: i,j
49
50      isin=0
51      j=0
52      DO i=size(s),1,-1
53         IF (c==s(i)) THEN
54            j=j+1
55            isin(j)=i
56         END IF
57      END DO
```

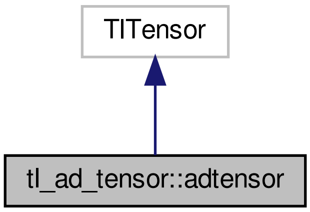
Chapter 7

Data Type Documentation

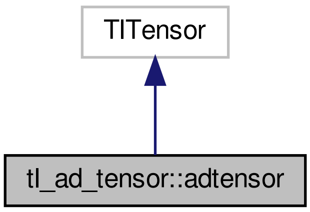
7.1 tl_ad_tensor::adtensor Type Reference

Tensor representation of the Adjoint tendencies.

Inheritance diagram for tl_ad_tensor::adtensor:



Collaboration diagram for tl_ad_tensor::adtensor:



7.1.1 Detailed Description

Tensor representation of the Adjoint tendencies.

Definition at line 46 of file tl_ad_tensor.f90.

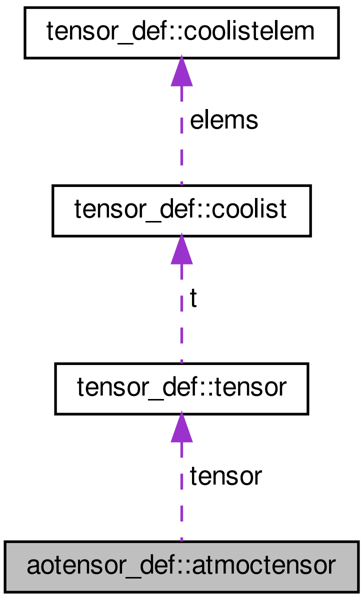
The documentation for this type was generated from the following file:

- tl_ad_tensor.f90

7.2 aotensor_def::atmoctensor Type Reference

Class to hold the tensor $\mathcal{T}_{i,j,k}$ representation of the tendencies.

Collaboration diagram for aotensor_def::atmoctensor:



Public Attributes

- type(tensor) tensor

  The tensor object.
- integer, dimension(:), allocatable count_elems

  A list of the number of non-zero entries of the tensor component along i.

7.2.1 Detailed Description

Class to hold the tensor $\mathcal{T}_{i,j,k}$ representation of the tendencies.

Definition at line 37 of file aotensor_def.f90.

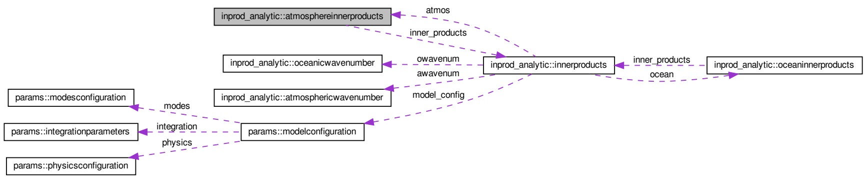
The documentation for this type was generated from the following file:

- aotensor_def.f90

7.3 inprod_analytic::atmosphereinnerproducts Type Reference

Class holding the atmospheric inner products functions.

Collaboration diagram for inprod_analytic::atmosphereinnerproducts:



Private Member Functions

- PROCEDURE a => calculate_a

  Eigenvalues of the Laplacian (atmospheric)
- PROCEDURE b => calculate_b

  Streamfunction advection terms (atmospheric)
- PROCEDURE c => calculate_c_atm

  Beta term for the atmosphere.
- PROCEDURE d => calculate_d

  Forcing of the ocean on the atmosphere.
- PROCEDURE g => calculate_g

  Temperature advection terms (atmospheric)
- PROCEDURE s => calculate_s

  Forcing (thermal) of the ocean on the atmosphere.

Private Attributes

- type(innerproducts), pointer inner_products

  Pointer to a global inner products object.

7.3.1 Detailed Description

Class holding the atmospheric inner products functions.

Definition at line 51 of file inprod_analytic.f90.

The documentation for this type was generated from the following file:

- inprod_analytic.f90

7.4 inprod_analytic::atmosphericwavenumber Type Reference

Atmospheric bloc specification object.

