

Reference Manual

Generated by Doxygen 1.8.11

Contents

1	Modular arbitrary-order ocean-atmosphere model: MAOOAM -- Stochastic Fortran implementation	1
2	Modular arbitrary-order ocean-atmosphere model: The Tangent Linear and Adjoint model	7
3	Modular arbitrary-order ocean-atmosphere model: The MTV and WL parameterizations	9
4	Modules Index	15
4.1	Modules List	15
5	Data Type Index	17
5.1	Data Types List	17
6	File Index	19
6.1	File List	19
7	Module Documentation	21
7.1	aotensor_def Module Reference	21
7.1.1	Detailed Description	22
7.1.2	Function/Subroutine Documentation	22
7.1.2.1	a(i)	22
7.1.2.2	add_count(i, j, k, v)	22
7.1.2.3	coeff(i, j, k, v)	22
7.1.2.4	compute_aotensor(func)	23
7.1.2.5	init_aotensor	23
7.1.2.6	kdelta(i, j)	24
7.1.2.7	psi(i)	24

7.1.2.8	t(i)	24
7.1.2.9	theta(i)	24
7.1.3	Variable Documentation	25
7.1.3.1	aotensor	25
7.1.3.2	count_elems	25
7.1.3.3	real_eps	25
7.2	corr_tensor Module Reference	25
7.2.1	Detailed Description	26
7.2.2	Function/Subroutine Documentation	26
7.2.2.1	init_corr_tensor	26
7.2.3	Variable Documentation	27
7.2.3.1	dumb_mat1	27
7.2.3.2	dumb_mat2	27
7.2.3.3	dumb_vec	28
7.2.3.4	dy	28
7.2.3.5	dyy	28
7.2.3.6	expm	28
7.2.3.7	ydy	28
7.2.3.8	ydyd	29
7.2.3.9	yy	29
7.3	corrmod Module Reference	29
7.3.1	Detailed Description	30
7.3.2	Function/Subroutine Documentation	30
7.3.2.1	corrcomp_from_def(s)	30
7.3.2.2	corrcomp_from_fit(s)	34
7.3.2.3	corrcomp_from_spline(s)	34
7.3.2.4	fs(s, p)	35
7.3.2.5	init_corr	35
7.3.2.6	splint(xa, ya, y2a, n, x, y)	36
7.3.3	Variable Documentation	37

7.3.3.1	<code>corr_i</code>	37
7.3.3.2	<code>corr_i_full</code>	37
7.3.3.3	<code>corr_ij</code>	37
7.3.3.4	<code>corrcomp</code>	38
7.3.3.5	<code>inv_corr_i</code>	38
7.3.3.6	<code>inv_corr_i_full</code>	38
7.3.3.7	<code>khi</code>	38
7.3.3.8	<code>klo</code>	38
7.3.3.9	<code>mean</code>	38
7.3.3.10	<code>mean_full</code>	39
7.3.3.11	<code>nspl</code>	39
7.3.3.12	<code>xa</code>	39
7.3.3.13	<code>y2</code>	39
7.3.3.14	<code>ya</code>	39
7.4	<code>dec_tensor</code> Module Reference	40
7.4.1	Detailed Description	41
7.4.2	Function/Subroutine Documentation	41
7.4.2.1	<code>init_dec_tensor</code>	41
7.4.2.2	<code>init_sub_tensor(t, cst, v)</code>	45
7.4.2.3	<code>reorder(t, cst, v)</code>	46
7.4.2.4	<code>suppress_and(t, cst, v1, v2)</code>	46
7.4.2.5	<code>suppress_or(t, cst, v1, v2)</code>	47
7.4.3	Variable Documentation	48
7.4.3.1	<code>bxxx</code>	48
7.4.3.2	<code>bxyy</code>	48
7.4.3.3	<code>bxyy</code>	48
7.4.3.4	<code>byxx</code>	48
7.4.3.5	<code>byxy</code>	48
7.4.3.6	<code>byyy</code>	49
7.4.3.7	<code>dumb</code>	49

7.4.3.8	ff_tensor	49
7.4.3.9	fs_tensor	49
7.4.3.10	hx	49
7.4.3.11	hy	50
7.4.3.12	lxx	50
7.4.3.13	lxy	50
7.4.3.14	lyx	50
7.4.3.15	lyy	50
7.4.3.16	sf_tensor	51
7.4.3.17	ss_tensor	51
7.4.3.18	ss_tl_tensor	51
7.5	ic_def Module Reference	51
7.5.1	Detailed Description	52
7.5.2	Function/Subroutine Documentation	52
7.5.2.1	load_ic	52
7.5.3	Variable Documentation	53
7.5.3.1	exists	53
7.5.3.2	ic	53
7.6	inprod_analytic Module Reference	54
7.6.1	Detailed Description	55
7.6.2	Function/Subroutine Documentation	55
7.6.2.1	b1(Pi, Pj, Pk)	55
7.6.2.2	b2(Pi, Pj, Pk)	56
7.6.2.3	calculate_a	56
7.6.2.4	calculate_b	56
7.6.2.5	calculate_c_atm	57
7.6.2.6	calculate_c_oc	57
7.6.2.7	calculate_d	58
7.6.2.8	calculate_g	58
7.6.2.9	calculate_k	60

7.6.2.10	<code>calculate_m</code>	60
7.6.2.11	<code>calculate_n</code>	61
7.6.2.12	<code>calculate_o</code>	61
7.6.2.13	<code>calculate_s</code>	62
7.6.2.14	<code>calculate_w</code>	62
7.6.2.15	<code>deallocate_inprod</code>	63
7.6.2.16	<code>delta(r)</code>	64
7.6.2.17	<code>flambda(r)</code>	64
7.6.2.18	<code>init_inprod</code>	64
7.6.2.19	<code>s1(Pj, Pk, Mj, Hk)</code>	65
7.6.2.20	<code>s2(Pj, Pk, Mj, Hk)</code>	66
7.6.2.21	<code>s3(Pj, Pk, Hj, Hk)</code>	66
7.6.2.22	<code>s4(Pj, Pk, Hj, Hk)</code>	66
7.6.3	Variable Documentation	66
7.6.3.1	<code>atmos</code>	66
7.6.3.2	<code>awavenum</code>	66
7.6.3.3	<code>ocean</code>	67
7.6.3.4	<code>owavenum</code>	67
7.7	<code>int_comp</code> Module Reference	67
7.7.1	Detailed Description	67
7.7.2	Function/Subroutine Documentation	67
7.7.2.1	<code>integrate(func, ss)</code>	67
7.7.2.2	<code>midexp(func, aa, bb, s, n)</code>	68
7.7.2.3	<code>midpnt(func, a, b, s, n)</code>	68
7.7.2.4	<code>polint(xa, ya, n, x, y, dy)</code>	69
7.7.2.5	<code>qromb(func, a, b, ss)</code>	70
7.7.2.6	<code>qromo(func, a, b, ss, choose)</code>	70
7.7.2.7	<code>trapzd(func, a, b, s, n)</code>	71
7.8	<code>int_corr</code> Module Reference	71
7.8.1	Detailed Description	72

7.8.2	Function/Subroutine Documentation	72
7.8.2.1	comp_corrint	72
7.8.2.2	func_ij(s)	74
7.8.2.3	func_ijkl(s)	74
7.8.2.4	init_corrint	74
7.8.3	Variable Documentation	75
7.8.3.1	corr2int	75
7.8.3.2	corrint	75
7.8.3.3	oi	75
7.8.3.4	oj	75
7.8.3.5	ok	75
7.8.3.6	ol	75
7.8.3.7	real_eps	76
7.9	integrator Module Reference	76
7.9.1	Detailed Description	76
7.9.2	Function/Subroutine Documentation	77
7.9.2.1	init_integrator	77
7.9.2.2	step(y, t, dt, res)	77
7.9.2.3	tendencies(t, y, res)	77
7.9.3	Variable Documentation	78
7.9.3.1	buf_f0	78
7.9.3.2	buf_f1	78
7.9.3.3	buf_ka	78
7.9.3.4	buf_kb	78
7.9.3.5	buf_y1	79
7.10	mar Module Reference	79
7.10.1	Detailed Description	79
7.10.2	Function/Subroutine Documentation	80
7.10.2.1	init_mar	80
7.10.2.2	mar_step(x)	80

7.10.2.3	mar_step_red(xred)	81
7.10.2.4	stoch_vec(dW)	81
7.10.3	Variable Documentation	81
7.10.3.1	buf_y	81
7.10.3.2	dw	82
7.10.3.3	ms	82
7.10.3.4	q	82
7.10.3.5	qred	82
7.10.3.6	rred	82
7.10.3.7	w	82
7.10.3.8	wred	83
7.11	memory Module Reference	83
7.11.1	Detailed Description	83
7.11.2	Function/Subroutine Documentation	84
7.11.2.1	compute_m3(y, dt, dtn, savey, save_ev, evolve, inter, h_int)	84
7.11.2.2	init_memory	85
7.11.2.3	test_m3(y, dt, dtn, h_int)	85
7.11.3	Variable Documentation	86
7.11.3.1	buf_m	86
7.11.3.2	buf_m3	86
7.11.3.3	step	86
7.11.3.4	t_index	86
7.11.3.5	x	87
7.11.3.6	xs	87
7.11.3.7	zs	87
7.12	mtv_int_tensor Module Reference	87
7.12.1	Detailed Description	89
7.12.2	Function/Subroutine Documentation	89
7.12.2.1	init_mtv_int_tensor	89
7.12.3	Variable Documentation	91

7.12.3.1	b1	92
7.12.3.2	b2	92
7.12.3.3	btot	92
7.12.3.4	dumb_mat1	92
7.12.3.5	dumb_mat2	92
7.12.3.6	dumb_mat3	93
7.12.3.7	dumb_mat4	93
7.12.3.8	dumb_vec	93
7.12.3.9	h1	93
7.12.3.10	h2	93
7.12.3.11	h3	94
7.12.3.12	htot	94
7.12.3.13	l1	94
7.12.3.14	l2	94
7.12.3.15	l3	94
7.12.3.16	ltot	95
7.12.3.17	mtot	95
7.12.3.18	q1	95
7.12.3.19	q2	95
7.12.3.20	utot	95
7.12.3.21	vtot	96
7.13	params Module Reference	96
7.13.1	Detailed Description	98
7.13.2	Function/Subroutine Documentation	99
7.13.2.1	init_nml	99
7.13.2.2	init_params	99
7.13.3	Variable Documentation	100
7.13.3.1	ams	100
7.13.3.2	betp	100
7.13.3.3	ca	100

7.13.3.4	co	101
7.13.3.5	cpa	101
7.13.3.6	cpo	101
7.13.3.7	d	101
7.13.3.8	dp	101
7.13.3.9	dt	102
7.13.3.10	epsa	102
7.13.3.11	f0	102
7.13.3.12	g	102
7.13.3.13	ga	102
7.13.3.14	go	103
7.13.3.15	gp	103
7.13.3.16	h	103
7.13.3.17	k	103
7.13.3.18	kd	103
7.13.3.19	kdp	104
7.13.3.20	kp	104
7.13.3.21	l	104
7.13.3.22	lambda	104
7.13.3.23	lpa	104
7.13.3.24	lpo	105
7.13.3.25	lr	105
7.13.3.26	lsbpa	105
7.13.3.27	lsbpo	105
7.13.3.28	n	105
7.13.3.29	natm	106
7.13.3.30	nbatm	106
7.13.3.31	nboc	106
7.13.3.32	ndim	106
7.13.3.33	noc	106

7.13.3.34 oms	107
7.13.3.35 phi0	107
7.13.3.36 phi0_npi	107
7.13.3.37 pi	107
7.13.3.38 r	107
7.13.3.39 rp	108
7.13.3.40 rr	108
7.13.3.41 rra	108
7.13.3.42 sb	108
7.13.3.43 sbpa	108
7.13.3.44 sbpo	109
7.13.3.45 sc	109
7.13.3.46 scale	109
7.13.3.47 sig0	109
7.13.3.48 t_run	109
7.13.3.49 t_trans	110
7.13.3.50 ta0	110
7.13.3.51 to0	110
7.13.3.52 tw	110
7.13.3.53 writeout	110
7.14 rk2_mtv_integrator Module Reference	110
7.14.1 Detailed Description	112
7.14.2 Function/Subroutine Documentation	112
7.14.2.1 compg(y)	112
7.14.2.2 full_step(y, t, dt, dtn, res)	112
7.14.2.3 init_g	113
7.14.2.4 init_integrator	113
7.14.2.5 init_noise	113
7.14.2.6 step(y, t, dt, dtn, res, tend)	114
7.14.3 Variable Documentation	115

7.14.3.1	anoise	115
7.14.3.2	buf_f0	115
7.14.3.3	buf_f1	115
7.14.3.4	buf_g	115
7.14.3.5	buf_y1	115
7.14.3.6	compute_mult	116
7.14.3.7	dw	116
7.14.3.8	dwar	116
7.14.3.9	dwau	116
7.14.3.10	dwmult	116
7.14.3.11	dwor	116
7.14.3.12	dwou	116
7.14.3.13	g	117
7.14.3.14	mult	117
7.14.3.15	noise	117
7.14.3.16	noisemult	117
7.14.3.17	q1fill	117
7.14.3.18	sq2	117
7.15	rk2_ss_integrator Module Reference	118
7.15.1	Detailed Description	118
7.15.2	Function/Subroutine Documentation	119
7.15.2.1	init_ss_integrator	119
7.15.2.2	ss_step(y, ys, t, dt, dtn, res)	119
7.15.2.3	ss_tl_step(y, ys, t, dt, dtn, res)	120
7.15.2.4	tendencies(t, y, res)	120
7.15.2.5	tl_tendencies(t, y, ys, res)	121
7.15.3	Variable Documentation	121
7.15.3.1	anoise	121
7.15.3.2	buf_f0	121
7.15.3.3	buf_f1	121

7.15.3.4	buf_y1	121
7.15.3.5	dwar	122
7.15.3.6	dwor	122
7.16	rk2_stoch_integrator Module Reference	122
7.16.1	Detailed Description	123
7.16.2	Function/Subroutine Documentation	123
7.16.2.1	init_integrator(force)	123
7.16.2.2	step(y, t, dt, dtn, res, tend)	124
7.16.2.3	tendencies(t, y, res)	124
7.16.3	Variable Documentation	125
7.16.3.1	anoise	125
7.16.3.2	buf_f0	125
7.16.3.3	buf_f1	125
7.16.3.4	buf_y1	125
7.16.3.5	dwar	125
7.16.3.6	dwau	126
7.16.3.7	dwor	126
7.16.3.8	dwou	126
7.16.3.9	int_tensor	126
7.17	rk2_wl_integrator Module Reference	126
7.17.1	Detailed Description	127
7.17.2	Function/Subroutine Documentation	127
7.17.2.1	compute_m1(y)	127
7.17.2.2	compute_m2(y)	128
7.17.2.3	full_step(y, t, dt, dtn, res)	128
7.17.2.4	init_integrator	129
7.17.2.5	step(y, t, dt, dtn, res, tend)	129
7.17.3	Variable Documentation	130
7.17.3.1	anoise	130
7.17.3.2	buf_f0	131

7.17.3.3	buf_f1	131
7.17.3.4	buf_m	131
7.17.3.5	buf_m1	131
7.17.3.6	buf_m2	131
7.17.3.7	buf_m3	131
7.17.3.8	buf_m3s	131
7.17.3.9	buf_y1	131
7.17.3.10	dwar	132
7.17.3.11	dwau	132
7.17.3.12	dwor	132
7.17.3.13	dwou	132
7.17.3.14	x1	132
7.17.3.15	x2	132
7.18	sf_def Module Reference	132
7.18.1	Detailed Description	133
7.18.2	Function/Subroutine Documentation	133
7.18.2.1	load_sf	133
7.18.3	Variable Documentation	135
7.18.3.1	bar	135
7.18.3.2	bau	135
7.18.3.3	bor	135
7.18.3.4	bou	135
7.18.3.5	exists	135
7.18.3.6	ind	135
7.18.3.7	n_res	136
7.18.3.8	n_unres	136
7.18.3.9	rind	136
7.18.3.10	sf	136
7.18.3.11	sl_ind	136
7.18.3.12	sl_rind	136

7.19 sigma Module Reference	137
7.19.1 Detailed Description	137
7.19.2 Function/Subroutine Documentation	137
7.19.2.1 compute_mult_sigma(y)	137
7.19.2.2 init_sigma(mult, Q1fill)	138
7.19.3 Variable Documentation	139
7.19.3.1 dumb_mat1	139
7.19.3.2 dumb_mat2	139
7.19.3.3 dumb_mat3	139
7.19.3.4 dumb_mat4	139
7.19.3.5 ind1	139
7.19.3.6 ind2	139
7.19.3.7 n1	140
7.19.3.8 n2	140
7.19.3.9 rind1	140
7.19.3.10 rind2	140
7.19.3.11 sig1	140
7.19.3.12 sig1r	140
7.19.3.13 sig2	140
7.20 sqrt_mod Module Reference	141
7.20.1 Detailed Description	141
7.20.2 Function/Subroutine Documentation	141
7.20.2.1 chol(A, sqA, info)	141
7.20.2.2 csqrtm_triu(A, sqA, info, bs)	142
7.20.2.3 init_sqrt	143
7.20.2.4 rsf2csf(T, Z, Tz, Zz)	143
7.20.2.5 selectev(a, b)	144
7.20.2.6 sqrtm(A, sqA, info, info_triu, bs)	144
7.20.2.7 sqrtm_svd(A, sqA, info, info_triu, bs)	145
7.20.2.8 sqrtm_triu(A, sqA, info, bs)	146

7.20.3	Variable Documentation	147
7.20.3.1	lwork	147
7.20.3.2	real_eps	147
7.20.3.3	work	147
7.21	stat Module Reference	148
7.21.1	Detailed Description	148
7.21.2	Function/Subroutine Documentation	148
7.21.2.1	acc(x)	148
7.21.2.2	init_stat	149
7.21.2.3	iter()	149
7.21.2.4	mean()	149
7.21.2.5	reset	149
7.21.2.6	var()	149
7.21.3	Variable Documentation	150
7.21.3.1	i	150
7.21.3.2	m	150
7.21.3.3	mprev	150
7.21.3.4	mtmp	150
7.21.3.5	v	150
7.22	stoch_mod Module Reference	150
7.22.1	Detailed Description	151
7.22.2	Function/Subroutine Documentation	151
7.22.2.1	gasdev()	151
7.22.2.2	stoch_atm_res_vec(dW)	152
7.22.2.3	stoch_atm_unres_vec(dW)	152
7.22.2.4	stoch_atm_vec(dW)	152
7.22.2.5	stoch_oc_res_vec(dW)	153
7.22.2.6	stoch_oc_unres_vec(dW)	153
7.22.2.7	stoch_oc_vec(dW)	153
7.22.2.8	stoch_vec(dW)	154

7.22.3	Variable Documentation	154
7.22.3.1	gset	154
7.22.3.2	iset	154
7.23	stoch_params Module Reference	154
7.23.1	Detailed Description	155
7.23.2	Function/Subroutine Documentation	156
7.23.2.1	init_stoch_params	156
7.23.3	Variable Documentation	156
7.23.3.1	dtn	156
7.23.3.2	dts	156
7.23.3.3	dtsn	156
7.23.3.4	eps_pert	157
7.23.3.5	int_corr_mode	157
7.23.3.6	load_mode	157
7.23.3.7	maxint	157
7.23.3.8	meml	157
7.23.3.9	mems	158
7.23.3.10	mnuti	158
7.23.3.11	mode	158
7.23.3.12	muti	158
7.23.3.13	q_ar	158
7.23.3.14	q_au	159
7.23.3.15	q_or	159
7.23.3.16	q_ou	159
7.23.3.17	t_trans_mem	159
7.23.3.18	t_trans_stoch	159
7.23.3.19	tdelta	160
7.23.3.20	x_int_mode	160
7.24	tensor Module Reference	160
7.24.1	Detailed Description	162

7.24.2	Function/Subroutine Documentation	163
7.24.2.1	<code>add_matc_to_tensor(i, src, dst)</code>	163
7.24.2.2	<code>add_matc_to_tensor4(i, j, src, dst)</code>	164
7.24.2.3	<code>add_to_tensor(src, dst)</code>	165
7.24.2.4	<code>add_vec_ijk_to_tensor4(i, j, k, src, dst)</code>	166
7.24.2.5	<code>add_vec_ikl_to_tensor4(i, k, l, src, dst)</code>	167
7.24.2.6	<code>add_vec_ikl_to_tensor4_perm(i, k, l, src, dst)</code>	168
7.24.2.7	<code>add_vec_jk_to_tensor(j, k, src, dst)</code>	169
7.24.2.8	<code>coo_to_mat_i(i, src, dst)</code>	170
7.24.2.9	<code>coo_to_mat_ij(src, dst)</code>	170
7.24.2.10	<code>coo_to_mat_ik(src, dst)</code>	171
7.24.2.11	<code>coo_to_mat_j(j, src, dst)</code>	171
7.24.2.12	<code>coo_to_vec_jk(j, k, src, dst)</code>	172
7.24.2.13	<code>copy_tensor(src, dst)</code>	172
7.24.2.14	<code>jsparse_mul(coolist_ijk, arr_j, jcoo_ij)</code>	173
7.24.2.15	<code>jsparse_mul_mat(coolist_ijk, arr_j, jcoo_ij)</code>	174
7.24.2.16	<code>load_tensor4_from_file(s, t)</code>	174
7.24.2.17	<code>mat_to_coo(src, dst)</code>	175
7.24.2.18	<code>matc_to_coo(src, dst)</code>	175
7.24.2.19	<code>print_tensor(t)</code>	176
7.24.2.20	<code>print_tensor4(t)</code>	177
7.24.2.21	<code>scal_mul_coo(s, t)</code>	177
7.24.2.22	<code>simplify(tensor)</code>	177
7.24.2.23	<code>sparse_mul2_j(coolist_ijk, arr_j, res)</code>	178
7.24.2.24	<code>sparse_mul2_k(coolist_ijk, arr_k, res)</code>	179
7.24.2.25	<code>sparse_mul3(coolist_ijk, arr_j, arr_k, res)</code>	179
7.24.2.26	<code>sparse_mul3_mat(coolist_ijk, arr_k, res)</code>	180
7.24.2.27	<code>sparse_mul3_with_mat(coolist_ijk, mat_jk, res)</code>	181
7.24.2.28	<code>sparse_mul4(coolist_ijkl, arr_j, arr_k, arr_l, res)</code>	181
7.24.2.29	<code>sparse_mul4_mat(coolist_ijkl, arr_k, arr_l, res)</code>	182

7.24.2.30	<code>sparse_mul4_with_mat_jl(coolist_ijkl, mat_jl, res)</code>	183
7.24.2.31	<code>sparse_mul4_with_mat_kl(coolist_ijkl, mat_kl, res)</code>	183
7.24.2.32	<code>tensor4_empty(t)</code>	184
7.24.2.33	<code>tensor4_to_coo4(src, dst)</code>	184
7.24.2.34	<code>tensor_empty(t)</code>	185
7.24.2.35	<code>tensor_to_coo(src, dst)</code>	185
7.24.2.36	<code>write_tensor4_to_file(s, t)</code>	186
7.24.3	Variable Documentation	186
7.24.3.1	<code>real_eps</code>	187
7.25	<code>tl_ad_integrator</code> Module Reference	187
7.25.1	Detailed Description	187
7.25.2	Function/Subroutine Documentation	188
7.25.2.1	<code>ad_step(y, ystar, t, dt, res)</code>	188
7.25.2.2	<code>init_tl_ad_integrator</code>	188
7.25.2.3	<code>tl_step(y, ystar, t, dt, res)</code>	188
7.25.3	Variable Documentation	189
7.25.3.1	<code>buf_f0</code>	189
7.25.3.2	<code>buf_f1</code>	189
7.25.3.3	<code>buf_ka</code>	189
7.25.3.4	<code>buf_kb</code>	190
7.25.3.5	<code>buf_y1</code>	190
7.26	<code>tl_ad_tensor</code> Module Reference	190
7.26.1	Detailed Description	191
7.26.2	Function/Subroutine Documentation	191
7.26.2.1	<code>ad(t, ystar, deltat, buf)</code>	191
7.26.2.2	<code>ad_add_count(i, j, k, v)</code>	191
7.26.2.3	<code>ad_add_count_ref(i, j, k, v)</code>	192
7.26.2.4	<code>ad_coeff(i, j, k, v)</code>	192
7.26.2.5	<code>ad_coeff_ref(i, j, k, v)</code>	193
7.26.2.6	<code>compute_adtensor(func)</code>	193

7.26.2.7	<code>compute_adtensor_ref(func)</code>	194
7.26.2.8	<code>compute_tltensor(func)</code>	194
7.26.2.9	<code>init_adtensor</code>	194
7.26.2.10	<code>init_adtensor_ref</code>	194
7.26.2.11	<code>init_tltensor</code>	195
7.26.2.12	<code>jacobian(ystar)</code>	195
7.26.2.13	<code>jacobian_mat(ystar)</code>	196
7.26.2.14	<code>tl(t, ystar, deltat, buf)</code>	196
7.26.2.15	<code>tl_add_count(i, j, k, v)</code>	197
7.26.2.16	<code>tl_coeff(i, j, k, v)</code>	197
7.26.3	Variable Documentation	198
7.26.3.1	<code>adtensor</code>	198
7.26.3.2	<code>count_elems</code>	198
7.26.3.3	<code>real_eps</code>	198
7.26.3.4	<code>tltensor</code>	198
7.27	util Module Reference	198
7.27.1	Detailed Description	199
7.27.2	Function/Subroutine Documentation	199
7.27.2.1	<code>cdiag(A, d)</code>	199
7.27.2.2	<code>choldc(a, p)</code>	200
7.27.2.3	<code>cprintmat(A)</code>	200
7.27.2.4	<code>diag(A, d)</code>	200
7.27.2.5	<code>floordiv(i, j)</code>	200
7.27.2.6	<code>init_one(A)</code>	201
7.27.2.7	<code>init_random_seed()</code>	201
7.27.2.8	<code>invmat(A)</code>	201
7.27.2.9	<code>ireduce(A, Ared, n, ind, rind)</code>	201
7.27.2.10	<code>mat_contract(A, B)</code>	202
7.27.2.11	<code>mat_trace(A)</code>	202
7.27.2.12	<code>printmat(A)</code>	202

7.27.2.13	<code>reduce(A, Ared, n, ind, rind)</code>	202
7.27.2.14	<code>rstr(x, fm)</code>	203
7.27.2.15	<code>str(k)</code>	203
7.27.2.16	<code>triu(A, T)</code>	203
7.27.2.17	<code>vector_outer(u, v, A)</code>	203
7.28	<code>wl_tensor</code> Module Reference	203
7.28.1	Detailed Description	205
7.28.2	Function/Subroutine Documentation	205
7.28.2.1	<code>init_wl_tensor</code>	205
7.28.3	Variable Documentation	208
7.28.3.1	<code>b1</code>	208
7.28.3.2	<code>b14</code>	208
7.28.3.3	<code>b14def</code>	208
7.28.3.4	<code>b2</code>	209
7.28.3.5	<code>b23</code>	209
7.28.3.6	<code>b23def</code>	209
7.28.3.7	<code>b3</code>	209
7.28.3.8	<code>b4</code>	209
7.28.3.9	<code>dumb_mat1</code>	209
7.28.3.10	<code>dumb_mat2</code>	210
7.28.3.11	<code>dumb_mat3</code>	210
7.28.3.12	<code>dumb_mat4</code>	210
7.28.3.13	<code>dumb_vec</code>	210
7.28.3.14	<code>l1</code>	210
7.28.3.15	<code>l2</code>	211
7.28.3.16	<code>l4</code>	211
7.28.3.17	<code>l5</code>	211
7.28.3.18	<code>ldef</code>	211
7.28.3.19	<code>ltot</code>	211
7.28.3.20	<code>m11</code>	211
7.28.3.21	<code>m12</code>	212
7.28.3.22	<code>m12def</code>	212
7.28.3.23	<code>m13</code>	212
7.28.3.24	<code>m1tot</code>	212
7.28.3.25	<code>m21</code>	212
7.28.3.26	<code>m21def</code>	212
7.28.3.27	<code>m22</code>	213
7.28.3.28	<code>m22def</code>	213
7.28.3.29	<code>mdef</code>	213
7.28.3.30	<code>mtot</code>	213

8 Data Type Documentation	215
8.1 <code>inprod_analytic::atm_tensors</code> Type Reference	215
8.1.1 Detailed Description	215
8.1.2 Member Data Documentation	215
8.1.2.1 <code>a</code>	215
8.1.2.2 <code>b</code>	215
8.1.2.3 <code>c</code>	216
8.1.2.4 <code>d</code>	216
8.1.2.5 <code>g</code>	216
8.1.2.6 <code>s</code>	216
8.2 <code>inprod_analytic::atm_wavenum</code> Type Reference	216
8.2.1 Detailed Description	216
8.2.2 Member Data Documentation	216
8.2.2.1 <code>h</code>	216
8.2.2.2 <code>m</code>	217
8.2.2.3 <code>nx</code>	217
8.2.2.4 <code>ny</code>	217
8.2.2.5 <code>p</code>	217
8.2.2.6 <code>typ</code>	217
8.3 <code>tensor::coolist</code> Type Reference	217
8.3.1 Detailed Description	218
8.3.2 Member Data Documentation	218
8.3.2.1 <code>elems</code>	218
8.3.2.2 <code>nelems</code>	218
8.4 <code>tensor::coolist4</code> Type Reference	218
8.4.1 Detailed Description	218
8.4.2 Member Data Documentation	219
8.4.2.1 <code>elems</code>	219
8.4.2.2 <code>nelems</code>	219
8.5 <code>tensor::coolist_elem</code> Type Reference	219

8.5.1	Detailed Description	219
8.5.2	Member Data Documentation	219
8.5.2.1	j	219
8.5.2.2	k	220
8.5.2.3	v	220
8.6	tensor::coolist_elem4 Type Reference	220
8.6.1	Detailed Description	220
8.6.2	Member Data Documentation	220
8.6.2.1	j	220
8.6.2.2	k	221
8.6.2.3	l	221
8.6.2.4	v	221
8.7	inprod_analytic::ocean_tensors Type Reference	221
8.7.1	Detailed Description	221
8.7.2	Member Data Documentation	221
8.7.2.1	c	221
8.7.2.2	k	222
8.7.2.3	m	222
8.7.2.4	n	222
8.7.2.5	o	222
8.7.2.6	w	222
8.8	inprod_analytic::ocean_wavenum Type Reference	222
8.8.1	Detailed Description	223
8.8.2	Member Data Documentation	223
8.8.2.1	h	223
8.8.2.2	nx	223
8.8.2.3	ny	223
8.8.2.4	p	223

9 File Documentation	225
9.1 aotensor_def.f90 File Reference	225
9.2 corr_tensor.f90 File Reference	226
9.3 corrmmod.f90 File Reference	226
9.4 dec_tensor.f90 File Reference	227
9.5 doc/gen_doc.md File Reference	229
9.6 doc/sto_doc.md File Reference	229
9.7 doc/tl_ad_doc.md File Reference	229
9.8 ic_def.f90 File Reference	229
9.9 inprod_analytic.f90 File Reference	229
9.10 int_comp.f90 File Reference	231
9.11 int_corr.f90 File Reference	231
9.12 LICENSE.txt File Reference	232
9.12.1 Function Documentation	233
9.12.1.1 files(the""Software"")	234
9.12.1.2 License(MIT) Copyright(c) 2015-2017 Lesley De Cruz and Jonathan Demaeyer Permission is hereby granted	234
9.12.2 Variable Documentation	234
9.12.2.1 charge	234
9.12.2.2 CLAIM	234
9.12.2.3 conditions	234
9.12.2.4 CONTRACT	234
9.12.2.5 copy	234
9.12.2.6 distribute	234
9.12.2.7 FROM	235
9.12.2.8 IMPLIED	235
9.12.2.9 KIND	235
9.12.2.10 LIABILITY	235
9.12.2.11 MERCHANTABILITY	235
9.12.2.12 merge	235
9.12.2.13 modify	235

9.12.2.14 OTHERWISE	236
9.12.2.15 publish	236
9.12.2.16 restriction	236
9.12.2.17 so	236
9.12.2.18 Software	236
9.12.2.19 sublicense	236
9.12.2.20 use	236
9.13 maoam.f90 File Reference	236
9.13.1 Function/Subroutine Documentation	237
9.13.1.1 maoam	237
9.14 maoam_MTV.f90 File Reference	237
9.14.1 Function/Subroutine Documentation	237
9.14.1.1 maoam_mtv	237
9.15 maoam_stoch.f90 File Reference	237
9.15.1 Function/Subroutine Documentation	238
9.15.1.1 maoam_stoch	238
9.16 maoam_WL.f90 File Reference	238
9.16.1 Function/Subroutine Documentation	238
9.16.1.1 maoam_wl	238
9.17 MAR.f90 File Reference	238
9.18 memory.f90 File Reference	239
9.19 MTV_int_tensor.f90 File Reference	240
9.20 MTV_sigma_tensor.f90 File Reference	241
9.21 params.f90 File Reference	242
9.22 rk2_integrator.f90 File Reference	245
9.23 rk2_MTV_integrator.f90 File Reference	245
9.24 rk2_ss_integrator.f90 File Reference	246
9.25 rk2_stoch_integrator.f90 File Reference	247
9.26 rk2_tl_ad_integrator.f90 File Reference	248
9.27 rk2_WL_integrator.f90 File Reference	248

9.28 rk4_integrator.f90 File Reference	249
9.29 rk4_tl_ad_integrator.f90 File Reference	250
9.30 sf_def.f90 File Reference	250
9.31 sqrt_mod.f90 File Reference	251
9.32 stat.f90 File Reference	252
9.33 stoch_mod.f90 File Reference	252
9.34 stoch_params.f90 File Reference	253
9.35 tensor.f90 File Reference	254
9.36 test_aotensor.f90 File Reference	257
9.36.1 Function/Subroutine Documentation	257
9.36.1.1 test_aotensor	257
9.37 test_corr.f90 File Reference	257
9.37.1 Function/Subroutine Documentation	257
9.37.1.1 test_corr	257
9.38 test_corr_tensor.f90 File Reference	258
9.38.1 Function/Subroutine Documentation	258
9.38.1.1 test_corr_tensor	258
9.39 test_dec_tensor.f90 File Reference	258
9.39.1 Function/Subroutine Documentation	258
9.39.1.1 test_dec_tensor	258
9.40 test_inprod_analytic.f90 File Reference	258
9.40.1 Function/Subroutine Documentation	259
9.40.1.1 inprod_analytic_test	259
9.41 test_MAR.f90 File Reference	259
9.41.1 Function/Subroutine Documentation	259
9.41.1.1 test_mar	259
9.42 test_memory.f90 File Reference	259
9.42.1 Function/Subroutine Documentation	260
9.42.1.1 test_memory	260
9.43 test_MTV_int_tensor.f90 File Reference	260

9.43.1	Function/Subroutine Documentation	260
9.43.1.1	test_mtv_int_tensor	260
9.44	test_MTV_sigma_tensor.f90 File Reference	260
9.44.1	Function/Subroutine Documentation	260
9.44.1.1	test_sigma	260
9.45	test_sqrtm.f90 File Reference	261
9.45.1	Function/Subroutine Documentation	261
9.45.1.1	test_sqrtm	261
9.46	test_tl_ad.f90 File Reference	261
9.46.1	Function/Subroutine Documentation	261
9.46.1.1	test_tl_ad	261
9.47	test_WL_tensor.f90 File Reference	261
9.47.1	Function/Subroutine Documentation	262
9.47.1.1	test_wl_tensor	262
9.48	tl_ad_tensor.f90 File Reference	262
9.49	util.f90 File Reference	263
9.49.1	Function/Subroutine Documentation	264
9.49.1.1	lcg(s)	264
9.50	WL_tensor.f90 File Reference	264
Index		267

Chapter 1

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

About

(c) 2013-2017 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: MAOOAM v1.0, Geosci. Model Dev., 9, 2793-2808, [doi:10.5194/gmd-9-2793-2016](#), 2016.

for the MAOOAM original code, and with

- for the stochastic part.

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

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

Consult the MAOOAM [code repository](#) for updates, and [our website](#) for additional resources.

A pdf version of this manual is available [here](#).

Installation

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

By default, gfortran is selected. To select one or the other, simply modify the Makefile accordingly. If gfortran is selected, the code should be compiled with gfortran 4.7+ (allows for allocatable arrays in namelists). If ifort is selected, the code has been tested with the version 14.0.2 and we do not guarantee compatibility with older compiler version.

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

Remark: The command "make clean" removes the compiled files.

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.

The following files are part of the MAOOAM model alone:

- [maooam.f90](#) : Main program.
- [aotensor_def.f90](#) : Tensor aotensor computation module.
- [IC_def.f90](#) : A module which loads the user specified initial condition.
- [inprod_analytic.f90](#) : Inner products computation module.
- [rk2_integrator.f90](#) : A module which contains the Heun integrator for the model equations.
- [rk4_integrator.f90](#) : A module which contains the RK4 integrator for the model equations.
- Makefile : The Makefile.
- gfortran.mk : Gfortran compiler options file.
- ifort.mk : Ifort compiler options file.
- [params.f90](#) : The model parameters module.
- [tl_ad_tensor.f90](#) : Tangent Linear (TL) and Adjoint (AD) model tensors definition module
- [rk2_tl_ad_integrator.f90](#) : Heun Tangent Linear (TL) and Adjoint (AD) model integrators module
- [rk4_tl_ad_integrator.f90](#) : RK4 Tangent Linear (TL) and Adjoint (AD) model integrators module
- [test_tl_ad.f90](#) : Tests for the Tangent Linear (TL) and Adjoint (AD) model versions
- README.md : A read me file.
- [LICENSE.txt](#) : The license text of the program.
- [util.f90](#) : A module with various useful functions.
- [tensor.f90](#) : Tensor utility module.
- [stat.f90](#) : A module for statistic accumulation.
- [params.nml](#) : A namelist to specify the model parameters.
- [int_params.nml](#) : A namelist to specify the integration parameters.
- [modeselection.nml](#) : A namelist to specify which spectral decomposition will be used.

with the addition of the files:

- [maooam_stoch.f90](#) : Stochastic implementation of MAOOAM.
- [maooam_MTV.f90](#) : Main program - MTV implementation for MAOOAM.
- [maooam_WL.f90](#) : Main program - WL implementation for MAOOAM.
- [corrmod.f90](#) : Unresolved variables correlation matrix initialization module.
- [corr_tensor.f90](#) : Correlations and derivatives for the memory term of the WL parameterization.
- [dec_tensor.f90](#) : Tensor resolved-unresolved components decomposition module.

- [int_comp.f90](#) : Utility module containing the routines to perform the integration of functions.
- [int_corr.f90](#) : Module to compute or load the integrals of the correlation matrices.
- [MAR.f90](#) : Multidimensional AutoRegressive (MAR) module to generate the correlation for the WL parameterization.
- [memory.f90](#) : WL parameterization memory term M_3 computation module.
- [MTV_int_tensor.f90](#) : MTV tensors computation module.
- [MTV_sigma_tensor.f90](#) : MTV noise sigma matrices computation module.
- [WL_tensor.f90](#) : WL tensors computation module.
- [rk2_stoch_integrator.f90](#) : Stochastic RK2 integration routines module.
- [rk2_ss_integrator.f90](#) : Stochastic uncoupled resolved nonlinear and tangent linear RK2 dynamics integration module.
- [rk2_MTV_integrator.f90](#) : MTV RK2 integration routines module.
- [rk2_WL_integrator.f90](#) : WL RK2 integration routines module.
- [sf_def.f90](#) : Module to select the resolved-unresolved components.
- [SF.nml](#) : A namelist to select the resolved-unresolved components.
- [sqrt_mod.f90](#) : Utility module with various routine to compute matrix square root.
- [stoch_mod.f90](#) : Utility module containing the stochastic related routines.
- [stoch_params.f90](#) : Stochastic models parameters module.
- [stoch_params.nml](#) : A namelist to specify the stochastic models parameters.

which belong specifically to the stochastic implementation.

MAOOAM 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 $AMS(NBATM,2)$.
- The first dimension specifies the number attributed by the user to the block and the second dimension specifies the x and the y wavenumbers.
- The VDDG model, described in Vannitsem et al. (2015) is given as an example in the archive.
- Note that the variables of the model are numbered according to the chosen order of the blocks.

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

```
./maooam
```

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

```
./maooam
```

It will generate two files :

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

The tangent linear and adjoint models of MAOOAM are provided in the [tl_ad_tensor](#), rk2_tl_ad_integrator and rk4_tl_ad_integrator modules. It is documented [here](#).

Stochastic code usage

The user first has to fill the MAOOAM model namelist files according to their needs (see the previous section). Additional namelist files for the fine tuning of the parameterization must then be filled, and some "definition" files (with the extension .def) must be provided. An example is provided with the code.

Full details over the parameterization options and definition files are available [here](#).

The program "maooam_stoch" will generate the evolution of the full stochastic dynamics with the command:

```
./maooam_stoch
```

or any other dynamics if specified as an argument (see the header of [maooam_stoch.f90](#)). It will generate two files :

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

The program "maooam_MTV" will generate the evolution of the MTV parameterization evolution, with the command:

```
./maooam_MTV
```

It will generate three files :

- evol_MTV.dat : the recorded time evolution of the variables.
- ptend_MTV.dat : the recorded time evolution of the tendencies (used for debugging).
- mean_field_MTV.dat : the mean field (the climatology)

The program "maooam_WL" will generate the evolution of the MTV parameterization evolution, with the command:

```
./maooam_WL
```

It will generate three files :

- evol_WL.dat : the recorded time evolution of the variables.
- ptend_WL.dat : the recorded time evolution of the tendencies (used for debugging).
- mean_field_WL.dat : the mean field (the climatology)

MAOOAM Implementation notes

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

$$\frac{dz_i}{dt} = \sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} z_k z_j$$

with $z_0 = 1$.

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 dz_i/dt proportional to $z_j z_k$.
- Furthermore, z_0 is always equal to 1, so that $\mathcal{T}_{i,0,0}$ is the constant contribution to dz_i/dt
- $\mathcal{T}_{i,j,0} + \mathcal{T}_{i,0,j}$ is the contribution to dz_i/dt which is linear in z_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.

Stochastic code implementation notes

A stochastic version of MAOOAM and two stochastic parameterization methods (MTV and WL) are provided with this code.

The stochastic version of MAOOAM is given by

$$\frac{dz}{dt} = f(z) + \mathbf{q} \cdot d\mathbf{W}(t)$$

where $d\mathbf{W}$ is a vector of standard Gaussian White noise and where several choice for $f(z)$ are available. For instance, the default choice is to use the full dynamics:

$$f(z) = \sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} z_k z_j.$$

The implementation uses the tensorial framework described above and add some noise to it. This stochastic version is further detailed [here](#).

The MTV parameterization for MAOOAM is given by

$$\frac{dx}{dt} = F_x(x) + \frac{1}{\delta} R(x) + G(x) + \sqrt{2} \sigma(x) \cdot d\mathbf{W}$$

where x is the set of resolved variables and $d\mathbf{W}$ is a vector of standard Gaussian White noise. F_x is the set of tendencies of resolved system alone and δ is the timescale separation parameter.

The WL parameterizations for MAOOAM is given by

$$\frac{dx}{dt} = F_x(x) + \varepsilon M_1(x) + \varepsilon^2 M_2(x, t) + \varepsilon^2 M_3(x, t)$$

where ε is the resolved-unresolved components coupling strenght and where the different terms M_i account for different effect.

The implementation for these two approaches uses the tensorial framework described above, with the addition of new tensors to account for the terms $R, G, \sigma, M_1, M_2, M_3$. They are detailed more completely [here](#).

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

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

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

where y^* is the point where the Tangent model is defined (with $z_0^* = 1$).

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_tlensor\(\)](#) 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

Modular arbitrary-order ocean-atmosphere model: The MTV and WL parameterizations

The stochastic version of MAOOAM

The stochastic version of MAOOAM is given by

$$\frac{dz}{dt} = f(z) + \mathbf{q} \cdot d\mathbf{W}(t)$$

where $d\mathbf{W}$ is a vector of standard Gaussian White noise and where several choice for $f(z)$ are available. For instance, the default choice is to use the full dynamics:

$$f(z) = \sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} z_k z_j.$$

The implementation uses thus the tensorial framework of MAOOAM and add some noise to it. To study parameterization methods in MAOOAM, the models variables z is divided in two components: the resolved component x and the unresolved component y (see below for more details).

Since MAOOAM is a ocean-atmosphere model, it can be decomposed further into oceanic and atmospheric components:

$$z = \{x_a, x_o, y_a, y_o\}$$

and in the present implementation, the noise amplitude can be set in each component:

$$\frac{dx_a}{dt} = f_{x,a}(z) + \mathbf{q}_{x,a} \cdot d\mathbf{W}_{x,a}(t)$$

$$\frac{dx_o}{dt} = f_{x,o}(z) + \mathbf{q}_{x,o} \cdot d\mathbf{W}_{x,o}(t)$$

$$\frac{dy_a}{dt} = f_{y,a}(z) + \mathbf{q}_{y,a} \cdot d\mathbf{W}_{y,a}(t)$$

$$\frac{dy_o}{dt} = f_{y,o}(z) + \mathbf{q}_{y,o} \cdot d\mathbf{W}_{y,o}(t)$$

through the parameters `stoch_params::q_ar`, `stoch_params::q_au`, `stoch_params::q_or` and `stoch_params::q_ou`.

The resolved-unresolved components

Due to the decomposition into resolved variables x and unresolved variables y , the equation of the MAOOAM model can be rewritten:

$$\begin{aligned}\frac{dx}{dt} &= H^x + L^{xx} \cdot x + L^{xy} \cdot y + B^{xxx} : x \otimes x + B^{xxy} : x \otimes y + B^{xyy} : y \otimes y + q_x \cdot dW_x \\ \frac{dy}{dt} &= H^y + L^{yx} \cdot x + L^{yy} \cdot y + B^{yxx} : x \otimes x + B^{yyx} : x \otimes y + B^{yyy} : y \otimes y + q_y \cdot dW_y\end{aligned}$$

where $q_x = \{q_{x,a}, q_{x,o}\}$ and $q_y = \{q_{y,a}, q_{y,o}\}$. We have thus also $dW_x = \{dW_{x,a}, dW_{x,o}\}$ and $dW_y = \{dW_{y,a}, dW_{y,o}\}$. The various terms of the equations above are accessible in the `dec_tensor` module. To specify which variables belong to the resolved (unresolved) component, the user must fill the SF.nml namelist file by setting the component of the vector `sf_def::sf` to 0 (1). This file must be filled before starting any of the stochastic and parameterization codes. If this file is not present, launch one of the programs. It will generate a new SF.nml file and then abort.

The purpose of the parameterization is to reduce the x equation by closing it while keeping the statistical properties of the full system. To apply the parameterizations proposed in this implementation, we consider a modified version of the equation above:

$$\begin{aligned}\frac{dx}{dt} &= F_x(x) + q_x \cdot dW_x + \frac{\varepsilon}{\delta} \Psi_x(x, y) \\ \frac{dy}{dt} &= \frac{1}{\delta^2} \left(F_y(y) + \delta q_y \cdot dW_y \right) + \frac{\varepsilon}{\delta} \Psi_y(x, y)\end{aligned}$$

where ε is the resolved-unresolved components coupling strength given by the parameter `stoch_params::eps_pert`. δ is the timescale separation parameter given by the parameter `stoch_params::tdelta`. By setting those to 1, one recover the first equations above.

The function Ψ_x includes all the x terms, and thus F_x and Ψ_x are unequivocally defined. On the other hand, depending on the value of the parameter `stoch_params::mode`, the terms regrouped in the function F_y can be different. Indeed, if `stoch_params::mode` is set to

- 'qfst', then:

$$F_y(y) = B^{yyy} : y \otimes y$$

- 'ures', then:

$$F_y(y) = H^y + L^{yy} \cdot y + B^{yyy} : y \otimes y$$

However, for the WL parameterization, this parameter must be set to 'ures' by definition. See the article accompanying this code for more details.

The MTV parameterization

This parameterization is also called homogenization. Its acronym comes from the names of the authors that proposed this approach for climate modes: Majda, Timofeyev and Vanden Eijnden (Majda et al., 2001). It is given by

$$\frac{dx}{dt} = F_X(x) + \frac{1}{\delta} R(x) + G(x) + \sqrt{2} \sigma(x) \cdot dW$$

where x is the set of resolved variables and dW is a vector of standard Gaussian White noise. F_x is the set of tendencies of resolved system alone and δ is the timescale separation parameter.

Correlations specification

The ingredients needed to compute the terms R, G, σ of this parametrization are the unresolved variables covariance matrix and the integrated correlation matrices. The unresolved variables covariance matrix is given by

$$\sigma_y = \langle \mathbf{y} \otimes \mathbf{y} \rangle$$

and is present in the implementation through the matrices `corrmod::corr_i` and `corrmod::corr_i_full`. Their inverses are also available through `corrmod::inv_corr_i` and `corrmod::inv_corr_i_full`. The integrated correlation matrices are given by

$$\Sigma = \int_0^\infty ds \langle \mathbf{y} \otimes \mathbf{y}^s \rangle$$

$$\Sigma_2 = \int_0^\infty ds (\langle \mathbf{y} \otimes \mathbf{y}^s \rangle \otimes \langle \mathbf{y} \otimes \mathbf{y}^s \rangle)$$

and is present in the implementation through the matrices `int_corr::corrint` and `int_corr::corr2int`.

These matrices are computed from the correlation matrix $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ which is accessible through the function `corrmod::corrcomp`. For instance, the covariance matrix σ_y is then simply the correlation matrix at the lagtime 0, and Σ and Σ_2 can be computed via integration over the lagtime.

There exists three different ways to load the correlation matrix, specified by the value of the parameters `stoch_params::load_mode` and `stoch_params::int_corr_mode`. The `stoch_params::load_mode` specify how the correlation matrix is loaded can take three different values:

- 'defi': from an analytical definition encoded in the corrmod module function `corrmod::corrcomp_from_def`.
- 'spli': from a spline definition file 'corrspline.def'.
- 'expo': from a fit with exponentials definition file 'correxpo.def'

The `stoch_params::int_corr_mode` specify how the correlation are integrated and can take two different values:

- 'file': Integration results provided by files 'corrint.def' and 'corr2int.def'
- 'prog': Integration computed directly by the program with the correlation matrix. Write 'corrint.def' and 'corr2int.def' files to be reused later.

These parameters can be set up in the namelist file `stoch_params.nml`. Examples of the ".def" files specifying the integrals are provided with the code.

Other MTV setup parameters

Some additional parameters complete the options possible for the MTV parameters :

- `stoch_params::mnuti` : Multiplicative noise update time interval – Time interval over which the matrix $\sigma(x)$ is updated.
- `stoch_params::t_trans_stoch` : Transient period of the stochastic model.
- `stoch_params::maxint` : Specify the upper limit of the numerical integration if `stoch_params::int_corr_mode` is set to 'prog'.

Definition files

The following definition files are needed by the parameterization, depending on the value of the parameters described above. Examples of those files are joined to the code. The files include:

- 'correxpo.def': Coefficients a_i of the fit of the correlations with the function

$$a_4 + a_0 \exp\left(-\frac{t}{a_1}\right) \cos(a_2 t + a_3)$$

where t is the lag-time and τ is the decorrelation time. Used if `stoch_params::load_mode` is set to 'expo'.

- 'corr spline.def': Coefficients b_i of the spline used to model the correlation functions. Used if `stoch_params::load_mode` is set to 'spli'.
- 'corrint.def': File holding the matrix Σ . Used if `stoch_params::int_corr_mode` is set to 'file'.
- 'corr2int.def': File holding the matrix Σ_2 .

The WL parameterization

This parameterization is based on the Ruelle response theory. Its acronym comes from the names of the authors that proposed this approach: Wouters and Lucarini (Wouters and Lucarini, 2012). It is given by

$$\frac{d\mathbf{x}}{dt} = F_x(\mathbf{x}) + \varepsilon M_1(\mathbf{x}) + \varepsilon^2 M_2(\mathbf{x}, t) + \varepsilon^3 M_3(\mathbf{x}, t)$$

where ε is the resolved-unresolved components coupling strength and where the different terms M_i account for average, correlation and memory effects.

Correlations specification

The ingredients needed to compute the M_i terms of this parametrization are the unresolved variable covariance matrix $\langle \mathbf{y} \otimes \mathbf{y} \rangle$ and correlation matrix $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$. The unresolved variables covariance matrix is given by

$$\sigma_y = \langle \mathbf{y} \otimes \mathbf{y} \rangle$$

and is present in the implementation through the matrices `corrmod::corr_i` and `corrmod::corr_i_full`. Their inverses are also available through `corrmod::inv_corr_i` and `corrmod::inv_corr_i_full`.

The correlation matrix $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ is accessible through the function `corrmod::corrcomp`.

As for the MTV case, there exists three different ways to load the correlation matrix, specified by the value of the parameters `stoch_params::load_mode` and `stoch_params::int_corr_mode`. The `stoch_params::load_mode` specify how the correlation matrix is loaded can take three different values:

- 'defi': from an analytical definition encoded in the corrmod module function `corrmod::corrcomp_from_def`.
- 'spli': from a spline definition file 'corr spline.def'.
- 'expo': from a fit with exponentials definition file 'correxpo.def'

The correlation term M_2 is emulated by an order m multidimensional AutoRegressive (MAR) process:

$$\mathbf{u}_n = \sum_{i=1}^m \mathbf{u}_{n-i} \cdot \mathbf{W}_i + \mathbf{Q} \cdot \boldsymbol{\xi}_n$$

of which the \mathbf{W}_i and \mathbf{Q} matrices are also needed (the $\boldsymbol{\xi}_n$ are vectors of standard Gaussian white noise). It is implemented in the MAR module.

Other WL setup parameters

Some additional parameters complete the options possible for the WL parameters :

- `stoch_params::muti` : Memory term M_3 update time interval.
- `stoch_params::t_trans_stoch` : Transient period of the stochastic model.
- `stoch_params::meml` : Time over which the memory kernel is numerically integrated.
- `stoch_params::t_trans_mem` : Transient period of the stochastic model to initialize the memory term.
- `stoch_params::dts` : Intrinsic resolved dynamics time step.
- `stoch_params::x_int_mode` : Integration mode for the resolved component (not used for the moment – must be set to 'reso').

Note that the `stoch_params::mode` must absolutely be set to 'ures', by definition.

Definition files

The following definition files are needed by the parameterization, depending on the value of the parameters described above. Examples of those files are joined to the code. The files include:

- 'correxpo.def': Coefficients a_i of the fit of the correlations with the function

$$a_4 + a_0 \exp\left(-\frac{t}{a_1}\right) \cos(a_2 t + a_3)$$

where t is the lag-time and τ is the decorrelation time. Used if `stoch_params::load_mode` is set to 'expo'.

- 'corrspline.def': Coefficients b_i of the spline used to model the correlation functions. Used if `stoch_params::load_mode` is set to 'spli'.
- 'MAR_R_params.def': File specifying the $\mathbf{R} = \mathbf{Q}^2$ matrix for the MAR.
- 'MAR_W_params.def': File specifying the \mathbf{W}_i matrices for the MAR.

The various terms are then constructed according to these definition files.

References

- Vannitsem, S., Demaeyer, J., De Cruz, L., and Ghil, M.: Low-frequency variability and heat transport in a loworder nonlinear coupled ocean-atmosphere model, *Physica D: Nonlinear Phenomena*, 309, 71-85, 2015.
- De Cruz, L., Demaeyer, J., & Vannitsem, S.: The Modular Arbitrary-Order Ocean-Atmosphere Model: MA↔OOAM v1.0, *Geoscientific Model Development*, 9(8), 2793-2808, 2016.
- Majda, A. J., Timofeyev, I., & Vanden Eijnden, E.: A mathematical framework for stochastic climate models, *Communications on Pure and Applied Mathematics*, 54(8), 891-974, 2001.
- Franzke, C., Majda, A. J., & Vanden-Eijnden, E.: Low-order stochastic mode reduction for a realistic barotropic model climate, *Journal of the atmospheric sciences*, 62(6), 1722-1745, 2005.
- Wouters, J., & Lucarini, V.: Disentangling multi-level systems: averaging, correlations and memory. *Journal of Statistical Mechanics: Theory and Experiment*, 2012(03), P03003, 2012.
- Demaeyer, J., & Vannitsem, S.: Stochastic parametrization of subgrid-scale processes in coupled ocean-atmosphere systems: benefits and limitations of response theory, *Quarterly Journal of the Royal Meteorological Society*, 143(703), 881-896, 2017.

Please see the main article for the full list of references.

Chapter 4

Modules Index

4.1 Modules List

Here is a list of all modules with brief descriptions:

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	21
corr_tensor	Module to compute the correlations and derivatives used to compute the memory term of the WL parameterization	25
corrmod	Module to initialize the correlation matrix of the unresolved variables	29
dec_tensor	The resolved-unresolved components decomposition of the tensor	40
ic_def	Module to load the initial condition	51
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	54
int_comp	Utility module containing the routines to perform the integration of functions	67
int_corr	Module to compute or load the integrals of the correlation matrices	71
integrator	Module with the integration routines	76
mar	Multidimensional Autoregressive module to generate the correlation for the WL parameterization	79
memory	Module that compute the memory term M_3 of the WL parameterization	83
mtv_int_tensor	The MTV tensors used to integrate the MTV model	87
params	The model parameters module	96
rk2_mtv_integrator	Module with the MTV rk2 integration routines	110

rk2_ss_integrator	Module with the stochastic uncoupled resolved nonlinear and tangent linear rk2 dynamics integration routines	118
rk2_stoch_integrator	Module with the stochastic rk2 integration routines	122
rk2_wl_integrator	Module with the WL rk2 integration routines	126
sf_def	Module to select the resolved-unresolved components	132
sigma	The MTV noise sigma matrices used to integrate the MTV model	137
sqrt_mod	Utility module with various routine to compute matrix square root	141
stat	Statistics accumulators	148
stoch_mod	Utility module containing the stochastic related routines	150
stoch_params	The stochastic models parameters module	154
tensor	Tensor utility module	160
tl_ad_integrator	Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Integrators module	187
tl_ad_tensor	Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Tensors definition module .	190
util	Utility module	198
wl_tensor	The WL tensors used to integrate the model	203

Chapter 5

Data Type Index

5.1 Data Types List

Here are the data types with brief descriptions:

inprod_analytic::atm_tensors	
Type holding the atmospheric inner products tensors	215
inprod_analytic::atm_wavenum	
Atmospheric bloc specification type	216
tensor::coolist	
Coordinate list. Type used to represent the sparse tensor	217
tensor::coolist4	
4d coordinate list. Type used to represent the rank-4 sparse tensor	218
tensor::coolist_elem	
Coordinate list element type. Elementary elements of the sparse tensors	219
tensor::coolist_elem4	
4d coordinate list element type. Elementary elements of the 4d sparse tensors	220
inprod_analytic::ocean_tensors	
Type holding the oceanic inner products tensors	221
inprod_analytic::ocean_wavenum	
Oceanic bloc specification type	222

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

aotensor_def.f90	225
corr_tensor.f90	226
corrmod.f90	226
dec_tensor.f90	227
ic_def.f90	229
inprod_analytic.f90	229
int_comp.f90	231
int_corr.f90	231
maooam.f90	236
maooam_MTV.f90	237
maooam_stoch.f90	237
maooam_WL.f90	238
MAR.f90	238
memory.f90	239
MTV_int_tensor.f90	240
MTV_sigma_tensor.f90	241
params.f90	242
rk2_integrator.f90	245
rk2_MTV_integrator.f90	245
rk2_ss_integrator.f90	246
rk2_stoch_integrator.f90	247
rk2_tl_ad_integrator.f90	248
rk2_WL_integrator.f90	248
rk4_integrator.f90	249
rk4_tl_ad_integrator.f90	250
sf_def.f90	250
sqrt_mod.f90	251
stat.f90	252
stoch_mod.f90	252
stoch_params.f90	253
tensor.f90	254
test_aotensor.f90	257
test_corr.f90	257
test_corr_tensor.f90	258
test_dec_tensor.f90	258

test_inprod_analytic.f90	258
test_MAR.f90	259
test_memory.f90	259
test_MTV_int_tensor.f90	260
test_MTV_sigma_tensor.f90	260
test_sqrtm.f90	261
test_tl_ad.f90	261
test_WL_tensor.f90	261
tl_ad_tensor.f90	262
util.f90	263
WL_tensor.f90	264

Chapter 7

Module Documentation

7.1 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.

