

Contents

1	Overview	20
1.1	Introduction	20
1.1.1	Philosophy – Advantages over Other Systems	20
1.1.1.1	Open Source Software	20
1.1.1.2	Speed of Development	21
1.1.1.3	Bridging the Gap between Users And Developers	21
1.1.1.4	Link with Other Softwares	21
1.1.2	Information	21
1.1.3	Installation	22
1.1.3.1	Basic Installation	22
1.1.3.2	Installation for Windows Users	23
1.1.4	Tutorial	23
1.1.4.1	Sample Session	23
1.1.5	Note on the Document	25
2	Basic Utilities	26
2.1	config – setting features	26
2.1.1	Default Settings	26
2.1.1.1	Dependencies	26
2.1.1.2	Plug-ins	26
2.1.1.3	Assumptions	27
2.1.1.4	Files	27
2.1.2	Automatic Configuration	27
2.1.2.1	Checks	27
2.1.3	User Settings	28
2.2	bigrandom – random numbers	28
2.2.1	random – random number generator	28
2.2.2	randrange – random integer generator	29
2.2.3	map_choice – choice from image of mapping	29
2.3	bigrange – range-like generator functions	30
2.3.1	count – count up	30
2.3.2	range – range-like iterator	30
2.3.3	arithmetic_progression – arithmetic progression iterator	30
2.3.4	geometric_progression – geometric progression iterator	31

2.3.5	multirange – multiple range iterator	31
2.3.6	multirange_restrictions – multiple range iterator with restrictions	31
2.4	compatibility – Keep compatibility between Python versions	33
2.4.1	set, frozenset	33
2.4.2	card(virtualset)	33
3	Functions	34
3.1	algorithm – basic number theoretic algorithms	34
3.1.1	digital_method – univariate polynomial evaluation	34
3.1.2	digital_method_func – function of univariate polynomial evaluation	34
3.1.3	rl_binary_powering – right-left powering	35
3.1.4	lr_binary_powering – left-right powering	35
3.1.5	window_powering – window powering	35
3.1.6	powering_func – function of powering	36
3.2	arith1 - miscellaneous arithmetic functions	37
3.2.1	floorsqrt – floor of square root	37
3.2.2	floorpowerroot – floor of some power root	37
3.2.3	legendre - Legendre(Jacobi) Symbol	37
3.2.4	modsqrt – square root of a for modulo p	37
3.2.5	expand – p -adic expansion	37
3.2.6	inverse – inverse	38
3.2.7	CRT – Chinese Remainder Theorem	38
3.2.8	AGM – Arithmetic Geometric Mean	38
3.2.9	vp – p -adic valuation	38
3.2.10	issquare - Is it square?	38
3.2.11	log – integer part of logarithm	39
3.2.12	product – product of some numbers	39
3.3	arygcd – binary-like gcd algorithms	40
3.3.1	bit_num – the number of bits	40
3.3.2	binarygcd – gcd by the binary algorithm	40
3.3.3	arygcd_i – gcd over gauss-integer	40
3.3.4	arygcd_w – gcd over Eisenstein-integer	40
3.4	combinatorial – combinatorial functions	42
3.4.1	binomial – binomial coefficient	42
3.4.2	combinationIndexGenerator – iterator for combinations	42
3.4.3	factorial – factorial	42
3.4.4	permutationGenerator – iterator for permutation	42
3.4.5	fallingfactorial – the falling factorial	43
3.4.6	risingfactorial – the rising factorial	43
3.4.7	multinomial – the multinomial coefficient	43
3.4.8	bernoulli – the Bernoulli number	43
3.4.9	catalan – the Catalan number	43
3.4.10	euler – the Euler number	43
3.4.11	bell – the Bell number	44

3.4.12	stirling1 – Stirling number of the first kind	44
3.4.13	stirling2 – Stirling number of the second kind	44
3.4.14	partition_number – the number of partitions	45
3.4.15	partitionGenerator – iterator for partition	45
3.4.16	partition_conjugate – the conjugate of partition	45
3.5	cubic_root – cubic root, residue, and so on	47
3.5.1	c_root_p – cubic root mod p	47
3.5.2	c_residue – cubic residue mod p	47
3.5.3	c_symbol – cubic residue symbol for Eisenstein-integers	47
3.5.4	decompose_p – decomposition to Eisenstein-integers	47
3.5.5	cornacchia – solve $x^2 + dy^2 = p$	48
3.6	ecpp – elliptic curve primality proving	49
3.6.1	ecpp – elliptic curve primality proving	49
3.6.2	hilbert – Hilbert class polynomial	49
3.6.3	dedekind – Dedekind’s eta function	49
3.6.4	cmm – CM method	50
3.6.5	cmm_order – CM method with order	50
3.6.6	cornacchiamodify – Modified cornacchia algorithm	50
3.7	equation – solving equations, congruences	52
3.7.1	e1 – solve equation with degree 1	52
3.7.2	e1_ZnZ – solve congruent equation modulo n with degree 1	52
3.7.3	e2 – solve equation with degree 2	52
3.7.4	e2_Fp – solve congruent equation modulo p with degree 2	53
3.7.5	e3 – solve equation with degree 3	53
3.7.6	e3_Fp – solve congruent equation modulo p with degree 3	53
3.7.7	Newton – solve equation using Newton’s method	53
3.7.8	SimMethod – find all roots simultaneously	54
3.7.9	root_Fp – solve congruent equation modulo p	54
3.7.10	allroots_Fp – solve congruent equation modulo p	54
3.8	gcd – gcd algorithm	56
3.8.1	gcd – the greatest common divisor	56
3.8.2	binarygcd – binary gcd algorithm	56
3.8.3	extgcd – extended gcd algorithm	56
3.8.4	lcm – the least common multiple	56
3.8.5	gcd_of_list – gcd of many integers	57
3.8.6	coprime – coprime check	57
3.8.7	pairwise_coprime – coprime check of many integers	57
3.9	multiplicative – multiplicative number theoretic functions	59
3.9.1	euler – the Euler totient function	59
3.9.2	moebius – the Möbius function	59
3.9.3	sigma – sum of divisor powers)	59
3.10	prime – primality test , prime generation	61
3.10.1	trialDivision – trial division test	61
3.10.2	spsp – strong pseudo-prime test	61
3.10.3	smallSpsp – strong pseudo-prime test for small number	61
3.10.4	miller – Miller’s primality test	61

3.10.5	millerRabin – Miller-Rabin primality test	61
3.10.6	lpsp – Lucas test	62
3.10.7	fpsp – Frobenius test	62
3.10.8	apr – Jacobi sum test	62
3.10.9	primeq – primality test automatically	62
3.10.10	prime – n -th prime number	63
3.10.11	nextPrime – generate next prime	63
3.10.12	randPrime – generate random prime	63
3.10.13	generator – generate primes	63
3.10.14	generator_eratosthenes – generate primes using Eratosthenes sieve	63
3.10.15	primonial – product of primes	63
3.10.16	properDivisors – proper divisors	64
3.10.17	primitive_root – primitive root	64
3.10.18	Lucas_chain – Lucas sequence	64
3.11	prime_decomp – prime decomposition	66
3.11.1	prime_decomp – prime decomposition	66
3.12	quad – Imaginary Quadratic Field	67
3.12.1	ReducedQuadraticForm – Reduced Quadratic Form Class	67
3.12.1.1	inverse	68
3.12.1.2	disc	68
3.12.2	ClassGroup – Class Group Class	68
3.12.3	class_formula	69
3.12.4	class_number	69
3.12.5	class_group	69
3.12.6	class_number_bsgs	69
3.12.7	class_group_bsgs	70
3.13	round2 – the round 2 method	71
3.13.1	ModuleWithDenominator – bases of \mathbb{Z} -module with denominator.	72
3.13.1.1	get_rationals – get the bases as a list of rationals	73
3.13.1.2	get_polynomials – get the bases as a list of polynomials	73
3.13.1.3	determinant – determinant of the bases	73
3.13.2	round2(function)	74
3.13.3	Dedekind(function)	74
3.14	squarefree – Squarefreeness tests	75
3.14.1	Definition	75
3.14.2	lenstra – Lenstra's condition	75
3.14.3	trial_division – trial division	75
3.14.4	trivial_test – trivial tests	76
3.14.5	viafactor – via factorization	76
3.14.6	viadecomposition – via partial factorization	76
3.14.7	lenstra_ternary – Lenstra's condition, ternary version	76
3.14.8	trivial_test_ternary – trivial tests, ternary version	77
3.14.9	trial_division_ternary – trial division, ternary version	77

3.14.10	viafactor_ternary – via factorization, ternary version . . .	77
4	Classes	78
4.1	algfield – Algebraic Number Field	78
4.1.1	NumberField – number field	78
4.1.1.1	getConj – roots of polynomial	80
4.1.1.2	disc – polynomial discriminant	80
4.1.1.3	integer_ring – integer ring	80
4.1.1.4	field_discriminant – discriminant	80
4.1.1.5	basis – standard basis	80
4.1.1.6	signature – signature	81
4.1.1.7	POLRED – polynomial reduction	81
4.1.1.8	isIntBasis – check integral basis	81
4.1.1.9	isGaloisField – check Galois field	81
4.1.1.10	isFieldElement – check field element	81
4.1.1.11	getCharacteristic – characteristic	82
4.1.1.12	createElement – create an element	82
4.1.2	BasicAlgNumber – Algebraic Number Class by standard basis	83
4.1.2.1	inverse – inverse	85
4.1.2.2	getConj – roots of polynomial	85
4.1.2.3	getApprox – approximate conjugates	85
4.1.2.4	getCharPoly – characteristic polynomial	85
4.1.2.5	getRing – the field	85
4.1.2.6	trace – trace	85
4.1.2.7	norm – norm	86
4.1.2.8	isAlgInteger – check (algebraic) integer	86
4.1.2.9	ch_matrix – obtain MatAlgNumber object	86
4.1.3	MatAlgNumber – Algebraic Number Class by matrix rep- resentation	87
4.1.3.1	inverse – inverse	89
4.1.3.2	getRing – the field	89
4.1.3.3	trace – trace	89
4.1.3.4	norm – norm	89
4.1.3.5	ch_basic – obtain BasicAlgNumber object	89
4.1.4	changetype(function) – obtain BasicAlgNumber object . .	91
4.1.5	disc(function) – discriminant	91
4.1.6	fppoly(function) – polynomial over finite prime field . . .	91
4.1.7	qpoly(function) – polynomial over rational field	91
4.1.8	zpoly(function) – polynomial over integer ring	92
4.2	elliptic – elliptic class object	93
4.2.1	†ECGeneric – generic elliptic curve class	94
4.2.1.1	simple – simplify the curve coefficient	96
4.2.1.2	changeCurve – change the curve by coordinate change	96

4.2.1.3	changePoint – change coordinate of point on the curve	96
4.2.1.4	coordinateY – Y-coordinate from X-coordinate	96
4.2.1.5	whetherOn – Check point is on curve	97
4.2.1.6	add – Point addition on the curve	97
4.2.1.7	sub – Point subtraction on the curve	97
4.2.1.8	mul – Scalar point multiplication on the curve	97
4.2.1.9	divPoly – division polynomial	97
4.2.2	ECoverQ – elliptic curve over rational field	98
4.2.2.1	point – obtain random point on curve	99
4.2.3	ECoverGF – elliptic curve over finite field	100
4.2.3.1	point – find random point on curve	101
4.2.3.2	naive – Frobenius trace by naive method	101
4.2.3.3	Shanks_Mestre – Frobenius trace by Shanks and Mestre method	101
4.2.3.4	Schoof – Frobenius trace by Schoof’s method	101
4.2.3.5	trace – Frobenius trace	102
4.2.3.6	order – order of group of rational points on the curve	102
4.2.3.7	pointorder – order of point on the curve	102
4.2.3.8	TatePairing – Tate Pairing	103
4.2.3.9	TatePairing_Extend – Tate Pairing with final exponentiation	103
4.2.3.10	WeilPairing – Weil Pairing	103
4.2.3.11	BSGS – point order by Baby-Step and Giant-Step	103
4.2.3.12	DLP_BSGS – solve Discrete Logarithm Problem by Baby-Step and Giant-Step	104
4.2.3.13	structure – structure of group of rational points	104
4.2.3.14	issupersingular – check supersingular curve	104
4.2.4	EC(function)	106
4.3	finitefield – Finite Field	107
4.3.1	†FiniteField – finite field, abstract	108
4.3.2	†FiniteFieldElement – element in finite field, abstract	109
4.3.3	FinitePrimeField – finite prime field	110
4.3.3.1	createElement – create element of finite prime field	111
4.3.3.2	getCharacteristic – get characteristic	111
4.3.3.3	issubring – subring test	111
4.3.3.4	issuperring – superring test	111
4.3.4	FinitePrimeFieldElement – element of finite prime field	112
4.3.4.1	getRing – get ring object	113
4.3.4.2	order – order of multiplicative group	113
4.3.5	ExtendedField – extended field of finite field	114
4.3.5.1	createElement – create element of extended field	115
4.3.5.2	getCharacteristic – get characteristic	115
4.3.5.3	issubring – subring test	115

	4.3.5.4	issuperring – superring test	115
	4.3.5.5	primitive_element – generator of multiplicative group	115
	4.3.6	ExtendedFieldElement – element of finite field	116
	4.3.6.1	getRing – get ring object	117
	4.3.6.2	inverse – inverse element	117
4.4	group	– algorithms for finite groups	118
	4.4.1	†Group – group structure	119
	4.4.1.1	setOperation – change operation	121
	4.4.1.2	†createElement – generate a GroupElement instance	121
	4.4.1.3	†identity – identity element	121
	4.4.1.4	grouporder – order of the group	121
	4.4.2	GroupElement – elements of group structure	123
	4.4.2.1	setOperation – change operation	125
	4.4.2.2	†getGroup – generate a Group instance	125
	4.4.2.3	order – order by factorization method	125
	4.4.2.4	t_order – order by baby-step giant-step	125
	4.4.3	†GenerateGroup – group structure with generator	127
	4.4.4	AbelianGenerate – abelian group structure with generator	128
	4.4.4.1	relationLattice – relation between generators	128
	4.4.4.2	computeStructure – abelian group structure	128
4.5	imaginary	– complex numbers and its functions	129
	4.5.1	ComplexField – field of complex numbers	130
	4.5.1.1	createElement – create Imaginary object	131
	4.5.1.2	getCharacteristic – get characteristic	131
	4.5.1.3	issubring – subring test	131
	4.5.1.4	issuperring – superring test	131
	4.5.2	Complex – a complex number	132
	4.5.2.1	getRing – get ring object	133
	4.5.2.2	arg – argument of complex	133
	4.5.2.3	conjugate – complex conjugate	133
	4.5.2.4	copy – copied number	133
	4.5.2.5	inverse – complex inverse	133
	4.5.3	ExponentialPowerSeries – exponential power series	134
	4.5.4	AbsoluteError – absolute error	134
	4.5.5	RelativeError – relative error	134
	4.5.6	exp(function) – exponential value	134
	4.5.7	expi(function) – imaginary exponential value	134
	4.5.8	log(function) – logarithm	134
	4.5.9	sin(function) – sine function	134
	4.5.10	cos(function) – cosine function	134
	4.5.11	tan(function) – tangent function	134
	4.5.12	sinh(function) – hyperbolic sine function	134
	4.5.13	cosh(function) – hyperbolic cosine function	134
	4.5.14	tanh(function) – hyperbolic tangent function	134

4.5.15	atanh(function) – hyperbolic arc tangent function	135
4.5.16	sqrt(function) – square root	135
4.6	intresidue – integer residue	136
4.6.1	IntegerResidueClass – integer residue class	137
4.6.1.1	getRing – get ring object	138
4.6.1.2	getResidue – get residue	138
4.6.1.3	getModulus – get modulus	138
4.6.1.4	inverse – inverse element	138
4.6.1.5	minimumAbsolute – minimum absolute repre- sentative	138
4.6.1.6	minimumNonNegative – smallest non-negative representative	138
4.6.2	IntegerResidueClassRing – ring of integer residue	139
4.6.2.1	createElement – create IntegerResidueClass object	140
4.6.2.2	isfield – field test	140
4.6.2.3	getInstance – get instance of IntegerResidueClass- Ring	140
4.7	lattice – Lattice	141
4.7.1	Lattice – lattice	141
4.7.1.1	createElement – create element	142
4.7.1.2	bilinearForm – bilinear form	142
4.7.1.3	isCyclic – Check whether cyclic lattice or not	142
4.7.1.4	isIdeal – Check whether ideal lattice or not	142
4.7.2	LatticeElement – element of lattice	143
4.7.2.1	getLattice – Find lattice belongs to	144
4.7.3	LLL(function) – LLL reduction	145
4.8	matrix – matrices	146
4.8.1	Matrix – matrices	147
4.8.1.1	map – apply function to elements	149
4.8.1.2	reduce – reduce elements iteratively	149
4.8.1.3	copy – create a copy	149
4.8.1.4	set – set compo	149
4.8.1.5	setRow – set m-th row vector	150
4.8.1.6	setColumn – set n-th column vector	150
4.8.1.7	getRow – get i-th row vector	150
4.8.1.8	getColumn – get j-th column vector	150
4.8.1.9	swapRow – swap two row vectors	150
4.8.1.10	swapColumn – swap two column vectors	151
4.8.1.11	insertRow – insert row vectors	151
4.8.1.12	insertColumn – insert column vectors	151
4.8.1.13	extendRow – extend row vectors	151
4.8.1.14	extendColumn – extend column vectors	151
4.8.1.15	deleteRow – delete row vector	152
4.8.1.16	deleteColumn – delete column vector	152
4.8.1.17	transpose – transpose matrix	152
4.8.1.18	getBlock – block matrix	152

4.8.1.19	subMatrix – submatrix	152
4.8.2	SquareMatrix – square matrices	154
4.8.2.1	isUpperTriangularMatrix – check upper triangular	155
4.8.2.2	isLowerTriangularMatrix – check lower triangular	155
4.8.2.3	isDiagonalMatrix – check diagonal matrix	155
4.8.2.4	isScalarMatrix – check scalar matrix	155
4.8.2.5	isSymmetricMatrix – check symmetric matrix . .	155
4.8.3	RingMatrix – matrix whose elements belong ring	157
4.8.3.1	getCoefficientRing – get coefficient ring	158
4.8.3.2	toFieldMatrix – set field as coefficient ring . . .	158
4.8.3.3	toSubspace – regard as vector space	158
4.8.3.4	hermiteNormalForm (HNF) – Hermite Normal Form	158
4.8.3.5	exthermiteNormalForm (extHNF) – extended Her- mite Normal Form algorithm	158
4.8.3.6	kernelAsModule – kernel as \mathbb{Z} -module	159
4.8.4	RingSquareMatrix – square matrix whose elements belong ring	160
4.8.4.1	getRing – get matrix ring	161
4.8.4.2	isOrthogonalMatrix – check orthogonal matrix . .	161
4.8.4.3	isAlternatingMatrix (isAntiSymmetricMatrix, isSkewSym- metricMatrix) – check alternating matrix	161
4.8.4.4	isSingular – check singular matrix	161
4.8.4.5	trace – trace	161
4.8.4.6	determinant – determinant	162
4.8.4.7	cofactor – cofactor	162
4.8.4.8	commutator – commutator	162
4.8.4.9	characteristicMatrix – characteristic matrix . . .	162
4.8.4.10	adjugateMatrix – adjugate matrix	162
4.8.4.11	cofactorMatrix (cofactors) – cofactor matrix . .	163
4.8.4.12	smithNormalForm (SNF, elementary_divisor) – Smith Normal Form (SNF)	163
4.8.4.13	extsmithNormalForm (extSNF) – Smith Normal Form (SNF)	163
4.8.5	FieldMatrix – matrix whose elements belong field	165
4.8.5.1	kernel – kernel	166
4.8.5.2	image – image	166
4.8.5.3	rank – rank	166
4.8.5.4	inverseImage – inverse image: base solution of linear system	166
4.8.5.5	solve – solve linear system	166
4.8.5.6	columnEchelonForm – column echelon form . . .	167
4.8.6	FieldSquareMatrix – square matrix whose elements be- long field	168
4.8.6.1	triangulate - triangulate by elementary row op- eration	169

4.8.6.2	inverse - inverse matrix	169
4.8.6.3	hessenbergForm - Hessenberg form	169
4.8.6.4	LUDecomposition - LU decomposition	169
4.8.7	†MatrixRing - ring of matrices	170
4.8.7.1	unitMatrix - unit matrix	171
4.8.7.2	zeroMatrix - zero matrix	171
4.8.7.3	getInstance(class function) - get cached instance	172
4.8.8	Subspace - subspace of finite dimensional vector space	173
4.8.8.1	issubspace - check subspace of self	174
4.8.8.2	toBasis - select basis	174
4.8.8.3	supplementBasis - to full rank	174
4.8.8.4	sumOfSubspaces - sum as subspace	174
4.8.8.5	intersectionOfSubspaces - intersection as subspace	174
4.8.8.6	fromMatrix(class function) - create subspace	176
4.8.9	createMatrix[function] - create an instance	177
4.8.10	identityMatrix(unitMatrix)[function] - unit matrix	177
4.8.11	zeroMatrix[function] - zero matrix	177
4.9	module - module/ideal with HNF	179
4.9.1	Submodule - submodule as matrix representation	180
4.9.1.1	getGenerators - generator of module	181
4.9.1.2	isSubmodule - Check whether submodule of self	181
4.9.1.3	isEqual - Check whether self and other are same module	181
4.9.1.4	isContain - Check whether other is in self	181
4.9.1.5	toHNF - change to HNF	181
4.9.1.6	sumOfSubmodules - sum as submodule	182
4.9.1.7	intersectionOfSubmodules - intersection as submodule	182
4.9.1.8	represent_element - represent element as linear combination	182
4.9.1.9	linear_combination - compute linear combination	182
4.9.2	fromMatrix(class function) - create submodule	184
4.9.3	Module - module over a number field	185
4.9.3.1	toHNF - change to hermite normal form(HNF)	187
4.9.3.2	copy - create copy	187
4.9.3.3	intersect - intersection	187
4.9.3.4	issubmodule - Check submodule	187
4.9.3.5	issupermodule - Check supermodule	187
4.9.3.6	represent_element - Represent as linear combination	187
4.9.3.7	change_base_module - Change base	188
4.9.3.8	index - size of module	188
4.9.3.9	smallest_rational - a \mathbf{Z} -generator in the rational field	188
4.9.4	Ideal - ideal over a number field	190
4.9.4.1	inverse - inverse	191

4.9.4.2	issubideal – Check subideal	191
4.9.4.3	issuperideal – Check superideal	191
4.9.4.4	gcd – greatest common divisor	191
4.9.4.5	lcm – least common multiplier	191
4.9.4.6	norm – norm	192
4.9.4.7	isIntegral – Check integral	192
4.9.5	Ideal_with_generator - ideal with generator	193
4.9.5.1	copy - create copy	195
4.9.5.2	to_HNFRepresentation - change to ideal with HNF	195
4.9.5.3	twoElementRepresentation - Represent with two element	195
4.9.5.4	smallest_rational - a \mathbf{Z} -generator in the rational field	195
4.9.5.5	inverse – inverse	195
4.9.5.6	norm – norm	196
4.9.5.7	intersect - intersection	196
4.9.5.8	issubideal – Check subideal	196
4.9.5.9	issuperideal – Check superideal	196
4.10	permute – permutation (symmetric) group	198
4.10.1	Permute – element of permutation group	199
4.10.1.1	setKey – change key	201
4.10.1.2	getValue – get “value”	201
4.10.1.3	getGroup – get PermGroup	201
4.10.1.4	numbering – give the index	201
4.10.1.5	order – order of the element	201
4.10.1.6	ToTranspose – represent as transpositions	202
4.10.1.7	ToCyclic – corresponding ExPermute element	202
4.10.1.8	sgn – sign of the permutation	202
4.10.1.9	types – type of cyclic representation	202
4.10.1.10	ToMatrix – permutation matrix	203
4.10.2	ExPermute – element of permutation group as cyclic rep- resentation	204
4.10.2.1	setKey – change key	206
4.10.2.2	getValue – get “value”	206
4.10.2.3	getGroup – get PermGroup	206
4.10.2.4	order – order of the element	206
4.10.2.5	ToNormal – represent as normal style	206
4.10.2.6	simplify – use simple value	207
4.10.2.7	sgn – sign of the permutation	207
4.10.3	PermGroup – permutation group	208
4.10.3.1	createElement – create an element from seed	209
4.10.3.2	identity – group identity	209
4.10.3.3	identity_c – group identity as cyclic type	209
4.10.3.4	grouporder – order as group	209
4.10.3.5	randElement – random permute element	209

