Reference Manual

Generated by Doxygen 1.8.11

Contents

Chapter 1

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

About

(c) 2013-2016 Lesley De Cruz and Jonathan Demaeyer

See LICENSE.txt for license information.

This software is provided as supplementary material with:

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

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

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

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

A pdf version of this manual is available here.

Installation

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

By default, gfortran is selected. To select one or the other, simply modify the Makefile accordingly or pass the CO↩ MPILER flag to make. 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 or clone with git:

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

and run:

```
1 make
```

By default, the inner products of the basis functions, used to compute the coefficients of the ODEs, are not stored in memory. If you want to enable the storage in memory of these inner products, run make with the following flag:

```
1 make RES=store
```

Depending on the chosen resolution, storing the inner products may result in a huge memory usage and is not recommended unless you need them for a specific purpose.

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.
- 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.

The Makefile allows to change the integrator being used for the time evolution. The user should modify it according to its need. By default a RK2 scheme is selected.

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

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

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 203 of file aotensor_def.f90.

```
203      INTEGER :: i
204      INTEGER :: allocstat
205
206      CALL init_params  ! Iniatialise the parameter
207
208      CALL init_inprod  ! Initialise the inner product tensors
209
210      ALLOCATE(aotensor(ndim),count_elems(ndim), stat=allocstat)
211      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)

```

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             ic(0)=1.0d0
82             ! except IC(0), nothing has to be done IC has already the right values
83             WRITE(6,*) "*** IC.nml namelist written. Starting with initial condition in IC.nml !***"
84         END SELECT
85      ELSE
86         CALL init_random_seed()
87         CALL random_seed(get=seed)
88         ic=0
89         ic(0)=1.0d0
```

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

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

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

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

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

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

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

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

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

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

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

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

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

  Initialisation of the inner product.

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

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

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 100 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 106 of file inprod_analytic.f90.

```
106      INTEGER :: pi,pj,pk
107      b2 = (pk - pj) / REAL(pi)
```

6.3.2.3 real(kind=8) function inprod_analytic::calculate_a (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Eigenvalues of the Laplacian (atmospheric)

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

Definition at line 164 of file inprod_analytic.f90.

```
164      INTEGER, INTENT(IN) :: i,j
165      TYPE(atm_wavenum) :: ti
166
167      calculate_a = 0.d0
168      IF (i==j) THEN
169         ti = awavenum(i)
170         calculate_a = -(n**2) * ti%Nx**2 - ti%Ny**2
171      END IF
```

6.3.2.4 real(kind=8) function inprod_analytic::calculate_b (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*) [private]

Streamfunction advection terms (atmospheric)

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

Definition at line 178 of file inprod_analytic.f90.

```
178      INTEGER, INTENT(IN) :: i,j,k
179
180      calculate_b = calculate_a(k,k) * calculate_g(i,j,k)
181
```

6.3.2.5 real(kind=8) function inprod_analytic::calculate_c_atm (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Beta term for the atmosphere.

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

Definition at line 188 of file inprod_analytic.f90.

```
188      INTEGER, INTENT(IN) :: i,j
189      TYPE(atm_wavenum) :: ti, tj
190
191      ti = awavenum(i)
192      tj = awavenum(j)
193      calculate_c_atm = 0.d0
194      IF ((ti%typ == "K") .AND. (tj%typ == "L")) THEN
195         calculate_c_atm = n * ti%M * delta(ti%M - tj%H) * delta(ti%P - tj%P)
196      ELSE IF ((ti%typ == "L") .AND. (tj%typ == "K")) THEN
197         ti = awavenum(j)
198         tj = awavenum(i)
199         calculate_c_atm = - n * ti%M * delta(ti%M - tj%H) * delta(ti%P - tj%P)
200      END IF
201
```

6.3.2.6 real(kind=8) function inprod_analytic::calculate_c_oc (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*) [private]

Streamfunction advection terms (oceanic)

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

Definition at line 412 of file inprod_analytic.f90.

```
412     INTEGER, INTENT(IN) :: i,j,k
413
414     calculate_c_oc = calculate_m(k,k) * calculate_o(i,j,k)
415
```

6.3.2.7 real(kind=8) function inprod_analytic::calculate_d (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Forcing of the ocean on the atmosphere.

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

Definition at line 208 of file inprod_analytic.f90.

```
208     INTEGER, INTENT(IN) :: i,j
209
210     calculate_d=calculate_s(i,j) * calculate_m(j,j)
211
```

6.3.2.8 real(kind=8) function inprod_analytic::calculate_g (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*) [private]

Temperature advection terms (atmospheric)

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

Definition at line 218 of file inprod_analytic.f90.

```
218     INTEGER, INTENT(IN) :: i,j,k
219     TYPE(atm_wavenum) :: ti,tj,tk
220     REAL(KIND=8) :: val,vb1, vb2, vs1, vs2, vs3, vs4
221     INTEGER, DIMENSION(3) :: a,b
222     INTEGER, DIMENSION(3,3) :: w
223     CHARACTER, DIMENSION(3) :: s
224     INTEGER :: par
225
226     ti = awavenum(i)
227     tj = awavenum(j)
228     tk = awavenum(k)
229
230     a(1)=i
231     a(2)=j
232     a(3)=k
233
234     val=0.d0
235
236     IF ((ti%typ == "L") .AND. (tj%typ == "L") .AND. (tk%typ == "L")) THEN
237
238        CALL piksrt(3,a,par)
239
240        ti = awavenum(a(1))
241        tj = awavenum(a(2))
242        tk = awavenum(a(3))
243
```

```
      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)))
   ELSE

      s(1)=ti%typ
      s(2)=tj%typ
      s(3)=tk%typ

      w(1,:)=isin("A",s)
      w(2,:)=isin("K",s)
      w(3,:)=isin("L",s)

      IF (any(w(1,:)/=0) .AND. any(w(2,:)/=0) .AND. any(w(3,:)/=0)) THEN
         b=w(:,1)
         ti = awavenum(a(b(1)))
         tj = awavenum(a(b(2)))
         tk = awavenum(a(b(3)))
         call piksrt(3,b,par)
         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 ((w(2,2)/=0) .AND. (w(2,3)==0) .AND. any(w(3,:)/=0)) THEN
         ti = awavenum(a(w(2,1)))
         tj = awavenum(a(w(2,2)))
         tk = awavenum(a(w(3,1)))
         b(1)=w(2,1)
         b(2)=w(2,2)
         b(3)=w(3,1)
         call piksrt(3,b,par)
         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)))
      ENDIF
   ENDIF
   calculate_g=par*val*n

```

6.3.2.9 real(kind=8) function inprod_analytic::calculate_k (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Forcing of the atmosphere on the ocean.

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

Definition at line 336 of file inprod_analytic.f90.

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

   calculate_k = calculate_s(j,i) * calculate_a(j,j)
```

6.3.2.10 real(kind=8) function inprod_analytic::calculate_m (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Forcing of the ocean fields on the ocean.

