

# Reference Manual

Generated by Doxygen 1.8.11



# Contents

<b>1</b>	<b>Modular arbitrary-order ocean-atmosphere model: MAOOAM -- Stochastic Fortran implementation</b>	<b>1</b>
<b>2</b>	<b>Modular arbitrary-order ocean-atmosphere model: The Tangent Linear and Adjoint model</b>	<b>9</b>
<b>3</b>	<b>Modular arbitrary-order ocean-atmosphere model: The MTV and WL parameterizations</b>	<b>11</b>
<b>4</b>	<b>Modular arbitrary-order ocean-atmosphere model: Definition files formats</b>	<b>17</b>
<b>5</b>	<b>Modules Index</b>	<b>19</b>
5.1	Modules List . . . . .	19
<b>6</b>	<b>Data Type Index</b>	<b>21</b>
6.1	Data Types List . . . . .	21
<b>7</b>	<b>File Index</b>	<b>23</b>
7.1	File List . . . . .	23
<b>8</b>	<b>Module Documentation</b>	<b>25</b>
8.1	aotensor_def Module Reference . . . . .	25
8.1.1	Detailed Description . . . . .	26
8.1.2	Function/Subroutine Documentation . . . . .	26
8.1.2.1	a(i) . . . . .	26
8.1.2.2	add_count(i, j, k, v) . . . . .	26
8.1.2.3	coeff(i, j, k, v) . . . . .	26
8.1.2.4	compute_aotensor(func) . . . . .	27
8.1.2.5	init_aotensor . . . . .	27
8.1.2.6	kdelta(i, j) . . . . .	28

8.1.2.7	<a href="#">psi(i)</a>	28
8.1.2.8	<a href="#">t(i)</a>	28
8.1.2.9	<a href="#">theta(i)</a>	28
8.1.3	<a href="#">Variable Documentation</a>	29
8.1.3.1	<a href="#">aotensor</a>	29
8.1.3.2	<a href="#">count_elems</a>	29
8.1.3.3	<a href="#">real_eps</a>	29
8.2	<a href="#">corr_tensor Module Reference</a>	29
8.2.1	<a href="#">Detailed Description</a>	30
8.2.2	<a href="#">Function/Subroutine Documentation</a>	30
8.2.2.1	<a href="#">init_corr_tensor</a>	30
8.2.3	<a href="#">Variable Documentation</a>	31
8.2.3.1	<a href="#">dumb_mat1</a>	31
8.2.3.2	<a href="#">dumb_mat2</a>	31
8.2.3.3	<a href="#">dumb_vec</a>	32
8.2.3.4	<a href="#">dy</a>	32
8.2.3.5	<a href="#">dyy</a>	32
8.2.3.6	<a href="#">expm</a>	32
8.2.3.7	<a href="#">ydy</a>	32
8.2.3.8	<a href="#">ydyy</a>	33
8.2.3.9	<a href="#">yy</a>	33
8.3	<a href="#">corrmod Module Reference</a>	33
8.3.1	<a href="#">Detailed Description</a>	34
8.3.2	<a href="#">Function/Subroutine Documentation</a>	34
8.3.2.1	<a href="#">corrcomp_from_def(s)</a>	34
8.3.2.2	<a href="#">corrcomp_from_fit(s)</a>	38
8.3.2.3	<a href="#">corrcomp_from_spline(s)</a>	38
8.3.2.4	<a href="#">fs(s, p)</a>	39
8.3.2.5	<a href="#">init_corr</a>	39
8.3.2.6	<a href="#">splint(xa, ya, y2a, n, x, y)</a>	40

8.3.3	Variable Documentation	41
8.3.3.1	corr_i	41
8.3.3.2	corr_i_full	41
8.3.3.3	corr_ij	41
8.3.3.4	corrcomp	42
8.3.3.5	inv_corr_i	42
8.3.3.6	inv_corr_i_full	42
8.3.3.7	khi	42
8.3.3.8	klo	42
8.3.3.9	mean	42
8.3.3.10	mean_full	43
8.3.3.11	nspl	43
8.3.3.12	xa	43
8.3.3.13	y2	43
8.3.3.14	ya	43
8.4	dec_tensor Module Reference	44
8.4.1	Detailed Description	45
8.4.2	Function/Subroutine Documentation	45
8.4.2.1	init_dec_tensor	45
8.4.2.2	init_sub_tensor(t, cst, v)	49
8.4.2.3	reorder(t, cst, v)	49
8.4.2.4	suppress_and(t, cst, v1, v2)	50
8.4.2.5	suppress_or(t, cst, v1, v2)	50
8.4.3	Variable Documentation	51
8.4.3.1	bxxx	51
8.4.3.2	bxyy	51
8.4.3.3	bxyy	52
8.4.3.4	byxx	52
8.4.3.5	byxy	52
8.4.3.6	byyy	52

8.4.3.7	dumb	52
8.4.3.8	ff_tensor	53
8.4.3.9	fs_tensor	53
8.4.3.10	hx	53
8.4.3.11	hy	53
8.4.3.12	lxx	53
8.4.3.13	lxy	54
8.4.3.14	lyx	54
8.4.3.15	lyy	54
8.4.3.16	sf_tensor	54
8.4.3.17	ss_tensor	54
8.4.3.18	ss_tl_tensor	55
8.5	ic_def Module Reference	55
8.5.1	Detailed Description	55
8.5.2	Function/Subroutine Documentation	55
8.5.2.1	load_ic	55
8.5.3	Variable Documentation	57
8.5.3.1	exists	57
8.5.3.2	ic	57
8.6	inprod_analytic Module Reference	57
8.6.1	Detailed Description	59
8.6.2	Function/Subroutine Documentation	59
8.6.2.1	b1(Pi, Pj, Pk)	59
8.6.2.2	b2(Pi, Pj, Pk)	59
8.6.2.3	calculate_a(i, j)	60
8.6.2.4	calculate_b(i, j, k)	60
8.6.2.5	calculate_c_atm(i, j)	60
8.6.2.6	calculate_c_oc(i, j, k)	61
8.6.2.7	calculate_d(i, j)	61
8.6.2.8	calculate_g(i, j, k)	61

8.6.2.9	<code>calculate_k(i, j)</code>	62
8.6.2.10	<code>calculate_m(i, j)</code>	62
8.6.2.11	<code>calculate_n(i, j)</code>	63
8.6.2.12	<code>calculate_o(i, j, k)</code>	63
8.6.2.13	<code>calculate_s(i, j)</code>	64
8.6.2.14	<code>calculate_w(i, j)</code>	64
8.6.2.15	<code>delta(r)</code>	64
8.6.2.16	<code>flambda(r)</code>	65
8.6.2.17	<code>init_inprod</code>	65
8.6.2.18	<code>s1(Pj, Pk, Mj, Hk)</code>	66
8.6.2.19	<code>s2(Pj, Pk, Mj, Hk)</code>	66
8.6.2.20	<code>s3(Pj, Pk, Hj, Hk)</code>	66
8.6.2.21	<code>s4(Pj, Pk, Hj, Hk)</code>	67
8.6.3	Variable Documentation	67
8.6.3.1	<code>atmos</code>	67
8.6.3.2	<code>awavenum</code>	67
8.6.3.3	<code>ocean</code>	67
8.6.3.4	<code>owavenum</code>	67
8.7	<code>int_comp</code> Module Reference	68
8.7.1	Detailed Description	68
8.7.2	Function/Subroutine Documentation	68
8.7.2.1	<code>integrate(func, ss)</code>	68
8.7.2.2	<code>midexp(func, aa, bb, s, n)</code>	69
8.7.2.3	<code>midpnt(func, a, b, s, n)</code>	69
8.7.2.4	<code>polint(xa, ya, n, x, y, dy)</code>	70
8.7.2.5	<code>qromb(func, a, b, ss)</code>	70
8.7.2.6	<code>qromo(func, a, b, ss, choose)</code>	71
8.7.2.7	<code>trapzd(func, a, b, s, n)</code>	72
8.8	<code>int_corr</code> Module Reference	72
8.8.1	Detailed Description	73

8.8.2	Function/Subroutine Documentation	73
8.8.2.1	comp_corrint	73
8.8.2.2	func_ij(s)	74
8.8.2.3	func_ijkl(s)	75
8.8.2.4	init_corrint	75
8.8.3	Variable Documentation	75
8.8.3.1	corr2int	75
8.8.3.2	corrint	76
8.8.3.3	oi	76
8.8.3.4	oj	76
8.8.3.5	ok	76
8.8.3.6	ol	76
8.8.3.7	real_eps	76
8.9	integrator Module Reference	76
8.9.1	Detailed Description	77
8.9.2	Function/Subroutine Documentation	78
8.9.2.1	init_integrator	78
8.9.2.2	step(y, t, dt, res)	78
8.9.2.3	tendencies(t, y, res)	78
8.9.3	Variable Documentation	79
8.9.3.1	buf_f0	79
8.9.3.2	buf_f1	79
8.9.3.3	buf_ka	79
8.9.3.4	buf_kb	79
8.9.3.5	buf_y1	80
8.10	mar Module Reference	80
8.10.1	Detailed Description	80
8.10.2	Function/Subroutine Documentation	81
8.10.2.1	init_mar	81
8.10.2.2	mar_step(x)	81



8.10.2.3	<a href="#">mar_step_red(xred)</a>	82
8.10.2.4	<a href="#">stoch_vec(dW)</a>	82
8.10.3	<a href="#">Variable Documentation</a>	82
8.10.3.1	<a href="#">buf_y</a>	82
8.10.3.2	<a href="#">dw</a>	83
8.10.3.3	<a href="#">ms</a>	83
8.10.3.4	<a href="#">q</a>	83
8.10.3.5	<a href="#">qred</a>	83
8.10.3.6	<a href="#">rred</a>	83
8.10.3.7	<a href="#">w</a>	83
8.10.3.8	<a href="#">wred</a>	84
8.11	<a href="#">memory Module Reference</a>	84
8.11.1	<a href="#">Detailed Description</a>	84
8.11.2	<a href="#">Function/Subroutine Documentation</a>	85
8.11.2.1	<a href="#">compute_m3(y, dt, dtn, savey, save_ev, evolve, inter, h_int)</a>	85
8.11.2.2	<a href="#">init_memory</a>	86
8.11.2.3	<a href="#">test_m3(y, dt, dtn, h_int)</a>	86
8.11.3	<a href="#">Variable Documentation</a>	87
8.11.3.1	<a href="#">buf_m</a>	87
8.11.3.2	<a href="#">buf_m3</a>	87
8.11.3.3	<a href="#">step</a>	87
8.11.3.4	<a href="#">t_index</a>	87
8.11.3.5	<a href="#">x</a>	88
8.11.3.6	<a href="#">xs</a>	88
8.11.3.7	<a href="#">zs</a>	88
8.12	<a href="#">mtv_int_tensor Module Reference</a>	88
8.12.1	<a href="#">Detailed Description</a>	90
8.12.2	<a href="#">Function/Subroutine Documentation</a>	90
8.12.2.1	<a href="#">init_mtv_int_tensor</a>	90
8.12.3	<a href="#">Variable Documentation</a>	92

8.12.3.1	b1	93
8.12.3.2	b2	93
8.12.3.3	btot	93
8.12.3.4	dumb_mat1	93
8.12.3.5	dumb_mat2	93
8.12.3.6	dumb_mat3	94
8.12.3.7	dumb_mat4	94
8.12.3.8	dumb_vec	94
8.12.3.9	h1	94
8.12.3.10	h2	94
8.12.3.11	h3	95
8.12.3.12	htot	95
8.12.3.13	l1	95
8.12.3.14	l2	95
8.12.3.15	l3	95
8.12.3.16	ltot	96
8.12.3.17	mtot	96
8.12.3.18	q1	96
8.12.3.19	q2	96
8.12.3.20	utot	96
8.12.3.21	vtot	97
8.13	params Module Reference	97
8.13.1	Detailed Description	99
8.13.2	Function/Subroutine Documentation	100
8.13.2.1	init_nml	100
8.13.2.2	init_params	100
8.13.3	Variable Documentation	101
8.13.3.1	ams	101
8.13.3.2	betp	101
8.13.3.3	ca	101

8.13.3.4	co	102
8.13.3.5	cpa	102
8.13.3.6	cpo	102
8.13.3.7	d	102
8.13.3.8	dp	102
8.13.3.9	dt	103
8.13.3.10	epsa	103
8.13.3.11	f0	103
8.13.3.12	g	103
8.13.3.13	ga	103
8.13.3.14	go	104
8.13.3.15	gp	104
8.13.3.16	h	104
8.13.3.17	k	104
8.13.3.18	kd	104
8.13.3.19	kdp	105
8.13.3.20	kp	105
8.13.3.21	l	105
8.13.3.22	lambda	105
8.13.3.23	lpa	105
8.13.3.24	lpo	106
8.13.3.25	lr	106
8.13.3.26	lsbpa	106
8.13.3.27	lsbpo	106
8.13.3.28	n	106
8.13.3.29	natm	107
8.13.3.30	nbatm	107
8.13.3.31	nboc	107
8.13.3.32	ndim	107
8.13.3.33	noc	107

8.13.3.34 nua . . . . .	108
8.13.3.35 nuap . . . . .	108
8.13.3.36 nuo . . . . .	108
8.13.3.37 nuop . . . . .	108
8.13.3.38 oms . . . . .	108
8.13.3.39 phi0 . . . . .	109
8.13.3.40 phi0_npi . . . . .	109
8.13.3.41 pi . . . . .	109
8.13.3.42 r . . . . .	109
8.13.3.43 rp . . . . .	109
8.13.3.44 rr . . . . .	110
8.13.3.45 rra . . . . .	110
8.13.3.46 sb . . . . .	110
8.13.3.47 sbpa . . . . .	110
8.13.3.48 sbpo . . . . .	110
8.13.3.49 sc . . . . .	111
8.13.3.50 scale . . . . .	111
8.13.3.51 sig0 . . . . .	111
8.13.3.52 t_run . . . . .	111
8.13.3.53 t_trans . . . . .	111
8.13.3.54 ta0 . . . . .	112
8.13.3.55 to0 . . . . .	112
8.13.3.56 tw . . . . .	112
8.13.3.57 writeout . . . . .	112
8.14 rk2_mtv_integrator Module Reference . . . . .	112
8.14.1 Detailed Description . . . . .	114
8.14.2 Function/Subroutine Documentation . . . . .	114
8.14.2.1 compg(y) . . . . .	114
8.14.2.2 full_step(y, t, dt, dtn, res) . . . . .	114
8.14.2.3 init_g . . . . .	115

8.14.2.4	<a href="#">init_integrator</a>	115
8.14.2.5	<a href="#">init_noise</a>	115
8.14.2.6	<a href="#">step(y, t, dt, dtn, res, tend)</a>	116
8.14.3	<a href="#">Variable Documentation</a>	117
8.14.3.1	<a href="#">anoise</a>	117
8.14.3.2	<a href="#">buf_f0</a>	117
8.14.3.3	<a href="#">buf_f1</a>	117
8.14.3.4	<a href="#">buf_g</a>	117
8.14.3.5	<a href="#">buf_y1</a>	117
8.14.3.6	<a href="#">compute_mult</a>	118
8.14.3.7	<a href="#">dw</a>	118
8.14.3.8	<a href="#">dwar</a>	118
8.14.3.9	<a href="#">dwau</a>	118
8.14.3.10	<a href="#">dwmult</a>	118
8.14.3.11	<a href="#">dwor</a>	118
8.14.3.12	<a href="#">dwou</a>	118
8.14.3.13	<a href="#">g</a>	119
8.14.3.14	<a href="#">mult</a>	119
8.14.3.15	<a href="#">noise</a>	119
8.14.3.16	<a href="#">noisemult</a>	119
8.14.3.17	<a href="#">q1fill</a>	119
8.14.3.18	<a href="#">sq2</a>	119
8.15	<a href="#">rk2_ss_integrator Module Reference</a>	120
8.15.1	<a href="#">Detailed Description</a>	120
8.15.2	<a href="#">Function/Subroutine Documentation</a>	121
8.15.2.1	<a href="#">init_ss_integrator</a>	121
8.15.2.2	<a href="#">ss_step(y, ys, t, dt, dtn, res)</a>	121
8.15.2.3	<a href="#">ss_tl_step(y, ys, t, dt, dtn, res)</a>	122
8.15.2.4	<a href="#">tendencies(t, y, res)</a>	122
8.15.2.5	<a href="#">tl_tendencies(t, y, ys, res)</a>	123

8.15.3	Variable Documentation	123
8.15.3.1	anoise	123
8.15.3.2	buf_f0	123
8.15.3.3	buf_f1	123
8.15.3.4	buf_y1	123
8.15.3.5	dwar	124
8.15.3.6	dwor	124
8.16	rk2_stoch_integrator Module Reference	124
8.16.1	Detailed Description	125
8.16.2	Function/Subroutine Documentation	125
8.16.2.1	init_integrator(force)	125
8.16.2.2	step(y, t, dt, dtn, res, tend)	126
8.16.2.3	tendencies(t, y, res)	126
8.16.3	Variable Documentation	127
8.16.3.1	anoise	127
8.16.3.2	buf_f0	127
8.16.3.3	buf_f1	127
8.16.3.4	buf_y1	127
8.16.3.5	dwar	127
8.16.3.6	dwau	128
8.16.3.7	dwor	128
8.16.3.8	dwou	128
8.16.3.9	int_tensor	128
8.17	rk2_wl_integrator Module Reference	128
8.17.1	Detailed Description	129
8.17.2	Function/Subroutine Documentation	129
8.17.2.1	compute_m1(y)	129
8.17.2.2	compute_m2(y)	130
8.17.2.3	full_step(y, t, dt, dtn, res)	130
8.17.2.4	init_integrator	131

8.17.2.5	<code>step(y, t, dt, dtn, res, tend)</code>	131
8.17.3	Variable Documentation	132
8.17.3.1	<code>anoise</code>	132
8.17.3.2	<code>buf_f0</code>	133
8.17.3.3	<code>buf_f1</code>	133
8.17.3.4	<code>buf_m</code>	133
8.17.3.5	<code>buf_m1</code>	133
8.17.3.6	<code>buf_m2</code>	133
8.17.3.7	<code>buf_m3</code>	133
8.17.3.8	<code>buf_m3s</code>	133
8.17.3.9	<code>buf_y1</code>	133
8.17.3.10	<code>dwar</code>	134
8.17.3.11	<code>dwau</code>	134
8.17.3.12	<code>dwor</code>	134
8.17.3.13	<code>dwou</code>	134
8.17.3.14	<code>x1</code>	134
8.17.3.15	<code>x2</code>	134
8.18	<code>sf_def</code> Module Reference	134
8.18.1	Detailed Description	135
8.18.2	Function/Subroutine Documentation	135
8.18.2.1	<code>load_sf</code>	135
8.18.3	Variable Documentation	137
8.18.3.1	<code>bar</code>	137
8.18.3.2	<code>bau</code>	137
8.18.3.3	<code>bor</code>	137
8.18.3.4	<code>bou</code>	137
8.18.3.5	<code>exists</code>	137
8.18.3.6	<code>ind</code>	137
8.18.3.7	<code>n_res</code>	138
8.18.3.8	<code>n_unres</code>	138

8.18.3.9	rind	138
8.18.3.10	sf	138
8.18.3.11	sl_ind	138
8.18.3.12	sl_rind	138
8.19	sigma Module Reference	139
8.19.1	Detailed Description	139
8.19.2	Function/Subroutine Documentation	139
8.19.2.1	compute_mult_sigma(y)	139
8.19.2.2	init_sigma(mult, Q1fill)	140
8.19.3	Variable Documentation	141
8.19.3.1	dumb_mat1	141
8.19.3.2	dumb_mat2	141
8.19.3.3	dumb_mat3	141
8.19.3.4	dumb_mat4	141
8.19.3.5	ind1	141
8.19.3.6	ind2	141
8.19.3.7	n1	142
8.19.3.8	n2	142
8.19.3.9	rind1	142
8.19.3.10	rind2	142
8.19.3.11	sig1	142
8.19.3.12	sig1r	142
8.19.3.13	sig2	142
8.20	sqrt_mod Module Reference	143
8.20.1	Detailed Description	143
8.20.2	Function/Subroutine Documentation	143
8.20.2.1	chol(A, sqA, info)	143
8.20.2.2	csqrtm_triu(A, sqA, info, bs)	144
8.20.2.3	init_sqrt	145
8.20.2.4	rsf2csf(T, Z, Tz, Zz)	145



8.20.2.5	<code>selectev(a, b)</code>	146
8.20.2.6	<code>sqrtn(A, sqA, info, info_triu, bs)</code>	146
8.20.2.7	<code>sqrtn_svd(A, sqA, info, info_triu, bs)</code>	147
8.20.2.8	<code>sqrtn_triu(A, sqA, info, bs)</code>	148
8.20.3	Variable Documentation	149
8.20.3.1	<code>lwork</code>	149
8.20.3.2	<code>real_eps</code>	149
8.20.3.3	<code>work</code>	149
8.21	stat Module Reference	150
8.21.1	Detailed Description	150
8.21.2	Function/Subroutine Documentation	150
8.21.2.1	<code>acc(x)</code>	150
8.21.2.2	<code>init_stat</code>	151
8.21.2.3	<code>iter()</code>	151
8.21.2.4	<code>mean()</code>	151
8.21.2.5	<code>reset</code>	151
8.21.2.6	<code>var()</code>	151
8.21.3	Variable Documentation	152
8.21.3.1	<code>i</code>	152
8.21.3.2	<code>m</code>	152
8.21.3.3	<code>mprev</code>	152
8.21.3.4	<code>mtmp</code>	152
8.21.3.5	<code>v</code>	152
8.22	stoch_mod Module Reference	152
8.22.1	Detailed Description	153
8.22.2	Function/Subroutine Documentation	153
8.22.2.1	<code>gasdev()</code>	153
8.22.2.2	<code>stoch_atm_res_vec(dW)</code>	154
8.22.2.3	<code>stoch_atm_unres_vec(dW)</code>	154
8.22.2.4	<code>stoch_atm_vec(dW)</code>	154

8.22.2.5	stoch_oc_res_vec(dW)	155
8.22.2.6	stoch_oc_unres_vec(dW)	155
8.22.2.7	stoch_oc_vec(dW)	155
8.22.2.8	stoch_vec(dW)	156
8.22.3	Variable Documentation	156
8.22.3.1	gset	156
8.22.3.2	iset	156
8.23	stoch_params Module Reference	156
8.23.1	Detailed Description	157
8.23.2	Function/Subroutine Documentation	158
8.23.2.1	init_stoch_params	158
8.23.3	Variable Documentation	158
8.23.3.1	dtn	158
8.23.3.2	dts	158
8.23.3.3	dtsn	158
8.23.3.4	eps_pert	159
8.23.3.5	int_corr_mode	159
8.23.3.6	load_mode	159
8.23.3.7	maxint	159
8.23.3.8	meml	159
8.23.3.9	mems	160
8.23.3.10	mnuti	160
8.23.3.11	mode	160
8.23.3.12	muti	160
8.23.3.13	q_ar	160
8.23.3.14	q_au	161
8.23.3.15	q_or	161
8.23.3.16	q_ou	161
8.23.3.17	t_trans_mem	161
8.23.3.18	t_trans_stoch	161

8.23.3.19 tdelta . . . . .	162
8.23.3.20 x_int_mode . . . . .	162
8.24 tensor Module Reference . . . . .	162
8.24.1 Detailed Description . . . . .	165
8.24.2 Function/Subroutine Documentation . . . . .	165
8.24.2.1 add_check(t, i, j, k, v, dst) . . . . .	165
8.24.2.2 add_elem(t, i, j, k, v) . . . . .	165
8.24.2.3 add_matc_to_tensor(i, src, dst) . . . . .	166
8.24.2.4 add_matc_to_tensor4(i, j, src, dst) . . . . .	167
8.24.2.5 add_to_tensor(src, dst) . . . . .	168
8.24.2.6 add_vec_ijk_to_tensor4(i, j, k, src, dst) . . . . .	169
8.24.2.7 add_vec_ikl_to_tensor4(i, k, l, src, dst) . . . . .	170
8.24.2.8 add_vec_ikl_to_tensor4_perm(i, k, l, src, dst) . . . . .	170
8.24.2.9 add_vec_jk_to_tensor(j, k, src, dst) . . . . .	171
8.24.2.10 coo_to_mat_i(i, src, dst) . . . . .	172
8.24.2.11 coo_to_mat_ij(src, dst) . . . . .	173
8.24.2.12 coo_to_mat_ik(src, dst) . . . . .	173
8.24.2.13 coo_to_mat_j(j, src, dst) . . . . .	174
8.24.2.14 coo_to_vec_jk(j, k, src, dst) . . . . .	174
8.24.2.15 copy_coo(src, dst) . . . . .	174
8.24.2.16 jsparse_mul(coolist_ijk, arr_j, jcoo_ij) . . . . .	175
8.24.2.17 jsparse_mul_mat(coolist_ijk, arr_j, jcoo_ij) . . . . .	176
8.24.2.18 load_tensor4_from_file(s, t) . . . . .	177
8.24.2.19 load_tensor_from_file(s, t) . . . . .	178
8.24.2.20 mat_to_coo(src, dst) . . . . .	179
8.24.2.21 matc_to_coo(src, dst) . . . . .	179
8.24.2.22 print_tensor(t, s) . . . . .	180
8.24.2.23 print_tensor4(t) . . . . .	180
8.24.2.24 scal_mul_coo(s, t) . . . . .	181
8.24.2.25 simplify(tensor) . . . . .	181

8.24.2.26	<code>sparse_mul2(coolist_ij, arr_j, res)</code>	182
8.24.2.27	<code>sparse_mul2_j(coolist_ijk, arr_j, res)</code>	183
8.24.2.28	<code>sparse_mul2_k(coolist_ijk, arr_k, res)</code>	183
8.24.2.29	<code>sparse_mul3(coolist_ijk, arr_j, arr_k, res)</code>	184
8.24.2.30	<code>sparse_mul3_mat(coolist_ijk, arr_k, res)</code>	185
8.24.2.31	<code>sparse_mul3_with_mat(coolist_ijk, mat_jk, res)</code>	185
8.24.2.32	<code>sparse_mul4(coolist_ijkl, arr_j, arr_k, arr_l, res)</code>	186
8.24.2.33	<code>sparse_mul4_mat(coolist_ijkl, arr_k, arr_l, res)</code>	186
8.24.2.34	<code>sparse_mul4_with_mat_jl(coolist_ijkl, mat_jl, res)</code>	187
8.24.2.35	<code>sparse_mul4_with_mat_kl(coolist_ijkl, mat_kl, res)</code>	188
8.24.2.36	<code>tensor4_empty(t)</code>	188
8.24.2.37	<code>tensor4_to_coo4(src, dst)</code>	189
8.24.2.38	<code>tensor_empty(t)</code>	189
8.24.2.39	<code>tensor_to_coo(src, dst)</code>	190
8.24.2.40	<code>write_tensor4_to_file(s, t)</code>	191
8.24.2.41	<code>write_tensor_to_file(s, t)</code>	191
8.24.3	Variable Documentation	191
8.24.3.1	<code>real_eps</code>	191
8.25	<code>tl_ad_integrator</code> Module Reference	192
8.25.1	Detailed Description	192
8.25.2	Function/Subroutine Documentation	192
8.25.2.1	<code>ad_step(y, ystar, t, dt, res)</code>	192
8.25.2.2	<code>init_tl_ad_integrator</code>	193
8.25.2.3	<code>tl_step(y, ystar, t, dt, res)</code>	193
8.25.3	Variable Documentation	194
8.25.3.1	<code>buf_f0</code>	194
8.25.3.2	<code>buf_f1</code>	194
8.25.3.3	<code>buf_ka</code>	194
8.25.3.4	<code>buf_kb</code>	194
8.25.3.5	<code>buf_y1</code>	195