4.11	rational – integer and rational number	211
4.11.1	Integer – integer	212
4.11.1.1	getRing – get ring object	213
4.11.1.2	actAdditive – addition of binary addition chain	213
4.11.1.3	actMultiplicative – multiplication of binary addition chain	213
4.11.2	IntegerRing – integer ring	214
4.11.2.1	createElement – create Integer object	215
4.11.2.2	gcd – greatest common divisor	215
4.11.2.3	extgcd – extended GCD	215
4.11.2.4	lcm – lowest common multiplier	215
4.11.2.5	getQuotientField – get rational field object	215
4.11.2.6	issubring – subring test	215
4.11.2.7	issuperring – superring test	216
4.11.3	Rational – rational number	217
4.11.3.1	getRing – get ring object	218
4.11.3.2	decimalString – represent decimal	218
4.11.3.3	expand – continued-fraction representation	218
4.11.4	RationalField – the rational field	219
4.11.4.1	createElement – create Rational object	220
4.11.4.2	classNumber – get class number	220
4.11.4.3	getQuotientField – get rational field object	220
4.11.4.4	issubring – subring test	220
4.11.4.5	issuperring – superring test	220
4.12	real – real numbers and its functions	221
4.12.1	RealField – field of real numbers	223
4.12.1.1	getCharacteristic – get characteristic	224
4.12.1.2	issubring – subring test	224
4.12.1.3	issuperring – superring test	224
4.12.2	Real – a Real number	225
4.12.2.1	getRing – get ring object	226
4.12.3	Constant – real number with error correction	227
4.12.4	ExponentialPowerSeries – exponential power series	227
4.12.5	AbsoluteError – absolute error	227
4.12.6	RelativeError – relative error	227
4.12.7	exp(function) – exponential value	227
4.12.8	sqrt(function) – square root	227
4.12.9	log(function) – logarithm	227
4.12.10	log1piter(function) – iterator of $\log(1+x)$	227
4.12.11	piGaussLegendre(function) – pi by Gauss-Legendre	227
4.12.12	eContinuedFraction(function) – Napier’s Constant by continued fraction expansion	227
4.12.13	floor(function) – floor the number	228
4.12.14	ceil(function) – ceil the number	228
4.12.15	tranc(function) – round-off the number	228
4.12.16	sin(function) – sine function	228

4.12.17	cos(function) – cosine function	228
4.12.18	tan(function) – tangent function	228
4.12.19	sinh(function) – hyperbolic sine function	228
4.12.20	cosh(function) – hyperbolic cosine function	228
4.12.21	tanh(function) – hyperbolic tangent function	228
4.12.22	asin(function) – arc sine function	229
4.12.23	acos(function) – arc cosine function	229
4.12.24	atan(function) – arc tangent function	229
4.12.25	atan2(function) – arc tangent function	229
4.12.26	hypot(function) – Euclidean distance function	229
4.12.27	pow(function) – power function	229
4.12.28	degrees(function) – convert angle to degree	229
4.12.29	radians(function) – convert angle to radian	229
4.12.30	fabs(function) – absolute value	229
4.12.31	fmod(function) – modulo function over real	229
4.12.32	frexp(function) – expression with base and binary exponent	230
4.12.33	ldexp(function) – construct number from base and binary exponent	230
4.12.34	EulerTransform(function) – iterator yields terms of Euler transform	230
4.13	ring – for ring object	231
4.13.1	†Ring – abstract ring	232
4.13.1.1	createElement – create an element	233
4.13.1.2	getCharacteristic – characteristic as ring	233
4.13.1.3	issubring – check subring	233
4.13.1.4	issuperring – check superring	233
4.13.1.5	getCommonSuperring – get common ring	234
4.13.2	†CommutativeRing – abstract commutative ring	235
4.13.2.1	getQuotientField – create quotient field	236
4.13.2.2	isdomain – check domain	236
4.13.2.3	isnoetherian – check Noetherian domain	236
4.13.2.4	isufd – check UFD	236
4.13.2.5	ispid – check PID	236
4.13.2.6	iseuclidean – check Euclidean domain	237
4.13.2.7	isfield – check field	237
4.13.2.8	registerModuleAction – register action as ring	237
4.13.2.9	hasaction – check if the action has	237
4.13.2.10	getaction – get the registered action	237
4.13.3	†Field – abstract field	239
4.13.3.1	gcd – gcd	240
4.13.4	†QuotientField – abstract quotient field	241
4.13.5	†RingElement – abstract element of ring	242
4.13.5.1	getRing – getRing	243
4.13.6	†CommutativeRingElement – abstract element of commu- tative ring	244
4.13.6.1	mul_module_action – apply a module action	245

4.13.6.2	exact_division – division exactly	245
4.13.7	†FieldElement – abstract element of field	246
4.13.8	†QuotientFieldElement – abstract element of quotient field	247
4.13.9	†Ideal – abstract ideal	248
4.13.9.1	issubset – check subset	249
4.13.9.2	issuperset – check superset	249
4.13.9.3	reduce – reduction with the ideal	249
4.13.10	†ResidueClassRing – abstract residue class ring	250
4.13.11	†ResidueClass – abstract an element of residue class ring	251
4.13.12	†CommutativeRingProperties – properties for Commu- tativeRingProperties	252
4.13.12.1	isfield – check field	253
4.13.12.2	setIsfield – set field	253
4.13.12.3	iseuclidean – check euclidean	253
4.13.12.4	setIseuclidean – set euclidean	253
4.13.12.5	ispid – check PID	254
4.13.12.6	setIspid – set PID	254
4.13.12.7	isufd – check UFD	254
4.13.12.8	setIsufd – set UFD	254
4.13.12.9	isnoetherian – check Noetherian	255
4.13.12.10	setIsnoetherian – set Noetherian	255
4.13.12.11	isdomain – check domain	255
4.13.12.12	setIsdomain – set domain	255
4.13.13	getRingInstance(function)	257
4.13.14	getRing(function)	257
4.13.15	inverse(function)	257
4.13.16	exact_division(function)	257
4.14	vector – vector object and arithmetic	259
4.14.1	Vector – vector class	260
4.14.1.1	copy – copy itself	262
4.14.1.2	set – set other compo	262
4.14.1.3	indexOfNoneZero – first non-zero coordinate	262
4.14.1.4	toMatrix – convert to Matrix object	262
4.14.2	innerProduct(function) – inner product	264
4.15	factor.ecm – ECM factorization	265
4.15.1	ecm – elliptic curve method	265
4.16	factor.find – find a factor	266
4.16.1	trialDivision – trial division	266
4.16.2	pmom – $p - 1$ method	266
4.16.3	rhomethod – ρ method	266
4.17	factor.methods – factoring methods	268
4.17.1	factor – easiest way to factor	268
4.17.2	ecm – elliptic curve method	268
4.17.3	mpqs – multi-polynomial quadratic sieve method	269
4.17.4	pmom – $p - 1$ method	269
4.17.5	rhomethod – ρ method	269

4.17.6	trialDivision – trial division	269
4.18	factor.misc – miscellaneous functions related factoring	271
4.18.1	allDivisors – all divisors	271
4.18.2	primeDivisors – prime divisors	271
4.18.3	primePowerTest – prime power test	271
4.18.4	squarePart – square part	271
4.18.5	FactoredInteger – integer with its factorization	273
4.18.5.1	is_divisible_by	274
4.18.5.2	exact_division	274
4.18.5.3	divisors	274
4.18.5.4	proper_divisors	274
4.18.5.5	prime_divisors	274
4.18.5.6	square_part	274
4.18.5.7	squarefree_part	275
4.18.5.8	copy	275
4.19	factor.mpqqs – MPQS	275
4.19.1	mpqsfind	275
4.19.2	mpqs	275
4.19.3	eratosthenes	276
4.20	factor.util – utilities for factorization	277
4.20.1	FactoringInteger – keeping track of factorization	278
4.20.1.1	getNextTarget – next target	279
4.20.1.2	getResult – result of factorization	279
4.20.1.3	register – register a new factor	279
4.20.1.4	sortFactors – sort factors	279
4.20.2	FactoringMethod – method of factorization	281
4.20.2.1	factor – do factorization	282
4.20.2.2	†continue_factor – continue factorization	282
4.20.2.3	†find – find a factor	282
4.20.2.4	†generate – generate prime factors	282
4.21	poly.factor – polynomial factorization	284
4.21.1	brute_force_search – search factorization by brute force	284
4.21.2	divisibility_test – divisibility test	284
4.21.3	minimum_absolute_injection – send coefficients to minimum absolute representation	284
4.21.4	padic_factorization – p-adic factorization	284
4.21.5	upper_bound_of_coefficient – Landau-Mignotte bound of coefficients	285
4.21.6	zassenhaus – squarefree integer polynomial factorization by Zassenhaus method	285
4.21.7	integerpolynomialfactorization – Integer polynomial factorization	285
4.22	poly.formalsum – formal sum	286
4.22.1	FormalSumContainerInterface – interface class	287
4.22.1.1	construct_with_default – copy-constructing	288
4.22.1.2	iterterms – iterator of terms	288

4.22.1.3	itercoefficients – iterator of coefficients	288
4.22.1.4	iterbases – iterator of bases	288
4.22.1.5	terms – list of terms	288
4.22.1.6	coefficients – list of coefficients	288
4.22.1.7	bases – list of bases	289
4.22.1.8	terms_map – list of terms	289
4.22.1.9	coefficients_map – list of coefficients	289
4.22.1.10	bases_map – list of bases	289
4.22.2	DictFormalSum – formal sum implemented with dictionary	290
4.22.3	ListFormalSum – formal sum implemented with list	290
4.23	poly.groebner – Gröbner Basis	291
4.23.1	buchberger – naïve algorithm for obtaining Gröbner basis	291
4.23.2	normal_strategy – normal algorithm for obtaining Gröbner basis	291
4.23.3	reduce_groebner – reduce Gröbner basis	291
4.23.4	s_polynomial – S-polynomial	292
4.24	poly.hensel – Hensel lift	292
4.24.1	HenselLiftPair – Hensel lift for a pair	294
4.24.1.1	lift – lift one step	295
4.24.1.2	lift_factors – lift a1 and a2	295
4.24.1.3	lift_ladder – lift u1 and u2	295
4.24.2	HenselLiftMulti – Hensel lift for multiple polynomials	295
4.24.2.1	lift – lift one step	297
4.24.2.2	lift_factors – lift factors	297
4.24.2.3	lift_ladder – lift u1 and u2	297
4.24.3	HenselLiftSimultaneously	298
4.24.3.1	lift – lift one step	299
4.24.3.2	first_lift – the first step	299
4.24.3.3	general_lift – next step	299
4.24.4	lift_upto – main function	299
4.25	poly.multiutil – utilities for multivariate polynomials	300
4.25.1	RingPolynomial	301
4.25.1.1	getRing	302
4.25.1.2	getCoefficientRing	302
4.25.1.3	leading_variable	302
4.25.1.4	nest	302
4.25.1.5	unnest	302
4.25.2	DomainPolynomial	302
4.25.2.1	pseudo_divmod	304
4.25.2.2	pseudo_floordiv	304
4.25.2.3	pseudo_mod	304
4.25.2.4	exact_division	304
4.25.3	UniqueFactorizationDomainPolynomial	305
4.25.3.1	gcd	306
4.25.3.2	resultant	306
4.25.4	polynomial – factory function for various polynomials	306

4.25.5	prepare_indeterminates – simultaneous declarations of indeterminates	306
4.26	poly.multivar – multivariate polynomial	308
4.26.1	PolynomialInterface – base class for all multivariate polynomials	309
4.26.2	BasicPolynomial – basic implementation of polynomial	309
4.26.3	TermIndices – Indices of terms of multivariate polynomials	309
4.26.3.1	pop	310
4.26.3.2	gcd	310
4.26.3.3	lcm	310
4.27	poly.ratfunc – rational function	311
4.27.1	RationalFunction – rational function class	312
4.27.1.1	getRing – get rational function field	313
4.28	poly.ring – polynomial rings	314
4.28.1	PolynomialRing – ring of polynomials	315
4.28.1.1	getInstance – classmethod	316
4.28.1.2	getCoefficientRing	316
4.28.1.3	getQuotientField	316
4.28.1.4	issubring	316
4.28.1.5	issuperring	316
4.28.1.6	getCharacteristic	316
4.28.1.7	createElement	316
4.28.1.8	gcd	316
4.28.1.9	isdomain	317
4.28.1.10	iseuclidean	317
4.28.1.11	isnoetherian	317
4.28.1.12	ispid	317
4.28.1.13	isufd	317
4.28.2	RationalFunctionField – field of rational functions	317
4.28.2.1	getInstance – classmethod	318
4.28.2.2	createElement	318
4.28.2.3	getQuotientField	318
4.28.2.4	issubring	318
4.28.2.5	issuperring	318
4.28.2.6	unnest	318
4.28.2.7	gcd	318
4.28.2.8	isdomain	319
4.28.2.9	iseuclidean	319
4.28.2.10	isnoetherian	319
4.28.2.11	ispid	319
4.28.2.12	isufd	319
4.28.3	PolynomialIdeal – ideal of polynomial ring	319
4.28.3.1	reduce	320
4.28.3.2	issubset	320
4.28.3.3	issuperset	320
4.29	poly.termorder – term orders	321

4.29.1	TermOrderInterface – interface of term order	322
4.29.1.1	cmp	323
4.29.1.2	format	323
4.29.1.3	leading_coefficient	323
4.29.1.4	leading_term	323
4.29.2	UnivarTermOrder – term order for univariate polynomials	323
4.29.2.1	format	325
4.29.2.2	degree	325
4.29.2.3	tail_degree	325
4.29.3	MultivarTermOrder – term order for multivariate polynomials	325
4.29.3.1	format	326
4.29.4	weight_order – weight order	326
4.30	poly.uniutil – univariate utilities	327
4.30.1	RingPolynomial – polynomial over commutative ring	328
4.30.1.1	getRing	329
4.30.1.2	getCoefficientRing	329
4.30.1.3	shift_degree_to	329
4.30.1.4	split_at	329
4.30.2	DomainPolynomial – polynomial over domain	329
4.30.2.1	pseudo_divmod	331
4.30.2.2	pseudo_floordiv	331
4.30.2.3	pseudo_mod	331
4.30.2.4	exact_division	331
4.30.2.5	scalar_exact_division	331
4.30.2.6	discriminant	332
4.30.2.7	to_field_polynomial	332
4.30.3	UniqueFactorizationDomainPolynomial – polynomial over UFD	332
4.30.3.1	content	332
4.30.3.2	primitive_part	332
4.30.3.3	subresultant_gcd	333
4.30.3.4	subresultant_extgcd	333
4.30.3.5	resultant	333
4.30.4	IntegerPolynomial – polynomial over ring of rational integers	333
4.30.5	FieldPolynomial – polynomial over field	334
4.30.5.1	content	335
4.30.5.2	primitive_part	335
4.30.5.3	mod	335
4.30.5.4	scalar_exact_division	335
4.30.5.5	gcd	335
4.30.5.6	extgcd	335
4.30.6	FinitePrimeFieldPolynomial – polynomial over finite prime field	336
4.30.6.1	mod_pow – powering with modulus	337
4.30.6.2	pthroot	337

4.30.6.3	squarefree_decomposition	337
4.30.6.4	distinct_degree_decomposition	337
4.30.6.5	split_same_degrees	338
4.30.6.6	factor	338
4.30.6.7	isirreducible	338
4.30.7	polynomial – factory function for various polynomials . .	338
4.31	poly.univar – univariate polynomial	339
4.31.1	PolynomialInterface – base class for all univariate polynomials	340
4.31.1.1	differentiate – formal differentiation	341
4.31.1.2	downshift_degree – decreased degree polynomial	341
4.31.1.3	upshift_degree – increased degree polynomial	341
4.31.1.4	ring_mul – multiplication in the ring	341
4.31.1.5	scalar_mul – multiplication with a scalar	341
4.31.1.6	term_mul – multiplication with a term	341
4.31.1.7	square – multiplication with itself	342
4.31.2	BasicPolynomial – basic implementation of polynomial . .	342
4.31.3	SortedPolynomial – polynomial keeping terms sorted . . .	342
4.31.3.1	degree – degree	343
4.31.3.2	leading_coefficient – the leading coefficient	343
4.31.3.3	leading_term – the leading term	343
4.31.3.4	ring_mul_karatsuba – the leading term	343

Chapter 1

Overview

1.1 Introduction

NZMATH[?] is a number theory oriented calculation system mainly developed by the Nakamura laboratory at Tokyo Metropolitan University. NZMATH system provides you mathematical, especially number-theoretic computational power. It is freely available and distributed under the BSD license. The most distinctive feature of NZMATH is that it is written entirely using a scripting language called Python.

If you want to learn how to start using NZMATH, see Installation (section 1.1.3) and Tutorial (section 1.1.4).

1.1.1 Philosophy – Advantages over Other Systems

In this section, we discuss philosophy of NZMATH, that is, the advantages of NZMATH compared to other similar systems.

1.1.1.1 Open Source Software

Many computational algebra systems, such as Maple[?], Mathematica[?], and Magma[?] are fare-paying systems. These non-free systems are not distributed with source codes. Then, users cannot modify such systems easily. It narrows these system's potentials for users not to take part in developing them. NZMATH, on the other hand, is an open-source software and the source codes are openly available. Furthermore, NZMATH is distributed under the BSD license. BSD license claims as-is and redistribution or commercial use are permitted provided that these packages retain the copyright notice. NZMATH users can develop it just as they like.

1.1.1.2 Speed of Development

We took over developing of SIMATH[?], which was developed under the leadership of Prof. Zimmer at Saarlandes University in Germany. However, it costs a lot of time and efforts to develop these system. Almost all systems including SIMATH are implemented in C or C++ for execution speed, but we have to take the time to work memory management, construction of an interactive interpreter, preparation for multiple precision package and so on. In this regard, we chose Python which is a modern programming language. Python provides automatic memory management, a sophisticated interpreter and many useful packages. We can concentrate on development of mathematical matters by using Python.

1.1.1.3 Bridging the Gap between Users And Developers

KANT/KASH[?] and PARI/GP[?] are similar systems to NZMATH. But programming languages for modifying these systems are different between users and developers. We think the gap makes evolution speed of these systems slow. On the other hand, NZMATH has been developed with Python for bridging this gap. Python grammar is easy to understand and users can read easily codes written by Python. And NZMATH, which is one of Python libraries, works on very wide platform including UNIX/Linux, Macintosh, Windows, and so forth. Users can modify the programs and feedback to developers with a light heart. So developers can absorb their thinking. Then NZMATH will progress to more flexible user-friendly system.

1.1.1.4 Link with Other Softwares

NZMATH distributed as a Python library enables us to link other Python packages with it. For example, NZMATH can be used with IPython[?], which is a comfortable interactive interpreter. And it can be linked with matplotlib[?], which is a powerful graphic software. Also mpmath[?], which is a module for floating-point operation, can improve efficiency of NZMATH. In fact, the module `ecpp` improves performance with mpmath. There are many softwares implemented in Python. Many of these packages are freely available. Users can use NZMATH with these packages and create an unthinkable powerful system.

1.1.2 Information

NZMATH has more than 25 modules. These modules cover a lot of territory including elementary number theoretic methods, combinatorial theoretic methods, solving equations, primality, factorization, multiplicative number theoretic functions, matrix, vector, polynomial, rational field, finite field, elliptic curve, and so on. NZMATH manual for users is at:

<http://tnt.math.se.tmu.ac.jp/nzmth/manual/>

If you are interested in NZMATH, please visit the official website below to obtain more information about it.

<http://tnt.math.se.tmu.ac.jp/nzmeth/>

Note that NZMATH can be used even if users do not have any experience of writing programs in Python.

1.1.3 Installation

In this section, we explain how to install NZMATH. If you use Windows (Windows XP, Windows Vista, Windows 7 etc.) as an operating system (OS), then see 1.1.3.2 “Install for Windows Users”.

1.1.3.1 Basic Installation

There are three steps for installation of NZMATH.

First, check whether Python is installed in the computer. Python 2.5 or a higher version is needed for NZMATH. If you do not have a copy of Python, please install it first. Python is available from <http://www.python.org/>.

Second, download a NZMATH package and expand it. It is distributed at official web site:

<http://tnt.math.se.tmu.ac.jp/nzmeth/download>

or at sourceforge.net:

http://sourceforge.net/project/showfiles.php?group_id=171032

The package can be easily extracted, depending on the operating system. For systems with recent GNU tar, type a single command below:

```
% tar xf NZMATH-*.tar.gz
```

where, % is a command line prompt. With standard tar, type

```
% gzip -cd NZMATH-*.tar.gz | tar xf -
```

. Please read *.* as the version number of which you downloaded the package. For example, if the latest version is 1.0.0, then type the following command.

```
% tar xf NZMATH-1.0.0.tar.gz
```

Then, a subdirectory named NZMATH-*.* is created.

Finally, install NZMATH to the standard python path. Usually, this can be translated into writing files somewhere under /usr/lib or /usr/local/lib. So the appropriate write permission may be required at this step. Typically, type commands below:

```
% cd NZMATH-*.*
% su
# python setup.py install
```

1.1.3.2 Installation for Windows Users

We also distribute installation packages for specific platforms. Especially, we started distributing the installer for Windows in 2007.

Please download the installer (NZMATH-*.*.win32Install.exe) from

<http://tnt.math.se.tmu.ac.jp/nzmeth/download>

or at sourceforge.net:

http://sourceforge.net/project/showfiles.php?group_id=171032

Here, we explain a way of installing NZMATH with the installer. First please open the installer. If you use Windows Vista or higher version, UAC (User Account Control) may ask if you run the program. click "Allow". Then the setup window will open. Following the steps in the setup wizard, you can install NZMATH with only three clicks.

1.1.4 Tutorial

In this section, we describe how to use NZMATH.

1.1.4.1 Sample Session

Start your Python interpreter. That is, open your command interpreter such as Terminal for MacOS or bash/csh for linux, type the strings "python" and press the key Enter.

Examples

```
% python
Python 2.6.1 (r261:67515, Jan 14 2009, 10:59:13)
[GCC 4.1.2 20071124 (Red Hat 4.1.2-42)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

For windows users, it normally means opening IDLE (Python GUI), which is a Python software.

Examples

```
Python 2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
```

```
*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
```

```
interface and no data is sent to or received from the Internet.  
*****
```

```
IDLE 2.6.1  
>>>
```

Here, '>>>' is a Python prompt, which means that the system waits you to input commands.

Then, type:

Examples

```
>>> from nzmeth import *  
>>>
```

This command enables you to use all NZMATH features. If you use only a specific module (the term “module” is explained later), for example, prime, type as the following:

Examples

```
>>> from nzmeth import prime  
>>>
```

You are ready to use NZMATH. For example, type the string “prime.nextPrime(1000)”, then you obtain ‘1009’ as the smallest prime among numbers greater than 1000.

Examples

```
>>> prime.nextPrime(1000)  
1009  
>>>
```

“prime” is a name of a module, which is a NZMATH file including Python codes. “nextPrime” is a name of a function, which outputs values after the system executes some processes for inputs. NZMATH has various functions for mathematical or algorithmic computations. See [3 Functions](#).

Also, we can create some mathematical objects. For example, you may use the module “matrix”. If you want to define the matrix

$$\begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}$$

and compute the square, then type as the following:

Examples

```
>>> A = matrix.Matrix(2, 2, [1, 2]+[5, 6])
>>> print A
1 2
5 6
>>> print A ** 2
11 14
35 46
>>>
```

“Matrix” is a name of a class, which is a template of mathematical objects. See [4](#) Classes for using NZMATH classes.

The command “print” enables us to represent outputs with good-looking forms. The data structure such as “[a, b, c, ...]” is called list. Also, we use various Python data structures like tuple “(a, b, c, ...)”, dictionary “{ $x_1 : y_1, x_2 : y_2, x_3 : y_3, \dots$ }” etc. Note that we do not explain Python’s syntax in detail because it is not absolutely necessary to use NZMATH. However, we recommend that you learn Python for developing your potential. Python grammar are easy to study. For information on how to use Python, see <http://docs.python.org> or many other documents about Python.

1.1.5 Note on the Document

† Some beginnings of lines or blocks such as sections or sentences may be marked †. This means these lines or blocks is for advanced users. For example, the class *FiniteFieldElement* (See **FinitePrimeFieldElement**) is one of abstract classes in NZMATH, which can be inherited to new classes similar to the finite field.

[...] For example, we may sometimes write as *function(a,b[,c,d])*. It means the argument “c, d” or only “d” can be discarded. Such functions use “default argument values”, which is one of the feature of Python.

(See <http://docs.python.org/tutorial/controlflow.html#default-argument-values>)

Warning: Python also have the feature “keyword arguments”. We have tried to keep the feature in NZMATH too. However, some functions cannot be used with this feature because these functions are written expecting that arguments are given in order.

Chapter 2

Basic Utilities

2.1 config – setting features

All constants in the module can be set in user's config file. See the [User Settings](#) section for more detailed description.

2.1.1 Default Settings

2.1.1.1 Dependencies

Some third party / platform dependent modules are possibly used, and they are configurable.

HAVE_MPMATH `mpmath` is a package providing multiprecision math. See its [project page](#). This package is used in **ecpp** module.

HAVE_SQLITE3 `sqlite3` is the default database module for `Python`, but it need to be enabled at the build time.

HAVE_NET Some functions will connect to the Net. Desktop machines are usually connected to the Net, but notebooks may have connectivity only occasionally.

2.1.1.2 Plug-ins

PLUGIN_MATH `Python` standard float/complex types and [math](#)/[cmath](#) modules only provide fixed precision (double precision), but sometimes multi-precision floating point is needed.

2.1.1.3 Assumptions

Some conjectures are useful for assuring the validity of a faster algorithm.

All assumptions are default to `False`, but you can set them `True` if you believe them.

GRH Generalized Riemann Hypothesis. For example, primality test is $O((\log n)^2)$ if GRH is true while $O((\log n)^6)$ or something without it.

2.1.1.4 Files

DATADIR The directory where `NZMATH` (static) data files are stored. The default will be `os.path.join(sys.prefix, 'share', 'nzmth')` or `os.path.join(sys.prefix, 'Data', 'nzmth')` on Windows.

2.1.2 Automatic Configuration

The items above can be set automatically by testing the environment.

2.1.2.1 Checks

Here are check functions.

The constants accompanying the check functions which enable the check if it is `True`, can be overridden in user settings.

Both check functions and constants are not exposed.

check_mpmath() Check whether `mpmath` is available or not.
constant: `CHECK_MPMATH`

check_sqlite3() Check if `sqlite3` is importable or not. `pysqlite2` may be a substitution.
constant: `CHECK_SQLITE3`

check_net() Check the net connection by HTTP call.
constant: `CHECK_NET`

check_plugin_math() Check which math plug-in is available.
constant: `CHECK_PLUGIN_MATH`

default_datadir() Return default value for `DATADIR`.

This function selects the value from various candidates. If this function is called with `DATADIR` set, the value of (previously-defined) `DATADIR` is the first candidate to be returned. Other possibilities are, `sys.prefix + 'Data/nzmth'` on Windows, or `sys.prefix + 'share/nzmth'` on other platforms.

Be careful that all the above paths do not exist, the function returns `None`.

constant: `CHECK_DATADIR`

2.1.3 User Settings

The module try to load the user's config file named *nzmathconf.py*. The search path is the following:

1. The directory which is specified by an environment variable `NZMATHCONFDIR`.
2. If the platform is Windows, then
 - (a) If an environment variable `APPDATA` is set, `APPDATA/nzmath`.
 - (b) If, alternatively, an environment variable `USERPROFILE` is set, `USERPROFILE/Application Data/nzmath`.
3. On other platforms, if an environment variable `HOME` is set, `HOME/.nzmath.d`.

nzmathconf.py is a `Python` script. Users can set the constants like `HAVE_MPMATH`, which will override the default settings. These constants, except assumption ones, are automatically set, unless constants accompanying the check functions are false (see the [Automatic Configuration](#) section above).

2.2 bigrandom – random numbers

Historical Note The module was written for replacement of the `Python` standard module `random`, because in the era of `Python 2.2` (prehistorical period of `NZMATH`) the random module raises `OverflowError` for long integer arguments for the `randrange` function, which is the only function having a use case in `NZMATH`.

After the creation of `Python 2.3`, it was theoretically possible to use `random.randrange`, since it started to accept long integer as its argument. Use of it was, however, not considered, since there had been the `bigrandom` module. It was lucky for us. In fall of 2006, we found a bug in `random.randrange` and reported it (see issue tracker); the `random.randrange` accepts long integers but returns unreliable result for truly big integers. The bug was fixed for `Python 2.5.1`. You can, therefore, use `random.randrange` instead of `bigrandom.randrange` for `Python 2.5.1` or higher.

2.2.1 random – random number generator

`random()` → *float*

Return a random floating point number in the interval $[0, 1)$.

This function is an alias to `random.random` in the `Python` standard library.

2.2.2 randrange – random integer generator