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

Definition at line 345 of file inprod_analytic.f90.

```
345     INTEGER, INTENT(IN) :: i,j
346     TYPE(ocean_wavenum) :: di
347
348     calculate_m=0.d0
349     IF (i==j) THEN
350        di = owavenum(i)
351        calculate_m = -(n**2) * di%Nx**2 - di%Ny**2
352     END IF
```

6.3.2.11 real(kind=8) function inprod_analytic::calculate_n (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Beta term for the ocean.

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

Definition at line 359 of file inprod_analytic.f90.

```
359     INTEGER, INTENT(IN) :: i,j
360     TYPE(ocean_wavenum) :: di,dj
361     REAL(KIND=8) :: val
362
363     di = owavenum(i)
364     dj = owavenum(j)
365     calculate_n = 0.d0
366     IF (dj%H/=di%H) THEN
367        val = delta(di%P - dj%P) * flambda(di%H + dj%H)
368        calculate_n = val * (-2) * dj%H * di%H * n / ((dj%H**2 - di%H**2) * pi)
369     ENDIF
370
```

6.3.2.12 real(kind=8) function inprod_analytic::calculate_o (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*) [private]

Temperature advection term (passive scalar)

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

Definition at line 377 of file inprod_analytic.f90.

```
377     INTEGER, INTENT(IN) :: i,j,k
378     TYPE(ocean_wavenum) :: di,dj,dk
379     REAL(KIND=8) :: vs3,vs4,val
380     INTEGER, DIMENSION(3) :: a
381     INTEGER :: par
382
383     val=0.d0
384
385     a(1)=i
386     a(2)=j
387     a(3)=k
388
389     CALL piksrt(3,a,par)
390
391     di = owavenum(a(1))
392     dj = owavenum(a(2))
393     dk = owavenum(a(3))
394
395     vs3 = s3(dj%P,dk%P,dj%H,dk%H)
396     vs4 = s4(dj%P,dk%P,dj%H,dk%H)
397     val = vs3*((delta(dk%H - dj%H - di%H) - delta(dk%H - dj&
398          &%H + di%H)) * delta(dk%P + dj%P - di%P) + delta(dk&
399          &%H + dj%H - di%H) * (delta(dk%P - dj%P + di%P) -&
400          & delta(dk%P - dj%P - di%P))) + vs4 * ((delta(dk%H &
401          &+ dj%H - di%H) * delta(dk%P - dj%P - di%P)) +&
402          & (delta(dk%H - dj%H + di%H) - delta(dk%H - dj%H -&
403          & di%H)) * (delta(dk%P - dj%P - di%P) - delta(dk%P &
404          &- dj%P + di%P)))
405     calculate_o = par * val * n / 2
```

6.3.2.13 real(kind=8) function inprod_analytic::calculate_s (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Forcing (thermal) of the ocean on the atmosphere.

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

Definition at line 303 of file inprod_analytic.f90.

```
303     INTEGER, INTENT(IN) :: i,j
304     TYPE(atm_wavenum) :: ti
305     TYPE(ocean_wavenum) :: dj
306     REAL(KIND=8) :: val
307
308     ti = awavenum(i)
309     dj = owavenum(j)
310     val=0.d0
311     IF (ti%typ == "A") THEN
312        val = flambda(dj%H) * flambda(dj%P + ti%P)
313        IF (val /= 0.d0) THEN
314           val = val*8*sqrt(2.)*dj%P/(pi**2 * (dj%P**2 - ti%P**2) * dj%H)
315        END IF
316     ELSEIF (ti%typ == "K") THEN
317        val = flambda(2 * ti%M + dj%H) * delta(dj%P - ti%P)
318        IF (val /= 0.d0) THEN
319           val = val*4*dj%H/(pi * (-4 * ti%M**2 + dj%H**2))
320        END IF
321     ELSEIF (ti%typ == "L") THEN
322        val = delta(dj%P - ti%P) * delta(2 * ti%H - dj%H)
323     END IF
324     calculate_s=val
325
```

6.3.2.14 real(kind=8) function inprod_analytic::calculate_w (integer, intent(in) *i*, integer, intent(in) *j*) [private]

Short-wave radiative forcing of the ocean.

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

Definition at line 422 of file inprod_analytic.f90.

```
422     INTEGER, INTENT(IN) :: i,j
423
424     calculate_w = calculate_s(j,i)
425
```

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

Integer Dirac delta function.

Definition at line 112 of file inprod_analytic.f90.

```
112     INTEGER :: r
113     IF (r==0) THEN
114        delta = 1.d0
115     ELSE
116        delta = 0.d0
117     ENDIF
```

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

"Odd or even" function

Definition at line 122 of file inprod_analytic.f90.

```
122     INTEGER :: r
123     IF (mod(r,2)==0) THEN
124        flambda = 0.d0
125     ELSE
126        flambda = 1.d0
127     ENDIF
```

6.3.2.17 subroutine, public inprod_analytic::init_inprod ( )

Initialisation of the inner product.