8.26	<a href="#">tl_ad_tensor Module Reference</a>	195
8.26.1	<a href="#">Detailed Description</a>	196
8.26.2	<a href="#">Function/Subroutine Documentation</a>	196
8.26.2.1	<a href="#">ad(t, ystar, deltax, buf)</a>	196
8.26.2.2	<a href="#">ad_add_count(i, j, k, v)</a>	196
8.26.2.3	<a href="#">ad_add_count_ref(i, j, k, v)</a>	197
8.26.2.4	<a href="#">ad_coeff(i, j, k, v)</a>	197
8.26.2.5	<a href="#">ad_coeff_ref(i, j, k, v)</a>	198
8.26.2.6	<a href="#">compute_adtensor(func)</a>	198
8.26.2.7	<a href="#">compute_adtensor_ref(func)</a>	199
8.26.2.8	<a href="#">compute_tltensor(func)</a>	199
8.26.2.9	<a href="#">init_adtensor</a>	199
8.26.2.10	<a href="#">init_adtensor_ref</a>	199
8.26.2.11	<a href="#">init_tltensor</a>	200
8.26.2.12	<a href="#">jacobian(ystar)</a>	200
8.26.2.13	<a href="#">jacobian_mat(ystar)</a>	201
8.26.2.14	<a href="#">tl(t, ystar, deltax, buf)</a>	201
8.26.2.15	<a href="#">tl_add_count(i, j, k, v)</a>	202
8.26.2.16	<a href="#">tl_coeff(i, j, k, v)</a>	202
8.26.3	<a href="#">Variable Documentation</a>	203
8.26.3.1	<a href="#">adtensor</a>	203
8.26.3.2	<a href="#">count_elems</a>	203
8.26.3.3	<a href="#">real_eps</a>	203
8.26.3.4	<a href="#">tltensor</a>	203
8.27	<a href="#">util Module Reference</a>	203
8.27.1	<a href="#">Detailed Description</a>	204
8.27.2	<a href="#">Function/Subroutine Documentation</a>	204
8.27.2.1	<a href="#">cdiag(A, d)</a>	204
8.27.2.2	<a href="#">choldc(a, p)</a>	205
8.27.2.3	<a href="#">cprintmat(A)</a>	205

8.27.2.4	<code>diag(A, d)</code>	205
8.27.2.5	<code>floordiv(i, j)</code>	205
8.27.2.6	<code>init_one(A)</code>	206
8.27.2.7	<code>init_random_seed()</code>	206
8.27.2.8	<code>invmat(A)</code>	206
8.27.2.9	<code>ireduce(A, Ared, n, ind, rind)</code>	206
8.27.2.10	<code>isin(c, s)</code>	207
8.27.2.11	<code>mat_contract(A, B)</code>	207
8.27.2.12	<code>mat_trace(A)</code>	207
8.27.2.13	<code>piksort(k, arr, par)</code>	208
8.27.2.14	<code>printmat(A)</code>	208
8.27.2.15	<code>reduce(A, Ared, n, ind, rind)</code>	208
8.27.2.16	<code>rstr(x, fm)</code>	209
8.27.2.17	<code>str(k)</code>	209
8.27.2.18	<code>triu(A, T)</code>	209
8.27.2.19	<code>vector_outer(u, v, A)</code>	209
8.28	<code>wl_tensor</code> Module Reference	209
8.28.1	Detailed Description	211
8.28.2	Function/Subroutine Documentation	211
8.28.2.1	<code>init_wl_tensor</code>	211
8.28.3	Variable Documentation	214
8.28.3.1	<code>b1</code>	214
8.28.3.2	<code>b14</code>	214
8.28.3.3	<code>b14def</code>	214
8.28.3.4	<code>b2</code>	215
8.28.3.5	<code>b23</code>	215
8.28.3.6	<code>b23def</code>	215
8.28.3.7	<code>b3</code>	215
8.28.3.8	<code>b4</code>	215
8.28.3.9	<code>dumb_mat1</code>	215

8.28.3.10 dumb_mat2 . . . . .	216
8.28.3.11 dumb_mat3 . . . . .	216
8.28.3.12 dumb_mat4 . . . . .	216
8.28.3.13 dumb_vec . . . . .	216
8.28.3.14 l1 . . . . .	216
8.28.3.15 l2 . . . . .	217
8.28.3.16 l4 . . . . .	217
8.28.3.17 l5 . . . . .	217
8.28.3.18 ldef . . . . .	217
8.28.3.19 ltot . . . . .	217
8.28.3.20 m11 . . . . .	217
8.28.3.21 m12 . . . . .	218
8.28.3.22 m12def . . . . .	218
8.28.3.23 m13 . . . . .	218
8.28.3.24 m1tot . . . . .	218
8.28.3.25 m21 . . . . .	218
8.28.3.26 m21def . . . . .	218
8.28.3.27 m22 . . . . .	219
8.28.3.28 m22def . . . . .	219
8.28.3.29 mdef . . . . .	219
8.28.3.30 mtot . . . . .	219

<b>9</b>	<b>Data Type Documentation</b>	<b>221</b>
9.1	<a href="#">inprod_analytic::atm_tensors Type Reference</a>	221
9.1.1	<a href="#">Detailed Description</a>	221
9.1.2	<a href="#">Member Data Documentation</a>	221
9.1.2.1	<a href="#">a</a>	221
9.1.2.2	<a href="#">b</a>	221
9.1.2.3	<a href="#">c</a>	222
9.1.2.4	<a href="#">d</a>	222
9.1.2.5	<a href="#">g</a>	222
9.1.2.6	<a href="#">s</a>	222
9.2	<a href="#">inprod_analytic::atm_wavenum Type Reference</a>	222
9.2.1	<a href="#">Detailed Description</a>	223
9.2.2	<a href="#">Member Data Documentation</a>	223
9.2.2.1	<a href="#">h</a>	223
9.2.2.2	<a href="#">m</a>	223
9.2.2.3	<a href="#">nx</a>	223
9.2.2.4	<a href="#">ny</a>	223
9.2.2.5	<a href="#">p</a>	223
9.2.2.6	<a href="#">typ</a>	223
9.3	<a href="#">tensor::coolist Type Reference</a>	224
9.3.1	<a href="#">Detailed Description</a>	224
9.3.2	<a href="#">Member Data Documentation</a>	224
9.3.2.1	<a href="#">elems</a>	224
9.3.2.2	<a href="#">nelems</a>	224
9.4	<a href="#">tensor::coolist4 Type Reference</a>	224
9.4.1	<a href="#">Detailed Description</a>	225
9.4.2	<a href="#">Member Data Documentation</a>	225
9.4.2.1	<a href="#">elems</a>	225
9.4.2.2	<a href="#">nelems</a>	225
9.5	<a href="#">tensor::coolist_elem Type Reference</a>	225



9.5.1	Detailed Description	226
9.5.2	Member Data Documentation	226
9.5.2.1	j	226
9.5.2.2	k	226
9.5.2.3	v	226
9.6	tensor::coolist_elem4 Type Reference	226
9.6.1	Detailed Description	227
9.6.2	Member Data Documentation	227
9.6.2.1	j	227
9.6.2.2	k	227
9.6.2.3	l	227
9.6.2.4	v	227
9.7	inprod_analytic::ocean_tensors Type Reference	227
9.7.1	Detailed Description	228
9.7.2	Member Data Documentation	228
9.7.2.1	c	228
9.7.2.2	k	228
9.7.2.3	m	228
9.7.2.4	n	228
9.7.2.5	o	229
9.7.2.6	w	229
9.8	inprod_analytic::ocean_wavenum Type Reference	229
9.8.1	Detailed Description	229
9.8.2	Member Data Documentation	229
9.8.2.1	h	229
9.8.2.2	nx	229
9.8.2.3	ny	230
9.8.2.4	p	230

<b>10 File Documentation</b>	<b>231</b>
10.1 aotensor_def.f90 File Reference	231
10.2 corr_tensor.f90 File Reference	232
10.3 corrmmod.f90 File Reference	232
10.4 dec_tensor.f90 File Reference	233
10.5 doc/def_doc.md File Reference	235
10.6 doc/gen_doc.md File Reference	235
10.7 doc/sto_doc.md File Reference	235
10.8 doc/tl_ad_doc.md File Reference	235
10.9 ic_def.f90 File Reference	235
10.10 inprod_analytic.f90 File Reference	235
10.11 int_comp.f90 File Reference	237
10.12 int_corr.f90 File Reference	237
10.13 LICENSE.txt File Reference	238
10.13.1 Function Documentation	239
10.13.1.1 files(the""Software"")	240
10.13.1.2 License(MIT) Copyright(c) 2015-2018 Lesley De Cruz and Jonathan Demaeyer Permission is hereby granted	240
10.13.2 Variable Documentation	240
10.13.2.1 charge	240
10.13.2.2 CLAIM	240
10.13.2.3 conditions	240
10.13.2.4 CONTRACT	240
10.13.2.5 copy	240
10.13.2.6 distribute	240
10.13.2.7 FROM	241
10.13.2.8 IMPLIED	241
10.13.2.9 KIND	241
10.13.2.10 LIABILITY	241
10.13.2.11 MERCHANTABILITY	241
10.13.2.12 merge	241

10.13.2.13modify . . . . .	241
10.13.2.14OTHERWISE . . . . .	242
10.13.2.15publish . . . . .	242
10.13.2.16restriction . . . . .	242
10.13.2.17so . . . . .	242
10.13.2.18Software . . . . .	242
10.13.2.19sublicense . . . . .	242
10.13.2.20use . . . . .	242
10.14maooam.f90 File Reference . . . . .	242
10.14.1 Function/Subroutine Documentation . . . . .	243
10.14.1.1 maooam . . . . .	243
10.15maooam_MTV.f90 File Reference . . . . .	243
10.15.1 Function/Subroutine Documentation . . . . .	243
10.15.1.1 maooam_mtv . . . . .	243
10.16maooam_stoch.f90 File Reference . . . . .	243
10.16.1 Function/Subroutine Documentation . . . . .	244
10.16.1.1 maooam_stoch . . . . .	244
10.17maooam_WL.f90 File Reference . . . . .	244
10.17.1 Function/Subroutine Documentation . . . . .	244
10.17.1.1 maooam_wl . . . . .	244
10.18MAR.f90 File Reference . . . . .	244
10.19memory.f90 File Reference . . . . .	245
10.20MTV_int_tensor.f90 File Reference . . . . .	246
10.21MTV_sigma_tensor.f90 File Reference . . . . .	247
10.22params.f90 File Reference . . . . .	248
10.23rk2_integrator.f90 File Reference . . . . .	251
10.24rk2_MTV_integrator.f90 File Reference . . . . .	251
10.25rk2_ss_integrator.f90 File Reference . . . . .	252
10.26rk2_stoch_integrator.f90 File Reference . . . . .	253
10.27rk2_tl_ad_integrator.f90 File Reference . . . . .	254

10.28rk2_WL_integrator.f90 File Reference . . . . .	254
10.29rk4_integrator.f90 File Reference . . . . .	255
10.30rk4_tl_ad_integrator.f90 File Reference . . . . .	256
10.31sf_def.f90 File Reference . . . . .	256
10.32sqrt_mod.f90 File Reference . . . . .	257
10.33stat.f90 File Reference . . . . .	258
10.34stoch_mod.f90 File Reference . . . . .	258
10.35stoch_params.f90 File Reference . . . . .	259
10.36tensor.f90 File Reference . . . . .	260
10.37test_aotensor.f90 File Reference . . . . .	263
10.37.1 Function/Subroutine Documentation . . . . .	263
10.37.1.1 test_aotensor . . . . .	263
10.38test_corr.f90 File Reference . . . . .	263
10.38.1 Function/Subroutine Documentation . . . . .	264
10.38.1.1 test_corr . . . . .	264
10.39test_corr_tensor.f90 File Reference . . . . .	264
10.39.1 Function/Subroutine Documentation . . . . .	264
10.39.1.1 test_corr_tensor . . . . .	264
10.40test_dec_tensor.f90 File Reference . . . . .	264
10.40.1 Function/Subroutine Documentation . . . . .	264
10.40.1.1 test_dec_tensor . . . . .	264
10.41test_inprod_analytic.f90 File Reference . . . . .	265
10.41.1 Function/Subroutine Documentation . . . . .	265
10.41.1.1 inprod_analytic_test . . . . .	265
10.42test_MAR.f90 File Reference . . . . .	265
10.42.1 Function/Subroutine Documentation . . . . .	265
10.42.1.1 test_mar . . . . .	265
10.43test_memory.f90 File Reference . . . . .	266
10.43.1 Function/Subroutine Documentation . . . . .	266
10.43.1.1 test_memory . . . . .	266

10.44test_MTV_int_tensor.f90 File Reference . . . . .	266
10.44.1 Function/Subroutine Documentation . . . . .	266
10.44.1.1 test_mtv_int_tensor . . . . .	266
10.45test_MTV_sigma_tensor.f90 File Reference . . . . .	266
10.45.1 Function/Subroutine Documentation . . . . .	267
10.45.1.1 test_sigma . . . . .	267
10.46test_sqrtm.f90 File Reference . . . . .	267
10.46.1 Function/Subroutine Documentation . . . . .	267
10.46.1.1 test_sqrtm . . . . .	267
10.47test_tl_ad.f90 File Reference . . . . .	267
10.47.1 Function/Subroutine Documentation . . . . .	267
10.47.1.1 test_tl_ad . . . . .	267
10.48test_WL_tensor.f90 File Reference . . . . .	268
10.48.1 Function/Subroutine Documentation . . . . .	268
10.48.1.1 test_wl_tensor . . . . .	268
10.49tl_ad_tensor.f90 File Reference . . . . .	268
10.50util.f90 File Reference . . . . .	269
10.50.1 Function/Subroutine Documentation . . . . .	270
10.50.1.1 lcg(s) . . . . .	270
10.51WL_tensor.f90 File Reference . . . . .	270
<b>Index</b>	<b>273</b>



## Chapter 1

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

### About

(c) 2013-2018 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](mailto:lesley.decruz@meteo.be), [jonathan.demaeyer@meteo.be](mailto:jonathan.demaeyer@meteo.be) and [svn@meteo.be](mailto:svn@meteo.be).

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

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

## Installation

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

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

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

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

and run:

```
1 make
```

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

```
1 make RES=store
```

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

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

## 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.
- [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:

- [maoam\\_stoch.f90](#) : Stochastic implementation of MAOOAM.
- [maoam\\_MTV.f90](#) : Main program - MTV implementation for MAOOAM.
- [maoam\\_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  $\text{OMS}(\text{NBOC}, 2)$  and  $\text{AMS}(\text{NBATM}, 2)$ .
- 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.

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

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

```
./maooam
```

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

```
./maooam
```

It will generate two files :

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

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

---

## 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

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

where  $\mathbf{x}$  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  $\delta$  is the timescale separation parameter.

The WL parameterizations for MAOOAM is given by

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

where  $\varepsilon$  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  $\varepsilon$  is the resolved-unresolved components coupling strength given by the parameter `stoch_params::eps_pert`.  $\delta$  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  $\Psi_x$  includes all the  $x$  terms, and thus  $F_x$  and  $\Psi_x$  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  $\delta$  is the timescale separation parameter.

## Correlations specification

The ingredients needed to compute the terms  $R, G, \sigma$  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  $\sigma_y$  is then simply the correlation matrix at the lagtime 0, and  $\Sigma$  and  $\Sigma_2$  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:

- 'mean.def' : Mean  $\langle \mathbf{y} \rangle$  of the unresolved variables.
- 'correxpo.def': Coefficients  $a_k$  of the fit of the elements of the correlations matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  with the function

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

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

- 'corrspline.def': Coefficients  $b_k$  of the spline used to model the elements of the correlation matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ . Used if `stoch_params::load_mode` is set to 'spli'.
- 'corrint.def': File holding the matrix  $\Sigma$ . Used if `stoch_params::int_corr_mode` is set to 'file'.
- 'corr2int.def': File holding the matrix  $\Sigma_2$ .

The various terms are then constructed according to these definition files. More details on the format of the definition files can be found [here](#).

### 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

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

where  $\varepsilon$  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 'corrspline.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  $a_k$  of the fit of the elements of the correlations matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  with the function

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

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

- 'corr spline.def': Coefficients  $b_k$  of the spline used to model the elements of the correlation matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ . 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. More details on the format of the definition files can be found [here](#).

## References

- Vannitsem, S., Demaeyer, J., De Cruz, L., and Ghil, M.: Low-frequency variability and heat transport in a low-order 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 $\leftrightarrow$ 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

# Modular arbitrary-order ocean-atmosphere model: Definition files formats

This page describes the format of the definition files needed by the stochastic model.

### MTV parameterization

The following definition files are needed by the MTV parameterization. Examples of those files are joined to the code. The files include:

- 'mean.def' : Mean  $\langle \mathbf{y} \rangle$  of the unresolved variables.
  - **Format**: one line per  $\langle y_i \rangle$  value
- 'correxpo.def': Coefficients  $a_k$  of the fit of the elements of the correlations matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  with the function

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

where  $t$  is the lag-time and  $\tau$  is the decorrelation time.

- **Format**: First line is two numbers: the number of unresolved variables and the value of `stoch_params::maxint` to be used (range of validity of the fit).  
Then each line specify the fit of an element  $i, j$  of the matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  as follow:

$$i, j, a_0, a_1, a_2, a_3$$

- Used if `stoch_params::load_mode` is set to 'expo'.
- 'corr spline.def': Coefficients  $b_k$  of the spline used to model the elements of the correlation matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ .
  - **Format**: First line is two numbers: the number of unresolved variables and the number of points used. Second line is the times  $\tau_k$  of the points in timeunits.  
Then  $i \times j$  sequences of 3 lines occurs as follow:
    1.  $i, j$
    2. Values of  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle_{i,j}$  at  $\tau_k$
    3. Coefficients  $b_k$  of the spline giving the second derivative of the interpolating function at  $\tau$
  - Used if `stoch_params::load_mode` is set to 'spli'.
- 'corr int.def': File holding the matrix  $\Sigma = \int_0^\infty ds \langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ .

- **Format:** Matrix in a Fortran-contiguous format
- Used if `stoch_params::int_corr_mode` is set to 'file'.
- 'corr2int.def': File holding the matrix  $\Sigma_2 = \int_0^\infty ds \langle (\mathbf{y} \otimes \mathbf{y}^s) \otimes (\mathbf{y} \otimes \mathbf{y}^s) \rangle$ .
  - **Format:** Matrix in a sparse format, `params::ndim` sequences with
    1. a first line with the first index  $i$  of the matrix and then the number of entries the sub-matrix  $\Sigma_{2,i,\dots}$  has
    2. a list of the entries of the matrix in the format:

$$i, j, k, l, v$$

where  $v$  is the value of the entry

## WL parameterization

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:

- 'mean.def' : Mean  $\langle \mathbf{y} \rangle$  of the unresolved variables.
  - **Format:** one line per  $\langle y_i \rangle$  value
- 'correxpo.def': Coefficients  $a_k$  of the fit of the elements of the correlations matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  with the function

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

where  $t$  is the lag-time and  $\tau$  is the decorrelation time.

- **Format:** First line is two numbers: the number of unresolved variables and the value of `stoch_params::maxint` to be used (range of validity of the fit).  
Then each line specify the fit of an element  $i, j$  of the matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$  as follow:
 
$$i, j, a_0, a_1, a_2, a_3$$
- Used if `stoch_params::load_mode` is set to 'expo'.
- 'corr spline.def': Coefficients  $b_k$  of the spline used to model the elements of the correlation matrix  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle$ .
  - **Format:** First line is two numbers: the number of unresolved variables and the number of points used. Second line is the times  $\tau_k$  of the points in timeunits.  
Then  $i \times j$  sequences of 3 lines occurs as follow:
    1.  $i, j$
    2. Values of  $\langle \mathbf{y} \otimes \mathbf{y}^s \rangle_{i,j}$  at  $\tau_k$
    3. Coefficients  $b_k$  of the spline giving the second derivative of the interpolating function at  $\tau$
  - 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.
  - **Format:** Matrix in a Fortran-contiguous format
- 'MAR\_W\_params.def': File specifying the  $\mathbf{W}_i$  matrices for the MAR.
  - **Format:** Matrix in a Fortran-contiguous format



## Chapter 5

# Modules Index

### 5.1 Modules List

Here is a list of all modules with brief descriptions:

<a href="#">aotensor_def</a>	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 . . . . .	25
<a href="#">corr_tensor</a>	Module to compute the correlations and derivatives used to compute the memory term of the WL parameterization . . . . .	29
<a href="#">corrmod</a>	Module to initialize the correlation matrix of the unresolved variables . . . . .	33
<a href="#">dec_tensor</a>	The resolved-unresolved components decomposition of the tensor . . . . .	44
<a href="#">ic_def</a>	Module to load the initial condition . . . . .	55
<a href="#">inprod_analytic</a>	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 . . . . .	57
<a href="#">int_comp</a>	Utility module containing the routines to perform the integration of functions . . . . .	68
<a href="#">int_corr</a>	Module to compute or load the integrals of the correlation matrices . . . . .	72
<a href="#">integrator</a>	Module with the integration routines . . . . .	76
<a href="#">mar</a>	Multidimensional Autoregressive module to generate the correlation for the WL parameterization . . . . .	80
<a href="#">memory</a>	Module that compute the memory term $M_3$ of the WL parameterization . . . . .	84
<a href="#">mtv_int_tensor</a>	The MTV tensors used to integrate the MTV model . . . . .	88
<a href="#">params</a>	The model parameters module . . . . .	97
<a href="#">rk2_mtv_integrator</a>	Module with the MTV rk2 integration routines . . . . .	112

<a href="#">rk2_ss_integrator</a>	Module with the stochastic uncoupled resolved nonlinear and tangent linear rk2 dynamics integration routines . . . . .	120
<a href="#">rk2_stoch_integrator</a>	Module with the stochastic rk2 integration routines . . . . .	124
<a href="#">rk2_wl_integrator</a>	Module with the WL rk2 integration routines . . . . .	128
<a href="#">sf_def</a>	Module to select the resolved-unresolved components . . . . .	134
<a href="#">sigma</a>	The MTV noise sigma matrices used to integrate the MTV model . . . . .	139
<a href="#">sqrt_mod</a>	Utility module with various routine to compute matrix square root . . . . .	143
<a href="#">stat</a>	Statistics accumulators . . . . .	150
<a href="#">stoch_mod</a>	Utility module containing the stochastic related routines . . . . .	152
<a href="#">stoch_params</a>	The stochastic models parameters module . . . . .	156
<a href="#">tensor</a>	Tensor utility module . . . . .	162
<a href="#">tl_ad_integrator</a>	Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Integrators module . . . .	192
<a href="#">tl_ad_tensor</a>	Tangent Linear (TL) and Adjoint (AD) model versions of MAOOAM. Tensors definition module .	195
<a href="#">util</a>	Utility module . . . . .	203
<a href="#">wl_tensor</a>	The WL tensors used to integrate the model . . . . .	209

## Chapter 6

# Data Type Index

### 6.1 Data Types List

Here are the data types with brief descriptions:

<a href="#">inprod_analytic::atm_tensors</a>	
Type holding the atmospheric inner products tensors . . . . .	<a href="#">221</a>
<a href="#">inprod_analytic::atm_wavenum</a>	
Atmospheric bloc specification type . . . . .	<a href="#">222</a>
<a href="#">tensor::coolist</a>	
Coordinate list. Type used to represent the sparse tensor . . . . .	<a href="#">224</a>
<a href="#">tensor::coolist4</a>	
4d coordinate list. Type used to represent the rank-4 sparse tensor . . . . .	<a href="#">224</a>
<a href="#">tensor::coolist_elem</a>	
Coordinate list element type. Elementary elements of the sparse tensors . . . . .	<a href="#">225</a>
<a href="#">tensor::coolist_elem4</a>	
4d coordinate list element type. Elementary elements of the 4d sparse tensors . . . . .	<a href="#">226</a>
<a href="#">inprod_analytic::ocean_tensors</a>	
Type holding the oceanic inner products tensors . . . . .	<a href="#">227</a>
<a href="#">inprod_analytic::ocean_wavenum</a>	
Oceanic bloc specification type . . . . .	<a href="#">229</a>



## Chapter 7

# File Index

### 7.1 File List

Here is a list of all files with brief descriptions:

aotensor_def.f90	231
corr_tensor.f90	232
corrmod.f90	232
dec_tensor.f90	233
ic_def.f90	235
inprod_analytic.f90	235
int_comp.f90	237
int_corr.f90	237
maooam.f90	242
maooam_MTV.f90	243
maooam_stoch.f90	243
maooam_WL.f90	244
MAR.f90	244
memory.f90	245
MTV_int_tensor.f90	246
MTV_sigma_tensor.f90	247
params.f90	248
rk2_integrator.f90	251
rk2_MTV_integrator.f90	251
rk2_ss_integrator.f90	252
rk2_stoch_integrator.f90	253
rk2_tl_ad_integrator.f90	254
rk2_WL_integrator.f90	254
rk4_integrator.f90	255
rk4_tl_ad_integrator.f90	256
sf_def.f90	256
sqr_mod.f90	257
stat.f90	258
stoch_mod.f90	258
stoch_params.f90	259
tensor.f90	260
test_aotensor.f90	263
test_corr.f90	263
test_corr_tensor.f90	264
test_dec_tensor.f90	264

test_inprod_analytic.f90	265
test_MAR.f90	265
test_memory.f90	266
test_MTV_int_tensor.f90	266
test_MTV_sigma_tensor.f90	266
test_sqrtm.f90	267
test_tl_ad.f90	267
test_WL_tensor.f90	268
tl_ad_tensor.f90	268
util.f90	269
WL_tensor.f90	270

## Chapter 8

# Module Documentation

### 8.1 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.*

### 8.1.1 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

### 8.1.2 Function/Subroutine Documentation

#### 8.1.2.1 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
```

#### 8.1.2.2 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
```

#### 8.1.2.3 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

```

#### 8.1.2.4 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.

#### 8.1.2.5 subroutine, public aotensor\_def::init\_aotensor ( )

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

## Remarks

This procedure will also call [params::init\\_params\(\)](#) and [inprod\\_analytic::init\\_inprod\(\)](#) . It will finally call [inprod\\_analytic::deallocate\\_inprod\(\)](#) to remove the inner products, which are not needed anymore at this point.

Definition at line 203 of file aotensor\_def.f90.

```

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

```

```

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

```

#### 8.1.2.6 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

```

#### 8.1.2.7 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

```

#### 8.1.2.8 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

```

#### 8.1.2.9 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

```

### 8.1.3 Variable Documentation

#### 8.1.3.1 `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
```

#### 8.1.3.2 `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
```

#### 8.1.3.3 `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
```

## 8.2 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  $s$ .*

### 8.2.1 Detailed Description

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.2.2 Function/Subroutine Documentation

#### 8.2.2.1 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

```

### 8.2.3 Variable Documentation

#### 8.2.3.1 `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

```

#### 8.2.3.2 `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

```

### 8.2.3.3 `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
```

### 8.2.3.4 `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$\langle \partial_Y \otimes Y^s \rangle$ terms
```

### 8.2.3.5 `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$\langle \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms
```

### 8.2.3.6 `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$s\f$
```

### 8.2.3.7 `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$\langle Y \otimes \partial_Y \otimes Y^s \rangle$ terms
```

### 8.2.3.8 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 :: yddyy !< Coolist holding the  \f$\langle Y \otimes
    \partial_Y \otimes Y^s \otimes Y^s \rangle$ terms
```

### 8.2.3.9 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
```

## 8.3 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.*

### 8.3.1 Detailed Description

Module to initialize the correlation matrix of the unresolved variables.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.3.2 Function/Subroutine Documentation

#### 8.3.2.1 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)=((-2.471792023403353*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(4,4)=((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

```

```

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

```

### 8.3.2.2 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

<b>s</b>	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

```

### 8.3.2.3 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

<b>s</b>	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

```

### 8.3.2.4 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

```

### 8.3.2.5 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

```

**8.3.2.6 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

```

### 8.3.3 Variable Documentation

#### 8.3.3.1 `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)

```

#### 8.3.3.2 `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)

```

#### 8.3.3.3 `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

```

**8.3.3.4 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
```

**8.3.3.5 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)
```

**8.3.3.6 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)
```

**8.3.3.7 integer corrmmod::khi [private]**

Definition at line 38 of file corrmmod.f90.