```
randrange(start: integer, stop: integer=None, step: integer=1 )  
→ integer
```

Return a random integer in the range.

The argument names do not correspond to their roles, but users are familiar with the **range** built-in function of Python and understand the semantics. Calling with one argument n , then the result is an integer in the range $[0, n)$ chosen randomly. With two arguments n and m , in $[n, m)$, and with third l , in $[n, m) \cap (n + l\mathbb{Z})$.

This function is almost the same as `random.randrange` in the Python standard library. See the historical note [2.2](#).

Examples

```
>>> randrange(4, 10000, 3)  
9919L  
>>> randrange(4 * 10**60)  
31925916908162253969182327491823596145612834799876775114620151L
```

2.2.3 map_choice – choice from image of mapping

```
map_choice(mapping: function, upperbound: integer )  
→ integer
```

Return a choice from a set given as the image of the mapping from natural numbers (more precisely `range(upperbound)`). In other words, it is equivalent to: `random.choice([mapping(i) for i in range(upperbound)])`, if `upperbound` is small enough for the list size limit.

The mapping can be a partial function, i.e. it may return `None` for some input. However, if the resulting set is empty, it will end up with an infinite loop.

2.3 bigrange – range-like generator functions

2.3.1 count – count up

`count(n: integer=0) → iterator`

Count up infinitely from `n` (default to 0). See [itertools.count](#).

`n` must be int, long or rational.Integer.

2.3.2 range – range-like iterator

`range(start: integer, stop: integer=None, step: integer=1)
→ iterator`

Return a range-like iterator which generates a finite integer sequence.

It can generate more than [sys.maxint](#) elements, which is the limitation of the [range](#) built-in function.

The argument names do not correspond to their roles, but users are familiar with the [range](#) built-in function of Python and understand the semantics. Note that the output is not a list.

Examples

```
>>> range(1, 100, 3) # built-in
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46,
 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82, 85, 88, 91,
 94, 97]
>>> bigrange.range(1, 100, 3)
<generator object at 0x18f8c8>
```

2.3.3 arithmetic_progression – arithmetic progression iterator

`arithmetic_progression(init: integer, difference: integer)
→ iterator`

Return an iterator which generates an arithmetic progression starting from `init` and `difference` step.

2.3.4 `geometric_progression` – geometric progression iterator

```
geometric_progression(init: integer, ratio: integer )  
    → iterator
```

Return an iterator which generates a geometric progression starting from `init` and multiplying `ratio`.

2.3.5 `multirange` – multiple range iterator

```
multirange(triples: list of range triples ) → iterator
```

Return an iterator over Cartesian product of elements of ranges.

Be cautious that using `multirange` usually means you are trying to do brute force looping.

The range triples may be doubles (`start, stop`) or single (`stop,`), but they have to be always tuples.

Examples

```
>>> bigrange.multirange([(1, 10, 3), (1, 10, 4)])  
<generator object at 0x18f968>  
>>> list(_)  
[(1, 1), (1, 5), (1, 9), (4, 1), (4, 5), (4, 9), (7, 1),  
 (7, 5), (7, 9)]
```

2.3.6 `multirange_restrictions` – multiple range iterator with restrictions

```
multirange_restrictions(triples: list of range triples, **kws: keyword arguments )  
    → iterator
```

`multirange_restrictions` is an iterator similar to the `multirange` but putting restrictions on each ranges.

Restrictions are specified by keyword arguments: `ascending`, `descending`, `strictly_ascending` and `strictly_descending`.

A restriction `ascending`, for example, is a sequence that specifies the indices where the number emitted by the range should be greater than or equal to the number at the previous index. Other restrictions `descending`, `strictly_ascending`

and `strictly_descending` are similar. Compare the examples below and of **`multirange`**.

Examples

```
>>> bigrange.multirange_restrictions([(1, 10, 3), (1, 10, 4)], ascending=(1,))
<generator object at 0x18f978>
>>> list(_)
[(1, 1), (1, 5), (1, 9), (4, 5), (4, 9), (7, 9)]
```


2.4 compatibility – Keep compatibility between Python versions

This module should be simply imported:

```
import nzmth.compatibility
```

then it will do its tasks.

2.4.1 set, frozenset

The module provides `set` for Python 2.3. Python ≥ 2.4 have `set` in built-in namespace, while Python 2.3 has `sets` module and `sets.Set`. The `set` the module provides for Python 2.3 is the `sets.Set`. Similarly, `sets.ImmutableSet` would be assigned to `frozenset`. Be careful that the compatibility is not perfect. Note also that `NZMATH`'s recommendation is Python 2.5 or higher in 2.x series.

2.4.2 card(virtualset)

Return cardinality of the virtualset.

The built-in `len()` raises `OverflowError` when the result is greater than `sys.maxint`. It is not clear this restriction will go away in the future. The function `card()` ought to be used instead of `len()` for obtaining cardinality of sets or set-like objects in `nzmth`.

Chapter 3

Functions

3.1 algorithm – basic number theoretic algorithms

3.1.1 digital_method – univariate polynomial evaluation

```
digital_method(coefficients: list, val: object, add: function, mul:  
function, act: function, power: function, zero: object, one: object )  
→ object
```

Evaluate a univariate polynomial corresponding to `coefficients` at `val`.

If the polynomial corresponding to `coefficients` is of R -coefficients for some ring R , then `val` should be in an R -algebra D .

`coefficients` should be a descending ordered list of tuples (d, c) , where d is an integer which expresses the degree and c is an element of R which expresses the coefficient. All operations 'add', 'mul', 'act', 'power', 'zero', 'one' should be explicitly given, where:

'add' means addition ($D \times D \rightarrow D$), 'mul' multiplication ($D \times D \rightarrow D$), 'act' action of R ($R \times D \rightarrow D$), 'power' powering ($D \times \mathbf{Z} \rightarrow D$), 'zero' the additive unit (an constant) in D and 'one', the multiplicative unit (an constant) in D .

3.1.2 digital_method_func – function of univariate polynomial evaluation

```
digital_method(add: function, mul: function, act: function, power:  
function, zero: object, one: object )  
→ function
```

Return a function which evaluates polynomial corresponding to 'coefficients' at 'val' from an iterator 'coefficients' and an object 'val'.

All operations 'add', 'mul', 'act', 'power', 'zero', 'one' should be inputted in

a manner similar to **digital_method**.

3.1.3 rl_binary_powering – right-left powering

```
rl_binary_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using right-left binary method.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

3.1.4 lr_binary_powering – left-right powering

```
lr_binary_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using left-right binary method.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

3.1.5 window_powering – window powering

```
window_powering(element: object, index: integer, mul: function,  
square: function=None, one: object=None, )  
→ object
```

Return `element` to the `index` power by using small-window method.

The window size is selected by average analytic optimization.

`index` should be a non-negative integer. If `square` is None, `square` is defined by using `mul`.

3.1.6 powering_func – function of powering

```
powering_func(mul: function, square: function=None, one: object=None, type: integer=0 )  
→ function
```

Return a function which computes 'element' to the 'index' power from an object 'element' and an integer 'index'.

If `square` is `None`, `square` is defined by using `mul`. `type` should be an integer which means one of the following:

```
0; rl_binary_powering  
1; lr_binary_powering  
2; window_powering
```

Examples

```
>>> d_func = algorithm.digital_method_func(  
... lambda a,b:a+b, lambda a,b:a*b, lambda i,a:i*a, lambda a,i:a**i,  
... matrix.zeroMatrix(3,0), matrix.identityMatrix(3,1)  
... )  
>>> coefficients = [(2,1), (1,2), (0,1)] # X^2+2*X+I  
>>> A = matrix.SquareMatrix(3, [1,2,3]+[4,5,6]+[7,8,9])  
>>> d_func(coefficients, A) # A**2+2*A+I  
[33L, 40L, 48L]+[74L, 92L, 108L]+[116L, 142L, 169L]  
>>> p_func = algorithm.powering_func(lambda a,b:a*b, type=2)  
>>> p_func(A, 10) # A**10 by window method  
[132476037840L, 162775103256L, 193074168672L]+[300005963406L, 368621393481L,  
437236823556L]+[467535888972L, 574467683706L, 681399478440L]
```

3.2 arith1 - miscellaneous arithmetic functions

3.2.1 floorsqrt – floor of square root

floorsqrt(a: *integer*/Rational) → *integer*

Return the floor of square root of a.

3.2.2 floorpowerroot – floor of some power root

floorpowerroot(n: *integer*, k: *integer*) → *integer*

Return the floor of k-th power root of n.

3.2.3 legendre - Legendre(Jacobi) Symbol

legendre(a: *integer*, m: *integer*) → *integer*

Return the Legendre symbol or Jacobi symbol $\left(\frac{a}{m}\right)$.

3.2.4 modsqrt – square root of *a* for modulo *p*

modsqrt(a: *integer*, p: *integer*) → *integer*

Return one of the square roots of a for modulo p if square roots are exist, raise ValueError otherwise.

p must be a prime number.

3.2.5 expand – p-adic expansion

expand(n: *integer*, m: *integer*) → *list*

Return the m-adic expansion of n.

n must be nonnegative integer. m must be greater than or equal to 2. The output is a list of expansion coefficients in ascending order.

3.2.6 inverse – inverse

inverse(*x*: *integer*, *p*: *integer*) → *integer*

Return the inverse of *x* for modulo *p*.

p must be a prime number.

3.2.7 CRT – Chinese Remainder Theorem

CRT(*nlist*: *list*) → *integer*

Return the uniquely determined integer satisfying all modulus conditions given by *nlist*.

Input list *nlist* must be the list of a list consisting of two elements. The first element is remainder and the second is divisor. They must be integer.

3.2.8 AGM – Arithmetic Geometric Mean

AGM(*a*: *integer*, *b*: *integer*) → *float*

Return the Arithmetic-Geometric Mean of *a* and *b*.

3.2.9 vp – *p*-adic valuation

vp(*n*: *integer*, *p*: *integer*, *k*: *integer*=0) → *tuple*

Return the *p*-adic valuation and other part for *n*.

†If *k* is given, return the valuation and the other part for *np^k*.

3.2.10 issquare - Is it square?

issquare(*n*: *integer*) → *integer*

Check if *n* is a square number and return square root of *n* if *n* is a square. Otherwise, return 0.

3.2.11 log – integer part of logarithm

`log(n: integer, base: integer=2) → integer`

Return the integer part of logarithm of `n` to the `base`.

3.2.12 product – product of some numbers

`product(iterable: list, init: integer/Rational=None)
→ prod: integer/Rational`

Return the products of all elements in `iterable`.

If `init` is given, the multiplication starts with `init` instead of the first element in `iterable`.

Input list `iterable` must be list of numbers including integers, **Rational** etc. The output `prod` may be determined by the type of elements of `iterable` and `init`.

Examples

```
>>> arith1.AGM(10, 15)
12.373402181181522
>>> arith1.CRT([[2, 5],[3,7]])
17
>>> arith1.CRT([[2, 5], [3, 7], [5, 11]])
192
>>> arith1.expand(194, 5)
[4, 3, 2, 1]
>>> arith1.vp(54, 3)
(3, 2)
>>> arith1.product([1.5, 2, 2.5])
7.5
>>> arith1.product([3, 4], 2)
24
>>> arith1.product([])
1
```

3.3 arygcd – binary-like gcd algorithms

3.3.1 bit_num – the number of bits

bit_num(a: integer) → integer

Return the number of bits for **a**

3.3.2 binarygcd – gcd by the binary algorithm

binarygcd(a: integer, b: integer) → integer

Return the greatest common divisor (gcd) of two integers **a**, **b** by the binary gcd algorithm.

3.3.3 arygcd_i – gcd over gauss-integer

**arygcd_i(a1: integer, a2: integer, b1: integer, b2: integer)
→ (integer, integer)**

Return the greatest common divisor (gcd) of two gauss-integers **a1+a2i**, **b1+b2i**, where “*i*” denotes the imaginary unit.

If the output of **arygcd_i(a1, a2, b1, b2)** is (**c1**, **c2**), then the gcd of **a1+a2i** and **b1+b2i** equals **c1+c2i**.

†This function uses $(1+i)$ -ary gcd algorithm, which is an generalization of the binary algorithm, proposed by A.Weilert[?].

3.3.4 arygcd_w – gcd over Eisenstein-integer

**arygcd_w(a1: integer, a2: integer, b1: integer, b2: integer)
→ (integer, integer)**

Return the greatest common divisor (gcd) of two Eisenstein-integers **a1+a2ω**, **b1+b2ω**, where “ ω ” denotes a primitive cubic root of unity.

If the output of **arygcd_w(a1, a2, b1, b2)** is (**c1**, **c2**), then the gcd of **a1+a2ω** and **b1+b2ω** equals **c1+c2ω**.

†This functions uses $(1-\omega)$ -ary gcd algorithm, which is an generalization of the binary algorithm, proposed by I.B. Damgård and G.S. Frandsen [?].

Examples

```
>>> arygcd.binarygcd(32, 48)
16
>>> arygcd_i(1, 13, 13, 9)
(-3, 1)
>>> arygcd_w(2, 13, 33, 15)
(4, 5)
```

3.4 combinatorial – combinatorial functions

3.4.1 binomial – binomial coefficient

`binomial(n: integer, m: integer) → integer`

Return the binomial coefficient for n and m . In other words, $\frac{n!}{(n-m)!m!}$.

†For convenience, `binomial(n, n+i)` returns 0 for positive i , and `binomial(0,0)` returns 1.

n must be a positive integer and m must be a non-negative integer.

3.4.2 combinationIndexGenerator – iterator for combinations

`combinationIndexGenerator(n: integer, m: integer) → iterator`

Return an iterator which generates indices of m element subsets of n element set.

`combination_index_generator` is an alias of `combinationIndexGenerator`.

3.4.3 factorial – factorial

`factorial(n: integer) → integer`

Return $n!$ for non-negative integer n .

3.4.4 permutationGenerator – iterator for permutation

`permutationGenerator(n: integer) → iterator`

Generate all permutations of n elements as list iterator.

The number of generated list is n 's **factorial**, so be careful to use big n .

`permutation_generator` is an alias of `permutationGenerator`.

3.4.5 fallingfactorial – the falling factorial

fallingfactorial(*n*: *integer*, *m*: *integer*) → *integer*

Return the falling factorial; *n* to the *m* falling, i.e. $n(n-1)\cdots(n-m+1)$.

3.4.6 risingfactorial – the rising factorial

risingfactorial(*n*: *integer*, *m*: *integer*) → *integer*

Return the rising factorial; *n* to the *m* rising, i.e. $n(n+1)\cdots(n+m-1)$.

3.4.7 multinomial – the multinomial coefficient

multinomial(*n*: *integer*, *parts*: *list*) → *integer*

Return the multinomial coefficient.

parts must be a sequence of natural numbers and the sum of elements in *parts* should be equal to *n*.

3.4.8 bernoulli – the Bernoulli number

bernoulli(*n*: *integer*) → *Rational*

Return the *n*-th Bernoulli number.

3.4.9 catalan – the Catalan number

catalan(*n*: *integer*) → *integer*

Return the *n*-th Catalan number.

3.4.10 euler – the Euler number

euler(*n*: *integer*) → *integer*

Return the *n*-th Euler number.

3.4.11 bell – the Bell number

bell(n: *integer*) → *integer*

Return the n-th Bell number.

The Bell number b is defined by:

$$b(n) = \sum_{i=0}^n S(n, i),$$

where S denotes Stirling number of the second kind (**stirling2**).

3.4.12 stirling1 – Stirling number of the first kind

stirling1(n: *integer*, m: *integer*) → *integer*

Return Stirling number of the first kind.

Let s denote the Stirling number and $(x)_n$ the falling factorial, then

$$(x)_n = \sum_{i=0}^n s(n, i)x^i.$$

s satisfies the recurrence relation:

$$s(n, m) = s(n-1, m-1) - (n-1)s(n-1, m).$$

3.4.13 stirling2 – Stirling number of the second kind

stirling2(n: *integer*, m: *integer*) → *integer*

Return Stirling number of the second kind.

Let S denote the Stirling number, $(x)_i$ falling factorial, then:

$$x^n = \sum_{i=0}^n S(n, i)(x)_i$$

S satisfies:

$$S(n, m) = S(n-1, m-1) + mS(n-1, m)$$

3.4.14 `partition_number` – the number of partitions

`partition_number(n: integer) → integer`

Return the number of partitions of `n`.

3.4.15 `partitionGenerator` – iterator for partition

`partitionGenerator(n: integer, maxi: integer=0) → iterator`

Return an iterator which generates partitions of `n`.

If `maxi` is given, then summands are limited not to exceed `maxi`.

The number of partitions (given by `partition_number`) grows exponentially, so be careful to use big `n`.

`partition_generator` is an alias of `partitionGenerator`.

3.4.16 `partition_conjugate` – the conjugate of partition

`partition_conjugate(partition: tuple) → tuple`

Return the conjugate of partition.

Examples

```
>>> combinatorial.binomial(5, 2)
10L
>>> combinatorial.factorial(3)
6L
>>> combinatorial.fallingfactorial(7, 3) == 7 * 6 * 5
True
>>> combinatorial.risingfactorial(7, 3) == 7 * 8 * 9
True
>>> combinatorial.multinomial(7, [2, 2, 3])
210L
>>> for idx in combinatorial.combinationIndexGenerator(5, 3):
...     print idx
...
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
```

```

[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
>>> for part in combinatorial.partitionGenerator(5):
...     print part
...
(5,)
(4, 1)
(3, 2)
(3, 1, 1)
(2, 2, 1)
(2, 1, 1, 1)
(1, 1, 1, 1, 1)
>>> combinatorial.partition_number(5)
7
>>> def limited_summands(n, maxi):
...     "partition with limited number of summands"
...     for part in combinatorial.partitionGenerator(n, maxi):
...         yield combinatorial.partition_conjugate(part)
...
>>> for part in limited_summands(5, 3):
...     print part
...
(2, 2, 1)
(3, 1, 1)
(3, 2)
(4, 1)
(5,)

```

3.5 cubic_root – cubic root, residue, and so on

3.5.1 c_root_p – cubic root mod p

c_root_p(a: integer, p: integer) → list

Return the cubic root of **a** modulo prime **p**. (i.e. solutions of the equation $x^3 = a \pmod{p}$).

p must be a prime integer.
This function returns the list of all cubic roots of **a**.

3.5.2 c_residue – cubic residue mod p

c_residue(a: integer, p: integer) → integer

Check whether the rational integer **a** is cubic residue modulo prime **p**.

If $p \mid a$, then this function returns 0, elif **a** is cubic residue modulo **p**, then it returns 1, otherwise (i.e. cubic non-residue), it returns -1 .

p must be a prime integer.

3.5.3 c_symbol – cubic residue symbol for Eisenstein-integers

**c_symbol(a1: integer, a2: integer, b1: integer, b2: integer)
→ integer**

Return the (Jacobi) cubic residue symbol of two Eisenstein-integers $\left(\frac{a1+a2\omega}{b1+b2\omega}\right)_3$, where ω is a primitive cubic root of unity.

If $b1 + b2\omega$ is a prime in $\mathbb{Z}[\omega]$, it shows $a1 + a2\omega$ is cubic residue or not.

We assume that $b1 + b2\omega$ is not divisible $1 - \omega$.

3.5.4 decompose_p – decomposition to Eisenstein-integers

decompose_p(p: integer) → (integer, integer)

Return one of prime factors of **p** in $\mathbb{Z}[\omega]$.

If the output is (a, b), then $\frac{p}{a+b\omega}$ is a prime in $\mathbb{Z}[\omega]$. In other words, p

decomposes into two prime factors $\mathbf{a} + \mathbf{b}\omega$ and $\mathbf{p}/(\mathbf{a} + \mathbf{b}\omega)$ in $\mathbb{Z}[\omega]$.

\mathbf{p} must be a prime rational integer. We assume that $\mathbf{p} \equiv 1 \pmod{3}$.

3.5.5 cornacchia – solve $x^2 + dy^2 = p$

cornacchia(*d*: integer, *p*: integer) \rightarrow (integer, integer)

Return the solution of $x^2 + dy^2 = p$.

This function uses Cornacchia's algorithm. See [?].

\mathbf{p} must be prime rational integer. \mathbf{d} must be satisfied with the condition $0 < \mathbf{d} < \mathbf{p}$. This function returns (\mathbf{x} , \mathbf{y}) as one of solutions of the equation $x^2 + dy^2 = p$.

Examples

```
>>> cubic_root.c_root_p(1, 13)
[1, 3, 9]
>>> cubic_root.c_residue(2, 7)
-1
>>> cubic_root.c_symbol(3, 6, 5, 6)
1
>>> cubic_root.decomposi_p(19)
(2, 5)
>>> cubic_root.cornacchia(5, 29)
(3, 2)
```


3.6 ecpp – elliptic curve primality proving

The module consists of various functions for ECPP (Elliptic Curve Primality Proving).

It is probable that the module will be refactored in the future so that each function be placed in other modules.

The ecpp module requires mpmath.

3.6.1 ecpp – elliptic curve primality proving

ecpp(*n*: *integer*, *era*: *list*=None) → *bool*

Do elliptic curve primality proving.
If *n* is prime, return True. Otherwise, return False.

The optional argument *era* is a list of primes (which stands for ERAtosthenes).

n must be a big integer.

3.6.2 hilbert – Hilbert class polynomial

hilbert(*D*: *integer*) → (*integer*, *list*)

Return the class number and Hilbert class polynomial for the imaginary quadratic field with fundamental discriminant *D*.

Note that this function returns Hilbert class polynomial as a list of coefficients.

†If the option **HAVE_NET** is set, at first try to retrieve the data in <http://hilbert-class-polynomial.appspot.com/>. If the data corresponding to *D* is not found, compute the Hilbert polynomial directly (for a long time).

D must be negative int or long. See [?].

3.6.3 dedekind – Dedekind’s eta function

dedekind(*tau*: *mpmath.mpc*, *floatpre*: *integer*) → *mpmath.mpc*

Return Dedekind’s eta of a complex number *tau* in the upper half-plane.

Additional argument *floatpre* specifies the precision of calculation in decimal digits.

`floatpre` must be positive int.

3.6.4 `cmm` – CM method

`cmm(p: integer) → list`

Return curve parameters for CM curves.

If you also need its orders, use **`cmm_order`**.

A prime `p` has to be odd.
This function returns a list of `(a, b)`, where `(a, b)` expresses Weierstrass' short form.

3.6.5 `cmm_order` – CM method with order

`cmm_order(p: integer) → list`

Return curve parameters for CM curves and its orders.

If you need only curves, use **`cmm`**.

A prime `p` has to be odd.
This function returns a list of `(a, b, order)`, where `(a, b)` expresses Weierstrass' short form and `order` is the order of the curve.

3.6.6 `cornacchiamodify` – Modified cornacchia algorithm

`cornacchiamodify(d: integer, p: integer) → list`

Return the solution (u, v) of $u^2 - dv^2 = 4p$.

If there is no solution, raise `ValueError`.

`p` must be a prime integer and `d` be an integer such that $d < 0$ and $d > -4p$ with $d \equiv 0, 1 \pmod{4}$.

Examples

