Reference Manual

Generated by Doxygen 1.8.11

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. If gfortran is selected, the code should be compiled with gfortran 4.7+ (allows for allocatable arrays in namelists). If ifort is selected, the code has been tested with the version 14.0.2 and we do not guarantee compatibility with older compiler version.

To install, unpack the archive in a folder, and run: make

Remark: 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 called aotensor which includes all the coefficients. This tensor is computed once at the program initialization.

- maooam.f90 : Main program.
- aotensor_def.f90 : Tensor aotensor computation module.
- IC_def.f90 : A module which loads the user specified initial condition.
- inprod_analytic.f90 : Inner products computation module.
- rk2_integrator.f90 : A module which contains the Heun integrator for the model equations.
- rk4_integrator.f90 : A module which contains the RK4 integrator for the model equations.
- Makefile : The Makefile.
- gfortran.mk : Gfortran compiler options file.
- ifort.mk : Ifort compiler options file.
- params.f90 : The model parameters module.
- tl_ad_tensor.f90 : Tangent Linear (TL) and Adjoint (AD) model tensors definition module
- rk2_tl_ad_integrator.f90 : Heun Tangent Linear (TL) and Adjoint (AD) model integrators module
- rk4_tl_ad_integrator.f90 : RK4 Tangent Linear (TL) and Adjoint (AD) model integrators 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.f90 : Tensor utility module.
- stat.f90 : A module for statistic accumulation.
- 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, described in Vannitsem et al. (2015) is given as an example in the archive.
- 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)

The tangent linear and adjoint models of MAOOAM are provided in the tl_ad_tensor, rk2_tl_ad_integrator and rk4_tl_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.

Ideally, the tensor aotensor_def::aotensor is composed as an upper triangular matrix (in the last two coordinates).

The tensor for this model is composed in the aotensor_def module and uses the inner products defined in the inprod_analytic module.

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 (after calling params::init_params and aotensor_def::aotensor) by calling tl_ad_tensor::init_tltensor() and tl_ad_tensor::init_adtensor(). The tendencies are then given by the routine tl(t,ystar,deltay,buf) and ad(t,ystar,deltay,buf). An integrator with the Heun method is available in the module rk2_tl_ad_integrator and a fourth-order Runge-Kutta integrator in rk4_tl_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 modules with brief descriptions:

Chapter 4

Data Type Index

4.1 Data Types List

Here are the data types with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Module Documentation

6.1 aotensor_def Module Reference

The equation tensor for the coupled ocean-atmosphere model with temperature which allows for an extensible set of modes in the ocean and in the atmosphere.

Functions/Subroutines

- integer function psi (i)

  Translate the $\psi_{a,i}$ coefficients into effective coordinates.
- integer function theta (i)

  Translate the $\theta_{a,i}$ coefficients into effective coordinates.
- integer function a (i)

  Translate the $\psi_{o,i}$ coefficients into effective coordinates.
- integer function t (i)

  Translate the $\delta T_{o,i}$ coefficients into effective coordinates.
- integer function kdelta (i, j)

  Kronecker delta function.
- subroutine coeff (i, j, k, v)

  Subroutine to add element in the aotensor $\mathcal{T}_{i,j,k}$ structure.
- subroutine add_count (i, j, k, v)

  Subroutine to count the elements of the aotensor $\mathcal{T}_{i,j,k}$. Add +1 to count_elems(i) for each value that is added to the tensor i-th component.
- subroutine compute_aotensor (func)

  Subroutine to compute the tensor aotensor.
- subroutine, public init_aotensor

  Subroutine to initialise the aotensor tensor.

Variables

- integer, dimension(:), allocatable count_elems

  Vector used to count the tensor elements.
- real(kind=8), parameter real_eps = 2.2204460492503131e-16

  Epsilon to test equality with 0.
- type(coolist), dimension(:), allocatable, public aotensor

  $\mathcal{T}_{i,j,k}$ *- Tensor representation of the tendencies.*

6.1.1 Detailed Description

The equation tensor for the coupled ocean-atmosphere model with temperature which allows for an extensible set of modes in the ocean and in the atmosphere.

Copyright

Remarks

Generated Fortran90/95 code from aotensor.lua

6.1.2 Function/Subroutine Documentation

6.1.2.1 integer function aotensor_def::a (integer *i*) [private]

Translate the $\psi_{o,i}$ coefficients into effective coordinates.

Definition at line 76 of file aotensor_def.f90.

```
76      INTEGER :: i,a
77      a = i + 2 * natm
```

6.1.2.2 subroutine aotensor_def::add_count (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Subroutine to count the elements of the aotensor $\mathcal{T}_{i,j,k}$. Add +1 to count_elems(i) for each value that is added to the tensor i-th component.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value that will be added

Definition at line 124 of file aotensor_def.f90.

```
124      INTEGER, INTENT(IN) :: i,j,k
125      REAL(KIND=8), INTENT(IN)  :: v
126      IF (abs(v) .ge. real_eps) count_elems(i)=count_elems(i)+1
```

6.1.2.3 subroutine aotensor_def::coeff (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Subroutine to add element in the aotensor $\mathcal{T}_{i,j,k}$ structure.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value to add

Definition at line 99 of file aotensor_def.f90.

```
99      INTEGER, INTENT(IN) :: i,j,k
100      REAL(KIND=8), INTENT(IN) :: v
101      INTEGER :: n
102      IF (.NOT. ALLOCATED(aotensor)) stop "*** coeff routine : tensor not yet allocated ***"
103      IF (.NOT. ALLOCATED(aotensor(i)%elems)) stop "*** coeff routine : tensor not yet allocated ***"
104      IF (abs(v) .ge. real_eps) THEN
105         n=(aotensor(i)%nelems)+1
106         IF (j .LE. k) THEN
107            aotensor(i)%elems(n)%j=j
108            aotensor(i)%elems(n)%k=k
109         ELSE
110            aotensor(i)%elems(n)%j=k
111            aotensor(i)%elems(n)%k=j
112         END IF
113         aotensor(i)%elems(n)%v=v
114         aotensor(i)%nelems=n
115      END IF
```

6.1.2.4 subroutine aotensor_def::compute_aotensor (external *func*) [private]

Subroutine to compute the tensor aotensor.

Parameters

func	External function to be used

Definition at line 132 of file aotensor_def.f90.

6.1.2.5 subroutine, public aotensor_def::init_aotensor ( )

Subroutine to initialise the aotensor tensor.

Remarks

This procedure will also call params::init_params() and inprod_analytic::init_inprod() . It will finally call inprod_analytic::deallocate_inprod() to remove the inner products, which are not needed anymore at this point.

Definition at line 202 of file aotensor_def.f90.

```
202      INTEGER :: i
203      INTEGER :: allocstat
204
205      CALL init_params  ! Iniatialise the parameter
206
207      CALL init_inprod  ! Initialise the inner product tensors
208
209      ALLOCATE(aotensor(ndim),count_elems(ndim), stat=allocstat)
210      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
```

```
    count_elems=0

    CALL compute_aotensor(add_count)

    DO i=1,ndim
       ALLOCATE(aotensor(i)%elems(count_elems(i)), stat=allocstat)
       IF (allocstat /= 0) stop "*** Not enough memory ! ***"
    END DO

    DEALLOCATE(count_elems, stat=allocstat)
    IF (allocstat /= 0) stop "*** Deallocation problem ! ***"

    CALL compute_aotensor(coeff)

    CALL simplify(aotensor)

    CALL deallocate_inprod ! Clean the inner product tensors

```

6.1.2.6 integer function aotensor_def::kdelta (integer *i*, integer *j*) [private]

Kronecker delta function.

Definition at line 88 of file aotensor_def.f90.

```
   INTEGER :: i,j,kdelta
   kdelta=0
   IF (i == j) kdelta = 1
```

6.1.2.7 integer function aotensor_def::psi (integer *i*) [private]

Translate the $\psi_{a,i}$ coefficients into effective coordinates.

Definition at line 64 of file aotensor_def.f90.

```
   INTEGER :: i,psi
   psi = i
```

6.1.2.8 integer function aotensor_def::t (integer *i*) [private]

Translate the $\delta T_{o,i}$ coefficients into effective coordinates.

Definition at line 82 of file aotensor_def.f90.

```
   INTEGER :: i,t
   t = i + 2 * natm + noc
```

6.1.2.9 integer function aotensor_def::theta (integer *i*) [private]

Translate the $\theta_{a,i}$ coefficients into effective coordinates.

Definition at line 70 of file aotensor_def.f90.

```
   INTEGER :: i,theta
   theta = i + natm
```

6.1.3 Variable Documentation

6.1.3.1 type(coolist), dimension(:), allocatable, public aotensor_def::aotensor

$\mathcal{T}_{i,j,k}$ - Tensor representation of the tendencies.

Definition at line 45 of file aotensor_def.f90.

```
45   TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: aotensor
```

6.1.3.2 integer, dimension(:), allocatable aotensor_def::count_elems [private]

Vector used to count the tensor elements.

Definition at line 37 of file aotensor_def.f90.

```
37   INTEGER, DIMENSION(:), ALLOCATABLE :: count_elems
```

6.1.3.3 real(kind=8), parameter aotensor_def::real_eps = 2.2204460492503131e-16 [private]

Epsilon to test equality with 0.

Definition at line 40 of file aotensor_def.f90.

```
40   REAL(KIND=8), PARAMETER :: real_eps = 2.2204460492503131e-16
```

6.2 ic_def Module Reference

Module to load the initial condition.

Functions/Subroutines

- subroutine, public load_ic

  Subroutine to load the initial condition if IC.nml exists. If it does not, then write IC.nml with 0 as initial condition.

Variables

- logical exists

  Boolean to test for file existence.
- real(kind=8), dimension(:), allocatable, public ic

  Initial condition vector.

6.2.1 Detailed Description

Module to load the initial condition.

Copyright

2016 Lesley De Cruz, Jonathan Demaeyer & Sebastian Schubert See LICENSE.txt for license information.

6.2.2 Function/Subroutine Documentation

6.2.2.1 subroutine, public ic_def::load_ic ( )

Subroutine to load the initial condition if IC.nml exists. If it does not, then write IC.nml with 0 as initial condition.

Definition at line 32 of file ic_def.f90.

```
32      INTEGER :: i,allocstat,j
33      CHARACTER(len=20) :: fm
34      REAL(KIND=8) :: size_of_random_noise
35      INTEGER, DIMENSION(:), ALLOCATABLE :: seed
36      CHARACTER(LEN=4) :: init_type
37      namelist /iclist/ ic
38      namelist /rand/ init_type,size_of_random_noise,seed
39
40
41      fm(1:6)='(F3.1)'
42
43      CALL random_seed(size=j)
44
45      IF (ndim == 0) stop "*** Number of dimensions is 0! ***"
46      ALLOCATE(ic(0:ndim),seed(j), stat=allocstat)
47      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
48
49      INQUIRE(file='./IC.nml',exist=exists)
50
51      IF (exists) THEN
52         OPEN(8, file="IC.nml", status='OLD', recl=80, delim='APOSTROPHE')
53         READ(8,nml=iclist)
54         READ(8,nml=rand)
55         CLOSE(8)
56         SELECT CASE (init_type)
57           CASE ('seed')
58             CALL random_seed(put=seed)
59             CALL random_number(ic)
60             ic=2*(ic-0.5)
61             ic=ic*size_of_random_noise*10.d0
62             ic(0)=1.0d0
63             WRITE(6,*) "*** IC.nml namelist written. Starting with 'seeded' random initial condition !***"
64           CASE ('rand')
65             CALL init_random_seed()
66             CALL random_seed(get=seed)
67             CALL random_number(ic)
68             ic=2*(ic-0.5)
69             ic=ic*size_of_random_noise*10.d0
70             ic(0)=1.0d0
71             WRITE(6,*) "*** IC.nml namelist written. Starting with random initial condition !***"
72           CASE ('zero')
73             CALL init_random_seed()
74             CALL random_seed(get=seed)
75             ic=0
76             ic(0)=1.0d0
77             WRITE(6,*) "*** IC.nml namelist written. Starting with initial condition in IC.nml !***"
78           CASE ('read')
79             CALL init_random_seed()
80             CALL random_seed(get=seed)
81             !nothing has to be done IC has already the right values
82             WRITE(6,*) "*** IC.nml namelist written. Starting with initial condition in IC.nml !***"
83         END SELECT
84      ELSE
85         CALL init_random_seed()
86         CALL random_seed(get=seed)
87         ic=0
88         ic(0)=1.0d0
89         init_type="zero"
```

```
      size_of_random_noise=0.d0
      WRITE(6,*) "*** IC.nml namelist written. Starting with 0 as initial condition !***"
   END IF
   OPEN(8, file="IC.nml", status='REPLACE')
   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= "&
            &//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
            &%Nx,fm))//", Ny= "//trim(rstr(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= "&
            &//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
            &%Nx,fm))//", Ny= "//trim(rstr(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(owavenum(i)%Nx,fm))//", Ny= "&
            &//trim(rstr(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(owavenum(i)%Nx,fm))//", Ny= "&
            &//trim(rstr(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)

```

6.2.3 Variable Documentation

6.2.3.1 logical ic_def::exists [private]

Boolean to test for file existence.

Definition at line 21 of file ic_def.f90.

```
21   LOGICAL :: exists !< Boolean to test for file existence.
```

6.2.3.2 real(kind=8), dimension(:), allocatable, public ic_def::ic

Initial condition vector.

Definition at line 23 of file ic_def.f90.

```
23   REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: ic !< Initial condition vector
```

6.3 inprod_analytic Module Reference

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

Data Types

- type atm_tensors

  Type holding the atmospheric inner products tensors.
- type atm_wavenum

  Atmospheric bloc specification type.
- type ocean_tensors

  Type holding the oceanic inner products tensors.
- type ocean_wavenum

  Oceanic bloc specification type.

Functions/Subroutines

- real(kind=8) function b1 (Pi, Pj, Pk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function b2 (Pi, Pj, Pk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function delta (r)

  Integer Dirac delta function.
- real(kind=8) function flambda (r)

  "Odd or even" function
- real(kind=8) function s1 (Pj, Pk, Mj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function s2 (Pj, Pk, Mj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function s3 (Pj, Pk, Hj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function s4 (Pj, Pk, Hj, Hk)

  Cehelsky & Tung Helper functions.
- subroutine calculate_a

  Eigenvalues of the Laplacian (atmospheric)
- subroutine calculate_b

  Streamfunction advection terms (atmospheric)
- subroutine calculate_c_atm

  Beta term for the atmosphere.
- subroutine calculate_d

  Forcing of the ocean on the atmosphere.
- subroutine calculate_g

  Temperature advection terms (atmospheric)
- subroutine calculate_s

  Forcing (thermal) of the ocean on the atmosphere.
- subroutine calculate_k

Forcing of the atmosphere on the ocean.
- subroutine calculate_m

  Forcing of the ocean fields on the ocean.
- subroutine calculate_n

  Beta term for the ocean.
- subroutine calculate_o

  Temperature advection term (passive scalar)
- subroutine calculate_c_oc

  Streamfunction advection terms (oceanic)
- subroutine calculate_w

  Short-wave radiative forcing of the ocean.
- subroutine, public init_inprod

  Initialisation of the inner product.
- subroutine, public deallocate_inprod

  Deallocation of the inner products.

Variables

- type(atm_wavenum), dimension(:), allocatable, public awavenum

  Atmospheric blocs specification.
- type(ocean_wavenum), dimension(:), allocatable, public owavenum

  Oceanic blocs specification.
- type(atm_tensors), public atmos

  Atmospheric tensors.
- type(ocean_tensors), public ocean

  Oceanic tensors.