7.1.1 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

7.1.2 Function/Subroutine Documentation

7.1.2.1 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
```

7.1.2.2 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
```

7.1.2.3 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)%elems)+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)%elems=n
115     END IF

```

7.1.2.4 subroutine aotensor_def::compute_aotensor (external func) [private]

Subroutine to compute the tensor [aotensor](#).

Parameters

<i>func</i>	External function to be used
-------------	------------------------------

Definition at line 132 of file aotensor_def.f90.

7.1.2.5 subroutine, public aotensor_def::init_aotensor ()

Subroutine to initialise the [aotensor](#) tensor.

Remarks

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

Definition at line 202 of file aotensor_def.f90.

```

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

```

```

211     count_elems=0
212
213     CALL compute_aotensor(add_count)
214
215     DO i=1,ndim
216         ALLOCATE(aotensor(i)%elems(count_elems(i)), stat=allocstat)
217         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
218     END DO
219
220     DEALLOCATE(count_elems, stat=allocstat)
221     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
222
223     CALL compute_aotensor(coeff)
224
225     CALL simplify(aotensor)
226
227     CALL deallocate_inprod ! Clean the inner product tensors
228

```

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

Kronecker delta function.

Definition at line 88 of file aotensor_def.f90.

```

88     INTEGER :: i,j,kdelta
89     kdelta=0
90     IF (i == j) kdelta = 1

```

7.1.2.7 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.

```

64     INTEGER :: i,psi
65     psi = i

```

7.1.2.8 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.

```

82     INTEGER :: i,t
83     t = i + 2 * natm + noc

```

7.1.2.9 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.

```

70     INTEGER :: i,theta
71     theta = i + natm

```

7.1.3 Variable Documentation

7.1.3.1 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
```

7.1.3.2 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
```

7.1.3.3 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
```

7.2 corr_tensor Module Reference

Module to compute the correlations and derivatives used to compute the memory term of the WL parameterization.

Functions/Subroutines

- subroutine, public [init_corr_tensor](#)

Subroutine to initialise the correlations tensors.

Variables

- type([coolist](#)), dimension(:, :), allocatable, public [yy](#)
Coolist holding the $\langle Y \otimes Y^s \rangle$ terms.
- type([coolist](#)), dimension(:, :), allocatable, public [dy](#)
Coolist holding the $\langle \partial_Y \otimes Y^s \rangle$ terms.
- type([coolist](#)), dimension(:, :), allocatable, public [ydy](#)
Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \rangle$ terms.
- type([coolist](#)), dimension(:, :), allocatable, public [dyy](#)
Coolist holding the $\langle \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.
- type([coolist4](#)), dimension(:, :), allocatable, public [ydydyy](#)
Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.
- real(kind=8), dimension(:), allocatable [dumb_vec](#)
Dumb vector to be used in the calculation.
- real(kind=8), dimension(:, :), allocatable [dumb_mat1](#)
Dumb matrix to be used in the calculation.
- real(kind=8), dimension(:, :), allocatable [dumb_mat2](#)
Dumb matrix to be used in the calculation.
- real(kind=8), dimension(:, :), allocatable [expm](#)
*Matrix holding the product $\text{inv_corr}_i * \text{corr}_{ij}$ at time s .*

7.2.1 Detailed Description

Module to compute the correlations and derivatives used to compute the memory term of the WL parameterization.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.2.2 Function/Subroutine Documentation

7.2.2.1 subroutine, public [corr_tensor::init_corr_tensor](#) ()

Subroutine to initialise the correlations tensors.

Definition at line 45 of file [corr_tensor.f90](#).

```

45     INTEGER :: i,j,m,allocstat
46
47     CALL init_corr
48
49     print*, 'Computing the time correlation tensors...'
50
51     ALLOCATE (yy(ndim,mems),dy(ndim,mems), dyy(ndim,mems), stat=allocstat)
52     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
53
54     ALLOCATE (ydy(ndim,mems), ydydyy(ndim,mems), stat=allocstat)
55     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
56
57     ALLOCATE (dumb_vec(ndim), stat=allocstat)

```

```

58     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
59
60     ALLOCATE(dumb_mat1(ndim,ndim), dumb_mat2(ndim,ndim), stat=allocstat)
61     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
62
63     ALLOCATE(expm(n_unres,n_unres), stat=allocstat)
64     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
65
66     DO m=1,mems
67         CALL corrcomp((m-1)*muti)
68
69         ! YY
70         CALL ireduce(dumb_mat2,corr_ij,n_unres,ind,rind)
71         CALL matc_to_coo(dumb_mat2,yy(:,m))
72
73         ! dY
74         expm=matmul(inv_corr_i,corr_ij)
75         CALL ireduce(dumb_mat2,expm,n_unres,ind,rind)
76         CALL matc_to_coo(dumb_mat2,dy(:,m))
77
78         ! YdY
79         DO i=1,n_unres
80             CALL ireduce(dumb_mat2,mean(i)*expm,n_unres,ind,rind)
81             CALL add_matc_to_tensor(ind(i),dumb_mat2,ydy(:,m))
82         ENDDO
83
84         ! dYY
85         dumb_vec(1:n_unres)=matmul(mean,expm)
86         DO i=1,n_unres
87             CALL vector_outer(expm(i,:),dumb_vec(1:n_unres),dumb_mat2(1:n_unres,1:n_unres))
88             CALL ireduce(dumb_mat1,dumb_mat2+transpose(dumb_mat2),n_unres,ind,rind)
89             CALL add_matc_to_tensor(ind(i),dumb_mat1,dyy(:,m))
90         ENDDO
91
92         ! YdYY
93         DO i=1,n_unres
94             DO j=1,n_unres
95                 CALL vector_outer(corr_ij(i,:),expm(j,:),dumb_mat2(1:n_unres,1:n_unres))
96                 CALL ireduce(dumb_mat1,dumb_mat2+transpose(dumb_mat2),n_unres,ind,rind)
97                 CALL add_matc_to_tensor4(ind(i),ind(j),dumb_mat1,ydyy(:,m))
98             ENDDO
99         ENDDO
100     ENDDO
101
102     DEALLOCATE(dumb_mat1, dumb_mat2, stat=allocstat)
103     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
104
105     DEALLOCATE(dumb_vec, stat=allocstat)
106     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
107
108

```

7.2.3 Variable Documentation

7.2.3.1 `real(kind=8), dimension(:,:), allocatable corr_tensor::dumb_mat1` [private]

Dumb matrix to be used in the calculation.

Definition at line 37 of file `corr_tensor.f90`.

```

37  REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: dumb_mat1 !< Dumb matrix to be used in the calculation

```

7.2.3.2 `real(kind=8), dimension(:,:), allocatable corr_tensor::dumb_mat2` [private]

Dumb matrix to be used in the calculation.

Definition at line 38 of file `corr_tensor.f90`.

```

38  REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: dumb_mat2 !< Dumb matrix to be used in the calculation

```

7.2.3.3 `real(kind=8), dimension(:), allocatable corr_tensor::dumb_vec` [private]

Dumb vector to be used in the calculation.

Definition at line 36 of file `corr_tensor.f90`.

```
36  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dumb_vec !< Dumb vector to be used in the calculation
```

7.2.3.4 `type(coolist), dimension(:,,:), allocatable, public corr_tensor::dy`

Coolist holding the $\langle \partial_Y \otimes Y^s \rangle$ terms.

Definition at line 31 of file `corr_tensor.f90`.

```
31  TYPE(coolist), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: dy !< Coolist holding the  \f$\langle \partial_Y
    \otimes Y^s \rangle \f$ terms
```

7.2.3.5 `type(coolist), dimension(:,,:), allocatable, public corr_tensor::dyy`

Coolist holding the $\langle \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.

Definition at line 33 of file `corr_tensor.f90`.

```
33  TYPE(coolist), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: dyy !< Coolist holding the  \f$\langle \partial_Y
    \otimes Y^s \otimes Y^s \rangle \f$ terms
```

7.2.3.6 `real(kind=8), dimension(:,,:), allocatable corr_tensor::expm` [private]

Matrix holding the product $\text{inv_corr}_i \cdot \text{corr}_{ij}$ at time s .

Definition at line 39 of file `corr_tensor.f90`.

```
39  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: expm !< Matrix holding the product inv_corr_i*corr_ij at
    time \f$s\f$
```

7.2.3.7 `type(coolist), dimension(:,,:), allocatable, public corr_tensor::ydy`

Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \rangle$ terms.

Definition at line 32 of file `corr_tensor.f90`.

```
32  TYPE(coolist), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: ydy !< Coolist holding the  \f$\langle Y \otimes
    \partial_Y \otimes Y^s \rangle \f$ terms
```


7.2.3.8 type(coolist4), dimension(:, :), allocatable, public corr_tensor::yddyy

Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.

Definition at line 34 of file corr_tensor.f90.

```
34  TYPE(coolist4), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: yddy !< Coolist holding the  \f$\langle Y \otimes
    \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms
```

7.2.3.9 type(coolist), dimension(:, :), allocatable, public corr_tensor::yy

Coolist holding the $\langle Y \otimes Y^s \rangle$ terms.

Definition at line 30 of file corr_tensor.f90.

```
30  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: yy !< Coolist holding the  \f$\langle Y \otimes Y^s
    \rangle$ terms
```

7.3 corrmmod Module Reference

Module to initialize the correlation matrix of the unresolved variables.

Functions/Subroutines

- subroutine, public `init_corr`
Subroutine to initialise the computation of the correlation.
- subroutine `corrcomp_from_def` (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the definition given inside the module.
- subroutine `corrcomp_from_spline` (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the spline representation.
- subroutine `splint` (xa, ya, y2a, n, x, y)
Routine to compute the spline representation parameters.
- real(kind=8) function `fs` (s, p)
Exponential fit function.
- subroutine `corrcomp_from_fit` (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the exponential representation.

Variables

- `real(kind=8), dimension(:), allocatable, public mean`
Vector holding the mean of the unresolved dynamics (reduced version)
- `real(kind=8), dimension(:), allocatable, public mean_full`
Vector holding the mean of the unresolved dynamics (full version)
- `real(kind=8), dimension(:,,:), allocatable, public corr_i_full`
Covariance matrix of the unresolved variables (full version)
- `real(kind=8), dimension(:,,:), allocatable, public inv_corr_i_full`
Inverse of the covariance matrix of the unresolved variables (full version)
- `real(kind=8), dimension(:,,:), allocatable, public corr_i`
Covariance matrix of the unresolved variables (reduced version)
- `real(kind=8), dimension(:,,:), allocatable, public inv_corr_i`
Inverse of the covariance matrix of the unresolved variables (reduced version)
- `real(kind=8), dimension(:,,:), allocatable, public corr_ij`
Matrix holding the correlation matrix at a given time.
- `real(kind=8), dimension(:,,:), allocatable y2`
Vector holding coefficient of the spline and exponential correlation representation.
- `real(kind=8), dimension(:,,:), allocatable ya`
Vector holding coefficient of the spline and exponential correlation representation.
- `real(kind=8), dimension(:), allocatable xa`
Vector holding coefficient of the spline and exponential correlation representation.
- `integer nspl`
Integers needed by the spline representation of the correlation.
- `integer klo`
- `integer khi`
- `procedure(corrcomp_from_spline), pointer, public corrcomp`
Pointer to the correlation computation routine.

7.3.1 Detailed Description

Module to initialize the correlation matrix of the unresolved variables.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.3.2 Function/Subroutine Documentation

7.3.2.1 subroutine `corrmod::corrcomp_from_def` (`real(kind=8), intent(in) s`) [private]

Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the definition given inside the module.

Parameters

s	time s at which the correlation is computed
---	---

Definition at line 148 of file corrmmod.f90.

```

148     REAL(KIND=8), INTENT(IN) :: s
149     REAL(KIND=8) :: y
150     INTEGER :: i,j
151
152     ! Definition of the corr_ij matrix as a function of time
153
154     corr_ij(10,10)=( (7.66977252307669*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (1.02409061
73830213*cos(&
155         &0.07283568782600224*s))/exp(0.017262015588746404*s) - (0.6434985372062336*sin(0.03959716051207145
4*s&
156         &))/exp(0.06567483898489704*s) + (0.6434985372062335*sin(0.07283568782600224*s))/exp(0.01726201558
8746404*s))
157     corr_ij(10,9)=( (0.6434985372062321*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) - (0.6434985
372062324*cos&
158         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) + (7.669772523076694*sin(0.0395971605120714
54*&
159         &s))/exp(0.06567483898489704*s) + (1.024090617383021*sin(0.07283568782600224*s))/exp(0.01726201558
8746404*s))
160     corr_ij(10,8)=0
161     corr_ij(10,7)=0
162     corr_ij(10,6)=0
163     corr_ij(10,5)=( (-2.236364132659011*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (6.9528041
48086198*cos&
164         &(0.07283568782600224*s))/exp(0.017262015588746404*s) - (1.4494534432272481*sin(0.0395971605120714
54*&
165         &s))/exp(0.06567483898489704*s) - (0.6818177416446283*sin(0.07283568782600224*s))/exp(0.0172620155
88746404*s))
166     corr_ij(10,4)=( (1.4494534432272483*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (0.6818177
416446293*cos&
167         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) - (2.2363641326590127*sin(0.039597160512071
454&
168         &s))/exp(0.06567483898489704*s) + (6.952804148086195*sin(0.07283568782600224*s))/exp(0.0172620155
88746404*s))
169     corr_ij(10,3)=0
170     corr_ij(10,2)=0
171     corr_ij(10,1)=0
172     corr_ij(9,10)=( (-0.6434985372062344*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (0.643498
537206234*cos&
173         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) - (7.669772523076689*sin(0.0395971605120714
54*&
174         &s))/exp(0.06567483898489704*s) - (1.0240906173830204*sin(0.07283568782600224*s))/exp(0.0172620155
88746404*s))
175     corr_ij(9,9)=( (7.66977252307669*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (1.0240906173
830204*cos(&
176         &0.07283568782600224*s))/exp(0.017262015588746404*s) - (0.643498537206233*sin(0.039597160512071454
*s)&
177         &))/exp(0.06567483898489704*s) + (0.6434985372062327*sin(0.07283568782600224*s))/exp(0.017262015588
746404*s))
178     corr_ij(9,8)=0
179     corr_ij(9,7)=0
180     corr_ij(9,6)=0
181     corr_ij(9,5)=( (-1.4494534432272477*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) - (0.6818177
416446249*cos&
182         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) + (2.2363641326590105*sin(0.03959716051207
145&
183         &4*s))/exp(0.06567483898489704*s) - (6.952804148086195*sin(0.07283568782600224*s))/exp(0.017262015
588746404*s))
184     corr_ij(9,4)=( (-2.2363641326590127*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (6.9528041
48086194*cos&
185         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) - (1.4494534432272486*sin(0.039597160512071
454&
186         &s))/exp(0.06567483898489704*s) - (0.6818177416446249*sin(0.07283568782600224*s))/exp(0.017262015
588746404*s))
187     corr_ij(9,3)=0
188     corr_ij(9,2)=0
189     corr_ij(9,1)=0
190     corr_ij(8,10)=0
191
192     corr_ij(8,9)=0
193     corr_ij(8,8)=(9.135647293470983/exp(0.05076718239027029*s) + 2.2233889637758932/exp(0.01628546700064885
4*s))
194     corr_ij(8,7)=0
195     corr_ij(8,6)=0
196     corr_ij(8,5)=0
197     corr_ij(8,4)=0
198     corr_ij(8,3)=( -5.938566084855411/exp(0.05076718239027029*s) + 11.97129741027622/exp(0.01628546700064885

```

```

4*s))
199     corr_ij(8,2)=0
200     corr_ij(8,1)=0
201     corr_ij(7,10)=0
202
203     corr_ij(7,9)=0
204     corr_ij(7,8)=0
205     corr_ij(7,7)=((11.518026982819887*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.05351107
793747755*cos&
206         &cos(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.14054811601869432*sin(0.0293414097268
719&
207         &26*s))/exp(0.04435489221745234*s) + (0.14054811601869702*sin(0.11425932545092894*s))/exp(0.019700
737327669783*s))
208     corr_ij(7,6)=((0.14054811601869532*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) - (0.1405481
1601869702*cos&
209         &cos(0.11425932545092894*s))/exp(0.019700737327669783*s) + (11.518026982819887*sin(0.0293414097268
719&
210         &26*s))/exp(0.04435489221745234*s) + (0.0535110779374777*sin(0.11425932545092894*s))/exp(0.0197007
37327669783*s))
211     corr_ij(7,5)=0
212     corr_ij(7,4)=0
213     corr_ij(7,3)=0
214     corr_ij(7,2)=((-0.732907009016115*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (2.72884503
1386875*cos&
215         &(0.11425932545092894*s))/exp(0.019700737327669783*s) - (2.4717920234033532*sin(0.0293414097268719
26*&
216         &s))/exp(0.04435489221745234*s) - (0.24003801347124257*sin(0.11425932545092894*s))/exp(0.019700737
327669783*s))
217     corr_ij(7,1)=((2.4717920234033532*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.24003801
34712426*cos&
218         &s(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.7329070090161153*sin(0.029341409726871
926&
219         &s))/exp(0.04435489221745234*s) + (2.728845031386876*sin(0.11425932545092894*s))/exp(0.0197007373
27669783*s))
220     corr_ij(6,10)=0
221
222     corr_ij(6,9)=0
223     corr_ij(6,8)=0
224     corr_ij(6,7)=((-0.1405481160186977*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.1405481
1601869713*cos&
225         &cos(0.11425932545092894*s))/exp(0.019700737327669783*s) - (11.518026982819885*sin(0.0293414097268
719&
226         &26*s))/exp(0.04435489221745234*s) - (0.05351107793747755*sin(0.11425932545092894*s))/exp(0.019700
737327669783*s))
227     corr_ij(6,6)=((11.518026982819885*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.05351107
793747768*cos&
228         &cos(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.14054811601869832*sin(0.0293414097268
719&
229         &26*s))/exp(0.04435489221745234*s) + (0.14054811601869707*sin(0.11425932545092894*s))/exp(0.019700
737327669783*s))
230     corr_ij(6,5)=0
231     corr_ij(6,4)=0
232     corr_ij(6,3)=0
233     corr_ij(6,2)=((-0.471792023403353*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) - (0.24003801
34712425*cos&
234         &s(0.11425932545092894*s))/exp(0.019700737327669783*s) + (0.7329070090161155*sin(0.029341409726871
926&
235         &s))/exp(0.04435489221745234*s) - (2.7288450313868755*sin(0.11425932545092894*s))/exp(0.019700737
327669783*s))
236     corr_ij(6,1)=((-0.7329070090161154*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (2.7288450
31386876*cos&
237         &s(0.11425932545092894*s))/exp(0.019700737327669783*s) - (2.4717920234033524*sin(0.029341409726871
926&
238         &s))/exp(0.04435489221745234*s) - (0.24003801347124343*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
239     corr_ij(5,10)=((0.5794534449999711*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (4.1369865
70427212*cos&
240         &(0.07283568782600224*s))/exp(0.017262015588746404*s) - (1.0360597341248128*sin(0.0395971605120714
54*&
241         &s))/exp(0.06567483898489704*s) + (3.167330918996692*sin(0.07283568782600224*s))/exp(0.01726201558
8746404*s))
242     corr_ij(5,9)=((1.0360597341248134*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) - (3.16733091
89966856*cos&
243         &s(0.07283568782600224*s))/exp(0.017262015588746404*s) + (0.5794534449999746*sin(0.039597160512071
454&
244         &s))/exp(0.06567483898489704*s) + (4.1369865704272115*sin(0.07283568782600224*s))/exp(0.017262015
588746404*s))
245     corr_ij(5,8)=0
246     corr_ij(5,7)=0
247     corr_ij(5,6)=0
248     corr_ij(5,5)=((-0.37825091063447547*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (30.09469
0926061638*cos&
249         &cos(0.07283568782600224*s))/exp(0.017262015588746404*s) + (0.16085380971100194*sin(0.039597160512
071&
250         &454*s))/exp(0.06567483898489704*s) - (0.1608538097109995*sin(0.07283568782600224*s))/exp(0.017262
015588746404*s))
251     corr_ij(5,4)=((-0.16085380971100238*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (0.160853

```

```

80971100127&
252      &*cos(0.07283568782600224*s))/exp(0.017262015588746404*s) - (0.37825091063447586*sin(0.03959716051
207&
253      &1454*s))/exp(0.06567483898489704*s) + (30.09469092606163*sin(0.07283568782600224*s))/exp(0.017262
015588746404*s))
254      corr_ij(5,3)=0
255      corr_ij(5,2)=0
256      corr_ij(5,1)=0
257      corr_ij(4,10)=((-1.0360597341248106*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (3.167330
918996689*co&
258      &s(0.07283568782600224*s))/exp(0.017262015588746404*s) - (0.5794534449999716*sin(0.039597160512071
454&
259      &s*s))/exp(0.06567483898489704*s) - (4.1369865704272115*sin(0.07283568782600224*s))/exp(0.017262015
588746404*s))
260      corr_ij(4,9)=((0.5794534449999711*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (4.13698657
04272115*co&
261      &s(0.07283568782600224*s))/exp(0.017262015588746404*s) - (1.0360597341248114*sin(0.039597160512071
454&
262      &s*s))/exp(0.06567483898489704*s) + (3.1673309189966843*sin(0.07283568782600224*s))/exp(0.017262015
588746404*s))
263      corr_ij(4,8)=0
264      corr_ij(4,7)=0
265      corr_ij(4,6)=0
266      corr_ij(4,5)=((0.16085380971100194*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) - (0.1608538
0971100371*&
267      &cos(0.07283568782600224*s))/exp(0.017262015588746404*s) + (0.37825091063447497*sin(0.039597160512
071&
268      &454*s))/exp(0.06567483898489704*s) - (30.094690926061617*sin(0.07283568782600224*s))/exp(0.017262
015588746404*s))
269      corr_ij(4,4)=((-0.37825091063447536*cos(0.039597160512071454*s))/exp(0.06567483898489704*s) + (30.09469
0926061617*&
270      &cos(0.07283568782600224*s))/exp(0.017262015588746404*s) + (0.16085380971100172*sin(0.039597160512
071&
271      &454*s))/exp(0.06567483898489704*s) - (0.16085380971100616*sin(0.07283568782600224*s))/exp(0.01726
2015588746404*s))
272      corr_ij(4,3)=0
273      corr_ij(4,2)=0
274      corr_ij(4,1)=0
275      corr_ij(3,10)=0
276
277      corr_ij(3,9)=0
278      corr_ij(3,8)=(0.24013456462471527/exp(0.05076718239027029*s) + 5.792596760796093/exp(0.0162854670006488
54*s))
279      corr_ij(3,7)=0
280      corr_ij(3,6)=0
281      corr_ij(3,5)=0
282      corr_ij(3,4)=0
283      corr_ij(3,3)=(-0.15609785880208227/exp(0.05076718239027029*s) + 31.18882918422289/exp(0.016285467000648
854*s))
284      corr_ij(3,2)=0
285      corr_ij(3,1)=0
286      corr_ij(2,10)=0
287
288      corr_ij(2,9)=0
289      corr_ij(2,8)=0
290      corr_ij(2,7)=((1.6172201305728584*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.37871789
179790255*co&
291      &os(0.11425932545092894*s))/exp(0.019700737327669783*s) + (1.2889451151208258*sin(0.02934140972687
192&
292      &6*s))/exp(0.04435489221745234*s) + (1.4228849217537705*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
293      corr_ij(2,6)=((-1.2889451151208255*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) - (1.4228849
217537702*co&
294      &os(0.11425932545092894*s))/exp(0.019700737327669783*s) + (1.6172201305728586*sin(0.02934140972687
192&
295      &6*s))/exp(0.04435489221745234*s) + (0.3787178917979035*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
296      corr_ij(2,5)=0
297      corr_ij(2,4)=0
298      corr_ij(2,3)=0
299      corr_ij(2,2)=((0.1789135645266575*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (26.8170244
57844113*co&
300      &s(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.4268927977731004*sin(0.029341409726871
926&
301      &s*s))/exp(0.04435489221745234*s) + (0.4268927977730982*sin(0.11425932545092894*s))/exp(0.019700737
327669783*s))
302      corr_ij(2,1)=((0.4268927977731007*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) - (0.42689279
777309963*co&
303      &os(0.11425932545092894*s))/exp(0.019700737327669783*s) + (0.17891356452665746*sin(0.0293414097268
719&
304      &26*s))/exp(0.04435489221745234*s) + (26.81702445784412*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
305      corr_ij(1,10)=0
306
307      corr_ij(1,9)=0
308      corr_ij(1,8)=0
309      corr_ij(1,7)=((1.288945115120824*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (1.422884921

```

```

7537711*cos&
310   &(0.11425932545092894*s))/exp(0.019700737327669783*s) - (1.617220130572856*sin(0.02934140972687192
6*s&
311   &))/exp(0.04435489221745234*s) - (0.3787178917979028*sin(0.11425932545092894*s))/exp(0.01970073732
7669783*s))
312   corr_ij(1,6)=((1.6172201305728564*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.37871789
179790377*c&
313   &os(0.11425932545092894*s))/exp(0.019700737327669783*s) + (1.2889451151208242*sin(0.02934140972687
192&
314   &6*s))/exp(0.04435489221745234*s) + (1.4228849217537711*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
315   corr_ij(1,5)=0
316   corr_ij(1,4)=0
317   corr_ij(1,3)=0
318   corr_ij(1,2)=((-0.4268927977731002*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (0.4268927
977730981*c&
319   &os(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.1789135645266573*sin(0.02934140972687
192&
320   &6*s))/exp(0.04435489221745234*s) - (26.81702445784412*sin(0.11425932545092894*s))/exp(0.019700737
327669783*s))
321   corr_ij(1,1)=((0.1789135645266574*cos(0.029341409726871926*s))/exp(0.04435489221745234*s) + (26.8170244
57844113*co&
322   &s(0.11425932545092894*s))/exp(0.019700737327669783*s) - (0.42689279777310024*sin(0.02934140972687
192&
323   &6*s))/exp(0.04435489221745234*s) + (0.4268927977730997*sin(0.11425932545092894*s))/exp(0.01970073
7327669783*s))
324
325   corr_ij=q_au**2*corr_ij
326

```

7.3.2.2 subroutine corrmod::corrcomp_from_fit (real(kind=8), intent(in) s) [private]

Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the exponential representation.

Parameters

s	time s at which the correlation is computed
----------	---

Definition at line 399 of file corrmod.f90.

```

399   REAL(KIND=8), INTENT(IN) :: s
400   REAL(KIND=8) :: y
401   INTEGER :: i,j
402
403   corr_ij=0.d0
404   DO i=1,n_unres
405     DO j=1,n_unres
406       corr_ij(i,j)=fs(s,ya(i,j,:))
407     END DO
408   END DO

```

7.3.2.3 subroutine corrmod::corrcomp_from_spline (real(kind=8), intent(in) s) [private]

Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the spline representation.

Parameters

s	time s at which the correlation is computed
----------	---

Definition at line 333 of file corrmod.f90.

```

333   REAL(KIND=8), INTENT(IN) :: s

```

```

334     REAL(KIND=8) :: y
335     INTEGER :: i, j
336     corr_ij=0.d0
337     DO i=1,n_unres
338         DO j=1,n_unres
339             CALL splint(xa,ya(i,j,:),y2(i,j,:),nspl,s,y)
340             corr_ij(i,j)=y
341         END DO
342     END DO

```

7.3.2.4 real(kind=8) function corrmmod::fs (real(kind=8), intent(in) s, real(kind=8), dimension(5), intent(in) p) [private]

Exponential fit function.

Parameters

<i>s</i>	time <i>s</i> at which the function is evaluated
<i>p</i>	vector holding the coefficients of the fit function

Definition at line 388 of file corrmmod.f90.

```

388     REAL(KIND=8), INTENT(IN) :: s
389     REAL(KIND=8), DIMENSION(5), INTENT(IN) :: p
390     REAL(KIND=8) :: fs
391     fs=p(1)*exp(-s/p(2))*cos(p(3)*s+p(4))
392     RETURN

```

7.3.2.5 subroutine, public corrmmod::init_corr ()

Subroutine to initialise the computation of the correlation.

Definition at line 46 of file corrmmod.f90.

```

46     INTEGER :: allocstat,i,j,k,nf
47     REAL(KIND=8), DIMENSION(5) :: dumb
48     LOGICAL :: ex
49
50     ! Selection of the loading mode
51     SELECT CASE (load_mode)
52     CASE ('defi')
53         corrcomp => corrcomp_from_def
54     CASE ('spli')
55         INQUIRE(file='corrspline.def',exist=ex)
56         IF (.not.ex) stop "*** File corrspline.def not found ! ***"
57         OPEN(20,file='corrspline.def',status='old')
58         READ(20,*) nf,nspl
59         IF (nf /= n_unres) stop "*** Dimension in files corrspline.def and sf.nml do not correspond ! ***"
60         ALLOCATE(xa(nspl), ya(n_unres,n_unres,nspl), y2(n_unres,n_unres,nspl),
stat=allocstat)
61         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
62         READ(20,*) xa
63         maxint=xa(nspl)/2
64         DO k=1,n_unres*n_unres
65             READ(20,*) i,j
66             READ(20,*) ya(i,j,:)
67             READ(20,*) y2(i,j,:)
68         ENDDO
69         CLOSE(20)
70         corrcomp => corrcomp_from_spline
71         klo=1
72         khi=nspl
73     CASE ('expo')
74         INQUIRE(file='correxpo.def',exist=ex)
75         IF (.not.ex) stop "*** File correxpo.def not found ! ***"
76         OPEN(20,file='correxpo.def',status='old')

```

```

77      READ(20,*) nf,maxint
78      IF (nf /= n_unres) stop "*** Dimension in files correxpo.def and sf.nml do not correspond ! ***"
79      ALLOCATE(ya(n_unres,n_unres,5), stat=allocstat)
80      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
81      DO k=1,n_unres*n_unres
82          READ(20,*) i,j,dumb
83          ya(i,j,:)=dumb
84      ENDDO
85      CLOSE(20)
86      corrcomp => corrcomp_from_fit
87      CASE DEFAULT
88          stop '*** LOAD_MODE variable not properly defined in corrmmod.nml ***'
89      END SELECT
90
91      ALLOCATE(mean(n_unres),mean_full(0:ndim), stat=allocstat)
92      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
93
94      ALLOCATE(inv_corr_i(n_unres,n_unres), stat=allocstat)
95      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
96
97      ALLOCATE(corr_i(n_unres,n_unres), stat=allocstat)
98      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
99
100     ALLOCATE(corr_ij(n_unres,n_unres), stat=allocstat)
101     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
102
103     ALLOCATE(corr_i_full(ndim,ndim), stat=allocstat)
104     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
105
106     ALLOCATE(inv_corr_i_full(ndim,ndim), stat=allocstat)
107     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
108
109     corr_ij=0.d0
110
111     CALL corrcomp(0.d0)
112     corr_i=corr_ij
113     inv_corr_i=invmat(corr_i)
114
115     corr_i_full=0.d0
116     DO i=1,n_unres
117         DO j=1,n_unres
118             corr_i_full(ind(i),ind(j))=corr_i(i,j)
119         ENDDO
120     ENDDO
121
122     inv_corr_i_full=0.d0
123     DO i=1,n_unres
124         DO j=1,n_unres
125             inv_corr_i_full(ind(i),ind(j))=inv_corr_i(i,j)
126         ENDDO
127     ENDDO
128
129     mean=0.d0
130     INQUIRE(file='mean.def',exist=ex)
131     IF (ex) THEN
132         OPEN(20,file='mean.def',status='old')
133         READ(20,*) mean
134         CLOSE(20)
135     ENDIF
136
137     mean_full=0.d0
138     DO i=1,n_unres
139         mean_full(ind(i))=mean(i)
140     ENDDO
141

```

7.3.2.6 subroutine corrmmod::splint (real(kind=8), dimension(n), intent(in) xa, real(kind=8), dimension(n), intent(in) ya, real(kind=8), dimension(n), intent(in) y2a, integer, intent(in) n, real(kind=8), intent(in) x, real(kind=8), intent(out) y) [private]

Routine to compute the spline representation parameters.

Definition at line 347 of file corrmmod.f90.

```

347     INTEGER, INTENT(IN) :: n
348     REAL(KIND=8), INTENT(IN), DIMENSION(n) :: xa,y2a,ya
349     REAL(KIND=8), INTENT(IN) :: x
350     REAL(KIND=8), INTENT(OUT) :: y

```



```

351     INTEGER :: k
352     REAL(KIND=8) :: a,b,h
353     if ((khi-klo.gt.1).or.(xa(klo).gt.x).or.(xa(khi).lt.x)) then
354         if ((khi-klo.eq.1).and.(xa(klo).lt.x)) then
355             khi=klo
356             DO WHILE (xa(khi).lt.x)
357                 khi=khi+1
358             END DO
359             klo=khi-1
360         else
361             khi=n
362             klo=1
363             DO WHILE (khi-klo.gt.1)
364                 k=(khi+klo)/2
365                 if (xa(k).gt.x) then
366                     khi=k
367                 else
368                     klo=k
369                 endif
370             END DO
371         end if
372         ! print*, "search",x,khi-klo,xa(klo),xa(khi)
373         ! else
374         ! print*, "ok",x,khi-klo,xa(klo),xa(khi)
375     endif
376     h=xa(khi)-xa(klo)
377     if (h.eq.0.) stop 'bad xa input in splint'
378     a=(xa(khi)-x)/h
379     b=(x-xa(klo))/h
380     y=a*ya(klo)+b*ya(khi)+((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.
381     return

```

7.3.3 Variable Documentation

7.3.3.1 `real(kind=8), dimension(:,:), allocatable, public corrmmod::corr_i`

Covariance matrix of the unresolved variables (reduced version)

Definition at line 30 of file corrmmod.f90.

```

30     REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE, PUBLIC :: corr_i !< Covariance matrix of the unresolved
        variables (reduced version)

```

7.3.3.2 `real(kind=8), dimension(:,:), allocatable, public corrmmod::corr_i_full`

Covariance matrix of the unresolved variables (full version)

Definition at line 28 of file corrmmod.f90.

```

28     REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE, PUBLIC :: corr_i_full !< Covariance matrix of the unresolved
        variables (full version)

```

7.3.3.3 `real(kind=8), dimension(:,:), allocatable, public corrmmod::corr_ij`

Matrix holding the correlation matrix at a given time.

Definition at line 32 of file corrmmod.f90.

```

32     REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE, PUBLIC :: corr_ij !< Matrix holding the correlation matrix at
        a given time

```

7.3.3.4 procedure(corrcomp_from_spline), pointer, public corrmmod::corrcomp

Pointer to the correlation computation routine.

Definition at line 41 of file corrmmod.f90.

```
41  PROCEDURE(corrcomp_from_spline), POINTER, PUBLIC :: corrcomp
```

7.3.3.5 real(kind=8), dimension(:,,:), allocatable, public corrmmod::inv_corr_i

Inverse of the covariance matrix of the unresolved variables (reduced version)

Definition at line 31 of file corrmmod.f90.

```
31  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: inv_corr_i !< Inverse of the covariance matrix of
    the unresolved variables (reduced version)
```

7.3.3.6 real(kind=8), dimension(:,,:), allocatable, public corrmmod::inv_corr_i_full

Inverse of the covariance matrix of the unresolved variables (full version)

Definition at line 29 of file corrmmod.f90.

```
29  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: inv_corr_i_full !< Inverse of the covariance matrix
    of the unresolved variables (full version)
```

7.3.3.7 integer corrmmod::khi [private]

Definition at line 38 of file corrmmod.f90.

7.3.3.8 integer corrmmod::klo [private]

Definition at line 38 of file corrmmod.f90.

7.3.3.9 real(kind=8), dimension(:), allocatable, public corrmmod::mean

Vector holding the mean of the unresolved dynamics (reduced version)

Definition at line 26 of file corrmmod.f90.

```
26  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: mean !< Vector holding the mean of the unresolved
    dynamics (reduced version)
```

7.3.3.10 `real(kind=8), dimension(:), allocatable, public corrmmod::mean_full`

Vector holding the mean of the unresolved dynamics (full version)

Definition at line 27 of file corrmmod.f90.

```
27  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: mean_full !< Vector holding the mean of the unresolved
    dynamics (full version)
```

7.3.3.11 `integer corrmmod::nspl [private]`

Integers needed by the spline representation of the correlation.

Definition at line 38 of file corrmmod.f90.

```
38  INTEGER :: nspl, klo, khi
```

7.3.3.12 `real(kind=8), dimension(:), allocatable corrmmod::xa [private]`

Vector holding coefficient of the spline and exponential correlation representation.

Definition at line 35 of file corrmmod.f90.

```
35  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: xa !< Vector holding coefficient of the spline and exponential
    correlation representation
```

7.3.3.13 `real(kind=8), dimension(:, :, :), allocatable corrmmod::y2 [private]`

Vector holding coefficient of the spline and exponential correlation representation.

Definition at line 33 of file corrmmod.f90.

```
33  REAL(KIND=8), DIMENSION(:, :, :), ALLOCATABLE :: y2 !< Vector holding coefficient of the spline and
    exponential correlation representation
```

7.3.3.14 `real(kind=8), dimension(:, :, :), allocatable corrmmod::ya [private]`

Vector holding coefficient of the spline and exponential correlation representation.

Definition at line 34 of file corrmmod.f90.

```
34  REAL(KIND=8), DIMENSION(:, :, :), ALLOCATABLE :: ya !< Vector holding coefficient of the spline and
    exponential correlation representation
```

7.4 dec_tensor Module Reference

The resolved-unresolved components decomposition of the tensor.

Functions/Subroutines

- subroutine [suppress_and](#) (t, cst, v1, v2)
Subroutine to suppress from the tensor t_{ijk} components satisfying $SF(j)=v1$ and $SF(k)=v2$.
- subroutine [suppress_or](#) (t, cst, v1, v2)
Subroutine to suppress from the tensor t_{ijk} components satisfying $SF(j)=v1$ or $SF(k)=v2$.
- subroutine [reorder](#) (t, cst, v)
Subroutine to reorder the tensor t_{ijk} components : if $SF(j)=v$ then it return t_{ikj} .
- subroutine [init_sub_tensor](#) (t, cst, v)
Subroutine that suppress all the components of a tensor t_{ijk} where if $SF(i)=v$.
- subroutine, public [init_dec_tensor](#)
Subroutine that initialize and compute the decomposed tensors.

Variables

- type([coolist](#)), dimension(:), allocatable, public [ff_tensor](#)
Tensor holding the part of the unresolved tensor involving only unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [sf_tensor](#)
Tensor holding the part of the resolved tensor involving unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [ss_tensor](#)
Tensor holding the part of the resolved tensor involving only resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [fs_tensor](#)
Tensor holding the part of the unresolved tensor involving resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [hx](#)
Tensor holding the constant part of the resolved tendencies.
- type([coolist](#)), dimension(:), allocatable, public [lxx](#)
Tensor holding the linear part of the resolved tendencies involving the resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [lxy](#)
Tensor holding the linear part of the resolved tendencies involving the unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [bxxx](#)
Tensor holding the quadratic part of the resolved tendencies involving resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [bxyx](#)
Tensor holding the quadratic part of the resolved tendencies involving both resolved and unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [bxxy](#)
Tensor holding the quadratic part of the resolved tendencies involving unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [hy](#)
Tensor holding the constant part of the unresolved tendencies.
- type([coolist](#)), dimension(:), allocatable, public [lyx](#)
Tensor holding the linear part of the unresolved tendencies involving the resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [lyy](#)
Tensor holding the linear part of the unresolved tendencies involving the unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [byxx](#)
Tensor holding the quadratic part of the unresolved tendencies involving resolved variables.
- type([coolist](#)), dimension(:), allocatable, public [byxy](#)
Tensor holding the quadratic part of the unresolved tendencies involving both resolved and unresolved variables.

- type([coolist](#)), dimension(:), allocatable, public [byyy](#)
Tensor holding the quadratic part of the unresolved tendencies involving unresolved variables.
- type([coolist](#)), dimension(:), allocatable, public [ss_tl_tensor](#)
Tensor of the tangent linear model tendencies of the resolved component alone.
- type([coolist](#)), dimension(:), allocatable [dumb](#)
Dumb coolist to make the computations.

7.4.1 Detailed Description

The resolved-unresolved components decomposition of the tensor.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.4.2 Function/Subroutine Documentation

7.4.2.1 subroutine, public dec_tensor::init_dec_tensor ()

Subroutine that initialize and compute the decomposed tensors.

Definition at line 195 of file dec_tensor.f90.

```

195     USE params, only: ndim
196     USE aotensor\_def, only: aotensor
197     USE sf\_def, only: load\_sf
198     USE tensor, only: copy\_tensor, add\_to\_tensor,
    scal_mul_coo
199     USE tl\_ad\_tensor, only: init\_tltensor, tltensor
200     USE stoch\_params, only: init\_stoch\_params, mode,
    tdelta, eps\_pert
201     INTEGER :: allocstat
202
203     CALL init\_stoch\_params
204
205     CALL init\_tltensor ! and tl tensor
206
207     CALL load\_sf ! Load the resolved-unresolved decomposition
208
209     ! Allocating the returned arrays
210
211     ALLOCATE(ff\_tensor(ndim), fs\_tensor(ndim), sf\_tensor(ndim), ss\_tensor(
    ndim), stat=allocstat)
212     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
213
214     ALLOCATE(ss\_tl\_tensor(ndim), stat=allocstat)
215     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
216
217     ALLOCATE(hx(ndim), lxx(ndim), lxy(ndim), bxxx(ndim), bxyx(ndim), bxyy(
    ndim), stat=allocstat)
218     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
219
220     ALLOCATE(hy(ndim), lyx(ndim), lyy(ndim), byxx(ndim), byxy(ndim), byyy(
    ndim), stat=allocstat)
221     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
222
223     ! General decomposition
224     ! ff tensor
225     ALLOCATE(dumb(ndim), stat=allocstat)
226     IF (allocstat /= 0) stop "*** Not enough memory ! ***"

```

```

227
228 IF (mode.ne.'qfst') THEN
229     CALL copy_tensor(aotensor,dumb) !Copy the tensors
230     CALL init_sub_tensor(dumb,0,0)
231     CALL suppress_or(dumb,1,0,0) ! Clear entries with resolved variables
232     CALL copy_tensor(dumb,ff_tensor)
233 ELSE
234     CALL copy_tensor(aotensor,dumb) !Copy the tensors
235     CALL init_sub_tensor(dumb,0,0)
236     CALL suppress_or(dumb,0,0,0) ! Clear entries with resolved variables and linear and constant terms
237     CALL copy_tensor(dumb,ff_tensor)
238 ENDIF
239
240 allocstat=0
241 DEALLOCATE(dumb, stat=allocstat)
242 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
243
244 ! fs tensor
245 ALLOCATE(dumb(ndim), stat=allocstat)
246 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
247
248 IF (mode.ne.'qfst') THEN
249     CALL copy_tensor(aotensor,dumb) !Copy the tensors
250     CALL init_sub_tensor(dumb,0,0)
251     CALL suppress_and(dumb,1,1,1) ! Clear entries with only unresolved variables and constant
252     CALL copy_tensor(dumb,fs_tensor)
253 ELSE
254     CALL copy_tensor(aotensor,dumb) !Copy the tensors
255     CALL init_sub_tensor(dumb,0,0)
256     CALL suppress_and(dumb,0,1,1) ! Clear entries with only quadratic unresolved variables
257     CALL copy_tensor(dumb,fs_tensor)
258 ENDIF
259
260 allocstat=0
261 DEALLOCATE(dumb, stat=allocstat)
262 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
263
264 ! sf tensor
265 ALLOCATE(dumb(ndim), stat=allocstat)
266 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
267
268
269 CALL copy_tensor(aotensor,dumb) !Copy the tensors
270 CALL init_sub_tensor(dumb,1,1)
271 CALL suppress_and(dumb,0,0,0) ! Clear entries with only unresolved variables and constant
272 CALL copy_tensor(dumb,sf_tensor)
273
274 allocstat=0
275 DEALLOCATE(dumb, stat=allocstat)
276 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
277
278 ! ss tensor
279 ALLOCATE(dumb(ndim), stat=allocstat)
280 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
281
282
283 CALL copy_tensor(aotensor,dumb) !Copy the tensors
284 CALL init_sub_tensor(dumb,1,1)
285 CALL suppress_or(dumb,0,1,1) ! Clear entries with only unresolved variables and constant
286 CALL copy_tensor(dumb,ss_tensor)
287
288 allocstat=0
289 DEALLOCATE(dumb, stat=allocstat)
290 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
291
292 ! ss tangent linear tensor
293
294 ALLOCATE(dumb(ndim), stat=allocstat)
295 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
296
297 CALL copy_tensor(tltensor,dumb) !Copy the tensors
298 CALL init_sub_tensor(dumb,1,1)
299 CALL suppress_or(dumb,0,1,1) ! Clear entries with only unresolved variables and constant
300 CALL copy_tensor(dumb,ss_tl_tensor)
301
302 allocstat=0
303 DEALLOCATE(dumb, stat=allocstat)
304 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
305
306 ! Multiply the aotensor part that need to be by the perturbation and time
307 ! separation parameter
308
309 ALLOCATE(dumb(ndim), stat=allocstat)
310 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
311
312 CALL copy_tensor(ss_tensor,dumb)
313 CALL scal_mul_coo(1.d0/tdelta**2,ff_tensor)

```

```

314 CALL scal_mul_coo(eps_pert/tdelta,fs_tensor)
315 CALL add_to_tensor(ff_tensor,dumb)
316 CALL add_to_tensor(fs_tensor,dumb)
317 CALL scal_mul_coo(eps_pert/tdelta,sf_tensor)
318 CALL add_to_tensor(sf_tensor,dumb)
319
320 allocstat=0
321 DEALLOCATE(aotensor, stat=allocstat)
322 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
323
324 ALLOCATE(aotensor(ndim), stat=allocstat)
325 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
326
327 CALL copy_tensor(dumb,aotensor)
328
329 allocstat=0
330 DEALLOCATE(dumb, stat=allocstat)
331 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
332
333 ! MTV decomposition
334 ! Unresolved tensors
335
336 ! Hy tensor
337 ALLOCATE(dumb(ndim), stat=allocstat)
338 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
339
340 CALL copy_tensor(aotensor,dumb) !Copy the tensors
341 CALL init_sub_tensor(dumb,0,0)
342 CALL suppress_or(dumb,0,1,1) ! Clear entries with unresolved variables
343 CALL suppress_or(dumb,1,0,0) ! Suppress linear and nonlinear resolved terms
344 CALL copy_tensor(dumb,hy)
345
346 allocstat=0
347 DEALLOCATE(dumb, stat=allocstat)
348 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
349
350 ! Lyx tensor
351 ALLOCATE(dumb(ndim), stat=allocstat)
352 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
353
354 CALL copy_tensor(aotensor,dumb) !Copy the tensors
355 CALL init_sub_tensor(dumb,0,0)
356 CALL suppress_or(dumb,0,1,1) ! Clear entries with unresolved variables
357 CALL suppress_and(dumb,1,1,1) ! Clear constant entries
358 CALL suppress_and(dumb,1,0,0) ! Clear entries with nonlinear resolved terms
359 CALL reorder(dumb,1,0) ! Resolved variables must be the third (k) index
360 CALL copy_tensor(dumb,lyx)
361
362 allocstat=0
363 DEALLOCATE(dumb, stat=allocstat)
364 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
365
366 ! Lyy tensor
367 ALLOCATE(dumb(ndim), stat=allocstat)
368 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
369
370 CALL copy_tensor(aotensor,dumb) !Copy the tensors
371 CALL init_sub_tensor(dumb,0,0)
372 CALL suppress_or(dumb,1,0,0) ! Clear entries with resolved variables
373 CALL suppress_and(dumb,0,1,1) ! Clear entries with nonlinear unresolved terms
374 CALL suppress_and(dumb,0,0,0) ! Clear constant entries
375 CALL reorder(dumb,0,1) ! Unresolved variables must be the third (k) index
376 CALL copy_tensor(dumb,lyy)
377
378 allocstat=0
379 DEALLOCATE(dumb, stat=allocstat)
380 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
381
382 ! Byxy tensor
383 ALLOCATE(dumb(ndim), stat=allocstat)
384 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
385
386 CALL copy_tensor(aotensor,dumb) !Copy the tensors
387 CALL init_sub_tensor(dumb,0,0)
388 CALL suppress_and(dumb,1,1,1) ! Clear constant or linear terms and nonlinear unresolved only entries
389 CALL suppress_and(dumb,0,0,0) ! Clear entries with only resolved variables and constant
390 CALL reorder(dumb,0,1) ! Unresolved variables must be the third (k) index
391 CALL copy_tensor(dumb,byxy)
392
393 allocstat=0
394 DEALLOCATE(dumb, stat=allocstat)
395 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
396
397 ! Byyy tensor
398 ALLOCATE(dumb(ndim), stat=allocstat)
399 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
400

```

```

401 CALL copy_tensor(aotensor,dumb) !Copy the tensors
402 CALL init_sub_tensor(dumb,0,0)
403 CALL suppress_or(dumb,0,0,0) ! Clear entries with resolved variables and linear and constant terms
404 CALL copy_tensor(dumb,byyy)
405
406 allocstat=0
407 DEALLOCATE(dumb, stat=allocstat)
408 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
409
410 ! Byxx tensor
411 ALLOCATE(dumb(ndim), stat=allocstat)
412 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
413
414 CALL copy_tensor(aotensor,dumb) !Copy the tensors
415 CALL init_sub_tensor(dumb,0,0)
416 CALL suppress_or(dumb,1,1,1) ! Clear entries with unresolved variables and linear and constant terms
417 CALL copy_tensor(dumb,byxx)
418
419 allocstat=0
420 DEALLOCATE(dumb, stat=allocstat)
421 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
422
423 ! Resolved tensors
424
425 ! Hx tensor
426 ALLOCATE(dumb(ndim), stat=allocstat)
427 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
428
429 CALL copy_tensor(aotensor,dumb) !Copy the tensors
430 CALL init_sub_tensor(dumb,1,1)
431 CALL suppress_or(dumb,1,0,0) ! Clear entries with resolved variables
432 CALL suppress_or(dumb,0,1,1) ! Suppress linear and nonlinear unresolved terms
433 CALL copy_tensor(dumb,hx)
434
435 allocstat=0
436 DEALLOCATE(dumb, stat=allocstat)
437 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
438
439 ! Lxy tensor
440 ALLOCATE(dumb(ndim), stat=allocstat)
441 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
442
443 CALL copy_tensor(aotensor,dumb) !Copy the tensors
444 CALL init_sub_tensor(dumb,1,1)
445 CALL suppress_or(dumb,1,0,0) ! Clear entries with resolved variables
446 CALL suppress_and(dumb,0,0,0) ! Clear constant entries
447 CALL suppress_and(dumb,0,1,1) ! Clear entries with nonlinear unresolved terms
448 CALL reorder(dumb,0,1) ! Resolved variables must be the third (k) index
449 CALL copy_tensor(dumb,lxy)
450
451 allocstat=0
452 DEALLOCATE(dumb, stat=allocstat)
453 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
454
455 ! Lxx tensor
456 ALLOCATE(dumb(ndim), stat=allocstat)
457 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
458
459 CALL copy_tensor(aotensor,dumb) !Copy the tensors
460 CALL init_sub_tensor(dumb,1,1)
461 CALL suppress_or(dumb,0,1,1) ! Clear entries with unresolved variables
462 CALL suppress_and(dumb,1,0,0) ! Clear entries with nonlinear resolved terms
463 CALL suppress_and(dumb,1,1,1) ! Clear constant entries
464 CALL reorder(dumb,1,0) ! Resolved variables must be the third (k) index
465 CALL copy_tensor(dumb,lxx)
466
467 allocstat=0
468 DEALLOCATE(dumb, stat=allocstat)
469 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
470
471 ! Bxy tensor
472 ALLOCATE(dumb(ndim), stat=allocstat)
473 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
474
475 CALL copy_tensor(aotensor,dumb) !Copy the tensors
476 CALL init_sub_tensor(dumb,1,1)
477 CALL suppress_and(dumb,1,1,1) ! Clear constant or linear terms and nonlinear unresolved only entries
478 CALL suppress_and(dumb,0,0,0) ! Clear entries with only resolved variables and constant
479 CALL reorder(dumb,0,1) ! Unresolved variables must be the third (k) index
480 CALL copy_tensor(dumb,bxy)
481
482 allocstat=0
483 DEALLOCATE(dumb, stat=allocstat)
484 IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
485
486 ! Bxxx tensor
487 ALLOCATE(dumb(ndim), stat=allocstat)

```



```

488     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
489
490     CALL copy_tensor(aotensor,dumb) !Copy the tensors
491     CALL init_sub_tensor(dumb,1,1)
492     CALL suppress_or(dumb,1,1,1) ! Clear entries with unresolved variables and linear and constant terms
493     CALL copy_tensor(dumb,bxxx)
494
495     allocstat=0
496     DEALLOCATE(dumb, stat=allocstat)
497     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
498
499     ! Bxyy tensor
500     ALLOCATE(dumb(ndim), stat=allocstat)
501     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
502
503     CALL copy_tensor(aotensor,dumb) !Copy the tensors
504     CALL init_sub_tensor(dumb,1,1)
505     CALL suppress_or(dumb,0,0,0) ! Clear entries with resolved variables and linear and constant terms
506     CALL copy_tensor(dumb,bxyy)
507
508     allocstat=0
509     DEALLOCATE(dumb, stat=allocstat)
510     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
511
512
513     ! ! Dropping the unneeded part of the tensors to define them
514     ! ! Resolved tensors :
515     ! ! ss tensor
516     ! CALL suppress_or(ss_tensor,0,1,1) ! Clear entries with unresolved variables
517     ! ! sf tensor
518     ! CALL suppress_and(sf_tensor,0,0,0) ! Clear entries with only resolved variables and constant
519     ! ! Hx tensor
520     ! CALL suppress_or(Hx,0,1,1) ! Clear entries with unresolved variables
521     ! CALL suppress_or(Hx,1,0,0) ! Suppress linear and nonlinear resolved terms
522     ! ! Lxx tensor
523     ! CALL suppress_or(Lxx,0,1,1) ! Clear entries with unresolved variables
524     ! CALL suppress_and(Lxx,1,0,0) ! Clear entries with nonlinear resolved terms
525     ! CALL suppress_and(Lxx,1,1,1) ! Clear constant entries
526     ! CALL reorder(Lxx,1,0) ! Resolved variables must be the third (k) index
527     ! ! Lxy tensor
528     ! CALL suppress_or(Lxy,1,0,0) ! Clear entries with resolved variables
529     ! CALL suppress_and(Lxy,0,0,0) ! Clear constant entries
530     ! CALL suppress_and(Lxy,0,1,1) ! Clear entries with nonlinear unresolved terms
531     ! CALL reorder(Lxy,0,1) ! Unresolved variables must be the third (k) index
532     ! ! Bxxy tensor
533     ! CALL suppress_and(Bxxy,1,1,1) ! Clear constant or linear terms and nonlinear unresolved only entries
534     ! CALL suppress_and(Bxxy,0,0,0) ! Clear entries with only resolved variables and constant
535     ! CALL reorder(Bxxy,0,1) ! Unresolved variables must be the third (k) index
536     ! ! Bxxx tensor
537     ! CALL suppress_or(Bxxx,1,1,1) ! Clear entries with unresolved variables and linear and constant terms
538     ! ! Bxyy tensor
539     ! CALL suppress_or(Bxyy,0,0,0) ! Clear entries with resolved variables and linear and constant terms
540
541     ! ! Unresolved tensors :
542
543     ! ! Hy tensor
544     ! ! Lyy tensor
545     ! ! Lyx tensor
546     ! ! Byxy tensor
547     ! ! Byyy tensor
548     ! CALL suppress_or(Byyy,0,0,0) ! Clear entries with resolved variables and linear and constant terms
549     ! ! Byxx
550     ! CALL suppress_or(Byxx,1,1,1) ! Clear entries with unresolved variables and linear and constant terms
551
552

```

7.4.2.2 subroutine dec_tensor::init_sub_tensor (type(coolist), dimension(ndim), intent(inout) t, integer, intent(in) cst, integer, intent(in) v)

Subroutine that suppress all the components of a tensor t_{ijk} where if SF(i)=v.

Parameters

<i>t</i>	tensor over which the routine acts
<i>cst</i>	constant which controls if the 0 index is taken as a unresolved or a resolved one
<i>v</i>	constant of the conditional (0 to suppress resolved, 1 for unresolved)

Definition at line 174 of file dec_tensor.f90.

```

174     USE params, only: ndim
175     USE sf_def, only: sf
176     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
177     INTEGER, INTENT(IN) :: cst, v
178     INTEGER :: i
179
180     sf(0)=cst ! control wether 0 index is considered unresolved or not
181     DO i=1, ndim
182         IF (sf(i)==v) t(i)%elems=0
183     ENDDO
184

```

7.4.2.3 subroutine dec_tensor::reorder (type(coolist), dimension(ndim), intent(inout) t, integer, intent(in) cst, integer, intent(in) v)

Subroutine to reorder the tensor t_{ijk} components : if SF(j)=v then it return t_{ikj} .

Parameters

<i>t</i>	tensor over which the routine acts
<i>cst</i>	constant which controls if the 0 index is taken as a unresolved or a resolved one
<i>v</i>	constant of the conditional (0 to invert resolved, 1 for unresolved)

Definition at line 148 of file dec_tensor.f90.

```

148     USE params, only: ndim
149     USE sf_def, only: sf
150     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
151     INTEGER, INTENT(IN) :: cst, v
152     INTEGER :: i, n, li, liii
153
154     sf(0)=cst ! control wether 0 index is considered unresolved or not
155     DO i=1, ndim
156
157         n=t(i)%elems
158         DO li=1, n
159             IF (sf(t(i)%elems(li)%j)==v) THEN
160                 liii=t(i)%elems(li)%j
161                 t(i)%elems(li)%j=t(i)%elems(li)%k
162                 t(i)%elems(li)%k=liii
163             ENDIF
164         ENDDO
165     ENDDO
166

```

7.4.2.4 subroutine dec_tensor::suppress_and (type(coolist), dimension(ndim), intent(inout) t, integer, intent(in) cst, integer, intent(in) v1, integer, intent(in) v2) [private]

Subroutine to suppress from the tensor t_{ijk} components satisfying SF(j)=v1 and SF(k)=v2.