**8.3.3.8 integer corrmmod::klo [private]**

Definition at line 38 of file corrmmod.f90.

**8.3.3.9 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)
```



**8.3.3.10** `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)
```

**8.3.3.11** `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
```

**8.3.3.12** `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
```

**8.3.3.13** `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
```

**8.3.3.14** `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
```

## 8.4 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.*

### 8.4.1 Detailed Description

The resolved-unresolved components decomposition of the tensor.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.4.2 Function/Subroutine Documentation

#### 8.4.2.1 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\_coo, 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_coo(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_coo(dumb,ff_tensor)
233 ELSE
234     CALL copy_coo(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_coo(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_coo(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_coo(dumb,fs_tensor)
253 ELSE
254     CALL copy_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(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_coo(dumb,bxyy)
507
508     allocstat=0
509     DEALLOCATE(dumb, stat=allocstat)
510     IF (allocstat /= 0) stop "*** Problem to deallocate ! ***"
511
512
513

```

#### 8.4.2.2 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

$t$	tensor over which the routine acts
$cst$	constant which controls if the 0 index is taken as a unresolved or a resolved one
$v$	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)%nelems=0
183     ENDDO
184

```

#### 8.4.2.3 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

$t$	tensor over which the routine acts
$cst$	constant which controls if the 0 index is taken as a unresolved or a resolved one
$v$	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

```

**8.4.2.4** 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)%elems
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)%elems, 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)%elems=t(i)%elems-1
98                 IF (li>t(i)%elems) exit
99             ENDDO
100             IF (li>t(i)%elems) exit
101         ENDDO
102     ENDDO
103

```

**8.4.2.5** 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 
```

### 8.4.3 Variable Documentation

#### 8.4.3.1 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

```

#### 8.4.3.2 type(coolist), dimension(:), allocatable, public dec\_tensor::bxyy

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 :: bxyy !< Tensor holding the quadratic part of
        the resolved tendencies involving both resolved and unresolved variables

```

#### 8.4.3.3 `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
```

#### 8.4.3.4 `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
```

#### 8.4.3.5 `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
```

#### 8.4.3.6 `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
```

#### 8.4.3.7 `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
```

**8.4.3.8 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
```

**8.4.3.9 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
```

**8.4.3.10 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
```

**8.4.3.11 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
```

**8.4.3.12 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
```

**8.4.3.13 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
```

**8.4.3.14 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
```

**8.4.3.15 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
```

**8.4.3.16 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
```

**8.4.3.17 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
```

8.4.3.18 `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
```

## 8.5 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.*

### 8.5.1 Detailed Description

Module to load the initial condition.

#### Copyright

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

### 8.5.2 Function/Subroutine Documentation

8.5.2.1 `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

```

### 8.5.3 Variable Documentation

#### 8.5.3.1 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.
```

#### 8.5.3.2 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
```

## 8.6 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.*
- real(kind=8) function [calculate\\_a](#) (i, j)  
*Eigenvalues of the Laplacian (atmospheric)*
- real(kind=8) function [calculate\\_b](#) (i, j, k)  
*Streamfunction advection terms (atmospheric)*
- real(kind=8) function [calculate\\_c\\_atm](#) (i, j)  
*Beta term for the atmosphere.*
- real(kind=8) function [calculate\\_d](#) (i, j)  
*Forcing of the ocean on the atmosphere.*
- real(kind=8) function [calculate\\_g](#) (i, j, k)  
*Temperature advection terms (atmospheric)*
- real(kind=8) function [calculate\\_s](#) (i, j)  
*Forcing (thermal) of the ocean on the atmosphere.*
- real(kind=8) function [calculate\\_k](#) (i, j)  
*Forcing of the atmosphere on the ocean.*
- real(kind=8) function [calculate\\_m](#) (i, j)  
*Forcing of the ocean fields on the ocean.*
- real(kind=8) function [calculate\\_n](#) (i, j)  
*Beta term for the ocean.*
- real(kind=8) function [calculate\\_o](#) (i, j, k)  
*Temperature advection term (passive scalar)*
- real(kind=8) function [calculate\\_c\\_oc](#) (i, j, k)



*Streamfunction advection terms (oceanic)*

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

*Short-wave radiative forcing of the ocean.*

- subroutine, public `init_inprod`

*Initialisation of the inner product.*

## Variables

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

*Atmospheric blocs specification.*

- `type(ocean_wavenum)`, `dimension(:)`, allocatable, public `owavenum`

*Oceanic blocs specification.*

- `type(atm_tensors)`, public `atmos`

*Atmospheric tensors.*

- `type(ocean_tensors)`, public `ocean`

*Oceanic tensors.*

### 8.6.1 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

### 8.6.2 Function/Subroutine Documentation

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

Cehelsky & Tung Helper functions.

Definition at line 100 of file inprod\_analytic.f90.

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

#### 8.6.2.2 `real(kind=8)` function `inprod_analytic::b2` ( `integer Pi`, `integer Pj`, `integer Pk` ) [`private`]

Cehelsky & Tung Helper functions.

Definition at line 106 of file inprod\_analytic.f90.

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

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

Eigenvalues of the Laplacian (atmospheric)

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

Definition at line 164 of file inprod\_analytic.f90.

```

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

```

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

Streamfunction advection terms (atmospheric)

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

Definition at line 178 of file inprod\_analytic.f90.

```

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

```

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

Beta term for the atmosphere.

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

Definition at line 188 of file inprod\_analytic.f90.

```

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

```

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

Streamfunction advection terms (oceanic)

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

Definition at line 412 of file inprod\_analytic.f90.

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

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

Forcing of the ocean on the atmosphere.

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

Definition at line 208 of file inprod\_analytic.f90.

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

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

Temperature advection terms (atmospheric)

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

Definition at line 218 of file inprod\_analytic.f90.

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

```

244     vs3 = s3(tj%P,tk%P,tj%H,tk%H)
245     vs4 = s4(tj%P,tk%P,tj%H,tk%H)
246     val = vs3 * ((delta(tk%H - tj%H - ti%H) - delta(tk%H &
247         &- tj%H + ti%H)) * delta(tk%P + tj%P - ti%P) +&
248         & delta(tk%H + tj%H - ti%H) * (delta(tk%P - tj%P&
249         & + ti%P) - delta(tk%P - tj%P - ti%P))) + vs4 * &
250         & ((delta(tk%H + tj%H - ti%H) * delta(tk%P - tj&
251         &%P - ti%P)) + (delta(tk%H - tj%H + ti%H) -&
252         & delta(tk%H - tj%H - ti%H)) * (delta(tk%P - tj&
253         &%P - ti%P) - delta(tk%P - tj%P + ti%P)))
254 ELSE
255
256     s(1)=ti%typ
257     s(2)=tj%typ
258     s(3)=tk%typ
259
260     w(1,:)=isin("A",s)
261     w(2,:)=isin("K",s)
262     w(3,:)=isin("L",s)
263
264     IF (any(w(1,)/=0) .AND. any(w(2,)/=0) .AND. any(w(3,)/=0)) THEN
265         b=w(:,1)
266         ti = awavenum(a(b(1)))
267         tj = awavenum(a(b(2)))
268         tk = awavenum(a(b(3)))
269         call piksrt(3,b,par)
270         vb1 = b1(ti%P,tj%P,tk%P)
271         vb2 = b2(ti%P,tj%P,tk%P)
272         val = -2 * sqrt(2.) / pi * tj%M * delta(tj%M - tk%H) * flambda(ti%P + tj%P + tk%P)
273         IF (val /= 0.d0) val = val * (vb1**2 / (vb1**2 - 1) - vb2**2 / (vb2**2 - 1))
274     ELSEIF ((w(2,2)/=0) .AND. (w(2,3)==0) .AND. any(w(3,)/=0)) THEN
275         ti = awavenum(a(w(2,1)))
276         tj = awavenum(a(w(2,2)))
277         tk = awavenum(a(w(3,1)))
278         b(1)=w(2,1)
279         b(2)=w(2,2)
280         b(3)=w(3,1)
281         call piksrt(3,b,par)
282         vs1 = s1(tj%P,tk%P,tj%M,tk%H)
283         vs2 = s2(tj%P,tk%P,tj%M,tk%H)
284         val = vs1 * (delta(ti%M - tk%H - tj%M) * delta(ti%P -&
285             & tk%P + tj%P) - delta(ti%M- tk%H - tj%M) * &
286             & delta(ti%P + tk%P - tj%P) + (delta(tk%H - tj%M&
287             & + ti%M) + delta(tk%H - tj%M - ti%M)) * &
288             & delta(tk%P + tj%P - ti%P)) + vs2 * (delta(ti%M&
289             & - tk%H - tj%M) * delta(ti%P - tk%P - tj%P) +&
290             & (delta(tk%H - tj%M - ti%M) + delta(ti%M + tk%H&
291             & - tj%M)) * (delta(ti%P - tk%P + tj%P) -&
292             & delta(tk%P - tj%P + ti%P)))
293     ENDIF
294 ENDIF
295 calculate_g=par*val*n
296

```

### 8.6.2.9 real(kind=8) function inprod\_analytic::calculate\_k ( integer, intent(in) i, integer, intent(in) j ) [private]

Forcing of the atmosphere on the ocean.

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

Definition at line 336 of file inprod\_analytic.f90.

```

336     INTEGER, INTENT(IN) :: i,j
337
338     calculate_k = calculate_s(j,i) * calculate_a(j,j)

```

### 8.6.2.10 real(kind=8) function inprod\_analytic::calculate\_m ( integer, intent(in) i, integer, intent(in) j ) [private]

Forcing of the ocean fields on the ocean.

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

Definition at line 345 of file inprod\_analytic.f90.

```

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

```

#### 8.6.2.11 real(kind=8) function inprod\_analytic::calculate\_n ( integer, intent(in) i, integer, intent(in) j ) [private]

Beta term for the ocean.

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

Definition at line 359 of file inprod\_analytic.f90.

```

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

```

#### 8.6.2.12 real(kind=8) function inprod\_analytic::calculate\_o ( integer, intent(in) i, integer, intent(in) j, integer, intent(in) k ) [private]

Temperature advection term (passive scalar)

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

Definition at line 377 of file inprod\_analytic.f90.

```

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

```

### 8.6.2.13 real(kind=8) function inprod\_analytic::calculate\_s ( integer, intent(in) i, integer, intent(in) j ) [private]

Forcing (thermal) of the ocean on the atmosphere.

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

Definition at line 303 of file inprod\_analytic.f90.

```

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

### 8.6.2.14 real(kind=8) function inprod\_analytic::calculate\_w ( integer, intent(in) i, integer, intent(in) j ) [private]

Short-wave radiative forcing of the ocean.

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

Definition at line 422 of file inprod\_analytic.f90.

```

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

### 8.6.2.15 real(kind=8) function inprod\_analytic::delta ( integer r ) [private]

Integer Dirac delta function.

Definition at line 112 of file inprod\_analytic.f90.

```

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

**8.6.2.16** real(kind=8) function inprod\_analytic::flambda ( integer *r* ) [private]

"Odd or even" function

Definition at line 122 of file inprod\_analytic.f90.

```

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

```

**8.6.2.17** subroutine, public inprod\_analytic::init\_inprod ( )

Initialisation of the inner product.

Definition at line 436 of file inprod\_analytic.f90.

```

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

```

```

492      owavenum(i)%P=oms(i,2)
493
494      owavenum(i)%Nx=oms(i,1)/2.d0
495      owavenum(i)%Ny=oms(i,2)
496
497      ENDDO
498
499      ! Pointing to the atmospheric inner products functions
500
501      atmos%a => calculate_a
502      atmos%g => calculate_g
503      atmos%s => calculate_s
504      atmos%b => calculate_b
505      atmos%d => calculate_d
506      atmos%c => calculate_c_atm
507
508      ! Pointing to the oceanic inner products functions
509
510      ocean%M => calculate_m
511      ocean%N => calculate_n
512      ocean%O => calculate_o
513      ocean%C => calculate_c_oc
514      ocean%W => calculate_w
515      ocean%K => calculate_k
516

```

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

Cehelsky & Tung Helper functions.

Definition at line 132 of file inprod\_analytic.f90.

```

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

```

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

Cehelsky & Tung Helper functions.

Definition at line 138 of file inprod\_analytic.f90.

```

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

```

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

Cehelsky & Tung Helper functions.

Definition at line 144 of file inprod\_analytic.f90.

```

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

```



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

Cehelsky & Tung Helper functions.

Definition at line 150 of file inprod\_analytic.f90.

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

### 8.6.3 Variable Documentation

**8.6.3.1** `type(atm_tensors), public inprod_analytic::atmos`

Atmospheric tensors.

Definition at line 78 of file inprod\_analytic.f90.

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

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

Atmospheric blocs specification.

Definition at line 73 of file inprod\_analytic.f90.

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

**8.6.3.3** `type(ocean_tensors), public inprod_analytic::ocean`

Oceanic tensors.

Definition at line 80 of file inprod\_analytic.f90.

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

**8.6.3.4** `type(ocean_wavenum), dimension(:), allocatable, public inprod_analytic::owavenum`

Oceanic blocs specification.

Definition at line 75 of file inprod\_analytic.f90.

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

## 8.7 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 0 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](#) (func, aa, bb, s, n)  
*Midpoint routine for bb infinite with func decreasing infinitely rapidly at infinity.*

### 8.7.1 Detailed Description

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

Most are taken from the Numerical Recipes

### 8.7.2 Function/Subroutine Documentation

#### 8.7.2.1 subroutine, public int\_comp::integrate ( external func, real(kind=8) ss )

Routine to compute integrals of function from 0 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)

```

**8.7.2.2** 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 funk 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

```

**8.7.2.3** 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

```

**8.7.2.4** 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

```

**8.7.2.5** 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'

```

**8.7.2.6** 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'

```

**8.7.2.7** 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

```

## 8.8 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.*

### 8.8.1 Detailed Description

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.8.2 Function/Subroutine Documentation

#### 8.8.2.1 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                  CALL integrate(func_ij,ss)
96                  corrint(ind(i),ind(j))=ss
97              END DO
98          END DO
99
100         OPEN(30,file='corrint.def')
```

```

101      WRITE(30,*) corr2int
102      CLOSE(30)
103  END SELECT
104
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
134              n=0
135              DO j= 1,n_unres
136                  DO k= 1,n_unres
137                      DO l = 1,n_unres
138                          oi=i
139                          oj=j
140                          ok=k
141                          ol=l
142
143                          CALL integrate(func_ijkl,ss)
144                          IF (abs(ss)>real_eps) THEN
145                              n=n+1
146                              corr2int(ind(i))%elems(n)%j=ind(j)
147                              corr2int(ind(i))%elems(n)%k=ind(k)
148                              corr2int(ind(i))%elems(n)%l=ind(l)
149                              corr2int(ind(i))%elems(n)%v=ss
150                          END IF
151                      ENDDO
152                  ENDDO
153              ENDDO
154              corr2int(ind(i))%elems=n
155          END IF
156      ENDDO
157
158      CALL write_tensor4_to_file("corr2int.def",corr2int)
159  CASE DEFAULT
160      stop '*** INT_CORR_MODE variable not properly defined in corrmmod.nml ***'
161  END SELECT
162

```

### 8.8.2.2 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

s	time at which the function is evaluated
---	---

Definition at line 55 of file int\_corr.f90.



```

56     REAL(KIND=8) :: s,func_ij
57     CALL corrcomp(s)
58     func_ij=corr_ij(oi,oj)
59     RETURN

```

#### 8.8.2.3 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

s	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

```

#### 8.8.2.4 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

```

### 8.8.3 Variable Documentation

#### 8.8.3.1 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

```

### 8.8.3.2 `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
```

### 8.8.3.3 `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
```

### 8.8.3.4 `integer int_corr::oj [private]`

Definition at line 26 of file `int_corr.f90`.

### 8.8.3.5 `integer int_corr::ok [private]`

Definition at line 26 of file `int_corr.f90`.

### 8.8.3.6 `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`.

### 8.8.3.7 `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
```

## 8.9 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.*

### 8.9.1 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.

## 8.9.2 Function/Subroutine Documentation

### 8.9.2.1 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 ! ***"
```

### 8.9.2.2 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
```

### 8.9.2.3 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)
```

**8.9.3 Variable Documentation****8.9.3.1 `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
```

**8.9.3.2 `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
```

**8.9.3.3 `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
```

**8.9.3.4 `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
```

### 8.9.3.5 `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)
```

## 8.10 `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*

### 8.10.1 Detailed Description

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

#### Copyright

2018 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

## 8.10.2 Function/Subroutine Documentation

### 8.10.2.1 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!'

```

### 8.10.2.2 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)

```

### 8.10.2.3 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)

```

### 8.10.2.4 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

```

## 8.10.3 Variable Documentation

### 8.10.3.1 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

```



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

Definition at line 34 of file MAR.f90.

**8.10.3.3** `integer, public mar::ms`

order of the MAR

Definition at line 36 of file MAR.f90.

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

**8.10.3.4** `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
```

**8.10.3.5** `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
```

**8.10.3.6** `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
```

**8.10.3.7** `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
```

### 8.10.3.8 `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
```

## 8.11 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  $X_s$  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.*

### 8.11.1 Detailed Description

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

## 8.11.2 Function/Subroutine Documentation

**8.11.2.1** 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     ! Computing the integral
117     h_int=0.d0
118
119     DO i=1,mems
120         j=modulo(t_index+i-2,mems)+1
121         buf_m3=0.d0
122         IF (l1def) THEN
123             CALL sparse_mul3(ltot(:,i),y,x(:,j),buf_m)
124             buf_m3=buf_m3+buf_m
125         ENDIF
126         IF (b14def) THEN
127             CALL sparse_mul3(b14(:,i),x(:,j),x(:,j),buf_m)
128             buf_m3=buf_m3+buf_m
129         ENDIF
130         IF (b23def) THEN
131             CALL sparse_mul3(b23(:,i),x(:,j),zs(:,j),buf_m)
132             buf_m3=buf_m3+buf_m
133         ENDIF
134     
```

```

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

```

### 8.11.2.2 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

```

### 8.11.2.3 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

```

### 8.11.3 Variable Documentation

#### 8.11.3.1 `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

```

#### 8.11.3.2 `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

```

#### 8.11.3.3 `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

```

#### 8.11.3.4 `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)

```

#### 8.11.3.5 `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
```

#### 8.11.3.6 `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
```

#### 8.11.3.7 `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
```

## 8.12 `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.*

### 8.12.1 Detailed Description

The MTV tensors used to integrate the MTV model.

#### Copyright

2018 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.

### 8.12.2 Function/Subroutine Documentation

#### 8.12.2.1 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_corrint
95     print*, "Computing the correlation integrated matrices and tensors..."
96     CALL comp_corrint
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,corrint)
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
136

```



```

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,bxyx,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,bxyx,dumb_mat1)
253     CALL coo_to_mat_i(j,bxyx,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,bxyx,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,bxyx,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,bxyx,dumb_mat1)
289     CALL coo_to_mat_i(j,bxyx,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

```

## 8.12.3 Variable Documentation

**8.12.3.1 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
```

**8.12.3.2 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
```

**8.12.3.3 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
```

**8.12.3.4 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
```

**8.12.3.5 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
```

**8.12.3.6** `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
```

**8.12.3.7** `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
```

**8.12.3.8** `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
```

**8.12.3.9** `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
```

**8.12.3.10** `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
```

#### 8.12.3.11 `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
```

#### 8.12.3.12 `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
```

#### 8.12.3.13 `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
```

#### 8.12.3.14 `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
```

#### 8.12.3.15 `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
```

**8.12.3.16 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
```

**8.12.3.17 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
```

**8.12.3.18 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
```

**8.12.3.19 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
```

**8.12.3.20 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
```

8.12.3.21 `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
```

## 8.13 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`  
 $\sigma_0$  - 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`  
 $\lambda$  - 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](#)  
 $\gamma_o$  - 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](#)  
 $\epsilon_a$  - Emissivity coefficient for the grey-body atmosphere.
- real(kind=8) [ga](#)  
 $\gamma_a$  - 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](#)  
 $\pi$
- real(kind=8) [lr](#)  
 $L_R$  - Rossby deformation radius
- real(kind=8) [g](#)  
 $\gamma$
- 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](#)  
 $\lambda'_o$  - 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](#)  
 $\lambda'_a$  - 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](#)
- *$\beta'$  - Non-dimensional beta parameter*  
real(kind=8) [nua](#) =0.D0
- *Dissipation in the atmosphere.*  
real(kind=8) [nuo](#) =0.D0
- *Dissipation in the ocean.*  
real(kind=8) [nuap](#)
- *Non-dimensional dissipation in the atmosphere.*  
real(kind=8) [nuop](#)
- *Non-dimensional dissipation in the ocean.*  
real(kind=8) [t\\_trans](#)
- *Transient time period.*  
real(kind=8) [t\\_run](#)
- *Effective intergration time (length of the generated trajectory)*  
real(kind=8) [dt](#)
- *Integration time step.*  
real(kind=8) [tw](#)
- *Write all variables every tw time units.*  
logical [writeout](#)
- *Write to file boolean.*  
integer [nboc](#)
- *Number of atmospheric blocks.*  
integer [nbatm](#)
- *Number of oceanic blocks.*  
integer [natm](#) =0
- *Number of atmospheric basis functions.*  
integer [noc](#) =0
- *Number of oceanic basis functions.*  
integer [ndim](#)
- *Number of variables (dimension of the model)*  
integer, dimension(:,:), allocatable [oms](#)
- *Ocean mode selection array.*  
integer, dimension(:,:), allocatable [ams](#)
- *Atmospheric mode selection array.*

### 8.13.1 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

## 8.13.2 Function/Subroutine Documentation

### 8.13.2.1 subroutine, private params::init\_nml ( ) [private]

Read the basic parameters and mode selection from the namelist.

Definition at line 97 of file params.f90.

```

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

#### 8.13.2.2 subroutine params::init\_params ( )

Parameters initialisation routine.

Definition at line 138 of file params.f90.

```

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

```

```

159
160     ndim=2*natm+2*noc
161
162     !-----!
163     !
164     ! Some general parameters (Domain, beta, gamma, coupling) !
165     !
166     !-----!
167
168     pi=dacos(-1.d0)
169     l=scale/pi
170     phi0=phi0_npi*pi
171     lr=sqrt(gp*h)/f0
172     g=-1**2/lr**2
173     betp=1/rra*cos(phi0)/sin(phi0)
174     rp=r/f0
175     dp=d/f0
176     kd=k*2
177     kdp=kp
178
179     !-----!
180     !
181     ! DERIVED QUANTITIES
182     !
183     !-----!
184
185     cpo=co/(go*f0) * rr/(f0**2*1**2)
186     lpo=lambda/(go*f0)
187     cpa=ca/(ga*f0) * rr/(f0**2*1**2)/2 ! Cpa acts on psil-psi3, not on theta
188     lpa=lambda/(ga*f0)
189     sbpo=4*sb*to0**3/(go*f0) ! long wave radiation lost by ocean to atmosphere space
190     sbpa=8*epsa*sb*ta0**3/(go*f0) ! long wave radiation from atmosphere absorbed by ocean
191     lsbpo=2*epsa*sb*to0**3/(ga*f0) ! long wave radiation from ocean absorbed by atmosphere
192     lsbpa=8*epsa*sb*ta0**3/(ga*f0) ! long wave radiation lost by atmosphere to space & ocea
193     nuap=nua/(f0*1**2)
194     nuop=nuo/(f0*1**2)
195

```

### 8.13.3 Variable Documentation

#### 8.13.3.1 integer, dimension(:, :), allocatable params::ams

Atmospheric mode selection array.

Definition at line 87 of file params.f90.

```

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

```

#### 8.13.3.2 real(kind=8) params::betp

$\beta'$  - Non-dimensional beta parameter

Definition at line 67 of file params.f90.

```

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

```

#### 8.13.3.3 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.

```

#### 8.13.3.4 `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.
```

#### 8.13.3.5 `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.
```

#### 8.13.3.6 `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.
```

#### 8.13.3.7 `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.
```

#### 8.13.3.8 `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.
```

**8.13.3.9 real(kind=8) params::dt**

Integration time step.

Definition at line 77 of file params.f90.

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

**8.13.3.10 real(kind=8) params::epsa**

$\epsilon_a$  - 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.
```

**8.13.3.11 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
```

**8.13.3.12 real(kind=8) params::g**

$\gamma$

Definition at line 50 of file params.f90.

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

**8.13.3.13 real(kind=8) params::ga**

$\gamma_a$  - 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.
```

**8.13.3.14 real(kind=8) params::go**

$\gamma_o$  - 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.
```

**8.13.3.15 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
```

**8.13.3.16 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.
```

**8.13.3.17 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.
```

**8.13.3.18 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.
```

**8.13.3.19 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.
```

**8.13.3.20 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.
```

**8.13.3.21 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
```

**8.13.3.22 real(kind=8) params::lambda**

$\lambda$  - 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.
```

**8.13.3.23 real(kind=8) params::lpa**

$\lambda'_a$  - 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.
```

#### 8.13.3.24 `real(kind=8) params::lpo`

$\lambda'_o$  - 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.
```

#### 8.13.3.25 `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
```

#### 8.13.3.26 `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.
```

#### 8.13.3.27 `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.
```

#### 8.13.3.28 `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
```



**8.13.3.29 integer params::natm =0**

Number of atmospheric basis functions.

Definition at line 83 of file params.f90.

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

**8.13.3.30 integer params::nbatm**

Number of oceanic blocks.

Definition at line 82 of file params.f90.

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

**8.13.3.31 integer params::nboc**

Number of atmospheric blocks.

Definition at line 81 of file params.f90.

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

**8.13.3.32 integer params::ndim**

Number of variables (dimension of the model)

Definition at line 85 of file params.f90.

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

**8.13.3.33 integer params::noc =0**

Number of oceanic basis functions.

Definition at line 84 of file params.f90.

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

#### 8.13.3.34 `real(kind=8) params::nua =0.D0`

Dissipation in the atmosphere.

Definition at line 69 of file params.f90.

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

#### 8.13.3.35 `real(kind=8) params::nuap`

Non-dimensional dissipation in the atmosphere.

Definition at line 72 of file params.f90.

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

#### 8.13.3.36 `real(kind=8) params::nuo =0.D0`

Dissipation in the ocean.

Definition at line 70 of file params.f90.

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

#### 8.13.3.37 `real(kind=8) params::nuop`

Non-dimensional dissipation in the ocean.

Definition at line 73 of file params.f90.

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

#### 8.13.3.38 `integer, dimension(:, :), allocatable params::oms`

Ocean mode selection array.

Definition at line 86 of file params.f90.

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

**8.13.3.39 real(kind=8) params::phi0**

Latitude in radian.

Definition at line 25 of file params.f90.

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

**8.13.3.40 real(kind=8) params::phi0\_npi**

Latitude exprimed in fraction of pi.

Definition at line 35 of file params.f90.

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

**8.13.3.41 real(kind=8) params::pi**

$\pi$

Definition at line 48 of file params.f90.

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

**8.13.3.42 real(kind=8) params::r**

Frictional coefficient at the bottom of the ocean.

Definition at line 30 of file params.f90.

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

**8.13.3.43 real(kind=8) params::rp**

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

Definition at line 51 of file params.f90.

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

**8.13.3.44 real(kind=8) params::rr**

$R$  - Gas constant of dry air

Definition at line 45 of file params.f90.

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

**8.13.3.45 real(kind=8) params::rra**

Earth radius.

Definition at line 26 of file params.f90.

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

**8.13.3.46 real(kind=8) params::sb**

Stefan–Boltzmann constant.

Definition at line 66 of file params.f90.

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

**8.13.3.47 real(kind=8) params::sbpa**

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

Definition at line 61 of file params.f90.