Definition at line 436 of file inprod_analytic.f90.

```
436     INTEGER :: i,j
437     INTEGER :: allocstat
438
439     IF (natm == 0 ) THEN
440        stop "*** Problem : natm==0 ! ***"
441     ELSEIF (noc == 0) then
442        stop "*** Problem : noc==0 ! ***"
443     END IF
444
445
446     ! Definition of the types and wave numbers tables
447
448     ALLOCATE(owavenum(noc),awavenum(natm), stat=allocstat)
449     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
450
451     j=0
452     DO i=1,nbatm
453        IF (ams(i,1)==1) THEN
454           awavenum(j+1)%typ='A'
455           awavenum(j+2)%typ='K'
456           awavenum(j+3)%typ='L'
457
458           awavenum(j+1)%P=ams(i,2)
459           awavenum(j+2)%M=ams(i,1)
460           awavenum(j+2)%P=ams(i,2)
461           awavenum(j+3)%H=ams(i,1)
462           awavenum(j+3)%P=ams(i,2)
463
464           awavenum(j+1)%Ny=REAL(ams(i,2))
465           awavenum(j+2)%Nx=REAL(ams(i,1))
466           awavenum(j+2)%Ny=REAL(ams(i,2))
467           awavenum(j+3)%Nx=REAL(ams(i,1))
468           awavenum(j+3)%Ny=REAL(ams(i,2))
469
470           j=j+3
471        ELSE
472           awavenum(j+1)%typ='K'
473           awavenum(j+2)%typ='L'
474
475           awavenum(j+1)%M=ams(i,1)
476           awavenum(j+1)%P=ams(i,2)
477           awavenum(j+2)%H=ams(i,1)
478           awavenum(j+2)%P=ams(i,2)
479
480           awavenum(j+1)%Nx=REAL(ams(i,1))
481           awavenum(j+1)%Ny=REAL(ams(i,2))
482           awavenum(j+2)%Nx=REAL(ams(i,1))
483           awavenum(j+2)%Ny=REAL(ams(i,2))
484
485           j=j+2
486
487        ENDIF
488     ENDDO
489
490     DO i=1,noc
491        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

   ! Pointing to the atmospheric inner products functions

   atmos%a => calculate_a
   atmos%g => calculate_g
   atmos%s => calculate_s
   atmos%b => calculate_b
   atmos%d => calculate_d
   atmos%c => calculate_c_atm

   ! Pointing to the oceanic inner products functions

   ocean%M => calculate_m
   ocean%N => calculate_n
   ocean%O => calculate_o
   ocean%C => calculate_c_oc
   ocean%W => calculate_w
   ocean%K => calculate_k

```

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

Cehelsky & Tung Helper functions.

Definition at line 132 of file inprod_analytic.f90.

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

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

Cehelsky & Tung Helper functions.

Definition at line 138 of file inprod_analytic.f90.

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

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

Cehelsky & Tung Helper functions.

Definition at line 144 of file inprod_analytic.f90.

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

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

Cehelsky & Tung Helper functions.

Definition at line 150 of file inprod_analytic.f90.

```
150     INTEGER :: pj,pk,hj,hk
151     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 78 of file inprod_analytic.f90.

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

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

Atmospheric blocs specification.

Definition at line 73 of file inprod_analytic.f90.

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

6.3.3.3 type(ocean_tensors), public inprod_analytic::ocean

Oceanic tensors.

Definition at line 80 of file inprod_analytic.f90.

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

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

Oceanic blocs specification.

Definition at line 75 of file inprod_analytic.f90.

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

6.4 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.
- real(kind=8), dimension(:), allocatable buf_ka

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

  Buffer B to hold tendencies.

6.4.1 Detailed Description

Module with the integration routines.

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 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.

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.

6.4.2 Function/Subroutine Documentation

6.4.2.1 subroutine public 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 ! ***"
```

6.4.2.2 subroutine public 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.

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 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
```

6.4.2.3 subroutine 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)
```

6.4.3 Variable Documentation

6.4.3.1 real(kind=8), dimension(:), allocatable 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
```

6.4.3.2 real(kind=8), dimension(:), allocatable 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
```

6.4.3.3 real(kind=8), dimension(:), allocatable 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
```

6.4.3.4 real(kind=8), dimension(:), allocatable 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
```

6.4.3.5 real(kind=8), dimension(:), allocatable 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)
```

6.5 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

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

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

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

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

  Non-dimensional dissipation in the ocean.
- 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.

6.5.1 Detailed Description

The model parameters module.

Copyright

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

Remarks

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

6.5.2 Function/Subroutine Documentation

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

Read the basic parameters and mode selection from the namelist.

Definition at line 97 of file params.f90.

```
97     INTEGER :: allocstat
98
99     namelist /aoscale/  scale,f0,n,rra,phi0_npi
100     namelist /oparams/  gp,r,h,d,nuo
101     namelist /aparams/  k,kp,sig0,nua
102     namelist /toparams/ go,co,to0
103     namelist /taparams/ ga,ca,epsa,ta0
104     namelist /otparams/ sc,lambda,rr,sb
105
106     namelist /modeselection/ oms,ams
107     namelist /numblocs/ nboc,nbatm
108
109     namelist /int_params/ t_trans,t_run,dt,tw,writeout
110
111     OPEN(8, file="params.nml", status='OLD', recl=80, delim='APOSTROPHE')
112
113     READ(8,nml=aoscale)
114     READ(8,nml=oparams)
115     READ(8,nml=aparams)
116     READ(8,nml=toparams)
117     READ(8,nml=taparams)
118     READ(8,nml=otparams)
119
120     CLOSE(8)
121
122     OPEN(8, file="modeselection.nml", status='OLD', recl=80, delim='APOSTROPHE')
123     READ(8,nml=numblocs)
124
125     ALLOCATE(oms(nboc,2),ams(nbatm,2), stat=allocstat)
126     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
127
128     READ(8,nml=modeselection)
129     CLOSE(8)
130
131     OPEN(8, file="int_params.nml", status='OLD', recl=80, delim='APOSTROPHE')
132     READ(8,nml=int_params)
133
```

6.5.2.2 subroutine params::init_params ( )

Parameters initialisation routine.

Definition at line 138 of file params.f90.