```
>>> ecpp.ecpp(300000000000000000053)
True
>>> ecpp.hilbert(-7)
```

```
(1, [3375, 1])  
>>> ecpp.cmm(7)  
[(6L, 3L), (5L, 4L)]  
>>> ecpp.cornacchiamodify(-7, 29)  
(2, 4)
```

3.7 equation – solving equations, congruences

In the following descriptions, some type aliases are used.

poly_list :

poly_list is a list `[a0, a1, ..., an]` representing a polynomial coefficients in ascending order, i.e., meaning $a_0 + a_1X + \dots + a_nX^n$. The type of each `ai` depends on each function (explained in their descriptions).

integer :

integer is one of *int*, *long* or **Integer**.

complex :

complex includes all number types in the complex field: **integer**, *float*, *complex* of Python, **Rational** of NZMATH, etc.

3.7.1 e1 – solve equation with degree 1

e1(f: poly_list) → complex

Return the solution of linear equation $ax + b = 0$.

`f` ought to be a **poly_list** `[b, a]` of **complex**.

3.7.2 e1_ZnZ – solve congruent equation modulo n with degree 1

e1_ZnZ(f: poly_list, n: integer) → integer

Return the solution of $ax + b \equiv 0 \pmod{n}$.

`f` ought to be a **poly_list** `[b, a]` of **integer**.

3.7.3 e2 – solve equation with degree 2

e2(f: poly_list) → tuple

Return the solution of quadratic equation $ax^2 + bx + c = 0$.

`f` ought to be a **poly_list** `[c, b, a]` of **complex**.

The result tuple will contain exactly 2 roots, even in the case of double root.

3.7.4 e2_Fp – solve congruent equation modulo p with degree 2

e2_Fp(f: **poly_list**, p: *integer*) → *list*

Return the solution of $ax^2 + bx + c \equiv 0 \pmod{p}$.

If the same values are returned, then the values are multiple roots.

f ought to be a **poly_list** of **integers** [c, b, a]. In addition, p must be a prime **integer**.

3.7.5 e3 – solve equation with degree 3

e3(f: **poly_list**) → *list*

Return the solution of cubic equation $ax^3 + bx^2 + cx + d = 0$.

f ought to be a **poly_list** [d, c, b, a] of **complex**.
The result tuple will contain exactly 3 roots, even in the case of including double roots.

3.7.6 e3_Fp – solve congruent equation modulo p with degree 3

e3_Fp(f: **poly_list**, p: *integer*) → *list*

Return the solutions of $ax^3 + bx^2 + cx + d \equiv 0 \pmod{p}$.

If the same values are returned, then the values are multiple roots.

f ought to be a **poly_list** [d, c, b, a] of **integer**. In addition, p must be a prime **integer**.

3.7.7 Newton – solve equation using Newton's method

Newton(f: **poly_list**, initial: **complex**=1, repeat: *integer*=250)
→ *complex*

Return one of the approximated roots of $a_n x^n + \dots + a_1 x + a_0 = 0$.

If you want to obtain all roots, then use **SimMethod** instead.

†If `initial` is a real number but there is no real roots, then this function returns meaningless values.

`f` ought to be a **poly_list** of **complex**. `initial` is an initial approximation **complex** number. `repeat` is the number of steps to approximate a root.

3.7.8 SimMethod – find all roots simultaneously

SimMethod(`f`: **poly_list**, `NewtonInitial`: **complex**=1, `repeat`: *integer*=250)
→ *list*

Return the approximated roots of $a_n x^n + \cdots + a_1 x + a_0$.

†If the equation has multiple root, maybe raise some error.

`f` ought to be a **poly_list** of **complex**.
`NewtonInitial` and `repeat` will be passed to **Newton** to obtain the first approximations.

3.7.9 root_Fp – solve congruent equation modulo p

root_Fp(`f`: **poly_list**, `p`: *integer*) → *integer*

Return one of the roots of $a_n x^n + \cdots + a_1 x + a_0 \equiv 0 \pmod{p}$.

If you want to obtain all roots, then use **allroots_Fp**.

`f` ought to be a **poly_list** of **integer**. In addition, `p` must be a prime **integer**.
If there is no root at all, then nothing will be returned.

3.7.10 allroots_Fp – solve congruent equation modulo p

allroots_Fp(`f`: **poly_list**, `p`: *integer*) → *integer*

Return all roots of $a_n x^n + \cdots + a_1 x + a_0 \equiv 0 \pmod{p}$.

`f` ought to be a **poly_list** of **integer**. In addition, `p` must be a prime **integer**.
If there is no root at all, then an empty list will be returned.

Examples

```

>>> equation.e1([1, 2])
-0.5
>>> equation.e1([1j, 2])
-0.5j
>>> equation.e1_ZnZ([3, 2], 5)
1
>>> equation.e2([-3, 1, 1])
(1.3027756377319946, -2.3027756377319948)
>>> equation.e2_Fp([-3, 1, 1], 13)
[6, 6]
>>> equation.e3([1, 1, 2, 1])
[(-0.12256116687665397-0.74486176661974479j),
(-1.7548776662466921+1.8041124150158794e-16j),
(-0.12256116687665375+0.74486176661974468j)]
>>> equation.e3_Fp([1, 1, 2, 1], 7)
[3]
>>> equation.Newton([-3, 2, 1, 1])
0.84373427789806899
>>> equation.Newton([-3, 2, 1, 1], 2)
0.84373427789806899
>>> equation.Newton([-3, 2, 1, 1], 2, 1000)
0.84373427789806899
>>> equation.SimMethod([-3, 2, 1, 1])
[(0.84373427789806887+0j),
(-0.92186713894903438+1.6449263775999723j),
(-0.92186713894903438-1.6449263775999723j)]
>>> equation.root_Fp([-3, 2, 1, 1], 7)
>>> equation.root_Fp([-3, 2, 1, 1], 11)
9L
>>> equation.allroots_Fp([-3, 2, 1, 1], 7)
[]
>>> equation.allroots_Fp([-3, 2, 1, 1], 11)
[9L]
>>> equation.allroots_Fp([-3, 2, 1, 1], 13)
[3L, 7L, 2L]

```

3.8 gcd – gcd algorithm

3.8.1 gcd – the greatest common divisor

gcd(a: *integer*, b: *integer*) → *integer*

Return the greatest common divisor of two integers **a** and **b**.

a, **b** must be int, long or **Integer**. Even if one of the arguments is negative, the result is non-negative.

3.8.2 binarygcd – binary gcd algorithm

binarygcd(a: *integer*, b: *integer*) → *integer*

Return the greatest common divisor of two integers **a** and **b** by binary gcd algorithm.

†This function is an alias of **binarygcd**

a, **b** must be int, long, or **Integer**.

3.8.3 extgcd – extended gcd algorithm

extgcd(a: *integer*, b: *integer*) → (*integer*, *integer*, *integer*)

Return the greatest common divisor d of two integers **a** and **b** and u , v such that $d = au + bv$.

a, **b** must be int, long, or **Integer**.
The returned value is a tuple (**u**, **v**, **d**).

3.8.4 lcm – the least common multiple

lcm(a: *integer*, b: *integer*) → *integer*

Return the least common multiple of two integers **a** and **b**.

†If both **a** and **b** are zero, then it raises an exception.

a, **b** must be int, long, or **Integer**.

3.8.5 gcd_of_list – gcd of many integers

gcd_of_list(integers: list) → list

Return gcd of multiple integers.

For given **integers** $[x_1, \dots, x_n]$, return a list $[d, [c_1, \dots, c_n]]$ such that $d = c_1x_1 + \dots + c_nx_n$, where d is the greatest common divisor of x_1, \dots, x_n .

integers is a list which elements are int or long
This function returns $[d, [c_1, \dots, c_n]]$, where d, c_i are an integer.

3.8.6 coprime – coprime check

coprime(a: integer, b: integer) → bool

Return True if a and b are coprime, False otherwise.

a, b are int, long, or **Integer**.

3.8.7 pairwise_coprime – coprime check of many integers

pairwise_coprime(integers: list) → bool

Return True if all integers in **integers** are pairwise coprime, False otherwise.

integers is a list which elements are int, long, or **Integer**.

Examples

```
>>> gcd.gcd(12, 18)
6
>>> gcd.gcd(12, -18)
6
>>> gcd.gcd(-12, -18)
6
>>> gcd.extgcd(12, -18)
(-1, -1, 6)
>>> gcd.extgcd(-12, -18)
(1, -1, 6)
>>> gcd.extgcd(0, -18)
(0, -1, 18)
```

```
>>> gcd.lcm(12, 18)
36
>>> gcd.lcm(12, -18)
-36
>>> gcd.gcd_of_list([60, 90, 210])
[30, [-1, 1, 0]]
```

3.9 multiplicative – multiplicative number theoretic functions

All functions of this module accept only positive integers, unless otherwise noted.

3.9.1 euler – the Euler totient function

euler(n: *integer*) → *integer*

Return the number of numbers relatively prime to *n* and smaller than *n*. In the literature, the function is referred often as φ .

3.9.2 moebius – the Möbius function

moebius(n: *integer*) → *integer*

Return:

- 1 if *n* has odd distinct prime factors,
- 1 if *n* has even distinct prime factors, or
- 0 if *n* has a squared prime factor.

In the literature, the function is referred often as μ .

3.9.3 sigma – sum of divisor powers)

sigma(m: *integer*, n: *integer*) → *integer*

Return the sum of *m*-th powers of the factors of *n*. The argument *m* can be zero, then return the number of factors. In the literature, the function is referred often as σ .

Examples

```
>>> multiplicative.euler(1)
1
>>> multiplicative.euler(2)
1
>>> multiplicative.euler(4)
2
>>> multiplicative.euler(5)
4
>>> multiplicative.moebius(1)
1
>>> multiplicative.moebius(2)
```

```
-1
>>> multiplicative.moebius(4)
0
>>> multiplicative.moebius(6)
1
>>> multiplicative.sigma(0, 1)
1
>>> multiplicative.sigma(1, 1)
1
>>> multiplicative.sigma(0, 2)
2
>>> multiplicative.sigma(1, 3)
4
>>> multiplicative.sigma(1, 4)
7
>>> multiplicative.sigma(1, 6)
12L
>>> multiplicative.sigma(2, 7)
50
```

3.10 prime – primality test , prime generation

3.10.1 trialDivision – trial division test

trialDivision(*n*: *integer*, bound: *integer/float=0*) → *True/False*

Trial division primality test for an odd natural number.

bound is a search bound of primes. If it returns 1 under the condition that bound is given and less than the square root of *n*, it only means there is no prime factor less than bound.

3.10.2 spsp – strong pseudo-prime test

spsp(*n*: *integer*, base: *integer*, s: *integer=None*, t: *integer=None*)
→ *True/False*

Strong Pseudo-Prime test on base base.

s and t are the numbers such that $n - 1 = 2^s t$ and t is odd.

3.10.3 smallSpsp – strong pseudo-prime test for small number

smallSpsp(*n*: *integer*) → *True/False*

Strong Pseudo-Prime test for integer *n* less than 10^{12} .

4 spsp tests are sufficient to determine whether an integer less than 10^{12} is prime or not.

3.10.4 miller – Miller’s primality test

miller(*n*: *integer*) → *True/False*

Miller’s primality test.

This test is valid under GRH. See [config](#).

3.10.5 millerRabin – Miller-Rabin primality test

millerRabin(*n*: *integer*, times: *integer=20*) → *True/False*

Miller’s primality test.

The difference from **miller** is that the Miller-Rabin method uses fast but probabilistic algorithm. On the other hand, **miller** employs deterministic algorithm valid under GRH.

times (default to 20) is the number of repetition. The error probability is at most $4^{-\text{times}}$.

3.10.6 lpsp – Lucas test

lpsp(*n: integer, a: integer, b: integer*) → *True/False*

Lucas Pseudo-Prime test.

Return True if *n* is a Lucas pseudo-prime of parameters *a*, *b*, i.e. with respect to $x^2 - ax + b$.

3.10.7 fpsp – Frobenius test

fpsp(*n: integer, a: integer, b: integer*) → *True/False*

Frobenius Pseudo-Prime test.

Return True if *n* is a Frobenius pseudo-prime of parameters *a*, *b*, i.e. with respect to $x^2 - ax + b$.

3.10.8 apr – Jacobi sum test

apr(*n: integer*) → *True/False*

APR (Adleman-Pomerance-Rumery) primality test or the Jacobi sum test.

Assuming *n* has no prime factors less than 32. Assuming *n* is spsp (strong pseudo-prime) for several bases.

3.10.9 primeq – primality test automatically

primeq(*n: integer*) → *True/False*

A convenient function for primality test.

It uses one of **trialDivision**, **smallSpsp** or **apr** depending on the size of *n*.

3.10.10 `prime` – n -th prime number

`prime(n: integer) → integer`

Return the n -th prime number.

3.10.11 `nextPrime` – generate next prime

`nextPrime(n: integer) → integer`

Return the smallest prime bigger than the given integer n .

3.10.12 `randPrime` – generate random prime

`randPrime(n: integer) → integer`

Return a random n -digits prime.

3.10.13 `generator` – generate primes

`generator((None)) → generator`

Generate primes from 2 to ∞ (as generator).

3.10.14 `generator_eratosthenes` – generate primes using Eratosthenes sieve

`generator_eratosthenes(n: integer) → generator`

Generate primes up to n using Eratosthenes sieve.

3.10.15 `primonial` – product of primes

`primonial(p: integer) → integer`

Return the product

$$\prod_{q \in \mathbb{P}_{\leq p}} q = 2 \cdot 3 \cdot 5 \cdots p .$$

3.10.16 properDivisors – proper divisors

properDivisors(*n*: *integer*) → *list*

Return proper divisors of *n* (all divisors of *n* excluding 1 and *n*).

It is only useful for a product of small primes. Use **proper_divisors** in a more general case.

The output is the list of all proper divisors.

3.10.17 primitive_root – primitive root

primitive_root(*p*: *integer*) → *integer*

Return a primitive root of *p*.

p must be an odd prime.

3.10.18 Lucas_chain – Lucas sequence

Lucas_chain(*n*: *integer*, *f*: *function*, *g*: *function*, *x_0*: *integer*, *x_1*: *integer*)
→ (*integer*, *integer*)

Return the value of (x_n, x_{n+1}) for the sequence $\{x_i\}$ defined as:

$$\begin{aligned} x_{2i} &= f(x_i) \\ x_{2i+1} &= g(x_i, x_{i+1}) , \end{aligned}$$

where the initial values *x_0*, *x_1*.

f is the function which can be input as 1-ary integer. *g* is the function which can be input as 2-ary integer.

Examples

```
>>> prime.primeq(131)
True
>>> prime.primeq(133)
False
>>> g = prime.generator()
>>> g.next()
2
>>> g.next()
3
>>> prime.prime(10)
29
>>> prime.nextPrime(100)
101
>>> prime.primitive_root(23)
5
```

3.11 prime_decomp – prime decomposition

3.11.1 prime_decomp – prime decomposition

prime_decomp(*p*: *Integer*, *polynomial*: *list*) → *list*

Return prime decomposition of the ideal (*p*) over the number field $\mathbf{Q}[x]/(\text{polynomial})$.

p should be a (rational) prime. *polynomial* should be a list of integers which defines a monic irreducible polynomial.

This method returns a list of (P_k, e_k, f_k) ,

where P_k is an instance of **Ideal_with_generator** expresses a prime ideal which divides (*p*), e_k is the ramification index of P_k , f_k is the residue degree of P_k .

Examples

```
>>> for fact in prime_decomp.prime_decomp(3,[1,9,0,1]):
...     print fact
...
(Ideal_with_generator([BasicAlgNumber([[3, 0, 0], 1], [1, 9, 0, 1]), BasicAlgNum
ber([[7L, 20L, 4L], 3L], [1, 9, 0, 1]])), 1, 1)
(Ideal_with_generator([BasicAlgNumber([[3, 0, 0], 1], [1, 9, 0, 1]), BasicAlgNum
ber([[10L, 20L, 4L], 3L], [1, 9, 0, 1]])), 2, 1)
```

3.12 quad – Imaginary Quadratic Field

- **Classes**
 - **ReducedQuadraticForm**
 - **ClassGroup**
- **Functions**
 - **class_formula**
 - **class_number**
 - **class_group**
 - **class_number_bsgs**
 - **class_group_bsgs**

3.12.1 ReducedQuadraticForm – Reduced Quadratic Form Class

Initialize (Constructor)

ReducedQuadraticForm(f: list, unit: list) → *ReducedQuadraticForm*

Create ReducedQuadraticForm object.

`f`, `unit` must be list of 3 integers [`a`, `b`, `c`], representing a quadratic form $ax^2 + bxy + cy^2$. `unit` represents the unit form.

Operations

operator	explanation
<code>M * N</code>	Return the composition form of <code>M</code> and <code>N</code> .
<code>M ** a</code>	Return the a -th powering of <code>M</code> .
<code>M / N</code>	Division of form.
<code>M == N</code>	Return whether <code>M</code> and <code>N</code> are equal or not.
<code>M != N</code>	Return whether <code>M</code> and <code>N</code> are unequal or not.

Methods

3.12.1.1 inverse

`inverse(self)` → *ReducedQuadraticForm*

Return the inverse of `self`.

3.12.1.2 disc

`disc(self)` → *ReducedQuadraticForm*

Return the discriminant of `self`.

3.12.2 ClassGroup – Class Group Class

Initialize (Constructor)

`ClassGroup(disc: integer, cl: integer, element: integer=None)`
→ *ClassGroup*

Create ClassGroup object.

Methods

3.12.3 class_formula

class_formula(*d*: *integer*, uprbd: *integer*) → *integer*

Return the approximation of class number h with discriminant d using class formula.

$$\text{class formula } h = \frac{\sqrt{|d|}}{\pi} \prod_p \left(1 - \left(\frac{d}{p} \right) \frac{1}{p} \right)^{-1}.$$

Input number d must be int, long or **Integer**.

3.12.4 class_number

class_number(*d*: *integer*, limit_of_d: *integer*=1000000000) → *integer*

Return the class number with the discriminant d by counting reduced forms.

d is not only fundamental discriminant.

Input number d must be int, long or **Integer**.

3.12.5 class_group

class_group(*d*: *integer*, limit_of_d: *integer*=1000000000) → *integer*

Return the class number and the class group with the discriminant d by counting reduced forms.

d is not only fundamental discriminant.

Input number d must be int, long or **Integer**.

3.12.6 class_number_bsgs

class_number_bsgs(*d*: *integer*) → *integer*

Return the class number with the discriminant d using Baby-step Giant-step algorithm.

d is not only fundamental discriminant.

Input number d must be int, long or **Integer**.

3.12.7 class_group_bsgs

```
class_group_bsgs(d: integer, cl: integer, qin: list)
    → integer
```

Return the construction of the class group of order p^{exp} with the discriminant $disc$, where $qin = [p, exp]$.

Input number d , cl must be int, long or **Integer**.

Examples

```
>>> quad.class_formula(-1200, 100000)
12
>>> quad.class_number(-1200)
12
>>> quad.class_group(-1200)
(12, [ReducedQuadraticForm(1, 0, 300), ReducedQuadraticForm(3, 0, 100),
ReducedQuadraticForm(4, 0, 75), ReducedQuadraticForm(12, 0, 25),
ReducedQuadraticForm(7, 2, 43), ReducedQuadraticForm(7, -2, 43),
ReducedQuadraticForm(16, 4, 19), ReducedQuadraticForm(16, -4, 19),
ReducedQuadraticForm(13, 10, 25), ReducedQuadraticForm(13, -10, 25),
ReducedQuadraticForm(16, 12, 21), ReducedQuadraticForm(16, -12, 21)])
>>> quad.class_number_bsgs(-1200)
12L
>>> quad.class_group_bsgs(-1200, 12, [3, 1])
([ReducedQuadraticForm(16, -12, 21)], [[3L]])
>>> quad.class_group_bsgs(-1200, 12, [2, 2])
([ReducedQuadraticForm(12, 0, 25), ReducedQuadraticForm(4, 0, 75)],
[[2L], [2L, 0]])
```

3.13 round2 – the round 2 method

- **Classes**
 - **ModuleWithDenominator**
- **Functions**
 - **round2**
 - **Dedekind**

The round 2 method is for obtaining the maximal order of a number field from an order generated by a root of a defining polynomial of the field.

This implementation of the method is based on [?](Algorithm 6.1.8) and [?](Chapter 3).

3.13.1 ModuleWithDenominator – bases of \mathbb{Z} -module with denominator.

Initialize (Constructor)

ModuleWithDenominator(basis: *list*, denominator: *integer*, ****hints:** *dict*)

→ *ModuleWithDenominator*

This class represents bases of \mathbb{Z} -module with denominator. It is not a general purpose \mathbb{Z} -module, you are warned. **basis** is a list of integer sequences.

denominator is a common denominator of all bases.

† Optionally you can supply keyword argument **dimension** if you would like to postpone the initialization of **basis**.

Operations

operator	explanation
A + B	sum of two modules
a * B	scalar multiplication
B / d	divide by an integer

Methods

3.13.1.1 `get_rationals` – get the bases as a list of rationals

`get_rationals(self)` → *list*

Return a list of lists of rational numbers, which is bases divided by denominator.

3.13.1.2 `get_polynomials` – get the bases as a list of polynomials

`get_polynomials(self)` → *list*

Return a list of rational polynomials, which is made from bases divided by denominator.

3.13.1.3 `determinant` – determinant of the bases

`determinant(self)` → *list*

Return determinant of the bases (bases ought to be of full rank and in Hermite normal form).

3.13.2 round2(function)

round2(minpoly_coeff: *list*) → (*list*, *integer*)

Return integral basis of the ring of integers of a field with its discriminant. The field is given by a list of integers, which is a polynomial of generating element θ . The polynomial ought to be monic, in other word, the generating element ought to be an algebraic integer.

The integral basis will be given as a list of rational vectors with respect to θ .

3.13.3 Dedekind(function)

Dedekind(minpoly_coeff: *list*, p: *integer*, e: *integer*)
→ (*bool*, *ModuleWithDenominator*)

This is the Dedekind criterion.

minpoly_coeff is an integer list of the minimal polynomial of θ .

p**e divides the discriminant of the minimal.

The first element of the returned tuple is whether the computation about p is finished or not.

3.14 squarefree – Squarefreeness tests

There are two method groups. A function in one group raises **Undetermined** when it cannot determine squarefreeness. A function in another group returns **None** in such cases. The latter group of functions have “_ternary” suffix on their names. We refer a set $\{\text{True}, \text{False}, \text{None}\}$ as *ternary*.

The parameter type *integer* means either *int*, *long* or **Integer**.

This module provides an exception class.

Undetermined : Report undetermined state of calculation. The exception will be raised by **lenstra** or **trivial_test**.

3.14.1 Definition

We define squarefreeness as:

n is squarefree \iff there is no prime p whose square divides n .

Examples:

- 0 is non-squarefree because any square of prime can divide 0.
- 1 is squarefree because there is no prime dividing 1.
- 2, 3, 5, and any other primes are squarefree.
- 4, 8, 9, 12, 16 are non-squarefree composites.
- 6, 10, 14, 15, 21 are squarefree composites.

3.14.2 lenstra – Lenstra’s condition

lenstra(n : *integer*) \rightarrow *bool*

If return value is True, n is squarefree. Otherwise, the squarefreeness is still unknown and **Undetermined** is raised. The algorithm is based on [?].

†The condition is so strong that it seems n has to be a prime or a Carmichael number to satisfy it.

Input parameter n ought to be an odd **integer**.

3.14.3 trial_division – trial division

trial_division(n : *integer*) \rightarrow *bool*

Check whether n is squarefree or not.

The method is a kind of trial division and inefficient for large numbers.

Input parameter `n` ought to be an **integer**.

3.14.4 `trivial_test` – trivial tests

`trivial_test(n: integer) → bool`

Check whether `n` is squarefree or not. If the squarefreeness is still unknown, then **Undetermined** is raised.

This method do anything but factorization including Lenstra's method.

Input parameter `n` ought to be an odd **integer**.

3.14.5 `viafactor` – via factorization

`viafactor(n: integer) → bool`

Check whether `n` is squarefree or not.

It is obvious that if one knows the prime factorization of the number, he/she can tell whether the number is squarefree or not.

Input parameter `n` ought to be an **integer**.

3.14.6 `viadecomposition` – via partial factorization

`viadecomposition(n: integer) → bool`

Test the squarefreeness of `n`. The return value is either one of **True** or **False**; **None** never be returned.

The method uses partial factorization into squarefree parts, if such partial factorization is possible. In other cases, It completely factor `n` by trial division.

Input parameter `n` ought to be an **integer**.

3.14.7 `lenstra_ternary` – Lenstra's condition, ternary version

`lenstra_ternary(n: integer) → ternary`

Test the squarefreeness of `n`. The return value is one of the ternary logical constants. If return value is **True**, `n` is squarefree. Otherwise, the squarefreeness is still unknown and **None** is returned.

†The condition is so strong that it seems n has to be a prime or a Carmichael number to satisfy it.

This is a ternary version of **lenstra**.

Input parameter n ought to be an odd **integer**.

3.14.8 **trivial_test_ternary** – trivial tests, ternary version

trivial_test_ternary(n : *integer*) \rightarrow *ternary*

Test the squarefreeness of n . The return value is one of the ternary logical constants.

The method uses a series of trivial tests including **lenstra_ternary**.

This is a ternary version of **trivial_test**.

Input parameter n ought to be an **integer**.

3.14.9 **trial_division_ternary** – trial division, ternary version

trial_division_ternary(n : *integer*) \rightarrow *ternary*

Test the squarefreeness of n . The return value is either one of **True** or **False**; **None** never be returned.

The method is a kind of trial division.

This is a ternary version of **trial_division**.

Input parameter n ought to be an **integer**.

3.14.10 **viafactor_ternary** – via factorization, ternary version

viafactor_ternary(n : *integer*) \rightarrow *ternary*

Just for symmetry, this function is defined as an alias of **viafactor**.

Input parameter n ought to be an **integer**.

Chapter 4

Classes

4.1 `algfield` – Algebraic Number Field

- **Classes**

- `NumberField`
- `BasicAlgNumber`
- `MatAlgNumber`

- **Functions**

- `changetype`
- `disc`
- `fppoly`
- `qpoly`
- `zpoly`

4.1.1 `NumberField` – number field

Initialize (Constructor)