7.4.1 Detailed Description

Atmospheric bloc specification object.

Definition at line 38 of file inprod_analytic.f90.

The documentation for this type was generated from the following file:

- inprod_analytic.f90

7.5 integrator_def::clean_int Interface Reference

Abstract interface for the procedure to clean the integrator objects.

7.5.1 Detailed Description

Abstract interface for the procedure to clean the integrator objects.

Parameters

`in,out`	*integr*	Integrator object to clean.

Definition at line 62 of file integrator_def.f90.

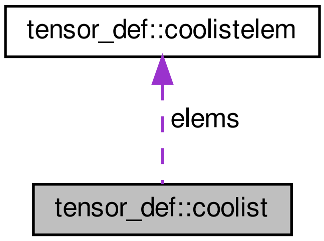
The documentation for this interface was generated from the following file:

- integrator_def.f90

7.6 tensor_def::coolist Type Reference

Coordinate list. Type used to represent the sparse tensor.

Collaboration diagram for tensor_def::coolist:



Private Attributes

- type(coolistelem), dimension(:), allocatable elems

    Lists of elements tensor_def::coolist_elem.
- integer nelems = 0

    Number of elements in the list.

7.6.1 Detailed Description

Coordinate list. Type used to represent the sparse tensor.

Definition at line 30 of file tensor_def.f90.

The documentation for this type was generated from the following file:

- tensor_def.f90

7.7 tensor_def::coolistelem Type Reference

Coordinate list element type. Elementary elements of the sparse tensors.

Private Attributes

- integer j

    Index j of the element.
- integer k

    Index k of the element.
- real(kind=8) v

    Value of the element.

7.7.1 Detailed Description

Coordinate list element type. Elementary elements of the sparse tensors.

Definition at line 23 of file tensor_def.f90.

The documentation for this type was generated from the following file:

- tensor_def.f90

7.8 integrator_def::init_int Interface Reference

Abstract interface for the procedures initializing the integrator objects.

7.8.1 Detailed Description

Abstract interface for the procedures initializing the integrator objects.

Parameters

`in,out`	*integr*	Integrator object to initialize.
`in`	*imodel*	Model object to initialize the integrator with.

Definition at line 37 of file integrator_def.f90.

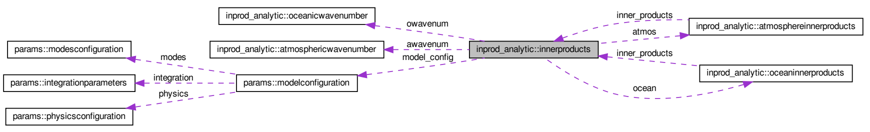
The documentation for this interface was generated from the following file:

- integrator_def.f90

7.9 inprod_analytic::innerproducts Type Reference

Global class for the inner products. Contains also the modes informations.

Collaboration diagram for inprod_analytic::innerproducts:



Public Member Functions

- PROCEDURE init => init_inner_products

    Procedure to initialize the inner products functions based on the modes configuration.
- PROCEDURE clean => delete_inner_products

    Procedure to clean the inner products object.

Public Attributes

- type(modelconfiguration), pointer model_config
  Pointer to a model configuration object.
- type(atmosphericwavenumber), dimension(:), allocatable, public awavenum
  Atmospheric blocs specification.
- type(oceanicwavenumber), dimension(:), allocatable, public owavenum
  Oceanic blocs specification.
- type(atmosphereinnerproducts), public atmos
  Atmospheric tensors.
- type(oceaninnerproducts), public ocean
  Oceanic tensors.

7.9.1 Detailed Description

Global class for the inner products. Contains also the modes informations.

Definition at line 75 of file inprod_analytic.f90.

The documentation for this type was generated from the following file:

- inprod_analytic.f90

7.10 params::integrationparameters Type Reference

The subclass containing the integration parameters.

Public Attributes

- real(kind=8) t_trans
  Transient time period.
- real(kind=8) t_run
  Effective intergration time (length of the generated trajectory)
- real(kind=8) dt
  Integration time step.
- real(kind=8) tw
  Write all variables every tw time units.
- real(kind=8) tw_snap
  Write a snapshot every tw_snap time units.