```
138     INTEGER, DIMENSION(2) :: s
139     INTEGER :: i
140     CALL init_nml
141
142     !---------------------------------------------------------!
143     !                                                         !
144     ! Computation of the dimension of the atmospheric         !
145     ! and oceanic components                                  !
146     !                                                         !
147     !---------------------------------------------------------!
148
149     natm=0
150     DO i=1,nbatm
151        IF (ams(i,1)==1) THEN
152           natm=natm+3
153        ELSE
154           natm=natm+2
155        ENDIF
156     ENDDO
157     s=shape(oms)
158     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
nuap=nua/(f0*l**2)
nuop=nuo/(f0*l**2)
```

6.5.3 Variable Documentation

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

Atmospheric mode selection array.

Definition at line 87 of file params.f90.

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

6.5.3.2 real(kind=8) params::betp

β' - Non-dimensional beta parameter

Definition at line 67 of file params.f90.

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

6.5.3.3 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.
```

6.5.3.4 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.
```

6.5.3.5 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.
```

6.5.3.6 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.
```

6.5.3.7 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.
```

6.5.3.8 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.
```

6.5.3.9 real(kind=8) params::dt

Integration time step.

Definition at line 77 of file params.f90.

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

6.5.3.10 real(kind=8) params::epsa

ϵ_a - Emissivity coefficient for the grey-body atmosphere.

Definition at line 43 of file params.f90.

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

6.5.3.11 real(kind=8) params::f0

f_0 - Coriolis parameter

Definition at line 32 of file params.f90.

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

6.5.3.12 real(kind=8) params::g

γ

Definition at line 50 of file params.f90.

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

6.5.3.13 real(kind=8) params::ga

γ_a - Specific heat capacity of the atmosphere.

Definition at line 44 of file params.f90.

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

6.5.3.14 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.
```

6.5.3.15 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
```

6.5.3.16 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.
```

6.5.3.17 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.
```

6.5.3.18 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.
```

6.5.3.19 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.
```

6.5.3.20 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.
```

6.5.3.21 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
```

6.5.3.22 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.
```

6.5.3.23 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.
```

6.5.3.24 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.

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

6.5.3.25 real(kind=8) params::lr

L_R - Rossby deformation radius

Definition at line 49 of file params.f90.

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

6.5.3.26 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.

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

6.5.3.27 real(kind=8) params::lsbpo

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

Definition at line 62 of file params.f90.

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

6.5.3.28 real(kind=8) params::n

$n = 2L_y/L_x$ - Aspect ratio

Definition at line 24 of file params.f90.

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

6.5.3.29 integer params::natm =0

Number of atmospheric basis functions.

Definition at line 83 of file params.f90.

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

6.5.3.30 integer params::nbatm

Number of oceanic blocks.

Definition at line 82 of file params.f90.

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

6.5.3.31 integer params::nboc

Number of atmospheric blocks.

Definition at line 81 of file params.f90.

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

6.5.3.32 integer params::ndim

Number of variables (dimension of the model)

Definition at line 85 of file params.f90.

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

6.5.3.33 integer params::noc =0

Number of oceanic basis functions.

Definition at line 84 of file params.f90.

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

6.5.3.34 real(kind=8) params::nua =0.D0

Dissipation in the atmosphere.

Definition at line 69 of file params.f90.

```
69  REAL(KIND=8) :: nua=0.d0  !< Dissipation in the atmosphere
```

6.5.3.35 real(kind=8) params::nuap

Non-dimensional dissipation in the atmosphere.

Definition at line 72 of file params.f90.

```
72  REAL(KIND=8) :: nuap      !< Non-dimensional dissipation in the atmosphere
```

6.5.3.36 real(kind=8) params::nuo =0.D0

Dissipation in the ocean.

Definition at line 70 of file params.f90.

```
70  REAL(KIND=8) :: nuo=0.d0  !< Dissipation in the ocean
```

6.5.3.37 real(kind=8) params::nuop

Non-dimensional dissipation in the ocean.

Definition at line 73 of file params.f90.

```
73  REAL(KIND=8) :: nuop      !< Non-dimensional dissipation in the ocean
```

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

Ocean mode selection array.

Definition at line 86 of file params.f90.

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

6.5.3.39 real(kind=8) params::phi0

Latitude in radian.

Definition at line 25 of file params.f90.

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

6.5.3.40 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.
```

6.5.3.41 real(kind=8) params::pi

π

Definition at line 48 of file params.f90.

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

6.5.3.42 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.
```

6.5.3.43 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.
```

6.5.3.44 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
```

6.5.3.45 real(kind=8) params::rra

Earth radius.

Definition at line 26 of file params.f90.

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

6.5.3.46 real(kind=8) params::sb

Stefan–Boltzmann constant.

Definition at line 66 of file params.f90.

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

6.5.3.47 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.
```

6.5.3.48 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.
```

6.5.3.49 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.
```

6.5.3.50 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.
```

6.5.3.51 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.
```

6.5.3.52 real(kind=8) params::t_run

Effective intergration time (length of the generated trajectory)

Definition at line 76 of file params.f90.

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

6.5.3.53 real(kind=8) params::t_trans

Transient time period.

Definition at line 75 of file params.f90.

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

6.5.3.54 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.
```

6.5.3.55 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.
```

6.5.3.56 real(kind=8) params::tw

Write all variables every tw time units.

Definition at line 78 of file params.f90.

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

6.5.3.57 logical params::writeout

Write to file boolean.

Definition at line 79 of file params.f90.

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

6.6 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

6.6.1 Detailed Description

Statistics accumulators.

Copyright

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

6.6.2 Function/Subroutine Documentation

6.6.2.1 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)
```

6.6.2.2 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
```

6.6.2.3 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
```

6.6.2.4 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
```

6.6.2.5 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
```

6.6.2.6 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)
```

6.6.3 Variable Documentation

6.6.3.1 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
```

6.6.3.2 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
```

6.6.3.3 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
```

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

Definition at line 26 of file stat.f90.

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

6.6.3.5 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
```

6.7 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_coo (src, dst)

  Routine to copy a coolist.
- 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:

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

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

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

  This version return a matrix.
- subroutine, public sparse_mul2 (coolist_ij, arr_j, res)

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

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

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

  .
- subroutine, public add_elem (t, i, j, k, v)

  Subroutine to add element to a coolist.
- subroutine, public add_check (t, i, j, k, v, dst)

  Subroutine to add element to a coolist and check for overflow. Once the t buffer tensor is full, add it to the destination buffer.
- subroutine, public add_to_tensor (src, dst)

  Routine to add a rank-3 tensor to another one.
- subroutine, public print_tensor (t, s)

  Routine to print a rank 3 tensor coolist.
- subroutine, public write_tensor_to_file (s, t)

  Load a rank-4 tensor coolist from a file definition.
- subroutine, public load_tensor_from_file (s, t)

  Load a rank-4 tensor coolist from a file definition.