6.3.1 Detailed Description

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

Remarks

Generated Fortran90/95 code from inprod_analytic.lua

6.3.2 Function/Subroutine Documentation

6.3.2.1 real(kind=8) function inprod_analytic::b1 (integer *Pi*, integer *Pj*, integer *Pk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 91 of file inprod_analytic.f90.

```
INTEGER :: pi,pj,pk
b1 = (pk + pj) / REAL(pi)
```

6.3.2.2 real(kind=8) function inprod_analytic::b2 (integer *Pi*, integer *Pj*, integer *Pk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 97 of file inprod_analytic.f90.

```
97      INTEGER :: pi,pj,pk
98      b2 = (pk - pj) / REAL(pi)
```

6.3.2.3 subroutine inprod_analytic::calculate_a ( ) [private]

Eigenvalues of the Laplacian (atmospheric)

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

Definition at line 155 of file inprod_analytic.f90.

```
155      INTEGER :: i
156      TYPE(atm_wavenum) :: ti
157      INTEGER :: allocstat
158      IF (natm == 0 ) THEN
159         stop "*** Problem with calculate_a : natm==0 ! ***"
160      ELSE
161         IF (.NOT. ALLOCATED(atmos%a)) THEN
162            ALLOCATE(atmos%a(natm,natm), stat=allocstat)
163            IF (allocstat /= 0) stop "*** Not enough memory ! ***"
164         END IF
165      END IF
166      atmos%a=0.d0
167
168      DO i=1,natm
169         ti = awavenum(i)
170         atmos%a(i,i) = -(n**2) * ti%Nx**2 - ti%Ny**2
171      ENDDO
```

6.3.2.4 subroutine inprod_analytic::calculate_b ( ) [private]

Streamfunction advection terms (atmospheric)

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

Remarks

> Atmospheric g and a tensors must be computed before calling this routine

Definition at line 182 of file inprod_analytic.f90.

```
182      INTEGER :: i,j,k
183      INTEGER :: allocstat
184
185      IF ((.NOT. ALLOCATED(atmos%a)) .OR. (.NOT. ALLOCATED(atmos%g))) THEN
186         stop "*** atmos%a and atmos%g must be defined before calling calculate_b ! ***"
187      END IF
188
189      IF (natm == 0 ) THEN
190         stop "*** Problem with calculate_b : natm==0 ! ***"
191      ELSE
192         IF (.NOT. ALLOCATED(atmos%b)) THEN
193            ALLOCATE(atmos%b(natm,natm,natm), stat=allocstat)
194            IF (allocstat /= 0) stop "*** Not enough memory ! ***"
195         END IF
196      END IF
197      atmos%b=0.d0
198
199      DO i=1,natm
200         DO j=1,natm
201            DO k=1,natm
202               atmos%b(i,j,k)= atmos%a(k,k) * atmos%g(i,j,k)
203            END DO
204         END DO
205      END DO
```

6.3.2.5 subroutine inprod_analytic::calculate_c_atm ( ) [private]

Beta term for the atmosphere.

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

Remarks

Strict function !! Only accepts KL type. For any other combination, it will not calculate anything

Definition at line 216 of file inprod_analytic.f90.

```
216     INTEGER :: i,j
217     TYPE(atm_wavenum) :: ti, tj
218     REAL(KIND=8) :: val
219     INTEGER :: allocstat
220
221     IF (natm == 0 ) THEN
222        stop "*** Problem with calculate_c_atm : natm==0 ! ***"
223     ELSE
224        IF (.NOT. ALLOCATED(atmos%c)) THEN
225           ALLOCATE(atmos%c(natm,natm), stat=allocstat)
226           IF (allocstat /= 0) stop "*** Not enough memory ! ***"
227        END IF
228     END IF
229     atmos%c=0.d0
230
231     DO i=1,natm
232        DO j=1,natm
233           ti = awavenum(i)
234           tj = awavenum(j)
235           val = 0.d0
236           IF ((ti%typ == "K") .AND. (tj%typ == "L")) THEN
237              val = n * ti%M * delta(ti%M - tj%H) * delta(ti%P - tj%P)
238           END IF
239           IF (val /= 0.d0) THEN
240              atmos%c(i,j)=val
241              atmos%c(j,i)= - val
242           ENDIF
243        END DO
244     END DO
```

6.3.2.6 subroutine inprod_analytic::calculate_c_oc ( ) [private]

Streamfunction advection terms (oceanic)

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

Remarks

Requires O_{i,j,k} and M_{i,j} to be calculated beforehand.

Definition at line 568 of file inprod_analytic.f90.

```
568     INTEGER :: i,j,k
569     REAL(KIND=8) :: val
570     INTEGER :: allocstat
571
572     IF ((.NOT. ALLOCATED(ocean%O)) .OR. (.NOT. ALLOCATED(ocean%M))) THEN
573        stop "*** ocean%O and ocean%M must be defined before calling calculate_C ! ***"
574     END IF
575
576     IF (noc == 0 ) THEN
577        stop "*** Problem with calculate_C : noc==0 ! ***"
578     ELSE
579        IF (.NOT. ALLOCATED(ocean%C)) THEN
```

```
        ALLOCATE(ocean%C(noc,noc,noc), stat=allocstat)
        IF (allocstat /= 0) stop "*** Not enough memory ! ***"
     END IF
  END IF
  ocean%C=0.d0
  val=0.d0

  DO i=1,noc
     DO j=1,noc
        DO k=1,noc
           val = ocean%M(k,k) * ocean%O(i,j,k)
           IF (val /= 0.d0) ocean%C(i,j,k) = val
        END DO
     END DO
  END DO
```

6.3.2.7 subroutine inprod_analytic::calculate_d ( ) [private]

Forcing of the ocean on the atmosphere.

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

Remarks

Atmospheric s tensor and oceanic M tensor must be computed before calling this routine !

Definition at line 255 of file inprod_analytic.f90.

```
  INTEGER :: i,j
  INTEGER :: allocstat

  IF ((.NOT. ALLOCATED(atmos%s)) .OR. (.NOT. ALLOCATED(ocean%M))) THEN
     stop "*** atmos%s and ocean%M must be defined before calling calculate_d ! ***"
  END IF


  IF (natm == 0 ) THEN
     stop "*** Problem with calculate_d : natm==0 ! ***"
  ELSE
     IF (.NOT. ALLOCATED(atmos%d)) THEN
        ALLOCATE(atmos%d(natm,noc), stat=allocstat)
        IF (allocstat /= 0) stop "*** Not enough memory ! ***"
     END IF
  END IF
  atmos%d=0.d0

  DO i=1,natm
     DO j=1,noc
        atmos%d(i,j)=atmos%s(i,j) * ocean%M(j,j)
     END DO
  END DO
```

6.3.2.8 subroutine inprod_analytic::calculate_g ( ) [private]

Temperature advection terms (atmospheric)

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

Definition at line 288 of file inprod_analytic.f90.

```
    INTEGER :: i,j,k
    TYPE(atm_wavenum) :: ti, tj, tk
    REAL(KIND=8) :: val,vb1, vb2, vs1, vs2, vs3, vs4
    INTEGER :: allocstat

    IF (natm == 0 ) THEN
       stop "*** Problem with calculate_g : natm==0 ! ***"
    ELSE
       IF (.NOT. ALLOCATED(atmos%g)) THEN
          ALLOCATE(atmos%g(natm,natm,natm), stat=allocstat)
          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
       END IF
    END IF
    atmos%g=0.d0

    DO i=1,natm
       DO j=1,natm
          DO k=1,natm
             ti = awavenum(i)
             tj = awavenum(j)
             tk = awavenum(k)
             val=0.d0
             IF ((ti%typ == "A") .AND. (tj%typ == "K") .AND. (tk%typ == "L")) THEN
                vb1 = b1(ti%P,tj%P,tk%P)
                vb2 = b2(ti%P,tj%P,tk%P)
                val = -2 * sqrt(2.) / pi * tj%M * delta(tj%M - tk%H) * flambda(ti%P + tj%P + tk%P)
                IF (val /= 0.d0) val = val * (vb1**2 / (vb1**2 - 1) - vb2**2 / (vb2**2 - 1))
             ELSEIF ((ti%typ == "K") .AND. (tj%typ == "K") .AND. (tk%typ == "L")) THEN
                vs1 = s1(tj%P,tk%P,tj%M,tk%H)
                vs2 = s2(tj%P,tk%P,tj%M,tk%H)
                val = vs1 * (delta(ti%M - tk%H - tj%M) * delta(ti%P -&
                     & tk%P + tj%P) - delta(ti%M- tk%H - tj%M) *&
                     & delta(ti%P + tk%P - tj%P) + (delta(tk%H - tj%M&
                     & + ti%M) + delta(tk%H - tj%M - ti%M)) *&
                     & delta(tk%P + tj%P - ti%P)) + vs2 * (delta(ti%M&
                     & - tk%H - tj%M) * delta(ti%P - tk%P - tj%P) +&
                     & (delta(tk%H - tj%M - ti%M) + delta(ti%M + tk%H&
                     & - tj%M)) * (delta(ti%P - tk%P + tj%P) -&
                     & delta(tk%P - tj%P + ti%P)))
             END IF
             val=val*n
             IF (val /= 0.d0) THEN
                atmos%g(i,j,k) = val
                atmos%g(j,k,i) = val
                atmos%g(k,i,j) = val
                atmos%g(i,k,j) = -val
                atmos%g(j,i,k) = -val
                atmos%g(k,j,i) = -val
             ENDIF
          END DO
       END DO
    END DO

    DO i=1,natm
       DO j=i,natm
          DO k=j,natm
             ti = awavenum(i)
             tj = awavenum(j)
             tk = awavenum(k)
             val=0.d0

             IF ((ti%typ == "L") .AND. (tj%typ == "L") .AND. (tk%typ == "L")) THEN
                vs3 = s3(tj%P,tk%P,tj%H,tk%H)
                vs4 = s4(tj%P,tk%P,tj%H,tk%H)
                val = vs3 * ((delta(tk%H - tj%H - ti%H) - delta(tk%H &
                     &- tj%H + ti%H)) * delta(tk%P + tj%P - ti%P) +&
                     & delta(tk%H + tj%H - ti%H) * (delta(tk%P - tj%P&
                     & + ti%P) - delta(tk%P - tj%P - ti%P))) + vs4 *&
                     & ((delta(tk%H + tj%H - ti%H) * delta(tk%P - tj&
                     &%P - ti%P)) + (delta(tk%H - tj%H + ti%H) -&
                     & delta(tk%H - tj%H - ti%H)) * (delta(tk%P - tj&
                     &%P - ti%P) - delta(tk%P - tj%P + ti%P)))
             ENDIF
             val=val*n
             IF (val /= 0.d0) THEN
                atmos%g(i,j,k) = val
                atmos%g(j,k,i) = val
                atmos%g(k,i,j) = val
                atmos%g(i,k,j) = -val
                atmos%g(j,i,k) = -val
                atmos%g(k,j,i) = -val
             ENDIF
          ENDDO
       ENDDO
    ENDDO
```

6.3.2.9 subroutine inprod_analytic::calculate_k ( ) [private]

Forcing of the atmosphere on the ocean.

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

Remarks

atmospheric a and s tensors must be computed before calling this function !

Definition at line 434 of file inprod_analytic.f90.

```
434     INTEGER :: i,j
435     INTEGER :: allocstat
436
437     IF ((.NOT. ALLOCATED(atmos%a)) .OR. (.NOT. ALLOCATED(atmos%s))) THEN
438        stop "*** atmos%a and atmos%s must be defined before calling calculate_K ! ***"
439     END IF
440
441     IF (noc == 0 ) THEN
442        stop "*** Problem with calculate_K : noc==0 ! ***"
443     ELSEIF (natm == 0 ) THEN
444        stop "*** Problem with calculate_K : natm==0 ! ***"
445     ELSE
446        IF (.NOT. ALLOCATED(ocean%K)) THEN
447           ALLOCATE(ocean%K(noc,natm), stat=allocstat)
448           IF (allocstat /= 0) stop "*** Not enough memory ! ***"
449        END IF
450     END IF
451     ocean%K=0.d0
452
453     DO i=1,noc
454        DO j=1,natm
455           ocean%K(i,j) = atmos%s(j,i) * atmos%a(j,j)
456        END DO
457     END DO
```

6.3.2.10 subroutine inprod_analytic::calculate_m ( ) [private]

Forcing of the ocean fields on the ocean.

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

Definition at line 464 of file inprod_analytic.f90.

```
464     INTEGER :: i
465     TYPE(ocean_wavenum) :: di
466     INTEGER :: allocstat
467     IF (noc == 0 ) THEN
468        stop "*** Problem with calculate_M : noc==0 ! ***"
469     ELSE
470        IF (.NOT. ALLOCATED(ocean%M)) THEN
471           ALLOCATE(ocean%M(noc,noc), stat=allocstat)
472           IF (allocstat /= 0) stop "*** Not enough memory ! ***"
473        END IF
474     END IF
475     ocean%M=0.d0
476
477     DO i=1,noc
478        di = owavenum(i)
479        ocean%M(i,i) = -(n**2) * di%Nx**2 - di%Ny**2
480     END DO
```

6.3.2.11 subroutine inprod_analytic::calculate_n ( ) [private]

Beta term for the ocean.

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

Definition at line 487 of file inprod_analytic.f90.

```
    INTEGER :: i,j
    TYPE(ocean_wavenum) :: di,dj
    REAL(KIND=8) :: val
    INTEGER :: allocstat
    IF (noc == 0 ) THEN
       stop "*** Problem with calculate_N : noc==0 ! ***"
    ELSE
       IF (.NOT. ALLOCATED(ocean%N)) THEN
          ALLOCATE(ocean%N(noc,noc), stat=allocstat)
          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
       END IF
    END IF
    ocean%N=0.d0
    val=0.d0

    DO i=1,noc
       DO j=1,noc
          di = owavenum(i)
          dj = owavenum(j)
          val = delta(di%P - dj%P) * flambda(di%H + dj%H)
          IF (val /= 0.d0) ocean%N(i,j) = val * (-2) * dj%H * di%H * n / ((dj%H**2 - di%H**2) * pi)
       END DO
    END DO
```

6.3.2.12 subroutine inprod_analytic::calculate_o ( ) [private]

Temperature advection term (passive scalar)

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

Definition at line 516 of file inprod_analytic.f90.

```
    INTEGER :: i,j,k
    REAL(KIND=8) :: vs3,vs4,val
    TYPE(ocean_wavenum) :: di,dj,dk
    INTEGER :: allocstat
    IF (noc == 0 ) THEN
       stop "*** Problem with calculate_O : noc==0 ! ***"
    ELSE
       IF (.NOT. ALLOCATED(ocean%O)) THEN
          ALLOCATE(ocean%O(noc,noc,noc), stat=allocstat)
          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
       END IF
    END IF
    ocean%O=0.d0
    val=0.d0

    DO i=1,noc
       DO j=i,noc
          DO k=j,noc
             di = owavenum(i)
             dj = owavenum(j)
             dk = owavenum(k)
             vs3 = s3(dj%P,dk%P,dj%H,dk%H)
             vs4 = s4(dj%P,dk%P,dj%H,dk%H)
             val = vs3*((delta(dk%H - dj%H - di%H) - delta(dk%H - dj&
                  &%H + di%H)) * delta(dk%P + dj%P - di%P) + delta(dk&
                  &%H + dj%H - di%H) * (delta(dk%P - dj%P + di%P) -&
                  & delta(dk%P - dj%P - di%P))) + vs4 * ((delta(dk%H &
                  &+ dj%H - di%H) * delta(dk%P - dj%P - di%P)) +&
                  & (delta(dk%H - dj%H + di%H) - delta(dk%H - dj%H -&
                  & di%H)) * (delta(dk%P - dj%P - di%P) - delta(dk%P &
                  &- dj%P + di%P)))
             val = val * n / 2
             IF (val /= 0.d0) THEN
```

```
549                ocean%O(i,j,k) = val
550                ocean%O(j,k,i) = val
551                ocean%O(k,i,j) = val
552                ocean%O(i,k,j) = -val
553                ocean%O(j,i,k) = -val
554                ocean%O(k,j,i) = -val
555             END IF
556          END DO
557       END DO
558    END DO
```

6.3.2.13 subroutine inprod_analytic::calculate_s ( ) [private]

Forcing (thermal) of the ocean on the atmosphere.

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

Definition at line 380 of file inprod_analytic.f90.

```
380    INTEGER :: i,j
381    TYPE(atm_wavenum) :: ti
382    TYPE(ocean_wavenum) :: dj
383    REAL(KIND=8) :: val
384    INTEGER :: allocstat
385    IF (natm == 0 ) THEN
386       stop "*** Problem with calculate_s : natm==0 ! ***"
387    ELSEIF (noc == 0) then
388       stop "*** Problem with calculate_s : noc==0 ! ***"
389    ELSE
390       IF (.NOT. ALLOCATED(atmos%s)) THEN
391          ALLOCATE(atmos%s(natm,noc), stat=allocstat)
392          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
393       END IF
394    END IF
395    atmos%s=0.d0
396
397    DO i=1,natm
398       DO j=1,noc
399          ti = awavenum(i)
400          dj = owavenum(j)
401          val=0.d0
402          IF (ti%typ == "A") THEN
403             val = flambda(dj%H) * flambda(dj%P + ti%P)
404             IF (val /= 0.d0) THEN
405                val = val*8*sqrt(2.)*dj%P/(pi**2 * (dj%P**2 - ti%P**2) * dj%H)
406             END IF
407          ELSEIF (ti%typ == "K") THEN
408             val = flambda(2 * ti%M + dj%H) * delta(dj%P - ti%P)
409             IF (val /= 0.d0) THEN
410                val = val*4*dj%H/(pi * (-4 * ti%M**2 + dj%H**2))
411             END IF
412          ELSEIF (ti%typ == "L") THEN
413             val = delta(dj%P - ti%P) * delta(2 * ti%H - dj%H)
414          END IF
415          IF (val /= 0.d0) THEN
416             atmos%s(i,j)=val
417          ENDIF
418       END DO
419    END DO
```

6.3.2.14 subroutine inprod_analytic::calculate_w ( ) [private]

Short-wave radiative forcing of the ocean.