- logical writeout
  Write to file boolean.

7.10.1 Detailed Description

The subclass containing the integration parameters.

Definition at line 79 of file params.f90.

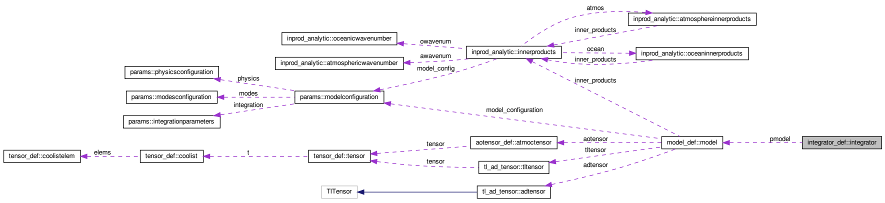
The documentation for this type was generated from the following file:

- params.f90

7.11 integrator_def::integrator Type Reference

Base class to be subclassed to create a new integrator.

Collaboration diagram for integrator_def::integrator:



Public Attributes

- type(model), pointer pmodel

    A pointer to the model to integrate.
- real(kind=8), pointer dt

    Time step of the integrator.
- integer, pointer ndim

    Dimension of the phase space of the model to integrate.

7.11.1 Detailed Description

Base class to be subclassed to create a new integrator.

Definition at line 23 of file integrator_def.f90.

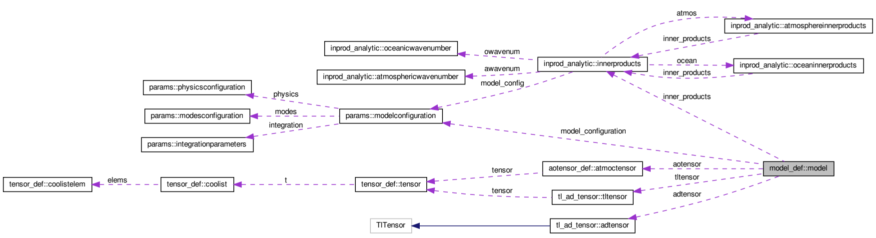
The documentation for this type was generated from the following file:

- integrator_def.f90

7.12 model_def::model Type Reference

Class to hold the components of a model version.

Collaboration diagram for model_def::model:



Public Attributes

- type(modelconfiguration) model_configuration

  Model configuration object of the model.
- type(innerproducts) inner_products

  Inner products object of the model.
- type(atmoctensor) aotensor

  Atmosphere-Ocean tendencies tensor of the model.
- type(tltensor) tltensor

  Tangent linear model tendencies.
- type(adtensor) adtensor

  Adjoint model tendencies.
- integer, pointer ndim

  Dimension of the phase space of the model to integrate.

7.12.1 Detailed Description

Class to hold the components of a model version.

Definition at line 25 of file model_def.f90.

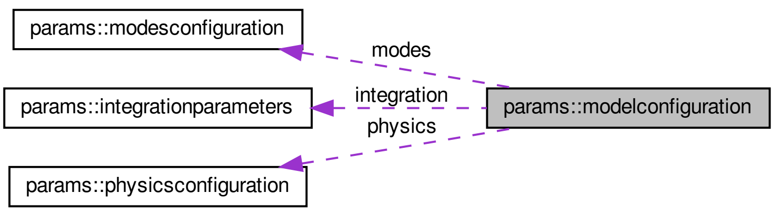
The documentation for this type was generated from the following file:

- model_def.f90

7.13 params::modelconfiguration Type Reference

The general class holding the model configuration.

Collaboration diagram for params::modelconfiguration:



7.13.1 Detailed Description

The general class holding the model configuration.

Definition at line 107 of file params.f90.

The documentation for this type was generated from the following file:

- params.f90

7.14 params::modesconfiguration Type Reference

The subclass containing the modes parameters.

Public Attributes

- integer nboc
  Number of atmospheric blocks.
- integer nbatm
  Number of oceanic blocks.
- integer, dimension(:,:), allocatable oms
  Ocean mode selection array.
- integer, dimension(:,:), allocatable ams
  Atmospheric mode selection array.
- integer natm =0
  Number of atmospheric basis functions.
- integer noc =0
  Number of oceanic basis functions.