Variables

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

  Parameter to test the equality with zero.

6.7.1 Detailed Description

Tensor utility module.

Copyright

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

6.7.2 Function/Subroutine Documentation

6.7.2.1 subroutine, public tensor::add_check (type(coolist), dimension(ndim), intent(inout) *t*, integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*, type(coolist), dimension(ndim), intent(inout) *dst*)

Subroutine to add element to a coolist and check for overflow. Once the t buffer tensor is full, add it to the destination buffer.

Parameters

t	temporary buffer tensor for the destination tensor
i	tensor i index
j	tensor j index
k	tensor k index
v	value to add
dst	destination tensor

Definition at line 303 of file tensor.f90.

```
303      TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
304      TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
305      INTEGER, INTENT(IN) :: i,j,k
306      REAL(KIND=8), INTENT(IN) :: v
307      INTEGER :: n
308      CALL add_elem(t,i,j,k,v)
309      IF (t(i)%nelems==size(t(i)%elems)) THEN
310         CALL add_to_tensor(t,dst)
311         DO n=1,ndim
312             t(n)%nelems=0
313         ENDDO
314      ENDIF
```

6.7.2.2 subroutine, public tensor::add_elem (type(coolist), dimension(ndim), intent(inout) *t*, integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), intent(in) *v*)

Subroutine to add element to a coolist.

Parameters

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

Definition at line 281 of file tensor.f90.

```
281     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
282     INTEGER, INTENT(IN) :: i,j,k
283     REAL(KIND=8), INTENT(IN) :: v
284     INTEGER :: n
285     IF (abs(v) .ge. real_eps) THEN
286        n=(t(i)%nelems)+1
287        t(i)%elems(n)%j=j
288        t(i)%elems(n)%k=k
289        t(i)%elems(n)%v=v
290        t(i)%nelems=n
291     END IF
```

6.7.2.3 subroutine, public tensor::add_to_tensor (type(coolist), dimension(ndim), intent(in) *src*, type(coolist), dimension(ndim), intent(inout) *dst*)

Routine to add a rank-3 tensor to another one.

Parameters

src	Tensor to add
dst	Destination tensor

Definition at line 321 of file tensor.f90.

```
321     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
322     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
323     TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
324     INTEGER :: i,j,n,allocstat
325
326     DO i=1,ndim
327        IF (src(i)%nelems/=0) THEN
328           IF (dst(i)%nelems==0) THEN
329              IF (ALLOCATED(dst(i)%elems)) THEN
330                 DEALLOCATE(dst(i)%elems, stat=allocstat)
331                 IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
332              ENDIF
333              ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
334              IF (allocstat /= 0) stop "*** Not enough memory ! ***"
335              n=0
336           ELSE
337              n=dst(i)%nelems
338              ALLOCATE(celems(n), stat=allocstat)
339              DO j=1,n
340                 celems(j)%j=dst(i)%elems(j)%j
341                 celems(j)%k=dst(i)%elems(j)%k
342                 celems(j)%v=dst(i)%elems(j)%v
343              ENDDO
344              DEALLOCATE(dst(i)%elems, stat=allocstat)
345              IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
346              ALLOCATE(dst(i)%elems(src(i)%nelems+n), stat=allocstat)
347              IF (allocstat /= 0) stop "*** Not enough memory ! ***"
348              DO j=1,n
349                 dst(i)%elems(j)%j=celems(j)%j
350                 dst(i)%elems(j)%k=celems(j)%k
351                 dst(i)%elems(j)%v=celems(j)%v
```

```
            ENDDO
            DEALLOCATE(celems, stat=allocstat)
            IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
         ENDIF
         DO j=1,src(i)%nelems
            dst(i)%elems(n+j)%j=src(i)%elems(j)%j
            dst(i)%elems(n+j)%k=src(i)%elems(j)%k
            dst(i)%elems(n+j)%v=src(i)%elems(j)%v
         ENDDO
         dst(i)%nelems=src(i)%nelems+n
      ENDIF
   ENDDO

```

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

Routine to copy a coolist.

Parameters

src	Source coolist
dst	Destination coolist

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 45 of file tensor.f90.

```
45    TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
46    TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
47    INTEGER :: i,j,allocstat
48
49    DO i=1,ndim
50       IF (dst(i)%nelems/=0) stop "*** copy_coo : Destination coolist not empty ! ***"
51       ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
52       IF (allocstat /= 0) stop "*** Not enough memory ! ***"
53       DO j=1,src(i)%nelems
54          dst(i)%elems(j)%j=src(i)%elems(j)%j
55          dst(i)%elems(j)%k=src(i)%elems(j)%k
56          dst(i)%elems(j)%v=src(i)%elems(j)%v
57       ENDDO
58       dst(i)%nelems=src(i)%nelems
59    ENDDO
```

6.7.2.5 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 124 of file tensor.f90.

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

6.7.2.6 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 167 of file tensor.f90.

```
TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT):: jcoo_ij
REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j
REAL(KIND=8) :: v
INTEGER :: i,j,k,n
jcoo_ij=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
      v=coolist_ijk(i)%elems(n)%v
      IF (j /=0) jcoo_ij(i,j)=jcoo_ij(i,j)+v*arr_j(k)
      IF (k /=0) jcoo_ij(i,k)=jcoo_ij(i,k)+v*arr_j(j)
   END DO
END DO
```

6.7.2.7 subroutine, public tensor::load_tensor_from_file (character (len=∗), intent(in) *s*, type(coolist), dimension(ndim), intent(out) *t*)

Load a rank-4 tensor coolist from a file definition.