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

**8.13.3.48 real(kind=8) params::sbpo**

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

Definition at line 60 of file params.f90.

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

**8.13.3.49 real(kind=8) params::sc**

Ratio of surface to atmosphere temperature.

Definition at line 65 of file params.f90.

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

**8.13.3.50 real(kind=8) params::scale**

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

Definition at line 47 of file params.f90.

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

**8.13.3.51 real(kind=8) params::sig0**

$\sigma_0$  - 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.
```

**8.13.3.52 real(kind=8) params::t\_run**

Effective intergration time (length of the generated trajectory)

Definition at line 76 of file params.f90.

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

**8.13.3.53 real(kind=8) params::t\_trans**

Transient time period.

Definition at line 75 of file params.f90.

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

**8.13.3.54 real(kind=8) params::ta0**

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

Definition at line 42 of file params.f90.

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

**8.13.3.55 real(kind=8) params::to0**

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

Definition at line 41 of file params.f90.

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

**8.13.3.56 real(kind=8) params::tw**

Write all variables every tw time units.

Definition at line 78 of file params.f90.

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

**8.13.3.57 logical params::writeout**

Write to file boolean.

Definition at line 79 of file params.f90.

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

**8.14 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.*

### 8.14.1 Detailed Description

Module with the MTV rk2 integration routines.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

This module actually contains the Heun algorithm routines.

### 8.14.2 Function/Subroutine Documentation

#### 8.14.2.1 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

```

#### 8.14.2.2 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

```

#### 8.14.2.3 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 ! ***"

```

#### 8.14.2.4 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

```

#### 8.14.2.5 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))

```

**8.14.2.6** 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

```

### 8.14.3 Variable Documentation

#### 8.14.3.1 `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

```

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

Definition at line 30 of file rk2\_MTV\_integrator.f90.

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

Integration buffers.

Definition at line 30 of file rk2\_MTV\_integrator.f90.

#### 8.14.3.4 `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

```

#### 8.14.3.5 `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

```

**8.14.3.6** `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
```

**8.14.3.7** `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
```

**8.14.3.8** `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
```

**8.14.3.9** `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwor` `[private]`

Definition at line 32 of file `rk2_MTV_integrator.f90`.

**8.14.3.10** `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwmult` `[private]`

Standard gaussian noise buffers.

Definition at line 31 of file `rk2_MTV_integrator.f90`.

**8.14.3.11** `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwor` `[private]`

Definition at line 32 of file `rk2_MTV_integrator.f90`.

**8.14.3.12** `real(kind=8), dimension(:), allocatable rk2_mtv_integrator::dwou` `[private]`

Standard gaussian noise buffers.

Definition at line 32 of file `rk2_MTV_integrator.f90`.

**8.14.3.13** `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
```

**8.14.3.14** `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
```

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

Additive noise term.

Definition at line 33 of file rk2\_MTV\_integrator.f90.

**8.14.3.16** `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
```

**8.14.3.17** `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
```

**8.14.3.18** `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
```

## 8.15 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.*

### 8.15.1 Detailed Description

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

This module actually contains the Heun algorithm routines.

## 8.15.2 Function/Subroutine Documentation

### 8.15.2.1 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
```

### 8.15.2.2 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

```

**8.15.2.3** `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

```

**8.15.2.4** `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)

```



8.15.2.5 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)

```

### 8.15.3 Variable Documentation

8.15.3.1 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

```

8.15.3.2 real(kind=8), dimension(:), allocatable rk2\_ss\_integrator::buf\_f0 [private]

Definition at line 32 of file rk2\_ss\_integrator.f90.

8.15.3.3 real(kind=8), dimension(:), allocatable rk2\_ss\_integrator::buf\_f1 [private]

Integration buffers.

Definition at line 32 of file rk2\_ss\_integrator.f90.

8.15.3.4 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

```

#### 8.15.3.5 `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
```

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

Standard gaussian noise buffers.

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

## 8.16 `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.*

### 8.16.1 Detailed Description

Module with the stochastic rk2 integration routines.

#### Copyright

2018 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

### 8.16.2 Function/Subroutine Documentation

#### 8.16.2.1 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_coo(aotensor,int_tensor)
79     CASE('ures')
80         CALL copy_coo(ff_tensor,int_tensor)

```

```

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

```

**8.16.2.2** 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

```

**8.16.2.3** 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)

```

**8.16.3 Variable Documentation****8.16.3.1** `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

```

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

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

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

Integration buffers.

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

**8.16.3.4** `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

```

**8.16.3.5** `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

```

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

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

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

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

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

Standard gaussian noise buffers.

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

8.16.3.9 `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
```

## 8.17 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.*

### 8.17.1 Detailed Description

Module with the WL rk2 integration routines.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

This module actually contains the Heun algorithm routines.

### 8.17.2 Function/Subroutine Documentation

#### 8.17.2.1 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

```

### 8.17.2.2 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

```

### 8.17.2.3 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)*dt
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

```



## 8.17.2.4 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

```

## 8.17.2.5 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
```

## 8.17.3 Variable Documentation

### 8.17.3.1 `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
```

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

Definition at line 31 of file rk2\_WL\_integrator.f90.

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

Integration buffers.

Definition at line 31 of file rk2\_WL\_integrator.f90.

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

Definition at line 32 of file rk2\_WL\_integrator.f90.

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

Definition at line 32 of file rk2\_WL\_integrator.f90.

8.17.3.6 `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
```

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

Definition at line 32 of file rk2\_WL\_integrator.f90.

8.17.3.8 `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.

8.17.3.9 `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
```

**8.17.3.10** `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
```

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

Definition at line 34 of file rk2\_WL\_integrator.f90.

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

Definition at line 34 of file rk2\_WL\_integrator.f90.

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

Standard gaussian noise buffers.

Definition at line 34 of file rk2\_WL\_integrator.f90.

**8.17.3.14** `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
```

**8.17.3.15** `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
```

## 8.18 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.*

### 8.18.1 Detailed Description

Module to select the resolved-unresolved components.

#### Copyright

2018 Jonathan Demaeyer See [LICENSE.txt](#) for license information.

### 8.18.2 Function/Subroutine Documentation

#### 8.18.2.1 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*nm))//") = 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

```

### 8.18.3 Variable Documentation

#### 8.18.3.1 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
```

#### 8.18.3.2 integer, dimension(:,:), allocatable, public sf\_def::bau

Definition at line 28 of file sf\_def.f90.

#### 8.18.3.3 integer, dimension(:,:), allocatable, public sf\_def::bor

Definition at line 28 of file sf\_def.f90.

#### 8.18.3.4 integer, dimension(:,:), allocatable, public sf\_def::bou

Filter matrices.

Definition at line 28 of file sf\_def.f90.

#### 8.18.3.5 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.
```

#### 8.18.3.6 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
```

**8.18.3.7 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
```

**8.18.3.8 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
```

**8.18.3.9 integer, dimension(:), allocatable, public sf\_def::rind**

Unresolved reduction indices.

Definition at line 24 of file sf\_def.f90.

**8.18.3.10 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
```

**8.18.3.11 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
```

**8.18.3.12 integer, dimension(:), allocatable, public sf\_def::sl\_rind**

Resolved reduction indices.

Definition at line 25 of file sf\_def.f90.



## 8.19 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  $\sigma_1$  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](#)  
 *$\sigma_2$  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](#)

### 8.19.1 Detailed Description

The MTV noise sigma matrices used to integrate the MTV model.

#### Copyright

2018 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.

### 8.19.2 Function/Subroutine Documentation

#### 8.19.2.1 subroutine, public sigma::compute\_mult\_sigma ( real(kind=8), dimension(0:ndim), intent(in) y )

Routine to actualize the matrix  $\sigma_1$  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

```

#### 8.19.2.2 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

```

### 8.19.3 Variable Documentation

#### 8.19.3.1 `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
```

#### 8.19.3.2 `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
```

#### 8.19.3.3 `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
```

#### 8.19.3.4 `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
```

#### 8.19.3.5 `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
```

#### 8.19.3.6 `integer, dimension(:), allocatable sigma::ind2` [private]

Definition at line 39 of file MTV\_sigma\_tensor.f90.

**8.19.3.7 integer sigma::n1 [private]**

Definition at line 41 of file MTV\_sigma\_tensor.f90.

```
41  INTEGER :: n1,n2
```

**8.19.3.8 integer sigma::n2 [private]**

Definition at line 41 of file MTV\_sigma\_tensor.f90.

**8.19.3.9 integer, dimension(:), allocatable sigma::rind1 [private]**

Definition at line 39 of file MTV\_sigma\_tensor.f90.

**8.19.3.10 integer, dimension(:), allocatable sigma::rind2 [private]**

Reduction indices.

Definition at line 39 of file MTV\_sigma\_tensor.f90.

**8.19.3.11 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
```

**8.19.3.12 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
```

**8.19.3.13 real(kind=8), dimension(:,,:), allocatable, public sigma::sig2**

$\sigma_2$  state-independent noise matrix

Definition at line 32 of file MTV\_sigma\_tensor.f90.

```
32  REAL(KIND=8), DIMENSION(:,,:), ALLOCATABLE, PUBLIC :: sig2 !< \f$\sigma_2\rf$ state-independent noise
    matrix
```

## 8.20 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

### 8.20.1 Detailed Description

Utility module with various routine to compute matrix square root.

#### Copyright

2018 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.

### 8.20.2 Function/Subroutine Documentation

**8.20.2.1** 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)

```

**8.20.2.2** 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

```

### 8.20.2.3 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 ! ***"

```

### 8.20.2.4 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=cmplx(t,kind=16)
352 zz=cmplx(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)=cmplx(0.d0,kind=16)
378 END DO

```

#### 8.20.2.5 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

```

#### 8.20.2.6 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 A.
<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

```

**8.20.2.7** 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,v,t
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,v,t,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,v,t))

```

**8.20.2.8 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

```

## 8.20.3 Variable Documentation

### 8.20.3.1 integer sqrt\_mod::lwork [private]

Definition at line 30 of file sqrt\_mod.f90.

```
30  INTEGER :: lwork
```

### 8.20.3.2 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
```

### 8.20.3.3 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
```

## 8.21 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](#)

### 8.21.1 Detailed Description

Statistics accumulators.

#### Copyright

2015 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

### 8.21.2 Function/Subroutine Documentation

#### 8.21.2.1 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)
```

### 8.21.2.2 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
```

### 8.21.2.3 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
```

### 8.21.2.4 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
```

### 8.21.2.5 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
```

### 8.21.2.6 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)
```

### 8.21.3 Variable Documentation

#### 8.21.3.1 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
```

#### 8.21.3.2 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
```

#### 8.21.3.3 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
```

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

Definition at line 26 of file stat.f90.

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

#### 8.21.3.5 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
```

## 8.22 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](#)

### 8.22.1 Detailed Description

Utility module containing the stochastic related routines.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.22.2 Function/Subroutine Documentation

#### 8.22.2.1 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

```

### 8.22.2.2 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

$dW$	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

```

### 8.22.2.3 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

$dW$	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

```

### 8.22.2.4 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

$dW$	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

```



**8.22.2.5** 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

```

**8.22.2.6** 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

```

**8.22.2.7** 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

```

#### 8.22.2.8 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

```

### 8.22.3 Variable Documentation

#### 8.22.3.1 real(kind=8) stoch\_mod::gset [private]

Definition at line 24 of file stoch\_mod.f90.

```

24      REAL(KIND=8) :: gset

```

#### 8.22.3.2 integer stoch\_mod::iset=0 [private]

Definition at line 23 of file stoch\_mod.f90.

```

23      INTEGER :: iset=0

```

## 8.23 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.*

### 8.23.1 Detailed Description

The stochastic models parameters module.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

## 8.23.2 Function/Subroutine Documentation

### 8.23.2.1 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

```

## 8.23.3 Variable Documentation

### 8.23.3.1 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

```

### 8.23.3.2 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

```

### 8.23.3.3 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

```

#### 8.23.3.4 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
```

#### 8.23.3.5 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
```

#### 8.23.3.6 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
```

#### 8.23.3.7 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
```

#### 8.23.3.8 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
```

#### 8.23.3.9 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
```

#### 8.23.3.10 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
```

#### 8.23.3.11 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
```

#### 8.23.3.12 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
```

#### 8.23.3.13 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
```

**8.23.3.14 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
```

**8.23.3.15 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
```

**8.23.3.16 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
```

**8.23.3.17 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
```

**8.23.3.18 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
```

### 8.23.3.19 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
```

### 8.23.3.20 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
```

## 8.24 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_coo` (src, dst)  
*Routine to copy a coolist.*
- subroutine, public `mat_to_coo` (src, dst)  
*Routine to convert a matrix to a tensor.*
- subroutine, public `sparse_mul3` (coolist\_ijk, arr\_j, arr\_k, res)  
*Sparse multiplication of a tensor with two vectors:  $\sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} a_j b_k$ .*
- subroutine, public `jsparse_mul` (coolist\_ijk, arr\_j, jcoo\_ij)



*Sparse multiplication of two tensors to determine the Jacobian:*

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

*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](#) (coolist\_ij, arr\_j, res)

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

- subroutine, public [simplify](#) (tensor)

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

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

- subroutine, public [add\\_elem](#) (t, i, j, k, v)

*Subroutine to add element to a coolist.*

- subroutine, public [add\\_check](#) (t, i, j, k, v, dst)

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

- subroutine, public [add\\_to\\_tensor](#) (src, dst)

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

- subroutine, public [print\\_tensor](#) (t, s)

*Routine to print a rank 3 tensor coolist.*

- subroutine, public [write\\_tensor\\_to\\_file](#) (s, t)

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

- subroutine, public [load\\_tensor\\_from\\_file](#) (s, t)

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

- 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 [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\\_tensor4](#) (t)  
*Routine to print a rank-4 tensor coolist.*
- subroutine, public [sparse\\_mul3\\_mat](#) (coolist\_ijk, arr\_k, res)  

$$\text{Sparse multiplication of a rank-3 tensor coolist with a vector: } \sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} b_k. \text{ Its output is a matrix.}$$
- subroutine, public [sparse\\_mul4](#) (coolist\_ijkl, arr\_j, arr\_k, arr\_l, res)  

$$\text{Sparse multiplication of a rank-4 tensor coolist with three vectors: } \sum_{j,k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} a_j b_k c_l.$$
- subroutine, public [sparse\\_mul4\\_mat](#) (coolist\_ijkl, arr\_k, arr\_l, res)  

$$\text{Sparse multiplication of a tensor with two vectors: } \sum_{k,l=0}^{ndim} \mathcal{T}_{i,j,k,l} b_k c_l.$$
- subroutine, public [sparse\\_mul2\\_j](#) (coolist\_ijk, arr\_j, res)  

$$\text{Sparse multiplication of a 3d sparse tensor with a vectors: } \sum_{j=0}^{ndim} \mathcal{T}_{i,j,k} a_j.$$
- subroutine, public [sparse\\_mul2\\_k](#) (coolist\_ijk, arr\_k, res)  

$$\text{Sparse multiplication of a rank-3 sparse tensor coolist with a vector: } \sum_{k=0}^{ndim} \mathcal{T}_{i,j,k} a_k.$$
- subroutine, public [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)  

$$\text{Sparse multiplication of a rank-4 tensor coolist with a matrix: } \sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{j,l}.$$
- subroutine, public [sparse\\_mul4\\_with\\_mat\\_kl](#) (coolist\_ijkl, mat\_kl, res)  

$$\text{Sparse multiplication of a rank-4 tensor coolist with a matrix: } \sum_{j,l=0}^{ndim} \mathcal{T}_{i,j,k,l} m_{k,l}.$$
- subroutine, public [sparse\\_mul3\\_with\\_mat](#) (coolist\_ijk, mat\_jk, res)  

$$\text{Sparse multiplication of a rank-3 tensor coolist with a matrix: } \sum_{j,k=0}^{ndim} \mathcal{T}_{i,j,k} m_{j,k}.$$
- 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.*

### 8.24.1 Detailed Description

Tensor utility module.

#### Copyright

2015-2018 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

`coolist` is a type and also means "coordinate list"

### 8.24.2 Function/Subroutine Documentation

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

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

#### Parameters

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

Definition at line 332 of file `tensor.f90`.

```

332  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
333  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
334  INTEGER, INTENT(IN) :: i, j, k
335  REAL(KIND=8), INTENT(IN) :: v
336  INTEGER :: n
337  CALL add_elem(t, i, j, k, v)
338  IF (t(i)%nelems==size(t(i)%elems)) THEN
339    CALL add_to_tensor(t, dst)
340    DO n=1, ndim
341      t(n)%nelems=0
342    ENDDO
343  ENDIF

```

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

Subroutine to add element to a coolist.

## Parameters

<i>t</i>	destination tensor
<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 310 of file tensor.f90.

```

310  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
311  INTEGER, INTENT(IN) :: i,j,k
312  REAL(KIND=8), INTENT(IN) :: v
313  INTEGER :: n
314  IF (abs(v) .ge. real_eps) THEN
315      n=(t(i)%elems)+1
316      t(i)%elems(n)%j=j
317      t(i)%elems(n)%k=k
318      t(i)%elems(n)%v=v
319      t(i)%elems=n
320  END IF

```

**8.24.2.3** 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>src</i>	Matrix to add
<i>dst</i>	Destination tensor

Definition at line 474 of file tensor.f90.

```

474  INTEGER, INTENT(IN) :: i
475  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
476  TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: dst
477  TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
478  INTEGER :: j,k,r,n,nsrc,allocstat
479
480  nsrc=0
481  DO j=1,ndim
482      DO k=1,ndim
483          IF (abs(src(j,k))>real_eps) nsrc=nsrc+1
484      END DO
485  END DO
486
487  IF (dst(i)%elems==0) THEN
488      IF (ALLOCATED(dst(i)%elems)) THEN
489          DEALLOCATE(dst(i)%elems, stat=allocstat)
490          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
491      ENDIF
492      ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
493      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
494      n=0
495  ELSE
496      n=dst(i)%elems
497      ALLOCATE(celems(n), stat=allocstat)
498      DO j=1,n
499          celems(j)%j=dst(i)%elems(j)%j
500          celems(j)%k=dst(i)%elems(j)%k
501          celems(j)%v=dst(i)%elems(j)%v
502      ENDDO

```

```

503     DEALLOCATE(dst(i)%elems, stat=allocstat)
504     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
505     ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
506     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
507     DO j=1,n
508         dst(i)%elems(j)%j=celems(j)%j
509         dst(i)%elems(j)%k=celems(j)%k
510         dst(i)%elems(j)%v=celems(j)%v
511     ENDDO
512     DEALLOCATE(celems, stat=allocstat)
513     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
514 ENDIF
515 r=0
516 DO j=1,ndim
517     DO k=1,ndim
518         IF (abs(src(j,k))>real_eps) THEN
519             r=r+1
520             dst(i)%elems(n+r)%j=j
521             dst(i)%elems(n+r)%k=k
522             dst(i)%elems(n+r)%v=src(j,k)
523         ENDIF
524     ENDDO
525 END DO
526 dst(i)%nelems=nsrc+n
527

```

#### 8.24.2.4 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(coolist4), 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 537 of file tensor.f90.

```

537     INTEGER, INTENT(IN) :: i,j
538     REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
539     TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
540     TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
541     INTEGER :: k,l,r,n,nsrc,allocstat
542
543     nsrc=0
544     DO k=1,ndim
545         DO l=1,ndim
546             IF (abs(src(k,l))>real_eps) nsrc=nsrc+1
547         END DO
548     END DO
549
550     IF (dst(i)%nelems==0) THEN
551         IF (ALLOCATED(dst(i)%elems)) THEN
552             DEALLOCATE(dst(i)%elems, stat=allocstat)
553             IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
554         ENDIF
555         ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
556         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
557         n=0
558     ELSE
559         n=dst(i)%nelems
560         ALLOCATE(celems(n), stat=allocstat)
561         DO k=1,n
562             celems(k)%j=dst(i)%elems(k)%j
563             celems(k)%k=dst(i)%elems(k)%k
564             celems(k)%l=dst(i)%elems(k)%l
565             celems(k)%v=dst(i)%elems(k)%v
566         ENDDO
567         DEALLOCATE(dst(i)%elems, stat=allocstat)
568         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
569

```

```

569     ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
570     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
571     DO k=1,n
572         dst(i)%elems(k)%j=celems(k)%j
573         dst(i)%elems(k)%k=celems(k)%k
574         dst(i)%elems(k)%l=celems(k)%l
575         dst(i)%elems(k)%v=celems(k)%v
576     ENDDO
577     DEALLOCATE(celems, stat=allocstat)
578     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
579     ENDO
580     r=0
581     DO k=1,ndim
582         DO l=1,ndim
583             IF (abs(src(k,l))>real_eps) THEN
584                 r=r+1
585                 dst(i)%elems(n+r)%j=j
586                 dst(i)%elems(n+r)%k=k
587                 dst(i)%elems(n+r)%l=l
588                 dst(i)%elems(n+r)%v=src(k,l)
589             ENDO
590         ENDDO
591     END DO
592     dst(i)%nelems=nsrc+n
593

```

#### 8.24.2.5 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 350 of file tensor.f90.

```

350     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
351     TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
352     TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
353     INTEGER :: i,j,n,allocstat
354
355     DO i=1,ndim
356         IF (src(i)%nelems/=0) THEN
357             IF (dst(i)%nelems==0) THEN
358                 IF (ALLOCATED(dst(i)%elems)) THEN
359                     DEALLOCATE(dst(i)%elems, stat=allocstat)
360                     IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
361                 ENDO
362                 ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
363                 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
364                 n=0
365             ELSE
366                 n=dst(i)%nelems
367                 ALLOCATE(celems(n), stat=allocstat)
368                 DO j=1,n
369                     celems(j)%j=dst(i)%elems(j)%j
370                     celems(j)%k=dst(i)%elems(j)%k
371                     celems(j)%v=dst(i)%elems(j)%v
372                 ENDDO
373                 DEALLOCATE(dst(i)%elems, stat=allocstat)
374                 IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
375                 ALLOCATE(dst(i)%elems(src(i)%nelems+n), stat=allocstat)
376                 IF (allocstat /= 0) stop "*** Not enough memory ! ***"
377                 DO j=1,n
378                     dst(i)%elems(j)%j=celems(j)%j
379                     dst(i)%elems(j)%k=celems(j)%k
380                     dst(i)%elems(j)%v=celems(j)%v
381                 ENDDO
382                 DEALLOCATE(celems, stat=allocstat)
383                 IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
384             ENDO
385             DO j=1,src(i)%nelems

```

```

386         dst(i)%elems(n+j)%j=src(i)%elems(j)%j
387         dst(i)%elems(n+j)%k=src(i)%elems(j)%k
388         dst(i)%elems(n+j)%v=src(i)%elems(j)%v
389     ENDDO
390     dst(i)%nelems=src(i)%nelems+n
391 ENDIF
392 ENDDO
393

```

**8.24.2.6** 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 785 of file tensor.f90.

```

785     INTEGER, INTENT(IN) :: i,j,k
786     REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
787     TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
788     TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
789     INTEGER :: l,ne,r,n,nsrc,allocstat
790
791     nsrc=0
792     DO l=1,ndim
793         IF (abs(src(l))>real_eps) nsrc=nsrc+1
794     ENDDO
795
796     IF (dst(i)%nelems==0) THEN
797         IF (ALLOCATED(dst(i)%elems)) THEN
798             DEALLOCATE(dst(i)%elems, stat=allocstat)
799             IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
800         ENDIF
801         ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
802         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
803         n=0
804     ELSE
805         n=dst(i)%nelems
806         ALLOCATE(celems(n), stat=allocstat)
807         DO ne=1,n
808             celems(ne)%j=dst(i)%elems(ne)%j
809             celems(ne)%k=dst(i)%elems(ne)%k
810             celems(ne)%l=dst(i)%elems(ne)%l
811             celems(ne)%v=dst(i)%elems(ne)%v
812         ENDDO
813         DEALLOCATE(dst(i)%elems, stat=allocstat)
814         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
815         ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
816         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
817         DO ne=1,n
818             dst(i)%elems(ne)%j=celems(ne)%j
819             dst(i)%elems(ne)%k=celems(ne)%k
820             dst(i)%elems(ne)%l=celems(ne)%l
821             dst(i)%elems(ne)%v=celems(ne)%v
822         ENDDO
823         DEALLOCATE(celems, stat=allocstat)
824         IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
825     ENDIF
826     r=0
827     DO l=1,ndim
828         IF (abs(src(l))>real_eps) THEN
829             r=r+1
830             dst(i)%elems(n+r)%j=j
831             dst(i)%elems(n+r)%k=k
832             dst(i)%elems(n+r)%l=1
833             dst(i)%elems(n+r)%v=src(l)
834         ENDIF
835     ENDDO
836     dst(i)%nelems=nsrc+n

```

**8.24.2.7** 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 726 of file tensor.f90.

```

726  INTEGER, INTENT(IN) :: i,k,l
727  REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
728  TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
729  TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
730  INTEGER :: j,ne,r,n,nsrc,allocstat
731
732  nsrc=0
733  DO j=1,ndim
734      IF (abs(src(j))>real_eps) nsrc=nsrc+1
735  ENDDO
736
737  IF (dst(i)%nelems==0) THEN
738      IF (ALLOCATED(dst(i)%elems)) THEN
739          DEALLOCATE(dst(i)%elems, stat=allocstat)
740          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
741      ENDIF
742      ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
743      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
744      n=0
745  ELSE
746      n=dst(i)%nelems
747      ALLOCATE(celems(n), stat=allocstat)
748      DO ne=1,n
749          celems(ne)%j=dst(i)%elems(ne)%j
750          celems(ne)%k=dst(i)%elems(ne)%k
751          celems(ne)%l=dst(i)%elems(ne)%l
752          celems(ne)%v=dst(i)%elems(ne)%v
753      ENDDO
754      DEALLOCATE(dst(i)%elems, stat=allocstat)
755      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
756      ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
757      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
758      DO ne=1,n
759          dst(i)%elems(ne)%j=celems(ne)%j
760          dst(i)%elems(ne)%k=celems(ne)%k
761          dst(i)%elems(ne)%l=celems(ne)%l
762          dst(i)%elems(ne)%v=celems(ne)%v
763      ENDDO
764      DEALLOCATE(celems, stat=allocstat)
765      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
766  ENDIF
767  r=0
768  DO j=1,ndim
769      IF (abs(src(j))>real_eps) THEN
770          r=r+1
771          dst(i)%elems(n+r)%j=j
772          dst(i)%elems(n+r)%k=k
773          dst(i)%elems(n+r)%l=l
774          dst(i)%elems(n+r)%v=src(j)
775      ENDIF
776  ENDDO
777  dst(i)%nelems=nsrc+n

```

**8.24.2.8** 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 657 of file tensor.f90.

```

657  INTEGER, INTENT(IN) :: i,k,l
658  REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
659  TYPE(coolist4), DIMENSION(ndim), INTENT(INOUT) :: dst
660  TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: celems
661  INTEGER :: j,ne,r,n,nsrc,allocstat
662
663  nsrc=0
664  DO j=1,ndim
665      IF (abs(src(j))>real_eps) nsrc=nsrc+1
666  ENDDO
667  nsrc=nsrc*3
668  IF (dst(i)%nelems==0) THEN
669      IF (ALLOCATED(dst(i)%elems)) THEN
670          DEALLOCATE(dst(i)%elems, stat=allocstat)
671          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
672      ENDIF
673      ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
674      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
675      n=0
676  ELSE
677      n=dst(i)%nelems
678      ALLOCATE(celems(n), stat=allocstat)
679      DO ne=1,n
680          celems(ne)%j=dst(i)%elems(ne)%j
681          celems(ne)%k=dst(i)%elems(ne)%k
682          celems(ne)%l=dst(i)%elems(ne)%l
683          celems(ne)%v=dst(i)%elems(ne)%v
684      ENDDO
685      DEALLOCATE(dst(i)%elems, stat=allocstat)
686      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
687      ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
688      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
689      DO ne=1,n
690          dst(i)%elems(ne)%j=celems(ne)%j
691          dst(i)%elems(ne)%k=celems(ne)%k
692          dst(i)%elems(ne)%l=celems(ne)%l
693          dst(i)%elems(ne)%v=celems(ne)%v
694      ENDDO
695      DEALLOCATE(celems, stat=allocstat)
696      IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
697  ENDIF
698  r=0
699  DO j=1,ndim
700      IF (abs(src(j))>real_eps) THEN
701          r=r+1
702          dst(i)%elems(n+r)%j=j
703          dst(i)%elems(n+r)%k=k
704          dst(i)%elems(n+r)%l=l
705          dst(i)%elems(n+r)%v=src(j)
706          r=r+1
707          dst(i)%elems(n+r)%j=k
708          dst(i)%elems(n+r)%k=l
709          dst(i)%elems(n+r)%l=j
710          dst(i)%elems(n+r)%v=src(j)
711          r=r+1
712          dst(i)%elems(n+r)%j=l
713          dst(i)%elems(n+r)%k=j
714          dst(i)%elems(n+r)%l=k
715          dst(i)%elems(n+r)%v=src(j)
716      ENDIF
717  ENDDO
718  dst(i)%nelems=nsrc+n