- integer ndim
  Number of variables (dimension of the model)

7.14.1 Detailed Description

The subclass containing the modes parameters.

Definition at line 92 of file params.f90.

The documentation for this type was generated from the following file:

- params.f90

7.15 inprod_analytic::oceanicwavenumber Type Reference

Oceanic bloc specification object.

7.15.1 Detailed Description

Oceanic bloc specification object.

Definition at line 45 of file inprod_analytic.f90.

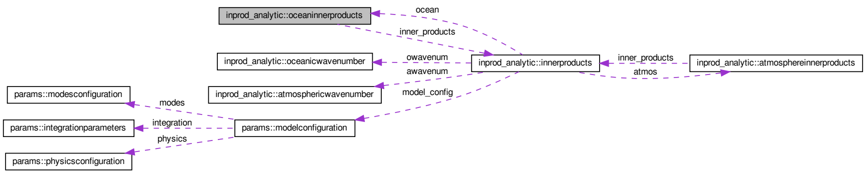
The documentation for this type was generated from the following file:

- inprod_analytic.f90

7.16 inprod_analytic::oceaninnerproducts Type Reference

Class holding the oceanic inner products functions.

Collaboration diagram for inprod_analytic::oceaninnerproducts:



Private Member Functions

- PROCEDURE k => calculate_K

  Forcing of the atmosphere on the ocean.
- PROCEDURE m => calculate_M

  Forcing of the ocean fields on the ocean.
- PROCEDURE c => calculate_C_oc

  Streamfunction advection terms (oceanic)
- PROCEDURE n => calculate_N

  Beta term for the ocean.
- PROCEDURE o => calculate_O

  Temperature advection term (passive scalar)
- PROCEDURE w => calculate_W

  Short-wave radiative forcing of the ocean.

Private Attributes

- type(innerproducts), pointer inner_products

  Pointer to a global inner products object.

7.16.1 Detailed Description

Class holding the oceanic inner products functions.

Definition at line 63 of file inprod_analytic.f90.

The documentation for this type was generated from the following file:

- inprod_analytic.f90

7.17 params::physicsconfiguration Type Reference

The subclass containing the physical parameters of the model.

Public Attributes

- real(kind=8) n

  $n = 2L_y/L_x$ *- Aspect ratio*
- real(kind=8) phi0

  Latitude in radian.
- real(kind=8) rra

  Earth radius.
- real(kind=8) sig0

  σ_0 *- Non-dimensional static stability of the atmosphere.*
- real(kind=8) k

  Bottom atmospheric friction coefficient.
- real(kind=8) kp

  k' *- Internal atmospheric friction coefficient.*
- real(kind=8) r

  Frictional coefficient at the bottom of the ocean.
- real(kind=8) d

  Merchanical coupling parameter between the ocean and the atmosphere.
- real(kind=8) f0

  f_0 *- Coriolis parameter*
- real(kind=8) gp

  g' *Reduced gravity*
- real(kind=8) h

  Depth of the active water layer of the ocean.
- real(kind=8) phi0_npi

  Latitude exprimed in fraction of pi.