Parameters

s	Filename of the tensor definition file
t	The loaded coolist

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 416 of file tensor.f90.

```
CHARACTER (LEN=*), INTENT(IN) :: s
TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: t
INTEGER :: i,ir,j,k,n,allocstat
REAL(KIND=8) :: v
OPEN(30,file=s,status='old')
DO i=1,ndim
   READ(30,*) ir,n
   IF (n /= 0) THEN
      ALLOCATE(t(i)%elems(n), stat=allocstat)
      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
      t(i)%nelems=n
   ENDIF
   DO n=1,t(i)%nelems
      READ(30,*) ir,j,k,v
      t(i)%elems(n)%j=j
      t(i)%elems(n)%k=k
      t(i)%elems(n)%v=v
   ENDDO
END DO
CLOSE(30)
```

6.7.2.8 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 67 of file tensor.f90.

```
67      REAL(KIND=8), DIMENSION(0:ndim,0:ndim), INTENT(IN) :: src
68      TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
69      INTEGER :: i,j,n,allocstat
70      DO i=1,ndim
71         n=0
72         DO j=1,ndim
73            IF (abs(src(i,j))>real_eps) n=n+1
74         ENDDO
75         IF (dst(i)%nelems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
76         ALLOCATE(dst(i)%elems(n), stat=allocstat)
77         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
78         n=0
79         DO j=1,ndim
80            IF (abs(src(i,j))>real_eps) THEN
81               n=n+1
82               dst(i)%elems(n)%j=j
83               dst(i)%elems(n)%k=0
84               dst(i)%elems(n)%v=src(i,j)
85            ENDIF
86         ENDDO
87         dst(i)%nelems=n
88      ENDDO
```

6.7.2.9 subroutine, public tensor::print_tensor (type(coolist), dimension(ndim), intent(in) *t*, character, intent(in), optional *s*)

Routine to print a rank 3 tensor coolist.

Parameters

t	coolist to print

Definition at line 370 of file tensor.f90.

```
370      USE util, only: str
371      TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: t
372      CHARACTER, INTENT(IN), OPTIONAL :: s
373      CHARACTER :: r
374      INTEGER :: i,n,j,k
375      IF (PRESENT(s)) THEN
376         r=s
377      ELSE
378         r="t"
379      END IF
380      DO i=1,ndim
381         DO n=1,t(i)%nelems
382            j=t(i)%elems(n)%j
383            k=t(i)%elems(n)%k
384            IF( abs(t(i)%elems(n)%v) .GE. real_eps) THEN
385               write(*,"(A,ES12.5)") s//"["//trim(str(i))//"]["//trim(str(j)) &
386                    &//"]["//trim(str(k))//"] = ",t(i)%elems(n)%v
387            END IF
388         END DO
389      END DO
```

6.7.2.10 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 209 of file tensor.f90.

```
209    TYPE(coolist), DIMENSION(ndim), INTENT(INOUT):: tensor
210    INTEGER :: i,j,k
211    INTEGER :: li,lii,liii,n
212    DO i= 1,ndim
213       n=tensor(i)%nelems
214       DO li=n,2,-1
215          j=tensor(i)%elems(li)%j
216          k=tensor(i)%elems(li)%k
217          DO lii=li-1,1,-1
218             IF (((j==tensor(i)%elems(lii)%j).AND.(k==tensor(i)&
219                &%elems(lii)%k)).OR.((j==tensor(i)%elems(lii)%k).AND.(k==
      tensor(i)%elems(lii)%j))) THEN
220                ! Found another entry with the same i,j,k: merge both into
221                ! the one listed first (of those two).
222                tensor(i)%elems(lii)%v=tensor(i)%elems(lii)%v+tensor(i)%elems(li)%v
223                IF (j>k) THEN
224                   tensor(i)%elems(lii)%j=tensor(i)%elems(li)%k
225                   tensor(i)%elems(lii)%k=tensor(i)%elems(li)%j
226                ENDIF
227
228                ! Shift the rest of the items one place down.
229                DO liii=li+1,n
230                   tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
231                   tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
232                   tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
233                END DO
234                tensor(i)%nelems=tensor(i)%nelems-1
235                ! Here we should stop because the li no longer points to the
236                ! original i,j,k element
237                EXIT
238             ENDIF
239          ENDDO
240       ENDDO
241       n=tensor(i)%nelems
242       DO li=1,n
243          ! Clear new "almost" zero entries and shift rest of the items one place down.
244          ! Make sure not to skip any entries while shifting!
245          DO WHILE (abs(tensor(i)%elems(li)%v) < real_eps)
246             DO liii=li+1,n
247                tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
248                tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
249                tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
250             ENDDO
251             tensor(i)%nelems=tensor(i)%nelems-1
252             if (li > tensor(i)%nelems) THEN
253                EXIT
254             ENDIF
255          ENDDO
256       ENDDO
257
258       n=tensor(i)%nelems
259       DO li=1,n
260          ! Upper triangularize
261          j=tensor(i)%elems(li)%j
262          k=tensor(i)%elems(li)%k
263          IF (j>k) THEN
264             tensor(i)%elems(li)%j=k
265             tensor(i)%elems(li)%k=j
266          ENDIF
267       ENDDO
268
269
270    ENDDO
```

6.7.2.11 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 vector: $\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k} \, a_j$.

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 192 of file tensor.f90.

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

6.7.2.12 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 100 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
```

6.7.2.13 subroutine, public tensor::write_tensor_to_file (character (len=∗), intent(in) *s*, type(coolist), dimension(ndim), intent(in) *t*)

Load a rank-4 tensor coolist from a file definition.

Parameters

s	Destination filename
t	The coolist to write

Definition at line 396 of file tensor.f90.

```
    CHARACTER (LEN=*), INTENT(IN) :: s
    TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: t
    INTEGER :: i,j,k,n
    OPEN(30,file=s)
    DO i=1,ndim
       WRITE(30,*) i,t(i)%nelems
       DO n=1,t(i)%nelems
          j=t(i)%elems(n)%j
          k=t(i)%elems(n)%k
          WRITE(30,*) i,j,k,t(i)%elems(n)%v
       END DO
    END DO
    CLOSE(30)
```

6.7.3 Variable Documentation

6.7.3.1 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
```

6.8 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.
- 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

2016 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.

Copyright

2016 Lesley De Cruz, Jonathan Demaeyer & Sebastian Schubert. 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 bufers will probably have to be defined.

6.8.2 Function/Subroutine Documentation

6.8.2.1 subroutine public 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.