Parameters

<i>t</i>	tensor over which the routine acts
<i>cst</i>	constant which controls if the 0 index is taken as a unresolved or a resolved one
<i>v1</i>	first constant of the conditional (0 to suppress resolved, 1 for unresolved)
<i>v2</i>	second constant of the conditional (0 to suppress resolved, 1 for unresolved)

Definition at line 77 of file dec_tensor.f90.

```

77     USE params, only: ndim
78     USE sf_def, only: sf
79     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
80     INTEGER, INTENT(IN) :: cst,v1,v2
81     INTEGER :: i,n,li,liii
82
83     sf(0)=cst ! control wether 0 index is considered unresolved or not
84     DO i=1,ndim
85         n=t(i)%nelems
86         DO li=1,n
87             ! Clear entries with only resolved variables and shift rest of the items one place down.
88             ! Make sure not to skip any entries while shifting!
89
90             DO WHILE ((sf(t(i)%elems(li)%j)==v1).and.(sf(t(i)%elems(li)%k)==v2))
91                 !print*, i,li,t(i)%nelems,n
92                 DO liii=li+1,n
93                     t(i)%elems(liii-1)%j=t(i)%elems(liii)%j
94                     t(i)%elems(liii-1)%k=t(i)%elems(liii)%k
95                     t(i)%elems(liii-1)%v=t(i)%elems(liii)%v
96                 ENDDO
97                 t(i)%nelems=t(i)%nelems-1
98                 IF (li>t(i)%nelems) exit
99             ENDDO
100             IF (li>t(i)%nelems) exit
101         ENDDO
102     ENDDO
103

```

7.4.2.5 subroutine dec_tensor::suppress_or (type(coolist), dimension(ndim), intent(inout) *t*, integer, intent(in) *cst*, integer, intent(in) *v1*, integer, intent(in) *v2*)

Subroutine to suppress from the tensor t_{ijk} components satisfying $SF(j)=v1$ or $SF(k)=v2$.

Parameters

<i>t</i>	tensor over which the routine acts
<i>cst</i>	constant which controls if the 0 index is taken as a unresolved or a resolved one
<i>v1</i>	first constant of the conditional (0 to suppress resolved, 1 for unresolved)
<i>v2</i>	second constant of the conditional (0 to suppress resolved, 1 for unresolved)

Definition at line 113 of file dec_tensor.f90.

```

113     USE params, only: ndim
114     USE sf_def, only: sf
115     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
116     INTEGER, INTENT(IN) :: cst,v1,v2
117     INTEGER :: i,n,li,liii
118
119     sf(0)=cst ! control wether 0 index is considered unresolved or not
120     DO i=1,ndim
121         n=t(i)%nelems
122         DO li=1,n
123             ! Clear entries with only resolved variables and shift rest of the items one place down.
124             ! Make sure not to skip any entries while shifting!
125
126             DO WHILE ((sf(t(i)%elems(li)%j)==v1).or.(sf(t(i)%elems(li)%k)==v2))
127                 !print*, i,li,t(i)%nelems,n
128                 DO liii=li+1,n
129                     t(i)%elems(liii-1)%j=t(i)%elems(liii)%j
130                     t(i)%elems(liii-1)%k=t(i)%elems(liii)%k
131                     t(i)%elems(liii-1)%v=t(i)%elems(liii)%v
132                 ENDDO
133                 t(i)%nelems=t(i)%nelems-1
134                 IF (li>t(i)%nelems) exit
135             ENDDO
136             IF (li>t(i)%nelems) exit
137         ENDDO
138     ENDDO
139

```

7.4.3 Variable Documentation

7.4.3.1 `type(coolist), dimension(:), allocatable, public dec_tensor::bxxx`

Tensor holding the quadratic part of the resolved tendencies involving resolved variables.

Definition at line 39 of file `dec_tensor.f90`.

```
39  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: bxxx !< Tensor holding the quadratic part of
    the resolved tendencies involving resolved variables
```

7.4.3.2 `type(coolist), dimension(:), allocatable, public dec_tensor::bxyx`

Tensor holding the quadratic part of the resolved tendencies involving both resolved and unresolved variables.

Definition at line 40 of file `dec_tensor.f90`.

```
40  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: bxyx !< Tensor holding the quadratic part of
    the resolved tendencies involving both resolved and unresolved variables
```

7.4.3.3 `type(coolist), dimension(:), allocatable, public dec_tensor::bxyy`

Tensor holding the quadratic part of the resolved tendencies involving unresolved variables.

Definition at line 41 of file `dec_tensor.f90`.

```
41  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: bxyy !< Tensor holding the quadratic part of
    the resolved tendencies involving unresolved variables
```

7.4.3.4 `type(coolist), dimension(:), allocatable, public dec_tensor::byxx`

Tensor holding the quadratic part of the unresolved tendencies involving resolved variables.

Definition at line 46 of file `dec_tensor.f90`.

```
46  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: byxx !< Tensor holding the quadratic part of
    the unresolved tendencies involving resolved variables
```

7.4.3.5 `type(coolist), dimension(:), allocatable, public dec_tensor::byxy`

Tensor holding the quadratic part of the unresolved tendencies involving both resolved and unresolved variables.

Definition at line 47 of file `dec_tensor.f90`.

```
47  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: byxy !< Tensor holding the quadratic part of
    the unresolved tendencies involving both resolved and unresolved variables
```

7.4.3.6 type(coolist), dimension(:), allocatable, public dec_tensor::byyy

Tensor holding the quadratic part of the unresolved tendencies involving unresolved variables.

Definition at line 48 of file dec_tensor.f90.

```
48  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: byyy !< Tensor holding the quadratic part of
    the unresolved tendencies involving unresolved variables
```

7.4.3.7 type(coolist), dimension(:), allocatable dec_tensor::dumb [private]

Dumb coolist to make the computations.

Definition at line 53 of file dec_tensor.f90.

```
53  TYPE(coolist), DIMENSION(:), ALLOCATABLE :: dumb !< Dumb coolist to make the computations
```

7.4.3.8 type(coolist), dimension(:), allocatable, public dec_tensor::ff_tensor

Tensor holding the part of the unresolved tensor involving only unresolved variables.

Definition at line 31 of file dec_tensor.f90.

```
31  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: ff_tensor !< Tensor holding the part of the
    unresolved tensor involving only unresolved variables
```

7.4.3.9 type(coolist), dimension(:), allocatable, public dec_tensor::fs_tensor

Tensor holding the part of the unresolved tensor involving resolved variables.

Definition at line 34 of file dec_tensor.f90.

```
34  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: fs_tensor !< Tensor holding the part of the
    unresolved tensor involving resolved variables
```

7.4.3.10 type(coolist), dimension(:), allocatable, public dec_tensor::hx

Tensor holding the constant part of the resolved tendencies.

Definition at line 36 of file dec_tensor.f90.

```
36  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: hx !< Tensor holding the constant part of the
    resolved tendencies
```

7.4.3.11 type(coolist), dimension(:), allocatable, public dec_tensor::hy

Tensor holding the constant part of the unresolved tendencies.

Definition at line 43 of file dec_tensor.f90.

```
43  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: hy !< Tensor holding the constant part of the
    unresolved tendencies
```

7.4.3.12 type(coolist), dimension(:), allocatable, public dec_tensor::lxx

Tensor holding the linear part of the resolved tendencies involving the resolved variables.

Definition at line 37 of file dec_tensor.f90.

```
37  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: lxx !< Tensor holding the linear part of the
    resolved tendencies involving the resolved variables
```

7.4.3.13 type(coolist), dimension(:), allocatable, public dec_tensor::lxy

Tensor holding the linear part of the resolved tendencies involving the unresolved variables.

Definition at line 38 of file dec_tensor.f90.

```
38  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: lxy !< Tensor holding the linear part of the
    resolved tendencies involving the unresolved variables
```

7.4.3.14 type(coolist), dimension(:), allocatable, public dec_tensor::lyx

Tensor holding the linear part of the unresolved tendencies involving the resolved variables.

Definition at line 44 of file dec_tensor.f90.

```
44  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: lyx !< Tensor holding the linear part of the
    unresolved tendencies involving the resolved variables
```

7.4.3.15 type(coolist), dimension(:), allocatable, public dec_tensor::lyy

Tensor holding the linear part of the unresolved tendencies involving the unresolved variables.

Definition at line 45 of file dec_tensor.f90.

```
45  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: lyy !< Tensor holding the linear part of the
    unresolved tendencies involving the unresolved variables
```

7.4.3.16 `type(coolist), dimension(:), allocatable, public dec_tensor::sf_tensor`

Tensor holding the part of the resolved tensor involving unresolved variables.

Definition at line 32 of file `dec_tensor.f90`.

```
32  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: sf_tensor !< Tensor holding the part of the
    resolved tensor involving unresolved variables
```

7.4.3.17 `type(coolist), dimension(:), allocatable, public dec_tensor::ss_tensor`

Tensor holding the part of the resolved tensor involving only resolved variables.

Definition at line 33 of file `dec_tensor.f90`.

```
33  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: ss_tensor !< Tensor holding the part of the
    resolved tensor involving only resolved variables
```

7.4.3.18 `type(coolist), dimension(:), allocatable, public dec_tensor::ss_tl_tensor`

Tensor of the tangent linear model tendencies of the resolved component alone.

Definition at line 50 of file `dec_tensor.f90`.

```
50  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: ss_tl_tensor !< Tensor of the tangent linear
    model tendencies of the resolved component alone
```

7.5 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.

7.5.1 Detailed Description

Module to load the initial condition.

Copyright

2016 Lesley De Cruz, Jonathan Demaeyer & Sebastian Schubert See [LICENSE.txt](#) for license information.

7.5.2 Function/Subroutine Documentation

7.5.2.1 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

```



```

90      init_type="zero"
91      size_of_random_noise=0.d0
92      WRITE(6,*) "*** IC.nml namelist written. Starting with 0 as initial condition !***"
93  END IF
94  OPEN(8, file="IC.nml", status='REPLACE')
95  WRITE(8,'(a)') " !-----!"
96  WRITE(8,'(a)') " ! Namelist file : !"
97  WRITE(8,'(a)') " ! Initial condition. !"
98  WRITE(8,'(a)') " !-----!"
99  WRITE(8,*) ""
100  WRITE(8,'(a)') "&ICLIST"
101  WRITE(8,*) " ! psi variables"
102  DO i=1,natm
103      WRITE(8,*) " IC("//trim(str(i))//") = ",ic(i)," ! typ= "&
104      &("//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
105      &%Nx, fm))//", Ny= "//trim(rstr(awavenum(i)%Ny, fm))
106  END DO
107  WRITE(8,*) " ! theta variables"
108  DO i=1,natm
109      WRITE(8,*) " IC("//trim(str(i+natm))//") = ",ic(i+natm)," ! typ= "&
110      &("//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
111      &%Nx, fm))//", Ny= "//trim(rstr(awavenum(i)%Ny, fm))
112  END DO
113
114  WRITE(8,*) " ! A variables"
115  DO i=1,noc
116      WRITE(8,*) " IC("//trim(str(i+2*natm))//") = ",ic(i+2*natm)," ! Nx&
117      &= "//trim(rstr(owavenum(i)%Nx, fm))//", Ny= "&
118      &("//trim(rstr(owavenum(i)%Ny, fm))
119  END DO
120  WRITE(8,*) " ! T variables"
121  DO i=1,noc
122      WRITE(8,*) " IC("//trim(str(i+noc+2*natm))//") = ",ic(i+2*natm+noc)," &
123      &! Nx= "//trim(rstr(owavenum(i)%Nx, fm))//", Ny= "&
124      &("//trim(rstr(owavenum(i)%Ny, fm))
125  END DO
126
127  WRITE(8,'(a)') "&END"
128  WRITE(8,*) ""
129  WRITE(8,'(a)') " !-----!"
130  WRITE(8,'(a)') " ! Initialisation type. !"
131  WRITE(8,'(a)') " !-----!"
132  WRITE(8,'(a)') " ! type = 'read': use IC above (will generate a new seed);"
133  WRITE(8,'(a)') " ! 'rand': random state (will generate a new seed);"
134  WRITE(8,'(a)') " ! 'zero': zero IC (will generate a new seed);"
135  WRITE(8,'(a)') " ! 'seed': use the seed below (generate the same IC)"
136  WRITE(8,*) ""
137  WRITE(8,'(a)') "&RAND"
138  WRITE(8,'(a)') " init_type= "//init_type//""
139  WRITE(8,'(a,d15.7)') " size_of_random_noise = ",size_of_random_noise
140  DO i=1,j
141      WRITE(8,*) " seed("//trim(str(i))//") = ",seed(i)
142  END DO
143  WRITE(8,'(a)') "&END"
144  WRITE(8,*) ""
145  CLOSE(8)
146

```

7.5.3 Variable Documentation

7.5.3.1 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.
```

7.5.3.2 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
```

7.6 inprod_analytic Module Reference

Inner products between the truncated set of basis functions for the ocean and atmosphere streamfunction fields. These are partly calculated using the analytical expressions from Cehelsky, P., & Tung, K. K. : Theories of multiple equilibria and weather regimes-A critical reexamination. Part II: Baroclinic two-layer models. Journal of the atmospheric sciences, 44(21), 3282-3303, 1987.

Data Types

- type [atm_tensors](#)
Type holding the atmospheric inner products tensors.
- type [atm_wavenum](#)
Atmospheric bloc specification type.
- type [ocean_tensors](#)
Type holding the oceanic inner products tensors.
- type [ocean_wavenum](#)
Oceanic bloc specification type.

Functions/Subroutines

- real(kind=8) function [b1](#) (Pi, Pj, Pk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [b2](#) (Pi, Pj, Pk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [delta](#) (r)
Integer Dirac delta function.
- real(kind=8) function [flambda](#) (r)
"Odd or even" function
- real(kind=8) function [s1](#) (Pj, Pk, Mj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [s2](#) (Pj, Pk, Mj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [s3](#) (Pj, Pk, Hj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [s4](#) (Pj, Pk, Hj, Hk)
Cehelsky & Tung Helper functions.
- subroutine [calculate_a](#)
Eigenvalues of the Laplacian (atmospheric)
- subroutine [calculate_b](#)
Streamfunction advection terms (atmospheric)
- subroutine [calculate_c_atm](#)
Beta term for the atmosphere.
- subroutine [calculate_d](#)
Forcing of the ocean on the atmosphere.
- subroutine [calculate_g](#)
Temperature advection terms (atmospheric)
- subroutine [calculate_s](#)
Forcing (thermal) of the ocean on the atmosphere.
- subroutine [calculate_k](#)

- Forcing of the atmosphere on the ocean.*
 - subroutine [calculate_m](#)
- Forcing of the ocean fields on the ocean.*
 - subroutine [calculate_n](#)
- Beta term for the ocean.*
 - subroutine [calculate_o](#)
- Temperature advection term (passive scalar)*
 - subroutine [calculate_c_oc](#)
- Streamfunction advection terms (oceanic)*
 - subroutine [calculate_w](#)
- Short-wave radiative forcing of the ocean.*
 - subroutine, public [init_inprod](#)
- Initialisation of the inner product.*
 - subroutine, public [deallocate_inprod](#)
- Deallocation of the inner products.*

Variables

- type([atm_wavenum](#)), dimension(:), allocatable, public [awavenum](#)
Atmospheric blocs specification.
- type([ocean_wavenum](#)), dimension(:), allocatable, public [owavenum](#)
Oceanic blocs specification.
- type([atm_tensors](#)), public [atmos](#)
Atmospheric tensors.
- type([ocean_tensors](#)), public [ocean](#)
Oceanic tensors.

7.6.1 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

7.6.2 Function/Subroutine Documentation

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

Cehelsky & Tung Helper functions.

Definition at line 91 of file inprod_analytic.f90.

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

7.6.2.2 real(kind=8) function inprod_analytic::b2 (integer P_i , integer P_j , integer P_k) [private]

Cehelsky & Tung Helper functions.

Definition at line 97 of file inprod_analytic.f90.

```

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

```

7.6.2.3 subroutine inprod_analytic::calculate_a () [private]

Eigenvalues of the Laplacian (atmospheric)

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

Definition at line 155 of file inprod_analytic.f90.

```

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

```

7.6.2.4 subroutine inprod_analytic::calculate_b () [private]

Streamfunction advection terms (atmospheric)

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

Remarks

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

Definition at line 182 of file inprod_analytic.f90.

```

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

```

7.6.2.5 subroutine inprod_analytic::calculate_c_atm () [private]

Beta term for the atmosphere.

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

Remarks

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

Definition at line 216 of file inprod_analytic.f90.

```

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

```

7.6.2.6 subroutine inprod_analytic::calculate_c_oc () [private]

Streamfunction advection terms (oceanic)

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

Remarks

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

Definition at line 568 of file inprod_analytic.f90.

```

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

```

```

580         ALLOCATE(ocean%C(noc,noc,noc), stat=allocstat)
581         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
582     END IF
583 END IF
584 ocean%C=0.d0
585 val=0.d0
586
587 DO i=1,noc
588     DO j=1,noc
589         DO k=1,noc
590             val = ocean%M(k,k) * ocean%O(i,j,k)
591             IF (val /= 0.d0) ocean%C(i,j,k) = val
592         END DO
593     END DO
594 END DO

```

7.6.2.7 subroutine inprod_analytic::calculate_d () [private]

Forcing of the ocean on the atmosphere.

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

Remarks

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

Definition at line 255 of file inprod_analytic.f90.

```

255     INTEGER :: i,j
256     INTEGER :: allocstat
257
258     IF ((.NOT. ALLOCATED(atmos%s)) .OR. (.NOT. ALLOCATED(ocean%M))) THEN
259         stop "*** atmos%s and ocean%M must be defined before calling calculate_d ! ***"
260     END IF
261
262
263     IF (natm == 0 ) THEN
264         stop "*** Problem with calculate_d : natm==0 ! ***"
265     ELSE
266         IF (.NOT. ALLOCATED(atmos%d)) THEN
267             ALLOCATE(atmos%d(natm,noc), stat=allocstat)
268             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
269         END IF
270     END IF
271     atmos%d=0.d0
272
273     DO i=1,natm
274         DO j=1,noc
275             atmos%d(i,j)=atmos%s(i,j) * ocean%M(j,j)
276         END DO
277     END DO

```

7.6.2.8 subroutine inprod_analytic::calculate_g () [private]

Temperature advection terms (atmospheric)

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

Definition at line 288 of file inprod_analytic.f90.

```

288  INTEGER :: i,j,k
289  TYPE(atm_wavenum) :: ti, tj, tk
290  REAL(KIND=8) :: val,vb1, vb2, vs1, vs2, vs3, vs4
291  INTEGER :: allocstat
292
293  IF (natm == 0 ) THEN
294    stop "*** Problem with calculate_g : natm==0 ! ***"
295  ELSE
296    IF (.NOT. ALLOCATED(atmos%g)) THEN
297      ALLOCATE(atmos%g(natm,natm,natm), stat=allocstat)
298      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
299    END IF
300  END IF
301  atmos%g=0.d0
302
303  DO i=1,natm
304    DO j=1,natm
305      DO k=1,natm
306        ti = awavenum(i)
307        tj = awavenum(j)
308        tk = awavenum(k)
309        val=0.d0
310        IF ((ti%typ == "A").AND. (tj%typ == "K").AND. (tk%typ == "L")) THEN
311          vb1 = b1(ti%P,tj%P,tk%P)
312          vb2 = b2(ti%P,tj%P,tk%P)
313          val = -2 * sqrt(2.) / pi * tj%M * delta(tj%M - tk%H) * flambda(ti%P + tj%P + tk%P)
314          IF (val /= 0.d0) val = val * (vb1**2 / (vb1**2 - 1) - vb2**2 / (vb2**2 - 1))
315        ELSEIF ((ti%typ == "K").AND. (tj%typ == "K").AND. (tk%typ == "L")) THEN
316          vs1 = s1(tj%P,tk%P,tj%M,tk%H)
317          vs2 = s2(tj%P,tk%P,tj%M,tk%H)
318          val = vs1 * (delta(ti%M - tk%H - tj%M) * delta(ti%P -&
319            & tk%P + tj%P) - delta(ti%M - tk%H - tj%M) *&
320            & delta(ti%P + tk%P - tj%P) + (delta(tk%H - tj%M&
321            & + ti%M) + delta(tk%H - tj%M - ti%M)) *&
322            & delta(tk%P + tj%P - ti%P)) + vs2 * (delta(ti%M&
323            & - tk%H - tj%M) * delta(ti%P - tk%P - tj%P) +&
324            & (delta(tk%H - tj%M - ti%M) + delta(ti%M + tk%H&
325            & - tj%M)) * (delta(ti%P - tk%P + tj%P) -&
326            & delta(tk%P - tj%P + ti%P)))
327        END IF
328        val=val*n
329        IF (val /= 0.d0) THEN
330          atmos%g(i,j,k) = val
331          atmos%g(j,k,i) = val
332          atmos%g(k,i,j) = val
333          atmos%g(i,k,j) = -val
334          atmos%g(j,i,k) = -val
335          atmos%g(k,j,i) = -val
336        ENDIF
337      END DO
338    END DO
339  END DO
340
341  DO i=1,natm
342    DO j=i,natm
343      DO k=j,natm
344        ti = awavenum(i)
345        tj = awavenum(j)
346        tk = awavenum(k)
347        val=0.d0
348
349        IF ((ti%typ == "L").AND. (tj%typ == "L").AND. (tk%typ == "L")) THEN
350          vs3 = s3(tj%P,tk%P,tj%H,tk%H)
351          vs4 = s4(tj%P,tk%P,tj%H,tk%H)
352          val = vs3 * ((delta(tk%H - tj%H - ti%H) - delta(tk%H &
353            & - tj%H + ti%H)) * delta(tk%P + tj%P - ti%P) +&
354            & delta(tk%H + tj%H - ti%H) * (delta(tk%P - tj%P&
355            & + ti%P) - delta(tk%P - tj%P - ti%P))) + vs4 *&
356            & ((delta(tk%H + tj%H - ti%H) * delta(tk%P - tj&
357            & %P - ti%P)) + (delta(tk%H - tj%H + ti%H) -&
358            & delta(tk%H - tj%H - ti%H)) * (delta(tk%P - tj&
359            & %P - ti%P) - delta(tk%P - tj%P + ti%P)))
360        ENDIF
361        val=val*n
362        IF (val /= 0.d0) THEN
363          atmos%g(i,j,k) = val
364          atmos%g(j,k,i) = val
365          atmos%g(k,i,j) = val
366          atmos%g(i,k,j) = -val
367          atmos%g(j,i,k) = -val
368          atmos%g(k,j,i) = -val
369        ENDIF
370      ENDDO
371    ENDDO
372  ENDDO
373

```

7.6.2.9 subroutine inprod_analytic::calculate_k () [private]

Forcing of the atmosphere on the ocean.

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

Remarks

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

Definition at line 434 of file inprod_analytic.f90.

```

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

```

7.6.2.10 subroutine inprod_analytic::calculate_m () [private]

Forcing of the ocean fields on the ocean.

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

Definition at line 464 of file inprod_analytic.f90.

```

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

```


7.6.2.11 subroutine inprod_analytic::calculate_n() [private]

Beta term for the ocean.

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

Definition at line 487 of file inprod_analytic.f90.

```

487     INTEGER :: i,j
488     TYPE(ocean_wavenum) :: di,dj
489     REAL(KIND=8) :: val
490     INTEGER :: allocstat
491     IF (noc == 0 ) THEN
492         stop "*** Problem with calculate_N : noc==0 ! ***"
493     ELSE
494         IF (.NOT. ALLOCATED(ocean%N)) THEN
495             ALLOCATE(ocean%N(noc,noc), stat=allocstat)
496             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
497         END IF
498     END IF
499     ocean%N=0.d0
500     val=0.d0
501
502     DO i=1,noc
503         DO j=1,noc
504             di = owavenum(i)
505             dj = owavenum(j)
506             val = delta(di%P - dj%P) * flambda(di%H + dj%H)
507             IF (val /= 0.d0) ocean%N(i,j) = val * (-2) * dj%H * di%H * n / ((dj%H**2 - di%H**2) * pi)
508         END DO
509     END DO

```

7.6.2.12 subroutine inprod_analytic::calculate_o() [private]

Temperature advection term (passive scalar)

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

Definition at line 516 of file inprod_analytic.f90.

```

516     INTEGER :: i,j,k
517     REAL(KIND=8) :: vs3,vs4,val
518     TYPE(ocean_wavenum) :: di,dj,dk
519     INTEGER :: allocstat
520     IF (noc == 0 ) THEN
521         stop "*** Problem with calculate_O : noc==0 ! ***"
522     ELSE
523         IF (.NOT. ALLOCATED(ocean%O)) THEN
524             ALLOCATE(ocean%O(noc,noc,noc), stat=allocstat)
525             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
526         END IF
527     END IF
528     ocean%O=0.d0
529     val=0.d0
530
531     DO i=1,noc
532         DO j=i,noc
533             DO k=j,noc
534                 di = owavenum(i)
535                 dj = owavenum(j)
536                 dk = owavenum(k)
537                 vs3 = s3(dj%P,dk%P,dj%H,dk%H)
538                 vs4 = s4(dj%P,dk%P,dj%H,dk%H)
539                 val = vs3*((delta(dk%H - dj%H - di%H) - delta(dk%H - dj%
540                     &%H + di%H)) * delta(dk%P + dj%P - di%P) + delta(dk%
541                     &%H + dj%H - di%H) * (delta(dk%P - dj%P + di%P) -&
542                     & delta(dk%P - dj%P - di%P))) + vs4 * ((delta(dk%H &
543                     & + dj%H - di%H) * delta(dk%P - dj%P - di%P)) +&
544                     & (delta(dk%H - dj%H + di%H) - delta(dk%H - dj%H -&
545                     & di%H)) * (delta(dk%P - dj%P - di%P) - delta(dk%P &
546                     & - dj%P + di%P)))
547                 val = val * n / 2
548             IF (val /= 0.d0) THEN

```

```

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

```

7.6.2.13 subroutine inprod_analytic::calculate_s () [private]

Forcing (thermal) of the ocean on the atmosphere.

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

Definition at line 380 of file inprod_analytic.f90.

```

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

```

7.6.2.14 subroutine inprod_analytic::calculate_w () [private]

Short-wave radiative forcing of the ocean.

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

Remarks

atmospheric s tensor must be computed before calling this function !

Definition at line 605 of file inprod_analytic.f90.

```

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

```

7.6.2.15 subroutine, public inprod_analytic::deallocate_inprod ()

Deallocation of the inner products.

Definition at line 722 of file inprod_analytic.f90.

```

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

```

```

760   IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
761
762   allocstat=0
763   IF (ALLOCATED(ocean%W)) DEALLOCATE(ocean%W, stat=allocstat)
764   IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
765
766   allocstat=0
767   IF (ALLOCATED(ocean%O)) DEALLOCATE(ocean%O, stat=allocstat)
768   IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
769
770   allocstat=0
771   IF (ALLOCATED(ocean%C)) DEALLOCATE(ocean%C, stat=allocstat)
772   IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"

```

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

Integer Dirac delta function.

Definition at line 103 of file inprod_analytic.f90.

```

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

```

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

"Odd or even" function

Definition at line 113 of file inprod_analytic.f90.

```

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

```

7.6.2.18 subroutine, public inprod_analytic::init_inprod ()

Initialisation of the inner product.

Definition at line 639 of file inprod_analytic.f90.

```

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

```

```

655         awavenum(j+2)%M=ams(i,1)
656         awavenum(j+2)%P=ams(i,2)
657         awavenum(j+3)%H=ams(i,1)
658         awavenum(j+3)%P=ams(i,2)
659
660         awavenum(j+1)%Ny=REAL(ams(i,2))
661         awavenum(j+2)%Nx=REAL(ams(i,1))
662         awavenum(j+2)%Ny=REAL(ams(i,2))
663         awavenum(j+3)%Nx=REAL(ams(i,1))
664         awavenum(j+3)%Ny=REAL(ams(i,2))
665
666         j=j+3
667     ELSE
668         awavenum(j+1)%typ='K'
669         awavenum(j+2)%typ='L'
670
671         awavenum(j+1)%M=ams(i,1)
672         awavenum(j+1)%P=ams(i,2)
673         awavenum(j+2)%H=ams(i,1)
674         awavenum(j+2)%P=ams(i,2)
675
676         awavenum(j+1)%Nx=REAL(ams(i,1))
677         awavenum(j+1)%Ny=REAL(ams(i,2))
678         awavenum(j+2)%Nx=REAL(ams(i,1))
679         awavenum(j+2)%Ny=REAL(ams(i,2))
680
681         j=j+2
682
683     ENDIF
684 ENDDO
685
686 DO i=1,noc
687     owavenum(i)%H=oms(i,1)
688     owavenum(i)%P=oms(i,2)
689
690     owavenum(i)%Nx=oms(i,1)/2.d0
691     owavenum(i)%Ny=oms(i,2)
692
693 ENDDO
694
695 ! Computation of the atmospheric inner products tensors
696
697 CALL calculate_a
698 CALL calculate_g
699 CALL calculate_s
700 CALL calculate_b
701 CALL calculate_c_atm
702
703 ! Computation of the oceanic inner products tensors
704
705 CALL calculate_m
706 CALL calculate_n
707 CALL calculate_o
708 CALL calculate_c_oc
709 CALL calculate_w
710 CALL calculate_k
711
712 ! A last atmospheric one that needs ocean%M
713
714 CALL calculate_d
715
716
717

```

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

Cehelsky & Tung Helper functions.

Definition at line 123 of file inprod_analytic.f90.

```

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

```

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

Cehelsky & Tung Helper functions.

Definition at line 129 of file `inprod_analytic.f90`.

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

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

Cehelsky & Tung Helper functions.

Definition at line 135 of file `inprod_analytic.f90`.

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

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

Cehelsky & Tung Helper functions.

Definition at line 141 of file `inprod_analytic.f90`.

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

7.6.3 Variable Documentation

7.6.3.1 `type(atm_tensors), public inprod_analytic::atmos`

Atmospheric tensors.

Definition at line 69 of file `inprod_analytic.f90`.

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

7.6.3.2 `type(atm_wavenum), dimension(:), allocatable, public inprod_analytic::awavenum`

Atmospheric blocs specification.

Definition at line 64 of file `inprod_analytic.f90`.

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

7.6.3.3 type(ocean_tensors), public inprod_analytic::ocean

Oceanic tensors.

Definition at line 71 of file inprod_analytic.f90.

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

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

Oceanic blocs specification.

Definition at line 66 of file inprod_analytic.f90.

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

7.7 int_comp Module Reference

Utility module containing the routines to perform the integration of functions.

Functions/Subroutines

- subroutine, public [integrate](#) (func, ss)
Routine to compute integrals of function from O to #maxint.
- subroutine [qromb](#) (func, a, b, ss)
Romberg integration routine.
- subroutine [qromo](#) (func, a, b, ss, choose)
Romberg integration routine on an open interval.
- subroutine [polint](#) (xa, ya, n, x, y, dy)
Polynomial interpolation routine.
- subroutine [trapzd](#) (func, a, b, s, n)
Trapezoidal rule integration routine.
- subroutine [midpnt](#) (func, a, b, s, n)
Midpoint rule integration routine.
- subroutine [midexp](#) (funk, aa, bb, s, n)
Midpoint routine for bb infinite with funk decreasing infinitely rapidly at infinity.

7.7.1 Detailed Description

Utility module containing the routines to perform the integration of functions.

Copyright

2017 Jonathan Demayer. See [LICENSE.txt](#) for license information.

Remarks

Most are taken from the Numerical Recipes

7.7.2 Function/Subroutine Documentation

7.7.2.1 subroutine, public int_comp::integrate (external func, real(kind=8) ss)

Routine to compute integrals of function from O to #maxint.

Parameters

<i>func</i>	function to integrate
<i>ss</i>	result of the integration

Definition at line 30 of file int_comp.f90.

```

30      REAL(KIND=8) :: ss,func,b
31      EXTERNAL func
32      b=maxint
33      ! CALL qromo(func,0.D0,1.D0,ss,midexp)
34      CALL qromb(func,0.d0,b,ss)

```

7.7.2.2 subroutine int_comp::midexp (external *func*, real(kind=8) *aa*, real(kind=8) *bb*, real(kind=8) *s*, integer *n*)
[private]

Midpoint routine for *bb* infinite with *func* decreasing infinitely rapidly at infinity.

Parameters

<i>func</i>	function to integrate
<i>aa</i>	lower limit of the integral
<i>bb</i>	higher limit of the integral
<i>s</i>	result of the integration
<i>n</i>	higher stage of the rule to be computed

Definition at line 200 of file int_comp.f90.

```

200      INTEGER :: n
201      REAL(KIND=8) :: aa,bb,s,func
202      EXTERNAL func
203      INTEGER :: it,j
204      REAL(KIND=8) :: ddel,del,sum,tnm,x,func,a,b
205      func(x)=func(-log(x))/x
206      b=exp(-aa)
207      a=0.
208      if (n.eq.1) then
209         s=(b-a)*func(0.5*(a+b))
210      else
211         it=3*(n-2)
212         tnm=it
213         del=(b-a)/(3.*tnm)
214         ddel=del+del
215         x=a+0.5*del
216         sum=0.
217         do j=1,it
218            sum=sum+func(x)
219            x=x+ddel
220            sum=sum+func(x)
221            x=x+del
222         end do
223         s=(s+(b-a)*sum/tnm)/3.
224      endif
225      return

```

7.7.2.3 subroutine int_comp::midpnt (external *func*, real(kind=8) *a*, real(kind=8) *b*, real(kind=8) *s*, integer *n*) [private]

Midpoint rule integration routine.

Parameters

<i>func</i>	function to integrate
<i>a</i>	lower limit of the integral
<i>b</i>	higher limit of the integral
<i>s</i>	result of the integration
<i>n</i>	higher stage of the rule to be computed

Definition at line 167 of file int_comp.f90.

```

167    INTEGER :: n
168    REAL(KIND=8) :: a,b,s,func
169    EXTERNAL func
170    INTEGER :: it,j
171    REAL(KIND=8) :: ddel,del,sum,tnm,x
172    if (n.eq.1) then
173        s=(b-a)*func(0.5*(a+b))
174    else
175        it=3**(n-2)
176        tnm=it
177        del=(b-a)/(3.*tnm)
178        ddel=del+del
179        x=a+0.5*del
180        sum=0.
181        do j=1,it
182            sum=sum+func(x)
183            x=x+ddel
184            sum=sum+func(x)
185            x=x+del
186        end do
187        s=(s+(b-a)*sum/tnm)/3.
188    endif
189    return

```

7.7.2.4 subroutine int_comp::polint (real(kind=8), dimension(n) xa, real(kind=8), dimension(n) ya, integer n, real(kind=8) x, real(kind=8) y, real(kind=8) dy) [private]

Polynomial interpolation routine.

Definition at line 91 of file int_comp.f90.

```

91    INTEGER :: n,nmax
92    REAL(KIND=8) :: dy,x,y,xa(n),ya(n)
93    parameter(nmax=10)
94    INTEGER :: i,m,ns
95    REAL(KIND=8) :: den,dif,dift,ho,hp,w,c(nmax),d(nmax)
96    ns=1
97    dif=abs(x-xa(1))
98    do i=1,n
99        dift=abs(x-xa(i))
100        if (dift.lt.dif) then
101            ns=i
102            dif=dift
103        endif
104        c(i)=ya(i)
105        d(i)=ya(i)
106    end do
107    y=ya(ns)
108    ns=ns-1
109    do m=1,n-1
110        do i=1,n-m
111            ho=xa(i)-x
112            hp=xa(i+m)-x
113            w=c(i+1)-d(i)
114            den=ho-hp
115            if(den.eq.0.)stop 'failure in polint'
116            den=w/den
117            d(i)=hp*den
118            c(i)=ho*den

```

```

119         end do
120         if (2*ns.lt.n-m) then
121             dy=c(ns+1)
122         else
123             dy=d(ns)
124             ns=ns-1
125         endif
126         y=y+dy
127     end do
128     return

```

7.7.2.5 subroutine `int_comp::qromb` (external *func*, real(kind=8) *a*, real(kind=8) *b*, real(kind=8) *ss*) [private]

Romberg integration routine.

Parameters

<i>func</i>	function to integrate
<i>a</i>	lower limit of the integral
<i>b</i>	higher limit of the integral
<i>func</i>	function to integrate
<i>ss</i>	result of the integration

Definition at line 44 of file `int_comp.f90`.

```

44     INTEGER :: jmax, jmaxp, k, km
45     REAL(KIND=8) :: a, b, func, ss, eps
46     EXTERNAL func
47     parameter(eps=1.d-6, jmax=20, jmaxp=jmax+1, k=5, km=k-1)
48     INTEGER j
49     REAL(KIND=8) :: dss, h(jmaxp), s(jmaxp)
50     h(1)=1.
51     DO j=1, jmax
52         CALL trapzd(func, a, b, s(j), j)
53         IF (j.ge.k) THEN
54             CALL polint(h(j-km), s(j-km), k, 0.d0, ss, dss)
55             IF (abs(dss).le.eps*abs(ss)) RETURN
56         ENDIF
57         s(j+1)=s(j)
58         h(j+1)=0.25*h(j)
59     ENDDO
60     stop 'too many steps in qromb'

```

7.7.2.6 subroutine `int_comp::qromo` (external *func*, real(kind=8) *a*, real(kind=8) *b*, real(kind=8) *ss*, external *choose*) [private]

Romberg integration routine on an open interval.

Parameters

<i>a</i>	lower limit of the integral
<i>b</i>	higher limit of the integral
<i>func</i>	function to integrate
<i>ss</i>	result of the integration
<i>chose</i>	routine to perform the integration

Definition at line 70 of file `int_comp.f90`.

```

70     INTEGER :: jmax, jmaxp, k, km
71     REAL(KIND=8) :: a, b, func, ss, eps
72     EXTERNAL func, choose
73     parameter(eps=1.e-6, jmax=14, jmaxp=jmax+1, k=5, km=k-1)
74     INTEGER :: j
75     REAL(KIND=8) :: dss, h(jmaxp), s(jmaxp)
76     h(1)=1.
77     DO j=1, jmax
78         CALL choose(func, a, b, s(j), j)
79         IF (j.ge.k) THEN
80             call polint(h(j-km), s(j-km), k, 0.d0, ss, dss)
81             if (abs(dss).le.eps*abs(ss)) return
82         ENDIF
83         s(j+1)=s(j)
84         h(j+1)=h(j)/9.
85     ENDDO
86     stop 'too many steps in qromo'

```

7.7.2.7 subroutine `int_comp::trapzd` (external *func*, real(kind=8) *a*, real(kind=8) *b*, real(kind=8) *s*, integer *n*) [private]

Trapezoidal rule integration routine.

Parameters

<i>func</i>	function to integrate
<i>a</i>	lower limit of the integral
<i>b</i>	higher limit of the integral
<i>s</i>	result of the integration
<i>n</i>	higher stage of the rule to be computed

Definition at line 138 of file `int_comp.f90`.

```

138     INTEGER :: n
139     REAL(KIND=8) :: a, b, s, func
140     EXTERNAL func
141     INTEGER :: it, j
142     REAL(KIND=8) :: del, sum, tnm, x
143     if (n.eq.1) then
144         s=0.5*(b-a)*(func(a)+func(b))
145     else
146         it=2*(n-2)
147         tnm=it
148         del=(b-a)/tnm
149         x=a+0.5*del
150         sum=0.
151         do j=1, it
152             sum=sum+func(x)
153             x=x+del
154         end do
155         s=0.5*(s+(b-a)*sum/tnm)
156     endif
157     return

```

7.8 int_corr Module Reference

Module to compute or load the integrals of the correlation matrices.

Functions/Subroutines

- subroutine, public [init_corrint](#)

Subroutine to initialise the integrated matrices and tensors.

- real(kind=8) function [func_ij](#) (s)

Function that returns the component oi and oj of the correlation matrix at time s.

- real(kind=8) function [func_ijkl](#) (s)

Function that returns the component oi,oj,ok and ol of the outer product of the correlation matrix with itself at time s.

- subroutine, public [comp_corrint](#)

Routine that actually compute or load the integrals.

Variables

- integer [oi](#)
- integer [oj](#)
- integer [ok](#)
- integer [ol](#)

Integers that specify the matrices and tensor component considered as a function of time.

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

Small epsilon constant to determine equality with zero.

- real(kind=8), dimension(:,:), allocatable, public [corrint](#)

Matrix holding the integral of the correlation matrix.

- type([coolist4](#)), dimension(:), allocatable, public [corr2int](#)

Tensor holding the integral of the correlation outer product with itself.

7.8.1 Detailed Description

Module to compute or load the integrals of the correlation matrices.

Copyright

2017 Jonathan Demayer. See [LICENSE.txt](#) for license information.

Remarks

7.8.2 Function/Subroutine Documentation

7.8.2.1 subroutine, public int_corr::comp_corrint ()

Routine that actually compute or load the integrals.

Definition at line 75 of file int_corr.f90.

```

75     IMPLICIT NONE
76     INTEGER :: i,j,k,l,n,allocstat
77     REAL(KIND=8) :: ss
78     LOGICAL :: ex
79
80     INQUIRE(file='corrint.def',exist=ex)
81     SELECT CASE (int_corr_mode)
82     CASE ('file')
83         IF (ex) THEN
84             OPEN(30,file='corrint.def',status='old')
85             READ(30,*) corrint
86             CLOSE(30)
87         ELSE
88             stop "*** File corrint.def not found ! ***"
89         END IF
90     CASE ('prog')
91         DO i = 1,n_unres
92             DO j= 1,n_unres
93                 oi=i
94                 oj=j
95                 ! print*, oi,oj
96                 CALL integrate(func_ij,ss)
97                 corrint(ind(i),ind(j))=ss
98             END DO
99         END DO
100
101         OPEN(30,file='corrint.def')
102         WRITE(30,*) corrint
103         CLOSE(30)
104     END SELECT
105
106     INQUIRE(file='corr2int.def',exist=ex)
107     SELECT CASE (int_corr_mode)
108     CASE ('file')
109         IF (ex) THEN
110             CALL load_tensor4_from_file("corr2int.def",corr2int)
111         ELSE
112             stop "*** File corr2int.def not found ! ***"
113         END IF
114     CASE ('prog')
115         DO i = 1,n_unres
116             n=0
117             DO j= 1,n_unres
118                 DO k= 1,n_unres
119                     DO l = 1,n_unres
120                         oi=i
121                         oj=j
122                         ok=k
123                         ol=l
124
125                         CALL integrate(func_ijkl,ss)
126                         IF (abs(ss)>real_eps) n=n+1
127                     ENDDO
128                 ENDDO
129             ENDDO
130             IF (n/=0) THEN
131                 ALLOCATE(corr2int(ind(i))%elems(n), stat=allocstat)
132                 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
133                 n=0
134             DO j= 1,n_unres
135                 DO k= 1,n_unres
136                     DO l = 1,n_unres
137                         oi=i
138                         oj=j
139                         ok=k
140                         ol=l
141
142                         CALL integrate(func_ijkl,ss)
143                         IF (abs(ss)>real_eps) THEN
144                             n=n+1
145                             corr2int(ind(i))%elems(n)%j=ind(j)
146                             corr2int(ind(i))%elems(n)%k=ind(k)
147                             corr2int(ind(i))%elems(n)%l=ind(l)
148                             corr2int(ind(i))%elems(n)%v=ss
149                         END IF
150                     ENDDO
151                 ENDDO
152             ENDDO
153             corr2int(ind(i))%elems=n
154         END IF
155     ENDDO
156
157     CALL write_tensor4_to_file("corr2int.def",corr2int)
158     CASE DEFAULT
159         stop '*** INT_CORR_MODE variable not properly defined in corrmod.nml ***'
160
161

```

```
162     END SELECT
163
```

7.8.2.2 `real(kind=8) function int_corr::func_ij (real(kind=8) s) [private]`

Function that returns the component oi and oj of the correlation matrix at time s.

Parameters

<code>s</code>	time at which the function is evaluated
----------------	---

Definition at line 55 of file int_corr.f90.

```
55     IMPLICIT NONE
56     REAL(KIND=8) :: s, func_ij
57     CALL corrcomp(s)
58     func_ij=corr_ij(oi,oj)
59     RETURN
```

7.8.2.3 `real(kind=8) function int_corr::func_ijkl (real(kind=8) s) [private]`

Function that returns the component oi,oj,ok and ol of the outer product of the correlation matrix with itself at time s.

Parameters

<code>s</code>	time at which the function is evaluated
----------------	---

Definition at line 66 of file int_corr.f90.

```
66     IMPLICIT NONE
67     REAL(KIND=8) :: s, func_ijkl
68     CALL corrcomp(s)
69     func_ijkl=corr_ij(oi,oj)+corr_ij(ok,ol)
70     RETURN
```

7.8.2.4 `subroutine, public int_corr::init_corrint ()`

Subroutine to initialise the integrated matrices and tensors.

Definition at line 38 of file int_corr.f90.

```
38     INTEGER :: allocstat
39
40     ALLOCATE(corrint(ndim,ndim), stat=allocstat)
41     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
42
43     ALLOCATE(corr2int(ndim), stat=allocstat)
44     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
45
46     CALL init_corr ! Initialize the correlation matrix function
47
48     corrint=0.d0
49
```

7.8.3 Variable Documentation

7.8.3.1 `type(coolist4), dimension(:), allocatable, public int_corr::corr2int`

Tensor holding the integral of the correlation outer product with itself.

Definition at line 30 of file int_corr.f90.

```
30  TYPE(coolist4), DIMENSION(:), ALLOCATABLE, PUBLIC :: corr2int !< Tensor holding the integral of
    the correlation outer product with itself
```

7.8.3.2 `real(kind=8), dimension(:,,:), allocatable, public int_corr::corrint`

Matrix holding the integral of the correlation matrix.

Definition at line 29 of file int_corr.f90.

```
29  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: corrint !< Matrix holding the integral of the
    correlation matrix
```

7.8.3.3 `integer int_corr::oi` [private]

Definition at line 26 of file int_corr.f90.

```
26  INTEGER :: oi,oj,ok,ol !< Integers that specify the matrices and tensor component considered as a
    function of time
```

7.8.3.4 `integer int_corr::oj` [private]

Definition at line 26 of file int_corr.f90.

7.8.3.5 `integer int_corr::ok` [private]

Definition at line 26 of file int_corr.f90.

7.8.3.6 `integer int_corr::ol` [private]

Integers that specify the matrices and tensor component considered as a function of time.

Definition at line 26 of file int_corr.f90.

7.8.3.7 `real(kind=8), parameter int_corr::real_eps = 2.2204460492503131e-16` [private]

Small epsilon constant to determine equality with zero.

Definition at line 27 of file `int_corr.f90`.

```
27  REAL(KIND=8), PARAMETER :: real_eps = 2.2204460492503131e-16 !< Small epsilon constant to determine
    equality with zero
```

7.9 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.

7.9.1 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.

7.9.2 Function/Subroutine Documentation

7.9.2.1 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 ! ***"
```

7.9.2.2 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

<i>y</i>	Initial point.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>res</i>	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
```

7.9.2.3 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

<i>t</i>	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
<i>y</i>	Point at which the tendencies have to be computed.
<i>res</i>	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)
```

7.9.3 Variable Documentation

7.9.3.1 `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
```

7.9.3.2 `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
```

7.9.3.3 `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
```

7.9.3.4 `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
```

7.9.3.5 `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)
```

7.10 mar Module Reference

Multidimensional Autoregressive module to generate the correlation for the WL parameterization.

Functions/Subroutines

- subroutine, public `init_mar`
Subroutine to initialise the MAR.
- subroutine, public `mar_step` (x)
Routine to generate one step of the MAR.
- subroutine, public `mar_step_red` (xred)
Routine to generate one step of the reduce MAR.
- subroutine `stoch_vec` (dW)

Variables

- `real(kind=8), dimension(:,,:), allocatable, public q`
Square root of the noise covariance matrix.
- `real(kind=8), dimension(:,,:), allocatable, public qred`
Reduce version of Q.
- `real(kind=8), dimension(:,,:), allocatable, public rred`
Covariance matrix of the noise.
- `real(kind=8), dimension(:,,:), allocatable, public w`
W_i matrix.
- `real(kind=8), dimension(:,,:), allocatable, public wred`
Reduce W_i matrix.
- `real(kind=8), dimension(:), allocatable buf_y`
- `real(kind=8), dimension(:), allocatable dw`
- integer, public `ms`
order of the MAR

7.10.1 Detailed Description

Multidimensional Autoregressive module to generate the correlation for the WL parameterization.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

Based on the equation $y_n = \sum_{i=1}^m y_{n-i} \cdot W_i + Q \cdot \xi_n$ for an order

7.10.2 Function/Subroutine Documentation

7.10.2.1 subroutine, public mar::init_mar ()

Subroutine to initialise the MAR.

Definition at line 45 of file MAR.f90.

```

45     INTEGER :: allocstat,nf,i,info,info2
46     INTEGER, DIMENSION(3) :: s
47
48     print*, 'Initializing the MAR integrator...'
49
50     print*, 'Loading the MAR config from files...'
51
52     OPEN(20,file='MAR_R_params.def',status='old')
53     READ(20,*) nf,ms
54     IF (nf /= n_unres) stop "*** Dimension in files MAR_R_params.def and sf.nml do not correspond ! ***"
55     ALLOCATE(qred(n_unres,n_unres),rred(n_unres,n_unres),wred(ms,n_unres,n_unres),
56     stat=allocstat)
57     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
58     ALLOCATE(q(ndim,ndim),w(ms,ndim,ndim), stat=allocstat)
59     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
60     ALLOCATE(buf_y(0:ndim), dw(ndim), stat=allocstat)
61     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
62     READ(20,*) rred
63     CLOSE(20)
64
65     OPEN(20,file='MAR_W_params.def',status='old')
66     READ(20,*) nf,ms
67     s=shape(wred)
68     IF (nf /= n_unres) stop "*** Dimension in files MAR_W_params.def and sf.nml do not correspond ! ***"
69     IF (s(1) /= ms) stop "*** MAR order in files MAR_R_params.def and MAR_W_params.def do not correspond ! ***"
70     DO i=1,ms
71         READ(20,*) wred(i,::)
72     ENDDO
73     CLOSE(20)
74
75     CALL init_sqrt
76     CALL sqrtm(rred,qred,info,info2)
77     CALL ireduce(q,qred,n_unres,ind,rind)
78
79     DO i=1,ms
80         CALL ireduce(w(i,::),wred(i,::),n_unres,ind,rind)
81     ENDDO
82
83     ! Kept for internal testing - Uncomment if not needed
84     ! DEALLOCATE(Wred,Rred,Qred, STAT=AllocStat)
85     ! IF (AllocStat /= 0) STOP "*** Deallocation problem ! ***"
86
87     print*, 'MAR of order',ms,'found!'

```

7.10.2.2 subroutine, public mar::mar_step (real(kind=8), dimension(0:ndim,ms), intent(inout) x)

Routine to generate one step of the MAR.

Parameters

x	State vector of the MAR (store the y_i)
---	--

Definition at line 93 of file MAR.f90.

```

93     REAL(KIND=8), DIMENSION(0:ndim,ms), INTENT(INOUT) :: x
94     INTEGER :: j
95

```

```

96      CALL stoch_vec(dw)
97      buf_y=0.d0
98      buf_y(1:ndim)=matmul(q,dw)
99      DO j=1,ms
100         buf_y(1:ndim)=buf_y(1:ndim)+matmul(x(1:ndim,j),w(j,,:))
101      ENDDO
102      x=eoshift(x,shift=-1,boundary=buf_y,dim=2)

```

7.10.2.3 subroutine, public mar::mar_step_red (real(kind=8), dimension(0:ndim,ms), intent(inout) xred)

Routine to generate one step of the reduce MAR.

Parameters

<i>xred</i>	State vector of the MAR (store the y_i)
-------------	--

Remarks

For debugging purpose only

Definition at line 110 of file MAR.f90.

```

110      REAL(KIND=8), DIMENSION(0:ndim,ms), INTENT(INOUT) :: xred
111      INTEGER :: j
112
113      CALL stoch_vec(dw)
114      buf_y=0.d0
115      buf_y(1:n_unres)=matmul(qred,dw(1:n_unres))
116      DO j=1,ms
117         buf_y(1:n_unres)=buf_y(1:n_unres)+matmul(xred(1:n_unres,j),wred(j,,:))
118      ENDDO
119      xred=eoshift(xred,shift=-1,boundary=buf_y,dim=2)

```

7.10.2.4 subroutine mar::stoch_vec (real(kind=8), dimension(ndim), intent(inout) dW) [private]

Definition at line 125 of file MAR.f90.

```

125      REAL(KIND=8), DIMENSION(ndim), INTENT(INOUT) :: dw
126      INTEGER :: i
127      DO i=1,ndim
128         dw(i)=gasdev()
129      ENDDO

```

7.10.3 Variable Documentation

7.10.3.1 real(kind=8), dimension(:), allocatable mar::buf_y [private]

Definition at line 34 of file MAR.f90.

```

34      REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y,dw

```

7.10.3.2 `real(kind=8), dimension(:), allocatable mar::dw` [private]

Definition at line 34 of file MAR.f90.

7.10.3.3 `integer, public mar::ms`

order of the MAR

Definition at line 36 of file MAR.f90.

```
36  INTEGER :: ms !< order of the MAR
```

7.10.3.4 `real(kind=8), dimension(:,,:), allocatable, public mar::q`

Square root of the noise covariance matrix.

Definition at line 29 of file MAR.f90.

```
29  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: q !< Square root of the noise covariance matrix
```

7.10.3.5 `real(kind=8), dimension(:,,:), allocatable, public mar::qred`

Reduce version of Q.

Definition at line 30 of file MAR.f90.

```
30  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: qred !< Reduce version of Q
```

7.10.3.6 `real(kind=8), dimension(:,,:), allocatable, public mar::rred`

Covariance matrix of the noise.

Definition at line 31 of file MAR.f90.

```
31  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: rred !< Covariance matrix of the noise
```

7.10.3.7 `real(kind=8), dimension(:,::,:), allocatable, public mar::w`

W_i matrix.

Definition at line 32 of file MAR.f90.

```
32  REAL(KIND=8), DIMENSION(:,::,:), ALLOCATABLE :: w !< W_i matrix
```

7.10.3.8 `real(kind=8), dimension(:, :, :), allocatable, public mar::wred`

Reduce W_i matrix.

Definition at line 33 of file MAR.f90.

```
33  REAL(KIND=8), DIMENSION(:, :, :), ALLOCATABLE :: wred !< Reduce W_i matrix
```

7.11 memory Module Reference

Module that compute the memory term M_3 of the WL parameterization.

Functions/Subroutines

- subroutine, public `init_memory`
Subroutine to initialise the memory.
- subroutine, public `compute_m3` (y, dt, dtn, savey, save_ev, evolve, inter, h_int)
Compute the integrand of M_3 at each time in the past and integrate to get the memory term.
- subroutine, public `test_m3` (y, dt, dtn, h_int)
Routine to test the #compute_M3 routine.

Variables

- `real(kind=8), dimension(:, :), allocatable x`
Array storing the previous state of the system.
- `real(kind=8), dimension(:, :), allocatable xs`
Array storing the resolved time evolution of the previous state of the system.
- `real(kind=8), dimension(:, :), allocatable zs`
Dummy array to replace Xs in case where the evolution is not stored.
- `real(kind=8), dimension(:), allocatable buf_m`
Dummy vector.
- `real(kind=8), dimension(:), allocatable buf_m3`
Dummy vector to store the M_3 integrand.
- integer `t_index`
Integer storing the time index (current position in the arrays)
- procedure(ss_step), pointer `step`
Procedural pointer pointing on the resolved dynamics step routine.

7.11.1 Detailed Description

Module that compute the memory term M_3 of the WL parameterization.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.11.2 Function/Subroutine Documentation

7.11.2.1 subroutine, public memory::compute_m3 (real(kind=8), dimension(0:ndim), intent(in) y, real(kind=8), intent(in) dt, real(kind=8), intent(in) dtn, logical, intent(in) savey, logical, intent(in) save_ev, logical, intent(in) evolve, real(kind=8), intent(in) inter, real(kind=8), dimension(0:ndim), intent(out) h_int)

Compute the integrand of M_3 at each time in the past and integrate to get the memory term.

Parameters

<i>y</i>	current state
<i>dt</i>	timestep
<i>dtn</i>	stochastic timestep
<i>savey</i>	set if the state is stored in X at the end
<i>save_ev</i>	set if the result of the resolved time evolution is stored in Xs at the end
<i>evolve</i>	set if the resolved time evolution is performed
<i>inter</i>	set over which time interval the resolved time evolution must be computed
<i>h_int</i>	result of the integration - give the memory term

Definition at line 86 of file memory.f90.

```

86     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
87     REAL(KIND=8), INTENT(IN) :: dt,dtn
88     LOGICAL, INTENT(IN) :: savey,save_ev,evolve
89     REAL(KIND=8), INTENT(IN) :: inter
90     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: h_int
91     REAL(KIND=8) :: t
92     INTEGER :: i,j
93
94     x(:,t_index)=y
95     IF (b23def) THEN
96         xs(:,t_index)=y
97         zs(:,t_index)=y
98         DO i=1,mems-1
99             j=modulo(t_index+i-1,mems)+1
100             zs(:,j)=xs(:,j)
101             IF (evolve) THEN
102                 IF (dt.lt.inter) THEN
103                     t=0.d0
104                     DO WHILE (t+dt<inter)
105                         CALL step(zs(:,j),y,t,dt,dtn,zs(:,j))
106                     ENDDO
107                     CALL step(zs(:,j),y,t,inter-t,sqrt(inter-t),zs(:,j))
108                 ELSE
109                     CALL step(zs(:,j),y,t,inter,sqrt(inter),zs(:,j))
110                 ENDIF
111             ENDIF
112             IF (save_ev) xs(:,j)=zs(:,j)
113         ENDDO
114     ENDIF
115
116
117     ! Computing the integral
118     h_int=0.d0
119
120     DO i=1,mems
121         j=modulo(t_index+i-2,mems)+1
122         buf_m3=0.d0
123         IF (l1def) THEN
124             CALL sparse_mul3(ltot(:,i),y,x(:,j),buf_m)
125             buf_m3=buf_m3+buf_m
126         ENDIF
127         IF (b14def) THEN
128             CALL sparse_mul3(b14(:,i),x(:,j),x(:,j),buf_m)
129             buf_m3=buf_m3+buf_m
130         ENDIF
131         IF (b23def) THEN
132             CALL sparse_mul3(b23(:,i),x(:,j),zs(:,j),buf_m)
133             buf_m3=buf_m3+buf_m
134         ENDIF

```



```

135      IF (mdef) THEN
136        CALL sparse_mul4 (mtot(:,i), x(:,j), x(:,j), zs(:,j), buf_m)
137        buf_m3=buf_m3+buf_m
138      ENDIF
139      IF ((i.eq.1).or.(i.eq.mems)) THEN
140        h_int=h_int+0.5*buf_m3
141      ELSE
142        h_int=h_int+buf_m3
143      ENDIF
144    ENDDO
145
146    h_int=muti*h_int
147    IF (savey) THEN
148      t_index=t_index-1
149      IF (t_index.eq.0) t_index=mems
150    ENDIF

```

7.11.2.2 subroutine, public memory::init_memory ()

Subroutine to initialise the memory.

Definition at line 45 of file memory.f90.