- real(kind=8) lambda

  λ *- Sensible + turbulent heat exchange between the ocean and the atmosphere.*
- real(kind=8) co

  C_a *- Constant short-wave radiation of the ocean.*
- real(kind=8) go

  γ_o *- Specific heat capacity of the ocean.*
- real(kind=8) ca

  C_a *- Constant short-wave radiation of the atmosphere.*
- real(kind=8) to0

  T_o^0 *- Stationary solution for the 0-th order ocean temperature.*
- real(kind=8) ta0

  T_a^0 *- Stationary solution for the 0-th order atmospheric temperature.*
- real(kind=8) epsa

  ϵ_a *- Emissivity coefficient for the grey-body atmosphere.*
- real(kind=8) ga

  γ_a *- Specific heat capacity of the atmosphere.*
- real(kind=8) rr

  R *- Gas constant of dry air*
- real(kind=8) scale

  $L_y = L\,\pi$ *- The characteristic space scale.*
- real(kind=8) pi

  π
- real(kind=8) lr

  L_R *- Rossby deformation radius*
- real(kind=8) g

γ

- real(kind=8) rp

  r' - *Frictional coefficient at the bottom of the ocean.*
- real(kind=8) dp

  d' - *Non-dimensional mechanical coupling parameter between the ocean and the atmosphere.*
- real(kind=8) kd

  k_d - *Non-dimensional bottom atmospheric friction coefficient.*
- real(kind=8) kdp

  k'_d - *Non-dimensional internal atmospheric friction coefficient.*
- real(kind=8) cpo

  C'_a - *Non-dimensional constant short-wave radiation of the ocean.*
- real(kind=8) lpo

  λ'_o - *Non-dimensional sensible + turbulent heat exchange from ocean to atmosphere.*
- real(kind=8) cpa

  C'_a - *Non-dimensional constant short-wave radiation of the atmosphere.*
- real(kind=8) lpa

  λ'_a - *Non-dimensional sensible + turbulent heat exchange from atmosphere to ocean.*
- real(kind=8) sbpo

  $\sigma'_{B,o}$ - *Long wave radiation lost by ocean to atmosphere & space.*
- real(kind=8) sbpa

  $\sigma'_{B,a}$ - *Long wave radiation from atmosphere absorbed by ocean.*
- real(kind=8) lsbpo

  $S'_{B,o}$ - *Long wave radiation from ocean absorbed by atmosphere.*
- real(kind=8) lsbpa

  $S'_{B,a}$ - *Long wave radiation lost by atmosphere to space & ocean.*
- real(kind=8) l

  L - *Domain length scale*
- real(kind=8) sc

  Ratio of surface to atmosphere temperature.
- real(kind=8) sb

  Stefan–Boltzmann constant.
- real(kind=8) betp

  β' - *Non-dimensional beta parameter*
- real(kind=8) nua =0.D0

  Dissipation in the atmosphere.
- real(kind=8) nuo =0.D0

  Dissipation in the ocean.
- real(kind=8) nuap

  Non-dimensional dissipation in the atmosphere.
- real(kind=8) nuop

  Non-dimensional dissipation in the ocean.

7.17.1 Detailed Description

The subclass containing the physical parameters of the model.

Definition at line 22 of file params.f90.

7.17.2 Member Data Documentation

7.17.2.1 cpa

```
real(kind=8) params::physicsconfiguration::cpa
```

C'_a - Non-dimensional constant short-wave radiation of the atmosphere.

Remarks

Cpa acts on psi1-psi3, not on theta.

Definition at line 57 of file params.f90.

```
57     REAL(KIND=8) :: cpa       !< \f$C'_a\f$ - Non-dimensional constant short-wave radiation of the
       atmosphere. @remark Cpa acts on psi1-psi3, not on theta.
```

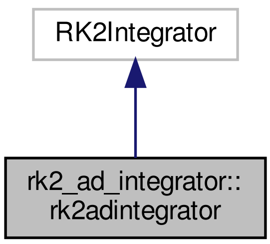
The documentation for this type was generated from the following file:

- params.f90

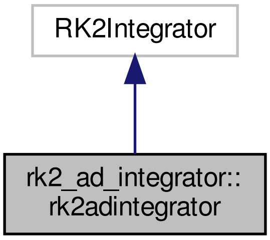
7.18 rk2_ad_integrator::rk2adintegrator Type Reference

Class for the Heun (RK2) AD integrator object.

Inheritance diagram for rk2_ad_integrator::rk2adintegrator:



Collaboration diagram for rk2_ad_integrator::rk2adintegrator:



7.18.1 Detailed Description

Class for the Heun (RK2) AD integrator object.

Definition at line 29 of file rk2_ad_integrator.f90.

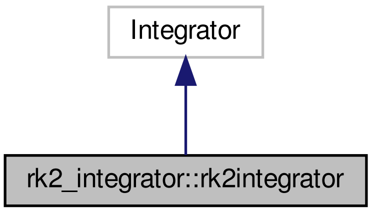
The documentation for this type was generated from the following file:

- rk2_ad_integrator.f90

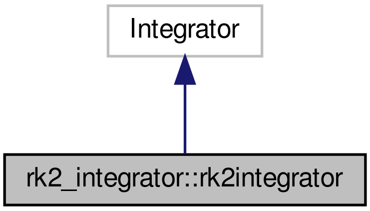
7.19 rk2_integrator::rk2integrator Type Reference

Class for the Heun (RK2) integrator object.