```

**8.24.2.9** 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 602 of file tensor.f90.

```

602  INTEGER, INTENT(IN) :: j,k
603  REAL(KIND=8), DIMENSION(ndim), INTENT(IN) :: src
604  TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
605  TYPE(coolist_elem), DIMENSION(:), ALLOCATABLE :: celems
606  INTEGER :: i,l,r,n,nsrc,allocstat
607
608  DO i=1,ndim
609      nsrc=0
610      IF (abs(src(i))>real_eps) nsrc=1
611      IF (dst(i)%nelems==0) THEN
612          IF (ALLOCATED(dst(i)%elems)) THEN
613              DEALLOCATE(dst(i)%elems, stat=allocstat)
614              IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
615          ENDIF
616          ALLOCATE(dst(i)%elems(nsrc), stat=allocstat)
617          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
618          n=0
619      ELSE
620          n=dst(i)%nelems
621          ALLOCATE(celems(n), stat=allocstat)
622          DO l=1,n
623              celems(l)%j=dst(i)%elems(l)%j
624              celems(l)%k=dst(i)%elems(l)%k
625              celems(l)%v=dst(i)%elems(l)%v
626          ENDDO
627          DEALLOCATE(dst(i)%elems, stat=allocstat)
628          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
629          ALLOCATE(dst(i)%elems(nsrc+n), stat=allocstat)
630          IF (allocstat /= 0) stop "*** Not enough memory ! ***"
631          DO l=1,n
632              dst(i)%elems(l)%j=celems(l)%j
633              dst(i)%elems(l)%k=celems(l)%k
634              dst(i)%elems(l)%v=celems(l)%v
635          ENDDO
636          DEALLOCATE(celems, stat=allocstat)
637          IF (allocstat /= 0) stop "*** Deallocation problem ! ***"
638      ENDIF
639      r=0
640      IF (abs(src(i))>real_eps) THEN
641          r=r+1
642          dst(i)%elems(n+r)%j=j
643          dst(i)%elems(n+r)%k=k
644          dst(i)%elems(n+r)%v=src(i)
645      ENDIF
646      dst(i)%nelems=nsrc+n
647  END DO
648
649

```

**8.24.2.10** 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 1112 of file tensor.f90.

```

1112  INTEGER, INTENT(IN) :: i
1113  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1114  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
1115  INTEGER :: n
1116
1117  dst=0.d0
1118  DO n=1,src(i)%elems
1119      dst(src(i)%elems(n)%j,src(i)%elems(n)%k)=src(i)%elems(n)%v
1120  ENDDO

```

#### 8.24.2.11 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 1079 of file tensor.f90.

```

1079  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1080  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
1081  INTEGER :: i,n
1082
1083  dst=0.d0
1084  DO i=1,ndim
1085      DO n=1,src(i)%elems
1086          dst(i,src(i)%elems(n)%j)=src(i)%elems(n)%v
1087      ENDDO
1088  ENDDO

```

#### 8.24.2.12 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 1063 of file tensor.f90.

```

1063  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1064  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
1065  INTEGER :: i,n
1066
1067  dst=0.d0
1068  DO i=1,ndim
1069      DO n=1,src(i)%elems
1070          dst(i,src(i)%elems(n)%k)=src(i)%elems(n)%v
1071      ENDDO
1072  ENDDO

```

**8.24.2.13** 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 1148 of file `tensor.f90`.

```

1148  INTEGER, INTENT(IN) :: j
1149  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1150  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: dst
1151  INTEGER :: i,n
1152
1153  dst=0.d0
1154  DO i=1,ndim
1155      DO n=1,src(i)%nelems
1156          IF (src(i)%elems(n)%j==j) dst(i,src(i)%elems(n)%k)=src(i)%elems(n)%v
1157      ENDDO
1158  END DO

```

**8.24.2.14** 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 1129 of file `tensor.f90`.

```

1129  INTEGER, INTENT(IN) :: j,k
1130  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
1131  REAL(KIND=8), DIMENSION(ndim), INTENT(OUT) :: dst
1132  INTEGER :: i,n
1133
1134  dst=0.d0
1135  DO i=1,ndim
1136      DO n=1,src(i)%nelems
1137          IF ((src(i)%elems(n)%j==j).and.(src(i)%elems(n)%k==k)) dst(i)=src(i)%elems(n)%v
1138      END DO
1139  ENDDO

```

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

Routine to copy a coolist.

## Parameters

<i>src</i>	Source coolist
<i>dst</i>	Destination coolist

## Remarks

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

Definition at line 72 of file tensor.f90.

```

72     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: src
73     TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
74     INTEGER :: i,j,allocstat
75
76     DO i=1,ndim
77         IF (dst(i)%nelems/=0) stop "*** copy_coo : Destination coolist not empty ! ***"
78         ALLOCATE(dst(i)%elems(src(i)%nelems), stat=allocstat)
79         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
80         DO j=1,src(i)%nelems
81             dst(i)%elems(j)%j=src(i)%elems(j)%j
82             dst(i)%elems(j)%k=src(i)%elems(j)%k
83             dst(i)%elems(j)%v=src(i)%elems(j)%v
84         ENDDO
85         dst(i)%nelems=src(i)%nelems
86     ENDDO

```

**8.24.2.16** 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 153 of file tensor.f90.

```

153     TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
154     TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: jcoo_ij
155     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
156     REAL(KIND=8) :: v
157     INTEGER :: i,j,k,n,nj,allocstat
158     DO i=1,ndim
159         IF (jcoo_ij(i)%nelems/=0) stop "*** jsparse_mul : Destination coolist not empty ! ***"
160         nj=2*coolist_ijk(i)%nelems

```

```

161      ALLOCATE(jcoo_ij(i)%elems(nj), stat=allocstat)
162      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
163      nj=0
164      DO n=1,coolist_ijk(i)%elems
165          j=coolist_ijk(i)%elems(n)%j
166          k=coolist_ijk(i)%elems(n)%k
167          v=coolist_ijk(i)%elems(n)%v
168          IF (j /=0) THEN
169              nj=nj+1
170              jcoo_ij(i)%elems(nj)%j=j
171              jcoo_ij(i)%elems(nj)%k=0
172              jcoo_ij(i)%elems(nj)%v=v*arr_j(k)
173          END IF
174
175          IF (k /=0) THEN
176              nj=nj+1
177              jcoo_ij(i)%elems(nj)%j=k
178              jcoo_ij(i)%elems(nj)%k=0
179              jcoo_ij(i)%elems(nj)%v=v*arr_j(j)
180          END IF
181      END DO
182      jcoo_ij(i)%elems=nj
183  END DO

```

**8.24.2.17** 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:

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

This version return a matrix.

#### Parameters

<code>coolist_ijk</code>	a coordinate list (sparse tensor) of which index 2 or 3 will be contracted.
<code>arr_j</code>	the vector to be contracted with index 2 and then index 3 of <code>ffi_coo_ijk</code>
<code>jcoo_ij</code>	a matrix to store the result of the contraction

Definition at line 196 of file `tensor.f90`.

```

196      TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
197      REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT):: jcoo_ij
198      REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
199      REAL(KIND=8) :: v
200      INTEGER :: i,j,k,n
201      jcoo_ij=0.d0
202      DO i=1,ndim
203          DO n=1,coolist_ijk(i)%elems
204              j=coolist_ijk(i)%elems(n)%j
205              k=coolist_ijk(i)%elems(n)%k
206              v=coolist_ijk(i)%elems(n)%v
207              IF (j /=0) jcoo_ij(i,j)=jcoo_ij(i,j)+v*arr_j(k)
208              IF (k /=0) jcoo_ij(i,k)=jcoo_ij(i,k)+v*arr_j(j)
209          END DO
210      END DO

```

8.24.2.18 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 1322 of file tensor.f90.

```

1322 CHARACTER (LEN=*), INTENT(IN) :: s
1323 TYPE(coolist4), DIMENSION(ndim), INTENT(OUT) :: t
1324 INTEGER :: i,ir,j,k,l,n,allocstat
1325 REAL(KIND=8) :: v
1326 OPEN(30,file=s,status='old')
1327 DO i=1,ndim
1328   READ(30,*) ir,n
1329   IF (n /= 0) THEN
1330     ALLOCATE(t(i)%elems(n), stat=allocstat)
1331     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
1332     t(i)%nelems=n
1333   ENDIF
1334   DO n=1,t(i)%nelems
1335     READ(30,*) ir,j,k,l,v
1336     t(i)%elems(n)%j=j
1337     t(i)%elems(n)%k=k
1338     t(i)%elems(n)%l=l
1339     t(i)%elems(n)%v=v
1340   ENDDO
1341 END DO
1342 CLOSE(30)

```

**8.24.2.19** subroutine, public tensor::load\_tensor\_from\_file ( character (len=\*), intent(in) s, type(coolist), dimension(ndim), intent(out) t )

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

## Parameters

<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 445 of file tensor.f90.

```

445 CHARACTER (LEN=*), INTENT(IN) :: s
446 TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: t
447 INTEGER :: i,ir,j,k,n,allocstat
448 REAL(KIND=8) :: v
449 OPEN(30,file=s,status='old')
450 DO i=1,ndim
451   READ(30,*) ir,n
452   IF (n /= 0) THEN
453     ALLOCATE(t(i)%elems(n), stat=allocstat)
454     IF (allocstat /= 0) stop "*** Not enough memory ! ***"
455     t(i)%nelems=n
456   ENDIF

```



```

457      DO n=1,t(i)%nelems
458      READ(30,*) ir,j,k,v
459      t(i)%elems(n)%j=j
460      t(i)%elems(n)%k=k
461      t(i)%elems(n)%v=v
462      ENDDO
463    END DO
464    CLOSE(30)

```

**8.24.2.20** subroutine, public tensor::mat\_to\_coo ( real(kind=8), dimension(0:ndim,0:ndim), intent(in) *src*, type(coolist), dimension(ndim), intent(out) *dst* )

Routine to convert a matrix to a tensor.

#### Parameters

<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.

Definition at line 94 of file tensor.f90.

```

94      REAL(KIND=8), DIMENSION(0:ndim,0:ndim), INTENT(IN) :: src
95      TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
96      INTEGER :: i,j,n,allocstat
97      DO i=1,ndim
98      n=0
99      DO j=1,ndim
100      IF (abs(src(i,j))>real_eps) n=n+1
101      ENDDO
102      IF (n/=0) THEN
103      IF (dst(i)%nelems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
104      ALLOCATE(dst(i)%elems(n), stat=allocstat)
105      IF (allocstat /= 0) stop "*** Not enough memory ! ***"
106      n=0
107      DO j=1,ndim
108      IF (abs(src(i,j))>real_eps) THEN
109      n=n+1
110      dst(i)%elems(n)%j=j
111      dst(i)%elems(n)%k=0
112      dst(i)%elems(n)%v=src(i,j)
113      ENDIF
114      ENDDO
115      ENDIF
116      dst(i)%nelems=n
117      ENDDO

```

**8.24.2.21** 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 1244 of file tensor.f90.

```

1244 REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: src
1245 TYPE(coolist), DIMENSION(ndim), INTENT(OUT) :: dst
1246 INTEGER :: i,j,n,allocstat
1247 DO i=1,ndim
1248     n=0
1249     DO j=1,ndim
1250         IF (abs(src(i,j))>real_eps) n=n+1
1251     ENDDO
1252     IF (n/=0) THEN
1253         IF (dst(i)%nelems/=0) stop "*** mat_to_coo : Destination coolist not empty ! ***"
1254         ALLOCATE(dst(i)%elems(n), stat=allocstat)
1255         IF (allocstat /= 0) stop "*** Not enough memory ! ***"
1256         n=0
1257         DO j=1,ndim
1258             IF (abs(src(i,j))>real_eps) THEN
1259                 n=n+1
1260                 dst(i)%elems(n)%j=0
1261                 dst(i)%elems(n)%k=j
1262                 dst(i)%elems(n)%v=src(i,j)
1263             ENDDO
1264         ENDDO
1265     ENDIF
1266     dst(i)%nelems=n
1267 ENDDO

```

**8.24.2.22** subroutine, public tensor::print\_tensor ( type(coolist), dimension(ndim), intent(in) t, character, intent(in), optional s )

Routine to print a rank 3 tensor coolist.

## Parameters

<i>t</i>	coolist to print
----------	------------------

Definition at line 399 of file tensor.f90.

```

399 USE util, only: str
400 TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: t
401 CHARACTER, INTENT(IN), OPTIONAL :: s
402 CHARACTER :: r
403 INTEGER :: i,n,j,k
404 IF (PRESENT(s)) THEN
405     r=s
406 ELSE
407     r="t"
408 END IF
409 DO i=1,ndim
410     DO n=1,t(i)%nelems
411         j=t(i)%elems(n)%j
412         k=t(i)%elems(n)%k
413         IF (abs(t(i)%elems(n)%v) .GE. real_eps) THEN
414             write(*,"(A,ES12.5)" r//"["//trim(str(i))//"]["//trim(str(j)) &
415                 &("//")["//trim(str(k))//"] = ",t(i)%elems(n)%v
416         END IF
417     END DO
418 END DO

```

**8.24.2.23** 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 922 of file tensor.f90.

```

922     USE util, only: str
923     TYPE(coolist4), DIMENSION(ndim), INTENT(IN) :: t
924     INTEGER :: i,n,j,k,l
925     DO i=1,ndim
926         DO n=1,t(i)%elems
927             j=t(i)%elems(n)%j
928             k=t(i)%elems(n)%k
929             l=t(i)%elems(n)%l
930             IF ( abs(t(i)%elems(n)%v) .GE. real_eps) THEN
931                 write(*,"(A,ES12.5)") "tensor["//trim(str(i))//"]["//trim(str(j)) &
932                     &("//")["//trim(str(k))//"]["//trim(str(l))//"] = ",t(i)%elems(n)%v
933             END IF
934         END DO
935     END DO

```

**8.24.2.24** 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 1274 of file tensor.f90.

```

1274     REAL(KIND=8), INTENT(IN) :: s
1275     TYPE(coolist), DIMENSION(ndim), INTENT(INOUT) :: t
1276     INTEGER :: i,li,n
1277     DO i=1,ndim
1278         n=t(i)%elems
1279         DO li=1,n
1280             t(i)%elems(li)%v=s*t(i)%elems(li)%v
1281         ENDDO
1282     ENDDO

```

**8.24.2.25** 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 238 of file tensor.f90.

```

238 TYPE(coolist), DIMENSION(ndim), INTENT(INOUT):: tensor
239 INTEGER :: i,j,k
240 INTEGER :: li,lii,liii,n
241 DO i= 1,ndim
242   n=tensor(i)%elems
243   DO li=n,2,-1
244     j=tensor(i)%elems(li)%j
245     k=tensor(i)%elems(li)%k
246     DO lii=li-1,1,-1
247       IF ((j==tensor(i)%elems(lii)%j).AND.(k==tensor(i)%
248         &elems(lii)%k)).OR.((j==tensor(i)%elems(lii)%k).AND.(k==
tensor(i)%elems(lii)%j)) THEN
249         ! Found another entry with the same i,j,k: merge both into
250         ! the one listed first (of those two).
251         tensor(i)%elems(lii)%v=tensor(i)%elems(lii)%v+tensor(i)%elems(li)%v
252         IF (j>k) THEN
253           tensor(i)%elems(lii)%j=tensor(i)%elems(li)%k
254           tensor(i)%elems(lii)%k=tensor(i)%elems(li)%j
255         ENDIF
256
257         ! Shift the rest of the items one place down.
258         DO liii=li+1,n
259           tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
260           tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
261           tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
262         END DO
263         tensor(i)%elems=tensor(i)%elems-1
264         ! Here we should stop because the li no longer points to the
265         ! original i,j,k element
266         EXIT
267       ENDIF
268     ENDDO
269   ENDDO
270   n=tensor(i)%elems
271   DO li=1,n
272     ! Clear new "almost" zero entries and shift rest of the items one place down.
273     ! Make sure not to skip any entries while shifting!
274     DO WHILE (abs(tensor(i)%elems(li)%v) < real_eps)
275       DO liii=li+1,n
276         tensor(i)%elems(liii-1)%j=tensor(i)%elems(liii)%j
277         tensor(i)%elems(liii-1)%k=tensor(i)%elems(liii)%k
278         tensor(i)%elems(liii-1)%v=tensor(i)%elems(liii)%v
279       ENDDO
280       tensor(i)%elems=tensor(i)%elems-1
281       if (li > tensor(i)%elems) THEN
282         EXIT
283       ENDIF
284     ENDDO
285   ENDDO
286   n=tensor(i)%elems
287   DO li=1,n
288     ! Upper triangularize
289     j=tensor(i)%elems(li)%j
290     k=tensor(i)%elems(li)%k
291     IF (j>k) THEN
292       tensor(i)%elems(li)%j=k
293       tensor(i)%elems(li)%k=j
294     ENDIF
295   ENDDO
296 ENDDO
297
298
299 ENDDO

```

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

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

#### Parameters

<code>coolist_ij</code>	a coordinate list (sparse tensor) of which index 2 will be contracted.
<code>arr_j</code>	the vector to be contracted with index 2 of coolist_ijk
<code>res</code>	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 221 of file `tensor.f90`.

```

221  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ij
222  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
223  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
224  INTEGER :: i, j, n
225  res=0.d0
226  DO i=1, ndim
227      DO n=1, coolist_ij(i)%elems
228          j=coolist_ij(i)%elems(n)%j
229          res(i) = res(i) + coolist_ij(i)%elems(n)%v * arr_j(j)
230      END DO
231  END DO

```

**8.24.2.27** 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 1024 of file `tensor.f90`.

```

1024  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
1025  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_j
1026  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
1027  INTEGER :: i, j, n
1028  res=0.d0
1029  DO i=1, ndim
1030      DO n=1, coolist_ijk(i)%elems
1031          j=coolist_ijk(i)%elems(n)%j
1032          res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_j(j)
1033      END DO
1034  END DO

```

**8.24.2.28** 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</i> ↔ <i>_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 <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\_k* as a result buffer, as this operation does multiple passes.

Definition at line 1045 of file tensor.f90.

```

1045     TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
1046     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_k
1047     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
1048     INTEGER :: i,k,n
1049     res=0.d0
1050     DO i=1,ndim
1051         DO n=1,coolist_ijk(i)%nelems
1052             k=coolist_ijk(i)%elems(n)%k
1053             res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_k(k)
1054         END DO
1055     END DO

```

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

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

## Parameters

<i>coolist</i> ↔ <i>_ijk</i>	a coordinate list (sparse tensor) of which index 2 and 3 will be contracted.
<i>arr_j</i>	the vector to be contracted with index 2 of <i>coolist_ijk</i>
<i>arr_k</i>	the vector to be contracted with index 3 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/arr\_k* as a result buffer, as this operation does multiple passes.

Definition at line 129 of file tensor.f90.

```

129     TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
130     REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j, arr_k
131     REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
132     INTEGER :: i,j,k,n
133     res=0.d0
134     DO i=1,ndim
135         DO n=1,coolist_ijk(i)%nelems
136             j=coolist_ijk(i)%elems(n)%j
137             k=coolist_ijk(i)%elems(n)%k
138             res(i) = res(i) + coolist_ijk(i)%elems(n)%v * arr_j(j)*arr_k(k)
139         END DO
140     END DO

```

**8.24.2.30** 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 948 of file `tensor.f90`.

```

948  TYPE(coolist), DIMENSION(ndim), INTENT(IN) :: coolist_ijk
949  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN) :: arr_k
950  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
951  INTEGER :: i,j,k,n
952  res=0.d0
953  DO i=1,ndim
954      DO n=1,coolist_ijk(i)%elems
955          j=coolist_ijk(i)%elems(n)%j
956          IF (j /= 0) THEN
957              k=coolist_ijk(i)%elems(n)%k
958              res(i,j) = res(i,j) + coolist_ijk(i)%elems(n)%v * arr_k(k)
959          ENDIF
960      END DO
961  END DO

```

**8.24.2.31** 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 1220 of file `tensor.f90`.

```

1220  TYPE(coolist), DIMENSION(ndim), INTENT(IN):: coolist_ijk
1221  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN)  :: mat_jk
1222  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
1223  INTEGER i,j,k,n
1224
1225  res=0.d0
1226  DO i=1,ndim
1227    DO n=1,coolist_ijk(i)%elems
1228      j=coolist_ijk(i)%elems(n)%j
1229      k=coolist_ijk(i)%elems(n)%k
1230
1231      res(i) = res(i) + coolist_ijk(i)%elems(n)%v * mat_jk(j,k)
1232    ENDDO
1233  END DO
1234

```

**8.24.2.32** 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 974 of file tensor.f90.

```

974  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
975  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_j, arr_k, arr_l
976  REAL(KIND=8), DIMENSION(0:ndim), INTENT(OUT) :: res
977  INTEGER :: i,j,k,n,l
978  res=0.d0
979  DO i=1,ndim
980    DO n=1,coolist_ijkl(i)%elems
981      j=coolist_ijkl(i)%elems(n)%j
982      k=coolist_ijkl(i)%elems(n)%k
983      l=coolist_ijkl(i)%elems(n)%l
984      res(i) = res(i) + coolist_ijkl(i)%elems(n)%v * arr_j(j)*arr_k(k)*arr_l(l)
985    END DO
986  END DO

```

**8.24.2.33** 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 coolist_ijkl
<i>arr_l</i>	the vector to be contracted with index 4 of coolist_ijkl
<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 998 of file tensor.f90.

```

998  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
999  REAL(KIND=8), DIMENSION(0:ndim), INTENT(IN)  :: arr_k, arr_l
1000  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
1001  INTEGER :: i,j,k,n,l
1002  res=0.d0
1003  DO i=1,ndim
1004      DO n=1,coolist_ijkl(i)%elems
1005          j=coolist_ijkl(i)%elems(n)%j
1006          IF (j /= 0) THEN
1007              k=coolist_ijkl(i)%elems(n)%k
1008              l=coolist_ijkl(i)%elems(n)%l
1009              res(i,j) = res(i,j) + coolist_ijkl(i)%elems(n)%v * arr_k(k) * arr_l(l)
1010          ENDIF
1011      END DO
1012  END DO

```

**8.24.2.34** 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 1169 of file tensor.f90.

```

1169  TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
1170  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN)  :: mat_jl
1171  REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
1172  INTEGER i,j,k,l,n
1173
1174  res=0.d0
1175  DO i=1,ndim
1176      DO n=1,coolist_ijkl(i)%elems
1177          j=coolist_ijkl(i)%elems(n)%j

```

```

1178         k=coolist_ijkl(i)%elems(n)%k
1179         l=coolist_ijkl(i)%elems(n)%l
1180
1181         res(i,k) = res(i,k) + coolist_ijkl(i)%elems(n)%v * mat_jl(j,l)
1182     ENDDO
1183 END DO
1184

```

**8.24.2.35** 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 1194 of file tensor.f90.

```

1194     TYPE(coolist4), DIMENSION(ndim), INTENT(IN):: coolist_ijkl
1195     REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(IN) :: mat_kl
1196     REAL(KIND=8), DIMENSION(ndim,ndim), INTENT(OUT) :: res
1197     INTEGER i,j,k,l,n
1198
1199     res=0.d0
1200     DO i=1,ndim
1201         DO n=1,coolist_ijkl(i)%elems
1202             j=coolist_ijkl(i)%elems(n)%j
1203             k=coolist_ijkl(i)%elems(n)%k
1204             l=coolist_ijkl(i)%elems(n)%l
1205
1206             res(i,j) = res(i,j) + coolist_ijkl(i)%elems(n)%v * mat_kl(k,l)
1207         ENDDO
1208     END DO
1209

```

**8.24.2.36** 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 1304 of file tensor.f90.

```

1304     TYPE(coolist4), DIMENSION(ndim), INTENT(IN) :: t

```

```

1305     LOGICAL :: tensor4_empty
1306     INTEGER :: i
1307     tensor4_empty=.true.
1308     DO i=1,ndim
1309         IF (t(i)%nelems /= 0) THEN
1310             tensor4_empty=.false.
1311             RETURN
1312         ENDIF
1313     END DO
1314     RETURN

```

**8.24.2.37** 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 883 of file tensor.f90.

```

883     REAL(KIND=8), DIMENSION(ndim,0:ndim,0:ndim,0:ndim), INTENT(IN) :: src
884     TYPE(coolist4), DIMENSION(ndim), INTENT(OUT) :: dst
885     INTEGER :: i, j, k, l, n, allocstat
886
887     DO i=1,ndim
888         n=0
889         DO j=0,ndim
890             DO k=0,ndim
891                 DO l=0,ndim
892                     IF (abs(src(i, j, k, l))>real_eps) n=n+1
893                 ENDDO
894             ENDDO
895         ENDDO
896         IF (n/=0) THEN
897             IF (dst(i)%nelems/=0) stop "*** tensor_to_coo : Destination coolist not empty ! ***"
898             ALLOCATE(dst(i)%elems(n), stat=allocstat)
899             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
900             n=0
901             DO j=0,ndim
902                 DO k=0,ndim
903                     DO l=0,ndim
904                         IF (abs(src(i, j, k, l))>real_eps) THEN
905                             n=n+1
906                             dst(i)%elems(n)%j=j
907                             dst(i)%elems(n)%k=k
908                             dst(i)%elems(n)%l=l
909                             dst(i)%elems(n)%v=src(i, j, k, l)
910                         ENDIF
911                     ENDDO
912                 ENDDO
913             ENDDO
914             dst(i)%nelems=n
915         ENDDO

```

**8.24.2.38** 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 1288 of file tensor.f90.

```

1288     TYPE(colist), DIMENSION(ndim), INTENT(IN) :: t
1289     LOGICAL :: tensor_empty
1290     INTEGER :: i
1291     tensor_empty=.true.
1292     DO i=1,ndim
1293         IF (t(i)%nelems /= 0) THEN
1294             tensor_empty=.false.
1295             RETURN
1296         ENDIF
1297     END DO
1298     RETURN

```

**8.24.2.39** subroutine, public tensor::tensor\_to\_coo ( real(kind=8), dimension(ndim,0:ndim,0:ndim), intent(in) *src*, type(colist), 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 847 of file tensor.f90.

```

847     REAL(KIND=8), DIMENSION(ndim,0:ndim,0:ndim), INTENT(IN) :: src
848     TYPE(colist), DIMENSION(ndim), INTENT(OUT) :: dst
849     INTEGER :: i,j,k,n,allocstat
850
851     DO i=1,ndim
852         n=0
853         DO j=0,ndim
854             DO k=0,ndim
855                 IF (abs(src(i,j,k))>real_eps) n=n+1
856             ENDDO
857         ENDDO
858         IF (n/=0) THEN
859             IF (dst(i)%nelems/=0) stop "*** tensor_to_coo : Destination coolist not empty ! ***"
860             ALLOCATE(dst(i)%elems(n), stat=allocstat)
861             IF (allocstat /= 0) stop "*** Not enough memory ! ***"
862             n=0
863             DO j=0,ndim
864                 DO k=0,ndim
865                     IF (abs(src(i,j,k))>real_eps) THEN
866                         n=n+1
867                         dst(i)%elems(n)%j=j
868                         dst(i)%elems(n)%k=k
869                         dst(i)%elems(n)%v=src(i,j,k)
870                     ENDIF
871                 ENDDO
872             ENDDO
873             IF (n>0) THEN
874                 dst(i)%nelems=n
875             ENDDO