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

Remarks

atmospheric s tensor must be computed before calling this function !

Definition at line 605 of file inprod_analytic.f90.

```
605     INTEGER :: i,j
606     INTEGER :: allocstat
607
608     IF (.NOT. ALLOCATED(atmos%s)) THEN
609        stop "*** atmos%s must be defined before calling calculate_W ! ***"
610     END IF
611
612     IF (noc == 0 ) THEN
613        stop "*** Problem with calculate_W : noc==0 ! ***"
614     ELSEIF (natm == 0 ) THEN
615        stop "*** Problem with calculate_W : natm==0 ! ***"
616     ELSE
617        IF (.NOT. ALLOCATED(ocean%W)) THEN
618           ALLOCATE(ocean%W(noc,natm), stat=allocstat)
619           IF (allocstat /= 0) stop "*** Not enough memory ! ***"
620        END IF
621     END IF
622     ocean%W=0.d0
623
624     DO i=1,noc
625        DO j=1,natm
626           ocean%W(i,j) = atmos%s(j,i)
627        END DO
628     END DO
```

6.3.2.15 subroutine, public inprod_analytic::deallocate_inprod ( )

Deallocation of the inner products.

Definition at line 722 of file inprod_analytic.f90.

```
722     INTEGER :: allocstat
723
724     ! Deallocation of atmospheric inprod
725     allocstat=0
726     IF (ALLOCATED(atmos%a)) DEALLOCATE(atmos%a, stat=allocstat)
727     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
728
729     allocstat=0
730     IF (ALLOCATED(atmos%c)) DEALLOCATE(atmos%c, stat=allocstat)
731     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
732
733     allocstat=0
734     IF (ALLOCATED(atmos%d)) DEALLOCATE(atmos%d, stat=allocstat)
735     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
736
737     allocstat=0
738     IF (ALLOCATED(atmos%s)) DEALLOCATE(atmos%s, stat=allocstat)
739     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
740
741     allocstat=0
742     IF (ALLOCATED(atmos%g)) DEALLOCATE(atmos%g, stat=allocstat)
743     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
744
745     allocstat=0
746     IF (ALLOCATED(atmos%b)) DEALLOCATE(atmos%b, stat=allocstat)
747     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
748
749     ! Deallocation of oceanic inprod
750     allocstat=0
751     IF (ALLOCATED(ocean%K)) DEALLOCATE(ocean%K, stat=allocstat)
752     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
753
754     allocstat=0
755     IF (ALLOCATED(ocean%M)) DEALLOCATE(ocean%M, stat=allocstat)
756     IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
757
758     allocstat=0
759     IF (ALLOCATED(ocean%N)) DEALLOCATE(ocean%N, stat=allocstat)
```

```
    IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"

    allocstat=0
    IF (ALLOCATED(ocean%W)) DEALLOCATE(ocean%W, stat=allocstat)
    IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"

    allocstat=0
    IF (ALLOCATED(ocean%O)) DEALLOCATE(ocean%O, stat=allocstat)
    IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"

    allocstat=0
    IF (ALLOCATED(ocean%C)) DEALLOCATE(ocean%C, stat=allocstat)
    IF (allocstat /= 0)  stop "*** Problem to deallocate ! ***"
```

6.3.2.16 real(kind=8) function inprod_analytic::delta (integer *r*) [private]

Integer Dirac delta function.

Definition at line 103 of file inprod_analytic.f90.

```
103     INTEGER :: r
104     IF (r==0) THEN
105        delta = 1.d0
106     ELSE
107        delta = 0.d0
108     ENDIF
```

6.3.2.17 real(kind=8) function inprod_analytic::flambda (integer *r*) [private]

"Odd or even" function

Definition at line 113 of file inprod_analytic.f90.

```
113     INTEGER :: r
114     IF (mod(r,2)==0) THEN
115        flambda = 0.d0
116     ELSE
117        flambda = 1.d0
118     ENDIF
```

6.3.2.18 subroutine, public inprod_analytic::init_inprod ( )

Initialisation of the inner product.

Definition at line 639 of file inprod_analytic.f90.

```
639     INTEGER :: i,j
640     INTEGER :: allocstat
641
642     ! Definition of the types and wave numbers tables
643
644     ALLOCATE(owavenum(noc),awavenum(natm), stat=allocstat)
645     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
646
647     j=0
648     DO i=1,nbatm
649        IF (ams(i,1)==1) THEN
650           awavenum(j+1)%typ='A'
651           awavenum(j+2)%typ='K'
652           awavenum(j+3)%typ='L'
653
654           awavenum(j+1)%P=ams(i,2)
```

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

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

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

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

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

      j=j+2

   ENDIF
ENDDO

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

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

ENDDO

! Computation of the atmospheric inner products tensors

CALL calculate_a
CALL calculate_g
CALL calculate_s
CALL calculate_b
CALL calculate_c_atm

! Computation of the oceanic inner products tensors

CALL calculate_m
CALL calculate_n
CALL calculate_o
CALL calculate_c_oc
CALL calculate_w
CALL calculate_k

! A last atmospheric one that needs ocean%M

CALL calculate_d
```

6.3.2.19 real(kind=8) function inprod_analytic::s1 (integer *Pj*, integer *Pk*, integer *Mj*, integer *Hk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 123 of file inprod_analytic.f90.

```
123     INTEGER :: pk,pj,mj,hk
124     s1 = -((pk * mj + pj * hk)) / 2.d0
```

6.3.2.20 real(kind=8) function inprod_analytic::s2 (integer *Pj*, integer *Pk*, integer *Mj*, integer *Hk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 129 of file inprod_analytic.f90.

```
129     INTEGER :: pk,pj,mj,hk
130     s2 = (pk * mj - pj * hk) / 2.d0
```

6.3.2.21 real(kind=8) function inprod_analytic::s3 (integer *Pj*, integer *Pk*, integer *Hj*, integer *Hk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 135 of file inprod_analytic.f90.

```
135     INTEGER :: pj,pk,hj,hk
136     s3 = (pk * hj + pj * hk) / 2.d0
```

6.3.2.22 real(kind=8) function inprod_analytic::s4 (integer *Pj*, integer *Pk*, integer *Hj*, integer *Hk*) [private]

Cehelsky & Tung Helper functions.

Definition at line 141 of file inprod_analytic.f90.

```
141     INTEGER :: pj,pk,hj,hk
142     s4 = (pk * hj - pj * hk) / 2.d0
```

6.3.3 Variable Documentation

6.3.3.1 type(atm_tensors), public inprod_analytic::atmos

Atmospheric tensors.

Definition at line 69 of file inprod_analytic.f90.

```
69   TYPE(atm_tensors), PUBLIC :: atmos
```

6.3.3.2 type(atm_wavenum), dimension(:), allocatable, public inprod_analytic::awavenum

Atmospheric blocs specification.

Definition at line 64 of file inprod_analytic.f90.

```
64   TYPE(atm_wavenum), DIMENSION(:), ALLOCATABLE, PUBLIC :: awavenum
```

6.3.3.3 type(ocean_tensors), public inprod_analytic::ocean

Oceanic tensors.

Definition at line 71 of file inprod_analytic.f90.

```
71    TYPE(ocean_tensors), PUBLIC :: ocean
```

6.3.3.4 type(ocean_wavenum), dimension(:), allocatable, public inprod_analytic::owavenum

Oceanic blocs specification.

Definition at line 66 of file inprod_analytic.f90.

```
66    TYPE(ocean_wavenum), DIMENSION(:), ALLOCATABLE, PUBLIC :: owavenum
```

6.4 params Module Reference

The model parameters module.

Functions/Subroutines

- subroutine, private init_nml

  Read the basic parameters and mode selection from the namelist.
- subroutine init_params

  Parameters initialisation routine.

Variables

- 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
- 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.
- logical writeout

  Write to file boolean.
- integer nboc

  Number of atmospheric blocks.
- integer nbatm

  Number of oceanic blocks.
- 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)
- integer, dimension(:,:), allocatable oms

  Ocean mode selection array.
- integer, dimension(:,:), allocatable ams

  Atmospheric mode selection array.

6.4.1 Detailed Description

The model parameters module.

Copyright

Remarks

Once the init_params() subroutine is called, the parameters are loaded globally in the main program and its subroutines and function

6.4.2 Function/Subroutine Documentation

6.4.2.1 subroutine, private params::init_nml ( ) [private]

Read the basic parameters and mode selection from the namelist.

Definition at line 91 of file params.f90.

```
91    INTEGER :: allocstat
92
93    namelist /aoscale/  scale,f0,n,rra,phi0_npi
94    namelist /oparams/  gp,r,h,d
95    namelist /aparams/  k,kp,sig0
96    namelist /toparams/ go,co,to0
97    namelist /taparams/ ga,ca,epsa,ta0
98    namelist /otparams/ sc,lambda,rr,sb
99
100   namelist /modeselection/ oms,ams
101   namelist /numblocs/ nboc,nbatm
102
103   namelist /int_params/ t_trans,t_run,dt,tw,writeout
104
105   OPEN(8, file="params.nml", status='OLD', recl=80, delim='APOSTROPHE')
106
107   READ(8,nml=aoscale)
108   READ(8,nml=oparams)
109   READ(8,nml=aparams)
110   READ(8,nml=toparams)
111   READ(8,nml=taparams)
112   READ(8,nml=otparams)
113
114   CLOSE(8)
115
116   OPEN(8, file="modeselection.nml", status='OLD', recl=80, delim='APOSTROPHE')
117   READ(8,nml=numblocs)
118
119   ALLOCATE(oms(nboc,2),ams(nbatm,2), stat=allocstat)
120   IF (allocstat /= 0) stop "*** Not enough memory ! ***"
121
122   READ(8,nml=modeselection)
123   CLOSE(8)
124
125   ! Must still sort modeselection here
126
127   OPEN(8, file="int_params.nml", status='OLD', recl=80, delim='APOSTROPHE')
128   READ(8,nml=int_params)
129
130
```

6.4.2.2 subroutine params::init_params ( )

Parameters initialisation routine.

Definition at line 135 of file params.f90.

```
135   INTEGER, DIMENSION(2) :: s
136   INTEGER :: i
137   CALL init_nml
138
139   !---------------------------------------------------------!
140   !                                                         !
141   ! Computation of the dimension of the atmospheric         !
142   ! and oceanic components                                  !
143   !                                                         !
144   !---------------------------------------------------------!
145
146   natm=0
147   DO i=1,nbatm
148      IF (ams(i,1)==1) THEN
149         natm=natm+3
150      ELSE
151         natm=natm+2
152      ENDIF
```

```
    ENDDO
    s=shape(oms)
    noc=s(1)

    ndim=2*natm+2*noc

    !-----------------------------------------------------------!
    !                                                           !
    ! Some general parameters (Domain, beta, gamma, coupling) !
    !                                                           !
    !-----------------------------------------------------------!

    pi=dacos(-1.d0)
    l=scale/pi
    phi0=phi0_npi*pi
    lr=sqrt(gp*h)/f0
    g=-l**2/lr**2
    betp=l/rra*cos(phi0)/sin(phi0)
    rp=r/f0
    dp=d/f0
    kd=k*2
    kdp=kp

    !-------------------------------------------------------!
    !                                                       !
    ! DERIVED QUANTITIES                                    !
    !                                                       !
    !-------------------------------------------------------!

    cpo=co/(go*f0) * rr/(f0**2*l**2)
    lpo=lambda/(go*f0)
    cpa=ca/(ga*f0) * rr/(f0**2*l**2)/2 ! Cpa acts on psi1-psi3, not on theta
    lpa=lambda/(ga*f0)
    sbpo=4*sb*to0**3/(go*f0) ! long wave radiation lost by ocean to atmosphere space
    sbpa=8*epsa*sb*ta0**3/(go*f0) ! long wave radiation from atmosphere absorbed by ocean
    lsbpo=2*epsa*sb*to0**3/(ga*f0) ! long wave radiation from ocean absorbed by atmosphere
    lsbpa=8*epsa*sb*ta0**3/(ga*f0) ! long wave radiation lost by atmosphere to space & ocea
```

6.4.3 Variable Documentation

6.4.3.1 integer, dimension(:,:), allocatable params::ams

Atmospheric mode selection array.

Definition at line 81 of file params.f90.

```
81   INTEGER, DIMENSION(:,:), ALLOCATABLE :: ams   !< Atmospheric mode selection array
```

6.4.3.2 real(kind=8) params::betp

Definition at line 67 of file params.f90.

```
67   REAL(KIND=8) :: betp      ! \f$\beta'\f$ - Non-dimensional beta parameter
```

6.4.3.3 real(kind=8) params::ca

C_a - Constant short-wave radiation of the atmosphere.

Definition at line 40 of file params.f90.

```
40   REAL(KIND=8) :: ca        !< \f$C_a\f$ - Constant short-wave radiation of the atmosphere.
```

6.4.3.4 real(kind=8) params::co

C_a - Constant short-wave radiation of the ocean.

Definition at line 38 of file params.f90.

```
38    REAL(KIND=8) :: co        !< \f$C_a\f$ - Constant short-wave radiation of the ocean.
```

6.4.3.5 real(kind=8) params::cpa

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

Remarks

Cpa acts on psi1-psi3, not on theta.

Definition at line 58 of file params.f90.

```
58    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.
```

6.4.3.6 real(kind=8) params::cpo

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

Definition at line 56 of file params.f90.

```
56    REAL(KIND=8) :: cpo       !< \f$C'_a\f$ - Non-dimensional constant short-wave radiation of the ocean.
```

6.4.3.7 real(kind=8) params::d

Merchanical coupling parameter between the ocean and the atmosphere.

Definition at line 31 of file params.f90.

```
31    REAL(KIND=8) :: d         !< Merchanical coupling parameter between the ocean and the atmosphere.
```

6.4.3.8 real(kind=8) params::dp

d' - Non-dimensional mechanical coupling parameter between the ocean and the atmosphere.

Definition at line 52 of file params.f90.

```
52    REAL(KIND=8) :: dp        !< \f$d'\f$ - Non-dimensional mechanical coupling parameter between the ocean
        and the atmosphere.
```

6.4.3.9 real(kind=8) params::dt

Integration time step.

Definition at line 71 of file params.f90.

```
REAL(KIND=8) :: dt        !< Integration time step
```

6.4.3.10 real(kind=8) params::epsa

ϵ_a - Emissivity coefficient for the grey-body atmosphere.

Definition at line 43 of file params.f90.

```
REAL(KIND=8) :: epsa      !< \f$\epsilon_a\f$ - Emissivity coefficient for the grey-body atmosphere.
```

6.4.3.11 real(kind=8) params::f0

f_0 - Coriolis parameter

Definition at line 32 of file params.f90.

```
REAL(KIND=8) :: f0        !< \f$f_0\f$ - Coriolis parameter
```

6.4.3.12 real(kind=8) params::g

γ

Definition at line 50 of file params.f90.

```
REAL(KIND=8) :: g         !< \f$\gamma\f$
```

6.4.3.13 real(kind=8) params::ga

γ_a - Specific heat capacity of the atmosphere.

Definition at line 44 of file params.f90.

```
REAL(KIND=8) :: ga        !< \f$\gamma_a\f$ - Specific heat capacity of the atmosphere.
```

6.4.3.14 real(kind=8) params::go

γ_o - Specific heat capacity of the ocean.

Definition at line 39 of file params.f90.

```
39 REAL(KIND=8) :: go        !< \f$\gamma_o\f$ - Specific heat capacity of the ocean.
```

6.4.3.15 real(kind=8) params::gp

g'Reduced gravity

Definition at line 33 of file params.f90.

```
33 REAL(KIND=8) :: gp        !< \f$g'\f$Reduced gravity
```

6.4.3.16 real(kind=8) params::h

Depth of the active water layer of the ocean.

Definition at line 34 of file params.f90.

```
34 REAL(KIND=8) :: h         !< Depth of the active water layer of the ocean.
```

6.4.3.17 real(kind=8) params::k

Bottom atmospheric friction coefficient.

Definition at line 28 of file params.f90.

```
28 REAL(KIND=8) :: k         !< Bottom atmospheric friction coefficient.
```

6.4.3.18 real(kind=8) params::kd

k_d - Non-dimensional bottom atmospheric friction coefficient.

Definition at line 53 of file params.f90.

```
53 REAL(KIND=8) :: kd        !< \f$k_d\f$ - Non-dimensional bottom atmospheric friction coefficient.
```

6.4.3.19 real(kind=8) params::kdp

k'_d - Non-dimensional internal atmospheric friction coefficient.

Definition at line 54 of file params.f90.