```
NumberField( f: list, precompute: bool=False ) → NumberField
```

Create `NumberField` object.

This field defined by the polynomial `f`.
The class inherits `Field`.

`f`, which expresses coefficients of a polynomial, must be a list of integers. `f` should be written in ascending order. `f` must be monic irreducible over rational

field.

If `precompute` is True, all solutions of `f` (by `getConj`), the discriminant of `f` (by `disc`), the signature (by `signature`) and the field discriminant of the basis of the integer ring (by `integer_ring`) are precomputed.

Attribute

degree : The (absolute) extension degree of the number field.

polynomial : The defining polynomial of the number field.

Operations

operator	explanation
<code>K * F</code>	Return the composite field of K and F.
<code>K == F</code>	Check whether the equality of K and F.

Examples

```
>>> K = algfield.NumberField([-2, 0, 1])
>>> L = algfield.NumberField([-3, 0, 1])
>>> print K, L
NumberField([-2, 0, 1]) NumberField([-3, 0, 1])
>>> print K * L
NumberField([1L, 0L, -10L, 0L, 1L])
```

Methods

4.1.1.1 `getConj` – roots of polynomial

`getConj(self)` → *list*

Return all (approximate) roots of the `self.polynomial`.

The output is a list of (approximate) complex number.

4.1.1.2 `disc` – polynomial discriminant

`disc(self)` → *integer*

Return the (polynomial) discriminant of the `self.polynomial`.

†The output is not discriminant of the number field itself.

4.1.1.3 `integer_ring` – integer ring

`integer_ring(self)` → **FieldSquareMatrix**

Return a basis of the ring of integers of `self`.

†The function uses **round2**.

4.1.1.4 `field_discriminant` – discriminant

`field_discriminant(self)` → **Rational**

Return the field discriminant of `self`.

†The function uses **round2**.

4.1.1.5 `basis` – standard basis

`basis(self, j: integer)` → **BasicAlgNumber**

Return the j -th basis (over the rational field) of `self`.

Let θ be a solution of `self.polynomial`. Then θ^j is a part of basis of `self`, so

the method returns them. This basis is called “standard basis” or “power basis”.

4.1.1.6 **signature** – signature

signature(**self**) → *list*

Return the signature of **self**.

†The method uses Strum’s algorithm.

4.1.1.7 **POLRED** – polynomial reduction

POLRED(**self**) → *list*

Return some polynomials defining subfields of **self**.

†“POLRED” means “polynomial reduction”. That is, it finds polynomials whose coefficients are not so large.

4.1.1.8 **isIntBasis** – check integral basis

isIntBasis(**self**) → *bool*

Check whether power basis of **self** is also an integral basis of the field.

4.1.1.9 **isGaloisField** – check Galois field

isGaloisField(**self**) → *bool*

Check whether the extension **self** over the rational field is Galois.

†As it stands, it only checks the signature.

4.1.1.10 **isFieldElement** – check field element

isFieldElement(**self**, **A**: *BasicAlgNumber/MatAlgNumber*)
→ *bool*

Check whether **A** is an element of the field **self**.

4.1.1.11 `getCharacteristic` – characteristic

`getCharacteristic(self) → integer`

Return the characteristic of `self`.

It returns always zero. The method is only for ensuring consistency.

4.1.1.12 `createElement` – create an element

`createElement(self, seed: list) → BasicAlgNumber/MatAlgNumber`

Return an element of `self` with `seed`.

`seed` determines the class of returned element.

For example, if `seed` forms as $[[e_1, e_2, \dots, e_n], d]$, then it calls **BasicAlgNumber**.

Examples

```
>>> K = algfield.NumberField([3, 0, 1])
>>> K.getConj()
[-1.7320508075688774j, 1.7320508075688772j]
>>> K.disc()
-12L
>>> print K.integer_ring()
1/1 1/2
0/1 1/2
>>> K.field_discriminant()
Rational(-3, 1)
>>> K.basis(0), K.basis(1)
BasicAlgNumber([[1, 0], 1], [3, 0, 1]) BasicAlgNumber([[0, 1], 1], [3, 0, 1])
>>> K.signature()
(0, 1)
>>> K.POLRED()
[IntegerPolynomial([(0, 4L), (1, -2L), (2, 1L)], IntegerRing()),
IntegerPolynomial([(0, -1L), (1, 1L)], IntegerRing())]
>>> K.isIntBasis()
False
```

4.1.2 BasicAlgNumber – Algebraic Number Class by standard basis

Initialize (Constructor)

```
BasicAlgNumber( valuelist: list, polynomial: list, precompute:
bool=False )
    → BasicAlgNumber
```

Create an algebraic number with standard (power) basis.

`valuelist` = $[[e_1, e_2, \dots, e_n], d]$ means $\frac{1}{d}(e_1 + e_2\theta + e_3\theta^2 + \dots + e_n\theta^{n-1})$, where θ is a solution of the polynomial `polynomial`. Note that $\langle \theta^i \rangle$ is a (standard) basis of the field defining by `polynomial` over the rational field.

e_i, d must be integers. Also, `polynomial` should be list of integers. If `precompute` is True, all solutions of `polynomial` (by `getConj`), approximation values of all conjugates of `self` (by `getApprox`) and a polynomial which is a solution of `self` (by `getCharPoly`) are precomputed.

Attribute

value : The list of numerators (the integer part) and the denominator of `self`.

coeff : The coefficients of numerators (the integer part) of `self`.

denom : The denominator of the algebraic number for standard basis.

degree : The degree of extension of the field over the rational field.

polynomial : The defining polynomial of the field.

field : The number field in which `self` is.

Operations

operator	explanation
<code>a + b</code>	Return the sum of <code>a</code> and <code>b</code> .
<code>a - b</code>	Return the subtraction of <code>a</code> and <code>b</code> .
<code>- a</code>	Return the negation of <code>a</code> .
<code>a * b</code>	Return the product of <code>a</code> and <code>b</code> .
<code>a ** k</code>	Return the k-th power of <code>a</code> .
<code>a / b</code>	Return the quotient of <code>a</code> by <code>b</code> .

Examples

```
>>> a = algfield.BasicAlgNumber([[1, 1], 1], [-2, 0, 1])
>>> b = algfield.BasicAlgNumber([[-1, 2], 1], [-2, 0, 1])
>>> print a + b
BasicAlgNumber([[0, 3], 1], [-2, 0, 1])
>>> print a * b
BasicAlgNumber([[3L, 1L], 1], [-2, 0, 1])
>>> print a ** 3
BasicAlgNumber([[7L, 5L], 1], [-2, 0, 1])
>>> a // b
BasicAlgNumber([[5L, 3L], 7L], [-2, 0, 1])
```

Methods

4.1.2.1 `inverse` – `inverse`

`inverse(self) → BasicAlgNumber`

Return the inverse of `self`.

4.1.2.2 `getConj` – roots of polynomial

`getConj(self) → list`

Return all (approximate) roots of `self.polynomial`.

4.1.2.3 `getApprox` – approximate conjugates

`getApprox(self) → list`

Return all (approximate) conjugates of `self`.

4.1.2.4 `getCharPoly` – characteristic polynomial

`getCharPoly(self) → list`

Return the characteristic polynomial of `self`.

†`self` is a solution of the characteristic polynomial.

The output is a list of integers.

4.1.2.5 `getRing` – the field

`getRing(self) → NumberField`

Return the field which `self` belongs to.

4.1.2.6 `trace` – `trace`

`trace(self) → Rational`

Return the trace of `self` in the `self.field` over the rational field.

4.1.2.7 `norm` – `norm`

`norm(self) → Rational`

Return the norm of `self` in the `self.field` over the rational field.

4.1.2.8 `isAlgInteger` – check (algebraic) integer

`isAlgInteger(self) → bool`

Check whether `self` is an (algebraic) integer or not.

4.1.2.9 `ch_matrix` – obtain `MatAlgNumber` object

`ch_matrix(self) → MatAlgNumber`

Return `MatAlgNumber` object corresponding to `self`.

Examples

```
>>> a = algfield.BasicAlgNumber([[1, 1], 1], [-2, 0, 1])
>>> a.inverse()
BasicAlgNumber([[-1L, 1L], 1L], [-2, 0, 1])
>>> a.getConj()
[(1.4142135623730951+0j), (-1.4142135623730951+0j)]
>>> a.getApprox()
[(2.4142135623730949+0j), (-0.41421356237309515+0j)]
>>> a.getCharPoly()
[-1, -2, 1]
>>> a.getRing()
NumberField([-2, 0, 1])
>>> a.trace(), a.norm()
2 -1
>>> a.isAlgInteger()
True
>>> a.ch_matrix()
MatAlgNumber([1, 1]+[2, 1], [-2, 0, 1])
```

4.1.3 MatAlgNumber – Algebraic Number Class by matrix representation

Initialize (Constructor)

```
MatAlgNumber( coefficient: list, polynomial: list )
    → MatAlgNumber
```

Create an algebraic number represented by a matrix.

“matrix representation” means the matrix A over the rational field such that $(e_1 + e_2\theta + e_3\theta^2 + \dots + e_n\theta^{n-1})(1, \theta, \dots, \theta^{n-1})^T = A(1, \theta, \dots, \theta^{n-1})^T$, where t expresses transpose operation.

coefficient = $[e_1, e_2, \dots, e_n]$ means $e_1 + e_2\theta + e_3\theta^2 + \dots + e_n\theta^{n-1}$, where θ is a solution of the polynomial **polynomial**. Note that $\langle \theta^i \rangle$ is a (standard) basis of the field defining by **polynomial** over the rational field. **coefficient** must be a list of (not only integers) rational numbers. **polynomial** must be a list of integers.

Attribute

coeff : The coefficients of the algebraic number for standard basis.

degree : The degree of extension of the field over the rational field.

matrix : The representation matrix of the algebraic number.

polynomial : The defining polynomial of the field.

field : The number field in which **self** is.

Operations

operator	explanation
a + b	Return the sum of a and b .
a - b	Return the subtraction of a and b .
- a	Return the negation of a .
a * b	Return the product of a and b .
a ** k	Return the k-th power of a .
a / b	Return the quotient of a by b .

Examples

```
>>> a = algfield.MatAlgNumber([1, 2], [-2, 0, 1])
>>> b = algfield.MatAlgNumber([-2, 3], [-2, 0, 1])
>>> print a + b
MatAlgNumber([-1, 5]+[10, -1], [-2, 0, 1])
>>> print a * b
MatAlgNumber([10, -1]+[-2, 10], [-2, 0, 1])
>>> print a ** 3
MatAlgNumber([25L, 22L]+[44L, 25L], [-2, 0, 1])
>>> print a / b
MatAlgNumber([Rational(1, 1), Rational(1, 2)]+
[Rational(1, 1), Rational(1, 1)], [-2, 0, 1])
```


Methods

4.1.3.1 `inverse` – `inverse`

`inverse(self) → MatAlgNumber`

Return the inverse of `self`.

4.1.3.2 `getRing` – the field

`getRing(self) → NumberField`

Return the field which `self` belongs to.

4.1.3.3 `trace` – `trace`

`trace(self) → Rational`

Return the trace of `self` in the `self.field` over the rational field.

4.1.3.4 `norm` – `norm`

`norm(self) → Rational`

Return the norm of `self` in the `self.field` over the rational field.

4.1.3.5 `ch_basic` – obtain `BasicAlgNumber` object

`ch_basic(self) → BasicAlgNumber`

Return `BasicAlgNumber` object corresponding to `self`.

Examples

```
>>> a = algfield.MatAlgNumber([1, -1, 1], [-3, 1, 2, 1])
>>> a.inverse()
MatAlgNumber([Rational(2, 3), Rational(4, 9), Rational(1, 9)]+
[Rational(1, 3), Rational(5, 9), Rational(2, 9)]+
[Rational(2, 3), Rational(1, 9), Rational(1, 9)], [-3, 1, 2, 1])
>>> a.trace()
Rational(7, 1)
```

```
>>> a.norm()
Rational(27, 1)
>>> a.getRing()
NumberField([-3, 1, 2, 1])
>>> a.ch_basic()
BasicAlgNumber([[1, -1, 1], 1], [-3, 1, 2, 1])
```

4.1.4 `changetype(function)` – obtain `BasicAlgNumber` object

`changetype(a: integer, polynomial: list=[0, 1]) → BasicAlgNumber`

`changetype(a: Rational, polynomial: list=[0, 1]) → BasicAlgNumber`

`changetype(polynomial: list) → BasicAlgNumber`

Return a `BasicAlgNumber` object corresponding to `a`.

If `a` is an integer or an instance of `Rational`, the function returns `BasicAlgNumber` object whose field is defined by `polynomial`. If `a` is a list, the function returns `BasicAlgNumber` corresponding to a solution of `a`, considering `a` as the polynomial.

The input parameter `a` must be an integer, `Rational` or a list of integers.

4.1.5 `disc(function)` – discriminant

`disc(A: list) → Rational`

Return the discriminant of a_i , where $A = [a_1, a_2, \dots, a_n]$.

a_i must be an instance of `BasicAlgNumber` or `MatAlgNumber` defined over a same number field.

4.1.6 `fppoly(function)` – polynomial over finite prime field

`fppoly(coeffs: list, p: integer) → FinitePrimeFieldPolynomial`

Return the polynomial whose coefficients `coeffs` are defined over the prime field \mathbb{Z}_p .

`coeffs` should be a list of integers or of instances of `FinitePrimeFieldElement`.

4.1.7 `qpoly(function)` – polynomial over rational field

`qpoly(coeffs: list) → FieldPolynomial`

Return the polynomial whose coefficients `coeffs` are defined over the rational

field.

`coeffs` must be a list of integers or instances of **Rational**.

4.1.8 `zpoly(function)` – polynomial over integer ring

`zpoly(coeffs: list) → IntegerPolynomial`

Return the polynomial whose coefficients `coeffs` are defined over the (rational) integer ring.

`coeffs` must be a list of integers.

Examples

```
>>> a = algfield.changetype(3, [-2, 0, 1])
>>> b = algfield.BasicAlgNumber([[1, 2], 1], [-2, 0, 1])
>>> A = [a, b]
>>> algfield.disc(A)
288L
```

4.2 elliptic – elliptic class object

- **Classes**
 - **ECGeneric**
 - **ECoverQ**
 - **ECoverGF**
- **Functions**
 - **EC**

This module using following type:

weierstrassform :

weierstrassform is a list $(a_1, a_2, a_3, a_4, a_6)$ or (a_4, a_6) , it represents E : $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ or $E : y^2 = x^3 + a_4x + a_6$, respectively.

infpoint :

infpoint is the list $[0]$, which represents infinite point on the elliptic curve.

point :

point is two-dimensional coordinate list $[x, y]$ or **infpoint**.

4.2.1 †ECGeneric – generic elliptic curve class

Initialize (Constructor)

```
ECGeneric( coefficient: weierstrassform, basefield: Field=None )  
→ ECGeneric
```

Create an elliptic curve object.

The class is for the definition of elliptic curves over general fields. Instead of using this class directly, we recommend that you call **EC**.

†The class precomputes the following values.

- shorter form: $y^2 = b_2x^3 + b_4x^2 + b_6x + b_8$
- shortest form: $y^2 = x^3 + c_4x + c_6$
- discriminant
- j-invariant

All elements of **coefficient** must be in **basefield**.

See **weierstrassform** for more information about **coefficient**. If discriminant of **self** equals 0, it raises `ValueError`.

Attribute

basefield :

It expresses the field which each coordinate of all points in **self** is on.
(This means not only **self** is defined over **basefield**.)

ch :

It expresses the characteristic of **basefield**.

infpoint :

It expresses infinity point (i.e. $[0]$).

a1, a2, a3, a4, a6 :

It expresses the coefficients **a1**, **a2**, **a3**, **a4**, **a6**.

b2, b4, b6, b8 :

It expresses the coefficients **b2**, **b4**, **b6**, **b8**.

c4, c6 :

It expresses the coefficients **c4**, **c6**.

disc :

It expresses the discriminant of **self**.

j :
It expresses the j-invariant of **self**.

coefficient :
It expresses the **weierstrassform** of **self**.

Methods

4.2.1.1 `simple` – simplify the curve coefficient

`simple(self) → ECGeneric`

Return elliptic curve corresponding to the short Weierstrass form of `self` by changing the coordinates.

4.2.1.2 `changeCurve` – change the curve by coordinate change

`changeCurve(self, V: list) → ECGeneric`

Return elliptic curve corresponding to the curve obtained by some coordinate change $x = u^2x' + r$, $y = u^3y' + su^2x' + t$.

For $u \neq 0$, the coordinate change gives some curve which is **basefield**-isomorphic to `self`.

V must be a list of the form $[u, r, s, t]$, where u, r, s, t are in **basefield**.

4.2.1.3 `changePoint` – change coordinate of point on the curve

`changePoint(self, P: point, V: list) → point`

Return the point corresponding to the point obtained by the coordinate change $x' = (x - r)u^{-2}$, $y' = (y - s(x - r) + t)u^{-3}$.

Note that the inverse coordinate change is $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. See **`changeCurve`**.

V must be a list of the form $[u, r, s, t]$, where u, r, s, t are in **basefield**. u must be non-zero.

4.2.1.4 `coordinateY` – Y-coordinate from X-coordinate

`coordinateY(self, x: FieldElement) → FieldElement / False`

Return Y-coordinate of the point on `self` whose X-coordinate is `x`.

The output would be one Y-coordinate (if a coordinate is found). If such a Y-coordinate does not exist, it returns False.

4.2.1.5 whetherOn – Check point is on curve

```
whetherOn(self, P: point) → bool
```

Check whether the point P is on self or not.

4.2.1.6 add – Point addition on the curve

```
add(self, P: point, Q: point) → point
```

Return the sum of the point P and Q on self.

4.2.1.7 sub – Point subtraction on the curve

```
sub(self, P: point, Q: point) → point
```

Return the subtraction of the point P from Q on self.

4.2.1.8 mul – Scalar point multiplication on the curve

```
mul(self, k: integer, P: point) → point
```

Return the scalar multiplication of the point P by a scalar k on self.

4.2.1.9 divPoly – division polynomial

```
divPoly(self, m: integer=None) → FieldPolynomial/(f: list, H: integer)
```

Return the division polynomial.

If m is odd, this method returns the usual division polynomial. If m is even, return the quotient of the usual division polynomial by $2y + a_1x + a_3$.

†If m is not specified (i.e. m=None), then return (f, H). H is the least prime satisfying $\prod_{2 \leq l \leq H, l: \text{prime}} l > 4\sqrt{q}$, where q is the order of **basefield**. f is the list of k-division polynomials up to $k \leq H$. These are used for Schoof's algorithm.

4.2.2 ECoverQ – elliptic curve over rational field

The class is for elliptic curves over the rational field \mathbb{Q} (**RationalField** in `nzmath.rational`).

The class is a subclass of **ECGeneric**.

Initialize (Constructor)

ECoverQ(coefficient: **weierstrassform**) \rightarrow **ECoverQ**

Create elliptic curve over the rational field.

All elements of `coefficient` must be integer or **Rational**.
See **weierstrassform** for more information about `coefficient`.

Examples

```
>>> E = elliptic.ECoverQ([rational.Rational(1, 2), 3])
>>> print E.disc
-3896/1
>>> print E.j
1728/487
```

Methods

4.2.2.1 `point` – obtain random point on curve

`point(self, limit: integer=1000) → point`

Return a random point on `self`.

`limit` expresses the time of trying to choose points. If failed, raise `ValueError`.
†Because it is difficult to search the rational point over the rational field, it might raise error with high frequency.

Examples

```
>>> print E.changeCurve([1, 2, 3, 4])
y ** 2 + 6/1 * x * y + 8/1 * y = x ** 3 - 3/1 * x ** 2 - 23/2 * x - 4/1
>>> E.divPoly(3)
FieldPolynomial([(0, Rational(-1, 4)), (1, Rational(36, 1)), (2, Rational(3, 1)), (4, Rational(3, 1))], RationalField())
```

4.2.3 ECoverGF – elliptic curve over finite field

The class is for elliptic curves over a finite field, denoted by \mathbb{F}_q (**FiniteField** and its subclasses in `nzmath`).

The class is a subclass of **ECGeneric**.

Initialize (Constructor)

```
ECoverGF( coefficient: weierstrassform, basefield: FiniteField )  
→ ECoverGF
```

Create elliptic curve over a finite field.

All elements of `coefficient` must be in `basefield`. `basefield` should be an instance of **FiniteField**.

See **weierstrassform** for more information about `coefficient`.

Examples

```
>>> E = elliptic.ECoverGF([2, 5], finitefield.FinitePrimeField(11))  
>>> print E.j  
7 in F_11  
>>> E.whetherOn([8, 4])  
True  
>>> E.add([3, 4], [9, 9])  
[FinitePrimeFieldElement(0, 11), FinitePrimeFieldElement(4, 11)]  
>>> E.mul(5, [9, 9])  
[FinitePrimeFieldElement(0, 11)]
```

Methods

4.2.3.1 point – find random point on curve

point(self) → **point**

Return a random point on **self**.

This method uses a probabilistic algorithm.

4.2.3.2 naive – Frobenius trace by naive method

naive(self) → *integer*

Return Frobenius trace t by a naive method.

†The function counts up the Legendre symbols of all rational points on **self**. Frobenius trace of the curve is t such that $\#E(\mathbb{F}_q) = q + 1 - t$, where $\#E(\mathbb{F}_q)$ stands for the number of points on **self** over **self.basefield** \mathbb{F}_q .

The characteristic of **self.basefield** must not be 2 nor 3.

4.2.3.3 Shanks_Mestre – Frobenius trace by Shanks and Mestre method

Shanks_Mestre(self) → *integer*

Return Frobenius trace t by Shanks and Mestre method.

†This uses the method proposed by Shanks and Mestre. †See Algorithm 7.5.3 of [?] for more information about the algorithm. Frobenius trace of the curve is t such that $\#E(\mathbb{F}_q) = q + 1 - t$, where $\#E(\mathbb{F}_q)$ stands for the number of points on **self** over **self.basefield** \mathbb{F}_q .

self.basefield must be an instance of **FinitePrimeField**.

4.2.3.4 Schoof – Frobenius trace by Schoof’s method

Schoof(self) → *integer*

Return Frobenius trace t by Schoof’s method.

†This uses the method proposed by Schoof.

Frobenius trace of the curve is t such that $\#E(\mathbb{F}_q) = q + 1 - t$, where $\#E(\mathbb{F}_q)$ stands for the number of points on `self` over `self.basefield` \mathbb{F}_q .

4.2.3.5 `trace` – Frobenius trace

`trace(self, r: integer=None) → integer`

Return Frobenius trace t .

Frobenius trace of the curve is t such that $\#E(\mathbb{F}_q) = q + 1 - t$, where $\#E(\mathbb{F}_q)$ stands for the number of points on `self` over `self.basefield` \mathbb{F}_q .

If positive r given, it returns $q^r + 1 - \#E(\mathbb{F}_{q^r})$.

†The method selects algorithms by investigating `self.ch` when `self.basefield` is an instance of `FinitePrimeField`. If `ch` < 1000, the method uses `naive`. If $10^4 < \text{ch} < 10^{30}$, the method uses `Shanks_Mestre`. Otherwise, it uses `Schoof`.

The parameter r must be positive integer.

4.2.3.6 `order` – order of group of rational points on the curve

`order(self, r: integer=None) → integer`

Return order $\#E(\mathbb{F}_q) = q + 1 - t$.

If positive r given, this computes $\#E(\mathbb{F}_{q^r})$ instead.

†On the computation of Frobenius trace t , the method calls `trace`.

The parameter r must be positive integer.

4.2.3.7 `pointorder` – order of point on the curve

`pointorder(self, P: point, ord_factor: list=None)
→ integer`

Return order of a point P .

†The method uses factorization of `order`.

If `ord_factor` is given, computation of factorizing the order of `self` is omitted and it applies `ord_factor` instead.

4.2.3.8 TatePairing – Tate Pairing

TatePairing(self, m: *integer*, P: **point**, Q: **point**) → **FiniteFieldElement**

Return Tate-Lichtenbaum pairing $\langle P, Q \rangle_m$.

†The method uses Miller’s algorithm.
The image of the Tate pairing is $\mathbb{F}_q^*/\mathbb{F}_q^{*m}$, but the method returns an element of \mathbb{F}_q , so the value is not uniquely defined. If uniqueness is needed, use **TatePairing_Extend**.

The point P has to be a m-torsion point (i.e. $mP = [0]$). Also, the number m must divide **order**.

4.2.3.9 TatePairing_Extend – Tate Pairing with final exponentiation

TatePairing_Extend(self, m: *integer*, P: **point**, Q: **point**)
→ **FiniteFieldElement**

Return Tate Pairing with final exponentiation, i.e. $\langle P, Q \rangle_m^{(q-1)/m}$.

†The method calls **TatePairing**.

The point P has to be a m-torsion point (i.e. $mP = [0]$). Also the number m must divide **order**.
The output is in the group generated by m -th root of unity in \mathbb{F}_q^* .

4.2.3.10 WeilPairing – Weil Pairing

WeilPairing(self, m: *integer*, P: **point**, Q: **point**) → **FiniteFieldElement**

Return Weil pairing $e_m(P, Q)$.

†The method uses Miller’s algorithm.

The points P and Q has to be a m-torsion point (i.e. $mP = mQ = [0]$). Also, the number m must divide **order**.

The output is in the group generated by m -th root of unity in \mathbb{F}_q^* .

4.2.3.11 BSGS – point order by Baby-Step and Giant-Step

BSGS(self, P: **point**) → *integer*

Return order of point P by Baby-Step and Giant-Step method.

†See [?] for more information about the algorithm.

4.2.3.12 DLP_BSGS – solve Discrete Logarithm Problem by Baby-Step and Giant-Step

DLP_BSGS(self, n: *integer*, P: **point**, Q: **point**) → m: *integer*

Return m such that $Q = mP$ by Baby-Step and Giant-Step method.

The points P and Q has to be a n-torsion point (i.e. $nP = nQ = [0]$). Also, the number n must divide **order**. The output m is an integer.

4.2.3.13 structure – structure of group of rational points

structure(self) → **structure**: *tuple*

Return the group structure of **self**.

The structure of $E(\mathbb{F}_q)$ is represented as $\mathbb{Z}/d\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$. The method uses **WeilPairing**.

The output **structure** is a tuple of positive two integers (d, n). d divides n.

4.2.3.14 issupersingular – check supersingular curve

structure(self) → *bool*

Check whether **self** is a supersingular curve or not.

Examples