```

#### 8.24.2.40 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

<i>s</i>	Destination filename
<i>t</i>	The coolist to write

Definition at line 1349 of file tensor.f90.

```

1349     CHARACTER (LEN=*) , INTENT(IN) :: s
1350     TYPE(coolist4) , DIMENSION(ndim) , INTENT(IN) :: t
1351     INTEGER :: i,j,k,l,n
1352     OPEN(30,file=s)
1353     DO i=1,ndim
1354         WRITE(30,*) i,t(i)%nelems
1355         DO n=1,t(i)%nelems
1356             j=t(i)%elems(n)%j
1357             k=t(i)%elems(n)%k
1358             l=t(i)%elems(n)%l
1359             WRITE(30,*) i,j,k,l,t(i)%elems(n)%v
1360         END DO
1361     END DO
1362     CLOSE(30)

```

#### 8.24.2.41 subroutine, public tensor::write\_tensor\_to\_file ( character (len=\*) , intent(in) s, type(coolist), dimension(ndim), intent(in) t )

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

##### Parameters

<i>s</i>	Destination filename
<i>t</i>	The coolist to write

Definition at line 425 of file tensor.f90.

```

425     CHARACTER (LEN=*) , INTENT(IN) :: s
426     TYPE(coolist) , DIMENSION(ndim) , INTENT(IN) :: t
427     INTEGER :: i,j,k,n
428     OPEN(30,file=s)
429     DO i=1,ndim
430         WRITE(30,*) i,t(i)%nelems
431         DO n=1,t(i)%nelems
432             j=t(i)%elems(n)%j
433             k=t(i)%elems(n)%k
434             WRITE(30,*) i,j,k,t(i)%elems(n)%v
435         END DO
436     END DO
437     CLOSE(30)

```

### 8.24.3 Variable Documentation

#### 8.24.3.1 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

```

## 8.25 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.*

### 8.25.1 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.

### 8.25.2 Function/Subroutine Documentation

- 8.25.2.1 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

```

### 8.25.2.2 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 ! ***"

```

### 8.25.2.3 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 t1(t,ystar,y,buf_f0)
92      buf_y1 = y+dt*buf_f0
93      CALL t1(t+dt,ystar,buf_y1,buf_f1)
94      res=y+0.5*(buf_f0+buf_f1)*dt
95      t=t+dt

```

### 8.25.3 Variable Documentation

#### 8.25.3.1 `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

```

#### 8.25.3.2 `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

```

#### 8.25.3.3 `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

```

#### 8.25.3.4 `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

```



## 8.25.3.5 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
```

## 8.26 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\\_tltensor](#)  
*Routine to initialize the TL tensor.*
- subroutine [compute\\_tltensor](#) (func)  
*Routine to compute the TL tensor from the original MAOOAM one.*
- subroutine [tl\\_add\\_count](#) (i, j, k, v)  
*Subroutine used to count the number of TL tensor entries.*
- subroutine [tl\\_coeff](#) (i, j, k, v)  
*Subroutine used to compute the TL tensor entries.*
- subroutine, public [init\\_adtensor](#)  
*Routine to initialize the AD tensor.*
- subroutine [compute\\_adtensor](#) (func)  
*Subroutine to compute the AD tensor from the original MAOOAM one.*
- subroutine [ad\\_add\\_count](#) (i, j, k, v)  
*Subroutine used to count the number of AD tensor entries.*
- subroutine [ad\\_coeff](#) (i, j, k, v)
- subroutine, public [init\\_adtensor\\_ref](#)  
*Alternate method to initialize the AD tensor from the TL tensor.*
- subroutine [compute\\_adtensor\\_ref](#) (func)  
*Alternate subroutine to compute the AD tensor from the TL one.*
- subroutine [ad\\_add\\_count\\_ref](#) (i, j, k, v)  
*Alternate subroutine used to count the number of AD tensor entries from the TL tensor.*
- subroutine [ad\\_coeff\\_ref](#) (i, j, k, v)  
*Alternate subroutine used to compute the AD tensor entries from the TL tensor.*
- subroutine, public [ad](#) (t, ystar, deltay, buf)  
*Tendencies for the AD of MAOOAM in point ystar for perturbation deltay.*
- subroutine, public [tl](#) (t, ystar, deltay, buf)  
*Tendencies for the TL of MAOOAM in point ystar for perturbation deltay.*

## Variables

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

### 8.26.1 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 !

### 8.26.2 Function/Subroutine Documentation

**8.26.2.1** `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 t
<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)

```

**8.26.2.2** `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>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 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

```

**8.26.2.3** 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>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 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

```

**8.26.2.4** 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>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 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

```

**8.26.2.5** 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

```

**8.26.2.6** 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`.

**8.26.2.7** 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.

**8.26.2.8** 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.

**8.26.2.9** 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
```

**8.26.2.10** 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

```

#### 8.26.2.11 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

```

#### 8.26.2.12 `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(coolist), DIMENSION(ndim) :: jacobian
77     CALL jsparse_mul(aotensor, ystar, jacobian)

```

**8.26.2.13** `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)

```

**8.26.2.14** `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)

```

**8.26.2.15** 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

```

**8.26.2.16** 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(tltensor)) stop "*** tl_coeff routine : tensor not yet allocated ***"
165  IF (.NOT. ALLOCATED(tltensor(i)%elems)) stop "*** tl_coeff routine : tensor not yet allocated ***"
166  IF (abs(v) .ge. real_eps) THEN
167      IF (j /=0) THEN
168          n=(tltensor(i)%elems)+1
169          tltensor(i)%elems(n)%j=j
170          tltensor(i)%elems(n)%k=k
171          tltensor(i)%elems(n)%v=v
172          tltensor(i)%elems=n
173      END IF
174      IF (k /=0) THEN
175          n=(tltensor(i)%elems)+1
176          tltensor(i)%elems(n)%j=k
177          tltensor(i)%elems(n)%k=j
178          tltensor(i)%elems(n)%v=v
179          tltensor(i)%elems=n
180      END IF
181  END IF

```



### 8.26.3 Variable Documentation

#### 8.26.3.1 `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
```

#### 8.26.3.2 `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
```

#### 8.26.3.3 `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
```

#### 8.26.3.4 `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
```

## 8.27 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.*
- `integer` function, `dimension(size(s))`, public `isin` (c, s)  
*Determine if a character is in a string and where.*
- subroutine, public `init_random_seed` ()  
*Random generator initialization routine.*
- subroutine, public `piksort` (k, arr, par)  
*Simple card player sorting function.*
- 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)

### 8.27.1 Detailed Description

Utility module.

#### Copyright

2018 Lesley De Cruz & Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

### 8.27.2 Function/Subroutine Documentation

#### 8.27.2.1 subroutine, public `util::cdiag` ( `complex(kind=16)`, `dimension(:, :)`, `intent(in)` A, `complex(kind=16)`, `dimension(:, :)`, `intent(out)` d )

Definition at line 269 of file `util.f90`.

```

269  COMPLEX(KIND=16), DIMENSION(:, :), INTENT(IN) :: a
270  COMPLEX(KIND=16), DIMENSION(:, :), INTENT(OUT) :: d
271  INTEGER :: i
272
273  DO i=1, SIZE(a, 1)
274      d(i)=a(i, i)
275  END DO

```

**8.27.2.2** subroutine, public util::choldc ( real(kind=8), dimension(:, :) a, real(kind=8), dimension(:) p )

Definition at line 176 of file util.f90.

```

176    REAL(KIND=8), DIMENSION(:, :) :: a
177    REAL(KIND=8), DIMENSION(:) :: p
178    INTEGER :: n
179    INTEGER :: i, j, k
180    REAL(KIND=8) :: sum
181    n=SIZE(a,1)
182    DO i=1,n
183        DO j=i,n
184            sum=a(i,j)
185            DO k=i-1,1,-1
186                sum=sum-a(i,k)*a(j,k)
187            END DO
188            IF (i.eq.j) THEN
189                IF (sum.le.0.) stop 'choldc failed'
190                p(i)=sqrt(sum)
191            ELSE
192                a(j,i)=sum/p(i)
193            ENDIF
194        END DO
195    END DO
196    RETURN

```

**8.27.2.3** subroutine, public util::cprintmat ( complex(kind=16), dimension(:, :), intent(in) A )

Definition at line 209 of file util.f90.

```

209    COMPLEX(KIND=16), DIMENSION(:, :), INTENT(IN) :: a
210    INTEGER :: i
211
212    DO i=1,SIZE(a,1)
213        print*, a(i,:)
214    END DO

```

**8.27.2.4** subroutine, public util::diag ( real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:), intent(out) d )

Definition at line 259 of file util.f90.

```

259    REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
260    REAL(KIND=8), DIMENSION(:), INTENT(OUT) :: d
261    INTEGER :: i
262
263    DO i=1,SIZE(a,1)
264        d(i)=a(i,i)
265    END DO

```

**8.27.2.5** integer function, public util::floordiv ( integer i, integer j )

Definition at line 280 of file util.f90.

```

280    INTEGER :: i, j, floordiv
281    floordiv=int(floor(real(i)/real(j)))
282    RETURN

```

### 8.27.2.6 subroutine, public util::init\_one ( real(kind=8), dimension(:,,:), intent(inout) A )

Initialize a square matrix A as a unit matrix.

Definition at line 139 of file util.f90.

```

139      REAL(KIND=8), DIMENSION(:,,:), INTENT(INOUT) :: a
140      INTEGER :: i, n
141      n=size(a,1)
142      a=0.0d0
143      DO i=1,n
144          a(i,i)=1.0d0
145      END DO
146

```

### 8.27.2.7 subroutine, public util::init\_random\_seed ( )

Random generator initialization routine.

Definition at line 64 of file util.f90.

### 8.27.2.8 real(kind=8) function, dimension(size(a,1),size(a,2)), public util::invmat ( real(kind=8), dimension(:,,:), intent(in) A )

Definition at line 218 of file util.f90.

```

218      REAL(KIND=8), DIMENSION(:,,:), INTENT(IN) :: a
219      REAL(KIND=8), DIMENSION(SIZE(A,1),SIZE(A,2)) :: ainv
220
221      REAL(KIND=8), DIMENSION(SIZE(A,1)) :: work ! work array for LAPACK
222      INTEGER, DIMENSION(SIZE(A,1)) :: ipiv ! pivot indices
223      INTEGER :: n, info
224
225      ! Store A in Ainv to prevent it from being overwritten by LAPACK
226      ainv = a
227      n = size(a,1)
228
229      ! DGETRF computes an LU factorization of a general M-by-N matrix A
230      ! using partial pivoting with row interchanges.
231      CALL dgetrf(n, n, ainv, n, ipiv, info)
232
233      IF (info /= 0) THEN
234          stop 'Matrix is numerically singular!'
235      ENDIF
236
237      ! DGETRI computes the inverse of a matrix using the LU factorization
238      ! computed by DGETRF.
239      CALL dgetri(n, ainv, n, ipiv, work, n, info)
240
241      IF (info /= 0) THEN
242          stop 'Matrix inversion failed!'
243      ENDIF

```

### 8.27.2.9 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 314 of file util.f90.

```

314      REAL(KIND=8), DIMENSION(:,,:), INTENT(OUT) :: a
315      REAL(KIND=8), DIMENSION(:,,:), INTENT(IN) :: ared
316      INTEGER, INTENT(IN) :: n
317      INTEGER, DIMENSION(:), INTENT(IN) :: ind, rind
318      INTEGER :: i, j
319      a=0.d0
320      DO i=1,n
321          DO j=1,n
322              a(ind(i), ind(j))=ared(i, j)
323          END DO
324      END DO

```

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

Determine if a character is in a string and where.

**Remarks**

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

Definition at line 47 of file util.f90.

```

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

```

**8.27.2.11 real(kind=8) function, public util::mat\_contract ( real(kind=8), dimension(:,:) A, real(kind=8), dimension(:,:) B )**

Definition at line 162 of file util.f90.

```

162     REAL(KIND=8), DIMENSION(:,:) :: a,b
163     REAL(KIND=8) :: mat_contract
164     INTEGER :: i, j, n
165     n=size(a,1)
166     mat_contract=0.d0
167     DO i=1,n
168         DO j=1,n
169             mat_contract=mat_contract+a(i,j)*b(i,j)
170         END DO
171     ENDDO
172     RETURN

```

**8.27.2.12 real(kind=8) function, public util::mat\_trace ( real(kind=8), dimension(:,:) A )**

Definition at line 150 of file util.f90.

```

150     REAL(KIND=8), DIMENSION(:,:) :: a
151     REAL(KIND=8) :: mat_trace
152     INTEGER :: i, n
153     n=size(a,1)
154     mat_trace=0.d0
155     DO i=1,n
156         mat_trace=mat_trace+a(i,i)
157     END DO
158     RETURN

```

### 8.27.2.13 subroutine, public util::piksort ( integer, intent(in) *k*, integer, dimension(*k*), intent(inout) *arr*, integer, intent(out) *par* )

Simple card player sorting function.

Definition at line 118 of file util.f90.

```

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

```

### 8.27.2.14 subroutine, public util::printmat ( real(kind=8), dimension(:, :), intent(in) *A* )

Definition at line 200 of file util.f90.

```

200  REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
201  INTEGER :: i
202
203  DO i=1,SIZE(a,1)
204    print*, a(i,:)
205  END DO

```

### 8.27.2.15 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 286 of file util.f90.

```

286  REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
287  REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: ared
288  INTEGER, INTENT(OUT) :: n
289  INTEGER, DIMENSION(:, :), INTENT(OUT) :: ind, rind
290  LOGICAL, DIMENSION(SIZE(A,1)) :: sel
291  INTEGER :: i, j
292
293  ind=0
294  rind=0
295  sel=.false.
296  n=0
297  DO i=1,SIZE(a,1)
298    IF (any(a(i,:) /= 0)) THEN
299      n=n+1
300      sel(i)=.true.
301      ind(n)=i
302      rind(i)=n
303    ENDIF
304  END DO
305  ared=0.d0
306  DO i=1,SIZE(a,1)
307    DO j=1,SIZE(a,1)
308      IF (sel(i).and.sel(j)) ared(rind(i),rind(j))=a(i,j)
309    ENDDO
310  ENDDO

```

**8.27.2.16** `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 38 of file util.f90.

```

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

```

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

Convert an integer to string.

Definition at line 31 of file util.f90.

```

31      INTEGER, INTENT(IN) :: k
32      WRITE (str, *) k
33      str = adjustl(str)

```

**8.27.2.18** `subroutine, public util::triu ( real(kind=8), dimension(:, :), intent(in) A, real(kind=8), dimension(:, :), intent(out) T )`

Definition at line 247 of file util.f90.

```

247      REAL(KIND=8), DIMENSION(:, :), INTENT(IN) :: a
248      REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: t
249      INTEGER i, j
250      t=0.d0
251      DO i=1, SIZE(a,1)
252          DO j=i, SIZE(a,1)
253              t(i, j)=a(i, j)
254          END DO
255      END DO

```

**8.27.2.19** `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 328 of file util.f90.

```

328      REAL(KIND=8), DIMENSION(:), INTENT(IN) :: u, v
329      REAL(KIND=8), DIMENSION(:, :), INTENT(OUT) :: a
330      INTEGER :: i, j
331
332      a=0.d0
333      DO i=1, SIZE(u)
334          DO j=1, SIZE(v)
335              a(i, j)=u(i) * v(j)
336          ENDDO
337      ENDDO

```

**8.28** `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.*

### 8.28.1 Detailed Description

The WL tensors used to integrate the model.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

#### Remarks

### 8.28.2 Function/Subroutine Documentation

#### 8.28.2.1 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_coo(lxy,m21)
145      CALL add_to_tensor(bxyy,m21)
146
147      m21def=.not.tensor_empty(m21)
148
149      !M22
150      CALL copy_coo(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

```

## 8.28.3 Variable Documentation

### 8.28.3.1 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

```

### 8.28.3.2 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

```

### 8.28.3.3 logical, public wl\_tensor::b14def

Definition at line 75 of file WL\_tensor.f90.

**8.28.3.4 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
```

**8.28.3.5 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
```

**8.28.3.6 logical, public wl\_tensor::b23def**

Definition at line 75 of file WL\_tensor.f90.

**8.28.3.7 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
```

**8.28.3.8 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
```

**8.28.3.9 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
```

**8.28.3.10** `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
```

**8.28.3.11** `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
```

**8.28.3.12** `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
```

**8.28.3.13** `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
```

**8.28.3.14** `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
```

**8.28.3.15 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
```

**8.28.3.16 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
```

**8.28.3.17 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
```

**8.28.3.18 logical, public wl\_tensor::ldef**

Definition at line 75 of file WL\_tensor.f90.

**8.28.3.19 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
```

**8.28.3.20 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
```

**8.28.3.21** `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
```

**8.28.3.22** `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
```

**8.28.3.23** `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
```

**8.28.3.24** `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
```

**8.28.3.25** `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
```

**8.28.3.26** `logical, public wl_tensor::m21def`

Definition at line 75 of file WL\_tensor.f90.



**8.28.3.27** 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
```

**8.28.3.28** logical, public wl\_tensor::m22def

Definition at line 75 of file WL\_tensor.f90.

**8.28.3.29** logical, public wl\_tensor::mdef

Boolean to (de)activate the computation of the terms.

Definition at line 75 of file WL\_tensor.f90.

**8.28.3.30** 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 9

# Data Type Documentation

### 9.1 inprod\_analytic::atm\_tensors Type Reference

Type holding the atmospheric inner products tensors.

#### Private Attributes

- procedure([calculate\\_a](#)), pointer, nopass [a](#)
- procedure([calculate\\_b](#)), pointer, nopass [b](#)
- procedure([calculate\\_c\\_atm](#)), pointer, nopass [c](#)
- procedure([calculate\\_d](#)), pointer, nopass [d](#)
- procedure([calculate\\_g](#)), pointer, nopass [g](#)
- procedure([calculate\\_s](#)), pointer, nopass [s](#)

#### 9.1.1 Detailed Description

Type holding the atmospheric inner products tensors.

Definition at line 53 of file inprod\_analytic.f90.

#### 9.1.2 Member Data Documentation

##### 9.1.2.1 procedure([calculate\\_a](#)), pointer, nopass inprod\_analytic::atm\_tensors::a [private]

Definition at line 54 of file inprod\_analytic.f90.

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

##### 9.1.2.2 procedure([calculate\\_b](#)), pointer, nopass inprod\_analytic::atm\_tensors::b [private]

Definition at line 55 of file inprod\_analytic.f90.

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

### 9.1.2.3 `procedure(calculate_c_atm), pointer, nopass inprod_analytic::atm_tensors::c` [private]

Definition at line 56 of file `inprod_analytic.f90`.

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

### 9.1.2.4 `procedure(calculate_d), pointer, nopass inprod_analytic::atm_tensors::d` [private]

Definition at line 57 of file `inprod_analytic.f90`.

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

### 9.1.2.5 `procedure(calculate_g), pointer, nopass inprod_analytic::atm_tensors::g` [private]

Definition at line 58 of file `inprod_analytic.f90`.

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

### 9.1.2.6 `procedure(calculate_s), pointer, nopass inprod_analytic::atm_tensors::s` [private]

Definition at line 59 of file `inprod_analytic.f90`.

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

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

- [inprod\\_analytic.f90](#)

## 9.2 `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.

### 9.2.1 Detailed Description

Atmospheric bloc specification type.

Definition at line 40 of file inprod\_analytic.f90.

### 9.2.2 Member Data Documentation

#### 9.2.2.1 integer inprod\_analytic::atm\_wavenum::h =0 [private]

Definition at line 42 of file inprod\_analytic.f90.

#### 9.2.2.2 integer inprod\_analytic::atm\_wavenum::m =0 [private]

Definition at line 42 of file inprod\_analytic.f90.

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

#### 9.2.2.3 real(kind=8) inprod\_analytic::atm\_wavenum::nx =0. [private]

Definition at line 43 of file inprod\_analytic.f90.

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

#### 9.2.2.4 real(kind=8) inprod\_analytic::atm\_wavenum::ny =0. [private]

Definition at line 43 of file inprod\_analytic.f90.

#### 9.2.2.5 integer inprod\_analytic::atm\_wavenum::p =0 [private]

Definition at line 42 of file inprod\_analytic.f90.

#### 9.2.2.6 character inprod\_analytic::atm\_wavenum::typ [private]

Definition at line 41 of file inprod\_analytic.f90.

```
41      CHARACTER :: typ
```

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

- [inprod\\_analytic.f90](#)

## 9.3 `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.*

### 9.3.1 Detailed Description

Coordinate list. Type used to represent the sparse tensor.

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

### 9.3.2 Member Data Documentation

#### 9.3.2.1 `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
```

#### 9.3.2.2 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](#)

## 9.4 `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

### 9.4.1 Detailed Description

4d coordinate list. Type used to represent the rank-4 sparse tensor.

Definition at line 44 of file tensor.f90.

### 9.4.2 Member Data Documentation

#### 9.4.2.1 `type(coolist_elem4)`, `dimension(:)`, allocatable `tensor::coolist4::elems`

Definition at line 45 of file tensor.f90.

```
45      TYPE(coolist_elem4), DIMENSION(:), ALLOCATABLE :: elems
```

#### 9.4.2.2 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](#)

## 9.5 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.*

### 9.5.1 Detailed Description

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

Definition at line 25 of file tensor.f90.

### 9.5.2 Member Data Documentation

#### 9.5.2.1 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
```

#### 9.5.2.2 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
```

#### 9.5.2.3 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](#)

## 9.6 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](#)

### 9.6.1 Detailed Description

4d coordinate list element type. Elementary elements of the 4d sparse tensors.

Definition at line 32 of file tensor.f90.

### 9.6.2 Member Data Documentation

#### 9.6.2.1 integer tensor::coolist\_elem4::j [private]

Definition at line 33 of file tensor.f90.

```
33      INTEGER :: j,k,l
```

#### 9.6.2.2 integer tensor::coolist\_elem4::k [private]

Definition at line 33 of file tensor.f90.

#### 9.6.2.3 integer tensor::coolist\_elem4::l [private]

Definition at line 33 of file tensor.f90.

#### 9.6.2.4 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](#)

## 9.7 inprod\_analytic::ocean\_tensors Type Reference

Type holding the oceanic inner products tensors.

## Private Attributes

- procedure([calculate\\_k](#)), pointer, nopass [k](#)
- procedure([calculate\\_m](#)), pointer, nopass [m](#)
- procedure([calculate\\_c\\_oc](#)), pointer, nopass [c](#)
- procedure([calculate\\_n](#)), pointer, nopass [n](#)
- procedure([calculate\\_o](#)), pointer, nopass [o](#)
- procedure([calculate\\_w](#)), pointer, nopass [w](#)

### 9.7.1 Detailed Description

Type holding the oceanic inner products tensors.

Definition at line 63 of file `inprod_analytic.f90`.

### 9.7.2 Member Data Documentation

#### 9.7.2.1 `procedure(calculate_c_oc), pointer, nopass inprod_analytic::ocean_tensors::c` `[private]`

Definition at line 66 of file `inprod_analytic.f90`.

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

#### 9.7.2.2 `procedure(calculate_k), pointer, nopass inprod_analytic::ocean_tensors::k` `[private]`

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

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

#### 9.7.2.3 `procedure(calculate_m), pointer, nopass inprod_analytic::ocean_tensors::m` `[private]`

Definition at line 65 of file `inprod_analytic.f90`.

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

#### 9.7.2.4 `procedure(calculate_n), pointer, nopass inprod_analytic::ocean_tensors::n` `[private]`

Definition at line 67 of file `inprod_analytic.f90`.

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

**9.7.2.5** `procedure(calculate_o), pointer, nopass inprod_analytic::ocean_tensors::o` `[private]`

Definition at line 68 of file inprod\_analytic.f90.

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

**9.7.2.6** `procedure(calculate_w), pointer, nopass inprod_analytic::ocean_tensors::w` `[private]`

Definition at line 69 of file inprod\_analytic.f90.

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

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

- [inprod\\_analytic.f90](#)

**9.8 inprod\_analytic::ocean\_wavenum Type Reference**

Oceanic bloc specification type.

**Private Attributes**

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

**9.8.1 Detailed Description**

Oceanic bloc specification type.

Definition at line 47 of file inprod\_analytic.f90.

**9.8.2 Member Data Documentation****9.8.2.1** `integer inprod_analytic::ocean_wavenum::h` `[private]`

Definition at line 48 of file inprod\_analytic.f90.

**9.8.2.2** `real(kind=8) inprod_analytic::ocean_wavenum::nx` `[private]`

Definition at line 49 of file inprod\_analytic.f90.

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

**9.8.2.3** `real(kind=8) inprod_analytic::ocean_wavenum::ny` `[private]`

Definition at line 49 of file `inprod_analytic.f90`.

**9.8.2.4** `integer inprod_analytic::ocean_wavenum::p` `[private]`

Definition at line 48 of file `inprod_analytic.f90`.