```
54    REAL(KIND=8) :: kdp       !< \f$k'_d\f$ - Non-dimensional internal atmospheric friction coefficient.
```

6.4.3.20 real(kind=8) params::kp

k' - Internal atmospheric friction coefficient.

Definition at line 29 of file params.f90.

```
29    REAL(KIND=8) :: kp        !< \f$k'\f$ - Internal atmospheric friction coefficient.
```

6.4.3.21 real(kind=8) params::l

L - Domain length scale

Definition at line 64 of file params.f90.

```
64    REAL(KIND=8) :: l         !< \f$L\f$ - Domain length scale
```

6.4.3.22 real(kind=8) params::lambda

λ - Sensible + turbulent heat exchange between the ocean and the atmosphere.

Definition at line 37 of file params.f90.

```
37    REAL(KIND=8) :: lambda    !< \f$\lambda\f$ - Sensible + turbulent heat exchange between the ocean and the
        atmosphere.
```

6.4.3.23 real(kind=8) params::lpa

λ'_a - Non-dimensional sensible + turbulent heat exchange from atmosphere to ocean.

Definition at line 59 of file params.f90.

```
59    REAL(KIND=8) :: lpa       !< \f$\lambda'_a\f$ - Non-dimensional sensible + turbulent heat exchange from
        atmosphere to ocean.
```

6.4.3.24 real(kind=8) params::lpo

λ'_o - Non-dimensional sensible + turbulent heat exchange from ocean to atmosphere.

Definition at line 57 of file params.f90.

```
REAL(KIND=8) :: lpo       !< \f$\lambda'_o\f$ - Non-dimensional sensible + turbulent heat exchange from
  ocean to atmosphere.
```

6.4.3.25 real(kind=8) params::lr

L_R - Rossby deformation radius

Definition at line 49 of file params.f90.

```
REAL(KIND=8) :: lr        !< \f$L_R\f$ - Rossby deformation radius
```

6.4.3.26 real(kind=8) params::lsbpa

$S'_{B,a}$ - Long wave radiation lost by atmosphere to space & ocean.

Definition at line 63 of file params.f90.

```
REAL(KIND=8) :: lsbpa     !< \f$S'_{B,a}\f$ - Long wave radiation lost by atmosphere to space & ocean.
```

6.4.3.27 real(kind=8) params::lsbpo

$S'_{B,o}$ - Long wave radiation from ocean absorbed by atmosphere.

Definition at line 62 of file params.f90.

```
REAL(KIND=8) :: lsbpo     !< \f$S'_{B,o}\f$ - Long wave radiation from ocean absorbed by atmosphere.
```

6.4.3.28 real(kind=8) params::n

$n = 2L_y/L_x$ - Aspect ratio

Definition at line 24 of file params.f90.

```
REAL(KIND=8) :: n         !< \f$n = 2 L_y / L_x\f$ - Aspect ratio
```

6.4.3.29 integer params::natm =0

Number of atmospheric basis functions.

Definition at line 77 of file params.f90.

```
77   INTEGER :: natm=0 !< Number of atmospheric basis functions
```

6.4.3.30 integer params::nbatm

Number of oceanic blocks.

Definition at line 76 of file params.f90.

```
76   INTEGER :: nbatm  !< Number of oceanic blocks
```

6.4.3.31 integer params::nboc

Number of atmospheric blocks.

Definition at line 75 of file params.f90.

```
75   INTEGER :: nboc   !< Number of atmospheric blocks
```

6.4.3.32 integer params::ndim

Number of variables (dimension of the model)

Definition at line 79 of file params.f90.

```
79   INTEGER :: ndim   !< Number of variables (dimension of the model)
```

6.4.3.33 integer params::noc =0

Number of oceanic basis functions.

Definition at line 78 of file params.f90.

```
78   INTEGER :: noc=0  !< Number of oceanic basis functions
```

6.4.3.34 integer, dimension(:,:), allocatable params::oms

Ocean mode selection array.

Definition at line 80 of file params.f90.

```
80    INTEGER, DIMENSION(:,:), ALLOCATABLE :: oms   !< Ocean mode selection array
```

6.4.3.35 real(kind=8) params::phi0

Latitude in radian.

Definition at line 25 of file params.f90.

```
25    REAL(KIND=8) :: phi0      !< Latitude in radian
```

6.4.3.36 real(kind=8) params::phi0_npi

Latitude exprimed in fraction of pi.

Definition at line 35 of file params.f90.

```
35    REAL(KIND=8) :: phi0_npi  !< Latitude exprimed in fraction of pi.
```

6.4.3.37 real(kind=8) params::pi

π

Definition at line 48 of file params.f90.

```
48    REAL(KIND=8) :: pi        !< \f$\pi\f$
```

6.4.3.38 real(kind=8) params::r

Frictional coefficient at the bottom of the ocean.

Definition at line 30 of file params.f90.

```
30    REAL(KIND=8) :: r         !< Frictional coefficient at the bottom of the ocean.
```

6.4.3.39 real(kind=8) params::rp

r' - Frictional coefficient at the bottom of the ocean.

Definition at line 51 of file params.f90.

```
51    REAL(KIND=8) :: rp         !< \f$r'\f$ - Frictional coefficient at the bottom of the ocean.
```

6.4.3.40 real(kind=8) params::rr

R - Gas constant of dry air

Definition at line 45 of file params.f90.

```
45    REAL(KIND=8) :: rr         !< \f$R\f$ - Gas constant of dry air
```

6.4.3.41 real(kind=8) params::rra

Earth radius.

Definition at line 26 of file params.f90.

```
26    REAL(KIND=8) :: rra        !< Earth radius
```

6.4.3.42 real(kind=8) params::sb

Stefan–Boltzmann constant.

Definition at line 66 of file params.f90.

```
66    REAL(KIND=8) :: sb         !< Stefan-Boltzmann constant
```

6.4.3.43 real(kind=8) params::sbpa

$\sigma'_{B,a}$ - Long wave radiation from atmosphere absorbed by ocean.

Definition at line 61 of file params.f90.

```
61    REAL(KIND=8) :: sbpa       !< \f$\sigma'_{B,a}\f$ - Long wave radiation from atmosphere absorbed by ocean.
```

6.4.3.44 real(kind=8) params::sbpo

$\sigma'_{B,o}$ - Long wave radiation lost by ocean to atmosphere & space.

Definition at line 60 of file params.f90.

```
60    REAL(KIND=8) :: sbpo       !< \f$\sigma'_{B,o}\f$ - Long wave radiation lost by ocean to atmosphere &
        space.
```

6.4.3.45 real(kind=8) params::sc

Ratio of surface to atmosphere temperature.

Definition at line 65 of file params.f90.

```
65    REAL(KIND=8) :: sc         !< Ratio of surface to atmosphere temperature.
```

6.4.3.46 real(kind=8) params::scale

$L_y = L\,\pi$ - The characteristic space scale.

Definition at line 47 of file params.f90.

```
47    REAL(KIND=8) :: scale      !< \f$L_y = L \, \pi\f$ - The characteristic space scale.
```

6.4.3.47 real(kind=8) params::sig0

σ_0 - Non-dimensional static stability of the atmosphere.

Definition at line 27 of file params.f90.

```
27    REAL(KIND=8) :: sig0       !< \f$\sigma_0\f$ - Non-dimensional static stability of the atmosphere.
```

6.4.3.48 real(kind=8) params::t_run

Effective intergration time (length of the generated trajectory)

Definition at line 70 of file params.f90.

```
70    REAL(KIND=8) :: t_run      !< Effective intergration time (length of the generated trajectory)
```

6.4.3.49 real(kind=8) params::t_trans

Transient time period.

Definition at line 69 of file params.f90.

```
69   REAL(KIND=8) :: t_trans   !< Transient time period
```

6.4.3.50 real(kind=8) params::ta0

T_a^0 - Stationary solution for the 0-th order atmospheric temperature.

Definition at line 42 of file params.f90.

```
42   REAL(KIND=8) :: ta0       !< \f$T_a^0\f$ -  Stationary solution for the 0-th order atmospheric
       temperature.
```

6.4.3.51 real(kind=8) params::to0

T_o^0 - Stationary solution for the 0-th order ocean temperature.

Definition at line 41 of file params.f90.

```
41   REAL(KIND=8) :: to0       !< \f$T_o^0\f$ -  Stationary solution for the 0-th order ocean temperature.
```

6.4.3.52 real(kind=8) params::tw

Write all variables every tw time units.

Definition at line 72 of file params.f90.

```
72   REAL(KIND=8) :: tw        !< Write all variables every tw time units
```

6.4.3.53 logical params::writeout

Write to file boolean.

Definition at line 73 of file params.f90.

```
73   LOGICAL :: writeout       !< Write to file boolean
```

6.5 rk2_integrator Module Reference

Module with the integration routines.

Functions/Subroutines

- subroutine, public init_integrator

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

  Routine computing the tendencies of the model.
- subroutine, public step (y, t, dt, res)

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

Variables

- 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.

6.5.1 Detailed Description

Module with the integration routines.

Copyright

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

Remarks

This module actually contains the Heun algorithm routines. The user can modify it according to its preferred integration scheme. For higher-order schemes, additional buffers will probably have to be defined.

6.5.2 Function/Subroutine Documentation

6.5.2.1 subroutine, public rk2_integrator::init_integrator ( )

Routine to initialise the integration buffers.

Definition at line 37 of file rk2_integrator.f90.

```
37     INTEGER :: allocstat
38     ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim) ,stat=allocstat)
39     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
```

6.5.2.2 subroutine, public rk2_integrator::step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 61 of file rk2_integrator.f90.

```
61      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
62      REAL(KIND=8), INTENT(INOUT) :: t
63      REAL(KIND=8), INTENT(IN) :: dt
64      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
65
66      CALL tendencies(t,y,buf_f0)
67      buf_y1 = y+dt*buf_f0
68      CALL tendencies(t+dt,buf_y1,buf_f1)
69      res=y+0.5*(buf_f0+buf_f1)*dt
70      t=t+dt
```

6.5.2.3 subroutine rk2_integrator::tendencies (real(kind=8), intent(in) *t*, real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(out) *res*) [private]

Routine computing the tendencies of the model.

Parameters

t	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
y	Point at which the tendencies have to be computed.
res	vector to store the result.

Remarks

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

Definition at line 49 of file rk2_integrator.f90.

```
49      REAL(KIND=8), INTENT(IN) :: t
50      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
51      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
52      CALL sparse_mul3(aotensor, y, y, res)
```

6.5.3 Variable Documentation

6.5.3.1 real(kind=8), dimension(:), allocatable rk2_integrator::buf_f0 [private]

Buffer to hold tendencies at the initial position.

Definition at line 28 of file rk2_integrator.f90.

```
28    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_f0 !< Buffer to hold tendencies at the initial position
```

6.5.3.2 real(kind=8), dimension(:), allocatable rk2_integrator::buf_f1 [private]

Buffer to hold tendencies at the intermediate position.

Definition at line 29 of file rk2_integrator.f90.

```
29    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_f1 !< Buffer to hold tendencies at the intermediate
        position
```

6.5.3.3 real(kind=8), dimension(:), allocatable rk2_integrator::buf_y1 [private]

Buffer to hold the intermediate position (Heun algorithm)

Definition at line 27 of file rk2_integrator.f90.

```
27    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1 !< Buffer to hold the intermediate position (Heun
        algorithm)
```

6.6 rk2_tl_ad_integrator Module Reference

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

Functions/Subroutines

- subroutine, public init_tl_ad_integrator

  Routine to initialise the integration buffers.
- subroutine, public ad_step (y, ystar, t, dt, res)

  Routine to perform an integration step (Heun algorithm) of the adjoint model. The incremented time is returned.
- subroutine, public tl_step (y, ystar, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable buf_y1

  Buffer to hold the intermediate position (Heun algorithm) of the tangent linear model.
- real(kind=8), dimension(:), allocatable buf_f0

  Buffer to hold tendencies at the initial position of the tangent linear model.
- real(kind=8), dimension(:), allocatable buf_f1

  Buffer to hold tendencies at the intermediate position of the tangent linear model.

6.6.1 Detailed Description

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

Copyright

Remarks

This module actually contains the Heun algorithm routines. The user can modify it according to its preferred integration scheme. For higher-order schemes, additional buffers will probably have to be defined.

6.6.2 Function/Subroutine Documentation

6.6.2.1 subroutine, public rk2_tl_ad_integrator::ad_step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
ystar	Adjoint model at the point ystar.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 61 of file rk2_tl_ad_integrator.f90.

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

CALL ad(t,ystar,y,buf_f0)
buf_y1 = y+dt*buf_f0
CALL ad(t+dt,ystar,buf_y1,buf_f1)
res=y+0.5*(buf_f0+buf_f1)*dt
t=t+dt
```

6.6.2.2 subroutine, public rk2_tl_ad_integrator::init_tl_ad_integrator ( )

Routine to initialise the integration buffers.

Definition at line 41 of file rk2_tl_ad_integrator.f90.

```
INTEGER :: allocstat
ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim),stat=allocstat)
IF (allocstat /= 0) stop "*** Not enough memory ! ***"
```

6.6.2.3 subroutine, public rk2_tl_ad_integrator::tl_step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
ystar	Adjoint model at the point ystar.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 86 of file rk2_tl_ad_integrator.f90.

```
86     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ystar
87     REAL(KIND=8), INTENT(INOUT) :: t
88     REAL(KIND=8), INTENT(IN) :: dt
89     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
90
91     CALL tl(t,ystar,y,buf_f0)
92     buf_y1 = y+dt*buf_f0
93     CALL tl(t+dt,ystar,buf_y1,buf_f1)
94     res=y+0.5*(buf_f0+buf_f1)*dt
95     t=t+dt
```

6.6.3 Variable Documentation

6.6.3.1 real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_f0 [private]

Buffer to hold tendencies at the initial position of the tangent linear model.

Definition at line 31 of file rk2_tl_ad_integrator.f90.

```
31   REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_f0 !< Buffer to hold tendencies at the initial position of
       the tangent linear model
```

6.6.3.2 real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_f1 [private]

Buffer to hold tendencies at the intermediate position of the tangent linear model.

Definition at line 32 of file rk2_tl_ad_integrator.f90.

```
32   REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_f1 !< Buffer to hold tendencies at the intermediate
       position of the tangent linear model
```

6.6.3.3 real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_y1 [private]

Buffer to hold the intermediate position (Heun algorithm) of the tangent linear model.

Definition at line 30 of file rk2_tl_ad_integrator.f90.

```
30   REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1 !< Buffer to hold the intermediate position (Heun
       algorithm) of the tangent linear model
```

6.7 rk4_integrator Module Reference

Module with the RK4 integration routines.

Functions/Subroutines

- subroutine, public init_integrator

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

    Routine computing the tendencies of the model.
- subroutine, public step (y, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable buf_y1

    Buffer to hold the intermediate position (Heun algorithm)
- real(kind=8), dimension(:), allocatable buf_ka

    Buffer A to hold tendencies.
- real(kind=8), dimension(:), allocatable buf_kb

    Buffer B to hold tendencies.

6.7.1 Detailed Description

Module with the RK4 integration routines.

Copyright

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

Remarks

This module actually contains the RK4 algorithm routines. The user can modify it according to its preferred integration scheme. For higher-order schemes, additional buffers will probably have to be defined.

6.7.2 Function/Subroutine Documentation

6.7.2.1 subroutine, public rk4_integrator::init_integrator ( )

Routine to initialise the integration buffers.

Definition at line 37 of file rk4_integrator.f90.

```
37      INTEGER :: allocstat
38      ALLOCATE(buf_y1(0:ndim),buf_ka(0:ndim),buf_kb(0:ndim) ,stat=allocstat)
39      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
```

6.7.2.2 subroutine, public rk4_integrator::step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 61 of file rk4_integrator.f90.

```
61      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
62      REAL(KIND=8), INTENT(INOUT) :: t
63      REAL(KIND=8), INTENT(IN) :: dt
64      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
65
66      CALL tendencies(t,y,buf_ka)
67      buf_y1 = y + 0.5*dt*buf_ka
68
69      CALL tendencies(t+0.5*dt,buf_y1,buf_kb)
70      buf_y1 = y + 0.5*dt*buf_kb
71      buf_ka = buf_ka + 2*buf_kb
72
73      CALL tendencies(t+0.5*dt,buf_y1,buf_kb)
74      buf_y1 = y + dt*buf_kb
75      buf_ka = buf_ka + 2*buf_kb
76
77      CALL tendencies(t+dt,buf_y1,buf_kb)
78      buf_ka = buf_ka + buf_kb
79
80      t=t+dt
81      res=y+buf_ka*dt/6
```

6.7.2.3 subroutine rk4_integrator::tendencies (real(kind=8), intent(in) *t*, real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(out) *res*) [private]

Routine computing the tendencies of the model.