Inheritance diagram for rk2_integrator::rk2integrator:



Collaboration diagram for rk2_integrator::rk2integrator:



Public Attributes

- real(kind=8), dimension(:), allocatable buf_y1
  Buffer to hold the intermediate position (Heun algorithm)
- real(kind=8), dimension(:), allocatable buf_f0
  Buffer to hold tendencies at the initial position.
- real(kind=8), dimension(:), allocatable buf_f1
  Buffer to hold tendencies at the intermediate position.

7.19.1 Detailed Description

Class for the Heun (RK2) integrator object.

Definition at line 25 of file rk2_integrator.f90.

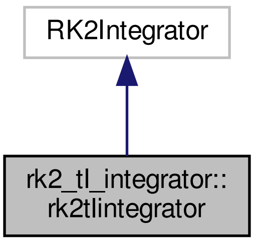
The documentation for this type was generated from the following file:

- rk2_integrator.f90

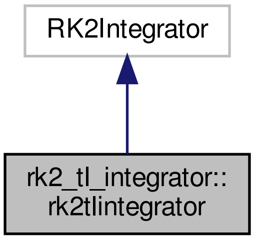
7.20 rk2_tl_integrator::rk2tlintegrator Type Reference

Class for the Heun (RK2) TL integrator object.

Inheritance diagram for rk2_tl_integrator::rk2tlintegrator:



Collaboration diagram for rk2_tl_integrator::rk2tlintegrator:



7.20.1 Detailed Description

Class for the Heun (RK2) TL integrator object.

Definition at line 29 of file rk2_tl_integrator.f90.

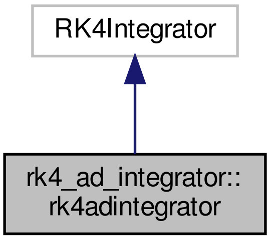
The documentation for this type was generated from the following file:

- rk2_tl_integrator.f90

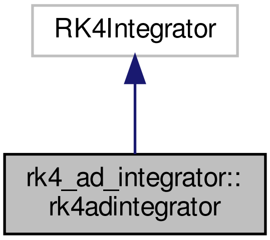
7.21 rk4_ad_integrator::rk4adintegrator Type Reference

Class for the fourth-order Runge-Kutta (RK4) AD integrator object.

Inheritance diagram for rk4_ad_integrator::rk4adintegrator:



Collaboration diagram for rk4_ad_integrator::rk4adintegrator:



7.21.1 Detailed Description

Class for the fourth-order Runge-Kutta (RK4) AD integrator object.

Definition at line 27 of file rk4_ad_integrator.f90.

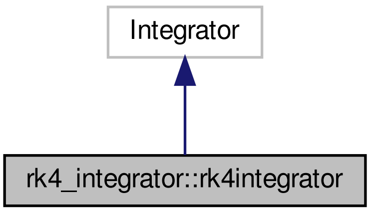
The documentation for this type was generated from the following file:

- rk4_ad_integrator.f90

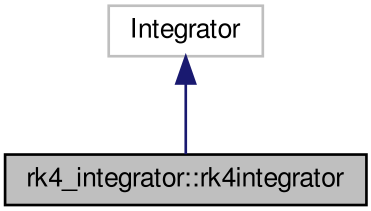
7.22 rk4_integrator::rk4integrator Type Reference

Class for the fourth-order Runge-Kutta (RK4) integrator object.

Inheritance diagram for rk4_integrator::rk4integrator:



Collaboration diagram for rk4_integrator::rk4integrator:



Public Attributes

- real(kind=8), dimension(:), allocatable buf_y1
  Buffer to hold the intermediate position.
- real(kind=8), dimension(:), allocatable buf_ka
  Buffer to hold tendencies at the initial position.
- real(kind=8), dimension(:), allocatable buf_kb
  Buffer to hold tendencies at the intermediate position.

7.22.1 Detailed Description

Class for the fourth-order Runge-Kutta (RK4) integrator object.

Definition at line 23 of file rk4_integrator.f90.