```
>>> E=nzmath.elliptic.ECoverGF([2, 5], nzmath.finitefield.FinitePrimeField(11))
>>> E.whetherOn([0, 4])
True
>>> print E.coordinateY(3)
4 in F_11
>>> E.trace()
2
>>> E.order()
```



```
10
>>> E.pointorder([3, 4])
10L
>>> E.TatePairing(10, [3, 4], [9, 9])
FinitePrimeFieldElement(3, 11)
>>> E.DLP_BSGS(10, [3, 4], [9, 9])
6
```

4.2.4 EC(function)

```
EC(coefficient: weierstrassform, basefield: Field)  
    → ECGeneric
```

Create an elliptic curve object.

All elements of `coefficient` must be in `basefield`.
`basefield` must be **RationalField** or **FiniteField** or their subclasses. See
also **weierstrassform** for `coefficient`.

4.3 finitefield – Finite Field

- Classes
 - †FiniteField
 - †FiniteFieldElement
 - FinitePrimeField
 - FinitePrimeFieldElement
 - ExtendedField
 - ExtendedFieldElement

4.3.1 †FiniteField – finite field, abstract

Abstract class for finite fields. Do not use the class directly, but use the subclasses **FinitePrimeField** or **ExtendedField**.

The class is a subclass of **Field**.

4.3.2 †FiniteFieldElement – element in finite field, abstract

Abstract class for finite field elements. Do not use the class directly, but use the subclasses **FinitePrimeFieldElement** or **ExtendedFieldElement**.

The class is a subclass of **FieldElement**.

4.3.3 FinitePrimeField – finite prime field

Finite prime field is also known as \mathbb{F}_p or $\text{GF}(p)$. It has prime number cardinality.

The class is a subclass of **FiniteField**.

Initialize (Constructor)

FinitePrimeField(characteristic: *integer*) \rightarrow *FinitePrimeField*

Create a FinitePrimeField instance with the given `characteristic`. `characteristic` must be positive prime integer.

Attribute

zero :

It expresses the additive unit 0. (read only)

one :

It expresses the multiplicative unit 1. (read only)

Operations

operator	explanation
<code>F==G</code>	equality test.
<code>x in F</code>	membership test.
<code>card(F)</code>	Cardinality of the field.

Methods

4.3.3.1 createElement – create element of finite prime field

createElement(self, seed: *integer*) → *FinitePrimeFieldElement*

Create **FinitePrimeFieldElement** with *seed*.
seed must be int or long.

4.3.3.2 getCharacteristic – get characteristic

getCharacteristic(self) → *integer*

Return the characteristic of the field.

4.3.3.3 issubring – subring test

issubring(self, other: **Ring**) → *bool*

Report whether another ring contains the field as subring.

4.3.3.4 issuperring – superring test

issuperring(self, other: **Ring**) → *bool*

Report whether the field is a superring of another ring.
Since the field is a prime field, it can be a superring of itself only.

4.3.4 FinitePrimeFieldElement – element of finite prime field

The class provides elements of finite prime fields.

It is a subclass of **FiniteFieldElement** and **IntegerResidueClass**.

Initialize (Constructor)

FinitePrimeFieldElement(representative: *integer*, modulus: *integer*)
→ *FinitePrimeFieldElement*

Create element in finite prime field of modulus with residue representative.
modulus must be positive prime integer.

Operations

operator	explanation
a+b	addition.
a-b	subtraction.
a*b	multiplication.
a**n, pow(a,n)	power.
-a	negation.
+a	make a copy.
a==b	equality test.
a!=b	inequality test.
repr(a)	return representation string.
str(a)	return string.

Methods

4.3.4.1 `getRing` – get ring object

`getRing(self)` \rightarrow *FinitePrimeField*

Return an instance of `FinitePrimeField` to which the element belongs.

4.3.4.2 `order` – order of multiplicative group

`order(self)` \rightarrow *integer*

Find and return the order of the element in the multiplicative group of \mathbb{F}_p .

4.3.5 ExtendedField – extended field of finite field

ExtendedField is a class for finite field, whose cardinality $q = p^n$ with a prime p and $n > 1$. It is usually called \mathbb{F}_q or $\text{GF}(q)$.

The class is a subclass of **FiniteField**.

Initialize (Constructor)

ExtendedField(basefield: *FiniteField*, modulus: *FiniteFieldPolynomial*)
→ *ExtendedField*

Create a field extension **basefield**[X]/(**modulus**(X)).

FinitePrimeField instance with the given **characteristic**. The **modulus** has to be an irreducible polynomial with coefficients in the **basefield**.

Attribute

zero :

It expresses the additive unit 0. (read only)

one :

It expresses the multiplicative unit 1. (read only)

Operations

operator	explanation
F==G	equality or not.
x in F	membership test.
card(F)	Cardinality of the field.
repr(F)	representation string.
str(F)	string.

Methods

4.3.5.1 createElement – create element of extended field

createElement(self, seed: *extended element seed*) → *ExtendedFieldElement*

Create an element of the field from seed. The result is an instance of **ExtendedFieldElement**.

The **seed** can be:

- a **FinitePrimeFieldPolynomial**
- an integer, which will be expanded in `card(basefield)` and interpreted as a polynomial.
- `basefield` element.
- a list of basefield elements interpreted as a polynomial coefficient.

4.3.5.2 getCharacteristic – get characteristic

getCharacteristic(self) → *integer*

Return the characteristic of the field.

4.3.5.3 issubring – subring test

issubring(self, other: **Ring**) → *bool*

Report whether another ring contains the field as subring.

4.3.5.4 issuperring – superring test

issuperring(self, other: **Ring**) → *bool*

Report whether the field is a superring of another ring.

4.3.5.5 primitive_element – generator of multiplicative group

primitive_element(self) → *ExtendedFieldElement*

Return a primitive element of the field, i.e., a generator of the multiplicative group.

4.3.6 ExtendedFieldElement – element of finite field

ExtendedFieldElement is a class for an element of F_q .

The class is a subclass of **FiniteFieldElement**.

Initialize (Constructor)

ExtendedFieldElement(representative: *FiniteFieldPolynomial*,
field: *ExtendedField*)
→ *ExtendedFieldElement*

Create an element of the finite extended field.

The argument **representative** must be an **FiniteFieldPolynomial** has same basefield. Another argument **field** must be an instance of ExtendedField.

Operations

operator	explanation
a+b	addition.
a-b	subtraction.
a*b	multiplication.
a/b	inverse multiplication.
a**n, pow(a,n)	power.
-a	negation.
+a	make a copy.
a==b	equality test.
a!=b	inequality test.
repr(a)	return representation string.
str(a)	return string.

Methods

4.3.6.1 `getRing` – get ring object

`getRing(self)` → *FinitePrimeField*

Return an instance of `FinitePrimeField` to which the element belongs.

4.3.6.2 `inverse` – inverse element

`inverse(self)` → *ExtendedFieldElement*

Return the inverse element.

4.4 group – algorithms for finite groups

- Classes
 - **Group**
 - **GroupElement**
 - **GenerateGroup**
 - **AbelianGenerate**

4.4.1 †Group – group structure

Initialize (Constructor)

Group(value: *class*, operation: *int=-1*) → **Group**

Create an object which wraps *value* (typically a ring or a field) only to expose its group structure.

The instance has methods defined for (abstract) group. For example, **identity** returns the identity element of the group from wrapped *value*.

value must be an instance of a class expresses group structure. *operation* must be 0 or 1; If *operation* is 0, *value* is regarded as the additive group. On the other hand, if *operation* is 1, *value* is considered as the multiplicative group. The default value of *operation* is 0.

†You can input an instance of **Group** itself as *value*. In this case, the default value of *operation* is the attribute **operation** of the instance.

Attribute

entity :

The wrapped object.

operation :

It expresses the mode of operation; 0 means additive, while 1 means multiplicative.

Operations

operator	explanation
A==B	Return whether A and B are equal or not.
A!=B	Check whether A and B are not equal.
repr(A)	representation
str(A)	simple representation

Examples

```
>>> G1=group.Group(finitefield.FinitePrimeField(37), 1)
>>> print G1
F_37
>>> G2=group.Group(intresidue.IntegerResidueClassRing(6), 0)
```

```
>>> print G2  
Z/6Z
```


Methods

4.4.1.1 `setOperation` – change operation

`setOperation(self, operation: int) → (None)`

Change group type to additive (0) or multiplicative (1).

`operation` must be 0 or 1.

4.4.1.2 `†createElement` – generate a `GroupElement` instance

`createElement(self, *value) → GroupElement`

Return **`GroupElement`** object whose group is `self`, initialized with `value`.

†This method calls `self.entity.createElement`.

`value` must fit the form of argument for `self.entity.createElement`.

4.4.1.3 `†identity` – identity element

`identity(self) → GroupElement`

Return identity element (unit) of group.

Return zero (additive) or one (multiplicative) corresponding to **`operation`**.

†This method calls `self.entity.identity` or **`entity`** does not have the attribute then returns one or zero.

4.4.1.4 `grouporder` – order of the group

`grouporder(self) → long`

Return group order (cardinality) of `self`.

†This method calls `self.entity.grouporder`, `card` or `__len__`.

We assume that the group is finite, so returned value is expected as some long integer. If the group is infinite, we do not define the type of output by the method.

Examples

```
>>> G1=group.Group(finitefield.FinitePrimeField(37), 1)
>>> G1.grouporder()
36
>>> G1.setOperation(0)
>>> print G1.identity()
FinitePrimeField,0 in F_37
>>> G1.grouporder()
37
```

4.4.2 GroupElement – elements of group structure

Initialize (Constructor)

GroupElement(value: *class*, operation: *int*==1) → **GroupElement**

Create an object which wraps *value* (typically a ring element or a field element) to make it behave as an element of group.

The instance has methods defined for an (abstract) element of group. For example, **inverse** returns the inverse element of *value* as the element of group object.

value must be an instance of a class expresses an element of group structure. *operation* must be 0 or 1; If *operation* is 0, *value* is regarded as the additive group. On the other hand, if *operation* is 1, *value* is considered as the multiplicative group. The default value of *operation* is 0.

†You can input an instance of **GroupElement** itself as *value*. In this case, the default value of *operation* is the attribute **operation** of the instance.

Attribute

entity :

The wrapped object.

set :

It is an instance of **Group**, which expresses the group to which **self** belongs.

operation :

It expresses the mode of operation; 0 means additive, while 1 means multiplicative.

Operations

operator	explanation
A==B	Return whether A and B are equal or not.
A!=B	Check whether A and B are not equal.
A.ope(B)	Basic operation (additive +, multiplicative *)
A.ope2(n)	Extended operation (additive *, multiplicative **)
A.inverse()	Return the inverse element of self
repr(A)	representation
str(A)	simple representation

Examples

```
>>> G1=group.GroupElement(finitefield.FinitePrimeFieldElement(18, 37), 1)
>>> print G1
FinitePrimeField,18 in F_37
>>> G2=group.Group(intresidue.IntegerResidueClass(3, 6), 0)
IntegerResidueClass(3, 6)
```

Methods

4.4.2.1 `setOperation` – change operation

`setOperation(self, operation: int) → (None)`

Change group type to additive (0) or multiplicative (1).

`operation` must be 0 or 1.

4.4.2.2 `†getGroup` – generate a `Group` instance

`getGroup(self) → Group`

Return **Group** object to which `self` belongs.

†This method calls `self.entity.getRing` or `getGroup`.

†In an initialization of **GroupElement**, the attribute `set` is set as the value returned from the method.

4.4.2.3 `order` – order by factorization method

`order(self) → long`

Return the order of `self`.

†This method uses the factorization of order of group.

†We assume that the group is finite, so returned value is expected as some long integer. †If the group is infinite, the method would raise an error or return an invalid value.

4.4.2.4 `t_order` – order by baby-step giant-step

`t_order(self, v: int=2) → long`

Return the order of `self`.

†This method uses Terr's baby-step giant-step algorithm.

This method does not use the order of group. You can put number of baby-step to `v`. †We assume that the group is finite, so returned value is expected as some

long integer. †If the group is infinite, the method would raise an error or return an invalid value.

v must be some int integer.

Examples

```
>>> G1=group.GroupElement(finitefield.FinitePrimeFieldElement(18, 37), 1)
>>> G1.order()
36
>>> G1.t_order()
36
```

4.4.3 †GenerateGroup – group structure with generator

Initialize (Constructor)

GenerateGroup(value: *class*, operation: *int*=-1) → **GroupElement**

Create an object which is generated by **value** as the element of group structure.

This initializes a group ‘including’ the group elements, not a group with generators, now. We do not recommend using this module now. The instance has methods defined for an (abstract) element of group. For example, **inverse** returns the inverse element of **value** as the element of group object.

The class inherits the class **Group**.

value must be a list of generators. Each generator should be an instance of a class expresses an element of group structure. **operation** must be 0 or 1; If **operation** is 0, **value** is regarded as the additive group. On the other hand, if **operation** is 1, **value** is considered as the multiplicative group. The default value of **operation** is 0.

Examples

```
>>> G1=group.GenerateGroup([intresidue.IntegerResidueClass(2, 20),  
... intresidue.IntegerResidueClass(6, 20)])  
>>> G1.identity()  
intresidue.IntegerResidueClass(0, 20)
```

4.4.4 AbelianGenerate – abelian group structure with generator

Initialize (Constructor)

The class inherits the class **GenerateGroup**.

4.4.4.1 relationLattice – relation between generators

relationLattice(self) → Matrix

Return a list of relation lattice basis as a square matrix each of whose column vector is a relation basis.

The relation basis, V satisfies that $\prod_i \text{generator}_i V_i = 1$.

4.4.4.2 computeStructure – abelian group structure

computeStructure(self) → tuple

Compute finite abelian group structure.

If **self** $G \simeq \oplus_i \langle h_i \rangle$, return $[(h_1, \text{ord}(h_1)), \dots, (h_n, \text{ord}(h_n))]$ and $\#G$, where $\langle h_i \rangle$ is a cyclic group with the generator h_i .

The output is a tuple which has two elements; the first element is a list which elements are a list of h_i and its order, on the other hand, the second element is the order of the group.

Examples

```
>>> G=AbelianGenerate([intresidue.IntegerResidueClass(2, 20),
... intresidue.IntegerResidueClass(6, 20)])
>>> G.relationLattice()
10 7
0 1
>>> G.computeStructure()
([IntegerResidueClassRing,IntegerResidueClass(2, 20), 10]), 10L)
```


4.5 imaginary – complex numbers and its functions

The module `imaginary` provides complex numbers. The functions provided are mainly corresponding to the `cmath` standard module.

- **Classes**

- **ComplexField**
- **Complex**
- †**ExponentialPowerSeries**
- †**AbsoluteError**
- †**RelativeError**

- **Functions**

- **exp**
- **expi**
- **log**
- **sin**
- **cos**
- **tan**
- **sinh**
- **cosh**
- **tanh**
- **atanh**
- **sqrt**

This module also provides following constants:

e :
This constant is obsolete (Ver 1.1.0).

pi :
This constant is obsolete (Ver 1.1.0).

j :
j is the imaginary unit.

theComplexField :
theComplexField is the instance of **ComplexField**.

4.5.1 ComplexField – field of complex numbers

The class is for the field of complex numbers. The class has the single instance **theComplexField**.

This class is a subclass of **Field**.

Initialize (Constructor)

ComplexField() \rightarrow *ComplexField*

Create an instance of ComplexField. You may not want to create an instance, since there is already **theComplexField**.

Attribute

zero :
It expresses The additive unit 0. (read only)

one :
It expresses The multiplicative unit 1. (read only)

Operations

operator	explanation
in	membership test; return whether an element is in or not.
repr	return representation string.
str	return string.

Methods

4.5.1.1 createElement – create Imaginary object

`createElement(self, seed: integer) → Integer`

Return a Complex object with `seed`.

`seed` must be complex or numbers having embedding to complex.

4.5.1.2 getCharacteristic – get characteristic

`getCharacteristic(self) → integer`

Return the characteristic, zero.

4.5.1.3 issubring – subring test

`issubring(self, aRing: Ring) → bool`

Report whether another ring contains the complex field as subring.

4.5.1.4 issuperring – superring test

`issuperring(self, aRing: Ring) → bool`

Report whether the complex field contains another ring as subring.

4.5.2 Complex – a complex number

Complex is a class of complex number. Each instance has a coupled numbers; real and imaginary part of the number.

This class is a subclass of **FieldElement**.

All implemented operators in this class are delegated to complex type.

Initialize (Constructor)

Complex(re: *number* im: *number*=0) → *Imaginary*

Create a complex number.

re can be either real or complex number. If **re** is real and **im** is not given, then its imaginary part is zero.

Attribute

real :

It expresses the real part of complex number.

imag :

It expresses the imaginary part of complex number.

Methods

4.5.2.1 `getRing` – get ring object

`getRing(self)` → *ComplexField*

Return the complex field instance.

4.5.2.2 `arg` – argument of complex

`arg(self)` → *radian*

Return the angle between the x-axis and the number in the Gaussian plane.
radian must be Float.

4.5.2.3 `conjugate` – complex conjugate

`conjugate(self)` → *Complex*

Return the complex conjugate of the number.

4.5.2.4 `copy` – copied number

`copy(self)` → *Complex*

Return the copy of the number itself.

4.5.2.5 `inverse` – complex inverse

`inverse(self)` → *Complex*

Return the inverse of the number.
If the number is zero, `ZeroDivisionError` is raised.

4.5.3 ExponentialPowerSeries – exponential power series

This class is obsolete (Ver 1.1.0).

4.5.4 AbsoluteError – absolute error

This class is obsolete (Ver 1.1.0).

4.5.5 RelativeError – relative error

This class is obsolete (Ver 1.1.0).

4.5.6 exp(function) – exponential value

This function is obsolete (Ver 1.1.0).

4.5.7 expi(function) – imaginary exponential value

This function is obsolete (Ver 1.1.0).

4.5.8 log(function) – logarithm

This function is obsolete (Ver 1.1.0).

4.5.9 sin(function) – sine function

This function is obsolete (Ver 1.1.0).

4.5.10 cos(function) – cosine function

This function is obsolete (Ver 1.1.0).

4.5.11 tan(function) – tangent function

This function is obsolete (Ver 1.1.0).

4.5.12 sinh(function) – hyperbolic sine function

This function is obsolete (Ver 1.1.0).

4.5.13 cosh(function) – hyperbolic cosine function

This function is obsolete (Ver 1.1.0).

4.5.14 tanh(function) – hyperbolic tangent function

This function is obsolete (Ver 1.1.0).

4.5.15 `atanh(function)` – hyperbolic arc tangent function

This function is obsolete (Ver 1.1.0).

4.5.16 `sqrt(function)` – square root

This function is obsolete (Ver 1.1.0).

4.6 intresidue – integer residue

intresidue module provides integer residue classes or $\mathbf{Z}/m\mathbf{Z}$.

- **Classes**
 - **IntegerResidueClass**
 - **IntegerResidueClassRing**

4.6.1 IntegerResidueClass – integer residue class

This class is a subclass of **CommutativeRingElement**.

Initialize (Constructor)

IntegerResidueClass(representative: *integer*, modulus: *integer*)
→ *Integer*

Create a residue class of modulus with residue representative.
modulus must be positive integer.

Operations

operator	explanation
a+b	addition.
a-b	subtraction.
a*b	multiplication.
a/b	division.
a**i, pow(a,i)	power.
-a	negation.
+a	make a copy.
a==b	equality or not.
a!=b	inequality or not.
repr(a)	return representation string.
str(a)	return string.

Methods

4.6.1.1 `getRing` – get ring object

`getRing(self)` → *IntegerResidueClassRing*

Return a ring to which it belongs.

4.6.1.2 `getResidue` – get residue

`getResidue(self)` → *integer*

Return the value of residue.

4.6.1.3 `getModulus` – get modulus

`getModulus(self)` → *integer*

Return the value of modulus.

4.6.1.4 `inverse` – inverse element

`inverse(self)` → *IntegerResidueClass*

Return the inverse element if it is invertible. Otherwise raise `ValueError`.

4.6.1.5 `minimumAbsolute` – minimum absolute representative

`minimumAbsolute(self)` → **Integer**

Return the minimum absolute representative integer of the residue class.

4.6.1.6 `minimumNonNegative` – smallest non-negative representative

`minimumNonNegative(self)` → **Integer**

Return the smallest non-negative representative element of the residue class.

†this method has an alias, named `toInteger`.

4.6.2 IntegerResidueClassRing – ring of integer residue

The class is for rings of integer residue classes.

This class is a subclass of **CommutativeRing**.

Initialize (Constructor)

IntegerResidueClassRing(modulus: *integer*) \rightarrow *IntegerResidueClassRing*

Create an instance of IntegerResidueClassRing. The argument modulus = m specifies an ideal $m\mathbb{Z}$.

Attribute

zero :

It expresses The additive unit 0. (read only)

one :

It expresses The multiplicative unit 1. (read only)

Operations

operator	explanation
<code>R==A</code>	ring equality.
<code>card(R)</code>	return cardinality. See also compatibility module.
<code>e in R</code>	return whether an element is in or not.
<code>repr(R)</code>	return representation string.
<code>str(R)</code>	return string.

Methods

4.6.2.1 createElement – create IntegerResidueClass object

createElement(self, seed: *integer*) → *Integer*

Return an IntegerResidueClass instance with *seed*.

4.6.2.2 isfield – field test

isfield(self) → *bool*

Return True if the modulus is prime, False if not. Since a finite domain is a field, other ring property tests are merely aliases of isfield; they are isdomain, iseuclidean, isnoetherian, ispid, isufd.

4.6.2.3 getInstance – get instance of IntegerResidueClassRing

getInstance(cls, modulus: *integer*) → *IntegerResidueClass*

Return an instance of the class of specified modulus. Since this is a class method, use it as:

`IntegerResidueClassRing.getInstance(3)`
to create a $\mathbb{Z}/3\mathbb{Z}$ object, for example.

4.7 lattice – Lattice

- Classes
 - **Lattice**
 - **LatticeElement**
- Functions
 - **LLL**

4.7.1 Lattice – lattice

Initialize (Constructor)

```
Lattice(basis: RingSquareMatrix, quadraticForm: RingSquareMatrix)  
    → Lattice
```

Create Lattice object.

Attribute

basis : The basis of **self** lattice.

quadraticForm : The quadratic form corresponding the inner product.

Methods

4.7.1.1 createElement – create element

createElement(self, compo: *list*) → **LatticeElement**

Create the element which has coefficients with given compo.

4.7.1.2 bilinearForm – bilinear form

bilinearForm(self, v_1: **Vector**, v_2: **Vector**) → *integer*

Return the inner product of v_1 and v_2 with **quadraticForm**.

4.7.1.3 isCyclic – Check whether cyclic lattice or not

isCyclic(self) → *bool*

Check whether self lattice is a cyclic lattice or not.

4.7.1.4 isIdeal – Check whether ideal lattice or not

signature(self) → *bool*

Check whether self lattice is an ideal lattice or not.

4.7.2 LatticeElement – element of lattice

Initialize (Constructor)

```
LatticeElement( lattice: Lattice, compo: list, ) → LatticeElement
```

Create LatticeElement object.

Elements of lattices are represented as linear combinations of basis. The class inherits **Matrix**. Then, instances are regarded as $n \times 1$ matrix whose coefficients consist of `compo`, where n is the dimension of lattice.

`lattice` is an instance of Lattice object. `compo` is coefficients list of basis.

Attribute

`lattice` : the lattice which includes `self`

Methods

4.7.2.1 `getLattice` – Find lattice belongs to

`getLattice(self)` → **Lattice**

Obtain the Lattice object corresponding to `self`.

4.7.3 LLL(function) – LLL reduction

LLL(M: RingSquareMatrix) → L: RingSquareMatrix, T: RingSquareMatrix

Return LLL-reduced basis for the given basis M.

The output L is the LLL-reduced basis. T is the transportation matrix from the original basis to the LLL-reduced basis.

Examples

```
>>> M=mat.Matrix(3,3,[1,0,12,0,1,26,0,0,13]);
>>> lat.LLL(M);
([1, 0, 0]+[0, 1, 0]+[0, 0, 13], [1L, 0L, -12L]+[0L, 1L, -26L]+[0L, 0L, 1L])
```

4.8 matrix – matrices

- **Classes**
 - **Matrix**
 - **SquareMatrix**
 - **RingMatrix**
 - **RingSquareMatrix**
 - **FieldMatrix**
 - **FieldSquareMatrix**
 - **MatrixRing**
 - **Subspace**
- **Functions**
 - **createMatrix**
 - **identityMatrix**
 - **unitMatrix**
 - **zeroMatrix**

The module matrix has also some exception classes.

MatrixSizeError : Report contradicting given input to the matrix size.

VectorsNotIndependent : Report column vectors are not independent.

NoInverseImage : Report any inverse image does not exist.

NoInverse : Report the matrix is not invertible.

This module using following type:

compo : compo must be one of these forms below.

- concatenated row lists, such as $[1,2]+[3,4]+[5,6]$.
- list of row lists, such as $[[1,2], [3,4], [5,6]]$.
- list of column tuples, such as $[(1, 3, 5), (2, 4, 6)]$.
- list of vectors whose dimension equals column, such as *vector.Vector*([1, 3, 5]), *vector.Vector*([2, 4, 6]).

The examples above represent the same matrix form as follows:

$$\begin{array}{cc} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array}$$

4.8.1 Matrix – matrices

Initialize (Constructor)