Parameters

t	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
y	Point at which the tendencies have to be computed.
res	vector to store the result.

Remarks

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

Definition at line 49 of file rk4_integrator.f90.

```
49      REAL(KIND=8), INTENT(IN) :: t
50      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
51      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
52      CALL sparse_mul3(aotensor, y, y, res)
```

6.7.3 Variable Documentation

6.7.3.1 real(kind=8), dimension(:), allocatable rk4_integrator::buf_ka [private]

Buffer A to hold tendencies.

Definition at line 28 of file rk4_integrator.f90.

```
28 REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_ka !< Buffer A to hold tendencies
```

6.7.3.2 real(kind=8), dimension(:), allocatable rk4_integrator::buf_kb [private]

Buffer B to hold tendencies.

Definition at line 29 of file rk4_integrator.f90.

```
29 REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_kb !< Buffer B to hold tendencies
```

6.7.3.3 real(kind=8), dimension(:), allocatable rk4_integrator::buf_y1 [private]

Buffer to hold the intermediate position (Heun algorithm)

Definition at line 27 of file rk4_integrator.f90.

```
27 REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1 !< Buffer to hold the intermediate position (Heun
     algorithm)
```

6.8 rk4_tl_ad_integrator Module Reference

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

Functions/Subroutines

- subroutine, public init_tl_ad_integrator
  Routine to initialise the TL-AD integration bufers.
- subroutine, public ad_step (y, ystar, t, dt, res)
  Routine to perform an integration step (RK4 algorithm) of the adjoint model. The incremented time is returned.
- subroutine, public tl_step (y, ystar, t, dt, res)
  Routine to perform an integration step (RK4 algorithm) of the tangent linear model. The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable buf_y1
  Buffer to hold the intermediate position of the tangent linear model.
- real(kind=8), dimension(:), allocatable buf_ka
  Buffer to hold tendencies in the RK4 scheme for the tangent linear model.
- real(kind=8), dimension(:), allocatable buf_kb
  Buffer to hold tendencies in the RK4 scheme for the tangent linear model.

6.8.1 Detailed Description

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

Copyright

Remarks

This module actually contains the RK4 algorithm routines. The user can modify it according to its preferred integration scheme. For higher-order schemes, additional bufers will probably have to be defined.

6.8.2 Function/Subroutine Documentation

6.8.2.1 subroutine, public rk4_tl_ad_integrator::ad_step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
ystar	Adjoint model at the point ystar.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 65 of file rk4_tl_ad_integrator.f90.

```
65      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ystar
66      REAL(KIND=8), INTENT(INOUT) :: t
67      REAL(KIND=8), INTENT(IN) :: dt
68      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
69
70      CALL ad(t,ystar,y,buf_ka)
71      buf_y1 = y+0.5*dt*buf_ka
72      CALL ad(t+0.5*dt,ystar,buf_y1,buf_kb)
73      buf_y1 = y+0.5*dt*buf_kb
74      buf_ka = buf_ka+2*buf_kb
75      CALL ad(t+0.5*dt,ystar,buf_y1,buf_kb)
76      buf_y1 = y+0.5*dt*buf_kb
77      buf_ka = buf_ka+2*buf_kb
78      CALL ad(t+dt,ystar,buf_y1,buf_kb)
79      buf_ka = buf_ka+buf_kb
80      res=y+buf_ka*dt/6
81      t=t+dt
```

6.8.2.2 subroutine, public rk4_tl_ad_integrator::init_tl_ad_integrator ( )

Routine to initialise the TL-AD integration bufers.

Definition at line 43 of file rk4_tl_ad_integrator.f90.

```
43      INTEGER :: allocstat
44      ALLOCATE(buf_y1(0:ndim),buf_ka(0:ndim),buf_kb(0:ndim),stat=allocstat)
45      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
```

6.8.2.3 subroutine, public rk4_tl_ad_integrator::tl_step (real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), intent(inout) *t*, real(kind=8), intent(in) *dt*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

y	Initial point.
ystar	Adjoint model at the point ystar.
t	Actual integration time
dt	Integration timestep.
res	Final point after the step.

Definition at line 98 of file rk4_tl_ad_integrator.f90.

```
98      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ystar
99      REAL(KIND=8), INTENT(INOUT) :: t
100      REAL(KIND=8), INTENT(IN) :: dt
101      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
102
103      CALL tl(t,ystar,y,buf_ka)
104      buf_y1 = y+0.5*dt*buf_ka
105      CALL tl(t+0.5*dt,ystar,buf_y1,buf_kb)
106      buf_y1 = y+0.5*dt*buf_kb
107      buf_ka = buf_ka+2*buf_kb
108      CALL tl(t+0.5*dt,ystar,buf_y1,buf_kb)
109      buf_y1 = y+0.5*dt*buf_kb
110      buf_ka = buf_ka+2*buf_kb
111      CALL tl(t+dt,ystar,buf_y1,buf_kb)
112      buf_ka = buf_ka+buf_kb
113      res=y+buf_ka*dt/6
114      t=t+dt
```

6.8.3 Variable Documentation

6.8.3.1 real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_ka [private]

Buffer to hold tendencies in the RK4 scheme for the tangent linear model.

Definition at line 33 of file rk4_tl_ad_integrator.f90.

```
33    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_ka !< Buffer to hold tendencies in the RK4 scheme for the
        tangent linear model
```

6.8.3.2 real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_kb [private]

Buffer to hold tendencies in the RK4 scheme for the tangent linear model.

Definition at line 34 of file rk4_tl_ad_integrator.f90.

```
34    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_kb !< Buffer to hold tendencies in the RK4 scheme for the
        tangent linear model
```

6.8.3.3 real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_y1 [private]

Buffer to hold the intermediate position of the tangent linear model.

Definition at line 32 of file rk4_tl_ad_integrator.f90.

```
32    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1 !< Buffer to hold the intermediate position of the
        tangent linear model
```

6.9 stat Module Reference

Statistics accumulators.

Functions/Subroutines

- subroutine, public init_stat

  Initialise the accumulators.
- subroutine, public acc (x)

  Accumulate one state.
- real(kind=8) function, dimension(0:ndim), public mean ()

  Function returning the mean.
- real(kind=8) function, dimension(0:ndim), public var ()

  Function returning the variance.
- integer function, public iter ()

  Function returning the number of data accumulated.
- subroutine, public reset

  Routine resetting the accumulators.

Variables

- 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.
- real(kind=8), dimension(:), allocatable mtmp

6.9.1 Detailed Description

Statistics accumulators.

Copyright

6.9.2 Function/Subroutine Documentation

6.9.2.1 subroutine, public stat::acc (real(kind=8), dimension(0:ndim), intent(in) *x*)

Accumulate one state.

Definition at line 48 of file stat.f90.

```
48        IMPLICIT NONE
49        REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: x
50        i=i+1
51        mprev=m+(x-m)/i
52        mtmp=mprev
53        mprev=m
54        m=mtmp
55        v=v+(x-mprev)*(x-m)
```

6.9.2.2 subroutine, public stat::init_stat ( )

Initialise the accumulators.

Definition at line 35 of file stat.f90.

```
35        INTEGER :: allocstat
36
37        ALLOCATE(m(0:ndim),mprev(0:ndim),v(0:ndim),mtmp(0:ndim), stat=allocstat)
38        IF (allocstat /= 0) stop '*** Not enough memory ***'
39        m=0.d0
40        mprev=0.d0
41        v=0.d0
42        mtmp=0.d0
43
```

6.9.2.3 integer function, public stat::iter ( )

Function returning the number of data accumulated.

Definition at line 72 of file stat.f90.

```
72        INTEGER :: iter
73        iter=i
```

6.9.2.4 real(kind=8) function, dimension(0:ndim), public stat::mean ( )

Function returning the mean.

Definition at line 60 of file stat.f90.

```
60        REAL(KIND=8), DIMENSION(0:ndim) :: mean
61        mean=m
```

6.9.2.5 subroutine, public stat::reset ( )

Routine resetting the accumulators.

Definition at line 78 of file stat.f90.

```
78        m=0.d0
79        mprev=0.d0
80        v=0.d0
81        i=0
```

6.9.2.6 real(kind=8) function, dimension(0:ndim), public stat::var ( )

Function returning the variance.

Definition at line 66 of file stat.f90.

```
66        REAL(KIND=8), DIMENSION(0:ndim) :: var
67        var=v/(i-1)
```

6.9.3 Variable Documentation

6.9.3.1 integer stat::i =0 [private]

Number of stats accumulated.

Definition at line 20 of file stat.f90.

```
20    INTEGER :: i=0 !< Number of stats accumulated
```

6.9.3.2 real(kind=8), dimension(:), allocatable stat::m [private]

Vector storing the inline mean.

Definition at line 23 of file stat.f90.

```
23    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: m       !< Vector storing the inline mean
```

6.9.3.3 real(kind=8), dimension(:), allocatable stat::mprev [private]

Previous mean vector.

Definition at line 24 of file stat.f90.

```
24    REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: mprev   !< Previous mean vector
```

6.9.3.4 real(kind=8), dimension(:), allocatable stat::mtmp [private]

Definition at line 26 of file stat.f90.

```
26  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: mtmp
```

6.9.3.5 real(kind=8), dimension(:), allocatable stat::v [private]

Vector storing the inline variance.

Definition at line 25 of file stat.f90.

```
25  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: v        !< Vector storing the inline variance
```

6.10 tensor Module Reference

Tensor utility module.

Data Types

- type coolist

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

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

Functions/Subroutines

- subroutine, public copy_tensor (src, dst)

  Routine to copy a tensor.
- subroutine, public mat_to_coo (src, dst)

  Routine to convert a matrix to a tensor.
- subroutine, public sparse_mul3 (coolist_ijk, 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, public jsparse_mul (coolist_ijk, arr_j, jcoo_ij)

  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 coolist (sparse tensor).
- subroutine, public jsparse_mul_mat (coolist_ijk, arr_j, jcoo_ij)

Sparse multiplication of two tensors to determine the Jacobian:

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

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, public sparse_mul2 (coolist_ij, arr_j, res)

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

- subroutine, public simplify (tensor)

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

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

.

Variables

- real(kind=8), parameter real_eps = 2.2204460492503131e-16

  Parameter to test the equality with zero.

6.10.1 Detailed Description

Tensor utility module.

Copyright

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

6.10.2 Function/Subroutine Documentation

6.10.2.1 subroutine, public tensor::copy_tensor (type(coolist), dimension(ndim), intent(in) *src*, type(coolist), dimension(ndim), intent(out) *dst*)

Routine to copy a tensor.

Parameters

src	Source tensor
dst	Destination tensor

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 44 of file tensor.f90.

```fortran
   TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
   TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
   INTEGER :: i,j,allocstat

   DO i=1,ndim
      IF (dst(i)%nelems/=0) stop "*** copy_tensor : Destination coolist not empty ! ***"
      ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
      DO j=1,src(i)%nelems
         dst(i)%elems(j)%j=src(i)%elems(j)%j
         dst(i)%elems(j)%k=src(i)%elems(j)%k
         dst(i)%elems(j)%v=src(i)%elems(j)%v
      ENDDO
      dst(i)%nelems=src(i)%nelems
   ENDDO
```

6.10.2.2 subroutine, public tensor::jsparse_mul (type(coolist), dimension(ndim), intent(in) *coolist_ijk*, real(kind=8), dimension(0:ndim), intent(in) *arr_j*, type(coolist), dimension(ndim), intent(out) *jcoo_ij*)

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 coolist (sparse tensor).

Parameters

coolist←_ijk	a coordinate list (sparse tensor) of which index 2 or 3 will be contracted.
arr_j	the vector to be contracted with index 2 and then index 3 of ffi_coo_ijk
jcoo_ij	a coolist (sparse tensor) to store the result of the contraction

Definition at line 123 of file tensor.f90.

```fortran
123    TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
124    TYPE(coolist), DIMENSION(ndim), INTENT(OUT):: jcoo_ij
125    REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j
126    REAL(KIND=8) :: v
127    INTEGER :: i,j,k,n,nj,allocstat
128    DO i=1,ndim
129       IF (jcoo_ij(i)%nelems/=0) stop "*** jsparse_mul : Destination coolist not empty ! ***"
130       nj=2*coolist_ijk(i)%nelems
131       ALLOCATE(jcoo_ij(i)%elems(nj), stat=allocstat)
132       IF (allocstat /= 0) stop "*** Not enough memory ! ***"
133       nj=0
134       DO n=1,coolist_ijk(i)%nelems
135          j=coolist_ijk(i)%elems(n)%j
136          k=coolist_ijk(i)%elems(n)%k
137          v=coolist_ijk(i)%elems(n)%v
138          IF (j /=0) THEN
139             nj=nj+1
140             jcoo_ij(i)%elems(nj)%j=j
141             jcoo_ij(i)%elems(nj)%k=0
142             jcoo_ij(i)%elems(nj)%v=v*arr_j(k)
143          END IF
144
145          IF (k /=0) THEN
146             nj=nj+1
147             jcoo_ij(i)%elems(nj)%j=k
148             jcoo_ij(i)%elems(nj)%k=0
149             jcoo_ij(i)%elems(nj)%v=v*arr_j(j)
150          END IF
151       END DO
152       jcoo_ij(i)%nelems=nj
153    END DO
```

6.10.2.3 subroutine, public tensor::jsparse_mul_mat (type(coolist), dimension(ndim), intent(in) *coolist_ijk,* real(kind=8), dimension(0:ndim), intent(in) *arr_j,* real(kind=8), dimension(ndim,ndim), intent(out) *jcoo_ij*)

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

coolist↩_ijk	a coordinate list (sparse tensor) of which index 2 or 3 will be contracted.
arr_j	the vector to be contracted with index 2 and then index 3 of ffi_coo_ijk
jcoo_ij	a matrix to store the result of the contraction

Definition at line 166 of file tensor.f90.

```
166     TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
167     REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT):: jcoo_ij
168     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)   :: arr_j
169     REAL(KIND=8) :: v
170     INTEGER :: i,j,k,n
171     jcoo_ij=0.d0
172     DO i=1,ndim
173        DO n=1,coolist_ijk(i)%nelems
174           j=coolist_ijk(i)%elems(n)%j
175           k=coolist_ijk(i)%elems(n)%k
176           v=coolist_ijk(i)%elems(n)%v
177           IF (j /=0) jcoo_ij(i,j)=jcoo_ij(i,j)+v*arr_j(k)
178           IF (k /=0) jcoo_ij(i,k)=jcoo_ij(i,k)+v*arr_j(j)
179        END DO
180     END DO
```

6.10.2.4 subroutine, public tensor::mat_to_coo (real(kind=8), dimension(0:ndim,0:ndim), intent(in) *src,* type(coolist), dimension(ndim), intent(out) *dst*)

Routine to convert a matrix to a tensor.

Parameters

src	Source matrix
dst	Destination tensor

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 66 of file tensor.f90.

```
    REAL(KIND=8), DIMENSION(0:ndim,0:ndim), INTENT(IN) :: src
    TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
    INTEGER :: i,j,n,allocstat
    DO i=1,ndim
       n=0
       DO j=1,ndim
          IF (abs(src(i,j))>real_eps) n=n+1
       ENDDO
       IF (dst(i)%nelems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
       ALLOCATE(dst(i)%elems(n), stat=allocstat)
       IF (allocstat /= 0) stop "*** Not enough memory ! ***"
       n=0
       DO j=1,ndim
          IF (abs(src(i,j))>real_eps) THEN
             n=n+1
             dst(i)%elems(n)%j=j
             dst(i)%elems(n)%k=0
             dst(i)%elems(n)%v=src(i,j)
          ENDIF
       ENDDO
       dst(i)%nelems=n
    ENDDO
```

6.10.2.5 subroutine, public tensor::simplify (type(coolist), dimension(ndim), intent(inout) *tensor*)

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

tensor	a coordinate list (sparse tensor) which will be simplified.