```

45    INTEGER :: allocstat
46
47    t_index=mems
48
49    ALLOCATE(x(0:ndim,mems), stat=allocstat)
50    IF (allocstat /= 0) stop "*** Not enough memory ! ***"
51
52    x=0.d0
53
54    IF (b23def) THEN
55      ALLOCATE(xs(0:ndim,mems), zs(0:ndim,mems), stat=allocstat)
56      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
57
58      xs=0.d0
59    ENDIF
60
61    ALLOCATE(buf_m3(0:ndim), buf_m(0:ndim), stat=allocstat)
62    IF (allocstat /= 0) stop "*** Not enough memory ! ***"
63
64    SELECT CASE (x_int_mode)
65    CASE ('reso')
66      step => ss_step
67    CASE ('tang')
68      step => ss_tl_step
69    CASE DEFAULT
70      stop '*** X_INT_MODE variable not properly defined in stoch_params.nml ***'
71    END SELECT
72

```

7.11.2.3 subroutine, public memory::test_m3 (real(kind=8), dimension(0:ndim), intent(in) y, real(kind=8), intent(in) dt, real(kind=8), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) h_int)

Routine to test the #compute_M3 routine.

Parameters

<i>y</i>	current state
<i>dt</i>	timestep
<i>dtn</i>	stochastic timestep
<i>h_int</i>	result of the integration - give the memory term

Definition at line 159 of file memory.f90.

```

159     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
160     REAL(KIND=8), INTENT(IN) :: dt,dtn
161     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: h_int
162     INTEGER :: i,j
163
164     CALL compute_m3(y,dt,dtn,.true.,.true.,.true.,muti,h_int)
165     print*, t_index
166     print*, 'X'
167     DO i=1,mems
168         j=modulo(t_index+i-1,mems)+1
169         print*, i,j,x(1,j)
170     ENDDO
171
172     IF (b23def) THEN
173         print*, 'Xs'
174         DO i=1,mems
175             j=modulo(t_index+i-1,mems)+1
176             print*, i,j,xs(1,j)
177         ENDDO
178     ENDIF
179     print*, 'h_int',h_int

```

7.11.3 Variable Documentation

7.11.3.1 `real(kind=8), dimension(:), allocatable memory::buf_m` [private]

Dummy vector.

Definition at line 31 of file memory.f90.

```

31     REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_m !< Dummy vector

```

7.11.3.2 `real(kind=8), dimension(:), allocatable memory::buf_m3` [private]

Dummy vector to store the M_3 integrand.

Definition at line 32 of file memory.f90.

```

32     REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_m3 !< Dummy vector to store the \f$M_3\f$ integrand

```

7.11.3.3 `procedure(ss_step), pointer memory::step` [private]

Procedural pointer pointing on the resolved dynamics step routine.

Definition at line 36 of file memory.f90.

```

36     PROCEDURE(ss_step), POINTER :: step !< Procedural pointer pointing on the resolved dynamics step routine

```

7.11.3.4 `integer memory::t_index` [private]

Integer storing the time index (current position in the arrays)

Definition at line 34 of file memory.f90.

```

34     INTEGER :: t_index !< Integer storing the time index (current position in the arrays)

```

7.11.3.5 `real(kind=8), dimension(:,,:), allocatable memory::x` [private]

Array storing the previous state of the system.

Definition at line 28 of file memory.f90.

```
28  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: x !< Array storing the previous state of the system
```

7.11.3.6 `real(kind=8), dimension(:,,:), allocatable memory::xs` [private]

Array storing the resolved time evolution of the previous state of the system.

Definition at line 29 of file memory.f90.

```
29  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: xs !< Array storing the resolved time evolution of the
    previous state of the system
```

7.11.3.7 `real(kind=8), dimension(:,,:), allocatable memory::zs` [private]

Dummy array to replace Xs in case where the evolution is not stored.

Definition at line 30 of file memory.f90.

```
30  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: zs !< Dummy array to replace Xs in case where the evolution
    is not stored
```

7.12 mtv_int_tensor Module Reference

The MTV tensors used to integrate the MTV model.

Functions/Subroutines

- subroutine, public [init_mtv_int_tensor](#)
Subroutine to initialise the MTV tensor.

Variables

- `real(kind=8), dimension(:), allocatable, public h1`
First constant vector.
- `real(kind=8), dimension(:), allocatable, public h2`
Second constant vector.
- `real(kind=8), dimension(:), allocatable, public h3`
Third constant vector.
- `real(kind=8), dimension(:), allocatable, public htot`
Total constant vector.
- `type(coolist), dimension(:), allocatable, public l1`
First linear tensor.
- `type(coolist), dimension(:), allocatable, public l2`
Second linear tensor.
- `type(coolist), dimension(:), allocatable, public l3`
Third linear tensor.
- `type(coolist), dimension(:), allocatable, public ltot`
Total linear tensor.
- `type(coolist), dimension(:), allocatable, public b1`
First quadratic tensor.
- `type(coolist), dimension(:), allocatable, public b2`
Second quadratic tensor.
- `type(coolist), dimension(:), allocatable, public btot`
Total quadratic tensor.
- `type(coolist4), dimension(:), allocatable, public mtot`
Tensor for the cubic terms.
- `real(kind=8), dimension(:,,:), allocatable, public q1`
Constant terms for the state-dependent noise covariance matrix.
- `real(kind=8), dimension(:,,:), allocatable, public q2`
Constant terms for the state-independent noise covariance matrix.
- `type(coolist), dimension(:), allocatable, public utot`
Linear terms for the state-dependent noise covariance matrix.
- `type(coolist4), dimension(:), allocatable, public vtot`
Quadratic terms for the state-dependent noise covariance matrix.
- `real(kind=8), dimension(:), allocatable dumb_vec`
Dummy vector.
- `real(kind=8), dimension(:,,:), allocatable dumb_mat1`
Dummy matrix.
- `real(kind=8), dimension(:,,:), allocatable dumb_mat2`
Dummy matrix.
- `real(kind=8), dimension(:,,:), allocatable dumb_mat3`
Dummy matrix.
- `real(kind=8), dimension(:,,:), allocatable dumb_mat4`
Dummy matrix.

7.12.1 Detailed Description

The MTV tensors used to integrate the MTV model.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

See : Franzke, C., Majda, A. J., & Vanden-Eijnden, E. (2005). Low-order stochastic mode reduction for a realistic barotropic model climate. Journal of the atmospheric sciences, 62(6), 1722-1745.

7.12.2 Function/Subroutine Documentation

7.12.2.1 subroutine, public mtv_int_tensor::init_mtv_int_tensor ()

Subroutine to initialise the MTV tensor.

Definition at line 89 of file MTV_int_tensor.f90.

```

89     INTEGER :: allocstat,i,j,k,l
90
91     print*, 'Initializing the decomposition tensors...'
92     CALL init_dec_tensor
93     print*, "Initializing the correlation matrices and tensors..."
94     CALL init_corrnt
95     print*, "Computing the correlation integrated matrices and tensors..."
96     CALL comp_corrnt
97
98     !H part
99     print*, "Computing the H term..."
100
101     ALLOCATE(h1(0:ndim), h2(0:ndim), h3(0:ndim), htot(0:ndim),
102 stat=allocstat)
103     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
104     ALLOCATE(dumb_mat1(ndim,ndim), dumb_mat2(ndim,ndim), stat=allocstat)
105     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
106     ALLOCATE(dumb_mat3(ndim,ndim), dumb_mat4(ndim,ndim), stat=allocstat)
107     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
108
109     !H1
110     CALL coo_to_mat_ik(lxy,dumb_mat1)
111     dumb_mat2=matmul(dumb_mat1,corrnt)
112     CALL sparse_mul3_with_mat(bxyy,dumb_mat2,h1)
113
114     ! H2
115     h2=0.d0
116     IF (mode.ne.'ures') THEN
117         CALL coo_to_mat_ik(lyy,dumb_mat1)
118         dumb_mat1=matmul(inv_corr_i_full,dumb_mat1)
119
120         DO i=1,ndim
121             CALL coo_to_mat_i(i,bxyy,dumb_mat2)
122             CALL sparse_mul4_with_mat_j1(corr2int,dumb_mat2,dumb_mat3)
123             CALL sparse_mul4_with_mat_j1(corr2int,transpose(dumb_mat2),dumb_mat4)
124             dumb_mat3=dumb_mat3+dumb_mat4
125             h2(i)=mat_contract(dumb_mat1,dumb_mat3)
126         ENDDO
127     ENDIF
128
129     !H3
130     h3=0.d0
131     CALL sparse_mul3_with_mat(bxyy,corr_i_full,h3)
132
133     !Htot
134     htot=0.d0
135     htot=h1+h2+h3

```

```

137
138 print*, "Computing the L terms..."
139 ALLOCATE(l1(ndim), l2(ndim), l3(ndim), stat=allocstat)
140 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
141
142 !L1
143 CALL coo_to_mat_ik(lyx,dumb_mat1)
144 CALL coo_to_mat_ik(lxy,dumb_mat2)
145 dumb_mat3=matmul(inv_corr_i_full,corrint)
146 dumb_mat4=matmul(dumb_mat2,matmul(transpose(dumb_mat3),dumb_mat1))
147 CALL matc_to_coo(dumb_mat4,11)
148
149 !L2
150 dumb_mat4=0.d0
151 DO i=1,ndim
152     DO j=1,ndim
153         CALL coo_to_mat_i(i,bxyy,dumb_mat1)
154         CALL sparse_mul4_with_mat_j1(corr2int,dumb_mat1+transpose(dumb_mat1),dumb_mat2)
155
156         CALL coo_to_mat_j(j,byxy,dumb_mat1)
157         dumb_mat1=matmul(inv_corr_i_full,dumb_mat1)
158         dumb_mat4(i,j)=mat_contract(dumb_mat1,dumb_mat2)
159     END DO
160 END DO
161 CALL matc_to_coo(dumb_mat4,12)
162
163 !L3
164 dumb_mat4=0.d0
165 DO i=1,ndim
166     DO j=1,ndim
167         CALL coo_to_mat_j(j,bxyy,dumb_mat1)
168         CALL coo_to_mat_i(i,bxyy,dumb_mat2)
169         dumb_mat4(i,j)=mat_trace(matmul(dumb_mat1,matmul(corrint,transpose(dumb_mat2))))
170     ENDDO
171 END DO
172 CALL matc_to_coo(dumb_mat4,13)
173
174 !Ltot
175
176 ALLOCATE(ltot(ndim), stat=allocstat)
177 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
178
179 CALL add_to_tensor(l1,ltot)
180 CALL add_to_tensor(l2,ltot)
181 CALL add_to_tensor(l3,ltot)
182
183 print*, "Computing the B terms..."
184 ALLOCATE(b1(ndim), b2(ndim), btot(ndim), stat=allocstat)
185 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
186 ALLOCATE(dumb_vec(ndim), stat=allocstat)
187 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
188
189 ! B1
190 CALL coo_to_mat_ik(lxy,dumb_mat1)
191 dumb_mat2=matmul(inv_corr_i_full,corrint)
192
193 dumb_mat3=matmul(dumb_mat1,transpose(dumb_mat2))
194 DO j=1,ndim
195     DO k=1,ndim
196         CALL coo_to_vec_jk(j,k,byxx,dumb_vec)
197         dumb_vec=matmul(dumb_mat3,dumb_vec)
198         CALL add_vec_jk_to_tensor(j,k,dumb_vec,b1)
199     ENDDO
200 END DO
201
202 ! B2
203 CALL coo_to_mat_ik(lyx,dumb_mat3)
204 dumb_mat2=matmul(inv_corr_i_full,corrint)
205
206 dumb_mat4=matmul(transpose(dumb_mat2),dumb_mat3)
207 DO i=1,ndim
208     CALL coo_to_mat_i(i,bxyy,dumb_mat1)
209     dumb_mat2=matmul(dumb_mat1,dumb_mat4)
210     CALL add_matc_to_tensor(i,dumb_mat2,b2)
211 ENDDO
212
213 CALL add_to_tensor(b1,btot)
214 CALL add_to_tensor(b2,btot)
215
216 !M
217
218 print*, "Computing the M term..."
219
220 ALLOCATE(mtot(ndim), stat=allocstat)
221 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
222
223 dumb_mat2=matmul(inv_corr_i_full,corrint)

```

```

224
225     DO i=1,ndim
226         CALL coo_to_mat_i(i,bxxy,dumb_mat1)
227         dumb_mat3=matmul(dumb_mat1,transpose(dumb_mat2))
228         DO k=1,ndim
229             DO l=1,ndim
230                 CALL coo_to_vec_jk(k,l,byxx,dumb_vec)
231                 dumb_vec=matmul(dumb_mat3,dumb_vec)
232                 CALL add_vec_ikl_to_tensor4(i,k,l,dumb_vec,mtot)
233             ENDDO
234         END DO
235     END DO
236
237     !Q
238
239     print*, "Computing the Q terms..."
240     ALLOCATE(q1(ndim,ndim), q2(ndim,ndim), stat=allocstat)
241     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
242
243     !Q1
244
245     CALL coo_to_mat_ik(lxy,dumb_mat1)
246     q1=matmul(dumb_mat1,matmul(corrint,transpose(dumb_mat1)))
247
248     !Q2
249
250     DO i=1,ndim
251         DO j=1,ndim
252             CALL coo_to_mat_i(i,bxxy,dumb_mat1)
253             CALL coo_to_mat_i(j,bxxy,dumb_mat2)
254             CALL sparse_mul4_with_mat_j1(corr2int,dumb_mat2,dumb_mat3)
255             CALL sparse_mul4_with_mat_j1(corr2int,transpose(dumb_mat2),dumb_mat4)
256             dumb_mat2=dumb_mat3+dumb_mat4
257             q2(i,j)=mat_contract(dumb_mat1,dumb_mat2)
258         END DO
259     END DO
260
261     !U
262
263     ALLOCATE(utot(ndim), stat=allocstat)
264     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
265
266     CALL coo_to_mat_ik(lxy,dumb_mat1)
267     DO i=1,ndim
268         CALL coo_to_mat_i(i,bxxy,dumb_mat2)
269         dumb_mat3=matmul(dumb_mat1,matmul(corrint,transpose(dumb_mat2)))
270         CALL add_matc_to_tensor(i,dumb_mat3,utot)
271     ENDDO
272
273     DO j=1,ndim
274         CALL coo_to_mat_i(j,bxxy,dumb_mat2)
275         dumb_mat3=matmul(dumb_mat1,matmul(corrint,transpose(dumb_mat2)))
276         DO k=1,ndim
277             CALL add_vec_jk_to_tensor(j,k,dumb_mat3(:,k),utot)
278         ENDDO
279     ENDDO
280
281     !V
282
283     ALLOCATE(vtot(ndim), stat=allocstat)
284     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
285
286     DO i=1,ndim
287         DO j=1,ndim
288             CALL coo_to_mat_i(i,bxxy,dumb_mat1)
289             CALL coo_to_mat_i(j,bxxy,dumb_mat2)
290             dumb_mat3=matmul(dumb_mat1,matmul(corrint,transpose(dumb_mat2)))
291             CALL add_matc_to_tensor4(j,i,dumb_mat3,vtot)
292         ENDDO
293     ENDDO
294
295     DEALLOCATE(dumb_mat1, dumb_mat2, stat=allocstat)
296     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
297
298     DEALLOCATE(dumb_mat3, dumb_mat4, stat=allocstat)
299     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
300
301     DEALLOCATE(dumb_vec, stat=allocstat)
302     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
303
304

```

7.12.3 Variable Documentation

7.12.3.1 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::b1`

First quadratic tensor.

Definition at line 54 of file MTV_int_tensor.f90.

```
54  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: b1 !< First quadratic tensor
```

7.12.3.2 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::b2`

Second quadratic tensor.

Definition at line 55 of file MTV_int_tensor.f90.

```
55  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: b2 !< Second quadratic tensor
```

7.12.3.3 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::btot`

Total quadratic tensor.

Definition at line 56 of file MTV_int_tensor.f90.

```
56  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: btot !< Total quadratic tensor
```

7.12.3.4 `real(kind=8), dimension(:,:), allocatable mtv_int_tensor::dumb_mat1 [private]`

Dummy matrix.

Definition at line 67 of file MTV_int_tensor.f90.

```
67  REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: dumb_mat1 !< Dummy matrix
```

7.12.3.5 `real(kind=8), dimension(:,:), allocatable mtv_int_tensor::dumb_mat2 [private]`

Dummy matrix.

Definition at line 68 of file MTV_int_tensor.f90.

```
68  REAL(KIND=8), DIMENSION(:,:), ALLOCATABLE :: dumb_mat2 !< Dummy matrix
```


7.12.3.6 `real(kind=8), dimension(:, :), allocatable mtv_int_tensor::dumb_mat3` [private]

Dummy matrix.

Definition at line 69 of file MTV_int_tensor.f90.

```
69  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat3 !< Dummy matrix
```

7.12.3.7 `real(kind=8), dimension(:, :), allocatable mtv_int_tensor::dumb_mat4` [private]

Dummy matrix.

Definition at line 70 of file MTV_int_tensor.f90.

```
70  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat4 !< Dummy matrix
```

7.12.3.8 `real(kind=8), dimension(:), allocatable mtv_int_tensor::dumb_vec` [private]

Dummy vector.

Definition at line 66 of file MTV_int_tensor.f90.

```
66  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dumb_vec !< Dummy vector
```

7.12.3.9 `real(kind=8), dimension(:), allocatable, public mtv_int_tensor::h1`

First constant vector.

Definition at line 42 of file MTV_int_tensor.f90.

```
42  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: h1 !< First constant vector
```

7.12.3.10 `real(kind=8), dimension(:), allocatable, public mtv_int_tensor::h2`

Second constant vector.

Definition at line 43 of file MTV_int_tensor.f90.

```
43  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: h2 !< Second constant vector
```

7.12.3.11 `real(kind=8), dimension(:), allocatable, public mtv_int_tensor::h3`

Third constant vector.

Definition at line 44 of file MTV_int_tensor.f90.

```
44  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: h3    !< Third constant vector
```

7.12.3.12 `real(kind=8), dimension(:), allocatable, public mtv_int_tensor::htot`

Total constant vector.

Definition at line 45 of file MTV_int_tensor.f90.

```
45  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: htot !< Total constant vector
```

7.12.3.13 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::l1`

First linear tensor.

Definition at line 48 of file MTV_int_tensor.f90.

```
48  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: l1    !< First linear tensor
```

7.12.3.14 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::l2`

Second linear tensor.

Definition at line 49 of file MTV_int_tensor.f90.

```
49  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: l2    !< Second linear tensor
```

7.12.3.15 `type(coolist), dimension(:), allocatable, public mtv_int_tensor::l3`

Third linear tensor.

Definition at line 50 of file MTV_int_tensor.f90.

```
50  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: l3    !< Third linear tensor
```

7.12.3.16 type(coolist), dimension(:), allocatable, public mtv_int_tensor::ltot

Total linear tensor.

Definition at line 51 of file MTV_int_tensor.f90.

```
51  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: ltot !< Total linear tensor
```

7.12.3.17 type(coolist4), dimension(:), allocatable, public mtv_int_tensor::mtot

Tensor for the cubic terms.

Definition at line 58 of file MTV_int_tensor.f90.

```
58  TYPE(coolist4), DIMENSION(:), ALLOCATABLE, PUBLIC :: mtot !< Tensor for the cubic terms
```

7.12.3.18 real(kind=8), dimension(:, :), allocatable, public mtv_int_tensor::q1

Constant terms for the state-dependent noise covariance matrix.

Definition at line 61 of file MTV_int_tensor.f90.

```
61  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: q1 !< Constant terms for the state-dependent noise
    covariance matrix
```

7.12.3.19 real(kind=8), dimension(:, :), allocatable, public mtv_int_tensor::q2

Constant terms for the state-independent noise covariance matrix.

Definition at line 62 of file MTV_int_tensor.f90.

```
62  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: q2 !< Constant terms for the state-independent noise
    covariance matrix
```

7.12.3.20 type(coolist), dimension(:), allocatable, public mtv_int_tensor::utot

Linear terms for the state-dependent noise covariance matrix.

Definition at line 63 of file MTV_int_tensor.f90.

```
63  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: utot !< Linear terms for the state-dependent
    noise covariance matrix
```

7.12.3.21 `type(coolist4), dimension(:), allocatable, public mtv_int_tensor::vtot`

Quadratic terms for the state-dependent noise covariance matrix.

Definition at line 64 of file `MTV_int_tensor.f90`.

```
64  TYPE(coolist4), DIMENSION(:), ALLOCATABLE, PUBLIC :: vtot !< Quadratic terms for the
    state-dependent noise covariance matrix
```

7.13 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`
Mechanical 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) [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.*

7.13.1 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

7.13.2 Function/Subroutine Documentation

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

Read the basic parameters and mode selection from the namelist.

Definition at line 91 of file params.f90.

```

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

7.13.2.2 subroutine params::init_params ()

Parameters initialisation routine.

Definition at line 133 of file params.f90.

```

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

```

```

153     noc=s(1)
154
155     ndim=2*natm+2*noc
156
157     !-----!
158     !
159     ! Some general parameters (Domain, beta, gamma, coupling) !
160     !
161     !-----!
162
163     pi=dacos(-1.d0)
164     l=scale/pi
165     phi0=phi0_npi*pi
166     lr=sqrt(gp*h)/f0
167     g=-1**2/lr**2
168     betp=1/rra*cos(phi0)/sin(phi0)
169     rp=r/f0
170     dp=d/f0
171     kd=k*2
172     kdp=kp
173
174     !-----!
175     !
176     ! DERIVED QUANTITIES
177     !
178     !-----!
179
180     cpo=co/(go*f0) * rr/(f0**2*1**2)
181     lpo=lambda/(go*f0)
182     cpa=ca/(ga*f0) * rr/(f0**2*1**2)/2 ! Cpa acts on psi1-psi3, not on theta
183     lpa=lambda/(ga*f0)
184     sbpo=4*sb*to0**3/(go*f0) ! long wave radiation lost by ocean to atmosphere space
185     sbpa=8*epsa*sb*ta0**3/(go*f0) ! long wave radiation from atmosphere absorbed by ocean
186     lsbpo=2*epsa*sb*to0**3/(ga*f0) ! long wave radiation from ocean absorbed by atmosphere
187     lsbpa=8*epsa*sb*ta0**3/(ga*f0) ! long wave radiation lost by atmosphere to space & ocea
188
189

```

7.13.3 Variable Documentation

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

Atmospheric mode selection array.

Definition at line 81 of file params.f90.

```

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

```

7.13.3.2 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

```

7.13.3.3 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.

```


7.13.3.4 `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.
```

7.13.3.5 `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.
```

7.13.3.6 `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.
```

7.13.3.7 `real(kind=8) params::d`

Mechanical coupling parameter between the ocean and the atmosphere.

Definition at line 31 of file params.f90.

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

7.13.3.8 `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.
```

7.13.3.9 real(kind=8) params::dt

Integration time step.

Definition at line 71 of file params.f90.

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

7.13.3.10 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.
```

7.13.3.11 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
```

7.13.3.12 real(kind=8) params::g

γ

Definition at line 50 of file params.f90.

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

7.13.3.13 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.
```

7.13.3.14 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.
```

7.13.3.15 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
```

7.13.3.16 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.
```

7.13.3.17 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.
```

7.13.3.18 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.
```

7.13.3.19 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.
```

7.13.3.20 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.
```

7.13.3.21 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
```

7.13.3.22 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.
```

7.13.3.23 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.
```

7.13.3.24 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.
```

7.13.3.25 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
```

7.13.3.26 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.
```

7.13.3.27 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.
```

7.13.3.28 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
```

7.13.3.29 integer params::natm =0

Number of atmospheric basis functions.

Definition at line 77 of file params.f90.

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

7.13.3.30 integer params::nbatm

Number of oceanic blocks.

Definition at line 76 of file params.f90.

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

7.13.3.31 integer params::nboc

Number of atmospheric blocks.

Definition at line 75 of file params.f90.

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

7.13.3.32 integer params::ndim

Number of variables (dimension of the model)

Definition at line 79 of file params.f90.

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

7.13.3.33 integer params::noc =0

Number of oceanic basis functions.

Definition at line 78 of file params.f90.

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

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

Ocean mode selection array.

Definition at line 80 of file params.f90.

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

7.13.3.35 real(kind=8) params::phi0

Latitude in radian.

Definition at line 25 of file params.f90.

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

7.13.3.36 real(kind=8) params::phi0_npi

Latitude exprimed in fraction of pi.

Definition at line 35 of file params.f90.

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

7.13.3.37 real(kind=8) params::pi

π

Definition at line 48 of file params.f90.

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

7.13.3.38 real(kind=8) params::r

Frictional coefficient at the bottom of the ocean.

Definition at line 30 of file params.f90.

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

7.13.3.39 real(kind=8) params::rp

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

Definition at line 51 of file params.f90.

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

7.13.3.40 real(kind=8) params::rr

R - Gas constant of dry air

Definition at line 45 of file params.f90.

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

7.13.3.41 real(kind=8) params::rra

Earth radius.

Definition at line 26 of file params.f90.

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

7.13.3.42 real(kind=8) params::sb

Stefan–Boltzmann constant.

Definition at line 66 of file params.f90.

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

7.13.3.43 real(kind=8) params::sbpa

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

Definition at line 61 of file params.f90.

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


7.13.3.44 real(kind=8) params::sbpo

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

Definition at line 60 of file params.f90.

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

7.13.3.45 real(kind=8) params::sc

Ratio of surface to atmosphere temperature.

Definition at line 65 of file params.f90.

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

7.13.3.46 real(kind=8) params::scale

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

Definition at line 47 of file params.f90.

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

7.13.3.47 real(kind=8) params::sig0

σ_0 - Non-dimensional static stability of the atmosphere.

Definition at line 27 of file params.f90.

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

7.13.3.48 real(kind=8) params::t_run

Effective intergration time (length of the generated trajectory)

Definition at line 70 of file params.f90.

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

7.13.3.49 real(kind=8) params::t_trans

Transient time period.

Definition at line 69 of file params.f90.

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

7.13.3.50 real(kind=8) params::ta0

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

Definition at line 42 of file params.f90.

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

7.13.3.51 real(kind=8) params::to0

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

Definition at line 41 of file params.f90.

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

7.13.3.52 real(kind=8) params::tw

Write all variables every tw time units.

Definition at line 72 of file params.f90.

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

7.13.3.53 logical params::writeout

Write to file boolean.

Definition at line 73 of file params.f90.

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

7.14 rk2_mtv_integrator Module Reference

Module with the MTV rk2 integration routines.

Functions/Subroutines

- subroutine, public [init_integrator](#)
Subroutine to initialize the MTV rk2 integrator.
- subroutine [init_noise](#)
Routine to initialize the noise vectors and buffers.
- subroutine [init_g](#)
Routine to initialize the G term.
- subroutine [compg](#) (y)
Routine to actualize the G term based on the state y of the MTV system.
- subroutine, public [step](#) (y, t, dt, dtn, res, tend)
Routine to perform an integration step (Heun algorithm) of the MTV system. The incremented time is returned.
- subroutine, public [full_step](#) (y, t, dt, dtn, res)
Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [buf_y1](#)
- real(kind=8), dimension(:), allocatable [buf_f0](#)
- real(kind=8), dimension(:), allocatable [buf_f1](#)
Integration buffers.
- real(kind=8), dimension(:), allocatable [dw](#)
- real(kind=8), dimension(:), allocatable [dwmult](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [dwar](#)
- real(kind=8), dimension(:), allocatable [dwau](#)
- real(kind=8), dimension(:), allocatable [dwor](#)
- real(kind=8), dimension(:), allocatable [dwou](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [anoise](#)
- real(kind=8), dimension(:), allocatable [noise](#)
Additive noise term.
- real(kind=8), dimension(:), allocatable [noisemult](#)
Multiplicative noise term.
- real(kind=8), dimension(:), allocatable [g](#)
G term of the MTV tendencies.
- real(kind=8), dimension(:), allocatable [buf_g](#)
Buffer for the G term computation.
- logical [mult](#)
Logical indicating if the sigma1 matrix must be computed for every state change.
- logical [q1fill](#)
Logical indicating if the matrix Q1 is non-zero.
- logical [compute_mult](#)
Logical indicating if the Gaussian noise for the multiplicative noise must be computed.
- real(kind=8), parameter [sq2](#) = sqrt(2.D0)
Hard coded square root of 2.

7.14.1 Detailed Description

Module with the MTV rk2 integration routines.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

This module actually contains the Heun algorithm routines.

7.14.2 Function/Subroutine Documentation

7.14.2.1 subroutine rk2_mtv_integrator::compg (real(kind=8), dimension(0:ndim), intent(in) y) [private]

Routine to actualize the G term based on the state y of the MTV system.

Parameters

y	State of the MTV system
---	-------------------------

Definition at line 105 of file rk2_MTV_integrator.f90.

```

105     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
106
107     g=htot
108     CALL sparse_mul2_k(ltot,y,buf_g)
109     g=g+buf_g
110     CALL sparse_mul3(btot,y,y,buf_g)
111     g=g+buf_g
112     CALL sparse_mul4(mtot,y,y,y,buf_g)
113     g=g+buf_g

```

7.14.2.2 subroutine, public rk2_mtv_integrator::full_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), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res)

Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Parameters

y	Initial point.
t	Actual integration time
dt	Integration timestep.
dtn	Stochastoc integration timestep (normally square-root of dt).
res	Final point after the step.

Definition at line 170 of file rk2_MTV_integrator.f90.

```

170     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
171     REAL(KIND=8), INTENT(INOUT) :: t
172     REAL(KIND=8), INTENT(IN) :: dt,dtn
173     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
174     CALL stoch_atm_res_vec(dwar)
175     CALL stoch_atm_unres_vec(dwau)
176     CALL stoch_oc_res_vec(dwor)
177     CALL stoch_oc_unres_vec(dwou)
178     anoise=(q_ar*dwar+q_au*dwau+q_or*dwor+q_ou*dwou)*dtn
179     CALL sparse_mul3(aotensor,y,y,buf_f0)
180     buf_y1 = y+dt*buf_f0+anoise
181     CALL sparse_mul3(aotensor,buf_y1,buf_y1,buf_f1)
182     res=y+0.5*(buf_f0+buf_f1)*dt+anoise
183     t=t+dt

```

7.14.2.3 subroutine rk2_mtv_integrator::init_g () [private]

Routine to initialize the G term.

Definition at line 97 of file rk2_MTV_integrator.f90.

```

97     INTEGER :: allocstat
98     ALLOCATE(g(0:ndim), buf_g(0:ndim), stat=allocstat)
99     IF (allocstat /= 0) stop "*** Not enough memory ! ***"

```

7.14.2.4 subroutine, public rk2_mtv_integrator::init_integrator ()

Subroutine to initialize the MTV rk2 integrator.

Definition at line 50 of file rk2_MTV_integrator.f90.

```

50     INTEGER :: allocstat
51
52     CALL init_ss_integrator ! Initialize the uncoupled resolved dynamics
53
54     ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim),stat=allocstat)
55     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
56
57     buf_y1=0.d0
58     buf_f1=0.d0
59     buf_f0=0.d0
60
61     print*, 'Initializing the integrator ...'
62     CALL init_sigma(mult,qlfill)
63     CALL init_noise
64     CALL init_g

```

7.14.2.5 subroutine rk2_mtv_integrator::init_noise () [private]

Routine to initialize the noise vectors and buffers.

Definition at line 69 of file rk2_MTV_integrator.f90.

```

69     INTEGER :: allocstat
70     ALLOCATE(dw(0:ndim), dwmult(0:ndim), stat=allocstat)
71     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
72
73     ALLOCATE(dwar(0:ndim), dwau(0:ndim), dwor(0:ndim), dwou(0:ndim),
74     stat=allocstat)
75     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
76
77     ALLOCATE(anoise(0:ndim), noise(0:ndim), noisemult(0:ndim), stat=allocstat)
78     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
79
80     dw=0.d0
81     dwmult=0.d0
82
83     dwar=0.d0
84     dwor=0.d0
85     dwau=0.d0
86     dwou=0.d0
87
88     anoise=0.d0
89     noise=0.d0
90     noisemult=0.d0
91
92     compute_mult=((q1fill).OR.(mult))

```

7.14.2.6 subroutine, public rk2_mtv_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), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res, real(kind=8), dimension(0:ndim), intent(out) tend)

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

Parameters

<i>y</i>	Initial point.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastic integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.
<i>tend</i>	Partial or full tendencies used to perform the step (used for debugging).

Definition at line 124 of file rk2_MTV_integrator.f90.

```

124     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
125     REAL(KIND=8), INTENT(INOUT) :: t
126     REAL(KIND=8), INTENT(IN) :: dt,dtn
127     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res,tend
128
129     CALL compg(y)
130
131     CALL stoch_atm_res_vec(dwar)
132     CALL stoch_oc_res_vec(dwor)
133     anoise=q_ar*dwar+q_or*dwor
134     CALL stoch_vec(dw)
135     IF (compute_mult) CALL stoch_vec(dwmult)
136     noise(1:ndim)=matmul(sig2,dw(1:ndim))
137     IF ((mult).and.(mod(t,mnuti)<dt)) CALL compute_mult_sigma(y)
138     IF (compute_mult) noisemult(1:ndim)=matmul(sig1,dwmult(1:ndim))
139
140     CALL tendencies(t,y,buf_f0)
141     buf_y1 = y+dt*(buf_f0+g)+(anoise+sq2*(noise+noisemult))*dtn
142
143     buf_f1=g
144     CALL compg(buf_y1)
145     g=0.5*(g+buf_f1)
146
147     IF ((mult).and.(mod(t,mnuti)<dt)) CALL compute_mult_sigma(buf_y1)
148     IF (compute_mult) THEN
149         buf_f1(1:ndim)=matmul(sig1,dwmult(1:ndim))

```

```

150         noisemult(1:ndim)=0.5*(noisemult(1:ndim)+buf_f1(1:ndim))
151     ENDIF
152
153
154     CALL tendencies(t,buf_y1,buf_f1)
155     buf_f0=0.5*(buf_f0+buf_f1)
156     res=y+dt*(buf_f0+g)+(anoise+sq2*(noise+noisemult))*dtn
157     ! tend=G+sq2*(noise+noisemult)/dtn
158     tend=sq2*noisemult/dtn
159     t=t+dt
160

```

7.14.3 Variable Documentation

7.14.3.1 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::anoise` [private]

Definition at line 33 of file rk2_MTV_integrator.f90.

```

33  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: anoise,noise      !< Additive noise term

```

7.14.3.2 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::buf_f0` [private]

Definition at line 30 of file rk2_MTV_integrator.f90.

7.14.3.3 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::buf_f1` [private]

Integration buffers.

Definition at line 30 of file rk2_MTV_integrator.f90.

7.14.3.4 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::buf_g` [private]

Buffer for the G term computation.

Definition at line 36 of file rk2_MTV_integrator.f90.

```

36  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_g          !< Buffer for the G term computation

```

7.14.3.5 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::buf_y1` [private]

Definition at line 30 of file rk2_MTV_integrator.f90.

```

30  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1,buf_f0,buf_f1 !< Integration buffers

```

7.14.3.6 logical rk2_mtv_integrator::compute_mult [private]

Logical indicating if the Gaussian noise for the multiplicative noise must be computed.

Definition at line 40 of file rk2_MTV_integrator.f90.

```
40  LOGICAL :: compute_mult                                !< Logical indicating if the Gaussian
    noise for the multiplicative noise must be computed
```

7.14.3.7 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dw [private]

Definition at line 31 of file rk2_MTV_integrator.f90.

```
31  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dw, dwmult    !< Standard gaussian noise buffers
```

7.14.3.8 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwau [private]

Definition at line 32 of file rk2_MTV_integrator.f90.

```
32  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dwau, dwor, dwou !< Standard gaussian noise buffers
```

7.14.3.9 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwor [private]

Definition at line 32 of file rk2_MTV_integrator.f90.

7.14.3.10 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwmult [private]

Standard gaussian noise buffers.

Definition at line 31 of file rk2_MTV_integrator.f90.

7.14.3.11 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwor [private]

Definition at line 32 of file rk2_MTV_integrator.f90.

7.14.3.12 real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwou [private]

Standard gaussian noise buffers.

Definition at line 32 of file rk2_MTV_integrator.f90.

7.14.3.13 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::g` [private]

G term of the MTV tendencies.

Definition at line 35 of file rk2_MTV_integrator.f90.

```
35  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: g                                !< G term of the MTV tendencies
```

7.14.3.14 `logical rk2_mtv_integrator::mult` [private]

Logical indicating if the sigma1 matrix must be computed for every state change.

Definition at line 38 of file rk2_MTV_integrator.f90.

```
38  LOGICAL :: mult                                !< Logical indicating if the sigma1
    matrix must be computed for every state change
```

7.14.3.15 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::noise` [private]

Additive noise term.

Definition at line 33 of file rk2_MTV_integrator.f90.

7.14.3.16 `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::noisemult` [private]

Multiplicative noise term.

Definition at line 34 of file rk2_MTV_integrator.f90.

```
34  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: noisemult                        !< Multiplicative noise term
```

7.14.3.17 `logical rk2_mtv_integrator::q1fill` [private]

Logical indicating if the matrix Q1 is non-zero.

Definition at line 39 of file rk2_MTV_integrator.f90.

```
39  LOGICAL :: q1fill                                !< Logical indicating if the matrix Q1 is
    non-zero
```

7.14.3.18 `real(kind=8), parameter rk2_mtv_integrator::sq2 = sqrt(2.D0)` [private]

Hard coded square root of 2.

Definition at line 42 of file rk2_MTV_integrator.f90.

```
42  REAL(KIND=8), PARAMETER :: sq2 = sqrt(2.d0)                                !< Hard coded square root of 2
```

7.15 rk2_ss_integrator Module Reference

Module with the stochastic uncoupled resolved nonlinear and tangent linear rk2 dynamics integration routines.

Functions/Subroutines

- subroutine, public [init_ss_integrator](#)
Subroutine to initialize the uncoupled resolved rk2 integrator.
- subroutine, public [tendencies](#) (t, y, res)
Routine computing the tendencies of the uncoupled resolved model.
- subroutine, public [tl_tendencies](#) (t, y, ys, res)
Tendencies for the tangent linear model of the uncoupled resolved dynamics in point ystar for perturbation deltat.
- subroutine, public [ss_step](#) (y, ys, t, dt, dtn, res)
Routine to perform a stochastic integration step of the unresolved uncoupled dynamics (Heun algorithm). The incremented time is returned.
- subroutine, public [ss_tl_step](#) (y, ys, t, dt, dtn, res)
Routine to perform a stochastic integration step of the unresolved uncoupled tangent linear dynamics (Heun algorithm). The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [dwar](#)
- real(kind=8), dimension(:), allocatable [dwor](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [anoise](#)
Additive noise term.
- real(kind=8), dimension(:), allocatable [buf_y1](#)
- real(kind=8), dimension(:), allocatable [buf_f0](#)
- real(kind=8), dimension(:), allocatable [buf_f1](#)
Integration buffers.

7.15.1 Detailed Description

Module with the stochastic uncoupled resolved nonlinear and tangent linear rk2 dynamics integration routines.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

This module actually contains the Heun algorithm routines.

7.15.2 Function/Subroutine Documentation

7.15.2.1 subroutine, public rk2_ss_integrator::init_ss_integrator ()

Subroutine to initialize the uncoupled resolved rk2 integrator.

Definition at line 40 of file rk2_ss_integrator.f90.

```

40     INTEGER :: allocstat
41
42     ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim),stat=allocstat)
43     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
44
45     ALLOCATE(anoise(0:ndim),stat=allocstat)
46     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
47
48     ALLOCATE(dwar(0:ndim),dwor(0:ndim),stat=allocstat)
49     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
50
51     dwar=0.d0
52     dwor=0.d0
53

```

7.15.2.2 subroutine, public rk2_ss_integrator::ss_step (real(kind=8), dimension(0:ndim), intent(in) y, real(kind=8), dimension(0:ndim), intent(in) ys, real(kind=8), intent(inout) t, real(kind=8), intent(in) dt, real(kind=8), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res)

Routine to perform a stochastic integration step of the unresolved uncoupled dynamics (Heun algorithm). The incremented time is returned.

Parameters

<i>y</i>	Initial point.
<i>ys</i>	Dummy argument for compatibility.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastic integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.

Definition at line 92 of file rk2_ss_integrator.f90.

```

92     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ys
93     REAL(KIND=8), INTENT(INOUT) :: t
94     REAL(KIND=8), INTENT(IN) :: dt,dtn
95     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
96
97     CALL stoch_atm_res_vec(dwar)
98     CALL stoch_oc_res_vec(dwor)
99     anoise=(q_ar*dwar+q_or*dwor)*dtn
100     CALL tendencies(t,y,buf_f0)
101     buf_y1 = y+dt*buf_f0+anoise
102     CALL tendencies(t,buf_y1,buf_f1)
103     res=y+0.5*(buf_f0+buf_f1)*dt+anoise
104     t=t+dt

```

7.15.2.3 `subroutine, public rk2_ss_integrator::ss_tl_step (real(kind=8), dimension(0:ndim), intent(in) y, real(kind=8), dimension(0:ndim), intent(in) ys, real(kind=8), intent(inout) t, real(kind=8), intent(in) dt, real(kind=8), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res)`

Routine to perform a stochastic integration step of the unresolved uncoupled tangent linear dynamics (Heun algorithm). The incremented time is returned.

Parameters

<i>y</i>	Initial point.
<i>ys</i>	point in trajectory to which the tangent space belongs.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastic integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.

Definition at line 117 of file `rk2_ss_integrator.f90`.

```

117  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ys
118  REAL(KIND=8), INTENT(INOUT) :: t
119  REAL(KIND=8), INTENT(IN) :: dt,dtn
120  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
121
122  CALL stoch_atm_res_vec(dwar)
123  CALL stoch_oc_res_vec(dwor)
124  anoise=(q_ar*dwar+q_or*dwor)*dtn
125  CALL tl_tendencies(t,y,ys,buf_f0)
126  buf_y1 = y+dt*buf_f0+anoise
127  CALL tl_tendencies(t,buf_y1,ys,buf_f1)
128  res=y+0.5*(buf_f0+buf_f1)*dt+anoise
129  t=t+dt

```

7.15.2.4 `subroutine, public rk2_ss_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)`

Routine computing the tendencies of the uncoupled resolved model.

Parameters

<i>t</i>	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
<i>y</i>	Point at which the tendencies have to be computed.
<i>res</i>	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 63 of file `rk2_ss_integrator.f90`.

```

63  REAL(KIND=8), INTENT(IN) :: t
64  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
65  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
66  CALL sparse_mul3(ss_tensor, y, y, res)

```

7.15.2.5 subroutine, public rk2_ss_integrator::tl_tendencies (real(kind=8), intent(in) *t*, real(kind=8), dimension(0:ndim), intent(in) *y*, real(kind=8), dimension(0:ndim), intent(in) *ys*, real(kind=8), dimension(0:ndim), intent(out) *res*)

Tendencies for the tangent linear model of the uncoupled resolved dynamics in point ystar for perturbation deltax.

Parameters

<i>t</i>	time
<i>y</i>	point of the tangent space at which the tendencies have to be computed.
<i>ys</i>	point in trajectory to which the tangent space belongs.
<i>res</i>	vector to store the result.

Definition at line 76 of file rk2_ss_integrator.f90.

```

76      REAL(KIND=8), INTENT(IN) :: t
77      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y,ys
78      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
79      CALL sparse_mul3(ss_tl_tensor, y, ys, res)

```

7.15.3 Variable Documentation

7.15.3.1 real(kind=8), dimension(:), allocatable rk2_ss_integrator::anoise [private]

Additive noise term.

Definition at line 30 of file rk2_ss_integrator.f90.

```

30      REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: anoise !< Additive noise term

```

7.15.3.2 real(kind=8), dimension(:), allocatable rk2_ss_integrator::buf_f0 [private]

Definition at line 32 of file rk2_ss_integrator.f90.

7.15.3.3 real(kind=8), dimension(:), allocatable rk2_ss_integrator::buf_f1 [private]

Integration buffers.

Definition at line 32 of file rk2_ss_integrator.f90.

7.15.3.4 real(kind=8), dimension(:), allocatable rk2_ss_integrator::buf_y1 [private]

Definition at line 32 of file rk2_ss_integrator.f90.

```

32      REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1,buf_f0,buf_f1 !< Integration buffers

```

7.15.3.5 `real(kind=8), dimension(:), allocatable rk2_ss_integrator::dwar` [private]

Definition at line 28 of file `rk2_ss_integrator.f90`.

```
28  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dwar, dwor !< Standard gaussian noise buffers
```

7.15.3.6 `real(kind=8), dimension(:), allocatable rk2_ss_integrator::dwor` [private]

Standard gaussian noise buffers.

Definition at line 28 of file `rk2_ss_integrator.f90`.

7.16 `rk2_stoch_integrator` Module Reference

Module with the stochastic rk2 integration routines.

Functions/Subroutines

- subroutine, public `init_integrator` (force)
Subroutine to initialize the integrator.
- subroutine `tendencies` (t, y, res)
Routine computing the tendencies of the selected model.
- subroutine, public `step` (y, t, dt, dtn, res, tend)
Routine to perform a stochastic step of the selected dynamics (Heun algorithm). The incremented time is returned.

Variables

- `real(kind=8), dimension(:), allocatable dwar`
- `real(kind=8), dimension(:), allocatable dwau`
- `real(kind=8), dimension(:), allocatable dwor`
- `real(kind=8), dimension(:), allocatable dwou`
Standard gaussian noise buffers.
- `real(kind=8), dimension(:), allocatable buf_y1`
- `real(kind=8), dimension(:), allocatable buf_f0`
- `real(kind=8), dimension(:), allocatable buf_f1`
Integration buffers.
- `real(kind=8), dimension(:), allocatable anoise`
Additive noise term.
- `type(coolist), dimension(:), allocatable int_tensor`
Dummy tensor that will hold the tendencies tensor.

7.16.1 Detailed Description

Module with the stochastic rk2 integration routines.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

This module actually contains the Heun algorithm routines. There are four modes for this integrator:

- full: use the full dynamics
- ures: use the intrinsic unresolved dynamics
- qfst: use the quadratic terms of the unresolved tendencies
- reso: use the resolved dynamics alone

7.16.2 Function/Subroutine Documentation

7.16.2.1 subroutine, public rk2_stoch_integrator::init_integrator (character*4, intent(in), optional force)

Subroutine to initialize the integrator.

Parameters

<i>force</i>	Parameter to force the mode of the integrator
--------------	---

Definition at line 48 of file rk2_stoch_integrator.f90.

```

48     INTEGER :: allocstat
49     CHARACTER*4, INTENT(IN), OPTIONAL :: force
50     CHARACTER*4 :: test
51
52     ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim),stat=allocstat)
53     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
54
55     ALLOCATE(anoise(0:ndim),stat=allocstat)
56     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
57
58     ALLOCATE(dwar(0:ndim),dwau(0:ndim),dwor(0:ndim),dwou(0:ndim),
59     stat=allocstat)
60     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
61
62     ALLOCATE(int_tensor(ndim),stat=allocstat)
63     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
64
65     dwar=0.d0
66     dwor=0.d0
67     dwau=0.d0
68     dwou=0.d0
69
70     IF (PRESENT(force)) THEN
71         test=force
72     ELSE
73         test=mode
74     ENDIF
75
76     SELECT CASE (test)
77     CASE('full')
78         CALL copy_tensor(aotensor,int_tensor)
79     CASE('ures')
80         CALL copy_tensor(ff_tensor,int_tensor)

```

```

80      CASE('qfst')
81          CALL copy_tensor(byyy,int_tensor)
82      CASE('reso')
83          CALL copy_tensor(ss_tensor,int_tensor)
84      CASE DEFAULT
85          stop '*** MODE variable not properly defined ***'
86      END SELECT
87

```

7.16.2.2 subroutine, public `rk2_stoch_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)`, `intent(in)` *dtn*, `real(kind=8)`, `dimension(0:ndim)`, `intent(out)` *res*, `real(kind=8)`, `dimension(0:ndim)`, `intent(out)` *tend*)

Routine to perform a stochastic step of the selected dynamics (Heun algorithm). The incremented time is returned.

Parameters

<i>y</i>	Initial point.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastic integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.
<i>tend</i>	Partial or full tendencies used to perform the step (used for debugging).

Definition at line 112 of file `rk2_stoch_integrator.f90`.

```

112      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
113      REAL(KIND=8), INTENT(INOUT) :: t
114      REAL(KIND=8), INTENT(IN) :: dt,dtn
115      REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res,tend
116
117      CALL stoch_atm_res_vec(dwar)
118      CALL stoch_atm_unres_vec(dwau)
119      CALL stoch_oc_res_vec(dwor)
120      CALL stoch_oc_unres_vec(dwou)
121      anoise=(q_ar*dwar+q_au*dwau+q_or*dwor+q_ou*dwou)*dtn
122      CALL tendencies(t,y,buf_f0)
123      CALL sparse_mul3(int_tensor,y,y,tend)
124      buf_y1 = y+dt*buf_f0+anoise
125      CALL sparse_mul3(int_tensor,buf_y1,buf_y1,buf_f1)
126      tend=0.5*(tend+buf_f1)
127      CALL tendencies(t,buf_y1,buf_f1)
128      res=y+0.5*(buf_f0+buf_f1)*dt+anoise
129      t=t+dt

```

7.16.2.3 subroutine, public `rk2_stoch_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 selected model.

Parameters

<i>t</i>	Time at which the tendencies have to be computed. Actually not needed for autonomous systems.
<i>y</i>	Point at which the tendencies have to be computed.
<i>res</i>	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 97 of file `rk2_stoch_integrator.f90`.

```

97     REAL(KIND=8), INTENT(IN) :: t
98     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
99     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
100     CALL sparse_mul3(int_tensor, y, y, res)

```

7.16.3 Variable Documentation

7.16.3.1 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::anoise` [private]

Additive noise term.

Definition at line 37 of file `rk2_stoch_integrator.f90`.

```

37     REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: anoise !< Additive noise term

```

7.16.3.2 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::buf_f0` [private]

Definition at line 35 of file `rk2_stoch_integrator.f90`.

7.16.3.3 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::buf_f1` [private]

Integration buffers.

Definition at line 35 of file `rk2_stoch_integrator.f90`.

7.16.3.4 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::buf_y1` [private]

Definition at line 35 of file `rk2_stoch_integrator.f90`.

```

35     REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1,buf_f0,buf_f1 !< Integration buffers

```

7.16.3.5 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::dwar` [private]

Definition at line 33 of file `rk2_stoch_integrator.f90`.

```

33     REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dwar,dwau,dwor,dwou !< Standard gaussian noise buffers

```

7.16.3.6 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::dwau [private]`

Definition at line 33 of file `rk2_stoch_integrator.f90`.

7.16.3.7 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::dwor [private]`

Definition at line 33 of file `rk2_stoch_integrator.f90`.

7.16.3.8 `real(kind=8), dimension(:), allocatable rk2_stoch_integrator::dwou [private]`

Standard gaussian noise buffers.

Definition at line 33 of file `rk2_stoch_integrator.f90`.

7.16.3.9 `type(coolist), dimension(:), allocatable rk2_stoch_integrator::int_tensor [private]`

Dummy tensor that will hold the tendencies tensor.

Definition at line 39 of file `rk2_stoch_integrator.f90`.

```
39  TYPE(coolist), DIMENSION(:), ALLOCATABLE :: int_tensor !< Dummy tensor that will hold the
    tendencies tensor
```

7.17 rk2_wl_integrator Module Reference

Module with the WL rk2 integration routines.

Functions/Subroutines

- subroutine, public `init_integrator`
Subroutine that initialize the MARs, the memory unit and the integration buffers.
- subroutine `compute_m1` (y)
Routine to compute the M_1 term.
- subroutine `compute_m2` (y)
Routine to compute the M_2 term.
- subroutine, public `step` (y, t, dt, dtn, res, tend)
Routine to perform an integration step (Heun algorithm) of the WL system. The incremented time is returned.
- subroutine, public `full_step` (y, t, dt, dtn, res)
Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [buf_y1](#)
 - real(kind=8), dimension(:), allocatable [buf_f0](#)
 - real(kind=8), dimension(:), allocatable [buf_f1](#)
- Integration buffers.*
- real(kind=8), dimension(:), allocatable [buf_m2](#)
 - real(kind=8), dimension(:), allocatable [buf_m1](#)
 - real(kind=8), dimension(:), allocatable [buf_m3](#)
 - real(kind=8), dimension(:), allocatable [buf_m](#)
 - real(kind=8), dimension(:), allocatable [buf_m3s](#)
- Dummy buffers holding the terms /f\$M_i.*
- real(kind=8), dimension(:), allocatable [anoise](#)
- Additive noise term.*
- real(kind=8), dimension(:), allocatable [dwar](#)
 - real(kind=8), dimension(:), allocatable [dwau](#)
 - real(kind=8), dimension(:), allocatable [dwor](#)
 - real(kind=8), dimension(:), allocatable [dwou](#)
- Standard gaussian noise buffers.*
- real(kind=8), dimension(:, :), allocatable [x1](#)
- Buffer holding the subsequent states of the first MAR.*
- real(kind=8), dimension(:, :), allocatable [x2](#)
- Buffer holding the subsequent states of the second MAR.*

7.17.1 Detailed Description

Module with the WL rk2 integration routines.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

This module actually contains the Heun algorithm routines.

7.17.2 Function/Subroutine Documentation

7.17.2.1 subroutine rk2_wl_integrator::compute_m1 (real(kind=8), dimension(0:ndim), intent(in) y) [private]

Routine to compute the M_1 term.

Parameters

<i>y</i>	Present state of the WL system
----------	--------------------------------

Definition at line 106 of file rk2_WL_integrator.f90.

```
106      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
```

```

107     buf_m1=0.d0
108     IF (m12def) CALL sparse_mul2_k(m12, y, buf_m1)
109     buf_m1=buf_m1+m1tot

```

7.17.2.2 subroutine rk2_wl_integrator::compute_m2 (real(kind=8), dimension(0:ndim), intent(in) y) [private]

Routine to compute the M_2 term.

Parameters

<i>y</i>	Present state of the WL system
----------	--------------------------------

Definition at line 115 of file rk2_WL_integrator.f90.

```

115     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
116     buf_m=0.d0
117     buf_m2=0.d0
118     IF (m21def) CALL sparse_mul3(m21, y, x1(0:ndim,1), buf_m)
119     IF (m22def) CALL sparse_mul3(m22, x2(0:ndim,1), x2(0:ndim,1), buf_m2)
120     buf_m2=buf_m2+buf_m

```

7.17.2.3 subroutine, public rk2_wl_integrator::full_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), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res)

Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Parameters

<i>y</i>	Initial point.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastoc integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.

Definition at line 185 of file rk2_WL_integrator.f90.

```

185     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
186     REAL(KIND=8), INTENT(INOUT) :: t
187     REAL(KIND=8), INTENT(IN) :: dt,dtn
188     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
189     CALL stoch_atm_res_vec(dwar)
190     CALL stoch_atm_unres_vec(dwau)
191     CALL stoch_oc_res_vec(dwor)
192     CALL stoch_oc_unres_vec(dwou)
193     anoise=(q_ar*dwar+q_au*dwau+q_or*dwor+q_ou*dwou)*dtn
194     CALL sparse_mul3(aotensor,y,y,buf_f0)
195     buf_y1 = y+dt*buf_f0+anoise
196     CALL sparse_mul3(aotensor,buf_y1,buf_y1,buf_f1)
197     res=y+0.5*(buf_f0+buf_f1)*dt+anoise
198     t=t+dt

```

7.17.2.4 subroutine, public rk2_wl_integrator::init_integrator ()

Subroutine that initialize the MARs, the memory unit and the integration buffers.

Definition at line 44 of file rk2_WL_integrator.f90.

```

44     INTEGER :: allocstat,i
45
46     CALL init_ss_integrator
47
48     print*, 'Initializing the integrator ...'
49
50     IF (mode.ne.'ures') THEN
51         print*, '*** Mode set to ',mode,' in stoch_params.nml ***'
52         print*, '*** WL configuration only support unresolved mode ***'
53         stop '*** Please change to 'ures' and perform the configuration again ! ***'
54     ENDIF
55
56     ALLOCATE(buf_y1(0:ndim),buf_f0(0:ndim),buf_f1(0:ndim),stat=allocstat)
57     IF (allocstat /= 0) stop '*** Not enough memory ! ***'
58
59     ALLOCATE(buf_m1(0:ndim), buf_m2(0:ndim), buf_m3(0:ndim), buf_m(0:
ndim), buf_m3s(0:ndim), stat=allocstat)
60     IF (allocstat /= 0) stop '*** Not enough memory ! ***'
61
62     ALLOCATE(dwar(0:ndim),dwau(0:ndim),dwor(0:ndim),dwou(0:ndim),
stat=allocstat)
63     IF (allocstat /= 0) stop '*** Not enough memory ! ***'
64
65     ALLOCATE(anoise(0:ndim), stat=allocstat)
66     IF (allocstat /= 0) stop '*** Not enough memory ! ***'
67
68     buf_y1=0.d0
69     buf_f1=0.d0
70     buf_f0=0.d0
71
72     dwar=0.d0
73     dwor=0.d0
74     dwau=0.d0
75     dwou=0.d0
76
77     buf_m1=0.d0
78     buf_m2=0.d0
79     buf_m3=0.d0
80     buf_m3s=0.d0
81     buf_m=0.d0
82
83     print*, 'Initializing the MARs ...'
84
85     CALL init_mar
86
87     ALLOCATE(x1(0:ndim,ms), x2(0:ndim,ms), stat=allocstat)
88
89     x1=0.d0
90     DO i=1,50000
91         CALL mar_step(x1)
92     ENDDO
93
94     x2=0.d0
95     DO i=1,50000
96         CALL mar_step(x2)
97     ENDDO
98
99     CALL init_memory
100

```

7.17.2.5 subroutine, public rk2_wl_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), intent(in) dtn, real(kind=8), dimension(0:ndim), intent(out) res, real(kind=8), dimension(0:ndim), intent(out) tend)

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

Parameters

<i>y</i>	Initial point.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>dtn</i>	Stochastic integration timestep (normally square-root of dt).
<i>res</i>	Final point after the step.
<i>tend</i>	Partial or full tendencies used to perform the step (used for debugging).