```
Matrix(row: integer, column: integer, compo: compo=0, coeff_ring:
CommutativeRing=0)
    → Matrix
```

Create new matrices object.

†This constructor automatically changes the class to one of the following class: **RingMatrix**, **RingSquareMatrix**, **FieldMatrix**, **FieldSquareMatrix**.

Your input determines the class automatically by examining the matrix size and the coefficient ring. **row** and **column** must be integer, and **coeff_ring** must be an instance of **Ring**. Refer to **compo** for information about **compo**. If you abbreviate **compo**, it will be deemed to all zero list.

The list of expected inputs and outputs is as following:

- Matrix(**row**, **column**, **compo**, **coeff_ring**)
→ the **row**×**column** matrix whose elements are **compo** and coefficient ring is **coeff_ring**
- Matrix(**row**, **column**, **compo**)
→ the **row**×**column** matrix whose elements are **compo** (The coefficient ring is automatically determined.)
- Matrix(**row**, **column**, **coeff_ring**)
→ the **row**×**column** matrix whose coefficient ring is **coeff_ring** (All elements are 0 in **coeff_ring**.)
- Matrix(**row**, **column**)
→ the **row**×**column** matrix (The coefficient matrix is **Integer**. All elements are 0.)

Attribute

row : The row size of the matrix.

column : The column size of the matrix.

coeff_ring : The coefficient ring of the matrix.

compo : The elements of the matrix.

Operations

operator	explanation
<code>M==N</code>	Return whether M and N are equal or not.
<code>M[i, j]</code>	Return the coefficient of i-th row, j-th column term of matrix M.
<code>M[i]</code>	Return the vector of i-th column term of matrix M.
<code>M[i, j]=c</code>	Replace the coefficient of i-th row, j-th column term of matrix M by c.
<code>M[j]=c</code>	Replace the vector of i-th column term of matrix M by vector c.
<code>c in M</code>	Check whether some element of M equals c.
<code>repr(M)</code>	Return the repr string of the matrix M. string represents list concatenated row vector lists.
<code>str(M)</code>	Return the str string of the matrix M.

Examples

```

>>> A = matrix.Matrix(2, 3, [1,0,0]+[0,0,0])
>>> A.__class__.__name__
'RingMatrix'
>>> B = matrix.Matrix(2, 3, [1,0,0,0,0,0])
>>> A == B
True
>>> B[1, 1] = 0
>>> A != B
True
>>> B == 0
True
>>> A[1, 1]
1
>>> print repr(A)
[1, 0, 0]+[0, 0, 0]
>>> print str(A)
1 0 0
0 0 0

```

Methods

4.8.1.1 map – apply function to elements

map(self, function: *function*) → *Matrix*

Return the matrix whose elements is applied `function` to.

†The function `map` is an analogy of built-in function `map`.

4.8.1.2 reduce – reduce elements iteratively

reduce(self, function: *function*, initializer: *RingElement*=None)
→ *RingElement*

Apply `function` from upper-left to lower-right, so as to reduce the iterable to a single value.

†The function `map` is an analogy of built-in function `reduce`.

4.8.1.3 copy – create a copy

copy(self) → *Matrix*

create a copy of `self`.

†The matrix generated by the function is same matrix to `self`, but not same as a instance.

4.8.1.4 set – set compo

set(self, compo: *compo*) → (*None*)

Substitute the list `compo` for `compo`.

`compo` must be the form of `compo`.

4.8.1.5 `setRow` – set m-th row vector

```
setRow(self, m: integer, arg: list/Vector) → (None)
```

Substitute the list/Vector `arg` as m-th row.

The length of `arg` must be same to `self.column`.

4.8.1.6 `setColumn` – set n-th column vector

```
setColumn(self, n: integer, arg: list/Vector) → (None)
```

Substitute the list/Vector `arg` as n-th column.

The length of `arg` must be same to `self.row`.

4.8.1.7 `getRow` – get i-th row vector

```
getRow(self, i: integer) → Vector
```

Return i-th row in form of `self`.

The function returns a row vector (an instance of `Vector`).

4.8.1.8 `getColumn` – get j-th column vector

```
getColumn(self, j: integer) → Vector
```

Return j-th column in form of `self`.

4.8.1.9 `swapRow` – swap two row vectors

```
swapRow(self, m1: integer, m2: integer) → (None)
```

Swap `self`'s m1-th row vector and m2-th row one.

4.8.1.10 swapColumn – swap two column vectors

```
swapColumn(self, n1: integer, n2: integer) → (None)
```

Swap `self`'s `n1`-th column vector and `n2`-th column one.

4.8.1.11 insertRow – insert row vectors

```
insertRow(self, i: integer, arg: list/Vector/Matrix)  
→ (None)
```

Insert row vectors `arg` to `i`-th row.

`arg` must be list, **Vector** or **Matrix**. The length (or **column**) of it should be same to the column of `self`.

4.8.1.12 insertColumn – insert column vectors

```
insertColumn(self, j: integer, arg: list/Vector/Matrix)  
→ (None)
```

Insert column vectors `arg` to `j`-th column.

`arg` must be list, **Vector** or **Matrix**. The length (or **row**) of it should be same to the row of `self`.

4.8.1.13 extendRow – extend row vectors

```
extendRow(self, arg: list/Vector/Matrix) → (None)
```

Join `self` with row vectors `arg` (in vertical way).

The function combines `self` with the last row vector of `self`. That is, `extendRow(arg)` is same to `insertRow(self.row+1, arg)`.

`arg` must be list, **Vector** or **Matrix**. The length (or **column**) of it should be same to the column of `self`.

4.8.1.14 extendColumn – extend column vectors

```
extendColumn(self, arg: list/Vector/Matrix) → (None)
```

Join `self` with column vectors `arg` (in horizontal way).

The function combines `self` with the last column vector of `self`. That is,

`extendColumn(arg)` is same to `insertColumn(self.column+1, arg)`.

`arg` must be list, **Vector** or **Matrix**. The length (or **row**) of it should be same to the row of `self`.

4.8.1.15 deleteRow – delete row vector

`deleteRow(self, i: integer) → (None)`

Delete i-th row vector.

4.8.1.16 deleteColumn – delete column vector

`deleteColumn(self, j: integer) → (None)`

Delete j-th column vector.

4.8.1.17 transpose – transpose matrix

`transpose(self) → Matrix`

Return the transpose of `self`.

4.8.1.18 getBlock – block matrix

`getBlock(self, i: integer, j: integer, row: integer, column: integer=None)
→ Matrix`

Return the `row`×`column` block matrix from the (i, j)-element.

If `column` is omitted, `column` is considered as same value to `row`.

4.8.1.19 subMatrix – submatrix

`subMatrix(self, I: integer, J: integer=None) → Matrix`

`subMatrix(self, I: list, J: list=None) → Matrix`

The function has a twofold significance.

- I and J are integer:
Return submatrix deleted I-th row and J-th column.
- I and J are list:
Return the submatrix composed of elements from `self` assigned by rows I and columns J, respectively.

If J is omitted, J is considered as same value to I.

Examples

```
>>> A = matrix.Matrix(2, 3, [1,2,3]+[4,5,6])
>>> A
[1, 2, 3]+[4, 5, 6]
>>> A.map(complex)
[(1+0j), (2+0j), (3+0j)]+[(4+0j), (5+0j), (6+0j)]
>>> A.reduce(max)
6
>>> A.swapRow(1, 2)
>>> A
[4, 5, 6]+[1, 2, 3]
>>> A.extendColumn([-2, -1])
>>> A
[4, 5, 6, -2]+[1, 2, 3, -1]
>>> B = matrix.Matrix(3, 3, [1,2,3]+[4,5,6]+[7,8,9])
>>> B.subMatrix(2, 3)
[1, 2]+[7, 8]
>>> B.subMatrix([2, 3], [1, 2])
[4, 5]+[7, 8]
```

4.8.2 SquareMatrix – square matrices

Initialize (Constructor)

```
SquareMatrix(row: integer, column: integer=0, compo: compo=0,  
coeff_ring: CommutativeRing=0)  
→ SquareMatrix
```

Create new square matrices object.

SquareMatrix is subclass of **Matrix**. †This constructor automatically changes the class to one of the following class: **RingMatrix**, **RingSquareMatrix**, **FieldMatrix**, **FieldSquareMatrix**.

Your input determines the class automatically by examining the matrix size and the coefficient ring. **row** and **column** must be integer, and **coeff_ring** must be an instance of **Ring**. Refer to **compo** for information about **compo**. If you abbreviate **compo**, it will be deemed to all zero list.

The list of expected inputs and outputs is as following:

- Matrix(**row**, **compo**, **coeff_ring**)
→ the **row** square matrix whose elements are **compo** and coefficient ring is **coeff_ring**
- Matrix(**row**, **compo**)
→ the **row** square matrix whose elements are **compo** (coefficient ring is automatically determined)
- Matrix(**row**, **coeff_ring**)
→ the **row** square matrix whose coefficient ring is **coeff_ring** (All elements are 0 in **coeff_ring**.)
- Matrix(**row**)
→ the **row** square matrix (The coefficient ring is Integer. All elements are 0.)

†We can initialize as **Matrix**, but **column** must be same to **row** in the case.

Methods

4.8.2.1 isUpperTriangularMatrix – check upper triangular

`isUpperTriangularMatrix(self)` → *True/False*

Check whether `self` is upper triangular matrix or not.

4.8.2.2 isLowerTriangularMatrix – check lower triangular

`isLowerTriangularMatrix(self)` → *True/False*

Check whether `self` is lower triangular matrix or not.

4.8.2.3 isDiagonalMatrix – check diagonal matrix

`isDiagonalMatrix(self)` → *True/False*

Check whether `self` is diagonal matrix or not.

4.8.2.4 isScalarMatrix – check scalar matrix

`isScalarMatrix(self)` → *True/False*

Check whether `self` is scalar matrix or not.

4.8.2.5 isSymmetricMatrix – check symmetric matrix

`isSymmetricMatrix(self)` → *True/False*

Check whether `self` is symmetric matrix or not.

Examples

```
>>> A = matrix.SquareMatrix(3, [1,2,3]+[0,5,6]+[0,0,9])
>>> A.isUpperTriangularMatrix()
```

```
True
>>> B = matrix.SquareMatrix(3, [1,0,0]+[0,-2,0]+[0,0,7])
>>> B.isDiagonalMatrix()
True
```

4.8.3 RingMatrix – matrix whose elements belong ring

```
RingMatrix(row: integer, column: integer, compo: compo=0, coeff_ring:
CommutativeRing=0)
→ RingMatrix
```

Create matrix whose coefficient ring belongs ring.

RingMatrix is subclass of **Matrix**. See Matrix for getting information about the initialization.

Operations

operator	explanation
M+N	Return the sum of matrices M and N.
M-N	Return the difference of matrices M and N.
M*N	Return the product of M and N. N must be matrix, vector or scalar
M % d	Return M modulo d. d must be nonzero integer.
-M	Return the matrix whose coefficients have inverted signs of M.
+M	Return the copy of M.

Examples

```
>>> A = matrix.Matrix(2, 3, [1,2,3]+[4,5,6])
>>> B = matrix.Matrix(2, 3, [7,8,9]+[0,-1,-2])
>>> A + B
[8, 10, 12]+[4, 4, 4]
>>> A - B
[-6, -6, -6]+[4, 6, 8]
>>> A * B.transpose()
[50, -8]+[122, -17]
>>> -B * vector.Vector([1, -1, 0])
Vector([1, -1])
>>> 2 * A
[2, 4, 6]+[8, 10, 12]
>>> B % 3
[1, 2, 0]+[0, 2, 1]
```

Methods

4.8.3.1 getCoefficientRing – get coefficient ring

`getCoefficientRing(self)` → *CommutativeRing*

Return the coefficient ring of `self`.

This method checks all elements of `self` and set `coeff_ring` to the valid coefficient ring.

4.8.3.2 toFieldMatrix – set field as coefficient ring

`toFieldMatrix(self)` → (*None*)

Change the class of the matrix to **FieldMatrix** or **FieldSquareMatrix**, where the coefficient ring will be the quotient field of the current domain.

4.8.3.3 toSubspace – regard as vector space

`toSubspace(self, isbasis: True/False=None)` → (*None*)

Change the class of the matrix to `Subspace`, where the coefficient ring will be the quotient field of the current domain.

4.8.3.4 hermiteNormalForm (HNF) – Hermite Normal Form

`hermiteNormalForm(self)` → *RingMatrix*

`HNF(self)` → *RingMatrix*

Return upper triangular Hermite normal form (HNF).

4.8.3.5 exthermiteNormalForm (extHNF) – extended Hermite Normal Form algorithm

`exthermiteNormalForm(self)` → (*RingSquareMatrix, RingMatrix*)

`extHNF(self)` → (*RingSquareMatrix, RingMatrix*)

Return Hermite normal form M and U satisfied $\text{self}U = M$.

The function returns tuple (U, M) , where U is an instance of **RingSquareMatrix** and M is an instance of **RingMatrix**.

4.8.3.6 kernelAsModule – kernel as \mathbb{Z} -module

kernelAsModule(self) \rightarrow *RingMatrix*

Return kernel as \mathbb{Z} -module.

The difference between the function and **kernel** is that each elements of the returned value are integer.

Examples

```
>>> A = matrix.Matrix(3, 4, [1,2,3,4,5,6,7,8,9,-1,-2,-3])
>>> print A.hermiteNormalForm()
0 36 29 28
0 0 1 0
0 0 0 1
>>> U, M = A.hermiteNormalForm()
>>> A * U == M
True
>>> B = matrix.Matrix(1, 2, [2, 1])
>>> print B.kernelAsModule()
1
-2
```

4.8.4 RingSquareMatrix – square matrix whose elements belong ring

```
RingSquareMatrix(row: integer, column: integer=0, compo: compo=0,  
coeff_ring: CommutativeRing=0)  
→ RingMatrix
```

Create square matrix whose coefficient ring belongs ring.

RingSquareMatrix is subclass of **RingMatrix** and **SquareMatrix**. See SquareMatrix for getting information about the initialization.

Operations

operator	explanation
M**c	Return the c-th power of matrices M.

Examples

```
>>> A = matrix.RingSquareMatrix(3, [1,2,3]+[4,5,6]+[7,8,9])  
>>> A ** 2  
[30L, 36L, 42L]+[66L, 81L, 96L]+[102L, 126L, 150L]
```


Methods

4.8.4.1 `getRing` – get matrix ring

`getRing(self)` → *MatrixRing*

Return the **MatrixRing** belonged to by `self`.

4.8.4.2 `isOrthogonalMatrix` – check orthogonal matrix

`isOrthogonalMatrix(self)` → *True/False*

Check whether `self` is orthogonal matrix or not.

4.8.4.3 `isAlternatingMatrix` (`isAntiSymmetricMatrix`, `isSkewSymmetricMatrix`) – check alternating matrix

`isAlternatingMatrix(self)` → *True/False*

Check whether `self` is alternating matrix or not.

4.8.4.4 `isSingular` – check singular matrix

`isSingular(self)` → *True/False*

Check whether `self` is singular matrix or not.

The function determines whether determinant of `self` is 0. Note that the non-singular matrix does not automatically mean invertible matrix; the nature that the matrix is invertible depends on its coefficient ring.

4.8.4.5 `trace` – trace

`trace(self)` → *RingElement*

Return the trace of `self`.

4.8.4.6 determinant – determinant

determinant(self) → *RingElement*

Return the determinant of **self**.

4.8.4.7 cofactor – cofactor

cofactor(self, i: *integer*, j: *integer*) → *RingElement*

Return the (i, j)-cofactor.

4.8.4.8 commutator – commutator

commutator(self, N: *RingSquareMatrix element*) → *RingSquareMatrix*

Return the commutator for **self** and N.

The commutator for M and N, which is denoted as $[M, N]$, is defined as $[M, N] = MN - NM$.

4.8.4.9 characteristicMatrix – characteristic matrix

characteristicMatrix(self) → *RingSquareMatrix*

Return the characteristic matrix of **self**.

4.8.4.10 adjugateMatrix – adjugate matrix

adjugateMatrix(self) → *RingSquareMatrix*

Return the adjugate matrix of **self**.

The adjugate matrix for M is the matrix N such that $MN = NM = (\det M)E$, where E is the identity matrix.

4.8.4.11 cofactorMatrix (cofactors) – cofactor matrix

cofactorMatrix(self) → *RingSquareMatrix*

cofactors(self) → *RingSquareMatrix*

Return the cofactor matrix of **self**.

The cofactor matrix for **M** is the matrix whose (i, j) element is (i, j) -cofactor of **M**. The cofactor matrix is same to transpose of the adjugate matrix.

4.8.4.12 smithNormalForm (SNF, elementary_divisor) – Smith Normal Form (SNF)

smithNormalForm(self) → *RingSquareMatrix*

SNF(self) → *RingSquareMatrix*

elementary_divisor(self) → *RingSquareMatrix*

Return the list of diagonal elements of the Smith Normal Form (SNF) for **self**.

The function assumes that **self** is non-singular.

4.8.4.13 extsmithNormalForm (extSNF) – Smith Normal Form (SNF)

extsmithNormalForm(self) → (*RingSquareMatrix*, *RingSquareMatrix*, *RingSquareMatrix*)

extSNF(self) → *RingSquareMatrix*, *RingSquareMatrix*, *RingSquareMatrix*

Return the Smith normal form **M** for **self** and **U, V** satisfied **UselfV = M**.

Examples

```
>>> A = matrix.RingSquareMatrix(3, [3,-5,8]+[-9,2,7]+[6,1,-4])
>>> A.trace()
1L
>>> A.determinant()
-243L
>>> B = matrix.RingSquareMatrix(3, [87,38,80]+[13,6,12]+[65,28,60])
>>> U, V, M = B.extsmithNormalForm()
>>> U * B * V == M
True
```

```
>>> print M
4 0 0
0 2 0
0 0 1
>>> B.smithNormalForm()
[4L, 2L, 1L]
```

4.8.5 FieldMatrix – matrix whose elements belong field

```
FieldMatrix(row: integer, column: integer, compo: compo=0, coeff_ring:
CommutativeRing=0)
→ RingMatrix
```

Create matrix whose coefficient ring belongs field.

FieldMatrix is subclass of **RingMatrix**. See **Matrix** for getting information about the initialization.

Operations

operator	explanation
M/d	Return the division of M by d.d must be scalar.
M//d	Return the division of M by d.d must be scalar.

Examples

```
>>> A = matrix.FieldMatrix(3, 3, [1,2,3,4,5,6,7,8,9])
>>> A / 210
1/210 1/105 1/70
2/105 1/42 1/35
1/30 4/105 3/70
```

Methods

4.8.5.1 kernel – kernel

kernel(self) → *FieldMatrix*

Return the kernel of **self**.

The output is the matrix whose column vectors form basis of the kernel.
The function returns None if the kernel do not exist.

4.8.5.2 image – image

image(self) → *FieldMatrix*

Return the image of **self**.

The output is the matrix whose column vectors form basis of the image.
The function returns None if the kernel do not exist.

4.8.5.3 rank – rank

rank(self) → *integer*

Return the rank of **self**.

4.8.5.4 inverseImage – inverse image: base solution of linear system

inverseImage(self, V: *Vector/RingMatrix*) → *RingMatrix*

Return an inverse image of V by **self**.

The function returns one solution of the linear equation **self**X = V.

4.8.5.5 solve – solve linear system

solve(self, B: *Vector/RingMatrix*) → (*RingMatrix*, *RingMatrix*)

Solve **self**X = B.

The function returns a particular solution **sol** and the kernel of **self** as a

matrix. If you only have to obtain the particular solution, use **inverseImage**.

4.8.5.6 columnEchelonForm – column echelon form

columnEchelonForm(self) → *RingMatrix*

Return the column reduced echelon form.

Examples

```
>>> A = matrix.FieldMatrix(2, 3, [1,2,3]+[4,5,6])
>>> print A.kernel
1/1
-2/1
1
>>> print A.image()
1 2
4 5
>>> C = matrix.FieldMatrix(4, 3, [1,2,3]+[4,5,6]+[7,8,9]+[-1,-2,-3])
>>> D = matrix.FieldMatrix(4, 2, [1,0]+[7,6]+[13,12]+[-1,0])
>>> print C.inverseImage(D)
3/1 4/1
-1/1 -2/1
0/1 0/1
>>> sol, ker = C.solve(D)
>>> C * (sol + ker[0]) == D
True
>>> AA = matrix.FieldMatrix(3, 3, [1,2,3]+[4,5,6]+[7,8,9])
>>> print AA.columnEchelonForm()
0/1 2/1 -1/1
0/1 1/1 0/1
0/1 0/1 1/1
```

4.8.6 FieldSquareMatrix – square matrix whose elements belong field

```
FieldSquareMatrix(row: integer, column: integer=0, compo: compo=0,  
coeff_ring: CommutativeRing=0)  
→ FieldSquareMatrix
```

Create square matrix whose coefficient ring belongs field.

FieldSquareMatrix is subclass of **FieldMatrix** and **SquareMatrix**.

†The function **RingSquareMatrix**determinant is overridden and use different algorithm from one used in **RingSquareMatrix**determinant;the function calls **FieldSquareMatrix**triangulate. See **SquareMatrix** for getting information about the initialization.

Methods

4.8.6.1 triangulate - triangulate by elementary row operation

triangulate(self) → *FieldSquareMatrix*

Return an upper triangulated matrix obtained by elementary row operations.

4.8.6.2 inverse - inverse matrix

inverse(self V: *Vector*/*RingMatrix*=None) → *FieldSquareMatrix*

Return the inverse of **self**. If V is given, then return **self**⁽⁻¹⁾V.

‡If the matrix is not invertible, then raise **NoInverse**.

4.8.6.3 hessenbergForm - Hessenberg form

hessenbergForm(self) → *FieldSquareMatrix*

Return the Hessenberg form of **self**.

4.8.6.4 LUdecomposition - LU decomposition

LUdecomposition(self) → (*FieldSquareMatrix*, *FieldSquareMatrix*)

Return the lower triangular matrix L and the upper triangular matrix U such that **self** == LU.

4.8.7 †MatrixRing – ring of matrices

```
MatrixRing(size: integer, scalars: CommutativeRing)  
→ MatrixRing
```

Create a ring of matrices with given `size` and coefficient ring `scalars`.

MatrixRing is subclass of **Ring**.

Methods

4.8.7.1 unitMatrix - unit matrix

`unitMatrix(self)` → *RingSquareMatrix*

Return the unit matrix.

4.8.7.2 zeroMatrix - zero matrix

`zeroMatrix(self)` → *RingSquareMatrix*

Return the zero matrix.

Examples

```
>>> M = matrix.MatrixRing(3, rational.theIntegerRing)
>>> print M
M_3(Z)
>>> M.unitMatrix()
[1L, 0L, 0L]+[0L, 1L, 0L]+[0L, 0L, 1L]
>>> M.zero
[0L, 0L, 0L]+[0L, 0L, 0L]+[0L, 0L, 0L]
```

4.8.7.3 getInstance(class function) - get cached instance

getInstance(cls, size: *integer*, scalars: *CommutativeRing*)
→ *RingSquareMatrix*

Return an instance of MatrixRing of given size and ring of scalars.

The merit of using the method instead of the constructor is that the instances created by the method are cached and reused for efficiency.

Examples

```
>>> print MatrixRing.getInstance(3, rational.theIntegerRing)
M_3(Z)
```

4.8.8 Subspace – subspace of finite dimensional vector space

```
Subspace(row: integer, column: integer=0, compo: compo=0, coeff_ring:
CommutativeRing=0, isbasis: True/False=None)
→ Subspace
```

Create subspace of some finite dimensional vector space over a field.

Subspace is subclass of **FieldMatrix**.

See **Matrix** for getting information about the initialization. The subspace expresses the space generated by column vectors of **self**.

If **isbasis** is True, we assume that column vectors are linearly independent.

Attribute

isbasis The attribute indicates the linear independence of column vectors, i.e., if they form a basis of the space then **isbasis** should be True, otherwise False.

Methods

4.8.8.1 issubspace - check subspace of self

Subspace(self, other: *Subspace*) → *True/False*

Return True if the subspace instance is a subspace of the **other**, or False otherwise.

4.8.8.2 toBasis - select basis

toBasis(self) → (*None*)

Rewrite **self** so that its column vectors form a basis, and set True to its **isbasis**.

The function does nothing if **isbasis** is already True.

4.8.8.3 supplementBasis - to full rank

supplementBasis(self) → *Subspace*

Return full rank matrix by supplementing bases for **self**.

4.8.8.4 sumOfSubspaces - sum as subspace

sumOfSubspaces(self, other: *Subspace*) → *Subspace*

Return a matrix whose columns form a basis for sum of two subspaces.

4.8.8.5 intersectionOfSubspaces - intersection as subspace

intersectionOfSubspaces(self, other: *Subspace*) → *Subspace*

Return a matrix whose columns form a basis for intersection of two subspaces.

Examples

```
>>> A = matrix.Subspace(4, 3, [1,2,3]+[4,5,6]+[7,8,9]+[10,11,12])
>>> A.toBasis()
>>> print A
1 2
4 5
7 8
10 11
>>> B = matrix.Subspace(3, 2, [1,2]+[3,4]+[5,7])
>>> print B.supplementBasis()
1 2 0
3 4 0
5 7 1
>>> C = matrix.Subspace(4, 1, [1,2,3,4])
>>> D = matrix.Subspace(4, 2, [2,-4]+[4,-3]+[6,-2]+[8,-1])
>>> print C.intersectionOfSubspaces(D)
-2/1
-4/1
-6/1
-8/1
```

4.8.8.6 fromMatrix(class function) - create subspace