Definition at line 208 of file tensor.f90.

```
208    TYPE(coolist), DIMENSION(ndim), INTENT(INOUT):: tensor
209    INTEGER :: i,j,k
210    INTEGER :: li,lii,liii,n
211    DO i= 1,ndim
212       n=tensor(i)%nelems
213       DO li=n,2,-1
214          j=tensor(i)%elems(li)%j
215          k=tensor(i)%elems(li)%k
216          DO lii=li-1,1,-1
217             IF ((j==tensor(i)%elems(lii)%j).AND.(k==tensor(i)%elems(lii)%k)) THEN
218                ! Found another entry with the same i,j,k: merge both into
219                ! the one listed first (of those two).
220                tensor(i)%elems(lii)%v=tensor(i)%elems(lii)%v+tensor(i)%elems(li)%v
221                ! Shift the rest of the items one place down.
222                DO liii=li+1,n
223                   tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
224                   tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
225                   tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
226                END DO
227                tensor(i)%nelems=tensor(i)%nelems-1
228                ! Here we should stop because the li no longer points to the
229                ! original i,j,k element
230                EXIT
231             ENDIF
232          ENDDO
233       ENDDO
234       n=tensor(i)%nelems
235       DO li=1,n
236          ! Clear new "almost" zero entries and shift rest of the items one place down.
237          ! Make sure not to skip any entries while shifting!
238          DO WHILE (abs(tensor(i)%elems(li)%v) < real_eps)
239             DO liii=li+1,n
240                tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
241                tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
242                tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
243             ENDDO
```

```
244             tensor(i)%nelems=tensor(i)%nelems-1
245           ENDDO
246         ENDDO
247 
248     ENDDO
```

6.10.2.6 subroutine, public tensor::sparse_mul2 (type(coolist), dimension(ndim), intent(in) *coolist_ij*, real(kind=8), dimension(0:ndim), intent(in) *arr_j*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

coolist←_ij	a coordinate list (sparse tensor) of which index 2 will be contracted.
arr_j	the vector to be contracted with index 2 of coolist_ijk
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 191 of file tensor.f90.

```
191     TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ij
192     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j
193     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
194     INTEGER :: i,j,n
195     res=0.d0
196     DO i=1,ndim
197        DO n=1,coolist_ij(i)%nelems
198          j=coolist_ij(i)%elems(n)%j
199          res(i) = res(i) + coolist_ij(i)%elems(n)%v * arr_j(j)
200       END DO
201     END DO
```

6.10.2.7 subroutine, public tensor::sparse_mul3 (type(coolist), dimension(ndim), intent(in) *coolist_ijk*, real(kind=8), dimension(0:ndim), intent(in) *arr_j*, real(kind=8), dimension(0:ndim), intent(in) *arr_k*, real(kind=8), dimension(0:ndim), intent(out) *res*)

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

Parameters

coolist←_ijk	a coordinate list (sparse tensor) of which index 2 and 3 will be contracted.
arr_j	the vector to be contracted with index 2 of coolist_ijk
arr_k	the vector to be contracted with index 3 of coolist_ijk
res	vector (buffer) to store the result of the contraction

Remarks

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

Definition at line 99 of file tensor.f90.

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

6.10.3 Variable Documentation

6.10.3.1 real(kind=8), parameter tensor::real_eps = 2.2204460492503131e-16

Parameter to test the equality with zero.

Definition at line 33 of file tensor.f90.

```
REAL(KIND=8), PARAMETER :: real_eps = 2.2204460492503131e-16
```

6.11 tl_ad_tensor Module Reference

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

Functions/Subroutines

- type(coolist) function, dimension(ndim) jacobian (ystar)

  Compute the Jacobian of MAOOAM in point ystar.
- real(kind=8) function, dimension(ndim, ndim) jacobian_mat (ystar)

  Compute the Jacobian of MAOOAM in point ystar.
- subroutine, public init_tltensor

  Routine to initialize the TL tensor.
- subroutine compute_tltensor (func)

  Routine to compute the TL tensor from the original MAOOAM one.
- subroutine tl_add_count (i, j, k, v)

  Subroutine used to count the number of TL tensor entries.
- subroutine tl_coeff (i, j, k, v)

  Subroutine used to compute the TL tensor entries.
- subroutine, public init_adtensor

  Routine to initialize the AD tensor.
- subroutine compute_adtensor (func)

  Subroutine to compute the AD tensor from the original MAOOAM one.
- subroutine ad_add_count (i, j, k, v)

Subroutine used to count the number of AD tensor entries.

- subroutine ad_coeff (i, j, k, v)
- subroutine, public init_adtensor_ref

  Alternate method to initialize the AD tensor from the TL tensor.
- subroutine compute_adtensor_ref (func)

  Alternate subroutine to compute the AD tensor from the TL one.
- subroutine ad_add_count_ref (i, j, k, v)

  Alternate subroutine used to count the number of AD tensor entries from the TL tensor.
- subroutine ad_coeff_ref (i, j, k, v)

  Alternate subroutine used to compute the AD tensor entries from the TL tensor.
- subroutine, public ad (t, ystar, deltay, buf)

  Tendencies for the AD of MAOOAM in point ystar for perturbation deltay.
- subroutine, public tl (t, ystar, deltay, buf)

  Tendencies for the TL of MAOOAM in point ystar for perturbation deltay.

Variables

- real(kind=8), parameter real_eps = 2.2204460492503131e-16

  Epsilon to test equality with 0.
- integer, dimension(:), allocatable count_elems

  Vector used to count the tensor elements.
- type(coolist), dimension(:), allocatable, public tltensor

  Tensor representation of the Tangent Linear tendencies.
- type(coolist), dimension(:), allocatable, public adtensor

  Tensor representation of the Adjoint tendencies.

6.11.1 Detailed Description

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

Copyright

2016 Lesley De Cruz & Jonathan Demaeyer. See LICENSE.txt for license information.

Remarks

The routines of this module should be called only after params::init_params() and aotensor_def::init_↩ aotensor() have been called !

6.11.2 Function/Subroutine Documentation

6.11.2.1 subroutine, public tl_ad_tensor::ad (real(kind=8), intent(in) *t*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), dimension(0:ndim), intent(in) *deltay*, real(kind=8), dimension(0:ndim), intent(out) *buf*)

Tendencies for the AD of MAOOAM in point ystar for perturbation deltay.

Parameters

t	time
ystar	vector with the variables (current point in trajectory)
deltay	vector with the perturbation of the variables at time t
buf	vector (buffer) to store derivatives.

Definition at line 384 of file tl_ad_tensor.f90.

```
384     REAL(KIND=8), INTENT(IN) :: t
385     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: ystar,deltay
386     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: buf
387     CALL sparse_mul3(adtensor,deltay,ystar,buf)
```

6.11.2.2 subroutine tl_ad_tensor::ad_add_count (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Subroutine used to count the number of AD tensor entries.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value that will be added

Definition at line 243 of file tl_ad_tensor.f90.

```
243     INTEGER, INTENT(IN) :: i,j,k
244     REAL(KIND=8), INTENT(IN)  :: v
245     IF ((abs(v) .ge. real_eps).AND.(i /= 0)) THEN
246        IF (k /= 0) count_elems(k)=count_elems(k)+1
247        IF (j /= 0) count_elems(j)=count_elems(j)+1
248     ENDIF
```

6.11.2.3 subroutine tl_ad_tensor::ad_add_count_ref (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Alternate subroutine used to count the number of AD tensor entries from the TL tensor.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value that will be added

Definition at line 346 of file tl_ad_tensor.f90.

```
346     INTEGER, INTENT(IN) :: i,j,k
```

```
REAL(KIND=8), INTENT(IN)  :: v
IF ((abs(v) .ge. real_eps).AND.(j /= 0)) count_elems(j)=count_elems(j)+1
```

6.11.2.4 subroutine tl_ad_tensor::ad_coeff (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value to add

Definition at line 257 of file tl_ad_tensor.f90.

```
INTEGER, INTENT(IN) :: i,j,k
REAL(KIND=8), INTENT(IN) :: v
INTEGER :: n
IF (.NOT. ALLOCATED(adtensor)) stop "*** ad_coeff routine : tensor not yet allocated ***"
IF ((abs(v) .ge. real_eps).AND.(i /=0)) THEN
   IF (k /=0) THEN
      IF (.NOT. ALLOCATED(adtensor(k)%elems)) stop "*** ad_coeff routine : tensor not yet allocated
***"
      n=(adtensor(k)%nelems)+1
      adtensor(k)%elems(n)%j=i
      adtensor(k)%elems(n)%k=j
      adtensor(k)%elems(n)%v=v
      adtensor(k)%nelems=n
   END IF
   IF (j /=0) THEN
      IF (.NOT. ALLOCATED(adtensor(j)%elems)) stop "*** ad_coeff routine : tensor not yet allocated
***"
      n=(adtensor(j)%nelems)+1
      adtensor(j)%elems(n)%j=i
      adtensor(j)%elems(n)%k=k
      adtensor(j)%elems(n)%v=v
      adtensor(j)%nelems=n
   END IF
END IF
```

6.11.2.5 subroutine tl_ad_tensor::ad_coeff_ref (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Alternate subroutine used to compute the AD tensor entries from the TL tensor.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value to add

Definition at line 358 of file tl_ad_tensor.f90.

```
INTEGER, INTENT(IN) :: i,j,k
REAL(KIND=8), INTENT(IN) :: v
INTEGER :: n
```

```
    IF (.NOT. ALLOCATED(adtensor)) stop "*** ad_coeff_ref routine : tensor not yet allocated ***"
    IF ((abs(v) .ge. real_eps).AND.(j /=0)) THEN
       IF (.NOT. ALLOCATED(adtensor(j)%elems)) stop "*** ad_coeff_ref routine : tensor not yet allocated
  ***"
       n=(adtensor(j)%nelems)+1
       adtensor(j)%elems(n)%j=i
       adtensor(j)%elems(n)%k=k
       adtensor(j)%elems(n)%v=v
       adtensor(j)%nelems=n
    END IF
```

6.11.2.6 subroutine tl_ad_tensor::compute_adtensor (external *func*) [private]

Subroutine to compute the AD tensor from the original MAOOAM one.

Parameters

func	subroutine used to do the computation

Definition at line 217 of file tl_ad_tensor.f90.

6.11.2.7 subroutine tl_ad_tensor::compute_adtensor_ref (external *func*) [private]

Alternate subroutine to compute the AD tensor from the TL one.

Parameters

func	subroutine used to do the computation

Definition at line 318 of file tl_ad_tensor.f90.

6.11.2.8 subroutine tl_ad_tensor::compute_tltensor (external *func*) [private]

Routine to compute the TL tensor from the original MAOOAM one.

Parameters

func	subroutine used to do the computation

Definition at line 121 of file tl_ad_tensor.f90.

6.11.2.9 subroutine, public tl_ad_tensor::init_adtensor ( )

Routine to initialize the AD tensor.

Definition at line 193 of file tl_ad_tensor.f90.

```
193     INTEGER :: i
194     INTEGER :: allocstat
195     ALLOCATE(adtensor(ndim),count_elems(ndim), stat=allocstat)
196     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
197     count_elems=0
198     CALL compute_adtensor(ad_add_count)
199
200     DO i=1,ndim
201        ALLOCATE(adtensor(i)%elems(count_elems(i)), stat=allocstat)
202        IF (allocstat /= 0) stop "*** Not enough memory ! ***"
203     END DO
204
205     DEALLOCATE(count_elems, stat=allocstat)
206     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
207
208     CALL compute_adtensor(ad_coeff)
209
210     CALL simplify(adtensor)
211
```

6.11.2.10 subroutine, public tl_ad_tensor::init_adtensor_ref ( )

Alternate method to initialize the AD tensor from the TL tensor.

Remarks

The tltensor must be initialised before using this method.

Definition at line 294 of file tl_ad_tensor.f90.

```
294     INTEGER :: i
295     INTEGER :: allocstat
296     ALLOCATE(adtensor(ndim),count_elems(ndim), stat=allocstat)
297     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
298     count_elems=0
299     CALL compute_adtensor_ref(ad_add_count_ref)
300
301     DO i=1,ndim
302        ALLOCATE(adtensor(i)%elems(count_elems(i)), stat=allocstat)
303        IF (allocstat /= 0) stop "*** Not enough memory ! ***"
304     END DO
305
306     DEALLOCATE(count_elems, stat=allocstat)
307     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
308
309     CALL compute_adtensor_ref(ad_coeff_ref)
310
311     CALL simplify(adtensor)
312
```

6.11.2.11 subroutine, public tl_ad_tensor::init_tltensor ( )

Routine to initialize the TL tensor.

Definition at line 97 of file tl_ad_tensor.f90.

```
97      INTEGER :: i
98      INTEGER :: allocstat
99      ALLOCATE(tltensor(ndim),count_elems(ndim), stat=allocstat)
100     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
101     count_elems=0
102     CALL compute_tltensor(tl_add_count)
103
104     DO i=1,ndim
105        ALLOCATE(tltensor(i)%elems(count_elems(i)), stat=allocstat)
106        IF (allocstat /= 0) stop "*** Not enough memory ! ***"
107     END DO
108
109     DEALLOCATE(count_elems, stat=allocstat)
110     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
111
112     CALL compute_tltensor(tl_coeff)
113
114     CALL simplify(tltensor)
115
```

6.11.2.12 type(coolist) function, dimension(ndim) tl_ad_tensor::jacobian (real(kind=8), dimension(0:ndim), intent(in) *ystar*) [private]

Compute the Jacobian of MAOOAM in point ystar.

Parameters

ystar	array with variables in which the jacobian should be evaluated.

Returns

Jacobian in coolist-form (table of tuples {i,j,0,value})

Definition at line 75 of file tl_ad_tensor.f90.

```
REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: ystar
TYPE(coolist), DIMENSION(ndim) :: jacobian
CALL jsparse_mul(aotensor,ystar,jacobian)
```

6.11.2.13 real(kind=8) function, dimension(ndim,ndim) tl_ad_tensor::jacobian_mat (real(kind=8), dimension(0:ndim), intent(in) *ystar*) [private]

Compute the Jacobian of MAOOAM in point ystar.

Parameters

ystar	array with variables in which the jacobian should be evaluated.

Returns

Jacobian in matrix form

Definition at line 84 of file tl_ad_tensor.f90.

```
REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: ystar
REAL(KIND=8), DIMENSION(ndim,ndim) :: jacobian_mat
CALL jsparse_mul_mat(aotensor,ystar,jacobian_mat)
```

6.11.2.14 subroutine, public tl_ad_tensor::tl (real(kind=8), intent(in) *t*, real(kind=8), dimension(0:ndim), intent(in) *ystar*, real(kind=8), dimension(0:ndim), intent(in) *deltay*, real(kind=8), dimension(0:ndim), intent(out) *buf*)

Tendencies for the TL of MAOOAM in point ystar for perturbation deltay.

Parameters

t	time
ystar	vector with the variables (current point in trajectory)
deltay	vector with the perturbation of the variables at time t
buf	vector (buffer) to store derivatives.

Definition at line 396 of file tl_ad_tensor.f90.

```
396     REAL(KIND=8), INTENT(IN) :: t
397     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: ystar,deltay
398     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: buf
399     CALL sparse_mul3(tltensor,deltay,ystar,buf)
```

6.11.2.15 subroutine tl_ad_tensor::tl_add_count (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Subroutine used to count the number of TL tensor entries.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value that will be added

Definition at line 147 of file tl_ad_tensor.f90.

```
147     INTEGER, INTENT(IN) :: i,j,k
148     REAL(KIND=8), INTENT(IN)  :: v
149     IF (abs(v) .ge. real_eps) THEN
150        IF (j /= 0) count_elems(i)=count_elems(i)+1
151        IF (k /= 0) count_elems(i)=count_elems(i)+1
152     ENDIF
```

6.11.2.16 subroutine tl_ad_tensor::tl_coeff (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*) [private]

Subroutine used to compute the TL tensor entries.

Parameters

i	tensor i index
j	tensor j index
k	tensor k index
v	value to add

Definition at line 161 of file tl_ad_tensor.f90.

```
161     INTEGER, INTENT(IN) :: i,j,k
162     REAL(KIND=8), INTENT(IN) :: v
163     INTEGER :: n
164     IF (.NOT. ALLOCATED(tltensor)) stop "*** tl_coeff routine : tensor not yet allocated ***"
165     IF (.NOT. ALLOCATED(tltensor(i)%elems)) stop "*** tl_coeff routine : tensor not yet allocated ***"
166     IF (abs(v) .ge. real_eps) THEN
167        IF (j /=0) THEN
168           n=(tltensor(i)%nelems)+1
169           tltensor(i)%elems(n)%j=j
170           tltensor(i)%elems(n)%k=k
171           tltensor(i)%elems(n)%v=v
172           tltensor(i)%nelems=n
```

```
        END IF
        IF (k /=0) THEN
           n=(tltensor(i)%nelems)+1
           tltensor(i)%elems(n)%j=k
           tltensor(i)%elems(n)%k=j
           tltensor(i)%elems(n)%v=v
           tltensor(i)%nelems=n
        END IF
     END IF
```

6.11.3 Variable Documentation

6.11.3.1 type(coolist), dimension(:), allocatable, public tl_ad_tensor::adtensor

Tensor representation of the Adjoint tendencies.