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 61 of file rk2_tl_ad_integrator.f90.

```
61      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ystar
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 ad(t,ystar,y,buf_f0)
67      buf_y1 = y+dt*buf_f0
68      CALL ad(t+dt,ystar,buf_y1,buf_f1)
69      res=y+0.5*(buf_f0+buf_f1)*dt
70      t=t+dt
```

6.8.2.2 subroutine public tl_ad_integrator::init_tl_ad_integrator ( )

Routine to initialise the integration buffers.

Routine to initialise the TL-AD integration bufers.

Definition at line 41 of file rk2_tl_ad_integrator.f90.

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

6.8.2.3 subroutine public 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.

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 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
```

6.8.3 Variable Documentation

6.8.3.1 real(kind=8), dimension(:), allocatable 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
```

6.8.3.2 real(kind=8), dimension(:), allocatable 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
```

6.8.3.3 real(kind=8), dimension(:), allocatable 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
```

6.8.3.4 real(kind=8), dimension(:), allocatable 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
```

6.8.3.5 real(kind=8), dimension(:), allocatable tl_ad_integrator::buf_y1 [private]

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

Buffer to hold the intermediate position 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
```

6.9 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), public 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.

6.9.1 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 !

6.9.2 Function/Subroutine Documentation

6.9.2.1 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)
```

6.9.2.2 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
```

6.9.2.3 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
347     REAL(KIND=8), INTENT(IN)  :: v
348     IF ((abs(v) .ge. real_eps).AND.(j /= 0)) count_elems(j)=count_elems(j)+1
```

6.9.2.4 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.

```
257     INTEGER, INTENT(IN) :: i,j,k
258     REAL(KIND=8), INTENT(IN) :: v
259     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
```

6.9.2.5 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
```

6.9.2.6 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.

6.9.2.7 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.

6.9.2.8 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.

6.9.2.9 subroutine, public tl_ad_tensor::init_adtensor ( )

Routine to initialize the AD tensor.

Definition at line 193 of file tl_ad_tensor.f90.

```
    INTEGER :: i
    INTEGER :: allocstat
    ALLOCATE(adtensor(ndim),count_elems(ndim), stat=allocstat)
    IF (allocstat /= 0) stop "*** Not enough memory ! ***"
    count_elems=0
    CALL compute_adtensor(ad_add_count)

    DO i=1,ndim
       ALLOCATE(adtensor(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_adtensor(ad_coeff)

    CALL simplify(adtensor)

```

6.9.2.10 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
```

6.9.2.11 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
```

6.9.2.12 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)
```

6.9.2.13 real(kind=8) function, dimension(ndim,ndim), public tl_ad_tensor::jacobian_mat (real(kind=8), dimension(0:ndim), intent(in) *ystar*)

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)
```

6.9.2.14 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.

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

6.9.2.15 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.

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

6.9.2.16 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.

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

6.9.3 Variable Documentation

6.9.3.1 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
```

6.9.3.2 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
```

6.9.3.3 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
```

6.9.3.4 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
```

6.10 util Module Reference

Utility module.

Functions/Subroutines

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

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

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

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

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

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

  Initialize a square matrix A as a unit matrix.

6.10.1 Detailed Description

Utility module.

Copyright

2018 Lesley De Cruz & Jonathan Demaeyer. See LICENSE.txt for license information.

6.10.2 Function/Subroutine Documentation

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

Initialize a square matrix A as a unit matrix.

Definition at line 137 of file util.f90.

```
137       REAL(KIND=8), DIMENSION(:,:),INTENT(INOUT) :: a
138       INTEGER :: i,n
139       n=size(a,1)
140       a=0.0d0
141       DO i=1,n
142         a(i,i)=1.0d0
143       END DO
144
```

6.10.2.2 subroutine, public util::init_random_seed ( )

Random generator initialization routine.

Definition at line 62 of file util.f90.

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

Determine if a character is in a string and where.

Remarks

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

Definition at line 45 of file util.f90.

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

6.10.2.4 subroutine, public util::piksrt (integer, intent(in) *k*, integer, dimension(k), intent(inout) *arr*, integer, intent(out) *par*)

Simple card player sorting function.

Definition at line 116 of file util.f90.

```
116      INTEGER, INTENT(IN) :: k
117      INTEGER, DIMENSION(k), INTENT(INOUT) :: arr
118      INTEGER, INTENT(OUT) :: par
119      INTEGER :: i,j,a,b
120
121      par=1
122
123      DO j=2,k
124         a=arr(j)
125         DO i=j-1,1,-1
126            if(arr(i).le.a) EXIT
127            arr(i+1)=arr(i)
128            par=-par
129         END DO
130         arr(i+1)=a
131      ENDDO
132      RETURN
```

6.10.2.5 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)
```

6.10.2.6 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

- procedure(calculate_a), pointer, nopass a
- procedure(calculate_b), pointer, nopass b
- procedure(calculate_c_atm), pointer, nopass c
- procedure(calculate_d), pointer, nopass d
- procedure(calculate_g), pointer, nopass g
- procedure(calculate_s), pointer, nopass s

7.1.1 Detailed Description

Type holding the atmospheric inner products tensors.

Definition at line 53 of file inprod_analytic.f90.

7.1.2 Member Data Documentation

7.1.2.1 procedure(calculate_a), pointer, nopass inprod_analytic::atm_tensors::a [private]

Definition at line 54 of file inprod_analytic.f90.

```
PROCEDURE(calculate_a), POINTER, NOPASS :: a
```

7.1.2.2 procedure(calculate_b), pointer, nopass inprod_analytic::atm_tensors::b [private]

Definition at line 55 of file inprod_analytic.f90.

```
PROCEDURE(calculate_b), POINTER, NOPASS :: b
```

7.1.2.3 procedure(calculate_c_atm), pointer, nopass inprod_analytic::atm_tensors::c [private]

Definition at line 56 of file inprod_analytic.f90.

```
56      PROCEDURE(calculate_c_atm), POINTER, NOPASS :: c
```

7.1.2.4 procedure(calculate_d), pointer, nopass inprod_analytic::atm_tensors::d [private]

Definition at line 57 of file inprod_analytic.f90.

```
57      PROCEDURE(calculate_d), POINTER, NOPASS :: d
```

7.1.2.5 procedure(calculate_g), pointer, nopass inprod_analytic::atm_tensors::g [private]

Definition at line 58 of file inprod_analytic.f90.