Definition at line 132 of file rk2_WL_integrator.f90.

```

132  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
133  REAL(KIND=8), INTENT(INOUT) :: t
134  REAL(KIND=8), INTENT(IN) :: dt,dtn
135  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res,tend
136  INTEGER :: i
137
138  IF (mod(t,muti)<dt) THEN
139    CALL compute_m3(y,dts,dtsn,.true.,.true.,.true.,muti/2,buf_m3s)
140    buf_m3=buf_m3s
141    DO i=1,1
142      CALL compute_m3(y,dts,dtsn,.false.,.true.,.true.,muti/2,buf_m3s)
143      buf_m3=buf_m3+buf_m3s
144    ENDDO
145    !DO i=1,2
146    !  CALL compute_M3(y,dts,dtsn,.false.,.true.,.true.,muti/2,buf_M3s)
147    !  buf_M3=buf_M3+buf_M3s
148    !ENDDO
149    buf_m3=buf_m3/2
150  ENDIF
151
152
153  CALL stoch_atm_res_vec(dwar)
154  CALL stoch_oc_res_vec(dwor)
155  anoise=(q_ar*dwar+q_or*dwor)*dtn
156
157  CALL tendencies(t,y,buf_f0)
158  CALL mar_step(x1)
159  CALL mar_step(x2)
160  CALL compute_m1(y)
161  CALL compute_m2(y)
162  buf_f0= buf_f0+buf_m1+buf_m2+buf_m3
163  buf_y1 = y+dt*buf_f0+anoise
164
165  CALL tendencies(t+dt,buf_y1,buf_f1)
166  CALL compute_m1(buf_y1)
167  CALL compute_m2(buf_y1)
168  !IF (mod(t,muti)<dt) CALL compute_M3(buf_y1,dts,dtsn,.false.,.true.,buf_M3)
169
170  buf_f0=0.5*(buf_f0+buf_f1+buf_m1+buf_m2+buf_m3)
171  res=y+dt*buf_f0+anoise
172
173  tend=buf_m3
174  t=t+dt
175

```

7.17.3 Variable Documentation

7.17.3.1 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::anoise` [private]

Additive noise term.

Definition at line 33 of file rk2_WL_integrator.f90.

```

33  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: anoise      !< Additive noise term

```

7.17.3.2 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_f0` [private]

Definition at line 31 of file rk2_WL_integrator.f90.

7.17.3.3 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_f1` [private]

Integration buffers.

Definition at line 31 of file rk2_WL_integrator.f90.

7.17.3.4 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_m` [private]

Definition at line 32 of file rk2_WL_integrator.f90.

7.17.3.5 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_m1` [private]

Definition at line 32 of file rk2_WL_integrator.f90.

7.17.3.6 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_m2` [private]

Definition at line 32 of file rk2_WL_integrator.f90.

```
32  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_m2,buf_m1,buf_m3,buf_m,buf_m3s !< Dummy buffers holding
    the terms /f$M_i\f$ of the parameterization
```

7.17.3.7 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_m3` [private]

Definition at line 32 of file rk2_WL_integrator.f90.

7.17.3.8 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_m3s` [private]

Dummy buffers holding the terms /f\$M_i.

Definition at line 32 of file rk2_WL_integrator.f90.

7.17.3.9 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::buf_y1` [private]

Definition at line 31 of file rk2_WL_integrator.f90.

```
31  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: buf_y1,buf_f0,buf_f1 !< Integration buffers
```

7.17.3.10 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::dwar` [private]

Definition at line 34 of file rk2_WL_integrator.f90.

```
34  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dwar,dwau,dwor,dwou !< Standard gaussian noise buffers
```

7.17.3.11 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::dwau` [private]

Definition at line 34 of file rk2_WL_integrator.f90.

7.17.3.12 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::dwor` [private]

Definition at line 34 of file rk2_WL_integrator.f90.

7.17.3.13 `real(kind=8), dimension(:), allocatable rk2_wl_integrator::dwou` [private]

Standard gaussian noise buffers.

Definition at line 34 of file rk2_WL_integrator.f90.

7.17.3.14 `real(kind=8), dimension(:, :), allocatable rk2_wl_integrator::x1` [private]

Buffer holding the subsequent states of the first MAR.

Definition at line 36 of file rk2_WL_integrator.f90.

```
36  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: x1 !< Buffer holding the subsequent states of the first MAR
```

7.17.3.15 `real(kind=8), dimension(:, :), allocatable rk2_wl_integrator::x2` [private]

Buffer holding the subsequent states of the second MAR.

Definition at line 37 of file rk2_WL_integrator.f90.

```
37  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: x2 !< Buffer holding the subsequent states of the second MAR
```

7.18 sf_def Module Reference

Module to select the resolved-unresolved components.

Functions/Subroutines

- subroutine, public [load_sf](#)

Subroutine to load the unresolved variable definition vector SF from $SF.nml$ if it exists. If it does not, then write $SF.nml$ with no unresolved variables specified (null vector).

Variables

- logical [exists](#)

Boolean to test for file existence.

- integer, dimension(:), allocatable, public [sf](#)

Unresolved variable definition vector.

- integer, dimension(:), allocatable, public [ind](#)
- integer, dimension(:), allocatable, public [rind](#)

Unresolved reduction indices.

- integer, dimension(:), allocatable, public [sl_ind](#)
- integer, dimension(:), allocatable, public [sl_rind](#)

Resolved reduction indices.

- integer, public [n_unres](#)

Number of unresolved variables.

- integer, public [n_res](#)

Number of resolved variables.

- integer, dimension(:,:), allocatable, public [bar](#)
- integer, dimension(:,:), allocatable, public [bau](#)
- integer, dimension(:,:), allocatable, public [bor](#)
- integer, dimension(:,:), allocatable, public [bou](#)

Filter matrices.

7.18.1 Detailed Description

Module to select the resolved-unresolved components.

Copyright

2017 Jonathan Demaeyer See [LICENSE.txt](#) for license information.

7.18.2 Function/Subroutine Documentation

7.18.2.1 subroutine, public sf_def::load_sf ()

Subroutine to load the unresolved variable definition vector SF from $SF.nml$ if it exists. If it does not, then write $SF.nml$ with no unresolved variables specified (null vector).

Definition at line 37 of file sf_def.f90.

```

37     INTEGER :: i,allocstat,n,ns
38     CHARACTER(len=20) :: fm
39
40     namelist /sflist/ sf
41
42     fm(1:6)=' (F3.1)'
43
44     IF (ndim == 0) stop "*** Number of dimensions is 0! ***"
45     ALLOCATE(sf(0:ndim), stat=allocstat)
46     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
47
48     INQUIRE(file='./SF.nml',exist=exists)
49
50     IF (exists) THEN
51         OPEN(8, file="SF.nml", status='OLD', recl=80, delim='APOSTROPHE')
52         READ(8,nml=sflist)
53         CLOSE(8)
54         n_unres=0
55         DO i=1,ndim ! Computing the number of unresolved variables
56             IF (sf(i)==1) n_unres=n_unres+1
57         ENDDO
58         IF (n_unres==0) stop "*** No unresolved variable specified! ***"
59         n_res=ndim-n_unres
60         ALLOCATE(ind(n_unres), rind(0:ndim), sl_ind(n_res), sl_rind(0:ndim),
stat=allocstat)
61         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
62         ALLOCATE(bar(0:ndim,0:ndim), bau(0:ndim,0:ndim), bor(0:
ndim,0:ndim), bou(0:ndim,0:ndim), stat=allocstat)
63         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
64         rind=0
65         n=1
66         ns=1
67         DO i=1,ndim
68             IF (sf(i)==1) THEN
69                 ind(n)=i
70                 rind(i)=n
71                 n=n+1
72             ELSE
73                 sl_ind(ns)=i
74                 sl_rind(i)=ns
75                 ns=ns+1
76             ENDIF
77         ENDDO
78         bar=0
79         bau=0
80         bor=0
81         bou=0
82         DO i=1,2*natm
83             IF (sf(i)==1) THEN
84                 bau(i,i)=1
85             ELSE
86                 bar(i,i)=1
87             ENDIF
88         ENDDO
89         DO i=2*natm+1,ndim
90             IF (sf(i)==1) THEN
91                 bou(i,i)=1
92             ELSE
93                 bor(i,i)=1
94             ENDIF
95         ENDDO
96     ELSE
97         OPEN(8, file="SF.nml", status='NEW')
98         WRITE(8,'(a)') " !-----!"
99         WRITE(8,'(a)') " ! Namelist file : !"
100        WRITE(8,'(a)') " ! Unresolved variables specification (1 -> unresolved, 0 -> resolved) !"
101        WRITE(8,'(a)') " !-----!"
102        WRITE(8,*) ""
103        WRITE(8,'(a)') "%SFLIST"
104        WRITE(8,*) " ! psi variables"
105        DO i=1,natm
106            WRITE(8,*) " SF("//trim(str(i))//") = 0"// " ! typ= "&
&//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
&%Nx,fm))//", Ny= "//trim(rstr(awavenum(i)%Ny,fm))
107        END DO
108        WRITE(8,*) " ! theta variables"
109        DO i=1,natm
110            WRITE(8,*) " SF("//trim(str(i+natm))//") = 0"// " ! typ= "&
&//awavenum(i)%typ//", Nx= "//trim(rstr(awavenum(i)&
&%Nx,fm))//", Ny= "//trim(rstr(awavenum(i)%Ny,fm))
111        END DO
112        WRITE(8,*) " ! A variables"
113        DO i=1,noc
114            WRITE(8,*) " SF("//trim(str(i+2*natm))//") = 0"// " ! Nx&
&= "//trim(rstr(owavenum(i)%Nx,fm))//", Ny= "&
115
116
117
118
119
120

```

```

121         & //trim(rstr(owavenum(i)%Ny, fm))
122     END DO
123     WRITE(8,*) " ! T variables"
124     DO i=1,noc
125         WRITE(8,*) " SF("//trim(str(i+noc+2*natm))//") = 0"// " &
126             & ! Nx= "//trim(rstr(owavenum(i)%Nx, fm))//", Ny= "&
127             & //trim(rstr(owavenum(i)%Ny, fm))
128     END DO
129
130     WRITE(8,'(a)') "&END"
131     WRITE(8,*) ""
132     CLOSE(8)
133     stop "*** SF.nml namelist written. Fill in the file and rerun !***"
134 ENDIF

```

7.18.3 Variable Documentation

7.18.3.1 integer, dimension(:,:), allocatable, public sf_def::bar

Definition at line 28 of file sf_def.f90.

```
28  INTEGER, DIMENSION(:,:), ALLOCATABLE, PUBLIC :: bar,bau,bor,bou !< Filter matrices
```

7.18.3.2 integer, dimension(:,:), allocatable, public sf_def::bau

Definition at line 28 of file sf_def.f90.

7.18.3.3 integer, dimension(:,:), allocatable, public sf_def::bor

Definition at line 28 of file sf_def.f90.

7.18.3.4 integer, dimension(:,:), allocatable, public sf_def::bou

Filter matrices.

Definition at line 28 of file sf_def.f90.

7.18.3.5 logical sf_def::exists [private]

Boolean to test for file existence.

Definition at line 21 of file sf_def.f90.

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

7.18.3.6 integer, dimension(:), allocatable, public sf_def::ind

Definition at line 24 of file sf_def.f90.

```
24  INTEGER, DIMENSION(:), ALLOCATABLE, PUBLIC :: ind,rind !< Unresolved reduction indices
```

7.18.3.7 integer, public sf_def::n_res

Number of resolved variables.

Definition at line 27 of file sf_def.f90.

```
27  INTEGER, PUBLIC :: n_res !< Number of resolved variables
```

7.18.3.8 integer, public sf_def::n_unres

Number of unresolved variables.

Definition at line 26 of file sf_def.f90.

```
26  INTEGER, PUBLIC :: n_unres !< Number of unresolved variables
```

7.18.3.9 integer, dimension(:), allocatable, public sf_def::rind

Unresolved reduction indices.

Definition at line 24 of file sf_def.f90.

7.18.3.10 integer, dimension(:), allocatable, public sf_def::sf

Unresolved variable definition vector.

Definition at line 23 of file sf_def.f90.

```
23  INTEGER, DIMENSION(:), ALLOCATABLE, PUBLIC :: sf !< Unresolved variable definition vector
```

7.18.3.11 integer, dimension(:), allocatable, public sf_def::sl_ind

Definition at line 25 of file sf_def.f90.

```
25  INTEGER, DIMENSION(:), ALLOCATABLE, PUBLIC :: sl_ind, sl_rind !< Resolved reduction indices
```

7.18.3.12 integer, dimension(:), allocatable, public sf_def::sl_rind

Resolved reduction indices.

Definition at line 25 of file sf_def.f90.

7.19 sigma Module Reference

The MTV noise sigma matrices used to integrate the MTV model.

Functions/Subroutines

- subroutine, public [init_sigma](#) (mult, Q1fill)
Subroutine to initialize the sigma matrices.
- subroutine, public [compute_mult_sigma](#) (y)
Routine to actualize the matrix σ_1 based on the state y of the MTV system.

Variables

- real(kind=8), dimension(:,:), allocatable, public [sig1](#)
 $\sigma_1(X)$ state-dependent noise matrix
- real(kind=8), dimension(:,:), allocatable, public [sig2](#)
 σ_2 state-independent noise matrix
- real(kind=8), dimension(:,:), allocatable, public [sig1r](#)
Reduced $\sigma_1(X)$ state-dependent noise matrix.
- real(kind=8), dimension(:,:), allocatable [dumb_mat1](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [dumb_mat2](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [dumb_mat3](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [dumb_mat4](#)
Dummy matrix.
- integer, dimension(:), allocatable [ind1](#)
- integer, dimension(:), allocatable [rind1](#)
- integer, dimension(:), allocatable [ind2](#)
- integer, dimension(:), allocatable [rind2](#)
Reduction indices.
- integer [n1](#)
- integer [n2](#)

7.19.1 Detailed Description

The MTV noise sigma matrices used to integrate the MTV model.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

See : Franzke, C., Majda, A. J., & Vanden-Eijnden, E. (2005). Low-order stochastic mode reduction for a realistic barotropic model climate. Journal of the atmospheric sciences, 62(6), 1722-1745.

7.19.2 Function/Subroutine Documentation

7.19.2.1 subroutine, public sigma::compute_mult_sigma (real(kind=8), dimension(0:ndim), intent(in) y)

Routine to actualize the matrix σ_1 based on the state y of the MTV system.

Parameters

y	State of the MTV system
---	-------------------------

Definition at line 93 of file MTV_sigma_tensor.f90.

```

93      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: y
94      INTEGER :: info,info2
95      CALL sparse_mul3_mat(utot,y,dumb_mat1)
96      CALL sparse_mul4_mat(vtot,y,y,dumb_mat2)
97      dumb_mat3=dumb_mat1+dumb_mat2+q1
98      CALL reduce(dumb_mat3,dumb_mat1,n1,ind1,rind1)
99      IF (n1 /= 0) THEN
100         CALL sqrtm_svd(dumb_mat1(1:n1,1:n1),dumb_mat2(1:n1,1:n1),info,info2,min(max(n1/2,2),64))
101         ! dumb_mat2=0.D0
102         ! CALL chol(0.5*(dumb_mat1(1:n1,1:n1)+transpose(dumb_mat1(1:n1,1:n1))),dumb_mat2(1:n1,1:n1),info)
103         IF ((.not.any(isnan(dumb_mat2))) .and. (info.eq.0) .and. (.not.any(dumb_mat2>huge(0.d0)))) THEN
104            CALL ireduce(sig1,dumb_mat2,n1,ind1,rind1)
105         ELSE
106            sig1=sig1r
107         ENDIF
108     ELSE
109         sig1=sig1r
110     ENDIF

```

7.19.2.2 subroutine, public sigma::init_sigma (logical, intent(out) mult, logical, intent(out) Q1fill)

Subroutine to initialize the sigma matrices.

Definition at line 48 of file MTV_sigma_tensor.f90.

```

48      LOGICAL, INTENT(OUT) :: mult,q1fill
49      INTEGER :: allocstat,info1,info2
50
51      CALL init_sqrt
52
53      ALLOCATE(sig1(ndim,ndim), sig2(ndim,ndim), sig1r(ndim,
ndim),stat=allocstat)
54      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
55
56      ALLOCATE(ind1(ndim), rind1(ndim), ind2(ndim), rind2(ndim),
stat=allocstat)
57      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
58
59      ALLOCATE(dumb_mat1(ndim,ndim), dumb_mat2(ndim,ndim), stat=allocstat)
60      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
61
62      ALLOCATE(dumb_mat3(ndim,ndim), dumb_mat4(ndim,ndim), stat=allocstat)
63      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
64
65      print*, "Initializing the sigma matrices"
66
67      CALL reduce(q2,dumb_mat1,n2,ind2,rind2)
68      IF (n2 /= 0) THEN
69         CALL sqrtm_svd(dumb_mat1(1:n2,1:n2),dumb_mat2(1:n2,1:n2),info1,info2,min(max(n2/2,2),64))
70         CALL ireduce(sig2,dumb_mat2,n2,ind2,rind2)
71      ELSE
72         sig2=0.d0
73      ENDIF
74
75      mult=(.not.((tensor_empty(utot)).and.(tensor4_empty(vtot))))
76      q1fill=.true.
77      CALL reduce(q1,dumb_mat1,n1,ind1,rind1)
78      IF (n1 /= 0) THEN
79
80         CALL sqrtm_svd(dumb_mat1(1:n1,1:n1),dumb_mat2(1:n1,1:n1),info1,info2,min(max(n1/2,2),64))
81         CALL ireduce(sig1,dumb_mat2,n1,ind1,rind1)
82      ELSE
83         q1fill=.false.
84         sig1=0.d0
85      ENDIF
86      sig1r=sig1
87

```

7.19.3 Variable Documentation

7.19.3.1 `real(kind=8), dimension(:, :), allocatable sigma::dumb_mat1` [private]

Dummy matrix.

Definition at line 35 of file MTV_sigma_tensor.f90.

```
35  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat1 !< Dummy matrix
```

7.19.3.2 `real(kind=8), dimension(:, :), allocatable sigma::dumb_mat2` [private]

Dummy matrix.

Definition at line 36 of file MTV_sigma_tensor.f90.

```
36  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat2 !< Dummy matrix
```

7.19.3.3 `real(kind=8), dimension(:, :), allocatable sigma::dumb_mat3` [private]

Dummy matrix.

Definition at line 37 of file MTV_sigma_tensor.f90.

```
37  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat3 !< Dummy matrix
```

7.19.3.4 `real(kind=8), dimension(:, :), allocatable sigma::dumb_mat4` [private]

Dummy matrix.

Definition at line 38 of file MTV_sigma_tensor.f90.

```
38  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat4 !< Dummy matrix
```

7.19.3.5 `integer, dimension(:), allocatable sigma::ind1` [private]

Definition at line 39 of file MTV_sigma_tensor.f90.

```
39  INTEGER, DIMENSION(:), ALLOCATABLE :: ind1, rind1, ind2, rind2 !< Reduction indices
```

7.19.3.6 `integer, dimension(:), allocatable sigma::ind2` [private]

Definition at line 39 of file MTV_sigma_tensor.f90.

7.19.3.7 integer sigma::n1 [private]

Definition at line 41 of file MTV_sigma_tensor.f90.

```
41  INTEGER :: n1,n2
```

7.19.3.8 integer sigma::n2 [private]

Definition at line 41 of file MTV_sigma_tensor.f90.

7.19.3.9 integer, dimension(:), allocatable sigma::rind1 [private]

Definition at line 39 of file MTV_sigma_tensor.f90.

7.19.3.10 integer, dimension(:), allocatable sigma::rind2 [private]

Reduction indices.

Definition at line 39 of file MTV_sigma_tensor.f90.

7.19.3.11 real(kind=8), dimension(:,,:), allocatable, public sigma::sig1

$\sigma_1(X)$ state-dependent noise matrix

Definition at line 31 of file MTV_sigma_tensor.f90.

```
31  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: sig1 !< \f$\sigma_1(X)\f$ state-dependent noise
    matrix
```

7.19.3.12 real(kind=8), dimension(:,,:), allocatable, public sigma::sig1r

Reduced $\sigma_1(X)$ state-dependent noise matrix.

Definition at line 33 of file MTV_sigma_tensor.f90.

```
33  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: sig1r !< Reduced \f$\sigma_1(X)\f$ state-dependent
    noise matrix
```

7.19.3.13 real(kind=8), dimension(:,,:), allocatable, public sigma::sig2

σ_2 state-independent noise matrix

Definition at line 32 of file MTV_sigma_tensor.f90.

```
32  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: sig2 !< \f$\sigma_2\f$ state-independent noise
    matrix
```


7.20 sqrt_mod Module Reference

Utility module with various routine to compute matrix square root.

Functions/Subroutines

- subroutine, public [init_sqrt](#)
- subroutine, public [sqrtm](#) (A, sqA, info, info_triu, bs)
Routine to compute a real square-root of a matrix.
- logical function [selectev](#) (a, b)
- subroutine [sqrtm_triu](#) (A, sqA, info, bs)
- subroutine [csqrtm_triu](#) (A, sqA, info, bs)
- subroutine [rsf2csf](#) (T, Z, Tz, Zz)
- subroutine, public [chol](#) (A, sqA, info)
Routine to perform a Cholesky decomposition.
- subroutine, public [sqrtm_svd](#) (A, sqA, info, info_triu, bs)
Routine to compute a real square-root of a matrix via a SVD decomposition.

Variables

- real(kind=8), dimension(:), allocatable [work](#)
- integer [lwork](#)
- real(kind=8), parameter [real_eps](#) = 2.2204460492503131e-16

7.20.1 Detailed Description

Utility module with various routine to compute matrix square root.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

Mainly based on the numerical recipes and from: Edwin Deadman, Nicholas J. Higham, Rui Ralha (2013) "↔ Blocked Schur Algorithms for Computing the Matrix Square Root", Lecture Notes in Computer Science, 7782. pp. 171-182.

7.20.2 Function/Subroutine Documentation

- 7.20.2.1 subroutine, public sqrt_mod::chol (real(kind=8), dimension(:,,:), intent(in) A, real(kind=8), dimension(:,,:), intent(out) sqA, integer, intent(out) info)

Routine to perform a Cholesky decomposition.

Parameters

<i>A</i>	Matrix whose decomposition is evaluated.
<i>sqA</i>	Cholesky decomposition of <i>A</i> .
<i>info</i>	Information code returned by the Lapack routines.

Definition at line 386 of file `sqrt_mod.f90`.

```

386     REAL(KIND=8), DIMENSION(:,:), INTENT(IN) :: a
387     REAL(KIND=8), DIMENSION(:,:), INTENT(OUT) :: sqA
388     INTEGER, INTENT(OUT) :: info
389
390     sqA=a
391     CALL dpotrf('L',SIZE(sqA,1),sqA,SIZE(sqA,1),info)

```

7.20.2.2 subroutine `sqrt_mod::csqrtm_triu` (`complex(kind=16), dimension(:,:), intent(in) A`, `complex(kind=16), dimension(:,:), intent(out) sqA`, `integer, intent(out) info`, `integer, intent(in), optional bs`) [private]

Definition at line 235 of file `sqrt_mod.f90`.

```

235     COMPLEX(KIND=16), DIMENSION(:,:), INTENT(IN) :: a
236     INTEGER, INTENT(IN), OPTIONAL :: bs
237     COMPLEX(KIND=16), DIMENSION(:,:), INTENT(OUT) :: sqA
238     INTEGER, INTENT(OUT) :: info
239     COMPLEX(KIND=16), DIMENSION(SIZE(A,1)) :: a_diag
240     COMPLEX(KIND=16), DIMENSION(SIZE(A,1),SIZE(A,1)) :: r,sm,rii,rjj
241     INTEGER, DIMENSION(2*SIZE(A,1),2) :: start_stop_pairs
242     COMPLEX(KIND=16) :: s,denom,scale
243     INTEGER :: i,j,k,start,n,sstop,m
244     INTEGER :: istart,istop,jstart,jstop
245     INTEGER :: nblocks,blocksize
246     INTEGER :: bsmall,blarge,nlarge,nsmall
247
248     blocksize=64
249     IF (PRESENT(bs)) blocksize=bs
250     n=SIZE(a,1)
251     ! print*, blocksize
252
253     CALL cdiag(a,a_diag)
254     r=0.d0
255     DO i=1,n
256         r(i,i)=sqrt(a_diag(i))
257     ENDDO
258
259
260     nblocks=max(floordiv(n,blocksize),1)
261     bsmall=floordiv(n,nblocks)
262     nlarge=mod(n,nblocks)
263     blarge=bsmall+1
264     nsmall=nblocks-nlarge
265     IF (nsmall*bsmall + nlarge*blarge /= n) stop 'Sqrtm: Internal inconsistency'
266
267     ! print*, nblocks,bsmall,nsmall,blarge,nlarge
268
269     start=1
270     DO i=1,nsmall
271         start_stop_pairs(i,1)=start
272         start_stop_pairs(i,2)=start+bsmall-1
273         start=start+bsmall
274     ENDDO
275     DO i=nsmall+1,nsmall+nlarge
276         start_stop_pairs(i,1)=start
277         start_stop_pairs(i,2)=start+blarge-1
278         start=start+blarge
279     ENDDO
280
281     ! DO i=1,SIZE(start_stop_pairs,1)
282     !     print*, i
283     !     print*, start_stop_pairs(i,1),start_stop_pairs(i,2)
284     ! END DO
285
286     DO k=1,nsmall+nlarge

```

```

287      start=start_stop_pairs(k,1)
288      sstop=start_stop_pairs(k,2)
289      DO j=start,sstop
290          DO i=j-1,start,-1
291              s=0.d0
292              IF (j-i>1) s= dot_product(r(i,i+1:j-1),r(i+1:j-1,j))
293              denom= r(i,i)+r(j,j)
294              IF (denom==0.d0) stop 'Sqrtm: Failed to find the matrix square root'
295              r(i,j)=(a(i,j)-s)/denom
296          END DO
297      END DO
298  END DO
299
300      ! print*, 'R'
301      ! CALL printmat(R)
302
303      DO j=1,nblocks
304          jstart=start_stop_pairs(j,1)
305          jstop=start_stop_pairs(j,2)
306          DO i=j-1,1,-1
307              istart=start_stop_pairs(i,1)
308              istop=start_stop_pairs(i,2)
309              sm=0.d0
310              sm(istart:istop,jstart:jstop)=a(istart:istop,jstart:jstop)
311              IF (j-i>1) sm(istart:istop,jstart:jstop) = sm(istart:istop&
312                  &,jstart:jstop) - matmul(r(istart:istop,istop:jstart)&
313                  &,r(istop:jstart,jstart:jstop))
314              rii=0.d0
315              rii = r(istart:istop, istart:istop)
316              rjj=0.d0
317              rjj = r(jstart:jstop, jstart:jstop)
318              m=istop-istart+1
319              n=jstop-jstart+1
320              k=1
321              ! print*, m,n
322              ! print*, istart,istop
323              ! print*, jstart,jstop
324
325              ! print*, 'Rii',Rii(istart:istop, istart:istop)
326              ! print*, 'Rjj',Rjj(jstart:jstop,jstart:jstop)
327              ! print*, 'Sm',Sm(istart:istop,jstart:jstop)
328
329              CALL ztrsyl('N','N',k,m,n,rii(istart:istop, istart:istop),m&
330                  &,rjj(jstart:jstop,jstart:jstop),n,sm(istart:istop&
331                  &,jstart:jstop),m,scale,info)
332              r(istart:istop,jstart:jstop)=sm(istart:istop,jstart:jstop)*scale
333          ENDDO
334      ENDDO
335      sqs=r

```

7.20.2.3 subroutine, public sqrt_mod::init_sqrt ()

Definition at line 39 of file sqrt_mod.f90.

```

39      INTEGER :: allocstat
40      lwork=10
41      lwork=ndim*lwork
42
43      ! print*, lwork
44
45      IF (ALLOCATED(work)) THEN
46          DEALLOCATE(work, stat=allocstat)
47          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
48      ENDIF
49      ALLOCATE(work(lwork), stat=allocstat)
50      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
51
52      ! ALLOCATE(zwork(lwork), STAT=AllocStat)
53      ! IF (AllocStat /= 0) STOP "*** Not enough memory ! ***"

```

7.20.2.4 subroutine sqrt_mod::rsf2csf (real(kind=8), dimension(:,,:), intent(in) T, real(kind=8), dimension(:,,:), intent(in) Z, complex(kind=16), dimension(:,,:), intent(out) Tz, complex(kind=16), dimension(:,,:), intent(out) Zz) [private]

Definition at line 339 of file sqrt_mod.f90.

```

339 REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: t,z
340 COMPLEX(KIND=16), DIMENSION(:, :), INTENT(OUT) :: tz,zz
341 INTEGER, PARAMETER :: nb=2
342 COMPLEX(KIND=16), DIMENSION(nb) :: w
343 !COMPLEX(KIND=16), DIMENSION(nb,nb) :: vl,vr
344 COMPLEX(KIND=16) :: r,c,s,mu
345 COMPLEX(KIND=16), DIMENSION(nb,nb) :: g,gc
346 INTEGER :: n,m!,info
347 !REAL(KIND=8), DIMENSION(2*nb) :: rwork
348 !REAL(KIND=8), DIMENSION(2*nb) :: ztwork
349
350 ! print*, lwork
351 tz=cplx(t,kind=16)
352 zz=cplx(z,kind=16)
353 n=SIZE(t,1)
354 DO m=n,2,-1
355     IF (abs(tz(m,m-1)) > real_eps*(abs(tz(m-1,m-1)) + abs(tz(m,m)))) THEN
356         g=tz(m-1:m,m-1:m)
357         ! CALL printmat(dble(G))
358         ! CALL zgeev('N','N',nb,G,nb,w,vl,nb,vr,nb,ztwork,2*nb,rwork,info)
359         ! CALL cprintmat(G)
360         ! print*, m,w,info
361         s=g(1,1)+g(2,2)
362         c=g(1,1)*g(2,2)-g(1,2)*g(2,1)
363         w(1)=s/2+sqrt(s**2/4-c)
364         mu=w(1)-tz(m,m)
365         r=sqrt(mu*conjg(mu)+tz(m,m-1)*conjg(tz(m,m-1)))
366         c=mu/r
367         s=tz(m,m-1)/r
368         g(1,1)=conjg(c)
369         g(1,2)=s
370         g(2,1)=-s
371         g(2,2)=c
372         gc=conjg(transpose(g))
373         tz(m-1:m,m-1:n)=matmul(g,tz(m-1:m,m-1:n))
374         tz(1:m,m-1:m)=matmul(tz(1:m,m-1:m),gc)
375         zz(:,m-1:m)=matmul(zz(:,m-1:m),gc)
376     END IF
377     tz(m,m-1)=cplx(0.d0,kind=16)
378 END DO

```

7.20.2.5 logical function `sqrt_mod::selectev (real(kind=8) a, real(kind=8) b)` [private]

Definition at line 122 of file `sqrt_mod.f90`.

```

122 REAL(KIND=8) :: a,b
123 LOGICAL selectev
124 selectev=.false.
125 ! IF (a>b) selectev=.true.
126 RETURN

```

7.20.2.6 subroutine, public `sqrt_mod::sqrtm (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) sqA, integer, intent(out) info, integer, intent(out) info_triu, integer, intent(in), optional bs)`

Routine to compute a real square-root of a matrix.

Parameters

<i>A</i>	Matrix whose square root to evaluate.
<i>sqA</i>	Square root of <i>A</i> .
<i>info</i>	Information code returned by the Lapack routines.
<i>info_triu</i>	Information code returned by the triangular matrix Lapack routines.
<i>bs</i>	Optional blocksize specification variable.

Definition at line 63 of file `sqrt_mod.f90`.

```

63     REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
64     REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: sqA
65     INTEGER, INTENT(IN), OPTIONAL :: bs
66     INTEGER, INTENT(OUT) :: info, info_triu
67     REAL(KIND=8), DIMENSION(SIZE(A,1), SIZE(A,1)) :: t, z, r
68     COMPLEX(KIND=16), DIMENSION(SIZE(A,1), SIZE(A,1)) :: tz, zz, rz
69     REAL(KIND=8), DIMENSION(SIZE(A,1)) :: wr, wi
70     LOGICAL, DIMENSION(SIZE(A,1)) :: bwork
71     LOGICAL :: selectev
72     INTEGER :: n
73     INTEGER :: sdim=0
74     n=SIZE(a,1)
75     t=a
76     ! print*, n, size(work,1)
77     CALL dgees('v','n',selectev,n,t,n,sdim,wr,wi,z,n,work,lwork,bwork,info)
78     ! print*, 'Z'
79     ! CALL printmat(Z)
80     ! print*, 'T'
81     ! CALL printmat(T)
82     ! CALL DGEES('V','N',SIZE(T,1),T,SIZE(T,1),0,wr,wi,Z,SIZE(Z,1),work,lwork,info)
83     ! print*, info
84     CALL triu(t,r)
85     IF (any(t /= r)) THEN
86         ! print*, 'T'
87         ! CALL printmat(T)
88         ! print*, 'Z'
89         ! CALL printmat(Z)
90         CALL rsf2csf(t,z,tz,zz)
91         ! print*, 'Tz'
92         ! CALL printmat(dble(Tz))
93         ! print*, 'iTz'
94         ! CALL printmat(dble(aimag(Tz)))
95         ! print*, 'Zz'
96         ! CALL printmat(dble(Zz))
97         ! print*, 'iZz'
98         ! CALL printmat(dble(aimag(Zz)))
99     IF (PRESENT(bs)) THEN
100         CALL csqrtm_triu(tz,rz,info_triu,bs)
101     ELSE
102         CALL csqrtm_triu(tz,rz,info_triu)
103     END IF
104     rz=matmul(zz,matmul(rz,conjg(transpose(zz))))
105     ! print*, 'sqAz'
106     ! CALL printmat(dble(Rz))
107     ! print*, 'isqAz'
108     ! CALL printmat(dble(aimag(Rz)))
109     sqA=dble(rz)
110 ELSE
111     IF (PRESENT(bs)) THEN
112         CALL sqrtm_triu(t,r,info_triu,bs)
113     ELSE
114         CALL sqrtm_triu(t,r,info_triu)
115     END IF
116     sqA=matmul(z,matmul(r,transpose(z)))
117 ENDIF
118

```

7.20.2.7 subroutine, public sqrt_mod::sqrtm_svd (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) sqA, integer, intent(out) info, integer, intent(out) info_triu, integer, intent(in), optional bs)

Routine to compute a real square-root of a matrix via a SVD decomposition.

Parameters

<i>A</i>	Matrix whose square root to evaluate.
<i>sqA</i>	Square root of A.
<i>info</i>	Information code returned by the Lapack routines.
<i>info_triu</i>	Not used (present for compatibility).
<i>bs</i>	Not used (present for compatibility).

Definition at line 401 of file sqrt_mod.f90.

```

401 REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
402 REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: sqA
403 INTEGER, INTENT(IN), OPTIONAL :: bs
404 INTEGER, INTENT(OUT) :: info, info_triu
405 REAL(KIND=8), DIMENSION(SIZE(A,1)) :: s
406 REAL(KIND=8), DIMENSION(SIZE(A,1), SIZE(A,1)) :: sq, u, vt
407 INTEGER :: i, n
408
409 sqA=a
410 n=SIZE(sqA,1)
411 CALL dgesvd('A', 'A', n, n, sqA, n, s, u, n, vt, n, work, lwork, info)
412 sq=0.d0
413 DO i=1, n
414     sq(i,i)=sqrt(s(i))
415 ENDDO
416 sqA=matmul(u, matmul(sq, vt))

```

7.20.2.8 subroutine sqrt_mod::sqrtm_triu (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) sqA, integer, intent(out) info, integer, intent(in), optional bs) [private]

Definition at line 131 of file sqrt_mod.f90.

```

131 REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
132 INTEGER, INTENT(IN), OPTIONAL :: bs
133 REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: sqA
134 INTEGER, INTENT(OUT) :: info
135 REAL(KIND=8), DIMENSION(SIZE(A,1)) :: a_diag
136 REAL(KIND=8), DIMENSION(SIZE(A,1), SIZE(A,1)) :: r, sm, rii, rjj
137 INTEGER, DIMENSION(2*SIZE(A,1), 2) :: start_stop_pairs
138 REAL(KIND=8) :: s, denom, scale
139 INTEGER :: i, j, k, start, n, sstop, m
140 INTEGER :: istart, istop, jstart, jstop
141 INTEGER :: nblocks, blocksize
142 INTEGER :: bsmall, blarge, nlarge, nsmall
143
144 blocksize=64
145 IF (PRESENT(bs)) blocksize=bs
146 n=SIZE(a,1)
147 ! print*, blocksize
148
149 CALL diag(a, a_diag)
150 r=0.d0
151 DO i=1, n
152     r(i,i)=sqrt(a_diag(i))
153 ENDDO
154
155
156 nblocks=max(floordiv(n, blocksize), 1)
157 bsmall=floordiv(n, nblocks)
158 nlarge=mod(n, nblocks)
159 blarge=bsmall+1
160 nsmall=nblocks-nlarge
161 IF (nsmall*bsmall + nlarge*blarge /= n) stop 'Sqrtm: Internal inconsistency'
162
163 ! print*, nblocks, bsmall, nsmall, blarge, nlarge
164
165 start=1
166 DO i=1, nsmall
167     start_stop_pairs(i,1)=start
168     start_stop_pairs(i,2)=start+bsmall-1
169     start=start+bsmall
170 ENDDO
171 DO i=nsmall+1, nsmall+nlarge
172     start_stop_pairs(i,1)=start
173     start_stop_pairs(i,2)=start+blarge-1
174     start=start+blarge
175 ENDDO
176
177 ! DO i=1, SIZE(start_stop_pairs,1)
178 !     print*, i
179 !     print*, start_stop_pairs(i,1), start_stop_pairs(i,2)
180 ! END DO
181
182 DO k=1, nsmall+nlarge
183     start=start_stop_pairs(k,1)
184     sstop=start_stop_pairs(k,2)
185     DO j=start, sstop
186         DO i=j-1, start, -1
187             s=0.d0

```

```

188         IF (j-i>1) s= dot_product(r(i,i+1:j-1),r(i+1:j-1,j))
189         denom= r(i,i)+r(j,j)
190         IF (denom==0.d0) stop 'Sqrtm: Failed to find the matrix square root'
191         r(i,j)=(a(i,j)-s)/denom
192     END DO
193 END DO
194 END DO
195
196 ! print*, 'R'
197 ! CALL printmat(R)
198
199 DO j=1,nblocks
200     jstart=start_stop_pairs(j,1)
201     jstop=start_stop_pairs(j,2)
202     DO i=j-1,1,-1
203         istart=start_stop_pairs(i,1)
204         istop=start_stop_pairs(i,2)
205         sm=0.d0
206         sm(istart:istop,jstart:jstop)=a(istart:istop,jstart:jstop)
207         IF (j-i>1) sm(istart:istop,jstart:jstop) = sm(istart:istop&
208             &,jstart:jstop) - matmul(r(istart:istop,istop:jstart)&
209                 &,r(istop:jstart,jstart:jstop))
210         rii=0.d0
211         rii = r(istart:istop, istart:istop)
212         rjj=0.d0
213         rjj = r(jstart:jstop, jstart:jstop)
214         m=istop-istart+1
215         n=jstop-jstart+1
216         k=1
217         ! print*, m,n
218         ! print*, istart,istop
219         ! print*, jstart,jstop
220
221         ! print*, 'Rii',Rii(istart:istop, istart:istop)
222         ! print*, 'Rjj',Rjj(jstart:jstop,jstart:jstop)
223         ! print*, 'Sm',Sm(istart:istop,jstart:jstop)
224
225         CALL dtrsyl('N','N',k,m,n,rii(istart:istop, istart:istop),m&
226             &,rjj(jstart:jstop,jstart:jstop),n,sm(istart:istop&
227                 &,jstart:jstop),m,scale,info)
228         r(istart:istop,jstart:jstop)=sm(istart:istop,jstart:jstop)*scale
229     ENDDO
230 ENDDO
231 sqa=r

```

7.20.3 Variable Documentation

7.20.3.1 integer sqrt_mod::lwork [private]

Definition at line 30 of file sqrt_mod.f90.

```
30  INTEGER :: lwork
```

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

Definition at line 32 of file sqrt_mod.f90.

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

7.20.3.3 real(kind=8), dimension(:), allocatable sqrt_mod::work [private]

Definition at line 27 of file sqrt_mod.f90.

```
27  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: work
```

7.21 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](#)

7.21.1 Detailed Description

Statistics accumulators.

Copyright

2015 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

7.21.2 Function/Subroutine Documentation

7.21.2.1 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)
```


7.21.2.2 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),
38      stat=allocstat)
39      IF (allocstat /= 0) stop '*** Not enough memory ***'
40      m=0.d0
41      mprev=0.d0
42      v=0.d0
43      mtmp=0.d0

```

7.21.2.3 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

```

7.21.2.4 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

```

7.21.2.5 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

```

7.21.2.6 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)

```

7.21.3 Variable Documentation

7.21.3.1 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
```

7.21.3.2 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
```

7.21.3.3 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
```

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

Definition at line 26 of file stat.f90.

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

7.21.3.5 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
```

7.22 stoch_mod Module Reference

Utility module containing the stochastic related routines.

Functions/Subroutines

- real(kind=8) function, public [gasdev](#) ()
- subroutine, public [stoch_vec](#) (dW)
Routine to fill a vector with standard Gaussian noise process values.
- subroutine, public [stoch_atm_vec](#) (dW)
routine to fill the atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_atm_res_vec](#) (dW)
routine to fill the resolved atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_atm_unres_vec](#) (dW)
routine to fill the unresolved atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_oc_vec](#) (dW)
routine to fill the oceanic component of a vector with standard gaussian noise process values
- subroutine, public [stoch_oc_res_vec](#) (dW)
routine to fill the resolved oceanic component of a vector with standard gaussian noise process values
- subroutine, public [stoch_oc_unres_vec](#) (dW)
routine to fill the unresolved oceanic component of a vector with standard gaussian noise process values

Variables

- integer [iset](#) =0
- real(kind=8) [gset](#)

7.22.1 Detailed Description

Utility module containing the stochastic related routines.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.22.2 Function/Subroutine Documentation

7.22.2.1 real(kind=8) function, public stoch_mod::gasdev ()

Definition at line 32 of file stoch_mod.f90.

```

32      REAL(KIND=8) :: gasdev
33      REAL(KIND=8) :: fac,rsq,v1,v2,r
34      IF (iset.eq.0) THEN
35          DO
36              CALL random_number(r)
37              v1=2.d0*r-1.
38              CALL random_number(r)
39              v2=2.d0*r-1.
40              rsq=v1**2+v2**2
41              IF (rsq.lt.1.d0.and.rsq.ne.0.d0) EXIT
42          ENDDO
43          fac=sqrt(-2.*log(rsq)/rsq)
44          gset=v1*fac
45          gasdev=v2*fac
46          iset=1
47      ELSE
48          gasdev=gset
49          iset=0
50      ENDIF
51      RETURN

```

7.22.2.2 subroutine, public stoch_mod::stoch_atm_res_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the resolved atmospheric component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 77 of file stoch_mod.f90.

```

77      real(kind=8), dimension(0:ndim), intent(inout) :: dw
78      integer :: i
79      dw=0.d0
80      do i=1,2*natm
81          IF (sf(i)==0) dw(i)=gasdev()
82      enddo

```

7.22.2.3 subroutine, public stoch_mod::stoch_atm_unres_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the unresolved atmospheric component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 88 of file stoch_mod.f90.

```

88      real(kind=8), dimension(0:ndim), intent(inout) :: dw
89      integer :: i
90      dw=0.d0
91      do i=1,2*natm
92          IF (sf(i)==1) dw(i)=gasdev()
93      enddo

```

7.22.2.4 subroutine, public stoch_mod::stoch_atm_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the atmospheric component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 67 of file stoch_mod.f90.

```

67      real(kind=8), dimension(0:ndim), intent(inout) :: dw
68      integer :: i
69      do i=1,2*natm
70          dw(i)=gasdev()
71      enddo

```

7.22.2.5 subroutine, public stoch_mod::stoch_oc_res_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the resolved oceanic component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 109 of file stoch_mod.f90.

```

109     real(kind=8), dimension(0:ndim), intent(inout) :: dw
110     integer :: i
111     dw=0.d0
112     do i=2*natm+1, ndim
113         IF (sf(i)==0) dw(i)=gasdev()
114     enddo

```

7.22.2.6 subroutine, public stoch_mod::stoch_oc_unres_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the unresolved oceanic component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 120 of file stoch_mod.f90.

```

120     real(kind=8), dimension(0:ndim), intent(inout) :: dw
121     integer :: i
122     dw=0.d0
123     do i=2*natm+1, ndim
124         IF (sf(i)==1) dw(i)=gasdev()
125     enddo

```

7.22.2.7 subroutine, public stoch_mod::stoch_oc_vec (real(kind=8), dimension(0:ndim), intent(inout) *dW*)

routine to fill the oceanic component of a vector with standard gaussian noise process values

Parameters

<i>dW</i>	vector to fill
-----------	----------------

Definition at line 99 of file stoch_mod.f90.

```

99     real(kind=8), dimension(0:ndim), intent(inout) :: dw
100     integer :: i
101     do i=2*natm+1, ndim
102         dw(i)=gasdev()
103     enddo

```

7.22.2.8 subroutine, public stoch_mod::stoch_vec (real(kind=8), dimension(0:ndim), intent(inout) dW)

Routine to fill a vector with standard Gaussian noise process values.

Parameters

dW	Vector to fill
------	----------------

Definition at line 57 of file stoch_mod.f90.

```

57      REAL(KIND=8), DIMENSION(0:ndim), INTENT(INOUT) :: dw
58      INTEGER :: i
59      DO i=1, ndim
60          dw(i)=gasdev()
61      ENDDO

```

7.22.3 Variable Documentation

7.22.3.1 real(kind=8) stoch_mod::gset [private]

Definition at line 24 of file stoch_mod.f90.

```

24      REAL(KIND=8) :: gset

```

7.22.3.2 integer stoch_mod::iset=0 [private]

Definition at line 23 of file stoch_mod.f90.

```

23      INTEGER :: iset=0

```

7.23 stoch_params Module Reference

The stochastic models parameters module.

Functions/Subroutines

- subroutine [init_stoch_params](#)
Stochastic parameters initialization routine.

Variables

- real(kind=8) [mnuti](#)
Multiplicative noise update time interval.
- real(kind=8) [t_trans_stoch](#)
Transient time period of the stochastic model evolution.
- real(kind=8) [q_ar](#)
Atmospheric resolved component noise amplitude.
- real(kind=8) [q_au](#)
Atmospheric unresolved component noise amplitude.
- real(kind=8) [q_or](#)
Oceanic resolved component noise amplitude.
- real(kind=8) [q_ou](#)
Oceanic unresolved component noise amplitude.
- real(kind=8) [dtn](#)
Square root of the timestep.
- real(kind=8) [eps_pert](#)
Perturbation parameter for the coupling.
- real(kind=8) [tdelta](#)
Time separation parameter.
- real(kind=8) [muti](#)
Memory update time interval.
- real(kind=8) [meml](#)
Time over which the memory kernel is integrated.
- real(kind=8) [t_trans_mem](#)
Transient time period to initialize the memory term.
- character(len=4) [x_int_mode](#)
Integration mode for the resolved component.
- real(kind=8) [dts](#)
Intrinsic resolved dynamics time step.
- integer [mems](#)
Number of steps in the memory kernel integral.
- real(kind=8) [dtsn](#)
Square root of the intrinsic resolved dynamics time step.
- real(kind=8) [maxint](#)
Upper integration limit of the correlations.
- character(len=4) [load_mode](#)
Loading mode for the correlations.
- character(len=4) [int_corr_mode](#)
Correlation integration mode.
- character(len=4) [mode](#)
Stochastic mode parameter.

7.23.1 Detailed Description

The stochastic models parameters module.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.23.2 Function/Subroutine Documentation

7.23.2.1 subroutine stoch_params::init_stoch_params ()

Stochastic parameters initialization routine.

Definition at line 58 of file stoch_params.f90.

```

58
59     namelist /mtvparams/ mnuti
60     namelist /stparams/ q_ar,q_au,q_or,q_ou,eps_pert,tdelta,t_trans_stoch
61     namelist /wlparams/ muti,meml,x_int_mode,dts,t_trans_mem
62     namelist /corr_init_mode/ load_mode,int_corr_mode,maxint
63     namelist /stoch_int_params/ mode
64
65
66     OPEN(8, file="stoch_params.nml", status='OLD', recl=80, delim='APOSTROPHE')
67     READ(8,nml=mtvparams)
68     READ(8,nml=wlparams)
69     READ(8,nml=stparams)
70     READ(8,nml=stoch_int_params)
71     READ(8,nml=corr_init_mode)
72     CLOSE(8)
73
74     dtn=sqrt(dt)
75     dtsn=sqrt(dts)
76     mems=ceiling(meml/muti)
77
78     q_au=q_au/tdelta
79     q_ou=q_ou/tdelta
80
```

7.23.3 Variable Documentation

7.23.3.1 real(kind=8) stoch_params::dtn

Square root of the timestep.

Definition at line 32 of file stoch_params.f90.

```

32     REAL(KIND=8) :: dtn                !< Square root of the timestep
```

7.23.3.2 real(kind=8) stoch_params::dts

Intrinsic resolved dynamics time step.

Definition at line 40 of file stoch_params.f90.

```

40     REAL(KIND=8) :: dts                !< Intrinsic resolved dynamics time step
```

7.23.3.3 real(kind=8) stoch_params::dtsn

Square root of the intrinsic resolved dynamics time step.

Definition at line 43 of file stoch_params.f90.

```

43     REAL(KIND=8) :: dtsn                !< Square root of the intrinsic resolved dynamics time step
```


7.23.3.4 real(kind=8) stoch_params::eps_pert

Perturbation parameter for the coupling.

Definition at line 33 of file stoch_params.f90.

```
33  REAL(KIND=8) :: eps_pert          !< Perturbation parameter for the coupling
```

7.23.3.5 character(len=4) stoch_params::int_corr_mode

Correlation integration mode.

Definition at line 47 of file stoch_params.f90.

```
47  CHARACTER(LEN=4) :: int_corr_mode !< Correlation integration mode
```

7.23.3.6 character(len=4) stoch_params::load_mode

Loading mode for the correlations.

Definition at line 46 of file stoch_params.f90.

```
46  CHARACTER(LEN=4) :: load_mode     !< Loading mode for the correlations
```

7.23.3.7 real(kind=8) stoch_params::maxint

Upper integration limit of the correlations.

Definition at line 45 of file stoch_params.f90.

```
45  REAL(KIND=8) :: maxint            !< Upper integration limit of the correlations
```

7.23.3.8 real(kind=8) stoch_params::meml

Time over which the memory kernel is integrated.

Definition at line 37 of file stoch_params.f90.

```
37  REAL(KIND=8) :: meml              !< Time over which the memory kernel is integrated
```

7.23.3.9 integer stoch_params::mems

Number of steps in the memory kernel integral.

Definition at line 42 of file stoch_params.f90.

```
42  INTEGER :: mems                                !< Number of steps in the memory kernel integral
```

7.23.3.10 real(kind=8) stoch_params::mnuti

Multiplicative noise update time interval.

Definition at line 25 of file stoch_params.f90.

```
25  REAL(KIND=8) :: mnuti                        !< Multiplicative noise update time interval
```

7.23.3.11 character(len=4) stoch_params::mode

Stochastic mode parameter.

Definition at line 49 of file stoch_params.f90.

```
49  CHARACTER(len=4) :: mode                     !< Stochastic mode parameter
```

7.23.3.12 real(kind=8) stoch_params::muti

Memory update time interval.

Definition at line 36 of file stoch_params.f90.

```
36  REAL(KIND=8) :: muti                        !< Memory update time interval
```

7.23.3.13 real(kind=8) stoch_params::q_ar

Atmospheric resolved component noise amplitude.

Definition at line 28 of file stoch_params.f90.

```
28  REAL(KIND=8) :: q_ar                        !< Atmospheric resolved component noise amplitude
```

7.23.3.14 real(kind=8) stoch_params::q_au

Atmospheric unresolved component noise amplitude.

Definition at line 29 of file stoch_params.f90.

```
29  REAL(KIND=8) :: q_au                !< Atmospheric unresolved component noise amplitude
```

7.23.3.15 real(kind=8) stoch_params::q_or

Oceanic resolved component noise amplitude.

Definition at line 30 of file stoch_params.f90.

```
30  REAL(KIND=8) :: q_or                !< Oceanic resolved component noise amplitude
```

7.23.3.16 real(kind=8) stoch_params::q_ou

Oceanic unresolved component noise amplitude.

Definition at line 31 of file stoch_params.f90.

```
31  REAL(KIND=8) :: q_ou                !< Oceanic unresolved component noise amplitude
```

7.23.3.17 real(kind=8) stoch_params::t_trans_mem

Transient time period to initialize the memory term.

Definition at line 38 of file stoch_params.f90.

```
38  REAL(KIND=8) :: t_trans_mem        !< Transient time period to initialize the memory term
```

7.23.3.18 real(kind=8) stoch_params::t_trans_stoch

Transient time period of the stochastic model evolution.

Definition at line 27 of file stoch_params.f90.

```
27  REAL(KIND=8) :: t_trans_stoch      !< Transient time period of the stochastic model evolution
```

7.23.3.19 `real(kind=8) stoch_params::tdelta`

Time separation parameter.

Definition at line 34 of file `stoch_params.f90`.

```
34  REAL(KIND=8) :: tdelta           !< Time separation parameter
```

7.23.3.20 `character(len=4) stoch_params::x_int_mode`

Integration mode for the resolved component.

Definition at line 39 of file `stoch_params.f90`.

```
39  CHARACTER(len=4) :: x_int_mode    !< Integration mode for the resolved component
```

7.24 tensor Module Reference

Tensor utility module.

Data Types

- type `coolist`
Coordinate list. Type used to represent the sparse tensor.
- type `coolist4`
4d coordinate list. Type used to represent the rank-4 sparse tensor.
- type `coolist_elem`
Coordinate list element type. Elementary elements of the sparse tensors.
- type `coolist_elem4`
4d coordinate list element type. Elementary elements of the 4d sparse tensors.

Functions/Subroutines

- subroutine, public `copy_tensor` (src, dst)
Routine to copy a rank-3 tensor.
- subroutine, public `add_to_tensor` (src, dst)
Routine to add a rank-3 tensor to another one.
- subroutine, public `add_matc_to_tensor` (i, src, dst)
Routine to add a matrix to a rank-3 tensor.
- subroutine, public `add_matc_to_tensor4` (i, j, src, dst)
Routine to add a matrix to a rank-4 tensor.
- subroutine, public `add_vec_jk_to_tensor` (j, k, src, dst)
Routine to add a vector to a rank-3 tensor.
- subroutine, public `add_vec_ikl_to_tensor4_perm` (i, k, l, src, dst)
Routine to add a vector to a rank-4 tensor plus permutation.
- subroutine, public `add_vec_ikl_to_tensor4` (i, k, l, src, dst)

Routine to add a vector to a rank-4 tensor.

- subroutine, public [add_vec_ijk_to_tensor4](#) (i, j, k, src, dst)

Routine to add a vector to a rank-4 tensor.

- subroutine, public [mat_to_coo](#) (src, dst)

Routine to convert a matrix to a rank-3 tensor.

- subroutine, public [tensor_to_coo](#) (src, dst)

Routine to convert a rank-3 tensor from matrix to coolist representation.

- subroutine, public [tensor4_to_coo4](#) (src, dst)

Routine to convert a rank-4 tensor from matrix to coolist representation.

- subroutine, public [print_tensor](#) (t)

Routine to print a rank 3 tensor coolist.

- subroutine, public [print_tensor4](#) (t)

Routine to print a rank-4 tensor coolist.

- subroutine, public [sparse_mul3](#) (coolist_ijk, arr_j, arr_k, res)

Sparse multiplication of a rank-3 tensor coolist with two vectors:
$$\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} a_j b_k.$$

- subroutine, public [sparse_mul3_mat](#) (coolist_ijk, arr_k, res)

Sparse multiplication of a rank-3 tensor coolist with a vector:
$$\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} b_k.$$
 Its output is a matrix.

- subroutine, public [sparse_mul4](#) (coolist_ijkl, arr_j, arr_k, arr_l, res)

Sparse multiplication of a rank-4 tensor coolist with three vectors:
$$\sum_{j,k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} a_j b_k c_l.$$

- subroutine, public [sparse_mul4_mat](#) (coolist_ijkl, arr_k, arr_l, res)

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

- subroutine, public [jsparse_mul](#) (coolist_ijk, arr_j, jcoo_ij)

Sparse multiplication of two tensors to determine the Jacobian:

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

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

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

This version return a coolist (sparse tensor).

- subroutine, public [jsparse_mul_mat](#) (coolist_ijk, arr_j, jcoo_ij)

Sparse multiplication of two tensors to determine the Jacobian:

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

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

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

This version return a matrix.

- subroutine, public [sparse_mul2_j](#) (coolist_ijk, arr_j, res)

Sparse multiplication of a 3d sparse tensor with a vectors:
$$\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k} a_j.$$

- subroutine, public [sparse_mul2_k](#) (coolist_ijk, arr_k, res)

Sparse multiplication of a rank-3 sparse tensor coolist with a vector:
$$\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} a_k.$$

- subroutine, public [simplify](#) (tensor)

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

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

- subroutine, public [coo_to_mat_ik](#) (src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix with i and k indices.

- subroutine, public [coo_to_mat_ij](#) (src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix with i and j indices.

- subroutine, public [coo_to_mat_i](#) (i, src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix.

- subroutine, public [coo_to_vec_jk](#) (j, k, src, dst)

Routine to convert a rank-3 tensor coolist component into a vector.

- subroutine, public [coo_to_mat_j](#) (j, src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix.

- subroutine, public [sparse_mul4_with_mat_jl](#) (coolist_ijkl, mat_jl, res)

Sparse multiplication of a rank-4 tensor coolist with a matrix : $\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{j,l}$.

- subroutine, public [sparse_mul4_with_mat_kl](#) (coolist_ijkl, mat_kl, res)

Sparse multiplication of a rank-4 tensor coolist with a matrix : $\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{k,l}$.

- subroutine, public [sparse_mul3_with_mat](#) (coolist_ijk, mat_jk, res)

Sparse multiplication of a rank-3 tensor coolist with a matrix: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} m_{j,k}$.

- subroutine, public [matc_to_coo](#) (src, dst)

Routine to convert a matrix to a rank-3 tensor.

- subroutine, public [scal_mul_coo](#) (s, t)

Routine to multiply a rank-3 tensor by a scalar.

- logical function, public [tensor_empty](#) (t)

Test if a rank-3 tensor coolist is empty.

- logical function, public [tensor4_empty](#) (t)

Test if a rank-4 tensor coolist is empty.

- subroutine, public [load_tensor4_from_file](#) (s, t)

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

- subroutine, public [write_tensor4_to_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.

7.24.1 Detailed Description

Tensor utility module.

Copyright

2015-2017 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

coolist is a type and also means "coordinate list"

7.24.2 Function/Subroutine Documentation

7.24.2.1 subroutine, public `tensor::add_matc_to_tensor (integer, intent(in) i, real(kind=8), dimension(ndim,ndim), intent(in) src, type(coolist), dimension(ndim), intent(inout) dst)`

Routine to add a matrix to a rank-3 tensor.