```
48      INTEGER :: p,h
```

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

- [inprod\\_analytic.f90](#)

# Chapter 10

## File Documentation

### 10.1 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.

## 10.2 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  $s$ .*

## 10.3 corrmmod.f90 File Reference

### Modules

- module [corrmmod](#)  
*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.*

## 10.4 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.*



## 10.5 doc/def\_doc.md File Reference

## 10.6 doc/gen\_doc.md File Reference

## 10.7 doc/sto\_doc.md File Reference

## 10.8 doc/tl\_ad\_doc.md File Reference

## 10.9 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.*

## 10.10 inprod\_analytic.f90 File Reference

### Data Types

- type [inprod\\_analytic::atm\\_wavenum](#)  
*Atmospheric bloc specification type.*
- type [inprod\\_analytic::ocean\\_wavenum](#)  
*Oceanic bloc specification type.*
- type [inprod\\_analytic::atm\\_tensors](#)  
*Type holding the atmospheric inner products tensors.*
- type [inprod\\_analytic::ocean\\_tensors](#)  
*Type holding the oceanic inner products tensors.*

## Modules

- module `inprod_analytic`

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

## Functions/Subroutines

- `real(kind=8)` function `inprod_analytic::b1` (Pi, Pj, Pk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::b2` (Pi, Pj, Pk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::delta` (r)  
*Integer Dirac delta function.*
- `real(kind=8)` function `inprod_analytic::flambda` (r)  
*"Odd or even" function*
- `real(kind=8)` function `inprod_analytic::s1` (Pj, Pk, Mj, Hk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::s2` (Pj, Pk, Mj, Hk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::s3` (Pj, Pk, Hj, Hk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::s4` (Pj, Pk, Hj, Hk)  
*Cehelsky & Tung Helper functions.*
- `real(kind=8)` function `inprod_analytic::calculate_a` (i, j)  
*Eigenvalues of the Laplacian (atmospheric)*
- `real(kind=8)` function `inprod_analytic::calculate_b` (i, j, k)  
*Streamfunction advection terms (atmospheric)*
- `real(kind=8)` function `inprod_analytic::calculate_c_atm` (i, j)  
*Beta term for the atmosphere.*
- `real(kind=8)` function `inprod_analytic::calculate_d` (i, j)  
*Forcing of the ocean on the atmosphere.*
- `real(kind=8)` function `inprod_analytic::calculate_g` (i, j, k)  
*Temperature advection terms (atmospheric)*
- `real(kind=8)` function `inprod_analytic::calculate_s` (i, j)  
*Forcing (thermal) of the ocean on the atmosphere.*
- `real(kind=8)` function `inprod_analytic::calculate_k` (i, j)  
*Forcing of the atmosphere on the ocean.*
- `real(kind=8)` function `inprod_analytic::calculate_m` (i, j)  
*Forcing of the ocean fields on the ocean.*
- `real(kind=8)` function `inprod_analytic::calculate_n` (i, j)  
*Beta term for the ocean.*
- `real(kind=8)` function `inprod_analytic::calculate_o` (i, j, k)  
*Temperature advection term (passive scalar)*
- `real(kind=8)` function `inprod_analytic::calculate_c_oc` (i, j, k)  
*Streamfunction advection terms (oceanic)*
- `real(kind=8)` function `inprod_analytic::calculate_w` (i, j)  
*Short-wave radiative forcing of the ocean.*
- subroutine, public `inprod_analytic::init_inprod`  
*Initialisation of the inner product.*

## Variables

- type(atm\_wavenum), dimension(:), allocatable, public [inprod\\_analytic::awavenum](#)  
*Atmospheric blocs specification.*
- type(ocean\_wavenum), dimension(:), allocatable, public [inprod\\_analytic::owavenum](#)  
*Oceanic blocs specification.*
- type(atm\_tensors), public [inprod\\_analytic::atmos](#)  
*Atmospheric tensors.*
- type(ocean\_tensors), public [inprod\\_analytic::ocean](#)  
*Oceanic tensors.*

## 10.11 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.*

## 10.12 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.*

## 10.13 LICENSE.txt File Reference

### Functions

- The MIT [License](#) (MIT) Copyright(c) 2015-2018 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](#)

### 10.13.1 Function Documentation

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

10.13.1.2 The MIT License ( MIT )

## 10.13.2 Variable Documentation

10.13.2.1 The MIT free of charge

Definition at line 6 of file LICENSE.txt.

10.13.2.2 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.

10.13.2.3 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.

10.13.2.4 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.

10.13.2.5 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.

10.13.2.6 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.

- 10.13.2.7 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.

- 10.13.2.8 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.

- 10.13.2.9 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.

- 10.13.2.10 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.

- 10.13.2.11 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.

- 10.13.2.12 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.

- 10.13.2.13 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.

10.13.2.14 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.

10.13.2.15 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.

10.13.2.16 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.

10.13.2.17 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.

10.13.2.18 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.

10.13.2.19 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.

10.13.2.20 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.

## 10.14 maoam.f90 File Reference

### Functions/Subroutines

- program [maoam](#)

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



### 10.14.1 Function/Subroutine Documentation

#### 10.14.1.1 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.

## 10.15 maoam\_MTV.f90 File Reference

### Functions/Subroutines

- program [maoam\\_mtv](#)

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

### 10.15.1 Function/Subroutine Documentation

#### 10.15.1.1 program maoam\_mtv ( )

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file maoam\_MTV.f90.

## 10.16 maoam\_stoch.f90 File Reference

### Functions/Subroutines

- program [maoam\\_stoch](#)

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

## 10.16.1 Function/Subroutine Documentation

### 10.16.1.1 program maoam\_stoch ( )

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

#### Copyright

2018 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.

## 10.17 maoam\_WL.f90 File Reference

### Functions/Subroutines

- program [maoam\\_wl](#)

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

### 10.17.1 Function/Subroutine Documentation

#### 10.17.1.1 program maoam\_wl ( )

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

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file maoam\_WL.f90.

## 10.18 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::rrd](#)  
*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*

## 10.19 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.*

## 10.20 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.*

## 10.21 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  $\sigma_1$  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](#)  
 $\sigma_2$  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](#)

## 10.22 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](#)  
 $\sigma_0$  - 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](#)  
 *$\lambda$  - 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](#)  
 *$\gamma_o$  - 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](#)  
 *$\epsilon_a$  - Emissivity coefficient for the grey-body atmosphere.*
- real(kind=8) [params::ga](#)  
 *$\gamma_a$  - 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](#)  
 *$\pi$*
- real(kind=8) [params::lr](#)  
 *$L_R$  - Rossby deformation radius*
- real(kind=8) [params::g](#)  
 *$\gamma$*
- 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](#)
- $\lambda'_o$  - 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](#)
- $\lambda'_a$  - 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](#)
- $\beta'$  - Non-dimensional beta parameter

  - real(kind=8) [params::nua](#) =0.D0
- Dissipation in the atmosphere.

  - real(kind=8) [params::nuo](#) =0.D0
- Dissipation in the ocean.

  - real(kind=8) [params::nuap](#)
- Non-dimensional dissipation in the atmosphere.

  - real(kind=8) [params::nuop](#)
- Non-dimensional dissipation in the ocean.

  - real(kind=8) [params::t\\_trans](#)
- Transient time period.

  - real(kind=8) [params::t\\_run](#)
- Effective intergration time (length of the generated trajectory)

  - real(kind=8) [params::dt](#)
- Integration time step.

  - real(kind=8) [params::tw](#)
- Write all variables every tw time units.

  - logical [params::writeout](#)
- Write to file boolean.

  - integer [params::nboc](#)
- Number of atmospheric blocks.

  - integer [params::nbatm](#)
- Number of oceanic blocks.

  - integer [params::natm](#) =0
- Number of atmospheric basis functions.

  - integer [params::noc](#) =0
- Number of oceanic basis functions.

  - integer [params::ndim](#)
- Number of variables (dimension of the model)



- integer, dimension(:,:), allocatable [params::oms](#)  
*Ocean mode selection array.*
- integer, dimension(:,:), allocatable [params::ams](#)  
*Atmospheric mode selection array.*

## 10.23 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.*

## 10.24 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.*

## 10.25 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.*

## 10.26 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.*

## 10.27 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.*

## 10.28 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.*

## 10.29 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.*

## 10.30 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.*

## 10.31 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.*

## 10.32 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

## 10.33 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](#)

## 10.34 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](#)

## 10.35 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.*

## 10.36 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\\_coo](#) (src, dst)

*Routine to copy a coolist.*

- subroutine, public [tensor::mat\\_to\\_coo](#) (src, dst)

*Routine to convert a matrix to a tensor.*

- subroutine, public [tensor::sparse\\_mul3](#) (coolist\_ijk, arr\_j, arr\_k, res)

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

- subroutine, public [tensor::jsparse\\_mul](#) (coolist\_ijk, arr\_j, jcoo\_ij)

*Sparse multiplication of two tensors to determine the Jacobian:*

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

*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:*

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

*This version return a matrix.*

- subroutine, public [tensor::sparse\\_mul2](#) (coolist\_ij, arr\_j, res)

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

- subroutine, public [tensor::simplify](#) (tensor)

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

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

- subroutine, public [tensor::add\\_elem](#) (t, i, j, k, v)

*Subroutine to add element to a coolist.*

- subroutine, public [tensor::add\\_check](#) (t, i, j, k, v, dst)

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

- subroutine, public [tensor::add\\_to\\_tensor](#) (src, dst)

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

- subroutine, public [tensor::print\\_tensor](#) (t, s)

*Routine to print a rank 3 tensor coolist.*

- subroutine, public [tensor::write\\_tensor\\_to\\_file](#) (s, t)

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

- subroutine, public [tensor::load\\_tensor\\_from\\_file](#) (s, t)

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

- 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::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_tensor4` (t)  
*Routine to print a rank-4 tensor coolist.*
- 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::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::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.*

## 10.37 test\_aotensor.f90 File Reference

### Functions/Subroutines

- program [test\\_aotensor](#)
- Small program to print the inner products.*

### 10.37.1 Function/Subroutine Documentation

#### 10.37.1.1 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.

## 10.38 test\_corr.f90 File Reference

### Functions/Subroutines

- program [test\\_corr](#)
- Small program to print the correlation and covariance matrices.*

### 10.38.1 Function/Subroutine Documentation

#### 10.38.1.1 program test\_corr ( )

Small program to print the correlation and covariance matrices.

##### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_corr.f90.

## 10.39 test\_corr\_tensor.f90 File Reference

### Functions/Subroutines

- program [test\\_corr\\_tensor](#)  
*Small program to print the time correlations tensors.*

### 10.39.1 Function/Subroutine Documentation

#### 10.39.1.1 program test\_corr\_tensor ( )

Small program to print the time correlations tensors.

##### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_corr\_tensor.f90.

## 10.40 test\_dec\_tensor.f90 File Reference

### Functions/Subroutines

- program [test\\_dec\\_tensor](#)  
*Small program to print the decomposed tensors.*

### 10.40.1 Function/Subroutine Documentation

#### 10.40.1.1 program test\_dec\_tensor ( )

Small program to print the decomposed tensors.

##### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_dec\_tensor.f90.

## 10.41 test\_inprod\_analytic.f90 File Reference

### Functions/Subroutines

- program [inprod\\_analytic\\_test](#)  
*Small program to print the inner products.*

#### 10.41.1 Function/Subroutine Documentation

##### 10.41.1.1 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.

## 10.42 test\_MAR.f90 File Reference

### Functions/Subroutines

- program [test\\_mar](#)  
*Small program to test the Multivariate AutoRegressive model.*

#### 10.42.1 Function/Subroutine Documentation

##### 10.42.1.1 program test\_mar ( )

Small program to test the Multivariate AutoRegressive model.

### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_MAR.f90.

## 10.43 test\_memory.f90 File Reference

### Functions/Subroutines

- program [test\\_memory](#)  
*Small program to test the WL memory module.*

### 10.43.1 Function/Subroutine Documentation

#### 10.43.1.1 program test\_memory ( )

Small program to test the WL memory module.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_memory.f90.

## 10.44 test\_MTV\_int\_tensor.f90 File Reference

### Functions/Subroutines

- program [test\\_mtv\\_int\\_tensor](#)  
*Small program to print the MTV integrated tensors.*

### 10.44.1 Function/Subroutine Documentation

#### 10.44.1.1 program test\_mtv\_int\_tensor ( )

Small program to print the MTV integrated tensors.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_MTV\_int\_tensor.f90.

## 10.45 test\_MTV\_sigma\_tensor.f90 File Reference

### Functions/Subroutines

- program [test\\_sigma](#)  
*Small program to test the MTV noise sigma matrices.*



### 10.45.1 Function/Subroutine Documentation

#### 10.45.1.1 program test\_sigma ( )

Small program to test the MTV noise sigma matrices.

##### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_MTV\_sigma\_tensor.f90.

## 10.46 test\_sqrtm.f90 File Reference

### Functions/Subroutines

- program [test\\_sqrtm](#)  
*Small program to test the matrix square-root module.*

### 10.46.1 Function/Subroutine Documentation

#### 10.46.1.1 program test\_sqrtm ( )

Small program to test the matrix square-root module.

##### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 12 of file test\_sqrtm.f90.

## 10.47 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.*

### 10.47.1 Function/Subroutine Documentation

#### 10.47.1.1 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.

## 10.48 test\_WL\_tensor.f90 File Reference

### Functions/Subroutines

- program [test\\_wl\\_tensor](#)  
*Small program to print the WL tensors.*

### 10.48.1 Function/Subroutine Documentation

#### 10.48.1.1 program test\_wl\_tensor ( )

Small program to print the WL tensors.

#### Copyright

2018 Jonathan Demaeyer. See [LICENSE.txt](#) for license information.

Definition at line 11 of file test\_WL\_tensor.f90.

## 10.49 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.*

## 10.50 util.f90 File Reference

### Modules

- module `util`  
*Utility module.*

### Functions/Subroutines

- character(len=20) function, public `util::str` (k)  
*Convert an integer to string.*
- character(len=40) function, public `util::rstr` (x, fm)  
*Convert a real to string with a given format.*
- integer function, dimension(size(s)), public `util::isin` (c, s)  
*Determine if a character is in a string and where.*
- subroutine, public `util::init_random_seed` ()  
*Random generator initialization routine.*
- integer function `lcg` (s)
- subroutine, public `util::piksort` (k, arr, par)  
*Simple card player sorting function.*
- 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)

### 10.50.1 Function/Subroutine Documentation

#### 10.50.1.1 integer function `init_random_seed::lcg` ( integer(int64) s )

Definition at line 104 of file `util.f90`.