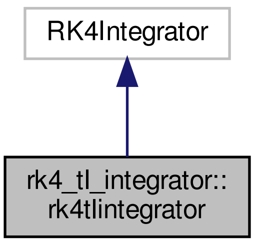
The documentation for this type was generated from the following file:

- rk4_integrator.f90

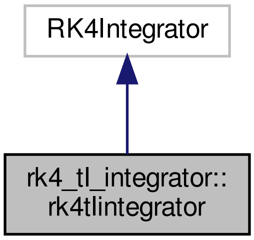
7.23 rk4_tl_integrator::rk4tlintegrator Type Reference

Class for the fourth-order Runge-Kutta (RK4) TL integrator object.

Inheritance diagram for rk4_tl_integrator::rk4tlintegrator:



Collaboration diagram for rk4_tl_integrator::rk4tlintegrator:



7.23.1 Detailed Description

Class for the fourth-order Runge-Kutta (RK4) TL integrator object.

Definition at line 27 of file rk4_tl_integrator.f90.

The documentation for this type was generated from the following file:

- rk4_tl_integrator.f90

7.24 stat::stataccumulator Type Reference

Statistics accumulator objects class.

Public Attributes

- integer i =0
    Number of stats accumulated.
- real(kind=8), dimension(:), allocatable m
    Vector storing the inline mean.
- real(kind=8), dimension(:), allocatable mprev
    Previous mean vector.
- real(kind=8), dimension(:), allocatable v
    Vector storing the inline variance.

7.24.1 Detailed Description

Statistics accumulator objects class.

Definition at line 20 of file stat.f90.

The documentation for this type was generated from the following file:

- stat.f90

7.25 integrator_def::step_int Interface Reference

Abstract interface for the procedure to make the integrator compute a model's time step.

7.25.1 Detailed Description

Abstract interface for the procedure to make the integrator compute a model's time step.

Parameters

in,out	*integr*	Integrator object to perform the step with.
in	*y*	Initial point.
in	*t*	Actual integration time
out	*res*	Final point after the step.

Definition at line 50 of file integrator_def.f90.

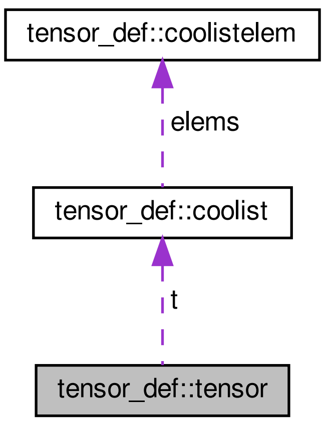
The documentation for this interface was generated from the following file:

- integrator_def.f90

7.26 tensor_def::tensor Type Reference

General class to represent a sparse tensor.

Collaboration diagram for tensor_def::tensor:



Public Attributes

- type(coolist), dimension(:), allocatable t

    Sparse representation of the tensor as a tensor_def::coolist .

7.26.1 Detailed Description

General class to represent a sparse tensor.

Definition at line 36 of file tensor_def.f90.

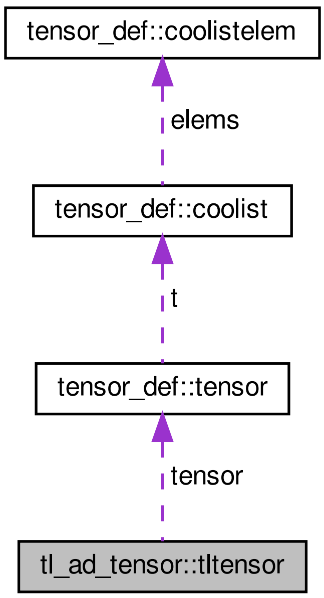
The documentation for this type was generated from the following file:

- tensor_def.f90

7.27 tl_ad_tensor::tltensor Type Reference

Tensor representation of the Tangent Linear tendencies.

Collaboration diagram for tl_ad_tensor::tltensor:



Public Attributes

- type(tensor) tensor

    The TL tensor object.

- integer, dimension(:), allocatable count_elems

    A list of the number of non-zero entries of the tensor component along the first index.

7.27.1 Detailed Description

Tensor representation of the Tangent Linear tendencies.

Definition at line 34 of file tl_ad_tensor.f90.

The documentation for this type was generated from the following file:

- tl_ad_tensor.f90

Index