Parameters

<i>i</i>	Add to tensor component <i>i</i>
<i>src</i>	Matrix to add
<i>dst</i>	Destination tensor

Definition at line 144 of file `tensor.f90`.

```

144  INTEGER, INTENT(IN) :: i
145  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
146  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
147  TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
148  INTEGER :: j,k,r,n,nsrc,allocstat
149
150  nsrc=0
151  DO j=1,ndim
152    DO k=1,ndim
153      IF (abs(src(j,k))>real_eps) nsrc=nsrc+1
154    END DO
155  END DO
156
157  IF (dst(i)%nelems==0) THEN
158    IF (ALLOCATED(dst(i)%elems)) THEN
159      DEALLOCATE(dst(i)%elems, stat=allocstat)
160      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
161    ENDIF
162    ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
163    IF (allocstat /= 0) stop "*** Not enough memory ! ***"
164    n=0
165  ELSE
166    n=dst(i)%nelems
167    ALLOCATE(celems(n), stat=allocstat)
168    DO j=1,n
169      celems(j)%j=dst(i)%elems(j)%j
170      celems(j)%k=dst(i)%elems(j)%k
171      celems(j)%v=dst(i)%elems(j)%v
172    ENDDO
173    DEALLOCATE(dst(i)%elems, stat=allocstat)
174    IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
175    ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
176    IF (allocstat /= 0) stop "*** Not enough memory ! ***"
177    DO j=1,n
178      dst(i)%elems(j)%j=celems(j)%j
179      dst(i)%elems(j)%k=celems(j)%k
180      dst(i)%elems(j)%v=celems(j)%v
181    ENDDO
182    DEALLOCATE(celems, stat=allocstat)
183    IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
184  ENDIF
185  r=0
186  DO j=1,ndim
187    DO k=1,ndim
188      IF (abs(src(j,k))>real_eps) THEN
189        r=r+1
190        dst(i)%elems(n+r)%j=j
191        dst(i)%elems(n+r)%k=k
192        dst(i)%elems(n+r)%v=src(j,k)
193      ENDIF
194    ENDDO
195  END DO
196  dst(i)%nelems=nsrc+n
197

```

7.24.2.2 subroutine, public tensor::add_matc_to_tensor4 (integer, intent(in) *i*, integer, intent(in) *j*, real(kind=8),
dimension(ndim,ndim), intent(in) *src*, type(colist4), dimension(ndim), intent(inout) *dst*)

Routine to add a matrix to a rank-4 tensor.

Parameters

<i>i</i>	Add to tensor component i,j
<i>j</i>	Add to tensor component i,j
<i>src</i>	Matrix to add
<i>dst</i>	Destination tensor

Definition at line 206 of file tensor.f90.

```

206  INTEGER, INTENT(IN) :: i,j
207  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
208  TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
209  TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
210  INTEGER :: k,l,r,n,nsrc,allocstat
211
212  nsrc=0
213  DO k=1,ndim
214      DO l=1,ndim
215          IF (abs(src(k,l))>real_eps) nsrc=nsrc+1
216      END DO
217  END DO
218
219  IF (dst(i)%nelems==0) THEN
220      IF (ALLOCATED(dst(i)%elems)) THEN
221          DEALLOCATE(dst(i)%elems, stat=allocstat)
222          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
223      ENDIF
224      ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
225      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
226      n=0
227  ELSE
228      n=dst(i)%nelems
229      ALLOCATE(celems(n), stat=allocstat)
230      DO k=1,n
231          celems(k)%j=dst(i)%elems(k)%j
232          celems(k)%k=dst(i)%elems(k)%k
233          celems(k)%l=dst(i)%elems(k)%l
234          celems(k)%v=dst(i)%elems(k)%v
235      ENDDO
236      DEALLOCATE(dst(i)%elems, stat=allocstat)
237      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
238      ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
239      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
240      DO k=1,n
241          dst(i)%elems(k)%j=celems(k)%j
242          dst(i)%elems(k)%k=celems(k)%k
243          dst(i)%elems(k)%l=celems(k)%l
244          dst(i)%elems(k)%v=celems(k)%v
245      ENDDO
246      DEALLOCATE(celems, stat=allocstat)
247      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
248  ENDIF
249  r=0
250  DO k=1,ndim
251      DO l=1,ndim
252          IF (abs(src(k,l))>real_eps) THEN
253              r=r+1
254              dst(i)%elems(n+r)%j=j
255              dst(i)%elems(n+r)%k=k
256              dst(i)%elems(n+r)%l=l
257              dst(i)%elems(n+r)%v=src(k,l)
258          ENDIF
259      ENDDO
260  END DO
261  dst(i)%nelems=nsrc+n
262

```

7.24.2.3 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

<i>src</i>	Tensor to add
<i>dst</i>	Destination tensor

Definition at line 92 of file tensor.f90.

```

92     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
93     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
94     TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
95     INTEGER :: i,j,n,allocstat
96
97     DO i=1,ndim
98         IF (src(i)%nelems/=0) THEN
99             IF (dst(i)%nelems==0) THEN
100                 IF (ALLOCATED(dst(i)%elems)) THEN
101                     DEALLOCATE(dst(i)%elems, stat=allocstat)
102                     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
103                 ENDIF
104                 ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
105                 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
106                 n=0
107             ELSE
108                 n=dst(i)%nelems
109                 ALLOCATE(celems(n), stat=allocstat)
110                 DO j=1,n
111                     celems(j)%j=dst(i)%elems(j)%j
112                     celems(j)%k=dst(i)%elems(j)%k
113                     celems(j)%v=dst(i)%elems(j)%v
114                 ENDDO
115                 DEALLOCATE(dst(i)%elems, stat=allocstat)
116                 IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
117                 ALLOCATE(dst(i)%elems(src(i)%nelems+n), stat=allocstat)
118                 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
119                 DO j=1,n
120                     dst(i)%elems(j)%j=celems(j)%j
121                     dst(i)%elems(j)%k=celems(j)%k
122                     dst(i)%elems(j)%v=celems(j)%v
123                 ENDDO
124                 DEALLOCATE(celems, stat=allocstat)
125                 IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
126             ENDIF
127             DO j=1,src(i)%nelems
128                 dst(i)%elems(n+j)%j=src(i)%elems(j)%j
129                 dst(i)%elems(n+j)%k=src(i)%elems(j)%k
130                 dst(i)%elems(n+j)%v=src(i)%elems(j)%v
131             ENDDO
132             dst(i)%nelems=src(i)%nelems+n
133         ENDIF
134     ENDDO
135

```

7.24.2.4 subroutine, public `tensor::add_vec_ijk_to_tensor4` (integer, intent(in) *i*, integer, intent(in) *j*, integer, intent(in) *k*, real(kind=8), dimension(ndim), intent(in) *src*, type(coolist4), dimension(ndim), intent(inout) *dst*)

Routine to add a vector to a rank-4 tensor.

Parameters

<i>i,j,k</i>	Add to tensor component i,j and k
<i>src</i>	Vector to add
<i>dst</i>	Destination tensor

Definition at line 454 of file tensor.f90.

```

454     INTEGER, INTENT(IN) :: i,j,k

```

```

455 REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
456 TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
457 TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
458 INTEGER :: l,ne,r,n,nsrc,allocstat
459
460 nsrc=0
461 DO l=1,ndim
462   IF (abs(src(l))>real_eps) nsrc=nsrc+1
463 ENDDO
464
465 IF (dst(i)%nelems==0) THEN
466   IF (ALLOCATED(dst(i)%elems)) THEN
467     DEALLOCATE(dst(i)%elems, stat=allocstat)
468     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
469   ENDIF
470   ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
471   IF (allocstat /= 0) stop "*** Not enough memory ! ***"
472   n=0
473 ELSE
474   n=dst(i)%nelems
475   ALLOCATE(celems(n), stat=allocstat)
476   DO ne=1,n
477     celems(ne)%j=dst(i)%elems(ne)%j
478     celems(ne)%k=dst(i)%elems(ne)%k
479     celems(ne)%l=dst(i)%elems(ne)%l
480     celems(ne)%v=dst(i)%elems(ne)%v
481   ENDDO
482   DEALLOCATE(dst(i)%elems, stat=allocstat)
483   IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
484   ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
485   IF (allocstat /= 0) stop "*** Not enough memory ! ***"
486   DO ne=1,n
487     dst(i)%elems(ne)%j=celems(ne)%j
488     dst(i)%elems(ne)%k=celems(ne)%k
489     dst(i)%elems(ne)%l=celems(ne)%l
490     dst(i)%elems(ne)%v=celems(ne)%v
491   ENDDO
492   DEALLOCATE(celems, stat=allocstat)
493   IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
494 ENDIF
495 r=0
496 DO l=1,ndim
497   IF (abs(src(l))>real_eps) THEN
498     r=r+1
499     dst(i)%elems(n+r)%j=j
500     dst(i)%elems(n+r)%k=k
501     dst(i)%elems(n+r)%l=l
502     dst(i)%elems(n+r)%v=src(l)
503   ENDIF
504 ENDDO
505 dst(i)%nelems=nsrc+n

```

7.24.2.5 subroutine, public tensor::add_vec_ikl_to_tensor4 (integer, intent(in) i, integer, intent(in) k, integer, intent(in) l, real(kind=8), dimension(ndim), intent(in) src, type(coolist4), dimension(ndim), intent(inout) dst)

Routine to add a vector to a rank-4 tensor.

Parameters

<i>i,k,l</i>	Add to tensor component i,k and l
<i>src</i>	Vector to add
<i>dst</i>	Destination tensor

Definition at line 395 of file tensor.f90.

```

395 INTEGER, INTENT(IN) :: i,k,l
396 REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
397 TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
398 TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
399 INTEGER :: j,ne,r,n,nsrc,allocstat
400
401 nsrc=0
402 DO j=1,ndim

```

```

403         IF (abs(src(j))>real_eps) nsrc=nsrc+1
404     ENDDO
405
406     IF (dst(i)%nelems==0) THEN
407         IF (ALLOCATED(dst(i)%elems)) THEN
408             DEALLOCATE(dst(i)%elems, stat=allocstat)
409             IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
410         ENDIF
411         ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
412         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
413         n=0
414     ELSE
415         n=dst(i)%nelems
416         ALLOCATE(celems(n), stat=allocstat)
417         DO ne=1,n
418             celems(ne)%j=dst(i)%elems(ne)%j
419             celems(ne)%k=dst(i)%elems(ne)%k
420             celems(ne)%l=dst(i)%elems(ne)%l
421             celems(ne)%v=dst(i)%elems(ne)%v
422         ENDDO
423         DEALLOCATE(dst(i)%elems, stat=allocstat)
424         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
425         ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
426         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
427         DO ne=1,n
428             dst(i)%elems(ne)%j=celems(ne)%j
429             dst(i)%elems(ne)%k=celems(ne)%k
430             dst(i)%elems(ne)%l=celems(ne)%l
431             dst(i)%elems(ne)%v=celems(ne)%v
432         ENDDO
433         DEALLOCATE(celems, stat=allocstat)
434         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
435     ENDIF
436     r=0
437     DO j=1,ndim
438         IF (abs(src(j))>real_eps) THEN
439             r=r+1
440             dst(i)%elems(n+r)%j=j
441             dst(i)%elems(n+r)%k=k
442             dst(i)%elems(n+r)%l=l
443             dst(i)%elems(n+r)%v=src(j)
444         ENDIF
445     ENDDO
446     dst(i)%nelems=nsrc+n

```

7.24.2.6 subroutine, public `tensor::add_vec_ikl_to_tensor4_perm` (integer, intent(in) *i*, integer, intent(in) *k*, integer, intent(in) *l*, real(kind=8), dimension(ndim), intent(in) *src*, type(coolist4), dimension(ndim), intent(inout) *dst*)

Routine to add a vector to a rank-4 tensor plus permutation.

Parameters

<i>i,k,l</i>	Add to tensor component i,k and l
<i>src</i>	Vector to add
<i>dst</i>	Destination tensor

Definition at line 326 of file `tensor.f90`.

```

326     INTEGER, INTENT(IN) :: i,k,l
327     REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
328     TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
329     TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
330     INTEGER :: j,ne,r,n,nsrc,allocstat
331
332     nsrc=0
333     DO j=1,ndim
334         IF (abs(src(j))>real_eps) nsrc=nsrc+1
335     ENDDO
336     nsrc=nsrc*3
337     IF (dst(i)%nelems==0) THEN
338         IF (ALLOCATED(dst(i)%elems)) THEN
339             DEALLOCATE(dst(i)%elems, stat=allocstat)
340             IF (allocstat /= 0) stop "*** Deallocation problem ! ***"

```

```

341         ENDIF
342         ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
343         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
344         n=0
345     ELSE
346         n=dst(i)%elems
347         ALLOCATE(celems(n), stat=allocstat)
348         DO ne=1,n
349             celems(ne)%j=dst(i)%elems(ne)%j
350             celems(ne)%k=dst(i)%elems(ne)%k
351             celems(ne)%l=dst(i)%elems(ne)%l
352             celems(ne)%v=dst(i)%elems(ne)%v
353         ENDDO
354         DEALLOCATE(dst(i)%elems, stat=allocstat)
355         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
356         ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
357         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
358         DO ne=1,n
359             dst(i)%elems(ne)%j=celems(ne)%j
360             dst(i)%elems(ne)%k=celems(ne)%k
361             dst(i)%elems(ne)%l=celems(ne)%l
362             dst(i)%elems(ne)%v=celems(ne)%v
363         ENDDO
364         DEALLOCATE(celems, stat=allocstat)
365         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
366     ENDIF
367     r=0
368     DO j=1,ndim
369         IF (abs(src(j))>real_eps) THEN
370             r=r+1
371             dst(i)%elems(n+r)%j=j
372             dst(i)%elems(n+r)%k=k
373             dst(i)%elems(n+r)%l=l
374             dst(i)%elems(n+r)%v=src(j)
375             r=r+1
376             dst(i)%elems(n+r)%j=k
377             dst(i)%elems(n+r)%k=l
378             dst(i)%elems(n+r)%l=j
379             dst(i)%elems(n+r)%v=src(j)
380             r=r+1
381             dst(i)%elems(n+r)%j=l
382             dst(i)%elems(n+r)%k=j
383             dst(i)%elems(n+r)%l=k
384             dst(i)%elems(n+r)%v=src(j)
385         ENDIF
386     ENDDO
387     dst(i)%elems=nsrc+n

```

7.24.2.7 subroutine, public `tensor::add_vec_jk_to_tensor (integer, intent(in) j, integer, intent(in) k, real(kind=8), dimension(ndim), intent(in) src, type(coolist), dimension(ndim), intent(inout) dst)`

Routine to add a vector to a rank-3 tensor.

Parameters

<i>j,k</i>	Add to tensor component j and k
<i>src</i>	Vector to add
<i>dst</i>	Destination tensor

Definition at line 271 of file `tensor.f90`.

```

271     INTEGER, INTENT(IN) :: j,k
272     REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
273     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
274     TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
275     INTEGER :: i,l,r,n,nsrc,allocstat
276
277     DO i=1,ndim
278         nsrc=0
279         IF (abs(src(i))>real_eps) nsrc=1
280         IF (dst(i)%elems==0) THEN
281             IF (ALLOCATED(dst(i)%elems)) THEN
282                 DEALLOCATE(dst(i)%elems, stat=allocstat)

```

```

283         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
284     ENDIF
285     ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
286     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
287     n=0
288     ELSE
289         n=dst(i)%elems
290         ALLOCATE(celems(n), stat=allocstat)
291         DO l=1,n
292             celems(l)%j=dst(i)%elems(l)%j
293             celems(l)%k=dst(i)%elems(l)%k
294             celems(l)%v=dst(i)%elems(l)%v
295         ENDDO
296         DEALLOCATE(dst(i)%elems, stat=allocstat)
297         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
298         ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
299         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
300         DO l=1,n
301             dst(i)%elems(l)%j=celems(l)%j
302             dst(i)%elems(l)%k=celems(l)%k
303             dst(i)%elems(l)%v=celems(l)%v
304         ENDDO
305         DEALLOCATE(celems, stat=allocstat)
306         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
307     ENDIF
308     r=0
309     IF (abs(src(i))>real_eps) THEN
310         r=r+1
311         dst(i)%elems(n+r)%j=j
312         dst(i)%elems(n+r)%k=k
313         dst(i)%elems(n+r)%v=src(i)
314     ENDIF
315     dst(i)%elems=nsrc+n
316 END DO
317
318

```

7.24.2.8 subroutine, public tensor::coo_to_mat_i (integer, intent(in) *i*, type(coolist), dimension(ndim), intent(in) *src*, real(kind=8), dimension(ndim,ndim), intent(out) *dst*)

Routine to convert a rank-3 tensor coolist component into a matrix.

Parameters

<i>i</i>	Component to convert
<i>src</i>	Source tensor
<i>dst</i>	Destination matrix

Definition at line 971 of file tensor.f90.

```

971     INTEGER, INTENT(IN) :: i
972     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
973     REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
974     INTEGER :: n
975
976     dst=0.d0
977     DO n=1,src(i)%elems
978         dst(src(i)%elems(n)%j,src(i)%elems(n)%k)=src(i)%elems(n)%v
979     ENDDO

```

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

Routine to convert a rank-3 tensor coolist component into a matrix with i and j indices.

Parameters

<i>src</i>	Source tensor
<i>dst</i>	Destination matrix

Definition at line 938 of file tensor.f90.

```

938  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
939  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
940  INTEGER :: i,n
941
942  dst=0.d0
943  DO i=1,ndim
944      DO n=1,src(i)%elems
945          dst(i,src(i)%elems(n)%j)=src(i)%elems(n)%v
946      ENDDO
947  ENDDO

```

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

Routine to convert a rank-3 tensor coolist component into a matrix with i and k indices.

Parameters

<i>src</i>	Source tensor
<i>dst</i>	Destination matrix

Definition at line 922 of file tensor.f90.

```

922  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
923  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
924  INTEGER :: i,n
925
926  dst=0.d0
927  DO i=1,ndim
928      DO n=1,src(i)%elems
929          dst(i,src(i)%elems(n)%k)=src(i)%elems(n)%v
930      ENDDO
931  ENDDO

```

7.24.2.11 subroutine, public tensor::coo_to_mat_j (integer, intent(in) *j*, type(**coolist**), dimension(ndim), intent(in) *src*, real(kind=8), dimension(ndim,ndim), intent(out) *dst*)

Routine to convert a rank-3 tensor coolist component into a matrix.

Parameters

<i>j</i>	Component to convert
<i>src</i>	Source tensor
<i>dst</i>	Destination matrix

Definition at line 1007 of file tensor.f90.

```

1007    INTEGER, INTENT(IN) :: j
1008    TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1009    REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
1010    INTEGER :: i,n
1011
1012    dst=0.d0
1013    DO i=1,ndim
1014        DO n=1,src(i)%nelems
1015            IF (src(i)%elems(n)%j==j) dst(i,src(i)%elems(n)%k)=src(i)%elems(n)%v
1016        ENDDO
1017    END DO

```

7.24.2.12 subroutine, public `tensor::coo_to_vec_jk` (integer, intent(in) *j*, integer, intent(in) *k*, type(coolist), dimension(ndim), intent(in) *src*, real(kind=8), dimension(ndim), intent(out) *dst*)

Routine to convert a rank-3 tensor coolist component into a vector.

Parameters

<i>j</i>	Component j,k to convert
<i>k</i>	Component j,k to convert
<i>src</i>	Source tensor
<i>dst</i>	Destination vector

Definition at line 988 of file tensor.f90.

```

988    INTEGER, INTENT(IN) :: j,k
989    TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
990    REAL(KIND=8), DIMENSION(ndim), INTENT(OUT) :: dst
991    INTEGER :: i,n
992
993    dst=0.d0
994    DO i=1,ndim
995        DO n=1,src(i)%nelems
996            IF ((src(i)%elems(n)%j==j).and.(src(i)%elems(n)%k==k)) dst(i)=src(i)%elems(n)%v
997        END DO
998    ENDDO

```

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

Routine to copy a rank-3 tensor.

Parameters

<i>src</i>	Source tensor
<i>dst</i>	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 71 of file tensor.f90.

```

71    TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src

```



```

72     TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
73     INTEGER :: i,j,allocstat
74
75     DO i=1,ndim
76         IF (dst(i)%nelems/=0) stop "*** copy_tensor : Destination coolist not empty ! ***"
77         ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
78         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
79         DO j=1,src(i)%nelems
80             dst(i)%elems(j)%j=src(i)%elems(j)%j
81             dst(i)%elems(j)%k=src(i)%elems(j)%k
82             dst(i)%elems(j)%v=src(i)%elems(j)%v
83         ENDDO
84         dst(i)%nelems=src(i)%nelems
85     ENDDO

```

7.24.2.14 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:

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

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

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

This version return a coolist (sparse tensor).

Parameters

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

Definition at line 767 of file tensor.f90.

```

767     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
768     TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: jcoo_ij
769     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
770     REAL(KIND=8) :: v
771     INTEGER :: i,j,k,n,nj,allocstat
772     DO i=1,ndim
773         IF (jcoo_ij(i)%nelems/=0) stop "*** jsparse_mul : Destination coolist not empty ! ***"
774         nj=2*coolist_ijk(i)%nelems
775         ALLOCATE(jcoo_ij(i)%elems(nj), stat=allocstat)
776         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
777         nj=0
778         DO n=1,coolist_ijk(i)%nelems
779             j=coolist_ijk(i)%elems(n)%j
780             k=coolist_ijk(i)%elems(n)%k
781             v=coolist_ijk(i)%elems(n)%v
782             IF (j /= 0) THEN
783                 nj=nj+1
784                 jcoo_ij(i)%elems(nj)%j=j
785                 jcoo_ij(i)%elems(nj)%k=0
786                 jcoo_ij(i)%elems(nj)%v=v*arr_j(k)
787             END IF
788
789             IF (k /= 0) THEN
790                 nj=nj+1
791                 jcoo_ij(i)%elems(nj)%j=k
792                 jcoo_ij(i)%elems(nj)%k=0
793                 jcoo_ij(i)%elems(nj)%v=v*arr_j(j)
794             END IF
795         END DO
796         jcoo_ij(i)%nelems=nj
797     ENDDO

```

7.24.2.15 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:

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

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

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

This version return a matrix.

Parameters

<i>coolist_ijk</i>	a coordinate list (sparse tensor) of which index 2 or 3 will be contracted.
<i>arr_j</i>	the vector to be contracted with index 2 and then index 3 of ffi_coo_ijk
<i>jcoo_ij</i>	a matrix to store the result of the contraction

Definition at line 810 of file tensor.f90.

```

810  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
811  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: jcoo_ij
812  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
813  REAL(KIND=8) :: v
814  INTEGER :: i,j,k,n
815  jcoo_ij=0.d0
816  DO i=1,ndim
817      DO n=1,coolist_ijk(i)%elems
818          j=coolist_ijk(i)%elems(n)%j
819          k=coolist_ijk(i)%elems(n)%k
820          v=coolist_ijk(i)%elems(n)%v
821          IF (j /=0) jcoo_ij(i,j)=jcoo_ij(i,j)+v*arr_j(k)
822          IF (k /=0) jcoo_ij(i,k)=jcoo_ij(i,k)+v*arr_j(j)
823      END DO
824  END DO

```

7.24.2.16 subroutine, public tensor::load_tensor4_from_file (character(len=*) intent(in) s, type(coolist4), dimension(ndim), intent(out) t)

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

Parameters

<i>s</i>	Filename of the tensor definition file
<i>t</i>	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 1181 of file tensor.f90.

```

1181 CHARACTER (LEN=*) , INTENT(IN) :: s
1182 TYPE(coolist4) , DIMENSION(ndim) , INTENT(OUT) :: t
1183 INTEGER :: i,ir,j,k,l,n,allocstat
1184 REAL(KIND=8) :: v
1185 OPEN(30,file=s,status='old')
1186 DO i=1,ndim
1187   READ(30,*) ir,n
1188   IF (n /= 0) THEN
1189     ALLOCATE(t(i)%elems(n), stat=allocstat)
1190     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
1191     t(i)%elems=n
1192   ENDIF
1193   DO n=1,t(i)%elems
1194     READ(30,*) ir,j,k,l,v
1195     t(i)%elems(n)%j=j
1196     t(i)%elems(n)%k=k
1197     t(i)%elems(n)%l=l
1198     t(i)%elems(n)%v=v
1199   ENDDO
1200 END DO
1201 CLOSE(30)

```

7.24.2.17 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 rank-3 tensor.

Parameters

<i>src</i>	Source matrix
<i>dst</i>	Destination tensor

Remarks

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

Definition at line 515 of file tensor.f90.

```

515 REAL(KIND=8) , DIMENSION(0:ndim,0:ndim) , INTENT(IN) :: src
516 TYPE(coolist) , DIMENSION(ndim) , INTENT(OUT) :: dst
517 INTEGER :: i,j,n,allocstat
518 DO i=1,ndim
519   n=0
520   DO j=1,ndim
521     IF (abs(src(i,j))>real_eps) n=n+1
522   ENDDO
523   IF (n/=0) THEN
524     IF (dst(i)%elems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
525     ALLOCATE(dst(i)%elems(n), stat=allocstat)
526     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
527     n=0
528     DO j=1,ndim
529       IF (abs(src(i,j))>real_eps) THEN
530         n=n+1
531         dst(i)%elems(n)%j=j
532         dst(i)%elems(n)%k=0
533         dst(i)%elems(n)%v=src(i,j)
534       ENDIF
535     ENDDO
536   ENDIF
537   dst(i)%elems=n
538 ENDDO

```

7.24.2.18 subroutine, public tensor::matc_to_coo (real(kind=8), dimension(ndim,ndim), intent(in) src, type(coolist), dimension(ndim), intent(out) dst)

Routine to convert a matrix to a rank-3 tensor.

Parameters

<i>src</i>	Source matrix
<i>dst</i>	Destination tensor

Remarks

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

Definition at line 1103 of file tensor.f90.

```

1103  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
1104  TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
1105  INTEGER :: i,j,n,allocstat
1106  DO i=1,ndim
1107      n=0
1108      DO j=1,ndim
1109          IF (abs(src(i,j))>real_eps) n=n+1
1110      ENDDO
1111      IF (n/=0) THEN
1112          IF (dst(i)%nelems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
1113          ALLOCATE(dst(i)%elems(n), stat=allocstat)
1114          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
1115          n=0
1116          DO j=1,ndim
1117              IF (abs(src(i,j))>real_eps) THEN
1118                  n=n+1
1119                  dst(i)%elems(n)%j=0
1120                  dst(i)%elems(n)%k=j
1121                  dst(i)%elems(n)%v=src(i,j)
1122              ENDIF
1123          ENDDO
1124      ENDIF
1125      dst(i)%nelems=n
1126  ENDDO

```

7.24.2.19 subroutine, public tensor::print_tensor (type(coolist), dimension(ndim), intent(in) t)

Routine to print a rank 3 tensor coolist.

Parameters

<i>t</i>	coolist to print
----------	------------------

Definition at line 622 of file tensor.f90.

```

622  USE util, only: str
623  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: t
624  INTEGER :: i,n,j,k
625  DO i=1,ndim
626      DO n=1,t(i)%nelems
627          j=t(i)%elems(n)%j
628          k=t(i)%elems(n)%k
629          IF ( abs(t(i)%elems(n)%v) .GE. real_eps) THEN
630              write(*,"(A,ES12.5)") "tensor["//trim(str(i))//"]["//trim(str(j)) &
631                  &("//")["//trim(str(k))//"] = ",t(i)%elems(n)%v
632          END IF
633      END DO
634  END DO

```

7.24.2.20 subroutine, public tensor::print_tensor4 (type(coolist4), dimension(ndim), intent(in) t)

Routine to print a rank-4 tensor coolist.

Parameters

<i>t</i>	coolist to print
----------	------------------

Definition at line 640 of file tensor.f90.

```

640  USE util, only: str
641  TYPE(coolist4), DIMENSION(ndim), INTENT(IN) :: t
642  INTEGER :: i,n,j,k,l
643  DO i=1,ndim
644      DO n=1,t(i)%elems
645          j=t(i)%elems(n)%j
646          k=t(i)%elems(n)%k
647          l=t(i)%elems(n)%l
648          IF ( abs(t(i)%elems(n)%v) .GE. real_eps) THEN
649              write(*,"(A,ES12.5)") "tensor["//trim(str(i))//"]["//trim(str(j)) &
650                  &("//")["//trim(str(k))//"]["//trim(str(l))//"] = ",t(i)%elems(n)%v
651          END IF
652      END DO
653  END DO

```

7.24.2.21 subroutine, public tensor::scal_mul_coo (real(kind=8), intent(in) s, type(coolist), dimension(ndim), intent(inout) t)

Routine to multiply a rank-3 tensor by a scalar.

Parameters

<i>s</i>	The scalar
<i>t</i>	The tensor

Definition at line 1133 of file tensor.f90.

```

1133  REAL(KIND=8), INTENT(IN) :: s
1134  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
1135  INTEGER :: i,li,n
1136  DO i=1,ndim
1137      n=t(i)%elems
1138      DO li=1,n
1139          t(i)%elems(li)%v=s*t(i)%elems(li)%v
1140      ENDDO
1141  ENDDO

```

7.24.2.22 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

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

.

Parameters

<i>tensor</i>	a coordinate list (sparse tensor) which will be simplified.
---------------	---

Definition at line 874 of file tensor.f90.

```

874  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: tensor
875  INTEGER :: i,j,k
876  INTEGER :: li,lil,liil,n
877  DO i= 1,ndim
878      n=tensor(i)%elems
879      DO li=n,2,-1
880          j=tensor(i)%elems(li)%j
881          k=tensor(i)%elems(li)%k
882          DO lil=li-1,1,-1
883              IF ((j==tensor(i)%elems(lil)%j).AND.(k==tensor(i)%elems(lil)%k)) THEN
884                  ! Found another entry with the same i,j,k: merge both into
885                  ! the one listed first (of those two).
886                  tensor(i)%elems(lil)%v=tensor(i)%elems(lil)%v+tensor(i)%elems(li)%v
887                  ! Shift the rest of the items one place down.
888                  DO liil=lil+1,n
889                      tensor(i)%elems(liil-1)%j=tensor(i)%elems(liil)%j
890                      tensor(i)%elems(liil-1)%k=tensor(i)%elems(liil)%k
891                      tensor(i)%elems(liil-1)%v=tensor(i)%elems(liil)%v
892                  END DO
893                  tensor(i)%elems=tensor(i)%elems-1
894                  ! Here we should stop because the li no longer points to the
895                  ! original i,j,k element
896                  EXIT
897              ENDIF
898          ENDDO
899      ENDDO
900      n=tensor(i)%elems
901      DO li=1,n
902          ! Clear new "almost" zero entries and shift rest of the items one place down.
903          ! Make sure not to skip any entries while shifting!
904          DO WHILE (abs(tensor(i)%elems(li)%v) < real_eps)
905              DO liil=li+1,n
906                  tensor(i)%elems(liil-1)%j=tensor(i)%elems(liil)%j
907                  tensor(i)%elems(liil-1)%k=tensor(i)%elems(liil)%k
908                  tensor(i)%elems(liil-1)%v=tensor(i)%elems(liil)%v
909              ENDDO
910              tensor(i)%elems=tensor(i)%elems-1
911          ENDDO
912      ENDDO
913  ENDDO
914  ENDDO

```

7.24.2.23 subroutine, public `tensor::sparse_mul2_j` (`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(out) res`)

Sparse multiplication of a 3d sparse tensor with a vectors: $\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k} a_j$.

Parameters

<i>coolist_ijk</i>	a coordinate list (sparse tensor) of which index 2 will be contracted.
<i>arr_j</i>	the vector to be contracted with index 2 of <i>coolist_ijk</i>
<i>res</i>	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 835 of file tensor.f90.

```

835  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
836  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
837  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
838  INTEGER :: i,j,n
839  res=0.d0
840  DO i=1,ndim
841      DO n=1,coolist_ijk(i)%elems
842          j=coolist_ijk(i)%elems(n)%j
843          res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_j(j)
844      END DO
845  END DO

```

7.24.2.24 subroutine, public tensor::sparse_mul2_k (type(coolist), dimension(ndim), intent(in) coolist_ijk, real(kind=8), dimension(0:ndim), intent(in) arr_k, real(kind=8), dimension(0:ndim), intent(out) res)

Sparse multiplication of a rank-3 sparse tensor coolist with a vector: $\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} a_k$.

Parameters

<i>coolist_ijk</i>	a coordinate list (sparse tensor) of which index k will be contracted.
<i>arr_k</i>	the vector to be contracted with index k of coolist_ijk
<i>res</i>	vector (buffer) to store the result of the contraction

Remarks

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

Definition at line 856 of file tensor.f90.

```

856  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
857  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_k
858  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
859  INTEGER :: i,k,n
860  res=0.d0
861  DO i=1,ndim
862      DO n=1,coolist_ijk(i)%elems
863          k=coolist_ijk(i)%elems(n)%k
864          res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_k(k)
865      END DO
866  END DO

```

7.24.2.25 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 rank-3 tensor coolist with two vectors: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} a_j b_k$.

Parameters

<i>coolist_ijk</i>	a coolist (sparse tensor) of which index 2 and 3 will be contracted.
<i>arr_j</i>	the vector to be contracted with index 2 of coolist_ijk
<i>arr_k</i>	the vector to be contracted with index 3 of coolist_ijk
<i>res</i>	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 666 of file tensor.f90.

```

666  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
667  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j, arr_k
668  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
669  INTEGER :: i, j, k, n
670  res=0.d0
671  DO i=1, ndim
672      DO n=1, coolist_ijk(i)%elems
673          j=coolist_ijk(i)%elems(n)%j
674          k=coolist_ijk(i)%elems(n)%k
675          res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_j(j)*arr_k(k)
676      END DO
677  END DO

```

7.24.2.26 subroutine, public `tensor::sparse_mul3_mat (type(coolist), dimension(ndim), intent(in) coolist_ijk, real(kind=8), dimension(0:ndim), intent(in) arr_k, real(kind=8), dimension(ndim,ndim), intent(out) res)`

Sparse multiplication of a rank-3 tensor coolist with a vector: $\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} b_k$. Its output is a matrix.

Parameters

<i>coolist_ijk</i>	a coolist (sparse tensor) of which index k will be contracted.
<i>arr_k</i>	the vector to be contracted with index k of coolist_ijk
<i>res</i>	matrix (buffer) to store the result of the contraction

Remarks

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

Definition at line 689 of file tensor.f90.

```

689  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
690  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_k
691  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
692  INTEGER :: i, j, k, n
693  res=0.d0
694  DO i=1, ndim
695      DO n=1, coolist_ijk(i)%elems
696          j=coolist_ijk(i)%elems(n)%j
697          IF (j /= 0) THEN

```



```

698         k=coolist_ijk(i)%elems(n)%k
699         res(i,j) = res(i,j) + coolist_ijk(i)%elems(n)%v * arr_k(k)
700     ENDIF
701 END DO
702 END DO

```

7.24.2.27 subroutine, public tensor::sparse_mul3_with_mat (type(coolist), dimension(ndim), intent(in) *coolist_ijk*, real(kind=8), dimension(ndim,ndim), intent(in) *mat_jk*, real(kind=8), dimension(0:ndim), intent(out) *res*)

Sparse multiplication of a rank-3 tensor coolist with a matrix: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} m_{j,k}$.

Parameters

<i>coolist_ijk</i>	a coolist (sparse tensor) of which index j and k will be contracted.
<i>mat_jk</i>	the matrix to be contracted with index j and k of coolist_ijk
<i>res</i>	vector (buffer) to store the result of the contraction

Remarks

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

Definition at line 1079 of file tensor.f90.

```

1079  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
1080  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: mat_jk
1081  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
1082  INTEGER i,j,k,n
1083
1084  res=0.d0
1085  DO i=1,ndim
1086      DO n=1,coolist_ijk(i)%elems
1087          j=coolist_ijk(i)%elems(n)%j
1088          k=coolist_ijk(i)%elems(n)%k
1089
1090          res(i) = res(i) + coolist_ijk(i)%elems(n)%v * mat_jk(j,k)
1091      ENDDO
1092  END DO
1093

```

7.24.2.28 subroutine, public tensor::sparse_mul4 (type(coolist4), dimension(ndim), intent(in) *coolist_ijkl*, 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(in) *arr_l*, real(kind=8), dimension(0:ndim), intent(out) *res*)

Sparse multiplication of a rank-4 tensor coolist with three vectors: $\sum_{j,k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} a_j b_k c_l$.

Parameters

<i>coolist_ijkl</i>	a coolist (sparse tensor) of which index j, k and l will be contracted.
<i>arr_j</i>	the vector to be contracted with index j of coolist_ijkl
<i>arr_k</i>	the vector to be contracted with index k of coolist_ijkl
<i>arr_l</i>	the vector to be contracted with index l of coolist_ijkl
<i>res</i>	vector (buffer) to store the result of the contraction

Remarks

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

Definition at line 715 of file `tensor.f90`.

```

715  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
716  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j, arr_k, arr_l
717  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
718  INTEGER :: i, j, k, n, l
719  res=0.d0
720  DO i=1,ndim
721      DO n=1,coolist_ijkl(i)%elems
722          j=coolist_ijkl(i)%elems(n)%j
723          k=coolist_ijkl(i)%elems(n)%k
724          l=coolist_ijkl(i)%elems(n)%l
725          res(i) = res(i) + coolist_ijkl(i)%elems(n)%v * arr_j(j)*arr_k(k)*arr_l(l)
726      END DO
727  END DO

```

7.24.2.29 subroutine, public `tensor::sparse_mul4_mat (type(coolist4), dimension(ndim), intent(in) coolist_ijkl, real(kind=8), dimension(0:ndim), intent(in) arr_k, real(kind=8), dimension(0:ndim), intent(in) arr_l, real(kind=8), dimension(ndim,ndim), intent(out) res)`

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

Parameters

<i>coolist_ijkl</i>	a coordinate list (sparse tensor) of which index 3 and 4 will be contracted.
<i>arr_k</i>	the vector to be contracted with index 3 of <i>coolist_ijkl</i>
<i>arr_l</i>	the vector to be contracted with index 4 of <i>coolist_ijkl</i>
<i>res</i>	matrix (buffer) to store the result of the contraction

Remarks

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

Definition at line 739 of file `tensor.f90`.

```

739  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
740  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_k, arr_l
741  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
742  INTEGER :: i, j, k, n, l
743  res=0.d0
744  DO i=1,ndim
745      DO n=1,coolist_ijkl(i)%elems
746          j=coolist_ijkl(i)%elems(n)%j
747          IF (j /= 0) THEN
748              k=coolist_ijkl(i)%elems(n)%k
749              l=coolist_ijkl(i)%elems(n)%l
750              res(i,j) = res(i,j) + coolist_ijkl(i)%elems(n)%v * arr_k(k) * arr_l(l)
751          ENDIF
752      END DO
753  END DO

```

7.24.2.30 subroutine, public tensor::sparse_mul4_with_mat_jl (type(coolist4), dimension(ndim), intent(in) *coolist_ijkl*, real(kind=8), dimension(ndim,ndim), intent(in) *mat_jl*, real(kind=8), dimension(ndim,ndim), intent(out) *res*)

Sparse multiplication of a rank-4 tensor coolist with a matrix :
$$\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{j,l}.$$

Parameters

<i>coolist_ijkl</i>	a coolist (sparse tensor) of which index j and l will be contracted.
<i>mat_jl</i>	the matrix to be contracted with indices j and l of coolist_ijkl
<i>res</i>	matrix (buffer) to store the result of the contraction

Remarks

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

Definition at line 1028 of file tensor.f90.

```

1028  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
1029  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: mat_jl
1030  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
1031  INTEGER i,j,k,l,n
1032
1033  res=0.d0
1034  DO i=1,ndim
1035      DO n=1,coolist_ijkl(i)%elems
1036          j=coolist_ijkl(i)%elems(n)%j
1037          k=coolist_ijkl(i)%elems(n)%k
1038          l=coolist_ijkl(i)%elems(n)%l
1039
1040          res(i,k) = res(i,k) + coolist_ijkl(i)%elems(n)%v * mat_jl(j,l)
1041      ENDDO
1042  END DO
1043
```

7.24.2.31 subroutine, public tensor::sparse_mul4_with_mat_kl (type(coolist4), dimension(ndim), intent(in) *coolist_ijkl*, real(kind=8), dimension(ndim,ndim), intent(in) *mat_kl*, real(kind=8), dimension(ndim,ndim), intent(out) *res*)

Sparse multiplication of a rank-4 tensor coolist with a matrix :
$$\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{k,l}.$$

Parameters

<i>coolist_ijkl</i>	a coolist (sparse tensor) of which index k and l will be contracted.
<i>mat_kl</i>	the matrix to be contracted with indices k and l of coolist_ijkl
<i>res</i>	matrix (buffer) to store the result of the contraction

Remarks

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

Definition at line 1053 of file tensor.f90.

```

1053  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
1054  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN)  :: mat_kl
1055  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
1056  INTEGER i,j,k,l,n
1057
1058  res=0.d0
1059  DO i=1,ndim
1060      DO n=1,coolist_ijkl(i)%nelems
1061          j=coolist_ijkl(i)%elems(n)%j
1062          k=coolist_ijkl(i)%elems(n)%k
1063          l=coolist_ijkl(i)%elems(n)%l
1064
1065          res(i,j) = res(i,j) + coolist_ijkl(i)%elems(n)%v * mat_kl(k,l)
1066      ENDDO
1067  END DO
1068

```

7.24.2.32 logical function, public tensor::tensor4_empty (type(coolist4), dimension(ndim), intent(in) t)

Test if a rank-4 tensor coolist is empty.

Parameters

<i>t</i>	rank-4 tensor coolist to be tested
----------	------------------------------------

Definition at line 1163 of file tensor.f90.

```

1163  TYPE(coolist4), DIMENSION(ndim), INTENT(IN) :: t
1164  LOGICAL :: tensor4_empty
1165  INTEGER :: i
1166  tensor4_empty=.true.
1167  DO i=1,ndim
1168      IF (t(i)%nelems /= 0) THEN
1169          tensor4_empty=.false.
1170          RETURN
1171      ENDIF
1172  END DO
1173  RETURN

```

7.24.2.33 subroutine, public tensor::tensor4_to_coo4 (real(kind=8), dimension(ndim,0:ndim,0:ndim,0:ndim), intent(in) src, type(coolist4), dimension(ndim), intent(out) dst)

Routine to convert a rank-4 tensor from matrix to coolist representation.

Parameters

<i>src</i>	Source matrix
<i>dst</i>	Destination coolist

Remarks

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

Definition at line 583 of file tensor.f90.

```

583  REAL(KIND=8), DIMENSION(ndim,0:ndim,0:ndim,0:ndim), INTENT(IN) :: src
584  TYPE(coolist4), DIMENSION(ndim), INTENT(OUT) :: dst

```

```

585     INTEGER :: i, j, k, l, n, allocstat
586
587     DO i=1, ndim
588         n=0
589         DO j=0, ndim
590             DO k=0, ndim
591                 DO l=0, ndim
592                     IF (abs(src(i, j, k, l)) > real_eps) n=n+1
593                 ENDDO
594             ENDDO
595         ENDDO
596         IF (n/=0) THEN
597             IF (dst(i)%elems/=0) stop "*** tensor_to_coo : Destination coolist not empty ! ***"
598             ALLOCATE(dst(i)%elems(n), stat=allocstat)
599             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
600             n=0
601             DO j=0, ndim
602                 DO k=0, ndim
603                     DO l=0, ndim
604                         IF (abs(src(i, j, k, l)) > real_eps) THEN
605                             n=n+1
606                             dst(i)%elems(n)%j=j
607                             dst(i)%elems(n)%k=k
608                             dst(i)%elems(n)%l=l
609                             dst(i)%elems(n)%v=src(i, j, k, l)
610                         ENDDO
611                     ENDDO
612                 ENDDO
613             ENDDO
614         ENDDO
615         dst(i)%elems=n
616     ENDDO

```

7.24.2.34 logical function, public tensor::tensor_empty (type(coolist), dimension(ndim), intent(in) t)

Test if a rank-3 tensor coolist is empty.

Parameters

<i>t</i>	rank-3 tensor coolist to be tested
----------	------------------------------------

Definition at line 1147 of file tensor.f90.

```

1147     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: t
1148     LOGICAL :: tensor_empty
1149     INTEGER :: i
1150     tensor_empty=.true.
1151     DO i=1, ndim
1152         IF (t(i)%elems /= 0) THEN
1153             tensor_empty=.false.
1154         RETURN
1155     ENDDO
1156     END DO
1157     RETURN

```

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

Routine to convert a rank-3 tensor from matrix to coolist representation.

Parameters

<i>src</i>	Source matrix
<i>dst</i>	Destination coolist

Remarks

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

Definition at line 547 of file tensor.f90.

```

547 REAL(KIND=8), DIMENSION(ndim,0:ndim,0:ndim), INTENT(IN) :: src
548 TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
549 INTEGER :: i,j,k,n,allocstat
550
551 DO i=1,ndim
552     n=0
553     DO j=0,ndim
554         DO k=0,ndim
555             IF (abs(src(i,j,k))>real_eps) n=n+1
556         ENDDO
557     ENDDO
558     IF (n/=0) THEN
559         IF (dst(i)%nelems/=0) stop "*** tensor_to_coo : Destination coolist not empty ! ***"
560         ALLOCATE(dst(i)%elems(n), stat=allocstat)
561         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
562         n=0
563         DO j=0,ndim
564             DO k=0,ndim
565                 IF (abs(src(i,j,k))>real_eps) THEN
566                     n=n+1
567                     dst(i)%elems(n)%j=j
568                     dst(i)%elems(n)%k=k
569                     dst(i)%elems(n)%v=src(i,j,k)
570                 ENDIF
571             ENDDO
572         ENDDO
573     ENDIF
574     dst(i)%nelems=n
575 ENDDO

```

7.24.2.36 subroutine, public tensor::write_tensor4_to_file (character (len=*) intent(in) s, type(coolist4), 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 1208 of file tensor.f90.

```

1208 CHARACTER (LEN=*) , INTENT(IN) :: s
1209 TYPE(coolist4), DIMENSION(ndim), INTENT(IN) :: t
1210 INTEGER :: i,j,k,l,n
1211 OPEN(30,file=s)
1212 DO i=1,ndim
1213     WRITE(30,*) i,t(i)%nelems
1214     DO n=1,t(i)%nelems
1215         j=t(i)%elems(n)%j
1216         k=t(i)%elems(n)%k
1217         l=t(i)%elems(n)%l
1218         WRITE(30,*) i,j,k,l,t(i)%elems(n)%v
1219     END DO
1220 END DO
1221 CLOSE(30)

```

7.24.3 Variable Documentation

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

Parameter to test the equality with zero.

Definition at line 50 of file tensor.f90.

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

7.25 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.

7.25.1 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 buffers will probably have to be defined.

7.25.2 Function/Subroutine Documentation

7.25.2.1 `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

<i>y</i>	Initial point.
<i>ystar</i>	Adjoint model at the point ystar.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>res</i>	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

```

7.25.2.2 `subroutine public tl_ad_integrator::init_tl_ad_integrator ()`

Routine to initialise the integration buffers.

Routine to initialise the TL-AD integration buffers.

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 ! ***"

```

7.25.2.3 `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

<i>y</i>	Initial point.
<i>ystar</i>	Adjoint model at the point ystar.
<i>t</i>	Actual integration time
<i>dt</i>	Integration timestep.
<i>res</i>	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

```

7.25.3 Variable Documentation

7.25.3.1 `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

```

7.25.3.2 `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

```

7.25.3.3 `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

```

7.25.3.4 `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
```

7.25.3.5 `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
```

7.26 `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_tlensor](#)
Routine to initialize the TL tensor.
- subroutine [compute_tlensor](#) (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, deltat, buf)
Tendencies for the AD of MAOOAM in point ystar for perturbation deltat.
- subroutine, public [tl](#) (t, ystar, deltat, buf)
Tendencies for the TL of MAOOAM in point ystar for perturbation deltat.

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

7.26.1 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 !

7.26.2 Function/Subroutine Documentation

7.26.2.1 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

<i>t</i>	time
<i>ystar</i>	vector with the variables (current point in trajectory)
<i>deltay</i>	vector with the perturbation of the variables at time <i>t</i>
<i>buf</i>	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)
```

7.26.2.2 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

```

7.26.2.3 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

```

7.26.2.4 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

```

```

260     IF (.NOT. ALLOCATED(adtensor)) stop "*** ad_coeff routine : tensor not yet allocated ***"
261     IF ((abs(v) .ge. real_eps).AND.(i /=0)) THEN
262         IF (k /=0) THEN
263             IF (.NOT. ALLOCATED(adtensor(k)%elems)) stop "*** ad_coeff routine : tensor not yet allocated
                ***"
264             n=(adtensor(k)%elems)+1
265             adtensor(k)%elems(n)%j=i
266             adtensor(k)%elems(n)%k=j
267             adtensor(k)%elems(n)%v=v
268             adtensor(k)%elems=n
269         END IF
270         IF (j /=0) THEN
271             IF (.NOT. ALLOCATED(adtensor(j)%elems)) stop "*** ad_coeff routine : tensor not yet allocated
                ***"
272             n=(adtensor(j)%elems)+1
273             adtensor(j)%elems(n)%j=i
274             adtensor(j)%elems(n)%k=k
275             adtensor(j)%elems(n)%v=v
276             adtensor(j)%elems=n
277         END IF
278     END IF

```

7.26.2.5 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>i</i>	tensor <i>i</i> index
<i>j</i>	tensor <i>j</i> index
<i>k</i>	tensor <i>k</i> index
<i>v</i>	value to add

Definition at line 358 of file `tl_ad_tensor.f90`.

```

358     INTEGER, INTENT(IN) :: i,j,k
359     REAL(KIND=8), INTENT(IN) :: v
360     INTEGER :: n
361     IF (.NOT. ALLOCATED(adtensor)) stop "*** ad_coeff_ref routine : tensor not yet allocated ***"
362     IF ((abs(v) .ge. real_eps).AND.(j /=0)) THEN
363         IF (.NOT. ALLOCATED(adtensor(j)%elems)) stop "*** ad_coeff_ref routine : tensor not yet allocated
                ***"
364         n=(adtensor(j)%elems)+1
365         adtensor(j)%elems(n)%j=i
366         adtensor(j)%elems(n)%k=k
367         adtensor(j)%elems(n)%v=v
368         adtensor(j)%elems=n
369     END IF

```

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

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

Parameters

<i>func</i>	subroutine used to do the computation
-------------	---------------------------------------

Definition at line 217 of file `tl_ad_tensor.f90`.

7.26.2.7 subroutine `tl_ad_tensor::compute_adtensor_ref (external func)` [private]

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

Parameters

<i>func</i>	subroutine used to do the computation
-------------	---------------------------------------

Definition at line 318 of file `tl_ad_tensor.f90`.

7.26.2.8 subroutine `tl_ad_tensor::compute_tltensor (external func)` [private]

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

Parameters

<i>func</i>	subroutine used to do the computation
-------------	---------------------------------------

Definition at line 121 of file `tl_ad_tensor.f90`.

7.26.2.9 subroutine, public `tl_ad_tensor::init_adtensor ()`

Routine to initialize the AD tensor.

Definition at line 193 of file `tl_ad_tensor.f90`.

```

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

7.26.2.10 subroutine, public `tl_ad_tensor::init_adtensor_ref ()`

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

Remarks

The `tlensor` 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

```

7.26.2.11 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

```

7.26.2.12 `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

<code>ystar</code>	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`.

```

75     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: ystar
76     TYPE(colist), DIMENSION(ndim) :: jacobian
77     CALL jsparse_mul(aotensor, ystar, jacobian)

```

7.26.2.13 `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

<i>ystar</i>	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`.

```

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

```

7.26.2.14 `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

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

Definition at line 396 of file `tl_ad_tensor.f90`.

```

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

```


7.26.2.15 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>i</i>	tensor <i>i</i> index
<i>j</i>	tensor <i>j</i> index
<i>k</i>	tensor <i>k</i> index
<i>v</i>	value that will be added

Definition at line 147 of file `tl_ad_tensor.f90`.

```

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

```

7.26.2.16 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>i</i>	tensor <i>i</i> index
<i>j</i>	tensor <i>j</i> index
<i>k</i>	tensor <i>k</i> index
<i>v</i>	value to add

Definition at line 161 of file `tl_ad_tensor.f90`.

```

161  INTEGER, INTENT(IN) :: i,j,k
162  REAL(KIND=8), INTENT(IN) :: v
163  INTEGER :: n
164  IF (.NOT. ALLOCATED(tl_tensor)) stop "*** tl_coeff routine : tensor not yet allocated ***"
165  IF (.NOT. ALLOCATED(tl_tensor(i)%elems)) stop "*** tl_coeff routine : tensor not yet allocated ***"
166  IF (abs(v) .ge. real_eps) THEN
167      IF (j /= 0) THEN
168          n=(tl_tensor(i)%elems)+1
169          tl_tensor(i)%elems(n)%j=j
170          tl_tensor(i)%elems(n)%k=k
171          tl_tensor(i)%elems(n)%v=v
172          tl_tensor(i)%elems=n
173      END IF
174      IF (k /= 0) THEN
175          n=(tl_tensor(i)%elems)+1
176          tl_tensor(i)%elems(n)%j=k
177          tl_tensor(i)%elems(n)%k=j
178          tl_tensor(i)%elems(n)%v=v
179          tl_tensor(i)%elems=n
180      END IF
181  END IF

```

7.26.3 Variable Documentation

7.26.3.1 `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
```

7.26.3.2 `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
```

7.26.3.3 `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
```

7.26.3.4 `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
```

7.27 util Module Reference

Utility module.

Functions/Subroutines

- `character(len=20)` function, public `str` (k)
Convert an integer to string.
- `character(len=40)` function, public `rstr` (x, fm)
Convert a real to string with a given format.
- subroutine, public `init_random_seed` ()
Random generator initialization routine.
- subroutine, public `init_one` (A)
Initialize a square matrix A as a unit matrix.
- `real(kind=8)` function, public `mat_trace` (A)
- `real(kind=8)` function, public `mat_contract` (A, B)
- subroutine, public `choldc` (a, p)
- subroutine, public `printmat` (A)
- subroutine, public `cprintmat` (A)
- `real(kind=8)` function, `dimension(size(a, 1), size(a, 2))`, public `invmat` (A)
- subroutine, public `triu` (A, T)
- subroutine, public `diag` (A, d)
- subroutine, public `cdiag` (A, d)
- integer function, public `floordiv` (i, j)
- subroutine, public `reduce` (A, Ared, n, ind, rind)
- subroutine, public `ireduce` (A, Ared, n, ind, rind)
- subroutine, public `vector_outer` (u, v, A)

7.27.1 Detailed Description

Utility module.

Copyright

2015 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

7.27.2 Function/Subroutine Documentation

7.27.2.1 subroutine, public `util::cdiag` (`complex(kind=16)`, `dimension(:, :)`, `intent(in)` A, `complex(kind=16)`, `dimension(:, :)`, `intent(out)` d)

Definition at line 230 of file `util.f90`.

```

230  COMPLEX(KIND=16), DIMENSION(:, :), INTENT(IN) :: a
231  COMPLEX(KIND=16), DIMENSION(:, :), INTENT(OUT) :: d
232  INTEGER :: i
233
234  DO i=1, SIZE(a, 1)
235      d(i)=a(i, i)
236  END DO

```

7.27.2.2 subroutine, public util::choldc (real(kind=8), dimension(:, :) a, real(kind=8), dimension(:) p)

Definition at line 137 of file util.f90.

```

137  REAL(KIND=8), DIMENSION(:, :) :: a
138  REAL(KIND=8), DIMENSION(:) :: p
139  INTEGER :: n
140  INTEGER :: i, j, k
141  REAL(KIND=8) :: sum
142  n=SIZE(a, 1)
143  DO i=1, n
144      DO j=i, n
145          sum=a(i, j)
146          DO k=i-1, 1, -1
147              sum=sum-a(i, k)*a(j, k)
148          END DO
149          IF (i.eq.j) THEN
150              IF (sum.le.0.) stop 'choldc failed'
151              p(i)=sqrt(sum)
152          ELSE
153              a(j, i)=sum/p(i)
154          ENDIF
155      END DO
156  END DO
157  RETURN

```

7.27.2.3 subroutine, public util::cprintmat (complex(kind=16), dimension(:, :), intent(in) A)

Definition at line 170 of file util.f90.

```

170  COMPLEX(KIND=16), DIMENSION(:, :), INTENT(IN) :: a
171  INTEGER :: i
172
173  DO i=1, SIZE(a, 1)
174      print*, a(i, :)
175  END DO

```

7.27.2.4 subroutine, public util::diag (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:), intent(out) d)

Definition at line 220 of file util.f90.

```

220  REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
221  REAL(KIND=8), DIMENSION(:), INTENT(OUT) :: d
222  INTEGER :: i
223
224  DO i=1, SIZE(a, 1)
225      d(i)=a(i, i)
226  END DO

```

7.27.2.5 integer function, public util::floordiv (integer i, integer j)

Definition at line 241 of file util.f90.

```

241  INTEGER :: i, j, floordiv
242  floordiv=int(floor(real(i)/real(j)))
243  RETURN

```

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

Initialize a square matrix A as a unit matrix.

Definition at line 100 of file util.f90.

```

100      REAL(KIND=8), DIMENSION(:,,:), INTENT(INOUT) :: a
101      INTEGER :: i, n
102      n=size(a,1)
103      a=0.0d0
104      DO i=1,n
105          a(i,i)=1.0d0
106      END DO
107

```

7.27.2.7 subroutine, public util::init_random_seed ()

Random generator initialization routine.

Definition at line 45 of file util.f90.

7.27.2.8 real(kind=8) function, dimension(size(a,1),size(a,2)), public util::invmat (real(kind=8), dimension(:,,:), intent(in) A)

Definition at line 179 of file util.f90.

```

179      REAL(KIND=8), DIMENSION(:,,:), INTENT(IN) :: a
180      REAL(KIND=8), DIMENSION(SIZE(A,1),SIZE(A,2)) :: ainv
181
182      REAL(KIND=8), DIMENSION(SIZE(A,1)) :: work ! work array for LAPACK
183      INTEGER, DIMENSION(SIZE(A,1)) :: ipiv ! pivot indices
184      INTEGER :: n, info
185
186      ! Store A in Ainv to prevent it from being overwritten by LAPACK
187      ainv = a
188      n = size(a,1)
189
190      ! DGETRF computes an LU factorization of a general M-by-N matrix A
191      ! using partial pivoting with row interchanges.
192      CALL dgetrf(n, n, ainv, n, ipiv, info)
193
194      IF (info /= 0) THEN
195          stop 'Matrix is numerically singular!'
196      ENDIF
197
198      ! DGETRI computes the inverse of a matrix using the LU factorization
199      ! computed by DGETRF.
200      CALL dgetri(n, ainv, n, ipiv, work, n, info)
201
202      IF (info /= 0) THEN
203          stop 'Matrix inversion failed!'
204      ENDIF

```

7.27.2.9 subroutine, public util::ireduce (real(kind=8), dimension(:,,:), intent(out) A, real(kind=8), dimension(:,,:), intent(in) Ared, integer, intent(in) n, integer, dimension(:), intent(in) ind, integer, dimension(:), intent(in) rind)

Definition at line 275 of file util.f90.

```

275      REAL(KIND=8), DIMENSION(:,,:), INTENT(OUT) :: a
276      REAL(KIND=8), DIMENSION(:,,:), INTENT(IN) :: ared
277      INTEGER, INTENT(IN) :: n
278      INTEGER, DIMENSION(:), INTENT(IN) :: ind, rind
279      INTEGER :: i, j
280      a=0.d0
281      DO i=1,n
282          DO j=1,n
283              a(ind(i), ind(j))=ared(i, j)
284          END DO
285      END DO

```

7.27.2.10 `real(kind=8) function, public util::mat_contract (real(kind=8), dimension(:, :) A, real(kind=8), dimension(:, :) B)`

Definition at line 123 of file util.f90.