```

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

```

## 10.51 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](#)1  
*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, [26](#)
  - inprod\_analytic::atm\_tensors, [221](#)
- acc
  - stat, [150](#)
- ad
  - tl\_ad\_tensor, [196](#)
- ad\_add\_count
  - tl\_ad\_tensor, [196](#)
- ad\_add\_count\_ref
  - tl\_ad\_tensor, [197](#)
- ad\_coeff
  - tl\_ad\_tensor, [197](#)
- ad\_coeff\_ref
  - tl\_ad\_tensor, [198](#)
- ad\_step
  - tl\_ad\_integrator, [192](#)
- add\_check
  - tensor, [165](#)
- add\_count
  - aotensor\_def, [26](#)
- add\_elem
  - tensor, [165](#)
- add\_matc\_to\_tensor
  - tensor, [166](#)
- add\_matc\_to\_tensor4
  - tensor, [167](#)
- add\_to\_tensor
  - tensor, [168](#)
- add\_vec\_ijk\_to\_tensor4
  - tensor, [169](#)
- add\_vec\_ikl\_to\_tensor4
  - tensor, [169](#)
- add\_vec\_ikl\_to\_tensor4\_perm
  - tensor, [170](#)
- add\_vec\_jk\_to\_tensor
  - tensor, [171](#)
- adtensor
  - tl\_ad\_tensor, [203](#)
- ams
  - params, [101](#)
- anoise
  - rk2\_mtv\_integrator, [117](#)
  - rk2\_ss\_integrator, [123](#)
  - rk2\_stoch\_integrator, [127](#)
  - rk2\_wl\_integrator, [132](#)
- aotensor
  - aotensor\_def, [29](#)
- aotensor\_def, [25](#)
- a, [26](#)
- add\_count, [26](#)
- aotensor, [29](#)
- coeff, [26](#)
- compute\_aotensor, [27](#)
- count\_elems, [29](#)
- init\_aotensor, [27](#)
- kdelta, [28](#)
- psi, [28](#)
- real\_eps, [29](#)
- t, [28](#)
- theta, [28](#)
- aotensor\_def.f90, [231](#)
- atmos
  - inprod\_analytic, [67](#)
- awavenum
  - inprod\_analytic, [67](#)
- b
  - inprod\_analytic::atm\_tensors, [221](#)
- b1
  - inprod\_analytic, [59](#)
  - mtv\_int\_tensor, [92](#)
  - wl\_tensor, [214](#)
- b14
  - wl\_tensor, [214](#)
- b14def
  - wl\_tensor, [214](#)
- b2
  - inprod\_analytic, [59](#)
  - mtv\_int\_tensor, [93](#)
  - wl\_tensor, [214](#)
- b23
  - wl\_tensor, [215](#)
- b23def
  - wl\_tensor, [215](#)
- b3
  - wl\_tensor, [215](#)
- b4
  - wl\_tensor, [215](#)
- bar
  - sf\_def, [137](#)
- bau
  - sf\_def, [137](#)
- betp
  - params, [101](#)
- bor
  - sf\_def, [137](#)
- bou
  - sf\_def, [137](#)

- btot
  - mtv\_int\_tensor, 93
- buf\_f0
  - integrator, 79
  - rk2\_mtv\_integrator, 117
  - rk2\_ss\_integrator, 123
  - rk2\_stoch\_integrator, 127
  - rk2\_wl\_integrator, 132
  - tl\_ad\_integrator, 194
- buf\_f1
  - integrator, 79
  - rk2\_mtv\_integrator, 117
  - rk2\_ss\_integrator, 123
  - rk2\_stoch\_integrator, 127
  - rk2\_wl\_integrator, 133
  - tl\_ad\_integrator, 194
- buf\_g
  - rk2\_mtv\_integrator, 117
- buf\_ka
  - integrator, 79
  - tl\_ad\_integrator, 194
- buf\_kb
  - integrator, 79
  - tl\_ad\_integrator, 194
- buf\_m
  - memory, 87
  - rk2\_wl\_integrator, 133
- buf\_m1
  - rk2\_wl\_integrator, 133
- buf\_m2
  - rk2\_wl\_integrator, 133
- buf\_m3
  - memory, 87
  - rk2\_wl\_integrator, 133
- buf\_m3s
  - rk2\_wl\_integrator, 133
- buf\_y
  - mar, 82
- buf\_y1
  - integrator, 79
  - rk2\_mtv\_integrator, 117
  - rk2\_ss\_integrator, 123
  - rk2\_stoch\_integrator, 127
  - rk2\_wl\_integrator, 133
  - tl\_ad\_integrator, 194
- bxxx
  - dec\_tensor, 51
- bxyx
  - dec\_tensor, 51
- bxyy
  - dec\_tensor, 51
- byxx
  - dec\_tensor, 52
- byxy
  - dec\_tensor, 52
- byyy
  - dec\_tensor, 52
- c
  - inprod\_analytic::atm\_tensors, 221
  - inprod\_analytic::ocean\_tensors, 228
  - CLAIM
    - LICENSE.txt, 240
  - CONTRACT
    - LICENSE.txt, 240
  - ca
    - params, 101
  - calculate\_a
    - inprod\_analytic, 59
  - calculate\_b
    - inprod\_analytic, 60
  - calculate\_c\_atm
    - inprod\_analytic, 60
  - calculate\_c\_oc
    - inprod\_analytic, 60
  - calculate\_d
    - inprod\_analytic, 61
  - calculate\_g
    - inprod\_analytic, 61
  - calculate\_k
    - inprod\_analytic, 62
  - calculate\_m
    - inprod\_analytic, 62
  - calculate\_n
    - inprod\_analytic, 63
  - calculate\_o
    - inprod\_analytic, 63
  - calculate\_s
    - inprod\_analytic, 63
  - calculate\_w
    - inprod\_analytic, 64
  - cdiag
    - util, 204
  - charge
    - LICENSE.txt, 240
  - chol
    - sqrt\_mod, 143
  - choldc
    - util, 204
  - co
    - params, 101
  - coeff
    - aotensor\_def, 26
  - comp\_corrnt
    - int\_corr, 73
  - compg
    - rk2\_mtv\_integrator, 114
  - compute\_adtensor
    - tl\_ad\_tensor, 198
  - compute\_adtensor\_ref
    - tl\_ad\_tensor, 198
  - compute\_aotensor
    - aotensor\_def, 27
  - compute\_m1
    - rk2\_wl\_integrator, 129
  - compute\_m2
    - rk2\_wl\_integrator, 130



- compute\_m3
  - memory, 85
- compute\_mult
  - rk2\_mtv\_integrator, 117
- compute\_mult\_sigma
  - sigma, 139
- compute\_tltensor
  - tl\_ad\_tensor, 199
- conditions
  - LICENSE.txt, 240
- coo\_to\_mat\_i
  - tensor, 172
- coo\_to\_mat\_ij
  - tensor, 173
- coo\_to\_mat\_ik
  - tensor, 173
- coo\_to\_mat\_j
  - tensor, 173
- coo\_to\_vec\_jk
  - tensor, 174
- copy
  - LICENSE.txt, 240
- copy\_coo
  - tensor, 174
- corr2int
  - int\_corr, 75
- corr\_i
  - corrmod, 41
- corr\_i\_full
  - corrmod, 41
- corr\_ij
  - corrmod, 41
- corr\_tensor, 29
  - dumb\_mat1, 31
  - dumb\_mat2, 31
  - dumb\_vec, 31
  - dy, 32
  - dyy, 32
  - expm, 32
  - init\_corr\_tensor, 30
  - ydy, 32
  - ydy, 32
  - yy, 33
- corr\_tensor.f90, 232
- corrcomp
  - corrmod, 41
- corrcomp\_from\_def
  - corrmod, 34
- corrcomp\_from\_fit
  - corrmod, 38
- corrcomp\_from\_spline
  - corrmod, 38
- corrint
  - int\_corr, 75
- corrmod, 33
  - corr\_i, 41
  - corr\_i\_full, 41
  - corr\_ij, 41
- corrmod.f90, 232
- count\_elems
  - aotensor\_def, 29
  - tl\_ad\_tensor, 203
- cpa
  - params, 102
- cpo
  - params, 102
- cprintmat
  - util, 205
- csqrtm\_triu
  - sqrt\_mod, 144
- d
  - inprod\_analytic::atm\_tensors, 222
  - params, 102
- dec\_tensor, 44
  - bxxx, 51
  - bxy, 51
  - bxy, 51
  - byxx, 52
  - byxy, 52
  - byyy, 52
  - dumb, 52
  - ff\_tensor, 52
  - fs\_tensor, 53
  - hx, 53
  - hy, 53
  - init\_dec\_tensor, 45
  - init\_sub\_tensor, 49
  - lxx, 53
  - lxy, 53
  - lyx, 54
  - lyy, 54
  - reorder, 49
  - sf\_tensor, 54
  - ss\_tensor, 54
  - ss\_tl\_tensor, 54
  - suppress\_and, 50
  - suppress\_or, 50
- dec\_tensor.f90, 233
- delta

- inprod\_analytic, 64
- diag
  - util, 205
- distribute
  - LICENSE.txt, 240
- doc/def\_doc.md, 235
- doc/gen\_doc.md, 235
- doc/sto\_doc.md, 235
- doc/tl\_ad\_doc.md, 235
- dp
  - params, 102
- dt
  - params, 102
- dtn
  - stoch\_params, 158
- dts
  - stoch\_params, 158
- dtsn
  - stoch\_params, 158
- dumb
  - dec\_tensor, 52
- dumb\_mat1
  - corr\_tensor, 31
  - mtv\_int\_tensor, 93
  - sigma, 141
  - wl\_tensor, 215
- dumb\_mat2
  - corr\_tensor, 31
  - mtv\_int\_tensor, 93
  - sigma, 141
  - wl\_tensor, 215
- dumb\_mat3
  - mtv\_int\_tensor, 93
  - sigma, 141
  - wl\_tensor, 216
- dumb\_mat4
  - mtv\_int\_tensor, 94
  - sigma, 141
  - wl\_tensor, 216
- dumb\_vec
  - corr\_tensor, 31
  - mtv\_int\_tensor, 94
  - wl\_tensor, 216
- dw
  - mar, 82
  - rk2\_mtv\_integrator, 118
- dwar
  - rk2\_mtv\_integrator, 118
  - rk2\_ss\_integrator, 123
  - rk2\_stoch\_integrator, 127
  - rk2\_wl\_integrator, 133
- dwau
  - rk2\_mtv\_integrator, 118
  - rk2\_stoch\_integrator, 127
  - rk2\_wl\_integrator, 134
- dwmult
  - rk2\_mtv\_integrator, 118
- dwor
  - rk2\_mtv\_integrator, 118
  - rk2\_ss\_integrator, 124
  - rk2\_stoch\_integrator, 128
  - rk2\_wl\_integrator, 134
- dwou
  - rk2\_mtv\_integrator, 118
  - rk2\_stoch\_integrator, 128
  - rk2\_wl\_integrator, 134
- dy
  - corr\_tensor, 32
- dyy
  - corr\_tensor, 32
- elems
  - tensor::coolist, 224
  - tensor::coolist4, 225
- eps\_pert
  - stoch\_params, 158
- epsa
  - params, 103
- exists
  - ic\_def, 57
  - sf\_def, 137
- expm
  - corr\_tensor, 32
- f0
  - params, 103
- FROM
  - LICENSE.txt, 240
- ff\_tensor
  - dec\_tensor, 52
- files
  - LICENSE.txt, 239
- flambda
  - inprod\_analytic, 64
- floordiv
  - util, 205
- fs
  - corrmod, 39
- fs\_tensor
  - dec\_tensor, 53
- full\_step
  - rk2\_mtv\_integrator, 114
  - rk2\_wl\_integrator, 130
- func\_ij
  - int\_corr, 74
- func\_ijkl
  - int\_corr, 75
- g
  - inprod\_analytic::atm\_tensors, 222
  - params, 103
  - rk2\_mtv\_integrator, 118
- ga
  - params, 103
- gasdev
  - stoch\_mod, 153
- go

- params, 103
- gp
  - params, 104
- gset
  - stoch\_mod, 156
- h
  - inprod\_analytic::atm\_wavenum, 223
  - inprod\_analytic::ocean\_wavenum, 229
  - params, 104
- h1
  - mtv\_int\_tensor, 94
- h2
  - mtv\_int\_tensor, 94
- h3
  - mtv\_int\_tensor, 94
- htot
  - mtv\_int\_tensor, 95
- hx
  - dec\_tensor, 53
- hy
  - dec\_tensor, 53
- i
  - stat, 152
- IMPLIED
  - LICENSE.txt, 241
- ic
  - ic\_def, 57
- ic\_def, 55
  - exists, 57
  - ic, 57
  - load\_ic, 55
- ic\_def.f90, 235
- ind
  - sf\_def, 137
- ind1
  - sigma, 141
- ind2
  - sigma, 141
- init\_adtensor
  - tl\_ad\_tensor, 199
- init\_adtensor\_ref
  - tl\_ad\_tensor, 199
- init\_aotensor
  - aotensor\_def, 27
- init\_corr
  - corrmod, 39
- init\_corr\_tensor
  - corr\_tensor, 30
- init\_corrint
  - int\_corr, 75
- init\_dec\_tensor
  - dec\_tensor, 45
- init\_g
  - rk2\_mtv\_integrator, 115
- init\_inprod
  - inprod\_analytic, 65
- init\_integrator
  - integrator, 78
  - rk2\_mtv\_integrator, 115
  - rk2\_stoch\_integrator, 125
  - rk2\_wl\_integrator, 130
- init\_mar
  - mar, 81
- init\_memory
  - memory, 86
- init\_mtv\_int\_tensor
  - mtv\_int\_tensor, 90
- init\_nml
  - params, 100
- init\_noise
  - rk2\_mtv\_integrator, 115
- init\_one
  - util, 205
- init\_params
  - params, 100
- init\_random\_seed
  - util, 206
- init\_sigma
  - sigma, 140
- init\_sqrt
  - sqrt\_mod, 145
- init\_ss\_integrator
  - rk2\_ss\_integrator, 121
- init\_stat
  - stat, 150
- init\_stoch\_params
  - stoch\_params, 158
- init\_sub\_tensor
  - dec\_tensor, 49
- init\_tl\_ad\_integrator
  - tl\_ad\_integrator, 193
- init\_tltensor
  - tl\_ad\_tensor, 200
- init\_wl\_tensor
  - wl\_tensor, 211
- inprod\_analytic, 57
  - atmos, 67
  - awavenum, 67
  - b1, 59
  - b2, 59
  - calculate\_a, 59
  - calculate\_b, 60
  - calculate\_c\_atm, 60
  - calculate\_c\_oc, 60
  - calculate\_d, 61
  - calculate\_g, 61
  - calculate\_k, 62
  - calculate\_m, 62
  - calculate\_n, 63
  - calculate\_o, 63
  - calculate\_s, 63
  - calculate\_w, 64
  - delta, 64
  - flambda, 64
  - init\_inprod, 65

- ocean, 67
- owavenum, 67
- s1, 66
- s2, 66
- s3, 66
- s4, 66
- inprod\_analytic.f90, 235
- inprod\_analytic::atm\_tensors, 221
  - a, 221
  - b, 221
  - c, 221
  - d, 222
  - g, 222
  - s, 222
- inprod\_analytic::atm\_wavenum, 222
  - h, 223
  - m, 223
  - nx, 223
  - ny, 223
  - p, 223
  - typ, 223
- inprod\_analytic::ocean\_tensors, 227
  - c, 228
  - k, 228
  - m, 228
  - n, 228
  - o, 228
  - w, 229
- inprod\_analytic::ocean\_wavenum, 229
  - h, 229
  - nx, 229
  - ny, 229
  - p, 230
- inprod\_analytic\_test
  - test\_inprod\_analytic.f90, 265
- int\_comp, 68
  - integrate, 68
  - midexp, 69
  - midpnt, 69
  - polint, 70
  - qromb, 70
  - qromo, 71
  - trapzd, 71
- int\_comp.f90, 237
- int\_corr, 72
  - comp\_corrint, 73
  - corr2int, 75
  - corrint, 75
  - func\_ij, 74
  - func\_ijkl, 75
  - init\_corrint, 75
  - oi, 76
  - oj, 76
  - ok, 76
  - ol, 76
  - real\_eps, 76
- int\_corr.f90, 237
- int\_corr\_mode
  - stoch\_params, 159
- int\_tensor
  - rk2\_stoch\_integrator, 128
- integrate
  - int\_comp, 68
- integrator, 76
  - buf\_f0, 79
  - buf\_f1, 79
  - buf\_ka, 79
  - buf\_kb, 79
  - buf\_y1, 79
  - init\_integrator, 78
  - step, 78
  - tendencies, 78
- inv\_corr\_i
  - corrmod, 42
- inv\_corr\_i\_full
  - corrmod, 42
- invmat
  - util, 206
- ireduce
  - util, 206
- iset
  - stoch\_mod, 156
- isin
  - util, 206
- iter
  - stat, 151
- j
  - tensor::coolist\_elem, 226
  - tensor::coolist\_elem4, 227
- jacobian
  - tl\_ad\_tensor, 200
- jacobian\_mat
  - tl\_ad\_tensor, 201
- jsparse\_mul
  - tensor, 175
- jsparse\_mul\_mat
  - tensor, 176
- k
  - inprod\_analytic::ocean\_tensors, 228
  - params, 104
  - tensor::coolist\_elem, 226
  - tensor::coolist\_elem4, 227
- KIND
  - LICENSE.txt, 241
- kd
  - params, 104
- kdelta
  - aotensor\_def, 28
- kdp
  - params, 104
- khi
  - corrmod, 42
- klo
  - corrmod, 42
- kp

- params, 105
- l
  - params, 105
  - tensor::coolist\_elem4, 227
- l1
  - mtv\_int\_tensor, 95
  - wl\_tensor, 216
- l2
  - mtv\_int\_tensor, 95
  - wl\_tensor, 216
- l3
  - mtv\_int\_tensor, 95
- l4
  - wl\_tensor, 217
- l5
  - wl\_tensor, 217
- LIABILITY
  - LICENSE.txt, 241
- LICENSE.txt, 238
  - CLAIM, 240
  - CONTRACT, 240
  - charge, 240
  - conditions, 240
  - copy, 240
  - distribute, 240
  - FROM, 240
  - files, 239
  - IMPLIED, 241
  - KIND, 241
  - LIABILITY, 241
  - License, 240
  - MERCHANTABILITY, 241
  - merge, 241
  - modify, 241
  - OTHERWISE, 241
  - publish, 242
  - restriction, 242
  - so, 242
  - Software, 242
  - sublicense, 242
  - use, 242
- lambda
  - params, 105
- lcg
  - util.f90, 270
- ldef
  - wl\_tensor, 217
- License
  - LICENSE.txt, 240
- load\_ic
  - ic\_def, 55
- load\_mode
  - stoch\_params, 159
- load\_sf
  - sf\_def, 135
- load\_tensor4\_from\_file
  - tensor, 176
- load\_tensor\_from\_file
  - tensor, 178
- lpa
  - params, 105
- lpo
  - params, 105
- lr
  - params, 106
- lsbpa
  - params, 106
- lsbpo
  - params, 106
- ltot
  - mtv\_int\_tensor, 95
  - wl\_tensor, 217
- lwork
  - sqrt\_mod, 149
- lxx
  - dec\_tensor, 53
- lxy
  - dec\_tensor, 53
- lyx
  - dec\_tensor, 54
- lyy
  - dec\_tensor, 54
- m
  - inprod\_analytic::atm\_wavenum, 223
  - inprod\_analytic::ocean\_tensors, 228
  - stat, 152
- m11
  - wl\_tensor, 217
- m12
  - wl\_tensor, 217
- m12def
  - wl\_tensor, 218
- m13
  - wl\_tensor, 218
- m1tot
  - wl\_tensor, 218
- m21
  - wl\_tensor, 218
- m21def
  - wl\_tensor, 218
- m22
  - wl\_tensor, 218
- m22def
  - wl\_tensor, 219
- MAR.f90, 244
- MERCHANTABILITY
  - LICENSE.txt, 241
- MTV\_int\_tensor.f90, 246
- MTV\_sigma\_tensor.f90, 247
- maooam
  - maooam.f90, 243
  - maooam.f90, 242
  - maooam, 243
  - maooam\_MTV.f90, 243
  - maooam\_mtv, 243
  - maooam\_WL.f90, 244

- maooam\_wl, 244
- maooam\_mtv
  - maooam\_MTV.f90, 243
- maooam\_stoch
  - maooam\_stoch.f90, 244
- maooam\_stoch.f90, 243
  - maooam\_stoch, 244
- maooam\_wl
  - maooam\_WL.f90, 244
- mar, 80
  - buf\_y, 82
  - dw, 82
  - init\_mar, 81
  - mar\_step, 81
  - mar\_step\_red, 82
  - ms, 83
  - q, 83
  - qred, 83
  - rred, 83
  - stoch\_vec, 82
  - w, 83
  - wred, 83
- mar\_step
  - mar, 81
- mar\_step\_red
  - mar, 82
- mat\_contract
  - util, 207
- mat\_to\_coo
  - tensor, 179
- mat\_trace
  - util, 207
- matc\_to\_coo
  - tensor, 179
- maxint
  - stoch\_params, 159
- mdef
  - wl\_tensor, 219
- mean
  - corrmod, 42
  - stat, 151
- mean\_full
  - corrmod, 42
- meml
  - stoch\_params, 159
- memory, 84
  - buf\_m, 87
  - buf\_m3, 87
  - compute\_m3, 85
  - init\_memory, 86
  - step, 87
  - t\_index, 87
  - test\_m3, 86
  - x, 87
  - xs, 88
  - zs, 88
- memory.f90, 245
- mems
  - stoch\_params, 159
- merge
  - LICENSE.txt, 241
- midexp
  - int\_comp, 69
- midpnt
  - int\_comp, 69
- mnuti
  - stoch\_params, 160
- mode
  - stoch\_params, 160
- modify
  - LICENSE.txt, 241
- mprev
  - stat, 152
- ms
  - mar, 83
- mtmp
  - stat, 152
- mtot
  - mtv\_int\_tensor, 96
  - wl\_tensor, 219
- mtv\_int\_tensor, 88
  - b1, 92
  - b2, 93
  - btot, 93
  - dumb\_mat1, 93
  - dumb\_mat2, 93
  - dumb\_mat3, 93
  - dumb\_mat4, 94
  - dumb\_vec, 94
  - h1, 94
  - h2, 94
  - h3, 94
  - htot, 95
  - init\_mtv\_int\_tensor, 90
  - l1, 95
  - l2, 95
  - l3, 95
  - ltot, 95
  - mtot, 96
  - q1, 96
  - q2, 96
  - utot, 96
  - vtot, 96
- mult
  - rk2\_mtv\_integrator, 119
- muti
  - stoch\_params, 160
- n
  - inprod\_analytic::ocean\_tensors, 228
  - params, 106
- n1
  - sigma, 141
- n2
  - sigma, 142
- n\_res
  - sf\_def, 137

n\_unres  
     sf\_def, 138  
 natm  
     params, 106  
 nbatm  
     params, 107  
 nboc  
     params, 107  
 ndim  
     params, 107  
 nelems  
     tensor::coolist, 224  
     tensor::coolist4, 225  
 noc  
     params, 107  
 noise  
     rk2\_mtv\_integrator, 119  
 noisemult  
     rk2\_mtv\_integrator, 119  
 nspl  
     corrmod, 43  
 nua  
     params, 107  
 nuap  
     params, 108  
 nuo  
     params, 108  
 nuop  
     params, 108  
 nx  
     inprod\_analytic::atm\_wavenum, 223  
     inprod\_analytic::ocean\_wavenum, 229  
 ny  
     inprod\_analytic::atm\_wavenum, 223  
     inprod\_analytic::ocean\_wavenum, 229  
 o  
     inprod\_analytic::ocean\_tensors, 228  
 OTHERWISE  
     LICENSE.txt, 241  
 ocean  
     inprod\_analytic, 67  
 oi  
     int\_corr, 76  
 oj  
     int\_corr, 76  
 ok  
     int\_corr, 76  
 ol  
     int\_corr, 76  
 oms  
     params, 108  
 owavenum  
     inprod\_analytic, 67  
 p  
     inprod\_analytic::atm\_wavenum, 223  
     inprod\_analytic::ocean\_wavenum, 230  
 params, 97  
 ams, 101  
 betp, 101  
 ca, 101  
 co, 101  
 cpa, 102  
 cpo, 102  
 d, 102  
 dp, 102  
 dt, 102  
 epsa, 103  
 f0, 103  
 g, 103  
 ga, 103  
 go, 103  
 gp, 104  
 h, 104  
 init\_nml, 100  
 init\_params, 100  
 k, 104  
 kd, 104  
 kdp, 104  
 kp, 105  
 l, 105  
 lambda, 105  
 lpa, 105  
 lpo, 105  
 lr, 106  
 lsbpa, 106  
 lsbpo, 106  
 n, 106  
 natm, 106  
 nbatm, 107  
 nboc, 107  
 ndim, 107  
 noc, 107  
 nua, 107  
 nuap, 108  
 nuo, 108  
 nuop, 108  
 oms, 108  
 phi0, 108  
 phi0\_npi, 109  
 pi, 109  
 r, 109  
 rp, 109  
 rr, 109  
 rra, 110  
 sb, 110  
 sbpa, 110  
 sbpo, 110  
 sc, 110  
 scale, 111  
 sig0, 111  
 t\_run, 111  
 t\_trans, 111  
 ta0, 111  
 to0, 112  
 tw, 112

- writeout, 112
- params.f90, 248
- phi0
  - params, 108
- phi0\_npi
  - params, 109
- pi
  - params, 109
- piksrt
  - util, 207
- polint
  - int\_comp, 70
- print\_tensor
  - tensor, 180
- print\_tensor4
  - tensor, 180
- printmat
  - util, 208
- psi
  - aotensor\_def, 28
- publish
  - LICENSE.txt, 242
- q
  - mar, 83
- q1
  - mtv\_int\_tensor, 96
- q1fill
  - rk2\_mtv\_integrator, 119
- q2
  - mtv\_int\_tensor, 96
- q\_ar
  - stoch\_params, 160
- q\_au
  - stoch\_params, 160
- q\_or
  - stoch\_params, 161
- q\_ou
  - stoch\_params, 161
- qred
  - mar, 83
- qromb
  - int\_comp, 70
- qromo
  - int\_comp, 71
- r
  - params, 109
- real\_eps
  - aotensor\_def, 29
  - int\_corr, 76
  - sqr\_mod, 149
  - tensor, 191
  - tl\_ad\_tensor, 203
- reduce
  - util, 208
- reorder
  - dec\_tensor, 49
- reset
  - stat, 151
- restriction
  - LICENSE.txt, 242
- rind
  - sf\_def, 138
- rind1
  - sigma, 142
- rind2
  - sigma, 142
- rk2\_MTV\_integrator.f90, 251
- rk2\_WL\_integrator.f90, 254
- rk2\_integrator.f90, 251
- rk2\_mtv\_integrator, 112
  - anoise, 117
  - buf\_f0, 117
  - buf\_f1, 117
  - buf\_g, 117
  - buf\_y1, 117
  - compg, 114
  - compute\_mult, 117
  - dw, 118
  - dwar, 118
  - dwau, 118
  - dwmult, 118
  - dwor, 118
  - dwou, 118
  - full\_step, 114
  - g, 118
  - init\_g, 115
  - init\_integrator, 115
  - init\_noise, 115
  - mult, 119
  - noise, 119
  - noisemult, 119
  - q1fill, 119
  - sq2, 119
  - step, 116
- rk2\_ss\_integrator, 120
  - anoise, 123
  - buf\_f0, 123
  - buf\_f1, 123
  - buf\_y1, 123
  - dwar, 123
  - dwor, 124
  - init\_ss\_integrator, 121
  - ss\_step, 121
  - ss\_tl\_step, 121
  - tendencies, 122
  - tl\_tendencies, 122
- rk2\_ss\_integrator.f90, 252
- rk2\_stoch\_integrator, 124
  - anoise, 127
  - buf\_f0, 127
  - buf\_f1, 127
  - buf\_y1, 127
  - dwar, 127
  - dwau, 127
  - dwor, 128



- dwou, 128
- init\_integrator, 125
- int\_tensor, 128
- step, 126
- tendencies, 126
- rk2\_stoch\_integrator.f90, 253
- rk2\_tl\_ad\_integrator.f90, 254
- rk2\_wl\_integrator, 128
  - anoise, 132
  - buf\_f0, 132
  - buf\_f1, 133
  - buf\_m, 133
  - buf\_m1, 133
  - buf\_m2, 133
  - buf\_m3, 133
  - buf\_m3s, 133
  - buf\_y1, 133
  - compute\_m1, 129
  - compute\_m2, 130
  - dwar, 133
  - dwau, 134
  - dwor, 134
  - dwou, 134
  - full\_step, 130
  - init\_integrator, 130
  - step, 131
  - x1, 134
  - x2, 134
- rk4\_integrator.f90, 255
- rk4\_tl\_ad\_integrator.f90, 256
- rp
  - params, 109
- rr
  - params, 109
- rra
  - params, 110
- rred
  - mar, 83
- rsf2csf
  - sqrt\_mod, 145
- rstr
  - util, 208
- s
  - inprod\_analytic::atm\_tensors, 222
- s1
  - inprod\_analytic, 66
- s2
  - inprod\_analytic, 66
- s3
  - inprod\_analytic, 66
- s4
  - inprod\_analytic, 66
- sb
  - params, 110
- sbpa
  - params, 110
- sbpo
  - params, 110
- sc
  - params, 110
- scal\_mul\_coo
  - tensor, 181
- scale
  - params, 111
- selectev
  - sqrt\_mod, 146
- sf
  - sf\_def, 138
- sf\_def, 134
  - bar, 137
  - bau, 137
  - bor, 137
  - bou, 137
  - exists, 137
  - ind, 137
  - load\_sf, 135
  - n\_res, 137
  - n\_unres, 138
  - rind, 138
  - sf, 138
  - sl\_ind, 138
  - sl\_rind, 138
- sf\_def.f90, 256
- sf\_tensor
  - dec\_tensor, 54
- sig0
  - params, 111
- sig1
  - sigma, 142
- sig1r
  - sigma, 142
- sig2
  - sigma, 142
- sigma, 139
  - compute\_mult\_sigma, 139
  - dumb\_mat1, 141
  - dumb\_mat2, 141
  - dumb\_mat3, 141
  - dumb\_mat4, 141
  - ind1, 141
  - ind2, 141
  - init\_sigma, 140
  - n1, 141
  - n2, 142
  - rind1, 142
  - rind2, 142
  - sig1, 142
  - sig1r, 142
  - sig2, 142
- simplify
  - tensor, 181
- sl\_ind
  - sf\_def, 138
- sl\_rind
  - sf\_def, 138
- so

- LICENSE.txt, [242](#)
- Software
  - LICENSE.txt, [242](#)
- sparse\_mul2
  - tensor, [182](#)
- sparse\_mul2\_j
  - tensor, [183](#)
- sparse\_mul2\_k
  - tensor, [183](#)
- sparse\_mul3
  - tensor, [184](#)
- sparse\_mul3\_mat
  - tensor, [184](#)
- sparse\_mul3\_with\_mat
  - tensor, [185](#)
- sparse\_mul4
  - tensor, [186](#)
- sparse\_mul4\_mat
  - tensor, [186](#)
- sparse\_mul4\_with\_mat\_jl
  - tensor, [187](#)
- sparse\_mul4\_with\_mat\_kl
  - tensor, [188](#)
- splint
  - corrmod, [40](#)
- sq2
  - rk2\_mtv\_integrator, [119](#)
- sqrt\_mod, [143](#)
  - chol, [143](#)
  - csqrtm\_triu, [144](#)
  - init\_sqrt, [145](#)
  - lwork, [149](#)
  - real\_eps, [149](#)
  - rsf2csf, [145](#)
  - selectev, [146](#)
  - sqrtm, [146](#)
  - sqrtm\_svd, [147](#)
  - sqrtm\_triu, [148](#)
  - work, [149](#)
- sqrt\_mod.f90, [257](#)
- sqrtm
  - sqrt\_mod, [146](#)
- sqrtm\_svd
  - sqrt\_mod, [147](#)
- sqrtm\_triu
  - sqrt\_mod, [148](#)
- ss\_step
  - rk2\_ss\_integrator, [121](#)
- ss\_tensor
  - dec\_tensor, [54](#)
- ss\_tl\_step
  - rk2\_ss\_integrator, [121](#)
- ss\_tl\_tensor
  - dec\_tensor, [54](#)
- stat, [150](#)
  - acc, [150](#)
  - i, [152](#)
  - init\_stat, [150](#)
  - iter, [151](#)
  - m, [152](#)
  - mean, [151](#)
  - mprev, [152](#)
  - mtmp, [152](#)
  - reset, [151](#)
  - v, [152](#)
  - var, [151](#)
- stat.f90, [258](#)
- step
  - integrator, [78](#)
  - memory, [87](#)
  - rk2\_mtv\_integrator, [116](#)
  - rk2\_stoch\_integrator, [126](#)
  - rk2\_wl\_integrator, [131](#)
- stoch\_atm\_res\_vec
  - stoch\_mod, [153](#)
- stoch\_atm\_unres\_vec
  - stoch\_mod, [154](#)
- stoch\_atm\_vec
  - stoch\_mod, [154](#)
- stoch\_mod, [152](#)
  - gasdev, [153](#)
  - gset, [156](#)
  - iset, [156](#)
  - stoch\_atm\_res\_vec, [153](#)
  - stoch\_atm\_unres\_vec, [154](#)
  - stoch\_atm\_vec, [154](#)
  - stoch\_oc\_res\_vec, [154](#)
  - stoch\_oc\_unres\_vec, [155](#)
  - stoch\_oc\_vec, [155](#)
  - stoch\_vec, [155](#)
- stoch\_mod.f90, [258](#)
- stoch\_oc\_res\_vec
  - stoch\_mod, [154](#)
- stoch\_oc\_unres\_vec
  - stoch\_mod, [155](#)
- stoch\_oc\_vec
  - stoch\_mod, [155](#)
- stoch\_params, [156](#)
  - dtn, [158](#)
  - dts, [158](#)
  - dtsn, [158](#)
  - eps\_pert, [158](#)
  - init\_stoch\_params, [158](#)
  - int\_corr\_mode, [159](#)
  - load\_mode, [159](#)
  - maxint, [159](#)
  - meml, [159](#)
  - mems, [159](#)
  - mnuti, [160](#)
  - mode, [160](#)
  - muti, [160](#)
  - q\_ar, [160](#)
  - q\_au, [160](#)
  - q\_or, [161](#)
  - q\_ou, [161](#)
  - t\_trans\_mem, [161](#)

- t\_trans\_stoch, 161
  - tdelta, 161
  - x\_int\_mode, 162
- stoch\_params.f90, 259
- stoch\_vec
  - mar, 82
  - stoch\_mod, 155
- str
  - util, 209
- sublicense
  - LICENSE.txt, 242
- suppress\_and
  - dec\_tensor, 50
- suppress\_or
  - dec\_tensor, 50
- t
  - aotensor\_def, 28
- t\_index
  - memory, 87
- t\_run
  - params, 111
- t\_trans
  - params, 111
- t\_trans\_mem
  - stoch\_params, 161
- t\_trans\_stoch
  - stoch\_params, 161
- ta0
  - params, 111
- tdelta
  - stoch\_params, 161
- tendencies
  - integrator, 78
  - rk2\_ss\_integrator, 122
  - rk2\_stoch\_integrator, 126
- tensor, 162
  - add\_check, 165
  - add\_elem, 165
  - add\_matc\_to\_tensor, 166
  - add\_matc\_to\_tensor4, 167
  - add\_to\_tensor, 168
  - add\_vec\_ijk\_to\_tensor4, 169
  - add\_vec\_ikl\_to\_tensor4, 169
  - add\_vec\_ikl\_to\_tensor4\_perm, 170
  - add\_vec\_jk\_to\_tensor, 171
  - coo\_to\_mat\_i, 172
  - coo\_to\_mat\_ij, 173
  - coo\_to\_mat\_ik, 173
  - coo\_to\_mat\_j, 173
  - coo\_to\_vec\_jk, 174
  - copy\_coo, 174
  - jsparse\_mul, 175
  - jsparse\_mul\_mat, 176
  - load\_tensor4\_from\_file, 176
  - load\_tensor\_from\_file, 178
  - mat\_to\_coo, 179
  - matc\_to\_coo, 179
  - print\_tensor, 180
  - print\_tensor4, 180
  - real\_eps, 191
  - scal\_mul\_coo, 181
  - simplify, 181
  - sparse\_mul2, 182
  - sparse\_mul2\_j, 183
  - sparse\_mul2\_k, 183
  - sparse\_mul3, 184
  - sparse\_mul3\_mat, 184
  - sparse\_mul3\_with\_mat, 185
  - sparse\_mul4, 186
  - sparse\_mul4\_mat, 186
  - sparse\_mul4\_with\_mat\_jl, 187
  - sparse\_mul4\_with\_mat\_kl, 188
  - tensor4\_empty, 188
  - tensor4\_to\_coo4, 189
  - tensor\_empty, 189
  - tensor\_to\_coo, 190
  - write\_tensor4\_to\_file, 190
  - write\_tensor\_to\_file, 191
- tensor.f90, 260
- tensor4\_empty
  - tensor, 188
- tensor4\_to\_coo4
  - tensor, 189
- tensor::coolist, 224
  - elems, 224
  - nelems, 224
- tensor::coolist4, 224
  - elems, 225
  - nelems, 225
- tensor::coolist\_elem, 225
  - j, 226
  - k, 226
  - v, 226
- tensor::coolist\_elem4, 226
  - j, 227
  - k, 227
  - l, 227
  - v, 227
- tensor\_empty
  - tensor, 189
- tensor\_to\_coo
  - tensor, 190
- test\_MAR.f90, 265
  - test\_mar, 265
- test\_MTV\_int\_tensor.f90, 266
  - test\_mtv\_int\_tensor, 266
- test\_MTV\_sigma\_tensor.f90, 266
  - test\_sigma, 267
- test\_WL\_tensor.f90, 268
  - test\_wl\_tensor, 268
- test\_aotensor
  - test\_aotensor.f90, 263
- test\_aotensor.f90, 263
  - test\_aotensor, 263
- test\_corr
  - test\_corr.f90, 264

- test\_corr.f90, 263
  - test\_corr, 264
- test\_corr\_tensor
  - test\_corr\_tensor.f90, 264
- test\_corr\_tensor.f90, 264
  - test\_corr\_tensor, 264
- test\_dec\_tensor
  - test\_dec\_tensor.f90, 264
- test\_dec\_tensor.f90, 264
  - test\_dec\_tensor, 264
- test\_inprod\_analytic.f90, 265
  - inprod\_analytic\_test, 265
- test\_m3
  - memory, 86
- test\_mar
  - test\_MAR.f90, 265
- test\_memory
  - test\_memory.f90, 266
- test\_memory.f90, 266
  - test\_memory, 266
- test\_mtv\_int\_tensor
  - test\_MTV\_int\_tensor.f90, 266
- test\_sigma
  - test\_MTV\_sigma\_tensor.f90, 267
- test\_sqrtm
  - test\_sqrtm.f90, 267
- test\_sqrtm.f90, 267
  - test\_sqrtm, 267
- test\_tl\_ad
  - test\_tl\_ad.f90, 267
- test\_tl\_ad.f90, 267
  - test\_tl\_ad, 267
- test\_wl\_tensor
  - test\_WL\_tensor.f90, 268
- theta
  - aotensor\_def, 28
- tl
  - tl\_ad\_tensor, 201
- tl\_ad\_integrator, 192
  - ad\_step, 192
  - buf\_f0, 194
  - buf\_f1, 194
  - buf\_ka, 194
  - buf\_kb, 194
  - buf\_y1, 194
  - init\_tl\_ad\_integrator, 193
  - tl\_step, 193
- tl\_ad\_tensor, 195
  - ad, 196
  - ad\_add\_count, 196
  - ad\_add\_count\_ref, 197
  - ad\_coeff, 197
  - ad\_coeff\_ref, 198
  - adtensor, 203
  - compute\_adtensor, 198
  - compute\_adtensor\_ref, 198
  - compute\_tltensor, 199
  - count\_elems, 203
  - init\_adtensor, 199
  - init\_adtensor\_ref, 199
  - init\_tltensor, 200
  - jacobian, 200
  - jacobian\_mat, 201
  - real\_eps, 203
  - tl, 201
  - tl\_add\_count, 201
  - tl\_coeff, 202
  - tltensor, 203
  - tl\_ad\_tensor.f90, 268
  - tl\_add\_count
    - tl\_ad\_tensor, 201
  - tl\_coeff
    - tl\_ad\_tensor, 202
  - tl\_step
    - tl\_ad\_integrator, 193
  - tl\_tendencies
    - rk2\_ss\_integrator, 122
  - tltensor
    - tl\_ad\_tensor, 203
  - to0
    - params, 112
  - trapzd
    - int\_comp, 71
  - triu
    - util, 209
  - tw
    - params, 112
  - typ
    - inprod\_analytic::atm\_wavenum, 223
  - use
    - LICENSE.txt, 242
  - util, 203
    - cdiag, 204
    - choldc, 204
    - cprintmat, 205
    - diag, 205
    - floordiv, 205
    - init\_one, 205
    - init\_random\_seed, 206
    - invmat, 206
    - ireduce, 206
    - isin, 206
    - mat\_contract, 207
    - mat\_trace, 207
    - piksort, 207
    - printmat, 208
    - reduce, 208
    - rstr, 208
    - str, 209
    - triu, 209
    - vector\_outer, 209
  - util.f90, 269
    - lcg, 270
  - utot
    - mtv\_int\_tensor, 96

v  
  stat, [152](#)  
  tensor::coolist\_elem, [226](#)  
  tensor::coolist\_elem4, [227](#)  
var  
  stat, [151](#)  
vector\_outer  
  util, [209](#)  
vtot  
  mtv\_int\_tensor, [96](#)  
  
w  
  inprod\_analytic::ocean\_tensors, [229](#)  
  mar, [83](#)  
WL\_tensor.f90, [270](#)  
wl\_tensor, [209](#)  
  b1, [214](#)  
  b14, [214](#)  
  b14def, [214](#)  
  b2, [214](#)  
  b23, [215](#)  
  b23def, [215](#)  
  b3, [215](#)  
  b4, [215](#)  
  dumb\_mat1, [215](#)  
  dumb\_mat2, [215](#)  
  dumb\_mat3, [216](#)  
  dumb\_mat4, [216](#)  
  dumb\_vec, [216](#)  
  init\_wl\_tensor, [211](#)  
  l1, [216](#)  
  l2, [216](#)  
  l4, [217](#)  
  l5, [217](#)  
  ldef, [217](#)  
  ltot, [217](#)  
  m11, [217](#)  
  m12, [217](#)  
  m12def, [218](#)  
  m13, [218](#)  
  m1tot, [218](#)  
  m21, [218](#)  
  m21def, [218](#)  
  m22, [218](#)  
  m22def, [219](#)  
  mdef, [219](#)  
  mtot, [219](#)  
work  
  sqrt\_mod, [149](#)  
wred  
  mar, [83](#)  
write\_tensor4\_to\_file  
  tensor, [190](#)  
write\_tensor\_to\_file  
  tensor, [191](#)  
writeout  
  params, [112](#)  
  
x  
  memory, [87](#)  
  x1  
    rk2\_wl\_integrator, [134](#)  
  x2  
    rk2\_wl\_integrator, [134](#)  
  x\_int\_mode  
    stoch\_params, [162](#)  
  xa  
    corrmod, [43](#)  
  xs  
    memory, [88](#)  
  
  y2  
    corrmod, [43](#)  
  ya  
    corrmod, [43](#)  
  ydy  
    corr\_tensor, [32](#)  
  ydyd  
    corr\_tensor, [32](#)  
  yy  
    corr\_tensor, [33](#)  
  
  zs  
    memory, [88](#)