Definition at line 44 of file tl_ad_tensor.f90.

```
44   TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: adtensor
```

6.11.3.2 integer, dimension(:), allocatable tl_ad_tensor::count_elems [private]

Vector used to count the tensor elements.

Definition at line 38 of file tl_ad_tensor.f90.

```
38   INTEGER, DIMENSION(:), ALLOCATABLE :: count_elems
```

6.11.3.3 real(kind=8), parameter tl_ad_tensor::real_eps = 2.2204460492503131e-16 [private]

Epsilon to test equality with 0.

Definition at line 35 of file tl_ad_tensor.f90.

```
35   REAL(KIND=8), PARAMETER :: real_eps = 2.2204460492503131e-16
```

6.11.3.4 type(coolist), dimension(:), allocatable, public tl_ad_tensor::tltensor

Tensor representation of the Tangent Linear tendencies.

Definition at line 41 of file tl_ad_tensor.f90.

```
41   TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: tltensor
```

6.12 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.
- subroutine, public init_random_seed ()

    Random generator initialization routine.
- subroutine, public init_one (A)

    Initialize a square matrix A as a unit matrix.

6.12.1 Detailed Description

Utility module.

Copyright

6.12.2 Function/Subroutine Documentation

6.12.2.1 subroutine, public util::init_one (real(kind=8), dimension(:,:), intent(inout) *A*)

Initialize a square matrix A as a unit matrix.

Definition at line 99 of file util.f90.

```
99        REAL(KIND=8), DIMENSION(:,:),INTENT(INOUT) :: a
100        INTEGER :: i,n
101        n=size(a,1)
102        a=0.0d0
103        DO i=1,n
104          a(i,i)=1.0d0
105        END DO
106
```

6.12.2.2 subroutine, public util::init_random_seed ( )

Random generator initialization routine.

Definition at line 44 of file util.f90.

6.12.2.3 character(len=40) function, public util::rstr (real(kind=8), intent(in) *x*, character(len=20), intent(in) *fm*)

Convert a real to string with a given format.

Definition at line 36 of file util.f90.

```
36       REAL(KIND=8), INTENT(IN) :: x
37       CHARACTER(len=20), INTENT(IN) :: fm
38       WRITE (rstr, trim(adjustl(fm))) x
39       rstr = adjustl(rstr)
```

6.12.2.4 character(len=20) function, public util::str (integer, intent(in) *k*)

Convert an integer to string.

Definition at line 29 of file util.f90.

```
29       INTEGER, INTENT(IN) :: k
30       WRITE (str, *) k
31       str = adjustl(str)
```

Chapter 7

Data Type Documentation

7.1 inprod_analytic::atm_tensors Type Reference

Type holding the atmospheric inner products tensors.

Private Attributes

- real(kind=8), dimension(:,:), allocatable a
- real(kind=8), dimension(:,:), allocatable c
- real(kind=8), dimension(:,:), allocatable d
- real(kind=8), dimension(:,:), allocatable s
- real(kind=8), dimension(:,:,:), allocatable b
- real(kind=8), dimension(:,:,:), allocatable g

7.1.1 Detailed Description

Type holding the atmospheric inner products tensors.

Definition at line 52 of file inprod_analytic.f90.

7.1.2 Member Data Documentation

7.1.2.1 real(kind=8), dimension(:,:), allocatable inprod_analytic::atm_tensors::a [private]

Definition at line 53 of file inprod_analytic.f90.

```
53     REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: a,c,d,s
```

7.1.2.2 real(kind=8), dimension(:,:,:), allocatable inprod_analytic::atm_tensors::b [private]

Definition at line 54 of file inprod_analytic.f90.

```
54     REAL(KIND=8), DIMENSION(:,:,:), ALLOCATABLE :: b,g
```

7.1.2.3 real(kind=8), dimension(:,:), allocatable inprod_analytic::atm_tensors::c [private]

Definition at line 53 of file inprod_analytic.f90.

7.1.2.4 real(kind=8), dimension(:,:), allocatable inprod_analytic::atm_tensors::d [private]

Definition at line 53 of file inprod_analytic.f90.

7.1.2.5 real(kind=8), dimension(:,:,:), allocatable inprod_analytic::atm_tensors::g [private]

Definition at line 54 of file inprod_analytic.f90.

7.1.2.6 real(kind=8), dimension(:,:), allocatable inprod_analytic::atm_tensors::s [private]

Definition at line 53 of file inprod_analytic.f90.

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

- inprod_analytic.f90

7.2 inprod_analytic::atm_wavenum Type Reference

Atmospheric bloc specification type.

Private Attributes

- character typ
- integer m =0
- integer p =0
- integer h =0
- real(kind=8) nx =0.
- real(kind=8) ny =0.

7.2.1 Detailed Description

Atmospheric bloc specification type.

Definition at line 39 of file inprod_analytic.f90.

7.2.2 Member Data Documentation

7.2.2.1 integer inprod_analytic::atm_wavenum::h =0 [private]

Definition at line 41 of file inprod_analytic.f90.

7.2.2.2 integer inprod_analytic::atm_wavenum::m =0 [private]

Definition at line 41 of file inprod_analytic.f90.

```
41      INTEGER :: m=0,p=0,h=0
```

7.2.2.3 real(kind=8) inprod_analytic::atm_wavenum::nx =0. [private]

Definition at line 42 of file inprod_analytic.f90.

```
42      REAL(KIND=8) :: nx=0.,ny=0.
```

7.2.2.4 real(kind=8) inprod_analytic::atm_wavenum::ny =0. [private]

Definition at line 42 of file inprod_analytic.f90.

7.2.2.5 integer inprod_analytic::atm_wavenum::p =0 [private]

Definition at line 41 of file inprod_analytic.f90.

7.2.2.6 character inprod_analytic::atm_wavenum::typ [private]

Definition at line 40 of file inprod_analytic.f90.

```
40      CHARACTER :: typ
```

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

- inprod_analytic.f90

7.3 tensor::coolist Type Reference

Coordinate list. Type used to represent the sparse tensor.

Public Attributes

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

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

  Number of elements in the list.

7.3.1 Detailed Description

Coordinate list. Type used to represent the sparse tensor.

Definition at line 27 of file tensor.f90.

7.3.2 Member Data Documentation

7.3.2.1 type(coolist_elem), dimension(:), allocatable tensor::coolist::elems

Lists of elements tensor::coolist_elem.

Definition at line 28 of file tensor.f90.

```
28      TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: elems !< Lists of elements tensor::coolist_elem
```

7.3.2.2 integer tensor::coolist::nelems = 0

Number of elements in the list.

Definition at line 29 of file tensor.f90.

```
29      INTEGER :: nelems = 0 !< Number of elements in the list.
```

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

- tensor.f90

7.4 tensor::coolist_elem 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.

7.4.1 Detailed Description

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

Definition at line 20 of file tensor.f90.

7.4.2 Member Data Documentation

7.4.2.1 integer tensor::coolist_elem::j [private]

Index j of the element.

Definition at line 21 of file tensor.f90.

```
21      INTEGER :: j !< Index \f$j\f$ of the element
```

7.4.2.2 integer tensor::coolist_elem::k [private]

Index k of the element.

Definition at line 22 of file tensor.f90.

```
22      INTEGER :: k !< Index \f$k\f$ of the element
```

7.4.2.3 real(kind=8) tensor::coolist_elem::v [private]

Value of the element.

Definition at line 23 of file tensor.f90.

```
23      REAL(KIND=8) :: v !< Value of the element
```

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

- tensor.f90

7.5 inprod_analytic::ocean_tensors Type Reference

Type holding the oceanic inner products tensors.

Private Attributes

- real(kind=8), dimension(:,:), allocatable k
- real(kind=8), dimension(:,:), allocatable m
- real(kind=8), dimension(:,:), allocatable n
- real(kind=8), dimension(:,:), allocatable w
- real(kind=8), dimension(:,:,:), allocatable o
- real(kind=8), dimension(:,:,:), allocatable c

7.5.1 Detailed Description

Type holding the oceanic inner products tensors.

Definition at line 58 of file inprod_analytic.f90.

7.5.2 Member Data Documentation

7.5.2.1 real(kind=8), dimension(:,:,:), allocatable inprod_analytic::ocean_tensors::c [private]

Definition at line 60 of file inprod_analytic.f90.

7.5.2.2 real(kind=8), dimension(:,:), allocatable inprod_analytic::ocean_tensors::k [private]

Definition at line 59 of file inprod_analytic.f90.

```
59      REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: k,m,n,w
```

7.5.2.3 real(kind=8), dimension(:,:), allocatable inprod_analytic::ocean_tensors::m [private]

Definition at line 59 of file inprod_analytic.f90.

7.5.2.4 real(kind=8), dimension(:,:), allocatable inprod_analytic::ocean_tensors::n [private]

Definition at line 59 of file inprod_analytic.f90.

7.5.2.5 real(kind=8), dimension(:,:,:), allocatable inprod_analytic::ocean_tensors::o [private]

Definition at line 60 of file inprod_analytic.f90.

```
60      REAL(KIND=8), DIMENSION(:,:,:), ALLOCATABLE :: o,c
```

7.5.2.6 real(kind=8), dimension(:,:), allocatable inprod_analytic::ocean_tensors::w [private]

Definition at line 59 of file inprod_analytic.f90.

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

- inprod_analytic.f90

7.6 inprod_analytic::ocean_wavenum Type Reference

Oceanic bloc specification type.

Private Attributes

- integer p
- integer h
- real(kind=8) nx
- real(kind=8) ny

7.6.1 Detailed Description

Oceanic bloc specification type.

Definition at line 46 of file inprod_analytic.f90.

7.6.2 Member Data Documentation

7.6.2.1 integer inprod_analytic::ocean_wavenum::h [private]

Definition at line 47 of file inprod_analytic.f90.

7.6.2.2 real(kind=8) inprod_analytic::ocean_wavenum::nx [private]

Definition at line 48 of file inprod_analytic.f90.

```
48      REAL(KIND=8) :: nx,ny
```

7.6.2.3 real(kind=8) inprod_analytic::ocean_wavenum::ny [private]

Definition at line 48 of file inprod_analytic.f90.

7.6.2.4 integer inprod_analytic::ocean_wavenum::p [private]

Definition at line 47 of file inprod_analytic.f90.

```
47      INTEGER :: p,h
```

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

- inprod_analytic.f90

Chapter 8

File Documentation

8.1 aotensor_def.f90 File Reference

Modules

- module aotensor_def

  The equation tensor for the coupled ocean-atmosphere model with temperature which allows for an extensible set of modes in the ocean and in the atmosphere.

Functions/Subroutines

- integer function aotensor_def::psi (i)

  Translate the $\psi_{a,i}$ coefficients into effective coordinates.
- integer function aotensor_def::theta (i)

  Translate the $\theta_{a,i}$ coefficients into effective coordinates.
- integer function aotensor_def::a (i)

  Translate the $\psi_{o,i}$ coefficients into effective coordinates.
- integer function aotensor_def::t (i)

  Translate the $\delta T_{o,i}$ coefficients into effective coordinates.
- integer function aotensor_def::kdelta (i, j)

  Kronecker delta function.
- subroutine aotensor_def::coeff (i, j, k, v)

  Subroutine to add element in the aotensor $\mathcal{T}_{i,j,k}$ structure.
- subroutine aotensor_def::add_count (i, j, k, v)

  Subroutine to count the elements of the aotensor $\mathcal{T}_{i,j,k}$. Add +1 to count_elems(i) for each value that is added to the tensor i-th component.
- subroutine aotensor_def::compute_aotensor (func)

  Subroutine to compute the tensor aotensor.
- subroutine, public aotensor_def::init_aotensor

  Subroutine to initialise the aotensor tensor.

Variables

- integer, dimension(:), allocatable aotensor_def::count_elems

  Vector used to count the tensor elements.
- real(kind=8), parameter aotensor_def::real_eps = 2.2204460492503131e-16

  Epsilon to test equality with 0.
- type(coolist), dimension(:), allocatable, public aotensor_def::aotensor

  $\mathcal{T}_{i,j,k}$ *- Tensor representation of the tendencies.*

8.2 doc/gen_doc.md File Reference

8.3 doc/tl_ad_doc.md File Reference

8.4 ic_def.f90 File Reference

Modules

- module ic_def

  Module to load the initial condition.

Functions/Subroutines

- subroutine, public ic_def::load_ic

  Subroutine to load the initial condition if IC.nml exists. If it does not, then write IC.nml with 0 as initial condition.

Variables

- logical ic_def::exists

  Boolean to test for file existence.
- real(kind=8), dimension(:), allocatable, public ic_def::ic

  Initial condition vector.

8.5 inprod_analytic.f90 File Reference

Data Types

- type inprod_analytic::atm_wavenum

  Atmospheric bloc specification type.
- type inprod_analytic::ocean_wavenum

  Oceanic bloc specification type.
- type inprod_analytic::atm_tensors

  Type holding the atmospheric inner products tensors.
- type inprod_analytic::ocean_tensors

  Type holding the oceanic inner products tensors.

Modules

- module inprod_analytic

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

Functions/Subroutines

- real(kind=8) function inprod_analytic::b1 (Pi, Pj, Pk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function inprod_analytic::b2 (Pi, Pj, Pk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function inprod_analytic::delta (r)

  Integer Dirac delta function.
- real(kind=8) function inprod_analytic::flambda (r)

  "Odd or even" function
- real(kind=8) function inprod_analytic::s1 (Pj, Pk, Mj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function inprod_analytic::s2 (Pj, Pk, Mj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function inprod_analytic::s3 (Pj, Pk, Hj, Hk)

  Cehelsky & Tung Helper functions.
- real(kind=8) function inprod_analytic::s4 (Pj, Pk, Hj, Hk)

  Cehelsky & Tung Helper functions.
- subroutine inprod_analytic::calculate_a

  Eigenvalues of the Laplacian (atmospheric)
- subroutine inprod_analytic::calculate_b

  Streamfunction advection terms (atmospheric)
- subroutine inprod_analytic::calculate_c_atm

  Beta term for the atmosphere.
- subroutine inprod_analytic::calculate_d

  Forcing of the ocean on the atmosphere.
- subroutine inprod_analytic::calculate_g

  Temperature advection terms (atmospheric)
- subroutine inprod_analytic::calculate_s

  Forcing (thermal) of the ocean on the atmosphere.
- subroutine inprod_analytic::calculate_k

  Forcing of the atmosphere on the ocean.
- subroutine inprod_analytic::calculate_m

  Forcing of the ocean fields on the ocean.
- subroutine inprod_analytic::calculate_n

  Beta term for the ocean.
- subroutine inprod_analytic::calculate_o

  Temperature advection term (passive scalar)
- subroutine inprod_analytic::calculate_c_oc

  Streamfunction advection terms (oceanic)
- subroutine inprod_analytic::calculate_w

  Short-wave radiative forcing of the ocean.
- subroutine, public inprod_analytic::init_inprod

  Initialisation of the inner product.
- subroutine, public inprod_analytic::deallocate_inprod

  Deallocation of the inner products.

Variables

- type(atm_wavenum), dimension(:), allocatable, public inprod_analytic::awavenum

  Atmospheric blocs specification.
- type(ocean_wavenum), dimension(:), allocatable, public inprod_analytic::owavenum

  Oceanic blocs specification.
- type(atm_tensors), public inprod_analytic::atmos

  Atmospheric tensors.
- type(ocean_tensors), public inprod_analytic::ocean

  Oceanic tensors.

8.6 LICENSE.txt File Reference

Functions

- The MIT License (MIT) Copyright(c) 2015-2016 Lesley De Cruz and Jonathan Demaeyer Permission is hereby granted
- The MIT free of to any person obtaining a copy of this software and associated documentation files (the"Software")

Variables

- The MIT free of charge
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without restriction
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to use
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to copy
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to modify
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to merge
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to publish
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to distribute
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to sublicense
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the Software
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do so
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following conditions
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY KIND

- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR IMPLIED
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF CONTRACT
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF TORT OR OTHERWISE
- The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF TORT OR ARISING FROM

8.6.1 Function Documentation

8.6.1.1 The MIT free of to any person obtaining a copy of this software and associated documentation files (the"Software")

8.6.1.2 The MIT License (MIT)

8.6.2 Variable Documentation

8.6.2.1 The MIT free of charge

Definition at line 6 of file LICENSE.txt.

8.6.2.2 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM

Definition at line 8 of file LICENSE.txt.

8.6.2.3 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following conditions

Definition at line 8 of file LICENSE.txt.

8.6.2.4 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF CONTRACT

Definition at line 8 of file LICENSE.txt.

8.6.2.5 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to copy

Definition at line 8 of file LICENSE.txt.

8.6.2.6 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to distribute

Definition at line 8 of file LICENSE.txt.

8.6.2.7 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF TORT OR ARISING FROM

Definition at line 8 of file LICENSE.txt.