```

123     REAL(KIND=8), DIMENSION(:, :) :: a,b
124     REAL(KIND=8) :: mat_contract
125     INTEGER :: i,j,n
126     n=size(a,1)
127     mat_contract=0.d0
128     DO i=1,n
129         DO j=1,n
130             mat_contract=mat_contract+a(i,j)*b(i,j)
131         END DO
132     ENDDO
133     RETURN

```

7.27.2.11 `real(kind=8) function, public util::mat_trace (real(kind=8), dimension(:, :) A)`

Definition at line 111 of file util.f90.

```

111     REAL(KIND=8), DIMENSION(:, :) :: a
112     REAL(KIND=8) :: mat_trace
113     INTEGER :: i,n
114     n=size(a,1)
115     mat_trace=0.d0
116     DO i=1,n
117         mat_trace=mat_trace+a(i,i)
118     END DO
119     RETURN

```

7.27.2.12 `subroutine, public util::printmat (real(kind=8), dimension(:, :), intent(in) A)`

Definition at line 161 of file util.f90.

```

161     REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
162     INTEGER :: i
163
164     DO i=1,SIZE(a,1)
165         print*, a(i,:)
166     END DO

```

7.27.2.13 `subroutine, public util::reduce (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) Ared, integer, intent(out) n, integer, dimension(:), intent(out) ind, integer, dimension(:), intent(out) rind)`

Definition at line 247 of file util.f90.

```

247     REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
248     REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: ared
249     INTEGER, INTENT(OUT) :: n
250     INTEGER, DIMENSION(:), INTENT(OUT) :: ind,rind
251     LOGICAL, DIMENSION(SIZE(A,1)) :: sel
252     INTEGER :: i,j
253
254     ind=0
255     rind=0
256     sel=.false.
257     n=0
258     DO i=1,SIZE(a,1)
259         IF (any(a(i, :)/=0)) THEN
260             n=n+1
261             sel(i)=.true.
262             ind(n)=i
263             rind(i)=n
264         ENDIF
265     END DO
266     ared=0.d0
267     DO i=1,SIZE(a,1)
268         DO j=1,SIZE(a,1)
269             IF (sel(i).and.sel(j)) ared(rind(i),rind(j))=a(i,j)
270         ENDDO
271     ENDDO

```

7.27.2.14 `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 37 of file util.f90.

```

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

```

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

Convert an integer to string.

Definition at line 30 of file util.f90.

```

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

```

7.27.2.16 `subroutine, public util::triu (real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) T)`

Definition at line 208 of file util.f90.

```

208     REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
209     REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: t
210     INTEGER i, j
211     t=0.d0
212     DO i=1, SIZE(a,1)
213         DO j=i, SIZE(a,1)
214             t(i, j)=a(i, j)
215         END DO
216     END DO

```

7.27.2.17 `subroutine, public util::vector_outer (real(kind=8), dimension(:), intent(in) u, real(kind=8), dimension(:), intent(in) v, real(kind=8), dimension(:, :), intent(out) A)`

Definition at line 289 of file util.f90.

```

289     REAL(KIND=8), DIMENSION(:), INTENT(IN) :: u, v
290     REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: a
291     INTEGER :: i, j
292
293     a=0.d0
294     DO i=1, SIZE(u)
295         DO j=1, SIZE(v)
296             a(i, j)=u(i)*v(j)
297         ENDDO
298     ENDDO

```

7.28 `wl_tensor` Module Reference

The WL tensors used to integrate the model.

Functions/Subroutines

- subroutine, public [init_wl_tensor](#)
Subroutine to initialise the WL tensor.

Variables

- real(kind=8), dimension(:), allocatable, public [m11](#)
First component of the M1 term.
- type([coolist](#)), dimension(:), allocatable, public [m12](#)
Second component of the M1 term.
- real(kind=8), dimension(:), allocatable, public [m13](#)
Third component of the M1 term.
- real(kind=8), dimension(:), allocatable, public [m1tot](#)
Total M_1 vector.
- type([coolist](#)), dimension(:), allocatable, public [m21](#)
First tensor of the M2 term.
- type([coolist](#)), dimension(:), allocatable, public [m22](#)
Second tensor of the M2 term.
- type([coolist](#)), dimension(:, :), allocatable, public [l1](#)
First linear tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [l2](#)
Second linear tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [l4](#)
Fourth linear tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [l5](#)
Fifth linear tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [ltot](#)
Total linear tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [b1](#)
First quadratic tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [b2](#)
Second quadratic tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [b3](#)
Third quadratic tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [b4](#)
Fourth quadratic tensor.
- type([coolist](#)), dimension(:, :), allocatable, public [b14](#)
Joint 1st and 4th tensors.
- type([coolist](#)), dimension(:, :), allocatable, public [b23](#)
Joint 2nd and 3rd tensors.
- type([coolist4](#)), dimension(:, :), allocatable, public [mtot](#)
Tensor for the cubic terms.
- real(kind=8), dimension(:), allocatable [dumb_vec](#)
Dummy vector.
- real(kind=8), dimension(:, :), allocatable [dumb_mat1](#)
Dummy matrix.
- real(kind=8), dimension(:, :), allocatable [dumb_mat2](#)
Dummy matrix.
- real(kind=8), dimension(:, :), allocatable [dumb_mat3](#)

Dummy matrix.

- `real(kind=8)`, `dimension(:, :)`, allocatable `dumb_mat4`

Dummy matrix.

- logical, public `m12def`
- logical, public `m21def`
- logical, public `m22def`
- logical, public `ldef`
- logical, public `b14def`
- logical, public `b23def`
- logical, public `mdef`

Boolean to (de)activate the computation of the terms.

7.28.1 Detailed Description

The WL tensors used to integrate the model.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

7.28.2 Function/Subroutine Documentation

7.28.2.1 subroutine, public `wl_tensor::init_wl_tensor ()`

Subroutine to initialise the WL tensor.

Definition at line 94 of file `WL_tensor.f90`.

```

94     INTEGER :: allocstat,i,j,k,m
95
96     print*, 'Initializing the decomposition tensors...'
97     CALL init_dec_tensor
98     print*, "Initializing the correlation matrices and tensors..."
99     CALL init_corr_tensor
100
101     !M1 part
102     print*, "Computing the M1 terms..."
103
104     ALLOCATE(m11(0:ndim), m12(ndim), m13(0:ndim), mltot(0:ndim),
105 stat=allocstat)
106     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
107     ALLOCATE(dumb_mat1(ndim,ndim), dumb_mat2(ndim,ndim), stat=allocstat)
108     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
109     ALLOCATE(dumb_mat3(ndim,ndim), dumb_mat4(ndim,ndim), stat=allocstat)
110     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
111     ALLOCATE(dumb_vec(ndim), stat=allocstat)
112     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
113
114     !M11
115     m11=0.d0
116     ! CALL coo_to_mat_ik(Lxy,dumb_mat1)
117     ! M11(1:ndim)=matmul(dumb_mat1,mean_full(1:ndim))
118
119     !M12

```

```

122      ! dumb_mat2=0.D0
123      ! DO i=1,ndim
124      !      CALL coo_to_mat_i(i,Bxyy,dumb_mat1)
125      !      dumb_mat2(i,:)=matmul(dumb_mat1,mean_full(1:ndim))
126      ! ENDDO
127      ! CALL matc_to_coo(dumb_mat2,M12)
128
129      m12def=.not.tensor_empty(m12)
130
131      !M13
132      m13=0.d0
133      CALL sparse_mul3_with_mat(bxyy,corr_i_full,m13)
134
135      !M1tot
136      m1tot=0.d0
137      m1tot=m11+m13
138
139      print*, "Computing the M2 terms..."
140      ALLOCATE(m21(ndim), m22(ndim), stat=allocstat)
141      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
142
143      !M21
144      CALL copy_tensor(lxy,m21)
145      CALL add_to_tensor(bxyy,m21)
146
147      m21def=.not.tensor_empty(m21)
148
149      !M22
150      CALL copy_tensor(bxyy,m22)
151
152      m22def=.not.tensor_empty(m22)
153
154      !M3 tensor
155      print*, "Computing the M3 terms..."
156      ! Linear terms
157      print*, "Computing the L subterms..."
158      ALLOCATE(l1(ndim,mems), l2(ndim,mems), l4(ndim,mems), l5(ndim,mems),
stat=allocstat)
159      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
160
161      !L1
162      CALL coo_to_mat_ik(lyx,dumb_mat1)
163      CALL coo_to_mat_ik(lxy,dumb_mat2)
164      DO m=1,mems
165          CALL coo_to_mat_ik(dy(:,m),dumb_mat3)
166          dumb_mat4=matmul(dumb_mat2,matmul(transpose(dumb_mat3),dumb_mat1))
167          CALL matc_to_coo(dumb_mat4,l1(:,m))
168      ENDDO
169
170      !L2
171      DO m=1,mems
172          dumb_mat4=0.d0
173          DO i=1,ndim
174              CALL coo_to_mat_i(i,bxyy,dumb_mat1)
175              CALL sparse_mul4_with_mat_kl(ydyy(:,m),dumb_mat1,dumb_mat2)
176              DO j=1,ndim
177                  CALL coo_to_mat_j(j,byxy,dumb_mat1)
178                  dumb_mat4(i,j)=mat_trace(matmul(dumb_mat1,dumb_mat2))
179              ENDDO
180          END DO
181          CALL matc_to_coo(dumb_mat4,l2(:,m))
182      ENDDO
183
184      !L4
185      ! DO m=1,mems
186      !      dumb_mat4=0.D0
187      !      DO i=1,ndim
188      !          CALL coo_to_mat_i(i,Bxyy,dumb_mat1)
189      !          CALL sparse_mul3_with_mat(dYY(:,m),dumb_mat1,dumb_vec) ! Bxyy*dYY
190      !          CALL coo_to_mat_ik(Lyx,dumb_mat1)
191      !          dumb_mat4(i,:)=matmul(transpose(dumb_mat1),dumb_vec)
192      !      ENDDO
193      !      CALL matc_to_coo(dumb_mat4,L4(:,m))
194      ! ENDDO
195
196      !L5
197
198      ! CALL coo_to_mat_ik(Lxy,dumb_mat1)
199      ! DO m=1,mems
200      !      dumb_mat4=0.D0
201      !      DO i=1,ndim
202      !          CALL sparse_mul3_mat(YdY(:,m),dumb_mat1(i,:),dumb_mat2)
203      !          DO j=1,ndim
204      !              CALL coo_to_mat_j(j,Byxy,dumb_mat3)
205      !              dumb_mat4(i,j)=mat_trace(matmul(dumb_mat3,dumb_mat2))
206      !          ENDDO
207      !      END DO

```

```

208      ! CALL matc_to_coo(dumb_mat4,L5(:,m))
209      ! ENDDO
210
211      !Ltot
212
213      ALLOCATE(ltot(ndim,mems), stat=allocstat)
214      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
215
216      DO m=1,mems
217          CALL add_to_tensor(l1(:,m),ltot(:,m))
218          CALL add_to_tensor(l2(:,m),ltot(:,m))
219          CALL add_to_tensor(l4(:,m),ltot(:,m))
220          CALL add_to_tensor(l5(:,m),ltot(:,m))
221      ENDDO
222
223      ldef=.not.tensor_empty(ltot)
224
225      print*, "Computing the B terms..."
226      ALLOCATE(b1(ndim,mems), b2(ndim,mems), b3(ndim,mems), b4(ndim,mems),
stat=allocstat)
227      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
228
229      ! B1
230      CALL coo_to_mat_ik(lxy,dumb_mat1)
231      dumb_mat1=transpose(dumb_mat1)
232      DO m=1,mems
233          CALL coo_to_mat_ik(dy(:,m),dumb_mat2)
234          dumb_mat2=matmul(dumb_mat2,dumb_mat1)
235          DO j=1,ndim
236              DO k=1,ndim
237                  CALL coo_to_vec_jk(j,k,byxx,dumb_vec)
238                  dumb_vec=matmul(dumb_vec,dumb_mat2)
239                  CALL add_vec_jk_to_tensor(j,k,dumb_vec,b1(:,m))
240              ENDDO
241          ENDDO
242      ENDDO
243
244      ! B2
245      CALL coo_to_mat_ik(lyx,dumb_mat3)
246      dumb_mat3=transpose(dumb_mat3)
247      DO m=1,mems
248          DO i=1,ndim
249              CALL coo_to_mat_i(i,bxxy,dumb_mat1)
250              CALL coo_to_mat_ik(dy(:,m),dumb_mat2)
251              dumb_mat1=matmul(dumb_mat2,transpose(dumb_mat1))
252              dumb_mat1=matmul(dumb_mat3,dumb_mat1)
253              CALL add_matc_to_tensor(i,dumb_mat1,b2(:,m))
254          ENDDO
255      ENDDO
256
257      ! B3
258      ! DO m=1,mems
259      !     DO i=1,ndim
260      !         CALL coo_to_mat_i(i,Bxxy,dumb_mat1)
261      !         dumb_mat4=0.D0
262      !         DO j=1,ndim
263      !             CALL coo_to_mat_j(j,Ydy(:,m),dumb_mat2)
264      !             CALL coo_to_mat_i(j,Byxy,dumb_mat3)
265      !             dumb_mat2=matmul(dumb_mat3,dumb_mat2)
266      !             dumb_mat4=dumb_mat4+dumb_mat2
267      !         ENDDO
268      !         dumb_mat4=matmul(dumb_mat4,transpose(dumb_mat1))
269      !         CALL add_matc_to_tensor(i,dumb_mat4,B3(:,m))
270      !     ENDDO
271      ! ENDDO
272
273      ! B4
274      ! DO m=1,mems
275      !     DO i=1,ndim
276      !         CALL coo_to_mat_i(i,Bxyy,dumb_mat1)
277      !         CALL sparse_mul3_with_mat(dYY(:,m),dumb_mat1,dumb_vec) ! Bxyy*dYY
278      !         DO j=1,ndim
279      !             CALL coo_to_mat_j(j,Byxx,dumb_mat1)
280      !             dumb_mat4(j,:)=matmul(transpose(dumb_mat1),dumb_vec)
281      !         ENDDO
282      !         CALL add_matc_to_tensor(i,dumb_mat4,B4(:,m))
283      !     ENDDO
284      ! ENDDO
285
286      ALLOCATE(b14(ndim,mems), b23(ndim,mems), stat=allocstat)
287      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
288
289      DO m=1,mems
290          CALL add_to_tensor(b1(:,m),b14(:,m))
291          CALL add_to_tensor(b2(:,m),b23(:,m))
292          CALL add_to_tensor(b4(:,m),b14(:,m))
293          CALL add_to_tensor(b3(:,m),b23(:,m))

```

```

294      ENDDO
295
296      b14def=.not.tensor_empty(b14)
297      b23def=.not.tensor_empty(b23)
298
299      !M
300
301      print*, "Computing the M term..."
302
303      ALLOCATE(mtot(ndim,mems), stat=allocstat)
304      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
305
306      DO m=1,mems
307          DO i=1,ndim
308              CALL coo_to_mat_i(i,bxxy,dumb_mat1)
309              CALL coo_to_mat_ik(dy(:,m),dumb_mat2)
310              dumb_mat1=matmul(dumb_mat2,transpose(dumb_mat1))
311              DO j=1,ndim
312                  DO k=1,ndim
313                      CALL coo_to_vec_jk(j,k,byxx,dumb_vec)
314                      dumb_vec=matmul(dumb_vec,dumb_mat1)
315                      CALL add_vec_ijk_to_tensor4(i,j,k,dumb_vec,mtot(:,m))
316                  ENDDO
317              END DO
318          END DO
319      END DO
320
321      mdef=.not.tensor4_empty(mtot)
322
323
324      DEALLOCATE(dumb_mat1, dumb_mat2, stat=allocstat)
325      IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
326
327      DEALLOCATE(dumb_mat3, dumb_mat4, stat=allocstat)
328      IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
329
330      DEALLOCATE(dumb_vec, stat=allocstat)
331      IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
332
333

```

7.28.3 Variable Documentation

7.28.3.1 type(coolist), dimension(:,:), allocatable, public wl_tensor::b1

First quadratic tensor.

Definition at line 60 of file WL_tensor.f90.

```
60  TYPE(coolist), DIMENSION(:,:), ALLOCATABLE, PUBLIC :: b1      !< First quadratic tensor
```

7.28.3.2 type(coolist), dimension(:,:), allocatable, public wl_tensor::b14

Joint 1st and 4th tensors.

Definition at line 64 of file WL_tensor.f90.

```
64  TYPE(coolist), DIMENSION(:,:), ALLOCATABLE, PUBLIC :: b14    !< Joint 1st and 4th tensors
```

7.28.3.3 logical, public wl_tensor::b14def

Definition at line 75 of file WL_tensor.f90.

7.28.3.4 type(coolist), dimension(:, :), allocatable, public wl_tensor::b2

Second quadratic tensor.

Definition at line 61 of file WL_tensor.f90.

```
61  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: b2      !< Second quadratic tensor
```

7.28.3.5 type(coolist), dimension(:, :), allocatable, public wl_tensor::b23

Joint 2nd and 3rd tensors.

Definition at line 65 of file WL_tensor.f90.

```
65  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: b23   !< Joint 2nd and 3rd tensors
```

7.28.3.6 logical, public wl_tensor::b23def

Definition at line 75 of file WL_tensor.f90.

7.28.3.7 type(coolist), dimension(:, :), allocatable, public wl_tensor::b3

Third quadratic tensor.

Definition at line 62 of file WL_tensor.f90.

```
62  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: b3    !< Third quadratic tensor
```

7.28.3.8 type(coolist), dimension(:, :), allocatable, public wl_tensor::b4

Fourth quadratic tensor.

Definition at line 63 of file WL_tensor.f90.

```
63  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: b4    !< Fourth quadratic tensor
```

7.28.3.9 real(kind=8), dimension(:, :), allocatable wl_tensor::dumb_mat1 [private]

Dummy matrix.

Definition at line 70 of file WL_tensor.f90.

```
70  REAL(KIND=8), DIMENSION(:, :), ALLOCATABLE :: dumb_mat1 !< Dummy matrix
```

7.28.3.10 `real(kind=8), dimension(:,,:), allocatable wl_tensor::dumb_mat2` [private]

Dummy matrix.

Definition at line 71 of file WL_tensor.f90.

```
71  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: dumb_mat2 !< Dummy matrix
```

7.28.3.11 `real(kind=8), dimension(:,,:), allocatable wl_tensor::dumb_mat3` [private]

Dummy matrix.

Definition at line 72 of file WL_tensor.f90.

```
72  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: dumb_mat3 !< Dummy matrix
```

7.28.3.12 `real(kind=8), dimension(:,,:), allocatable wl_tensor::dumb_mat4` [private]

Dummy matrix.

Definition at line 73 of file WL_tensor.f90.

```
73  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE :: dumb_mat4 !< Dummy matrix
```

7.28.3.13 `real(kind=8), dimension(:), allocatable wl_tensor::dumb_vec` [private]

Dummy vector.

Definition at line 69 of file WL_tensor.f90.

```
69  REAL(KIND=8), DIMENSION(:), ALLOCATABLE :: dumb_vec !< Dummy vector
```

7.28.3.14 `type(coolist), dimension(:,,:), allocatable, public wl_tensor::l1`

First linear tensor.

Definition at line 53 of file WL_tensor.f90.

```
53  TYPE(coolist), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: l1 !< First linear tensor
```

7.28.3.15 type(coolist), dimension(:, :), allocatable, public wl_tensor::l2

Second linear tensor.

Definition at line 54 of file WL_tensor.f90.

```
54  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: l2    !< Second linear tensor
```

7.28.3.16 type(coolist), dimension(:, :), allocatable, public wl_tensor::l4

Fourth linear tensor.

Definition at line 55 of file WL_tensor.f90.

```
55  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: l4    !< Fourth linear tensor
```

7.28.3.17 type(coolist), dimension(:, :), allocatable, public wl_tensor::l5

Fifth linear tensor.

Definition at line 56 of file WL_tensor.f90.

```
56  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: l5    !< Fifth linear tensor
```

7.28.3.18 logical, public wl_tensor::ldef

Definition at line 75 of file WL_tensor.f90.

7.28.3.19 type(coolist), dimension(:, :), allocatable, public wl_tensor::ltot

Total linear tensor.

Definition at line 57 of file WL_tensor.f90.

```
57  TYPE(coolist), DIMENSION(:, :), ALLOCATABLE, PUBLIC :: ltot  !< Total linear tensor
```

7.28.3.20 real(kind=8), dimension(:), allocatable, public wl_tensor::m11

First component of the M1 term.

Definition at line 42 of file WL_tensor.f90.

```
42  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: m11      !< First component of the M1 term
```

7.28.3.21 type(coolist), dimension(:), allocatable, public wl_tensor::m12

Second component of the M1 term.

Definition at line 43 of file WL_tensor.f90.

```
43  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: m12    !< Second component of the M1 term
```

7.28.3.22 logical, public wl_tensor::m12def

Definition at line 75 of file WL_tensor.f90.

```
75  LOGICAL, PUBLIC :: m12def,m21def,m22def,ldef,b14def,b23def,mdef !< Boolean to (de)activate the
    computation of the terms
```

7.28.3.23 real(kind=8), dimension(:), allocatable, public wl_tensor::m13

Third component of the M1 term.

Definition at line 44 of file WL_tensor.f90.

```
44  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: m13    !< Third component of the M1 term
```

7.28.3.24 real(kind=8), dimension(:), allocatable, public wl_tensor::m1tot

Total M_1 vector.

Definition at line 45 of file WL_tensor.f90.

```
45  REAL(KIND=8), DIMENSION(:), ALLOCATABLE, PUBLIC :: m1tot  !< Total \f$M_1\f$ vector
```

7.28.3.25 type(coolist), dimension(:), allocatable, public wl_tensor::m21

First tensor of the M2 term.

Definition at line 48 of file WL_tensor.f90.

```
48  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: m21    !< First tensor of the M2 term
```

7.28.3.26 logical, public wl_tensor::m21def

Definition at line 75 of file WL_tensor.f90.

7.28.3.27 type(coolist), dimension(:), allocatable, public wl_tensor::m22

Second tensor of the M2 term.

Definition at line 49 of file WL_tensor.f90.

```
49  TYPE(coolist), DIMENSION(:), ALLOCATABLE, PUBLIC :: m22    !< Second tensor of the M2 term
```

7.28.3.28 logical, public wl_tensor::m22def

Definition at line 75 of file WL_tensor.f90.

7.28.3.29 logical, public wl_tensor::mdef

Boolean to (de)activate the computation of the terms.

Definition at line 75 of file WL_tensor.f90.

7.28.3.30 type(coolist4), dimension(:, :, :), allocatable, public wl_tensor::mtot

Tensor for the cubic terms.

Definition at line 67 of file WL_tensor.f90.

```
67  TYPE(coolist4), DIMENSION(:, :, :), ALLOCATABLE, PUBLIC :: mtot !< Tensor for the cubic terms
```


Chapter 8

Data Type Documentation

8.1 inprod_analytic::atm_tensors Type Reference

Type holding the atmospheric inner products tensors.

Private Attributes

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

8.1.1 Detailed Description

Type holding the atmospheric inner products tensors.

Definition at line 52 of file `inprod_analytic.f90`.

8.1.2 Member Data Documentation

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

Definition at line 53 of file `inprod_analytic.f90`.

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

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

Definition at line 54 of file `inprod_analytic.f90`.

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

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

Definition at line 53 of file inprod_analytic.f90.

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

Definition at line 53 of file inprod_analytic.f90.

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

Definition at line 54 of file inprod_analytic.f90.

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

Definition at line 53 of file inprod_analytic.f90.

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

- [inprod_analytic.f90](#)

8.2 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.

8.2.1 Detailed Description

Atmospheric bloc specification type.

Definition at line 39 of file inprod_analytic.f90.

8.2.2 Member Data Documentation

8.2.2.1 `integer inprod_analytic::atm_wavenum::h =0` `[private]`

Definition at line 41 of file inprod_analytic.f90.

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

Definition at line 41 of file inprod_analytic.f90.

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

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

Definition at line 42 of file inprod_analytic.f90.

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

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

Definition at line 42 of file inprod_analytic.f90.

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

Definition at line 41 of file inprod_analytic.f90.

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

Definition at line 40 of file inprod_analytic.f90.

```
40      CHARACTER :: typ
```

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

- [inprod_analytic.f90](#)

8.3 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.

8.3.1 Detailed Description

Coordinate list. Type used to represent the sparse tensor.

Definition at line 38 of file tensor.f90.

8.3.2 Member Data Documentation

8.3.2.1 `type(coolist_elem)`, `dimension(:)`, allocatable `tensor::coolist::elems`

Lists of elements [tensor::coolist_elem](#).

Definition at line 39 of file tensor.f90.

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

8.3.2.2 integer `tensor::coolist::nelems` = 0

Number of elements in the list.

Definition at line 40 of file tensor.f90.

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

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

- [tensor.f90](#)

8.4 `tensor::coolist4` Type Reference

4d coordinate list. Type used to represent the rank-4 sparse tensor.

Public Attributes

- `type(coolist_elem4)`, `dimension(:)`, allocatable [elems](#)
- integer [nelems](#) = 0

8.4.1 Detailed Description

4d coordinate list. Type used to represent the rank-4 sparse tensor.

Definition at line 44 of file tensor.f90.

8.4.2 Member Data Documentation

8.4.2.1 type(coolist_elem4), dimension(:), allocatable tensor::coolist4::elems

Definition at line 45 of file tensor.f90.

```
45      TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: elems
```

8.4.2.2 integer tensor::coolist4::nelems = 0

Definition at line 46 of file tensor.f90.

```
46      INTEGER :: nelems = 0
```

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

- [tensor.f90](#)

8.5 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.

8.5.1 Detailed Description

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

Definition at line 25 of file tensor.f90.

8.5.2 Member Data Documentation

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

Index j of the element.

Definition at line 26 of file tensor.f90.

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

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

Index k of the element.

Definition at line 27 of file tensor.f90.

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

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

Value of the element.

Definition at line 28 of file tensor.f90.

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

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

- [tensor.f90](#)

8.6 tensor::coolist_elem4 Type Reference

4d coordinate list element type. Elementary elements of the 4d sparse tensors.

Private Attributes

- integer [j](#)
- integer [k](#)
- integer [l](#)
- real(kind=8) [v](#)

8.6.1 Detailed Description

4d coordinate list element type. Elementary elements of the 4d sparse tensors.

Definition at line 32 of file tensor.f90.

8.6.2 Member Data Documentation

8.6.2.1 integer tensor::coolist_elem4::j [private]

Definition at line 33 of file tensor.f90.

```
33      INTEGER :: j,k,l
```


8.6.2.2 integer tensor::coolist_elem4::k [private]

Definition at line 33 of file tensor.f90.

8.6.2.3 integer tensor::coolist_elem4::l [private]

Definition at line 33 of file tensor.f90.

8.6.2.4 real(kind=8) tensor::coolist_elem4::v [private]

Definition at line 34 of file tensor.f90.

```
34      REAL(KIND=8) :: v
```

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

- [tensor.f90](#)

8.7 inprod_analytic::ocean_tensors Type Reference

Type holding the oceanic inner products tensors.

Private Attributes

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

8.7.1 Detailed Description

Type holding the oceanic inner products tensors.

Definition at line 58 of file inprod_analytic.f90.

8.7.2 Member Data Documentation**8.7.2.1 real(kind=8), dimension(:,:), allocatable inprod_analytic::ocean_tensors::c [private]**

Definition at line 60 of file inprod_analytic.f90.

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

Definition at line 59 of file `inprod_analytic.f90`.

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

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

Definition at line 59 of file `inprod_analytic.f90`.

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

Definition at line 59 of file `inprod_analytic.f90`.

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

Definition at line 60 of file `inprod_analytic.f90`.

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

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

Definition at line 59 of file `inprod_analytic.f90`.

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

- [inprod_analytic.f90](#)

8.8 inprod_analytic::ocean_wavenum Type Reference

Oceanic bloc specification type.

Private Attributes

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

8.8.1 Detailed Description

Oceanic bloc specification type.

Definition at line 46 of file inprod_analytic.f90.

8.8.2 Member Data Documentation

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

Definition at line 47 of file inprod_analytic.f90.

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

Definition at line 48 of file inprod_analytic.f90.

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

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

Definition at line 48 of file inprod_analytic.f90.

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

Definition at line 47 of file inprod_analytic.f90.

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

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

- [inprod_analytic.f90](#)

Chapter 9

File Documentation

9.1 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.

9.2 corr_tensor.f90 File Reference

Modules

- module [corr_tensor](#)
Module to compute the correlations and derivatives used to compute the memory term of the WL parameterization.

Functions/Subroutines

- subroutine, public [corr_tensor::init_corr_tensor](#)
Subroutine to initialise the correlations tensors.

Variables

- type(coolist), dimension(:,:), allocatable, public [corr_tensor::yy](#)
Coolist holding the $\langle Y \otimes Y^s \rangle$ terms.
- type(coolist), dimension(:,:), allocatable, public [corr_tensor::dy](#)
Coolist holding the $\langle \partial_Y \otimes Y^s \rangle$ terms.
- type(coolist), dimension(:,:), allocatable, public [corr_tensor::ydy](#)
Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \rangle$ terms.
- type(coolist), dimension(:,:), allocatable, public [corr_tensor::dyy](#)
Coolist holding the $\langle \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.
- type(coolist4), dimension(:,:), allocatable, public [corr_tensor::ydydy](#)
Coolist holding the $\langle Y \otimes \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms.
- real(kind=8), dimension(:), allocatable [corr_tensor::dumb_vec](#)
Dumb vector to be used in the calculation.
- real(kind=8), dimension(:,:), allocatable [corr_tensor::dumb_mat1](#)
Dumb matrix to be used in the calculation.
- real(kind=8), dimension(:,:), allocatable [corr_tensor::dumb_mat2](#)
Dumb matrix to be used in the calculation.
- real(kind=8), dimension(:,:), allocatable [corr_tensor::expm](#)
Matrix holding the product $\text{inv_corr}_i \cdot \text{corr}_{ij}$ at time s .

9.3 corrmod.f90 File Reference

Modules

- module [corrmod](#)
Module to initialize the correlation matrix of the unresolved variables.

Functions/Subroutines

- subroutine, public [corrmod::init_corr](#)
Subroutine to initialise the computation of the correlation.
- subroutine [corrmod::corrcomp_from_def](#) (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the definition given inside the module.
- subroutine [corrmod::corrcomp_from_spline](#) (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the spline representation.
- subroutine [corrmod::splint](#) (xa, ya, y2a, n, x, y)
Routine to compute the spline representation parameters.
- real(kind=8) function [corrmod::fs](#) (s, p)
Exponential fit function.
- subroutine [corrmod::corrcomp_from_fit](#) (s)
Subroutine to compute the correlation of the unresolved variables $\langle Y \otimes Y^s \rangle$ at time s from the exponential representation.

Variables

- real(kind=8), dimension(:), allocatable, public [corrmod::mean](#)
Vector holding the mean of the unresolved dynamics (reduced version)
- real(kind=8), dimension(:), allocatable, public [corrmod::mean_full](#)
Vector holding the mean of the unresolved dynamics (full version)
- real(kind=8), dimension(:, :), allocatable, public [corrmod::corr_i_full](#)
Covariance matrix of the unresolved variables (full version)
- real(kind=8), dimension(:, :), allocatable, public [corrmod::inv_corr_i_full](#)
Inverse of the covariance matrix of the unresolved variables (full version)
- real(kind=8), dimension(:, :), allocatable, public [corrmod::corr_i](#)
Covariance matrix of the unresolved variables (reduced version)
- real(kind=8), dimension(:, :), allocatable, public [corrmod::inv_corr_i](#)
Inverse of the covariance matrix of the unresolved variables (reduced version)
- real(kind=8), dimension(:, :), allocatable, public [corrmod::corr_ij](#)
Matrix holding the correlation matrix at a given time.
- real(kind=8), dimension(:, :, :), allocatable [corrmod::y2](#)
Vector holding coefficient of the spline and exponential correlation representation.
- real(kind=8), dimension(:, :, :), allocatable [corrmod::ya](#)
Vector holding coefficient of the spline and exponential correlation representation.
- real(kind=8), dimension(:), allocatable [corrmod::xa](#)
Vector holding coefficient of the spline and exponential correlation representation.
- integer [corrmod::nspl](#)
Integers needed by the spline representation of the correlation.
- integer [corrmod::klo](#)
- integer [corrmod::khi](#)
- procedure(corrcomp_from_spline), pointer, public [corrmod::corrcomp](#)
Pointer to the correlation computation routine.

9.4 dec_tensor.f90 File Reference

Modules

- module [dec_tensor](#)
The resolved-unresolved components decomposition of the tensor.

Functions/Subroutines

- subroutine `dec_tensor::suppress_and` (t, cst, v1, v2)
Subroutine to suppress from the tensor t_{ijk} components satisfying $SF(j)=v1$ and $SF(k)=v2$.
- subroutine `dec_tensor::suppress_or` (t, cst, v1, v2)
Subroutine to suppress from the tensor t_{ijk} components satisfying $SF(j)=v1$ or $SF(k)=v2$.
- subroutine `dec_tensor::reorder` (t, cst, v)
Subroutine to reorder the tensor t_{ijk} components : if $SF(j)=v$ then it return t_{ikj} .
- subroutine `dec_tensor::init_sub_tensor` (t, cst, v)
Subroutine that suppress all the components of a tensor t_{ijk} where if $SF(i)=v$.
- subroutine, public `dec_tensor::init_dec_tensor`
Subroutine that initialize and compute the decomposed tensors.

Variables

- type(coolist), dimension(:), allocatable, public `dec_tensor::ff_tensor`
Tensor holding the part of the unresolved tensor involving only unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::sf_tensor`
Tensor holding the part of the resolved tensor involving unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::ss_tensor`
Tensor holding the part of the resolved tensor involving only resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::fs_tensor`
Tensor holding the part of the unresolved tensor involving resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::hx`
Tensor holding the constant part of the resolved tendencies.
- type(coolist), dimension(:), allocatable, public `dec_tensor::lxx`
Tensor holding the linear part of the resolved tendencies involving the resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::lxy`
Tensor holding the linear part of the resolved tendencies involving the unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::bxxx`
Tensor holding the quadratic part of the resolved tendencies involving resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::bxyx`
Tensor holding the quadratic part of the resolved tendencies involving both resolved and unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::bxxy`
Tensor holding the quadratic part of the resolved tendencies involving unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::hy`
Tensor holding the constant part of the unresolved tendencies.
- type(coolist), dimension(:), allocatable, public `dec_tensor::lyx`
Tensor holding the linear part of the unresolved tendencies involving the resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::lyy`
Tensor holding the linear part of the unresolved tendencies involving the unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::byxx`
Tensor holding the quadratic part of the unresolved tendencies involving resolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::byxy`
Tensor holding the quadratic part of the unresolved tendencies involving both resolved and unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::byyy`
Tensor holding the quadratic part of the unresolved tendencies involving unresolved variables.
- type(coolist), dimension(:), allocatable, public `dec_tensor::ss_tl_tensor`
Tensor of the tangent linear model tendencies of the resolved component alone.
- type(coolist), dimension(:), allocatable `dec_tensor::dumb`
Dumb coolist to make the computations.

9.5 doc/gen_doc.md File Reference

9.6 doc/sto_doc.md File Reference

9.7 doc/tl_ad_doc.md File Reference

9.8 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.

9.9 inprod_analytic.f90 File Reference

Data Types

- type `inprod_analytic::atm_wavenum`
Atmospheric bloc specification type.
- type `inprod_analytic::ocean_wavenum`
Oceanic bloc specification type.
- type `inprod_analytic::atm_tensors`
Type holding the atmospheric inner products tensors.
- type `inprod_analytic::ocean_tensors`
Type holding the oceanic inner products tensors.

Modules

- module `inprod_analytic`
Inner products between the truncated set of basis functions for the ocean and atmosphere streamfunction fields. These are partly calculated using the analytical expressions from Cehelsky, P., & Tung, K. K. : Theories of multiple equilibria and weather regimes-A critical reexamination. Part II: Baroclinic two-layer models. Journal of the atmospheric sciences, 44(21), 3282-3303, 1987.

Functions/Subroutines

- real(kind=8) function [inprod_analytic::b1](#) (Pi, Pj, Pk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [inprod_analytic::b2](#) (Pi, Pj, Pk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [inprod_analytic::delta](#) (r)
Integer Dirac delta function.
- real(kind=8) function [inprod_analytic::flambda](#) (r)
"Odd or even" function
- real(kind=8) function [inprod_analytic::s1](#) (Pj, Pk, Mj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [inprod_analytic::s2](#) (Pj, Pk, Mj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [inprod_analytic::s3](#) (Pj, Pk, Hj, Hk)
Cehelsky & Tung Helper functions.
- real(kind=8) function [inprod_analytic::s4](#) (Pj, Pk, Hj, Hk)
Cehelsky & Tung Helper functions.
- subroutine [inprod_analytic::calculate_a](#)
Eigenvalues of the Laplacian (atmospheric)
- subroutine [inprod_analytic::calculate_b](#)
Streamfunction advection terms (atmospheric)
- subroutine [inprod_analytic::calculate_c_atm](#)
Beta term for the atmosphere.
- subroutine [inprod_analytic::calculate_d](#)
Forcing of the ocean on the atmosphere.
- subroutine [inprod_analytic::calculate_g](#)
Temperature advection terms (atmospheric)
- subroutine [inprod_analytic::calculate_s](#)
Forcing (thermal) of the ocean on the atmosphere.
- subroutine [inprod_analytic::calculate_k](#)
Forcing of the atmosphere on the ocean.
- subroutine [inprod_analytic::calculate_m](#)
Forcing of the ocean fields on the ocean.
- subroutine [inprod_analytic::calculate_n](#)
Beta term for the ocean.
- subroutine [inprod_analytic::calculate_o](#)
Temperature advection term (passive scalar)
- subroutine [inprod_analytic::calculate_c_oc](#)
Streamfunction advection terms (oceanic)
- subroutine [inprod_analytic::calculate_w](#)
Short-wave radiative forcing of the ocean.
- subroutine, public [inprod_analytic::init_inprod](#)
Initialisation of the inner product.
- subroutine, public [inprod_analytic::deallocate_inprod](#)
Deallocation of the inner products.

Variables

- type(atm_wavenum), dimension(:), allocatable, public [inprod_analytic::awavenum](#)
Atmospheric blocs specification.
- type(ocean_wavenum), dimension(:), allocatable, public [inprod_analytic::owavenum](#)
Oceanic blocs specification.
- type(atm_tensors), public [inprod_analytic::atmos](#)
Atmospheric tensors.
- type(ocean_tensors), public [inprod_analytic::ocean](#)
Oceanic tensors.

9.10 int_comp.f90 File Reference

Modules

- module [int_comp](#)
Utility module containing the routines to perform the integration of functions.

Functions/Subroutines

- subroutine, public [int_comp::integrate](#) (func, ss)
Routine to compute integrals of function from 0 to #maxint.
- subroutine [int_comp::qromb](#) (func, a, b, ss)
Romberg integration routine.
- subroutine [int_comp::qromo](#) (func, a, b, ss, choose)
Romberg integration routine on an open interval.
- subroutine [int_comp::polint](#) (xa, ya, n, x, y, dy)
Polynomial interpolation routine.
- subroutine [int_comp::trapzd](#) (func, a, b, s, n)
Trapezoidal rule integration routine.
- subroutine [int_comp::midpnt](#) (func, a, b, s, n)
Midpoint rule integration routine.
- subroutine [int_comp::midexp](#) (funkt, aa, bb, s, n)
Midpoint routine for bb infinite with funk decreasing infinitely rapidly at infinity.

9.11 int_corr.f90 File Reference

Modules

- module [int_corr](#)
Module to compute or load the integrals of the correlation matrices.

Functions/Subroutines

- subroutine, public [int_corr::init_corrint](#)
Subroutine to initialise the integrated matrices and tensors.
- real(kind=8) function [int_corr::func_ij](#) (s)
Function that returns the component oi and oj of the correlation matrix at time s.
- real(kind=8) function [int_corr::func_ijkl](#) (s)
Function that returns the component oi,oj,ok and ol of the outer product of the correlation matrix with itself at time s.
- subroutine, public [int_corr::comp_corrint](#)
Routine that actually compute or load the integrals.

Variables

- integer [int_corr::oi](#)
- integer [int_corr::oj](#)
- integer [int_corr::ok](#)
- integer [int_corr::ol](#)
Integers that specify the matrices and tensor component considered as a function of time.
- real(kind=8), parameter [int_corr::real_eps](#) = 2.2204460492503131e-16
Small epsilon constant to determine equality with zero.
- real(kind=8), dimension(:,:), allocatable, public [int_corr::corrint](#)
Matrix holding the integral of the correlation matrix.
- type(coolist4), dimension(:), allocatable, public [int_corr::corr2int](#)
Tensor holding the integral of the correlation outer product with itself.

9.12 LICENSE.txt File Reference

Functions

- The MIT [License](#) (MIT) Copyright(c) 2015-2017 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](#)

9.12.1 Function Documentation

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

9.12.1.2 The MIT License (MIT)

9.12.2 Variable Documentation

9.12.2.1 The MIT free of charge

Definition at line 6 of file LICENSE.txt.

9.12.2.2 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.

9.12.2.3 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.

9.12.2.4 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.

9.12.2.5 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.

9.12.2.6 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.

9.12.2.7 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.

9.12.2.8 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.

9.12.2.9 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.

9.12.2.10 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.

9.12.2.11 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.

9.12.2.12 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.

9.12.2.13 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.

9.12.2.14 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.

9.12.2.15 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.

9.12.2.16 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.

9.12.2.17 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.

9.12.2.18 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.

9.12.2.19 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.

9.12.2.20 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.

9.13 maooam.f90 File Reference

Functions/Subroutines

- program [maooam](#)

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

9.13.1 Function/Subroutine Documentation

9.13.1.1 program maoam ()

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 maoam.f90.

9.14 maoam_MTV.f90 File Reference

Functions/Subroutines

- program [maoam_mtv](#)

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

9.14.1 Function/Subroutine Documentation

9.14.1.1 program maoam_mtv ()

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

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file maoam_MTV.f90.

9.15 maoam_stoch.f90 File Reference

Functions/Subroutines

- program [maoam_stoch](#)

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

9.15.1 Function/Subroutine Documentation

9.15.1.1 program maoam_stoch ()

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

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Remarks

There are four dynamics modes:

- full: generate the full dynamics
- unres: generate the intrinsic unresolved dynamics
- qfst: generate dynamics given by the quadratic terms of the unresolved tendencies
- reso: use the resolved dynamics alone

Definition at line 24 of file maoam_stoch.f90.

9.16 maoam_WL.f90 File Reference

Functions/Subroutines

- program [maoam_wl](#)

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

9.16.1 Function/Subroutine Documentation

9.16.1.1 program maoam_wl ()

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

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file maoam_WL.f90.

9.17 MAR.f90 File Reference

Modules

- module [mar](#)

Multidimensional Autoregressive module to generate the correlation for the WL parameterization.

Functions/Subroutines

- subroutine, public `mar::init_mar`
Subroutine to initialise the MAR.
- subroutine, public `mar::mar_step` (x)
Routine to generate one step of the MAR.
- subroutine, public `mar::mar_step_red` (xred)
Routine to generate one step of the reduce MAR.
- subroutine `mar::stoch_vec` (dW)

Variables

- real(kind=8), dimension(:,:), allocatable, public `mar::q`
Square root of the noise covariance matrix.
- real(kind=8), dimension(:,:), allocatable, public `mar::qred`
Reduce version of Q.
- real(kind=8), dimension(:,:), allocatable, public `mar::rrred`
Covariance matrix of the noise.
- real(kind=8), dimension(:,:), allocatable, public `mar::w`
W_i matrix.
- real(kind=8), dimension(:,:), allocatable, public `mar::wred`
Reduce W_i matrix.
- real(kind=8), dimension(:), allocatable `mar::buf_y`
- real(kind=8), dimension(:), allocatable `mar::dw`
- integer, public `mar::ms`
order of the MAR

9.18 memory.f90 File Reference

Modules

- module `memory`
Module that compute the memory term M_3 of the WL parameterization.

Functions/Subroutines

- subroutine, public `memory::init_memory`
Subroutine to initialise the memory.
- subroutine, public `memory::compute_m3` (y, dt, dtn, savey, save_ev, evolve, inter, h_int)
Compute the integrand of M_3 at each time in the past and integrate to get the memory term.
- subroutine, public `memory::test_m3` (y, dt, dtn, h_int)
Routine to test the #compute_M3 routine.

Variables

- real(kind=8), dimension(:,:), allocatable [memory::x](#)
Array storing the previous state of the system.
- real(kind=8), dimension(:,:), allocatable [memory::xs](#)
Array storing the resolved time evolution of the previous state of the system.
- real(kind=8), dimension(:,:), allocatable [memory::zs](#)
Dummy array to replace Xs in case where the evolution is not stored.
- real(kind=8), dimension(:), allocatable [memory::buf_m](#)
Dummy vector.
- real(kind=8), dimension(:), allocatable [memory::buf_m3](#)
Dummy vector to store the M_3 integrand.
- integer [memory::t_index](#)
Integer storing the time index (current position in the arrays)
- procedure(ss_step), pointer [memory::step](#)
Procedural pointer pointing on the resolved dynamics step routine.

9.19 MTV_int_tensor.f90 File Reference

Modules

- module [mtv_int_tensor](#)
The MTV tensors used to integrate the MTV model.

Functions/Subroutines

- subroutine, public [mtv_int_tensor::init_mtv_int_tensor](#)
Subroutine to initialise the MTV tensor.

Variables

- real(kind=8), dimension(:), allocatable, public [mtv_int_tensor::h1](#)
First constant vector.
- real(kind=8), dimension(:), allocatable, public [mtv_int_tensor::h2](#)
Second constant vector.
- real(kind=8), dimension(:), allocatable, public [mtv_int_tensor::h3](#)
Third constant vector.
- real(kind=8), dimension(:), allocatable, public [mtv_int_tensor::htot](#)
Total constant vector.
- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::l1](#)
First linear tensor.
- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::l2](#)
Second linear tensor.
- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::l3](#)
Third linear tensor.
- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::ltot](#)
Total linear tensor.
- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::b1](#)

First quadratic tensor.

- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::b2](#)

Second quadratic tensor.

- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::btot](#)

Total quadratic tensor.

- type(coolist4), dimension(:), allocatable, public [mtv_int_tensor::mtot](#)

Tensor for the cubic terms.

- real(kind=8), dimension(:,,:), allocatable, public [mtv_int_tensor::q1](#)

Constant terms for the state-dependent noise covariance matrix.

- real(kind=8), dimension(:,,:), allocatable, public [mtv_int_tensor::q2](#)

Constant terms for the state-independent noise covariance matrix.

- type(coolist), dimension(:), allocatable, public [mtv_int_tensor::utot](#)

Linear terms for the state-dependent noise covariance matrix.

- type(coolist4), dimension(:), allocatable, public [mtv_int_tensor::vtot](#)

Quadratic terms for the state-dependent noise covariance matrix.

- real(kind=8), dimension(:), allocatable [mtv_int_tensor::dumb_vec](#)

Dummy vector.

- real(kind=8), dimension(:,,:), allocatable [mtv_int_tensor::dumb_mat1](#)

Dummy matrix.

- real(kind=8), dimension(:,,:), allocatable [mtv_int_tensor::dumb_mat2](#)

Dummy matrix.

- real(kind=8), dimension(:,,:), allocatable [mtv_int_tensor::dumb_mat3](#)

Dummy matrix.

- real(kind=8), dimension(:,,:), allocatable [mtv_int_tensor::dumb_mat4](#)

Dummy matrix.

9.20 MTV_sigma_tensor.f90 File Reference

Modules

- module [sigma](#)

The MTV noise sigma matrices used to integrate the MTV model.

Functions/Subroutines

- subroutine, public [sigma::init_sigma](#) (mult, Q1fill)

Subroutine to initialize the sigma matrices.

- subroutine, public [sigma::compute_mult_sigma](#) (y)

Routine to actualize the matrix σ_1 based on the state y of the MTV system.

Variables

- real(kind=8), dimension(:,:), allocatable, public [sigma::sig1](#)
 $\sigma_1(X)$ state-dependent noise matrix
- real(kind=8), dimension(:,:), allocatable, public [sigma::sig2](#)
 σ_2 state-independent noise matrix
- real(kind=8), dimension(:,:), allocatable, public [sigma::sig1r](#)
Reduced $\sigma_1(X)$ state-dependent noise matrix.
- real(kind=8), dimension(:,:), allocatable [sigma::dumb_mat1](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [sigma::dumb_mat2](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [sigma::dumb_mat3](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [sigma::dumb_mat4](#)
Dummy matrix.
- integer, dimension(:), allocatable [sigma::ind1](#)
- integer, dimension(:), allocatable [sigma::rind1](#)
- integer, dimension(:), allocatable [sigma::ind2](#)
- integer, dimension(:), allocatable [sigma::rind2](#)
Reduction indices.
- integer [sigma::n1](#)
- integer [sigma::n2](#)

9.21 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`
Mechanical 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::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.
 - integer, dimension(:, :), allocatable [params::ams](#)

9.22 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.

9.23 rk2_MTV_integrator.f90 File Reference

Modules

- module [rk2_mtv_integrator](#)
Module with the MTV rk2 integration routines.

Functions/Subroutines

- subroutine, public [rk2_mtv_integrator::init_integrator](#)
Subroutine to initialize the MTV rk2 integrator.
- subroutine [rk2_mtv_integrator::init_noise](#)
Routine to initialize the noise vectors and buffers.
- subroutine [rk2_mtv_integrator::init_g](#)
Routine to initialize the G term.
- subroutine [rk2_mtv_integrator::compg](#) (y)
Routine to actualize the G term based on the state y of the MTV system.
- subroutine, public [rk2_mtv_integrator::step](#) (y, t, dt, dtn, res, tend)
Routine to perform an integration step (Heun algorithm) of the MTV system. The incremented time is returned.
- subroutine, public [rk2_mtv_integrator::full_step](#) (y, t, dt, dtn, res)
Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::buf_y1](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::buf_f0](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::buf_f1](#)
Integration buffers.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dw](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dwmult](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dwar](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dwau](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dwor](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::dwou](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::anoise](#)
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::noise](#)
Additive noise term.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::noisemult](#)
Multiplicative noise term.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::g](#)
G term of the MTV tendencies.
- real(kind=8), dimension(:), allocatable [rk2_mtv_integrator::buf_g](#)
Buffer for the G term computation.
- logical [rk2_mtv_integrator::mult](#)
Logical indicating if the sigma1 matrix must be computed for every state change.
- logical [rk2_mtv_integrator::q1fill](#)
Logical indicating if the matrix Q1 is non-zero.
- logical [rk2_mtv_integrator::compute_mult](#)
Logical indicating if the Gaussian noise for the multiplicative noise must be computed.
- real(kind=8), parameter [rk2_mtv_integrator::sq2](#) = sqrt(2.D0)
Hard coded square root of 2.

9.24 rk2_ss_integrator.f90 File Reference

Modules

- module [rk2_ss_integrator](#)
Module with the stochastic uncoupled resolved nonlinear and tangent linear rk2 dynamics integration routines.

Functions/Subroutines

- subroutine, public [rk2_ss_integrator::init_ss_integrator](#)
Subroutine to initialize the uncoupled resolved rk2 integrator.
- subroutine, public [rk2_ss_integrator::tendencies](#) (t, y, res)
Routine computing the tendencies of the uncoupled resolved model.
- subroutine, public [rk2_ss_integrator::tl_tendencies](#) (t, y, ys, res)
Tendencies for the tangent linear model of the uncoupled resolved dynamics in point ystar for perturbation deltat.
- subroutine, public [rk2_ss_integrator::ss_step](#) (y, ys, t, dt, dtn, res)
Routine to perform a stochastic integration step of the unresolved uncoupled dynamics (Heun algorithm). The incremented time is returned.
- subroutine, public [rk2_ss_integrator::ss_tl_step](#) (y, ys, t, dt, dtn, res)
Routine to perform a stochastic integration step of the unresolved uncoupled tangent linear dynamics (Heun algorithm). The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::dwar](#)
- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::dwor](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::anoise](#)
Additive noise term.
- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::buf_y1](#)
- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::buf_f0](#)
- real(kind=8), dimension(:), allocatable [rk2_ss_integrator::buf_f1](#)
Integration buffers.

9.25 rk2_stoch_integrator.f90 File Reference

Modules

- module [rk2_stoch_integrator](#)
Module with the stochastic rk2 integration routines.

Functions/Subroutines

- subroutine, public [rk2_stoch_integrator::init_integrator](#) (force)
Subroutine to initialize the integrator.
- subroutine [rk2_stoch_integrator::tendencies](#) (t, y, res)
Routine computing the tendencies of the selected model.
- subroutine, public [rk2_stoch_integrator::step](#) (y, t, dt, dtn, res, tend)
Routine to perform a stochastic step of the selected dynamics (Heun algorithm). The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::dwar](#)
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::dwau](#)
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::dwor](#)
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::dwou](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::buf_y1](#)
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::buf_f0](#)
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::buf_f1](#)
Integration buffers.
- real(kind=8), dimension(:), allocatable [rk2_stoch_integrator::anoise](#)
Additive noise term.
- type(coolist), dimension(:), allocatable [rk2_stoch_integrator::int_tensor](#)
Dummy tensor that will hold the tendencies tensor.

9.26 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.

9.27 rk2_WL_integrator.f90 File Reference

Modules

- module [rk2_wl_integrator](#)
Module with the WL rk2 integration routines.

Functions/Subroutines

- subroutine, public [rk2_wl_integrator::init_integrator](#)
Subroutine that initialize the MARs, the memory unit and the integration buffers.
- subroutine [rk2_wl_integrator::compute_m1](#) (y)
Routine to compute the M_1 term.
- subroutine [rk2_wl_integrator::compute_m2](#) (y)
Routine to compute the M_2 term.
- subroutine, public [rk2_wl_integrator::step](#) (y, t, dt, dtn, res, tend)
Routine to perform an integration step (Heun algorithm) of the WL system. The incremented time is returned.
- subroutine, public [rk2_wl_integrator::full_step](#) (y, t, dt, dtn, res)
Routine to perform an integration step (Heun algorithm) of the full stochastic system. The incremented time is returned.

Variables

- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_y1](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_f0](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_f1](#)
Integration buffers.
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_m2](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_m1](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_m3](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_m](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::buf_m3s](#)
Dummy buffers holding the terms /f\$M_i.
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::anoise](#)
Additive noise term.
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::dwar](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::dwau](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::dwor](#)
- real(kind=8), dimension(:), allocatable [rk2_wl_integrator::dwou](#)
Standard gaussian noise buffers.
- real(kind=8), dimension(:, :), allocatable [rk2_wl_integrator::x1](#)
Buffer holding the subsequent states of the first MAR.
- real(kind=8), dimension(:, :), allocatable [rk2_wl_integrator::x2](#)
Buffer holding the subsequent states of the second MAR.

9.28 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.

9.29 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.

9.30 sf_def.f90 File Reference

Modules

- module [sf_def](#)

Module to select the resolved-unresolved components.

Functions/Subroutines

- subroutine, public [sf_def::load_sf](#)

Subroutine to load the unresolved variable definition vector SF from $SF.nml$ if it exists. If it does not, then write $SF.nml$ with no unresolved variables specified (null vector).

Variables

- logical [sf_def::exists](#)
Boolean to test for file existence.
- integer, dimension(:), allocatable, public [sf_def::sf](#)
Unresolved variable definition vector.
- integer, dimension(:), allocatable, public [sf_def::ind](#)
- integer, dimension(:), allocatable, public [sf_def::rind](#)
Unresolved reduction indices.
- integer, dimension(:), allocatable, public [sf_def::sl_ind](#)
- integer, dimension(:), allocatable, public [sf_def::sl_rind](#)
Resolved reduction indices.
- integer, public [sf_def::n_unres](#)
Number of unresolved variables.
- integer, public [sf_def::n_res](#)
Number of resolved variables.
- integer, dimension(:,:), allocatable, public [sf_def::bar](#)
- integer, dimension(:,:), allocatable, public [sf_def::bau](#)
- integer, dimension(:,:), allocatable, public [sf_def::bor](#)
- integer, dimension(:,:), allocatable, public [sf_def::bou](#)
Filter matrices.

9.31 sqrt_mod.f90 File Reference

Modules

- module [sqrt_mod](#)
Utility module with various routine to compute matrix square root.

Functions/Subroutines

- subroutine, public [sqrt_mod::init_sqrt](#)
- subroutine, public [sqrt_mod::sqrtm](#) (A, sqA, info, info_triu, bs)
Routine to compute a real square-root of a matrix.
- logical function [sqrt_mod::selectev](#) (a, b)
- subroutine [sqrt_mod::sqrtm_triu](#) (A, sqA, info, bs)
- subroutine [sqrt_mod::csqrtm_triu](#) (A, sqA, info, bs)
- subroutine [sqrt_mod::rsf2csf](#) (T, Z, Tz, Zz)
- subroutine, public [sqrt_mod::chol](#) (A, sqA, info)
Routine to perform a Cholesky decomposition.
- subroutine, public [sqrt_mod::sqrtm_svd](#) (A, sqA, info, info_triu, bs)
Routine to compute a real square-root of a matrix via a SVD decomposition.

Variables

- real(kind=8), dimension(:), allocatable [sqrt_mod::work](#)
- integer [sqrt_mod::lwork](#)
- real(kind=8), parameter [sqrt_mod::real_eps](#) = 2.2204460492503131e-16

9.32 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](#)

9.33 stoch_mod.f90 File Reference

Modules

- module [stoch_mod](#)
Utility module containing the stochastic related routines.

Functions/Subroutines

- real(kind=8) function, public [stoch_mod::gasdev](#) ()
- subroutine, public [stoch_mod::stoch_vec](#) (dW)
Routine to fill a vector with standard Gaussian noise process values.
- subroutine, public [stoch_mod::stoch_atm_vec](#) (dW)
routine to fill the atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_mod::stoch_atm_res_vec](#) (dW)
routine to fill the resolved atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_mod::stoch_atm_unres_vec](#) (dW)
routine to fill the unresolved atmospheric component of a vector with standard gaussian noise process values
- subroutine, public [stoch_mod::stoch_oc_vec](#) (dW)
routine to fill the oceanic component of a vector with standard gaussian noise process values
- subroutine, public [stoch_mod::stoch_oc_res_vec](#) (dW)
routine to fill the resolved oceanic component of a vector with standard gaussian noise process values
- subroutine, public [stoch_mod::stoch_oc_unres_vec](#) (dW)
routine to fill the unresolved oceanic component of a vector with standard gaussian noise process values

Variables

- integer [stoch_mod::iset](#) =0
- real(kind=8) [stoch_mod::gset](#)

9.34 stoch_params.f90 File Reference

Modules

- module [stoch_params](#)
The stochastic models parameters module.

Functions/Subroutines

- subroutine [stoch_params::init_stoch_params](#)
Stochastic parameters initialization routine.

Variables

- real(kind=8) [stoch_params::mnuti](#)
Multiplicative noise update time interval.
- real(kind=8) [stoch_params::t_trans_stoch](#)
Transient time period of the stochastic model evolution.
- real(kind=8) [stoch_params::q_ar](#)
Atmospheric resolved component noise amplitude.
- real(kind=8) [stoch_params::q_au](#)
Atmospheric unresolved component noise amplitude.
- real(kind=8) [stoch_params::q_or](#)
Oceanic resolved component noise amplitude.

- real(kind=8) [stoch_params::q_ou](#)
Oceanic unresolved component noise amplitude.
- real(kind=8) [stoch_params::dtm](#)
Square root of the timestep.
- real(kind=8) [stoch_params::eps_pert](#)
Perturbation parameter for the coupling.
- real(kind=8) [stoch_params::tdelta](#)
Time separation parameter.
- real(kind=8) [stoch_params::muti](#)
Memory update time interval.
- real(kind=8) [stoch_params::meml](#)
Time over which the memory kernel is integrated.
- real(kind=8) [stoch_params::t_trans_mem](#)
Transient time period to initialize the memory term.
- character(len=4) [stoch_params::x_int_mode](#)
Integration mode for the resolved component.
- real(kind=8) [stoch_params::dts](#)
Intrinsic resolved dynamics time step.
- integer [stoch_params::mems](#)
Number of steps in the memory kernel integral.
- real(kind=8) [stoch_params::dtsn](#)
Square root of the intrinsic resolved dynamics time step.
- real(kind=8) [stoch_params::maxint](#)
Upper integration limit of the correlations.
- character(len=4) [stoch_params::load_mode](#)
Loading mode for the correlations.
- character(len=4) [stoch_params::int_corr_mode](#)
Correlation integration mode.
- character(len=4) [stoch_params::mode](#)
Stochastic mode parameter.

9.35 tensor.f90 File Reference

Data Types

- type [tensor::coolist_elem](#)
Coordinate list element type. Elementary elements of the sparse tensors.
- type [tensor::coolist_elem4](#)
4d coordinate list element type. Elementary elements of the 4d sparse tensors.
- type [tensor::coolist](#)
Coordinate list. Type used to represent the sparse tensor.
- type [tensor::coolist4](#)
4d coordinate list. Type used to represent the rank-4 sparse tensor.

Modules

- module [tensor](#)
Tensor utility module.

Functions/Subroutines

- subroutine, public [tensor::copy_tensor](#) (src, dst)
Routine to copy a rank-3 tensor.
- subroutine, public [tensor::add_to_tensor](#) (src, dst)
Routine to add a rank-3 tensor to another one.
- subroutine, public [tensor::add_matc_to_tensor](#) (i, src, dst)
Routine to add a matrix to a rank-3 tensor.
- subroutine, public [tensor::add_matc_to_tensor4](#) (i, j, src, dst)
Routine to add a matrix to a rank-4 tensor.
- subroutine, public [tensor::add_vec_jk_to_tensor](#) (j, k, src, dst)
Routine to add a vector to a rank-3 tensor.
- subroutine, public [tensor::add_vec_ikl_to_tensor4_perm](#) (i, k, l, src, dst)
Routine to add a vector to a rank-4 tensor plus permutation.
- subroutine, public [tensor::add_vec_ikl_to_tensor4](#) (i, k, l, src, dst)
Routine to add a vector to a rank-4 tensor.
- subroutine, public [tensor::add_vec_ijk_to_tensor4](#) (i, j, k, src, dst)
Routine to add a vector to a rank-4 tensor.
- subroutine, public [tensor::mat_to_coo](#) (src, dst)
Routine to convert a matrix to a rank-3 tensor.
- subroutine, public [tensor::tensor_to_coo](#) (src, dst)
Routine to convert a rank-3 tensor from matrix to coolist representation.
- subroutine, public [tensor::tensor4_to_coo4](#) (src, dst)
Routine to convert a rank-4 tensor from matrix to coolist representation.
- subroutine, public [tensor::print_tensor](#) (t)
Routine to print a rank 3 tensor coolist.
- subroutine, public [tensor::print_tensor4](#) (t)
Routine to print a rank-4 tensor coolist.
- subroutine, public [tensor::sparse_mul3](#) (coolist_ijk, arr_j, arr_k, res)
Sparse multiplication of a rank-3 tensor coolist with two vectors: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} a_j b_k$.
- subroutine, public [tensor::sparse_mul3_mat](#) (coolist_ijk, arr_k, res)
Sparse multiplication of a rank-3 tensor coolist with a vector: $\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} b_k$. Its output is a matrix.
- subroutine, public [tensor::sparse_mul4](#) (coolist_ijkl, arr_j, arr_k, arr_l, res)
Sparse multiplication of a rank-4 tensor coolist with three vectors: $\sum_{j,k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} a_j b_k c_l$.
- subroutine, public [tensor::sparse_mul4_mat](#) (coolist_ijkl, arr_k, arr_l, res)
Sparse multiplication of a tensor with two vectors: $\sum_{k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} b_k c_l$.
- subroutine, public [tensor::jsparse_mul](#) (coolist_ijk, arr_j, jcoo_ij)
Sparse multiplication of two tensors to determine the Jacobian:

$$J_{i,j} = \sum_{k=0}^{ndim} (\mathcal{T}_{i,j,k} + \mathcal{T}_{i,k,j}) a_k.$$
It's implemented slightly differently: for every $\mathcal{T}_{i,j,k}$, we add to $J_{i,j}$ as follows:

$$J_{i,j} = J_{i,j} + \mathcal{T}_{i,j,k} a_k J_{i,k} = J_{i,k} + \mathcal{T}_{i,j,k} a_j$$
This version return a coolist (sparse tensor).
- subroutine, public [tensor::jsparse_mul_mat](#) (coolist_ijk, arr_j, jcoo_ij)

Sparse multiplication of two tensors to determine the Jacobian:

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

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

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

This version return a matrix.

- subroutine, public [tensor::sparse_mul2_j](#) (coolist_ijk, arr_j, res)

Sparse multiplication of a 3d sparse tensor with a vectors: $\sum_{j=0}^{ndim} \mathcal{T}_{i,j,k} a_j$.

- subroutine, public [tensor::sparse_mul2_k](#) (coolist_ijk, arr_k, res)

Sparse multiplication of a rank-3 sparse tensor coolist with a vector: $\sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} a_k$.

- subroutine, public [tensor::simplify](#) (tensor)

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

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

- subroutine, public [tensor::coo_to_mat_ik](#) (src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix with i and k indices.

- subroutine, public [tensor::coo_to_mat_ij](#) (src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix with i and j indices.

- subroutine, public [tensor::coo_to_mat_i](#) (i, src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix.

- subroutine, public [tensor::coo_to_vec_jk](#) (j, k, src, dst)

Routine to convert a rank-3 tensor coolist component into a vector.

- subroutine, public [tensor::coo_to_mat_j](#) (j, src, dst)

Routine to convert a rank-3 tensor coolist component into a matrix.

- subroutine, public [tensor::sparse_mul4_with_mat_jl](#) (coolist_ijkl, mat_jl, res)

Sparse multiplication of a rank-4 tensor coolist with a matrix: $\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{j,l}$.

- subroutine, public [tensor::sparse_mul4_with_mat_kl](#) (coolist_ijkl, mat_kl, res)

Sparse multiplication of a rank-4 tensor coolist with a matrix: $\sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{k,l}$.

- subroutine, public [tensor::sparse_mul3_with_mat](#) (coolist_ijk, mat_jk, res)

Sparse multiplication of a rank-3 tensor coolist with a matrix: $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} m_{j,k}$.

- subroutine, public [tensor::matc_to_coo](#) (src, dst)

Routine to convert a matrix to a rank-3 tensor.

- subroutine, public [tensor::scal_mul_coo](#) (s, t)

Routine to multiply a rank-3 tensor by a scalar.

- logical function, public [tensor::tensor_empty](#) (t)

Test if a rank-3 tensor coolist is empty.

- logical function, public [tensor::tensor4_empty](#) (t)

Test if a rank-4 tensor coolist is empty.

- subroutine, public [tensor::load_tensor4_from_file](#) (s, t)

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

- subroutine, public [tensor::write_tensor4_to_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.

9.36 test_aotensor.f90 File Reference

Functions/Subroutines

- program [test_aotensor](#)
Small program to print the inner products.

9.36.1 Function/Subroutine Documentation

9.36.1.1 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.

9.37 test_corr.f90 File Reference

Functions/Subroutines

- program [test_corr](#)
Small program to print the correlation and covariance matrices.

9.37.1 Function/Subroutine Documentation

9.37.1.1 program test_corr ()

Small program to print the correlation and covariance matrices.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_corr.f90.

9.38 test_corr_tensor.f90 File Reference

Functions/Subroutines

- program [test_corr_tensor](#)
Small program to print the time correlations tensors.

9.38.1 Function/Subroutine Documentation

9.38.1.1 program test_corr_tensor ()

Small program to print the time correlations tensors.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_corr_tensor.f90.

9.39 test_dec_tensor.f90 File Reference

Functions/Subroutines

- program [test_dec_tensor](#)
Small program to print the decomposed tensors.

9.39.1 Function/Subroutine Documentation

9.39.1.1 program test_dec_tensor ()

Small program to print the decomposed tensors.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_dec_tensor.f90.

9.40 test_inprod_analytic.f90 File Reference

Functions/Subroutines

- program [inprod_analytic_test](#)
Small program to print the inner products.

9.40.1 Function/Subroutine Documentation

9.40.1.1 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.

9.41 test_MAR.f90 File Reference

Functions/Subroutines

- program [test_mar](#)
Small program to test the Multivariate AutoRegressive model.

9.41.1 Function/Subroutine Documentation

9.41.1.1 program test_mar ()

Small program to test the Multivariate AutoRegressive model.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_MAR.f90.

9.42 test_memory.f90 File Reference

Functions/Subroutines

- program [test_memory](#)
Small program to test the WL memory module.

9.42.1 Function/Subroutine Documentation

9.42.1.1 program test_memory ()

Small program to test the WL memory module.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_memory.f90.

9.43 test_MTV_int_tensor.f90 File Reference

Functions/Subroutines

- program [test_mtv_int_tensor](#)
Small program to print the MTV integrated tensors.

9.43.1 Function/Subroutine Documentation

9.43.1.1 program test_mtv_int_tensor ()

Small program to print the MTV integrated tensors.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_MTV_int_tensor.f90.

9.44 test_MTV_sigma_tensor.f90 File Reference

Functions/Subroutines

- program [test_sigma](#)
Small program to test the MTV noise sigma matrices.

9.44.1 Function/Subroutine Documentation

9.44.1.1 program test_sigma ()

Small program to test the MTV noise sigma matrices.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_MTV_sigma_tensor.f90.

9.45 test_sqrtm.f90 File Reference

Functions/Subroutines

- program [test_sqrtm](#)
Small program to test the matrix square-root module.

9.45.1 Function/Subroutine Documentation

9.45.1.1 program test_sqrtm ()

Small program to test the matrix square-root module.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test_sqrtm.f90.

9.46 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.

9.46.1 Function/Subroutine Documentation

9.46.1.1 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.

9.47 test_WL_tensor.f90 File Reference

Functions/Subroutines

- program [test_wl_tensor](#)
Small program to print the WL tensors.

9.47.1 Function/Subroutine Documentation

9.47.1.1 program test_wl_tensor ()

Small program to print the WL tensors.

Copyright

2017 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 11 of file test_WL_tensor.f90.

9.48 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, deltat, buf)
Tendencies for the AD of MAOOAM in point ystar for perturbation deltat.
- subroutine, public [tl_ad_tensor::tl](#) (t, ystar, deltat, buf)
Tendencies for the TL of MAOOAM in point ystar for perturbation deltat.

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.

9.49 util.f90 File Reference

Modules

- module `util`
Utility module.

Functions/Subroutines

- character(len=20) function, public `util::str` (k)
Convert an integer to string.
- character(len=40) function, public `util::rstr` (x, fm)
Convert a real to string with a given format.
- subroutine, public `util::init_random_seed` ()
Random generator initialization routine.
- integer function `lcg` (s)
- subroutine, public `util::init_one` (A)
Initialize a square matrix A as a unit matrix.
- real(kind=8) function, public `util::mat_trace` (A)
- real(kind=8) function, public `util::mat_contract` (A, B)
- subroutine, public `util::choldc` (a, p)
- subroutine, public `util::printmat` (A)
- subroutine, public `util::cprintmat` (A)
- real(kind=8) function, dimension(size(a, 1), size(a, 2)), public `util::invmat` (A)
- subroutine, public `util::triu` (A, T)
- subroutine, public `util::diag` (A, d)
- subroutine, public `util::cdiag` (A, d)
- integer function, public `util::floordiv` (i, j)
- subroutine, public `util::reduce` (A, Ared, n, ind, rind)
- subroutine, public `util::ireduce` (A, Ared, n, ind, rind)
- subroutine, public `util::vector_outer` (u, v, A)

9.49.1 Function/Subroutine Documentation

9.49.1.1 integer function init_random_seed::lcg (integer(int64) s)

Definition at line 85 of file util.f90.