```
58      PROCEDURE(calculate_g), POINTER, NOPASS :: g
```

7.1.2.6 procedure(calculate_s), pointer, nopass inprod_analytic::atm_tensors::s [private]

Definition at line 59 of file inprod_analytic.f90.

```
59      PROCEDURE(calculate_s), POINTER, NOPASS :: s
```

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 40 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 42 of file inprod_analytic.f90.

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

Definition at line 42 of file inprod_analytic.f90.

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

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

Definition at line 43 of file inprod_analytic.f90.

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

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

Definition at line 43 of file inprod_analytic.f90.

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

Definition at line 42 of file inprod_analytic.f90.

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

Definition at line 41 of file inprod_analytic.f90.

```
41      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

- procedure(calculate_k), pointer, nopass k
- procedure(calculate_m), pointer, nopass m
- procedure(calculate_c_oc), pointer, nopass c
- procedure(calculate_n), pointer, nopass n
- procedure(calculate_o), pointer, nopass o
- procedure(calculate_w), pointer, nopass w

7.5.1 Detailed Description

Type holding the oceanic inner products tensors.

Definition at line 63 of file inprod_analytic.f90.

7.5.2 Member Data Documentation

7.5.2.1 procedure(calculate_c_oc), pointer, nopass inprod_analytic::ocean_tensors::c [private]

Definition at line 66 of file inprod_analytic.f90.

```
66      PROCEDURE(calculate_c_oc), POINTER, NOPASS :: c
```

7.5.2.2 procedure(calculate_k), pointer, nopass inprod_analytic::ocean_tensors::k [private]

Definition at line 64 of file inprod_analytic.f90.

```
64      PROCEDURE(calculate_k), POINTER, NOPASS :: k
```

7.5.2.3 procedure(calculate_m), pointer, nopass inprod_analytic::ocean_tensors::m [private]

Definition at line 65 of file inprod_analytic.f90.

```
65      PROCEDURE(calculate_m), POINTER, NOPASS :: m
```

7.5.2.4 procedure(calculate_n), pointer, nopass inprod_analytic::ocean_tensors::n [private]

Definition at line 67 of file inprod_analytic.f90.

```
67      PROCEDURE(calculate_n), POINTER, NOPASS :: n
```

7.5.2.5 procedure(calculate_o), pointer, nopass inprod_analytic::ocean_tensors::o [private]

Definition at line 68 of file inprod_analytic.f90.

```
68     PROCEDURE(calculate_o), POINTER, NOPASS :: o
```

7.5.2.6 procedure(calculate_w), pointer, nopass inprod_analytic::ocean_tensors::w [private]

Definition at line 69 of file inprod_analytic.f90.

```
69     PROCEDURE(calculate_w), POINTER, NOPASS :: w
```

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 47 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 48 of file inprod_analytic.f90.

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

Definition at line 49 of file inprod_analytic.f90.

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

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

Definition at line 49 of file inprod_analytic.f90.

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

Definition at line 48 of file inprod_analytic.f90.

```
48       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.
- real(kind=8) function inprod_analytic::calculate_a (i, j)

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

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

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

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

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

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

    Forcing of the atmosphere on the ocean.
- real(kind=8) function inprod_analytic::calculate_m (i, j)

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

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

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

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

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

    Initialisation of the inner product.

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

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

    Dissipation in the atmosphere.

- real(kind=8) params::nuo =0.D0

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

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

  Non-dimensional dissipation in the ocean.
- 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 integrator

  Module with the integration routines.

Functions/Subroutines

- subroutine, public integrator::init_integrator

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

  Routine computing the tendencies of the model.
- subroutine, public 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 integrator::buf_y1

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

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

  Buffer to hold tendencies at the intermediate position.

8.10 rk2_tl_ad_integrator.f90 File Reference

Modules

- module tl_ad_integrator

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

Functions/Subroutines

- subroutine, public tl_ad_integrator::init_tl_ad_integrator

  Routine to initialise the integration buffers.
- subroutine, public 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 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 tl_ad_integrator::buf_y1

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

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

  Module with the integration routines.

Functions/Subroutines

- subroutine, public integrator::init_integrator

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

  Routine computing the tendencies of the model.
- subroutine, public 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 integrator::buf_ka

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

  Buffer B to hold tendencies.

8.12 rk4_tl_ad_integrator.f90 File Reference

Modules

- module tl_ad_integrator

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

Functions/Subroutines

- subroutine, public tl_ad_integrator::init_tl_ad_integrator

  Routine to initialise the integration buffers.
- subroutine, public 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 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 tl_ad_integrator::buf_ka

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

  Routine to copy a coolist.
- 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:

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

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

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

  This version return a matrix.
- subroutine, public tensor::sparse_mul2 (coolist_ij, arr_j, res)

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

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

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

  .
- subroutine, public tensor::add_elem (t, i, j, k, v)

  Subroutine to add element to a coolist.
- subroutine, public tensor::add_check (t, i, j, k, v, dst)

  Subroutine to add element to a coolist and check for overflow. Once the t buffer tensor is full, add it to the destination buffer.
- subroutine, public tensor::add_to_tensor (src, dst)

  Routine to add a rank-3 tensor to another one.
- subroutine, public tensor::print_tensor (t, s)

  Routine to print a rank 3 tensor coolist.
- subroutine, public tensor::write_tensor_to_file (s, t)

  Load a rank-4 tensor coolist from a file definition.
- subroutine, public tensor::load_tensor_from_file (s, t)

  Load a rank-4 tensor coolist from a file definition.

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

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

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), public 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.
- integer function, dimension(size(s)), public util::isin (c, s)

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

    Random generator initialization routine.
- integer function lcg (s)
- subroutine, public util::piksrt (k, arr, par)

    Simple card player sorting function.
- 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 102 of file util.f90.

```
102       integer :: lcg
103       integer(int64) :: s
104       IF (s == 0) THEN
105          s = 104729
106       ELSE
107          s = mod(s, 4294967296_int64)
108       END IF
109       s = mod(s * 279470273_int64, 4294967291_int64)
110       lcg = int(mod(s, int(huge(0), int64)), kind(0))
111     END FUNCTION lcg
```

Index