8.6.2.8 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR IMPLIED

Definition at line 8 of file LICENSE.txt.

8.6.2.9 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY KIND

Definition at line 8 of file LICENSE.txt.

8.6.2.10 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY

Definition at line 8 of file LICENSE.txt.

8.6.2.11 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY

Definition at line 8 of file LICENSE.txt.

8.6.2.12 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to merge

Definition at line 8 of file LICENSE.txt.

8.6.2.13 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to modify

Definition at line 8 of file LICENSE.txt.

8.6.2.14 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do subject to the following WITHOUT WARRANTY OF ANY EXPRESS OR INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER WHETHER IN AN ACTION OF TORT OR OTHERWISE

Definition at line 8 of file LICENSE.txt.

8.6.2.15 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to publish

Definition at line 8 of file LICENSE.txt.

8.6.2.16 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without restriction

Definition at line 8 of file LICENSE.txt.

8.6.2.17 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the and to permit persons to whom the Software is furnished to do so

Definition at line 8 of file LICENSE.txt.

8.6.2.18 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to and or sell copies of the Software

Definition at line 8 of file LICENSE.txt.

8.6.2.19 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to sublicense

Definition at line 8 of file LICENSE.txt.

8.6.2.20 The MIT free of to any person obtaining a copy of this software and associated documentation to deal in the Software without including without limitation the rights to use

Definition at line 8 of file LICENSE.txt.

8.7 maooam.f90 File Reference

Functions/Subroutines

- program maooam

  Fortran 90 implementation of the modular arbitrary-order ocean-atmosphere model MAOOAM.

8.7.1 Function/Subroutine Documentation

8.7.1.1 program maooam ( )

Fortran 90 implementation of the modular arbitrary-order ocean-atmosphere model MAOOAM.

Copyright

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

Definition at line 13 of file maooam.f90.

8.8 params.f90 File Reference

Modules

- module params
  The model parameters module.

Functions/Subroutines

- subroutine, private params::init_nml
  Read the basic parameters and mode selection from the namelist.
- subroutine params::init_params
  Parameters initialisation routine.

Variables

- real(kind=8) params::n
  $n = 2L_y/L_x$ *- Aspect ratio*
- real(kind=8) params::phi0
  Latitude in radian.
- real(kind=8) params::rra
  Earth radius.
- real(kind=8) params::sig0
  σ_0 *- Non-dimensional static stability of the atmosphere.*
- real(kind=8) params::k
  Bottom atmospheric friction coefficient.
- real(kind=8) params::kp
  k' *- Internal atmospheric friction coefficient.*
- real(kind=8) params::r
  Frictional coefficient at the bottom of the ocean.
- real(kind=8) params::d
  Merchanical coupling parameter between the ocean and the atmosphere.
- real(kind=8) params::f0
  f_0 *- Coriolis parameter*
- real(kind=8) params::gp
  g' *Reduced gravity*
- real(kind=8) params::h
  Depth of the active water layer of the ocean.
- real(kind=8) params::phi0_npi
  Latitude exprimed in fraction of pi.
- real(kind=8) params::lambda
  λ *- Sensible + turbulent heat exchange between the ocean and the atmosphere.*
- real(kind=8) params::co
  C_a *- Constant short-wave radiation of the ocean.*
- real(kind=8) params::go
  γ_o *- Specific heat capacity of the ocean.*
- real(kind=8) params::ca
  C_a *- Constant short-wave radiation of the atmosphere.*
- real(kind=8) params::to0

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

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

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

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

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

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

  π
- real(kind=8) params::lr

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

  γ
- real(kind=8) params::rp

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  Transient time period.
- real(kind=8) params::t_run

Effective intergration time (length of the generated trajectory)
- real(kind=8) params::dt

  Integration time step.
- real(kind=8) params::tw

  Write all variables every tw time units.
- logical params::writeout

  Write to file boolean.
- integer params::nboc

  Number of atmospheric blocks.
- integer params::nbatm

  Number of oceanic blocks.
- integer params::natm =0

  Number of atmospheric basis functions.
- integer params::noc =0

  Number of oceanic basis functions.
- integer params::ndim

  Number of variables (dimension of the model)
- integer, dimension(:,:), allocatable params::oms

  Ocean mode selection array.
- integer, dimension(:,:), allocatable params::ams

  Atmospheric mode selection array.

8.9 rk2_integrator.f90 File Reference

Modules

- module rk2_integrator

  Module with the integration routines.

Functions/Subroutines

- subroutine, public rk2_integrator::init_integrator

  Routine to initialise the integration buffers.
- subroutine rk2_integrator::tendencies (t, y, res)

  Routine computing the tendencies of the model.
- subroutine, public rk2_integrator::step (y, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable rk2_integrator::buf_y1

  Buffer to hold the intermediate position (Heun algorithm)
- real(kind=8), dimension(:), allocatable rk2_integrator::buf_f0

  Buffer to hold tendencies at the initial position.
- real(kind=8), dimension(:), allocatable rk2_integrator::buf_f1

  Buffer to hold tendencies at the intermediate position.

8.10 rk2_tl_ad_integrator.f90 File Reference

Modules

- module rk2_tl_ad_integrator

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

Functions/Subroutines

- subroutine, public rk2_tl_ad_integrator::init_tl_ad_integrator

    Routine to initialise the integration buffers.
- subroutine, public rk2_tl_ad_integrator::ad_step (y, ystar, t, dt, res)

    Routine to perform an integration step (Heun algorithm) of the adjoint model. The incremented time is returned.
- subroutine, public rk2_tl_ad_integrator::tl_step (y, ystar, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_y1

    Buffer to hold the intermediate position (Heun algorithm) of the tangent linear model.
- real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_f0

    Buffer to hold tendencies at the initial position of the tangent linear model.
- real(kind=8), dimension(:), allocatable rk2_tl_ad_integrator::buf_f1

    Buffer to hold tendencies at the intermediate position of the tangent linear model.

8.11 rk4_integrator.f90 File Reference

Modules

- module rk4_integrator

    Module with the RK4 integration routines.

Functions/Subroutines

- subroutine, public rk4_integrator::init_integrator

    Routine to initialise the integration buffers.
- subroutine rk4_integrator::tendencies (t, y, res)

    Routine computing the tendencies of the model.
- subroutine, public rk4_integrator::step (y, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable rk4_integrator::buf_y1

    Buffer to hold the intermediate position (Heun algorithm)
- real(kind=8), dimension(:), allocatable rk4_integrator::buf_ka

    Buffer A to hold tendencies.
- real(kind=8), dimension(:), allocatable rk4_integrator::buf_kb

    Buffer B to hold tendencies.

8.12 rk4_tl_ad_integrator.f90 File Reference

Modules

- module rk4_tl_ad_integrator

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

Functions/Subroutines

- subroutine, public rk4_tl_ad_integrator::init_tl_ad_integrator

  Routine to initialise the TL-AD integration bufers.
- subroutine, public rk4_tl_ad_integrator::ad_step (y, ystar, t, dt, res)

  Routine to perform an integration step (RK4 algorithm) of the adjoint model. The incremented time is returned.
- subroutine, public rk4_tl_ad_integrator::tl_step (y, ystar, t, dt, res)

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

Variables

- real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_y1

  Buffer to hold the intermediate position of the tangent linear model.
- real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_ka

  Buffer to hold tendencies in the RK4 scheme for the tangent linear model.
- real(kind=8), dimension(:), allocatable rk4_tl_ad_integrator::buf_kb

  Buffer to hold tendencies in the RK4 scheme for the tangent linear model.

8.13 stat.f90 File Reference

Modules

- module stat

  Statistics accumulators.

Functions/Subroutines

- subroutine, public stat::init_stat

  Initialise the accumulators.
- subroutine, public stat::acc (x)

  Accumulate one state.
- real(kind=8) function, dimension(0:ndim), public stat::mean ()

  Function returning the mean.
- real(kind=8) function, dimension(0:ndim), public stat::var ()

  Function returning the variance.
- integer function, public stat::iter ()

  Function returning the number of data accumulated.
- subroutine, public stat::reset

  Routine resetting the accumulators.

Variables

- integer stat::i =0

  Number of stats accumulated.
- real(kind=8), dimension(:), allocatable stat::m

  Vector storing the inline mean.
- real(kind=8), dimension(:), allocatable stat::mprev

  Previous mean vector.
- real(kind=8), dimension(:), allocatable stat::v

  Vector storing the inline variance.
- real(kind=8), dimension(:), allocatable stat::mtmp

8.14 tensor.f90 File Reference

Data Types

- type tensor::coolist_elem

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

  Coordinate list. Type used to represent the sparse tensor.

Modules

- module tensor

  Tensor utility module.

Functions/Subroutines

- subroutine, public tensor::copy_tensor (src, dst)

  Routine to copy a tensor.
- subroutine, public tensor::mat_to_coo (src, dst)

  Routine to convert a matrix to a tensor.
- subroutine, public tensor::sparse_mul3 (coolist_ijk, 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, public tensor::jsparse_mul (coolist_ijk, arr_j, jcoo_ij)

  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 coolist (sparse tensor).
- subroutine, public tensor::jsparse_mul_mat (coolist_ijk, arr_j, jcoo_ij)

Sparse multiplication of two tensors to determine the Jacobian:

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

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, public tensor::sparse_mul2 (coolist_ij, arr_j, res)

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

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

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

.

Variables

- real(kind=8), parameter tensor::real_eps = 2.2204460492503131e-16

  Parameter to test the equality with zero.

8.15 test_aotensor.f90 File Reference

Functions/Subroutines

- program test_aotensor

  Small program to print the inner products.

8.15.1 Function/Subroutine Documentation

8.15.1.1 program test_aotensor ( )

Small program to print the inner products.

Copyright

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

Definition at line 13 of file test_aotensor.f90.

8.16 test_inprod_analytic.f90 File Reference

Functions/Subroutines

- program inprod_analytic_test

  Small program to print the inner products.

8.16.1 Function/Subroutine Documentation

8.16.1.1 program inprod_analytic_test ( )

Small program to print the inner products.

Copyright

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

Remarks

Print in the same order as test_inprod.lua

Definition at line 18 of file test_inprod_analytic.f90.

8.17 test_tl_ad.f90 File Reference

Functions/Subroutines

- program test_tl_ad

    Tests for the Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM.
- real(kind=8) function gasdev (idum)
- real(kind=8) function ran2 (idum)

8.17.1 Function/Subroutine Documentation

8.17.1.1 real(kind=8) function gasdev (integer *idum*)

Definition at line 149 of file test_tl_ad.f90.

```
149   INTEGER :: idum
150   REAL(KIND=8) ::  gasdev,ran2
151   !    USES ran2
152   INTEGER :: iset
153   REAL(KIND=8) :: fac,gset,rsq,v1,v2
154   SAVE iset,gset
155   DATA iset/0/
156   if (idum.lt.0) iset=0
157   if (iset.eq.0) then
158 1    v1=2.d0*ran2(idum)-1.
159      v2=2.d0*ran2(idum)-1.
160      rsq=v1**2+v2**2
161      if (rsq.ge.1.d0.or.rsq.eq.0.d0) goto 1
162      fac=sqrt(-2.*log(rsq)/rsq)
163      gset=v1*fac
164      gasdev=v2*fac
165      iset=1
166   else
167      gasdev=gset
168      iset=0
169   endif
170   return
```

8.17.1.2 real(kind=8) function ran2 (integer *idum*)

Definition at line 174 of file test_tl_ad.f90.

```
174    INTEGER :: idum,im1,im2,imm1,ia1,ia2,iq1,iq2,ir1,ir2,ntab,ndiv
175    REAL(KIND=8) :: ran2,am,eps,rnmx
176    parameter(im1=2147483563,im2=2147483399,am=1.d0/im1,imm1=im1-1&
177         &,ia1=40014,ia2=40692,iq1=53668,iq2=52774,ir1=12211,ir2&
178         &=3791,ntab=32,ndiv=1+imm1/ntab,eps=1.2d-7,rnmx=1.d0-eps)
179    INTEGER :: idum2,j,k,iv(ntab),iy
180    SAVE iv,iy,idum2
181    DATA idum2/123456789/, iv/ntab*0/, iy/0/
182    if (idum.le.0) then
183       idum=max(-idum,1)
184       idum2=idum
185       do j=ntab+8,1,-1
186          k=idum/iq1
187          idum=ia1*(idum-k*iq1)-k*ir1
188          if (idum.lt.0) idum=idum+im1
189          if (j.le.ntab) iv(j)=idum
190       enddo
191       iy=iv(1)
192    endif
193    k=idum/iq1
194    idum=ia1*(idum-k*iq1)-k*ir1
195    if (idum.lt.0) idum=idum+im1
196    k=idum2/iq2
197    idum2=ia2*(idum2-k*iq2)-k*ir2
198    if (idum2.lt.0) idum2=idum2+im2
199    j=1+iy/ndiv
200    iy=iv(j)-idum2
201    iv(j)=idum
202    if (iy.lt.1) iy=iy+imm1
203    ran2=min(am*iy,rnmx)
204    return
```

8.17.1.3 program test_tl_ad ( )

Tests for the Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM.

Copyright

Definition at line 14 of file test_tl_ad.f90.

8.18 tl_ad_tensor.f90 File Reference

Modules

- module tl_ad_tensor

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

Functions/Subroutines

- type(coolist) function, dimension(ndim) tl_ad_tensor::jacobian (ystar)
  Compute the Jacobian of MAOOAM in point ystar.
- real(kind=8) function, dimension(ndim, ndim) tl_ad_tensor::jacobian_mat (ystar)
  Compute the Jacobian of MAOOAM in point ystar.
- subroutine, public tl_ad_tensor::init_tltensor
  Routine to initialize the TL tensor.
- subroutine tl_ad_tensor::compute_tltensor (func)
  Routine to compute the TL tensor from the original MAOOAM one.
- subroutine tl_ad_tensor::tl_add_count (i, j, k, v)
  Subroutine used to count the number of TL tensor entries.
- subroutine tl_ad_tensor::tl_coeff (i, j, k, v)
  Subroutine used to compute the TL tensor entries.
- subroutine, public tl_ad_tensor::init_adtensor
  Routine to initialize the AD tensor.
- subroutine tl_ad_tensor::compute_adtensor (func)
  Subroutine to compute the AD tensor from the original MAOOAM one.
- subroutine tl_ad_tensor::ad_add_count (i, j, k, v)
  Subroutine used to count the number of AD tensor entries.
- subroutine tl_ad_tensor::ad_coeff (i, j, k, v)
- subroutine, public tl_ad_tensor::init_adtensor_ref
  Alternate method to initialize the AD tensor from the TL tensor.
- subroutine tl_ad_tensor::compute_adtensor_ref (func)
  Alternate subroutine to compute the AD tensor from the TL one.
- subroutine tl_ad_tensor::ad_add_count_ref (i, j, k, v)
  Alternate subroutine used to count the number of AD tensor entries from the TL tensor.
- subroutine tl_ad_tensor::ad_coeff_ref (i, j, k, v)
  Alternate subroutine used to compute the AD tensor entries from the TL tensor.
- subroutine, public tl_ad_tensor::ad (t, ystar, deltay, buf)
  Tendencies for the AD of MAOOAM in point ystar for perturbation deltay.
- subroutine, public tl_ad_tensor::tl (t, ystar, deltay, buf)
  Tendencies for the TL of MAOOAM in point ystar for perturbation deltay.

Variables

- real(kind=8), parameter tl_ad_tensor::real_eps = 2.2204460492503131e-16
  Epsilon to test equality with 0.
- integer, dimension(:), allocatable tl_ad_tensor::count_elems
  Vector used to count the tensor elements.
- type(coolist), dimension(:), allocatable, public tl_ad_tensor::tltensor
  Tensor representation of the Tangent Linear tendencies.
- type(coolist), dimension(:), allocatable, public tl_ad_tensor::adtensor
  Tensor representation of the Adjoint tendencies.

8.19 util.f90 File Reference

Modules

- module util
  Utility module.

Functions/Subroutines

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

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

    Convert a real to string with a given format.
- subroutine, public util::init_random_seed ()

    Random generator initialization routine.
- integer function lcg (s)
- subroutine, public util::init_one (A)

    Initialize a square matrix A as a unit matrix.

8.19.1 Function/Subroutine Documentation

8.19.1.1 integer function init_random_seed::lcg (integer(int64) *s*)

Definition at line 84 of file util.f90.

```
84      integer :: lcg
85      integer(int64) :: s
86      IF (s == 0) THEN
87         s = 104729
88      ELSE
89         s = mod(s, 4294967296_int64)
90      END IF
91      s = mod(s * 279470273_int64, 4294967291_int64)
92      lcg = int(mod(s, int(huge(0), int64)), kind(0))
93    END FUNCTION lcg
```

Index