```

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

```

9.50 WL_tensor.f90 File Reference

Modules

- module [wl_tensor](#)
The WL tensors used to integrate the model.

Functions/Subroutines

- subroutine, public [wl_tensor::init_wl_tensor](#)
Subroutine to initialise the WL tensor.

Variables

- real(kind=8), dimension(:), allocatable, public [wl_tensor::m1](#)
First component of the M1 term.
- type(coolist), dimension(:), allocatable, public [wl_tensor::m12](#)
Second component of the M1 term.
- real(kind=8), dimension(:), allocatable, public [wl_tensor::m13](#)
Third component of the M1 term.
- real(kind=8), dimension(:), allocatable, public [wl_tensor::m1tot](#)
Total M_1 vector.
- type(coolist), dimension(:), allocatable, public [wl_tensor::m21](#)
First tensor of the M2 term.
- type(coolist), dimension(:), allocatable, public [wl_tensor::m22](#)
Second tensor of the M2 term.
- type(coolist), dimension(:, :), allocatable, public [wl_tensor::l1](#)
First linear tensor.
- type(coolist), dimension(:, :), allocatable, public [wl_tensor::l2](#)
Second linear tensor.
- type(coolist), dimension(:, :), allocatable, public [wl_tensor::l4](#)
Fourth linear tensor.
- type(coolist), dimension(:, :), allocatable, public [wl_tensor::l5](#)
Fifth linear tensor.

- type(coolist), dimension(:,:), allocatable, public [wl_tensor::ltot](#)
Total linear tensor.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b1](#)
First quadratic tensor.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b2](#)
Second quadratic tensor.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b3](#)
Third quadratic tensor.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b4](#)
Fourth quadratic tensor.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b14](#)
Joint 1st and 4th tensors.
- type(coolist), dimension(:,:), allocatable, public [wl_tensor::b23](#)
Joint 2nd and 3rd tensors.
- type(coolist4), dimension(:,:), allocatable, public [wl_tensor::mtot](#)
Tensor for the cubic terms.
- real(kind=8), dimension(:), allocatable [wl_tensor::dumb_vec](#)
Dummy vector.
- real(kind=8), dimension(:,:), allocatable [wl_tensor::dumb_mat1](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [wl_tensor::dumb_mat2](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [wl_tensor::dumb_mat3](#)
Dummy matrix.
- real(kind=8), dimension(:,:), allocatable [wl_tensor::dumb_mat4](#)
Dummy matrix.
- logical, public [wl_tensor::m12def](#)
- logical, public [wl_tensor::m21def](#)
- logical, public [wl_tensor::m22def](#)
- logical, public [wl_tensor::ldef](#)
- logical, public [wl_tensor::b14def](#)
- logical, public [wl_tensor::b23def](#)
- logical, public [wl_tensor::mdef](#)
Boolean to (de)activate the computation of the terms.

Index

- a
 - aotensor_def, 22
 - inprod_analytic::atm_tensors, 215
- acc
 - stat, 148
- ad
 - tl_ad_tensor, 191
- ad_add_count
 - tl_ad_tensor, 191
- ad_add_count_ref
 - tl_ad_tensor, 192
- ad_coeff
 - tl_ad_tensor, 192
- ad_coeff_ref
 - tl_ad_tensor, 193
- ad_step
 - tl_ad_integrator, 188
- add_count
 - aotensor_def, 22
- add_matc_to_tensor
 - tensor, 163
- add_matc_to_tensor4
 - tensor, 163
- add_to_tensor
 - tensor, 165
- add_vec_ijk_to_tensor4
 - tensor, 166
- add_vec_ikl_to_tensor4
 - tensor, 167
- add_vec_ikl_to_tensor4_perm
 - tensor, 168
- add_vec_jk_to_tensor
 - tensor, 169
- adtensor
 - tl_ad_tensor, 198
- ams
 - params, 100
- anoise
 - rk2_mtv_integrator, 115
 - rk2_ss_integrator, 121
 - rk2_stoch_integrator, 125
 - rk2_wl_integrator, 130
- aotensor
 - aotensor_def, 25
- aotensor_def, 21
 - a, 22
 - add_count, 22
 - aotensor, 25
 - coeff, 22
 - compute_aotensor, 23
 - count_elems, 25
 - init_aotensor, 23
 - kdelta, 24
 - psi, 24
 - real_eps, 25
 - t, 24
 - theta, 24
- aotensor_def.f90, 225
- atmos
 - inprod_analytic, 66
- awavenum
 - inprod_analytic, 66
- b
 - inprod_analytic::atm_tensors, 215
- b1
 - inprod_analytic, 55
 - mtv_int_tensor, 91
 - wl_tensor, 208
- b14
 - wl_tensor, 208
- b14def
 - wl_tensor, 208
- b2
 - inprod_analytic, 55
 - mtv_int_tensor, 92
 - wl_tensor, 208
- b23
 - wl_tensor, 209
- b23def
 - wl_tensor, 209
- b3
 - wl_tensor, 209
- b4
 - wl_tensor, 209
- bar
 - sf_def, 135
- bau
 - sf_def, 135
- betp
 - params, 100
- bor
 - sf_def, 135
- bou
 - sf_def, 135
- btot
 - mtv_int_tensor, 92
- buf_f0
 - integrator, 78

- rk2_mtv_integrator, [115](#)
- rk2_ss_integrator, [121](#)
- rk2_stoch_integrator, [125](#)
- rk2_wl_integrator, [130](#)
- tl_ad_integrator, [189](#)
- buf_f1
 - integrator, [78](#)
 - rk2_mtv_integrator, [115](#)
 - rk2_ss_integrator, [121](#)
 - rk2_stoch_integrator, [125](#)
 - rk2_wl_integrator, [131](#)
 - tl_ad_integrator, [189](#)
- buf_g
 - rk2_mtv_integrator, [115](#)
- buf_ka
 - integrator, [78](#)
 - tl_ad_integrator, [189](#)
- buf_kb
 - integrator, [78](#)
 - tl_ad_integrator, [189](#)
- buf_m
 - memory, [86](#)
 - rk2_wl_integrator, [131](#)
- buf_m1
 - rk2_wl_integrator, [131](#)
- buf_m2
 - rk2_wl_integrator, [131](#)
- buf_m3
 - memory, [86](#)
 - rk2_wl_integrator, [131](#)
- buf_m3s
 - rk2_wl_integrator, [131](#)
- buf_y
 - mar, [81](#)
- buf_y1
 - integrator, [78](#)
 - rk2_mtv_integrator, [115](#)
 - rk2_ss_integrator, [121](#)
 - rk2_stoch_integrator, [125](#)
 - rk2_wl_integrator, [131](#)
 - tl_ad_integrator, [190](#)
- bxxx
 - dec_tensor, [48](#)
- bxyy
 - dec_tensor, [48](#)
- bxyx
 - dec_tensor, [48](#)
- byxx
 - dec_tensor, [48](#)
- byxy
 - dec_tensor, [48](#)
- byyy
 - dec_tensor, [48](#)
- c
 - inprod_analytic::atm_tensors, [215](#)
 - inprod_analytic::ocean_tensors, [221](#)
- CLAIM
 - LICENSE.txt, [234](#)
- CONTRACT
 - LICENSE.txt, [234](#)
- ca
 - params, [100](#)
- calculate_a
 - inprod_analytic, [56](#)
- calculate_b
 - inprod_analytic, [56](#)
- calculate_c_atm
 - inprod_analytic, [56](#)
- calculate_c_oc
 - inprod_analytic, [57](#)
- calculate_d
 - inprod_analytic, [58](#)
- calculate_g
 - inprod_analytic, [58](#)
- calculate_k
 - inprod_analytic, [59](#)
- calculate_m
 - inprod_analytic, [60](#)
- calculate_n
 - inprod_analytic, [60](#)
- calculate_o
 - inprod_analytic, [61](#)
- calculate_s
 - inprod_analytic, [62](#)
- calculate_w
 - inprod_analytic, [62](#)
- cdiag
 - util, [199](#)
- charge
 - LICENSE.txt, [234](#)
- chol
 - sqrt_mod, [141](#)
- choldc
 - util, [199](#)
- co
 - params, [100](#)
- coeff
 - aotensor_def, [22](#)
- comp_corrnt
 - int_corr, [72](#)
- compg
 - rk2_mtv_integrator, [112](#)
- compute_adtensor
 - tl_ad_tensor, [193](#)
- compute_adtensor_ref
 - tl_ad_tensor, [193](#)
- compute_aotensor
 - aotensor_def, [23](#)
- compute_m1
 - rk2_wl_integrator, [127](#)
- compute_m2
 - rk2_wl_integrator, [128](#)
- compute_m3
 - memory, [84](#)
- compute_mult
 - rk2_mtv_integrator, [115](#)

- compute_mult_sigma
 - sigma, [137](#)
- compute_tltensor
 - tl_ad_tensor, [194](#)
- conditions
 - LICENSE.txt, [234](#)
- coo_to_mat_i
 - tensor, [170](#)
- coo_to_mat_ij
 - tensor, [170](#)
- coo_to_mat_ik
 - tensor, [171](#)
- coo_to_mat_j
 - tensor, [171](#)
- coo_to_vec_jk
 - tensor, [172](#)
- copy
 - LICENSE.txt, [234](#)
- copy_tensor
 - tensor, [172](#)
- corr2int
 - int_corr, [75](#)
- corr_i
 - corrmod, [37](#)
- corr_i_full
 - corrmod, [37](#)
- corr_ij
 - corrmod, [37](#)
- corr_tensor, [25](#)
 - dumb_mat1, [27](#)
 - dumb_mat2, [27](#)
 - dumb_vec, [27](#)
 - dy, [28](#)
 - dyy, [28](#)
 - expm, [28](#)
 - init_corr_tensor, [26](#)
 - ydy, [28](#)
 - ydy, [28](#)
 - yy, [29](#)
- corr_tensor.f90, [226](#)
- corrcomp
 - corrmod, [37](#)
- corrcomp_from_def
 - corrmod, [30](#)
- corrcomp_from_fit
 - corrmod, [34](#)
- corrcomp_from_spline
 - corrmod, [34](#)
- corrint
 - int_corr, [75](#)
- corrmod, [29](#)
 - corr_i, [37](#)
 - corr_i_full, [37](#)
 - corr_ij, [37](#)
 - corrcomp, [37](#)
 - corrcomp_from_def, [30](#)
 - corrcomp_from_fit, [34](#)
 - corrcomp_from_spline, [34](#)
- fs, [35](#)
- init_corr, [35](#)
- inv_corr_i, [38](#)
- inv_corr_i_full, [38](#)
- khi, [38](#)
- klo, [38](#)
- mean, [38](#)
- mean_full, [38](#)
- nspl, [39](#)
- splint, [36](#)
- xa, [39](#)
- y2, [39](#)
- ya, [39](#)
- corrmod.f90, [226](#)
- count_elems
 - aotensor_def, [25](#)
 - tl_ad_tensor, [198](#)
- cpa
 - params, [101](#)
- cpo
 - params, [101](#)
- cprintmat
 - util, [200](#)
- csqrtm_triu
 - sqrt_mod, [142](#)
- d
 - inprod_analytic::atm_tensors, [216](#)
 - params, [101](#)
- deallocate_inprod
 - inprod_analytic, [63](#)
- dec_tensor, [40](#)
 - bxxx, [48](#)
 - bxy, [48](#)
 - bxy, [48](#)
 - byxx, [48](#)
 - byxy, [48](#)
 - byyy, [48](#)
 - dumb, [49](#)
 - ff_tensor, [49](#)
 - fs_tensor, [49](#)
 - hx, [49](#)
 - hy, [49](#)
 - init_dec_tensor, [41](#)
 - init_sub_tensor, [45](#)
 - lxx, [50](#)
 - lxy, [50](#)
 - lyx, [50](#)
 - lyy, [50](#)
 - reorder, [46](#)
 - sf_tensor, [50](#)
 - ss_tensor, [51](#)
 - ss_tl_tensor, [51](#)
 - suppress_and, [46](#)
 - suppress_or, [47](#)
- dec_tensor.f90, [227](#)
- delta
 - inprod_analytic, [64](#)
- diag

- util, 200
- distribute
 - LICENSE.txt, 234
- doc/gen_doc.md, 229
- doc/sto_doc.md, 229
- doc/tl_ad_doc.md, 229
- dp
 - params, 101
- dt
 - params, 101
- dtn
 - stoch_params, 156
- dts
 - stoch_params, 156
- dtsn
 - stoch_params, 156
- dumb
 - dec_tensor, 49
- dumb_mat1
 - corr_tensor, 27
 - mtv_int_tensor, 92
 - sigma, 139
 - wl_tensor, 209
- dumb_mat2
 - corr_tensor, 27
 - mtv_int_tensor, 92
 - sigma, 139
 - wl_tensor, 209
- dumb_mat3
 - mtv_int_tensor, 92
 - sigma, 139
 - wl_tensor, 210
- dumb_mat4
 - mtv_int_tensor, 93
 - sigma, 139
 - wl_tensor, 210
- dumb_vec
 - corr_tensor, 27
 - mtv_int_tensor, 93
 - wl_tensor, 210
- dw
 - mar, 81
 - rk2_mtv_integrator, 116
- dwar
 - rk2_mtv_integrator, 116
 - rk2_ss_integrator, 121
 - rk2_stoch_integrator, 125
 - rk2_wl_integrator, 131
- dwau
 - rk2_mtv_integrator, 116
 - rk2_stoch_integrator, 125
 - rk2_wl_integrator, 132
- dwmult
 - rk2_mtv_integrator, 116
- dwor
 - rk2_mtv_integrator, 116
 - rk2_ss_integrator, 122
 - rk2_stoch_integrator, 126
- rk2_wl_integrator, 132
- dwou
 - rk2_mtv_integrator, 116
 - rk2_stoch_integrator, 126
 - rk2_wl_integrator, 132
- dy
 - corr_tensor, 28
- dyy
 - corr_tensor, 28
- elems
 - tensor::coolist, 218
 - tensor::coolist4, 219
- eps_pert
 - stoch_params, 156
- epsa
 - params, 102
- exists
 - ic_def, 53
 - sf_def, 135
- expm
 - corr_tensor, 28
- f0
 - params, 102
- FROM
 - LICENSE.txt, 234
- ff_tensor
 - dec_tensor, 49
- files
 - LICENSE.txt, 233
- flambda
 - inprod_analytic, 64
- floordiv
 - util, 200
- fs
 - corrmod, 35
- fs_tensor
 - dec_tensor, 49
- full_step
 - rk2_mtv_integrator, 112
 - rk2_wl_integrator, 128
- func_ij
 - int_corr, 74
- func_ijkl
 - int_corr, 74
- g
 - inprod_analytic::atm_tensors, 216
 - params, 102
 - rk2_mtv_integrator, 116
- ga
 - params, 102
- gasdev
 - stoch_mod, 151
- go
 - params, 102
- gp
 - params, 103

- gset
 - stoch_mod, 154
- h
 - inprod_analytic::atm_wavenum, 216
 - inprod_analytic::ocean_wavenum, 223
 - params, 103
- h1
 - mtv_int_tensor, 93
- h2
 - mtv_int_tensor, 93
- h3
 - mtv_int_tensor, 93
- htot
 - mtv_int_tensor, 94
- hx
 - dec_tensor, 49
- hy
 - dec_tensor, 49
- i
 - stat, 150
- IMPLIED
 - LICENSE.txt, 235
- ic
 - ic_def, 53
- ic_def, 51
 - exists, 53
 - ic, 53
 - load_ic, 52
- ic_def.f90, 229
- ind
 - sf_def, 135
- ind1
 - sigma, 139
- ind2
 - sigma, 139
- init_adtensor
 - tl_ad_tensor, 194
- init_adtensor_ref
 - tl_ad_tensor, 194
- init_aotensor
 - aotensor_def, 23
- init_corr
 - corrmod, 35
- init_corr_tensor
 - corr_tensor, 26
- init_corrint
 - int_corr, 74
- init_dec_tensor
 - dec_tensor, 41
- init_g
 - rk2_mtv_integrator, 113
- init_inprod
 - inprod_analytic, 64
- init_integrator
 - integrator, 77
 - rk2_mtv_integrator, 113
 - rk2_stoch_integrator, 123
 - rk2_wl_integrator, 128
- init_mar
 - mar, 80
- init_memory
 - memory, 85
- init_mtv_int_tensor
 - mtv_int_tensor, 89
- init_nml
 - params, 99
- init_noise
 - rk2_mtv_integrator, 113
- init_one
 - util, 200
- init_params
 - params, 99
- init_random_seed
 - util, 201
- init_sigma
 - sigma, 138
- init_sqrt
 - sqrt_mod, 143
- init_ss_integrator
 - rk2_ss_integrator, 119
- init_stat
 - stat, 148
- init_stoch_params
 - stoch_params, 156
- init_sub_tensor
 - dec_tensor, 45
- init_tl_ad_integrator
 - tl_ad_integrator, 188
- init_tltensor
 - tl_ad_tensor, 195
- init_wl_tensor
 - wl_tensor, 205
- inprod_analytic, 54
 - atmos, 66
 - awavenum, 66
 - b1, 55
 - b2, 55
 - calculate_a, 56
 - calculate_b, 56
 - calculate_c_atm, 56
 - calculate_c_oc, 57
 - calculate_d, 58
 - calculate_g, 58
 - calculate_k, 59
 - calculate_m, 60
 - calculate_n, 60
 - calculate_o, 61
 - calculate_s, 62
 - calculate_w, 62
 - deallocate_inprod, 63
 - delta, 64
 - flambda, 64
 - init_inprod, 64
 - ocean, 66
 - owavenum, 67

- s1, [65](#)
- s2, [65](#)
- s3, [66](#)
- s4, [66](#)
- inprod_analytic.f90, [229](#)
- inprod_analytic::atm_tensors, [215](#)
 - a, [215](#)
 - b, [215](#)
 - c, [215](#)
 - d, [216](#)
 - g, [216](#)
 - s, [216](#)
- inprod_analytic::atm_wavenum, [216](#)
 - h, [216](#)
 - m, [216](#)
 - nx, [217](#)
 - ny, [217](#)
 - p, [217](#)
 - typ, [217](#)
- inprod_analytic::ocean_tensors, [221](#)
 - c, [221](#)
 - k, [221](#)
 - m, [222](#)
 - n, [222](#)
 - o, [222](#)
 - w, [222](#)
- inprod_analytic::ocean_wavenum, [222](#)
 - h, [223](#)
 - nx, [223](#)
 - ny, [223](#)
 - p, [223](#)
- inprod_analytic_test
 - test_inprod_analytic.f90, [259](#)
- int_comp, [67](#)
 - integrate, [67](#)
 - midexp, [68](#)
 - midpnt, [68](#)
 - polint, [69](#)
 - qromb, [70](#)
 - qromo, [70](#)
 - trapzd, [71](#)
- int_comp.f90, [231](#)
- int_corr, [71](#)
 - comp_corrint, [72](#)
 - corr2int, [75](#)
 - corrint, [75](#)
 - func_ij, [74](#)
 - func_ijkl, [74](#)
 - init_corrint, [74](#)
 - oi, [75](#)
 - oj, [75](#)
 - ok, [75](#)
 - ol, [75](#)
 - real_eps, [75](#)
- int_corr.f90, [231](#)
- int_corr_mode
 - stoch_params, [157](#)
- int_tensor
 - rk2_stoch_integrator, [126](#)
- integrate
 - int_comp, [67](#)
- integrator, [76](#)
 - buf_f0, [78](#)
 - buf_f1, [78](#)
 - buf_ka, [78](#)
 - buf_kb, [78](#)
 - buf_y1, [78](#)
 - init_integrator, [77](#)
 - step, [77](#)
 - tendencies, [77](#)
- inv_corr_i
 - corrmod, [38](#)
- inv_corr_i_full
 - corrmod, [38](#)
- invmat
 - util, [201](#)
- ireduce
 - util, [201](#)
- iset
 - stoch_mod, [154](#)
- iter
 - stat, [149](#)
- j
 - tensor::coolist_elem, [219](#)
 - tensor::coolist_elem4, [220](#)
- jacobian
 - tl_ad_tensor, [195](#)
- jacobian_mat
 - tl_ad_tensor, [196](#)
- jsparse_mul
 - tensor, [173](#)
- jsparse_mul_mat
 - tensor, [173](#)
- k
 - inprod_analytic::ocean_tensors, [221](#)
 - params, [103](#)
 - tensor::coolist_elem, [219](#)
 - tensor::coolist_elem4, [220](#)
- KIND
 - LICENSE.txt, [235](#)
- kd
 - params, [103](#)
- kdelta
 - aotensor_def, [24](#)
- kdp
 - params, [103](#)
- khi
 - corrmod, [38](#)
- klo
 - corrmod, [38](#)
- kp
 - params, [104](#)
- l
 - params, [104](#)

- tensor::coolist_elem4, 221
- l1
 - mtv_int_tensor, 94
 - wl_tensor, 210
- l2
 - mtv_int_tensor, 94
 - wl_tensor, 210
- l3
 - mtv_int_tensor, 94
- l4
 - wl_tensor, 211
- l5
 - wl_tensor, 211
- LIABILITY
 - LICENSE.txt, 235
- LICENSE.txt, 232
 - CLAIM, 234
 - CONTRACT, 234
 - charge, 234
 - conditions, 234
 - copy, 234
 - distribute, 234
 - FROM, 234
 - files, 233
 - IMPLIED, 235
 - KIND, 235
 - LIABILITY, 235
 - License, 234
 - MERCHANTABILITY, 235
 - merge, 235
 - modify, 235
 - OTHERWISE, 235
 - publish, 236
 - restriction, 236
 - so, 236
 - Software, 236
 - sublicense, 236
 - use, 236
- lambda
 - params, 104
- lcg
 - util.f90, 264
- ldef
 - wl_tensor, 211
- License
 - LICENSE.txt, 234
- load_ic
 - ic_def, 52
- load_mode
 - stoch_params, 157
- load_sf
 - sf_def, 133
- load_tensor4_from_file
 - tensor, 174
- lpa
 - params, 104
- lpo
 - params, 104
- lr
 - params, 105
- lsbpa
 - params, 105
- lsbpo
 - params, 105
- ltot
 - mtv_int_tensor, 94
 - wl_tensor, 211
- lwork
 - sqrt_mod, 147
- lxx
 - dec_tensor, 50
- lxy
 - dec_tensor, 50
- lyx
 - dec_tensor, 50
- lyy
 - dec_tensor, 50
- m
 - inprod_analytic::atm_wavenum, 216
 - inprod_analytic::ocean_tensors, 222
 - stat, 150
- m11
 - wl_tensor, 211
- m12
 - wl_tensor, 211
- m12def
 - wl_tensor, 212
- m13
 - wl_tensor, 212
- m1tot
 - wl_tensor, 212
- m21
 - wl_tensor, 212
- m21def
 - wl_tensor, 212
- m22
 - wl_tensor, 212
- m22def
 - wl_tensor, 213
- MAR.f90, 238
- MERCHANTABILITY
 - LICENSE.txt, 235
- MTV_int_tensor.f90, 240
- MTV_sigma_tensor.f90, 241
- maoam
 - maoam.f90, 237
 - maoam.f90, 236
 - maoam, 237
 - maoam_MTV.f90, 237
 - maoam_mtv, 237
 - maoam_WL.f90, 238
 - maoam_wl, 238
 - maoam_mtv
 - maoam_MTV.f90, 237
 - maoam_stoch
 - maoam_stoch.f90, 238

- maooam_stoch.f90, [237](#)
 - maooam_stoch, [238](#)
- maooam_wl
 - maooam_WL.f90, [238](#)
- mar, [79](#)
 - buf_y, [81](#)
 - dw, [81](#)
 - init_mar, [80](#)
 - mar_step, [80](#)
 - mar_step_red, [81](#)
 - ms, [82](#)
 - q, [82](#)
 - qred, [82](#)
 - rred, [82](#)
 - stoch_vec, [81](#)
 - w, [82](#)
 - wred, [82](#)
- mar_step
 - mar, [80](#)
- mar_step_red
 - mar, [81](#)
- mat_contract
 - util, [201](#)
- mat_to_coo
 - tensor, [175](#)
- mat_trace
 - util, [202](#)
- matc_to_coo
 - tensor, [175](#)
- maxint
 - stoch_params, [157](#)
- mdef
 - wl_tensor, [213](#)
- mean
 - corrmod, [38](#)
 - stat, [149](#)
- mean_full
 - corrmod, [38](#)
- meml
 - stoch_params, [157](#)
- memory, [83](#)
 - buf_m, [86](#)
 - buf_m3, [86](#)
 - compute_m3, [84](#)
 - init_memory, [85](#)
 - step, [86](#)
 - t_index, [86](#)
 - test_m3, [85](#)
 - x, [86](#)
 - xs, [87](#)
 - zs, [87](#)
- memory.f90, [239](#)
- mems
 - stoch_params, [157](#)
- merge
 - LICENSE.txt, [235](#)
- midexp
 - int_comp, [68](#)
- midpnt
 - int_comp, [68](#)
- mnuti
 - stoch_params, [158](#)
- mode
 - stoch_params, [158](#)
- modify
 - LICENSE.txt, [235](#)
- mprev
 - stat, [150](#)
- ms
 - mar, [82](#)
- mtmp
 - stat, [150](#)
- mtot
 - mtv_int_tensor, [95](#)
 - wl_tensor, [213](#)
- mtv_int_tensor, [87](#)
 - b1, [91](#)
 - b2, [92](#)
 - btot, [92](#)
 - dumb_mat1, [92](#)
 - dumb_mat2, [92](#)
 - dumb_mat3, [92](#)
 - dumb_mat4, [93](#)
 - dumb_vec, [93](#)
 - h1, [93](#)
 - h2, [93](#)
 - h3, [93](#)
 - htot, [94](#)
 - init_mtv_int_tensor, [89](#)
 - l1, [94](#)
 - l2, [94](#)
 - l3, [94](#)
 - ltot, [94](#)
 - mtot, [95](#)
 - q1, [95](#)
 - q2, [95](#)
 - utot, [95](#)
 - vtot, [95](#)
- mult
 - rk2_mtv_integrator, [117](#)
- muti
 - stoch_params, [158](#)
- n
 - inprod_analytic::ocean_tensors, [222](#)
 - params, [105](#)
- n1
 - sigma, [139](#)
- n2
 - sigma, [140](#)
- n_res
 - sf_def, [135](#)
- n_unres
 - sf_def, [136](#)
- natm
 - params, [105](#)
- nbatm

- params, 106
- nboc
 - params, 106
- ndim
 - params, 106
- nelems
 - tensor::coolist, 218
 - tensor::coolist4, 219
- noc
 - params, 106
- noise
 - rk2_mtv_integrator, 117
- noisemult
 - rk2_mtv_integrator, 117
- nspl
 - corrmod, 39
- nx
 - inprod_analytic::atm_wavenum, 217
 - inprod_analytic::ocean_wavenum, 223
- ny
 - inprod_analytic::atm_wavenum, 217
 - inprod_analytic::ocean_wavenum, 223
- o
 - inprod_analytic::ocean_tensors, 222
- OTHERWISE
 - LICENSE.txt, 235
- ocean
 - inprod_analytic, 66
- oi
 - int_corr, 75
- oj
 - int_corr, 75
- ok
 - int_corr, 75
- ol
 - int_corr, 75
- oms
 - params, 106
- owavenum
 - inprod_analytic, 67
- p
 - inprod_analytic::atm_wavenum, 217
 - inprod_analytic::ocean_wavenum, 223
- params, 96
 - ams, 100
 - betp, 100
 - ca, 100
 - co, 100
 - cpa, 101
 - cpo, 101
 - d, 101
 - dp, 101
 - dt, 101
 - epsa, 102
 - f0, 102
 - g, 102
 - ga, 102
 - go, 102
 - gp, 103
 - h, 103
 - init_nml, 99
 - init_params, 99
 - k, 103
 - kd, 103
 - kdp, 103
 - kp, 104
 - l, 104
 - lambda, 104
 - lpa, 104
 - lpo, 104
 - lr, 105
 - lsbpa, 105
 - lsbpo, 105
 - n, 105
 - natm, 105
 - nbatm, 106
 - nboc, 106
 - ndim, 106
 - noc, 106
 - oms, 106
 - phi0, 107
 - phi0_npi, 107
 - pi, 107
 - r, 107
 - rp, 107
 - rr, 108
 - rra, 108
 - sb, 108
 - sbpa, 108
 - sbpo, 108
 - sc, 109
 - scale, 109
 - sig0, 109
 - t_run, 109
 - t_trans, 109
 - ta0, 110
 - to0, 110
 - tw, 110
 - writeout, 110
- params.f90, 242
- phi0
 - params, 107
- phi0_npi
 - params, 107
- pi
 - params, 107
- polint
 - int_comp, 69
- print_tensor
 - tensor, 176
- print_tensor4
 - tensor, 176
- printmat
 - util, 202
- psi

- aotensor_def, [24](#)
- publish
 - LICENSE.txt, [236](#)
- q
 - mar, [82](#)
- q1
 - mtv_int_tensor, [95](#)
- q1fill
 - rk2_mtv_integrator, [117](#)
- q2
 - mtv_int_tensor, [95](#)
- q_ar
 - stoch_params, [158](#)
- q_au
 - stoch_params, [158](#)
- q_or
 - stoch_params, [159](#)
- q_ou
 - stoch_params, [159](#)
- qred
 - mar, [82](#)
- qromb
 - int_comp, [70](#)
- qromo
 - int_comp, [70](#)
- r
 - params, [107](#)
- real_eps
 - aotensor_def, [25](#)
 - int_corr, [75](#)
 - sqrt_mod, [147](#)
 - tensor, [186](#)
 - tl_ad_tensor, [198](#)
- reduce
 - util, [202](#)
- reorder
 - dec_tensor, [46](#)
- reset
 - stat, [149](#)
- restriction
 - LICENSE.txt, [236](#)
- rind
 - sf_def, [136](#)
- rind1
 - sigma, [140](#)
- rind2
 - sigma, [140](#)
- rk2_MTV_integrator.f90, [245](#)
- rk2_WL_integrator.f90, [248](#)
- rk2_integrator.f90, [245](#)
- rk2_mtv_integrator, [110](#)
 - anoise, [115](#)
 - buf_f0, [115](#)
 - buf_f1, [115](#)
 - buf_g, [115](#)
 - buf_y1, [115](#)
 - compg, [112](#)
 - compute_mult, [115](#)
 - dw, [116](#)
 - dwar, [116](#)
 - dwau, [116](#)
 - dwmult, [116](#)
 - dwor, [116](#)
 - dwou, [116](#)
 - full_step, [112](#)
 - g, [116](#)
 - init_g, [113](#)
 - init_integrator, [113](#)
 - init_noise, [113](#)
 - mult, [117](#)
 - noise, [117](#)
 - noisemult, [117](#)
 - q1fill, [117](#)
 - sq2, [117](#)
 - step, [114](#)
- rk2_ss_integrator, [118](#)
 - anoise, [121](#)
 - buf_f0, [121](#)
 - buf_f1, [121](#)
 - buf_y1, [121](#)
 - dwar, [121](#)
 - dwor, [122](#)
 - init_ss_integrator, [119](#)
 - ss_step, [119](#)
 - ss_tl_step, [119](#)
 - tendencies, [120](#)
 - tl_tendencies, [120](#)
- rk2_ss_integrator.f90, [246](#)
- rk2_stoch_integrator, [122](#)
 - anoise, [125](#)
 - buf_f0, [125](#)
 - buf_f1, [125](#)
 - buf_y1, [125](#)
 - dwar, [125](#)
 - dwau, [125](#)
 - dwor, [126](#)
 - dwou, [126](#)
 - init_integrator, [123](#)
 - int_tensor, [126](#)
 - step, [124](#)
 - tendencies, [124](#)
- rk2_stoch_integrator.f90, [247](#)
- rk2_tl_ad_integrator.f90, [248](#)
- rk2_wl_integrator, [126](#)
 - anoise, [130](#)
 - buf_f0, [130](#)
 - buf_f1, [131](#)
 - buf_m, [131](#)
 - buf_m1, [131](#)
 - buf_m2, [131](#)
 - buf_m3, [131](#)
 - buf_m3s, [131](#)
 - buf_y1, [131](#)
 - compute_m1, [127](#)
 - compute_m2, [128](#)

- dwar, [131](#)
- dwau, [132](#)
- dwor, [132](#)
- dwou, [132](#)
- full_step, [128](#)
- init_integrator, [128](#)
- step, [129](#)
- x1, [132](#)
- x2, [132](#)
- rk4_integrator.f90, [249](#)
- rk4_tl_ad_integrator.f90, [250](#)
- rp
 - params, [107](#)
- rr
 - params, [108](#)
- rra
 - params, [108](#)
- rred
 - mar, [82](#)
- rsf2csf
 - sqr_mod, [143](#)
- rstr
 - util, [202](#)
- s
 - inprod_analytic::atm_tensors, [216](#)
- s1
 - inprod_analytic, [65](#)
- s2
 - inprod_analytic, [65](#)
- s3
 - inprod_analytic, [66](#)
- s4
 - inprod_analytic, [66](#)
- sb
 - params, [108](#)
- sbpa
 - params, [108](#)
- sbpo
 - params, [108](#)
- sc
 - params, [109](#)
- scal_mul_coo
 - tensor, [177](#)
- scale
 - params, [109](#)
- selectev
 - sqr_mod, [144](#)
- sf
 - sf_def, [136](#)
- sf_def, [132](#)
 - bar, [135](#)
 - bau, [135](#)
 - bor, [135](#)
 - bou, [135](#)
 - exists, [135](#)
 - ind, [135](#)
 - load_sf, [133](#)
 - n_res, [135](#)
 - n_unres, [136](#)
 - rind, [136](#)
 - sf, [136](#)
 - sl_ind, [136](#)
 - sl_rind, [136](#)
- sf_def.f90, [250](#)
- sf_tensor
 - dec_tensor, [50](#)
- sig0
 - params, [109](#)
- sig1
 - sigma, [140](#)
- sig1r
 - sigma, [140](#)
- sig2
 - sigma, [140](#)
- sigma, [137](#)
 - compute_mult_sigma, [137](#)
 - dumb_mat1, [139](#)
 - dumb_mat2, [139](#)
 - dumb_mat3, [139](#)
 - dumb_mat4, [139](#)
 - ind1, [139](#)
 - ind2, [139](#)
 - init_sigma, [138](#)
 - n1, [139](#)
 - n2, [140](#)
 - rind1, [140](#)
 - rind2, [140](#)
 - sig1, [140](#)
 - sig1r, [140](#)
 - sig2, [140](#)
- simplify
 - tensor, [177](#)
- sl_ind
 - sf_def, [136](#)
- sl_rind
 - sf_def, [136](#)
- so
 - LICENSE.txt, [236](#)
- Software
 - LICENSE.txt, [236](#)
- sparse_mul2_j
 - tensor, [178](#)
- sparse_mul2_k
 - tensor, [179](#)
- sparse_mul3
 - tensor, [179](#)
- sparse_mul3_mat
 - tensor, [180](#)
- sparse_mul3_with_mat
 - tensor, [181](#)
- sparse_mul4
 - tensor, [181](#)
- sparse_mul4_mat
 - tensor, [182](#)
- sparse_mul4_with_mat_jl
 - tensor, [182](#)

- sparse_mul4_with_mat_kl
 - tensor, 183
- splint
 - corrmod, 36
- sq2
 - rk2_mtv_integrator, 117
- sqrt_mod, 141
 - chol, 141
 - csqrtm_triu, 142
 - init_sqrt, 143
 - lwork, 147
 - real_eps, 147
 - rsf2csf, 143
 - selectev, 144
 - sqrtn, 144
 - sqrtn_svd, 145
 - sqrtn_triu, 146
 - work, 147
- sqrt_mod.f90, 251
- sqrtn
 - sqrt_mod, 144
- sqrtn_svd
 - sqrt_mod, 145
- sqrtn_triu
 - sqrt_mod, 146
- ss_step
 - rk2_ss_integrator, 119
- ss_tensor
 - dec_tensor, 51
- ss_tl_step
 - rk2_ss_integrator, 119
- ss_tl_tensor
 - dec_tensor, 51
- stat, 148
 - acc, 148
 - i, 150
 - init_stat, 148
 - iter, 149
 - m, 150
 - mean, 149
 - mprev, 150
 - mtmp, 150
 - reset, 149
 - v, 150
 - var, 149
- stat.f90, 252
- step
 - integrator, 77
 - memory, 86
 - rk2_mtv_integrator, 114
 - rk2_stoch_integrator, 124
 - rk2_wl_integrator, 129
- stoch_atm_res_vec
 - stoch_mod, 151
- stoch_atm_unres_vec
 - stoch_mod, 152
- stoch_atm_vec
 - stoch_mod, 152
- stoch_oc_res_vec
 - stoch_mod, 152
- stoch_oc_unres_vec
 - stoch_mod, 153
- stoch_oc_vec
 - stoch_mod, 153
- stoch_mod, 150
 - gasdev, 151
 - gset, 154
 - iset, 154
 - stoch_atm_res_vec, 151
 - stoch_atm_unres_vec, 152
 - stoch_atm_vec, 152
 - stoch_oc_res_vec, 152
 - stoch_oc_unres_vec, 153
 - stoch_oc_vec, 153
 - stoch_vec, 153
- stoch_mod.f90, 252
- stoch_oc_res_vec
 - stoch_mod, 152
- stoch_oc_unres_vec
 - stoch_mod, 153
- stoch_oc_vec
 - stoch_mod, 153
- stoch_params, 154
 - dtn, 156
 - dts, 156
 - dtsn, 156
 - eps_pert, 156
 - init_stoch_params, 156
 - int_corr_mode, 157
 - load_mode, 157
 - maxint, 157
 - meml, 157
 - mems, 157
 - mnuti, 158
 - mode, 158
 - muti, 158
 - q_ar, 158
 - q_au, 158
 - q_or, 159
 - q_ou, 159
 - t_trans_mem, 159
 - t_trans_stoch, 159
 - tdelta, 159
 - x_int_mode, 160
- stoch_params.f90, 253
- stoch_vec
 - mar, 81
 - stoch_mod, 153
- str
 - util, 203
- sublicense
 - LICENSE.txt, 236
- suppress_and
 - dec_tensor, 46
- suppress_or
 - dec_tensor, 47
- t
 - aotensor_def, 24
- t_index
 - memory, 86
- t_run
 - params, 109

- t_trans
 - params, 109
- t_trans_mem
 - stoch_params, 159
- t_trans_stoch
 - stoch_params, 159
- ta0
 - params, 110
- tdelta
 - stoch_params, 159
- tendencies
 - integrator, 77
 - rk2_ss_integrator, 120
 - rk2_stoch_integrator, 124
- tensor, 160
 - add_matc_to_tensor, 163
 - add_matc_to_tensor4, 163
 - add_to_tensor, 165
 - add_vec_ijk_to_tensor4, 166
 - add_vec_ikl_to_tensor4, 167
 - add_vec_ikl_to_tensor4_perm, 168
 - add_vec_jk_to_tensor, 169
 - coo_to_mat_i, 170
 - coo_to_mat_ij, 170
 - coo_to_mat_ik, 171
 - coo_to_mat_j, 171
 - coo_to_vec_jk, 172
 - copy_tensor, 172
 - jsparse_mul, 173
 - jsparse_mul_mat, 173
 - load_tensor4_from_file, 174
 - mat_to_coo, 175
 - matc_to_coo, 175
 - print_tensor, 176
 - print_tensor4, 176
 - real_eps, 186
 - scal_mul_coo, 177
 - simplify, 177
 - sparse_mul2_j, 178
 - sparse_mul2_k, 179
 - sparse_mul3, 179
 - sparse_mul3_mat, 180
 - sparse_mul3_with_mat, 181
 - sparse_mul4, 181
 - sparse_mul4_mat, 182
 - sparse_mul4_with_mat_jl, 182
 - sparse_mul4_with_mat_kl, 183
 - tensor4_empty, 184
 - tensor4_to_coo4, 184
 - tensor_empty, 185
 - tensor_to_coo, 185
 - write_tensor4_to_file, 186
- tensor.f90, 254
- tensor4_empty
 - tensor, 184
- tensor4_to_coo4
 - tensor, 184
- tensor::coolist, 217
 - elems, 218
 - nelems, 218
- tensor::coolist4, 218
 - elems, 219
 - nelems, 219
- tensor::coolist_elem, 219
 - j, 219
 - k, 219
 - v, 220
- tensor::coolist_elem4, 220
 - j, 220
 - k, 220
 - l, 221
 - v, 221
- tensor_empty
 - tensor, 185
- tensor_to_coo
 - tensor, 185
- test_MAR.f90, 259
 - test_mar, 259
- test_MTV_int_tensor.f90, 260
 - test_mtv_int_tensor, 260
- test_MTV_sigma_tensor.f90, 260
 - test_sigma, 260
- test_WL_tensor.f90, 261
 - test_wl_tensor, 262
- test_aotensor
 - test_aotensor.f90, 257
- test_aotensor.f90, 257
 - test_aotensor, 257
- test_corr
 - test_corr.f90, 257
- test_corr.f90, 257
 - test_corr, 257
- test_corr_tensor
 - test_corr_tensor.f90, 258
- test_corr_tensor.f90, 258
 - test_corr_tensor, 258
- test_dec_tensor
 - test_dec_tensor.f90, 258
- test_dec_tensor.f90, 258
 - test_dec_tensor, 258
- test_inprod_analytic.f90, 258
 - inprod_analytic_test, 259
- test_m3
 - memory, 85
- test_mar
 - test_MAR.f90, 259
- test_memory
 - test_memory.f90, 260
- test_memory.f90, 259
 - test_memory, 260
- test_mtv_int_tensor
 - test_MTV_int_tensor.f90, 260
- test_sigma
 - test_MTV_sigma_tensor.f90, 260
- test_sqrtm
 - test_sqrtm.f90, 261

- test_sqrtm.f90, [261](#)
 - test_sqrtm, [261](#)
- test_tl_ad
 - test_tl_ad.f90, [261](#)
- test_tl_ad.f90, [261](#)
 - test_tl_ad, [261](#)
- test_wl_tensor
 - test_WL_tensor.f90, [262](#)
- theta
 - aotensor_def, [24](#)
- tl
 - tl_ad_tensor, [196](#)
- tl_ad_integrator, [187](#)
 - ad_step, [188](#)
 - buf_f0, [189](#)
 - buf_f1, [189](#)
 - buf_ka, [189](#)
 - buf_kb, [189](#)
 - buf_y1, [190](#)
 - init_tl_ad_integrator, [188](#)
 - tl_step, [188](#)
- tl_ad_tensor, [190](#)
 - ad, [191](#)
 - ad_add_count, [191](#)
 - ad_add_count_ref, [192](#)
 - ad_coeff, [192](#)
 - ad_coeff_ref, [193](#)
 - adtensor, [198](#)
 - compute_adtensor, [193](#)
 - compute_adtensor_ref, [193](#)
 - compute_tltensor, [194](#)
 - count_elems, [198](#)
 - init_adtensor, [194](#)
 - init_adtensor_ref, [194](#)
 - init_tltensor, [195](#)
 - jacobian, [195](#)
 - jacobian_mat, [196](#)
 - real_eps, [198](#)
 - tl, [196](#)
 - tl_add_count, [196](#)
 - tl_coeff, [197](#)
 - tltensor, [198](#)
- tl_ad_tensor.f90, [262](#)
- tl_add_count
 - tl_ad_tensor, [196](#)
- tl_coeff
 - tl_ad_tensor, [197](#)
- tl_step
 - tl_ad_integrator, [188](#)
- tl_tendencies
 - rk2_ss_integrator, [120](#)
- tltensor
 - tl_ad_tensor, [198](#)
- to0
 - params, [110](#)
- trapzd
 - int_comp, [71](#)
- triu
 - util, [203](#)
- tw
 - params, [110](#)
- typ
 - inprod_analytic::atm_wavenum, [217](#)
- use
 - LICENSE.txt, [236](#)
- util, [198](#)
 - cdiag, [199](#)
 - choldc, [199](#)
 - cprintmat, [200](#)
 - diag, [200](#)
 - floordiv, [200](#)
 - init_one, [200](#)
 - init_random_seed, [201](#)
 - invmat, [201](#)
 - ireduce, [201](#)
 - mat_contract, [201](#)
 - mat_trace, [202](#)
 - printmat, [202](#)
 - reduce, [202](#)
 - rstr, [202](#)
 - str, [203](#)
 - triu, [203](#)
 - vector_outer, [203](#)
- util.f90, [263](#)
 - lcg, [264](#)
- utot
 - mtv_int_tensor, [95](#)
- v
 - stat, [150](#)
 - tensor::coolist_elem, [220](#)
 - tensor::coolist_elem4, [221](#)
- var
 - stat, [149](#)
- vector_outer
 - util, [203](#)
- vtot
 - mtv_int_tensor, [95](#)
- w
 - inprod_analytic::ocean_tensors, [222](#)
 - mar, [82](#)
- WL_tensor.f90, [264](#)
- wl_tensor, [203](#)
 - b1, [208](#)
 - b14, [208](#)
 - b14def, [208](#)
 - b2, [208](#)
 - b23, [209](#)
 - b23def, [209](#)
 - b3, [209](#)
 - b4, [209](#)
 - dumb_mat1, [209](#)
 - dumb_mat2, [209](#)
 - dumb_mat3, [210](#)
 - dumb_mat4, [210](#)

- dumb_vec, [210](#)
- init_wl_tensor, [205](#)
- l1, [210](#)
- l2, [210](#)
- l4, [211](#)
- l5, [211](#)
- ldef, [211](#)
- ltot, [211](#)
- m11, [211](#)
- m12, [211](#)
- m12def, [212](#)
- m13, [212](#)
- m1tot, [212](#)
- m21, [212](#)
- m21def, [212](#)
- m22, [212](#)
- m22def, [213](#)
- mdef, [213](#)
- mtot, [213](#)
- work
 - sqrt_mod, [147](#)
- wred
 - mar, [82](#)
- write_tensor4_to_file
 - tensor, [186](#)
- writeout
 - params, [110](#)
- x
 - memory, [86](#)
- x1
 - rk2_wl_integrator, [132](#)
- x2
 - rk2_wl_integrator, [132](#)
- x_int_mode
 - stoch_params, [160](#)
- xa
 - corrmod, [39](#)
- xs
 - memory, [87](#)
- y2
 - corrmod, [39](#)
- ya
 - corrmod, [39](#)
- ydy
 - corr_tensor, [28](#)
- ydyd
 - corr_tensor, [28](#)
- yy
 - corr_tensor, [29](#)
- zs
 - memory, [87](#)