```
fromMatrix(cls, mat: FieldMatrix, isbasis: True/False=None)  
→ Subspace
```

Create a Subspace instance from a matrix instance `mat`, whose class can be any of subclasses of `Matrix`.

Please use this method if you want a Subspace instance for sure.

4.8.9 createMatrix[function] – create an instance

```
createMatrix(row: integer, column: integer=0, compo: compo=0,  
coeff_ring: CommutativeRing=None)  
→ RingMatrix
```

Create an instance of **RingMatrix**, **RingSquareMatrix**, **FieldMatrix** or **FieldSquareMatrix**.

Your input determines the class automatically by examining the matrix size and the coefficient ring. See **Matrix** or **SquareMatrix** for getting information about the initialization.

4.8.10 identityMatrix(unitMatrix)[function] – unit matrix

```
identityMatrix(size: integer, coeff: CommutativeR-  
ing/CommutativeRingElement=None)  
→ RingMatrix
```

```
unitMatrix(size: integer, coeff: CommutativeR-  
ing/CommutativeRingElement=None)  
→ RingMatrix
```

Return size-dimensional unit matrix.

`coeff` enables us to create matrix not only in integer but in coefficient ring which is determined by `coeff`.

`coeff` must be an instance of **Ring** or a multiplicative unit (one).

4.8.11 zeroMatrix[function] – zero matrix

```
zeroMatrix(row: integer, column: 0=, coeff: CommutativeR-  
ing/CommutativeRingElement=None)  
→ RingMatrix
```

Return `row` × `column` zero matrix.

`coeff` enables us to create matrix not only in integer but in coefficient ring which is determined by `coeff`.

`coeff` must be an instance of **Ring** or an additive unit (zero). If `column` is abbreviated, `column` is set same to `row`.

Examples

```
>>> M = matrix.createMatrix(3, [1,2,3]+[4,5,6]+[7,8,9])
>>> print M
1 2 3
4 5 6
7 8 9
>>> O = matrix.zeroMatrix(2, 3, imaginary.ComplexField())
>>> print O
0 + 0j 0 + 0j 0 + 0j
0 + 0j 0 + 0j 0 + 0j
```

4.9 module – module/ideal with HNF

- **Classes**
 - **Submodule**
 - **Module**
 - **Ideal**
 - **Ideal_with_generator**

4.9.1 Submodule – submodule as matrix representation

Initialize (Constructor)

```
Submodule(row: integer, column: integer, compo: compo=0, coeff_ring: CommutativeRing=0, ishnf: True/False=None)  
→ Submodule
```

Create a submodule with matrix representation.

Submodule is subclass of **RingMatrix**.

We assume that `coeff_ring` is a PID (principal ideal domain). Then, we have the HNF(hermite normal form) corresponding to a matrices.

If `ishnf` is True, we assume that the input matrix is a HNF.

Attribute

ishnf If the matrix is a HNF, then `ishnf` should be True, otherwise False.

Methods

4.9.1.1 getGenerators – generator of module

`getGenerators(self) → list`

Return a (current) generator of the module `self`.

Return the list of vectors consisting of a generator.

4.9.1.2 isSubmodule – Check whether submodule of self

`isSubmodule(self, other: Submodule) → True/False`

Return True if the submodule instance is a submodule of the `other`, or False otherwise.

4.9.1.3 isEqual – Check whether self and other are same module

`isEqual(self, other: Submodule) → True/False`

Return True if the submodule instance is `other` as module, or False otherwise.

You should use the method for equality test of module, not matrix. For equality test of matrix simply, use `self==other`.

4.9.1.4 isContain – Check whether other is in self

`isContains(self, other: vector.Vector) → True/False`

Determine whether `other` is in `self` or not.

If you want to represent `other` as linear combination with the HNF generator of `self`, use `represent_element`.

4.9.1.5 toHNF - change to HNF

`toHNF(self) → (None)`

Rewrite `self` to HNF (hermite normal form), and set `True` to its `ishnf`.

Note that HNF do not always give basis of `self`. (i.e. HNF may be redundant.)

4.9.1.6 `sumOfSubmodules` - sum as submodule

`sumOfSubmodules(self, other: Submodule) → Submodule`

Return a module which is sum of two subspaces.

4.9.1.7 `intersectionOfSubmodules` - intersection as submodule

`intersectionOfSubmodules(self, other: Submodule)
→ Submodule`

Return a module which is intersection of two subspaces.

4.9.1.8 `represent_element` – represent element as linear combination

`represent_element(self, other: vector.Vector) → vector.Vector/False`

Represent `other` as a linear combination with HNF generators.

If `other` not in `self`, return `False`. Note that this method calls **toHNF**.

The method returns coefficients as an instance of **Vector**.

4.9.1.9 `linear_combination` – compute linear combination

`linear_combination(self, coeff: list) → vector.Vector`

For given **Z**-coefficients `coeff`, return a vector corresponding to a linear combination of (current) basis.

`coeff` must be a list of instances in **RingElement** whose size is the column of `self`.

Examples

```

>>> A = module.Submodule(4, 3, [1,2,3]+[4,5,6]+[7,8,9]+[10,11,12])
>>> A.toHNF()
>>> print A
9 1
6 1
3 1
0 1
>>> A.getGenerator
[Vector([9L, 6L, 3L, 0L]), Vector([1L, 1L, 1L, 1L])]
>>> V = vector.Vector([10,7,4,1])
>>> A.represent_element(V)
Vector([1L, 1L])
>>> V == A.linear_combination([1,1])
True
>>> B = module.Submodule(4, 1, [1,2,3,4])
>>> C = module.Submodule(4, 2, [2,-4]+[4,-3]+[6,-2]+[8,-1])
>>> print B.intersectionOfSubmodules(C)
2
4
6
8

```

4.9.2 fromMatrix(class function) - create submodule

```
fromMatrix(cls, mat: RingMatrix, ishnf: True/False=None)  
→ Submodule
```

Create a Submodule instance from a matrix instance `mat`, whose class can be any of subclasses of Matrix.

Please use this method if you want a Submodule instance for sure.

4.9.3 Module - module over a number field

Initialize (Constructor)

```
Module(pair_mat_repr:    list/matrix,    number_field:    al-
gfield.NumberField, base: list/matrix.SquareMatrix=None, ishnf:
bool=False)
    → Module
```

Create a new module object over a number field.

A module is a finitely generated sub \mathbf{Z} -module. Note that we do not assume rank of a module is $\deg(\text{number_field})$.

We represent a module as generators respect to base module over $\mathbf{Z}[\theta]$, where θ is a solution of `number_field.polynomial`.

`pair_mat_repr` should be one of the following form:

- $[M, d]$, where M is a list of integral tuple/vectors whose size is the degree of `number_field` and d is a denominator.
- $[M, d]$, where M is an integral matrix whose the number of row is the degree of `number_field` and d is a denominator.
- a rational matrix whose the number of row is the degree of `number_field`.

Also, `base` should be one of the following form:

- a list of rational tuple/vectors whose size is the degree of `number_field`
- a square non-singular rational matrix whose size is the degree of `number_field`

The module is internally represented as $\frac{1}{d}M$ with respect to `base`, where d is `denominator` and M is `mat_repr`. If `ishnf` is True, we assume that `mat_repr` is a HNF.

Attribute

`mat_repr` : an instance of `Submodule` M whose size is the degree of `number_field`

`denominator` : an integer d

`base` : a square non-singular rational matrix whose size is the degree of `number_field`

`number_field` : the number field over which the module is defined

Operations

operator	explanation
<code>M==N</code>	Return whether M and N are equal or not as module.
<code>c in M</code>	Check whether some element of M equals c.
<code>M+N</code>	Return the sum of M and N as module.
<code>M*N</code>	Return the product of M and N as the ideal computation. N must be module or scalar(i.e. an element of number_field). If you want to compute the intersection of <i>M</i> and <i>N</i> , see intersect .
<code>M**c</code>	Return M to c based on the ideal multiplication.
<code>repr(M)</code>	Return the repr string of the module M.
<code>str(M)</code>	Return the str string of the module M.

Examples

```

>>> F = algfield.NumberField([2,0,1])
>>> M_1 = module.Module([matrix.RingMatrix(2,2,[1,0]+[0,2]), 2], F)
>>> M_2 = module.Module([matrix.RingMatrix(2,2,[2,0]+[0,5]), 3], F)
>>> print M_1
([1, 0]+[0, 2], 2)
over
([1L, 0L]+[0L, 1L], NumberField([2, 0, 1]))
>>> print M_1 + M_2
([1L, 0L]+[0L, 2L], 6)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 * 2
([1L, 0L]+[0L, 2L], 1L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 * M_2
([2L, 0L]+[0L, 1L], 6L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 ** 2
([1L, 0L]+[0L, 2L], 4L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))

```

Methods

4.9.3.1 toHNF - change to hermite normal form(HNF)

`toHNF(self) → (None)`

Change `self.mat_repr` to the hermite normal form(HNF).

4.9.3.2 copy - create copy

`copy(self) → Module`

Create copy of `self`.

4.9.3.3 intersect - intersection

`intersect(self, other: Module) → Module`

Return intersection of `self` and `other`.

4.9.3.4 issubmodule - Check submodule

`submodule(self, other: Module) → True/False`

Check `self` is submodule of `other`.

4.9.3.5 issupermodule - Check supermodule

`supermodule(self, other: Module) → True/False`

Check `self` is supermodule of `other`.

4.9.3.6 represent_element - Represent as linear combination

`represent_element(self, other: algfield.BasicAlgNumber)
→ list/False`

Represent `other` as a linear combination with generators of `self`. If `other` is not in `self`, return False.

Note that we do not assume `self.mat_repr` is HNF.

The output is a list of integers if `other` is in `self`.

4.9.3.7 `change_base_module` - Change base

```
change_base_module(self, other_base: list/matrix.RingSquareMatrix)
    → Module
```

Return the module which is equal to `self` respect to `other_base`.

`other_base` follows the form `base`.

4.9.3.8 `index` - size of module

```
index(self) → rational.Rational
```

Return the order of a residue group over `self.base`. That is, return $[M : N]$ if $N \subset M$ or $[N : M]^{-1}$ if $M \subset N$, where M is the module `self` and N is the module corresponding to `self.base`.

4.9.3.9 `smallest_rational` - a \mathbb{Z} -generator in the rational field

```
smallest_rational(self) → rational.Rational
```

Return the \mathbb{Z} -generator of intersection of the module `self` and the rational field.

Examples

```
>>> F = algfield.NumberField([1,0,2])
>>> M_1=module.Module([matrix.RingMatrix(2,2,[1,0]+[0,2]), 2], F)
>>> M_2=module.Module([matrix.RingMatrix(2,2,[2,0]+[0,5]), 3], F)
>>> print M_1.intersect(M_2)
([2L, 0L]+[0L, 5L], 1L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
```

```

NumberField([2, 0, 1])
>>> M_1.represent_element( F.createElement( [[2,4], 1] ) )
[4L, 4L]
>>> print M_1.change_base_module( matrix.FieldSquareMatrix(2, 2, [1,0]+[0,1]) / 2 )
([1L, 0L]+[0L, 2L], 1L)
over
([Rational(1, 2), Rational(0, 1)]+[Rational(0, 1), Rational(1, 2)],
 NumberField([2, 0, 1]))
>>> M_2.index()
Rational(10, 9)
>>> M_2.smallest_rational()
Rational(2, 3)

```

4.9.4 Ideal - ideal over a number field

Initialize (Constructor)

```
Ideal(pair_mat_repr: list/matrix, number_field: algfield.NumberField,  
base: list/matrix.SquareMatrix=None, ishnf: bool=False)  
→ Ideal
```

Create a new ideal object over a number field.

Ideal is subclass of **Module**.

Refer to initialization of **Module**.

Methods

4.9.4.1 `inverse` – `inverse`

`inverse(self) → Ideal`

Return the inverse ideal of `self`.

This method calls `self.number_field.integer_ring`.

4.9.4.2 `issubideal` – Check subideal

`issubideal(self, other: Ideal) → Ideal`

Check `self` is subideal of `other`.

4.9.4.3 `issuperideal` – Check superideal

`issuperideal(self, other: Ideal) → Ideal`

Check `self` is superideal of `other`.

4.9.4.4 `gcd` – greatest common divisor

`gcd(self, other: Ideal) → Ideal`

Return the greatest common divisor(gcd) of `self` and `other` as ideal.

This method simply executes `self+other`.

4.9.4.5 `lcm` – least common multiplier

`lcm(self, other: Ideal) → Ideal`

Return the least common multiplier(lcm) of `self` and `other` as ideal.

This method simply calls the method `intersect`.

4.9.4.6 norm – norm

`norm(self)` → *rational.Rational*

Return the norm of `self`.

This method calls `self.number_field.integer_ring`.

4.9.4.7 isIntegral – Check integral

`isIntegral(self)` → *True/False*

Determine whether `self` is an integral ideal or not.

Examples

```
>>> M = module.Ideal([matrix.RingMatrix(2, 2, [1,0]+[0,2]), 2], F)
>>> print M.inverse()
([-2L, 0L]+[0L, 2L], 1L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
 NumberField([2, 0, 1]))
>>> print M * M.inverse()
([1L, 0L]+[0L, 1L], 1L)
over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
 NumberField([2, 0, 1]))
>>> M.norm()
Rational(1, 2)
>>> M.isIntegral()
False
```


4.9.5 Ideal_with_generator - ideal with generator

Initialize (Constructor)

Ideal_with_generator(generator: list) → Ideal_with_generator

Create a new ideal given as a generator.

generator is a list of instances in **BasicAlgNumber**, which represent generators, over a same number field.

Attribute

generator : generators of the ideal

number_field : the number field over which generators are defined

Operations

operator	explanation
M==N	Return whether M and N are equal or not as module.
c in M	Check whether some element of M equals c.
M+N	Return the sum of M and N as ideal with generators.
M*N	Return the product of M and N as ideal with generators.
M**c	Return M to c based on the ideal multiplication.
repr(M)	Return the repr string of the ideal M.
str(M)	Return the str string of the ideal M.

Examples

```
>>> F = algfield.NumberField([2,0,1])
>>> M_1 = module.Ideal_with_generator([
    F.createElement([[1,0], 2]), F.createElement([[0,1], 1])
])
>>> M_2 = module.Ideal_with_generator([
    F.createElement([[2,0], 3]), F.createElement([[0,5], 3])
])
>>> print M_1
[BasicAlgNumber([[1, 0], 2], [2, 0, 1]), BasicAlgNumber([[0, 1], 1], [2, 0, 1])]
>>> print M_1 + M_2
[BasicAlgNumber([[1, 0], 2], [2, 0, 1]), BasicAlgNumber([[0, 1], 1], [2, 0, 1]),
```

```

    BasicAlgNumber([[2, 0], 3], [2, 0, 1]), BasicAlgNumber([[0, 5], 3], [2, 0, 1]])
>>> print M_1 * M_2
[BasicAlgNumber([[1L, 0L], 3L], [2, 0, 1]), BasicAlgNumber([[0L, 5L], 6], [2, 0, 1]),
BasicAlgNumber([[0L, 2L], 3], [2, 0, 1]), BasicAlgNumber([[-10L, 0L], 3], [2, 0, 1])]
>>> print M_1 ** 2
[BasicAlgNumber([[1L, 0L], 4], [2, 0, 1]), BasicAlgNumber([[0L, 1L], 2], [2, 0, 1]),
BasicAlgNumber([[0L, 1L], 2], [2, 0, 1]), BasicAlgNumber([[-2L, 0L], 1], [2, 0, 1])]

```

Methods

4.9.5.1 copy - create copy

`copy(self) → Ideal_with_generator`

Create copy of `self`.

4.9.5.2 to_HNFRepresentation - change to ideal with HNF

`to_HNFRepresentation(self) → Ideal`

Transform `self` to the corresponding ideal as HNF(hermite normal form) representation.

4.9.5.3 twoElementRepresentation - Represent with two element

`twoElementRepresentation(self) → Ideal_with_generator`

Transform `self` to the corresponding ideal as HNF(hermite normal form) representation.

If `self` is not a prime ideal, this method is not efficient.

4.9.5.4 smallest_rational - a Z-generator in the rational field

`smallest_rational(self) → rational.Rational`

Return the **Z**-generator of intersection of the module `self` and the rational field.

This method calls **to_HNFRepresentation**.

4.9.5.5 inverse – inverse

`inverse(self) → Ideal`

Return the inverse ideal of `self`.

This method calls `to_HNFRepresentation`.

4.9.5.6 `norm` – `norm`

`norm(self) → rational.Rational`

Return the norm of `self`.

This method calls `to_HNFRepresentation`.

4.9.5.7 `intersect` - `intersection`

`intersect(self, other: Ideal_with_generator) → Ideal`

Return intersection of `self` and `other`.

This method calls `to_HNFRepresentation`.

4.9.5.8 `issubideal` – Check subideal

`issubideal(self, other: Ideal_with_generator) → Ideal`

Check `self` is subideal of `other`.

This method calls `to_HNFRepresentation`.

4.9.5.9 `issuperideal` – Check superideal

`issuperideal(self, other: Ideal_with_generator) → Ideal`

This method calls `to_HNFRepresentation`.

Examples

```
>>> M = module.Ideal_with_generator([
F.createElement([[2,0], 3]), F.createElement([[0,2], 3]), F.createElement([[1,0], 3])
])
>>> print M.to_HNFRepresentation()
([2L, 0L, 0L, -4L, 1L, 0L]+[0L, 2L, 2L, 0L, 0L, 1L], 3L)
over
([1L, 0L]+[0L, 1L], NumberField([2, 0, 1]))
>>> print M.twoElementRepresentation()
[BasicAlgNumber([[1L, 0], 3], [2, 0, 1]), BasicAlgNumber([[3, 2], 3], [2, 0, 1])]
>>> M.norm()
Rational(1, 9)
```

4.10 permute – permutation (symmetric) group

- Classes
 - **Permute**
 - **ExPermute**
 - **PermGroup**

4.10.1 Permute – element of permutation group

Initialize (Constructor)

Permute(value: *list/tuple*, key: *list/tuple*) → **Permute**

Permute(val_key: *dict*) → **Permute**

Permute(value: *list/tuple*, key: *int=*None) → **Permute**

Create an element of a permutation group.

An instance will be generated with “normal” way. That is, we input a **key**, which is a list of (indexed) all elements from some set, and a **value**, which is a list of all permuted elements.

Normally, you input two lists (or tuples) **value** and **key** with same length. Or you can input **val_key** as a dict whose **values()** is a list “value” and **keys()** is a list “key” in the sense of above. Also, there are some short-cut for inputting **key**:

- If key is $[1, 2, \dots, N]$, you do not have to input **key**.
- If key is $[0, 1, \dots, N]$, input 0 as **key**.
- If key equals the list arranged through **value** in ascending order, input 1.
- If key equals the list arranged through **value** in descending order, input -1 .

Attribute

key :

It expresses **key**.

data :

†It expresses indexed form of **value**.

Operations

operator	explanation
<code>A==B</code>	Check equality for A's value and B's one and A's key and B's one.
<code>A*B</code>	right multiplication (that is, $A \circ B$ with normal mapping way)
<code>A/B</code>	division (that is, $A \circ B^{-1}$)
<code>A**B</code>	powering
<code>A.inverse()</code>	inverse
<code>A[c]</code>	the element of value corresponding to c in key
<code>A(lst)</code>	permute lst with A

Examples

```
>>> p1 = permute.Permute(['b','c','d','a','e'], ['a','b','c','d','e'])
>>> print p1
['a', 'b', 'c', 'd', 'e'] -> ['b', 'c', 'd', 'a', 'e']
>>> p2 = permute.Permute([2, 3, 0, 1, 4], 0)
>>> print p2
[0, 1, 2, 3, 4] -> [2, 3, 0, 1, 4]
>>> p3 = permute.Permute(['c','a','b','e','d'], 1)
>>> print p3
['a', 'b', 'c', 'd', 'e'] -> ['c', 'a', 'b', 'e', 'd']
>>> print p1 * p3
['a', 'b', 'c', 'd', 'e'] -> ['d', 'b', 'c', 'e', 'a']
>>> print p3 * p1
['a', 'b', 'c', 'd', 'e'] -> ['a', 'b', 'e', 'c', 'd']
>>> print p1 ** 4
['a', 'b', 'c', 'd', 'e'] -> ['a', 'b', 'c', 'd', 'e']
>>> p1['d']
'a'
>>> p2([0, 1, 2, 3, 4])
[2, 3, 0, 1, 4]
```


Methods

4.10.1.1 setKey – change key

`setKey(self, key: list/tuple) → Permute`

Set other key.

key must be list or tuple with same length to **key**.

4.10.1.2 getValue – get “value”

`getValue(self) → list`

Return (not data) value of `self`.

4.10.1.3 getGroup – get PermGroup

`getGroup(self) → PermGroup`

Return **PermGroup** to which `self` belongs.

4.10.1.4 numbering – give the index

`numbering(self) → int`

Number `self` in the permutation group. (Slow method)

The numbering is made to fit the following inductive definition for dimension of the permutation group.

If numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, \sigma_{n-1}]$ on $(n-1)$ -dimension is k , numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, \sigma_{n-1}, n]$ on n -dimension is k and numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, n, \sigma_{n-1}]$ on n -dimension is $k + (n-1)!$, and so on. (See [Room of Points And Lines, part 2, section 15, paragraph 2 \(Japanese\)](#))

4.10.1.5 order – order of the element

`order(self) → int/long`

Return order as the element of group.

This method is faster than general group method.

4.10.1.6 ToTranspose – represent as transpositions

ToTranspose(self) → ExPermute

Represent **self** as a composition of transpositions.

Return the element of **ExPermute** with transpose (2-dimensional cyclic) type. It is recursive program, and it would take more time than the method **ToCyclic**.

4.10.1.7 ToCyclic – corresponding ExPermute element

ToCyclic(self) → ExPermute

Represent **self** as a composition of cyclic representations.

Return the element of **ExPermute**. †This method decomposes **self** into coprime cyclic permutations, so each cyclic is commutative.

4.10.1.8 sgn – sign of the permutation

sgn(self) → int

Return the sign of permutation group element.

If **self** is even permutation, that is, **self** can be written as a composition of an even number of transpositions, it returns 1. Otherwise, that is, for odd permutation, it returns -1.

4.10.1.9 types – type of cyclic representation

types(self) → str

Return cyclic type defined by each cyclic permutation element length.

4.10.1.10 ToMatrix – permutation matrix

ToMatrix(self) → **Matrix**

Return permutation matrix.

The row and column correspond to **key**. If **self** G satisfies $G[a] = b$, then (a, b) -element of the matrix is 1. Otherwise, the element is 0.

Examples

```
>>> p = Permute([2,3,1,5,4])
>>> p.numbering()
28
>>> p.order()
6
>>> p.ToTranspose()
[(4,5)(1,3)(1,2)](5)
>>> p.sgn()
-1
>>> p.ToCyclic()
[(1,2,3)(4,5)](5)
>>> p.types()
'(2,3)type'
>>> print p.ToMatrix()
0 1 0 0 0
0 0 1 0 0
1 0 0 0 0
0 0 0 0 1
0 0 0 1 0
```

4.10.2 ExPermute – element of permutation group as cyclic representation

Initialize (Constructor)

ExPermute(dim: *int*, value: *list*, key: *list*=None) → ExPermute

Create an element of a permutation group.

An instance will be generated with “cyclic” way. That is, we input a **key**, which is a list of tuples and each tuple expresses a cyclic permutation. For example, $(\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_k)$ is one-to-one mapping, $\sigma_1 \mapsto \sigma_2, \sigma_2 \mapsto \sigma_3, \dots, \sigma_k \mapsto \sigma_1$.

dim must be positive integer, that is, an instance of `int`, `long` or `.` **key** should be a list whose length equals **dim**. Input a list of tuples whose elements are in **key** as **value**. Note that you can abbreviate **key** if **key** has the form $[1, 2, \dots, N]$. Also, you can input 0 as **key** if **key** has the form $[0, 2, \dots, N - 1]$.

Attribute

dim:

It expresses **dim**.

key :

It expresses **key**.

data :

†It expresses indexed form of **value**.

Operations

operator	explanation
A==B	Check equality for A’s value and B’s one and A’s key and B’s one.
A*B	right multiplication (that is, $A \circ B$ with normal mapping way)
A/B	division (that is, $A \circ B^{-1}$)
A**B	powering
A.inverse()	inverse
A[c]	the element of value corresponding to c in key
A(1st)	permute 1st with A
str(A)	simple representation. use simplify .
repr(A)	representation

Examples

```
>>> p1 = permute.ExPermute(5, [('a', 'b')], ['a','b','c','d','e'])
>>> print p1
[('a', 'b')] <['a', 'b', 'c', 'd', 'e']>
>>> p2 = permute.ExPermute(5, [(0, 2), (3, 4, 1)], 0)
>>> print p2
[(0, 2), (1, 3, 4)] <[0, 1, 2, 3, 4]>
>>> p3 = permute.ExPermute(5, [('b','c')], ['a','b','c','d','e'])
>>> print p1 * p3
[('a', 'b'), ('b', 'c')] <['a', 'b', 'c', 'd', 'e']>
>>> print p3 * p1
[('b', 'c'), ('a', 'b')] <['a', 'b', 'c', 'd', 'e']>
>>> p1['c']
'c'
>>> p2([0, 1, 2, 3, 4])
[2, 4, 0, 1, 3]
```

Methods

4.10.2.1 setKey – change key

`setKey(self, key: list) → ExPermute`

Set other key.

key must be a list whose length equals **dim**.

4.10.2.2 getValue – get “value”

`getValue(self) → list`

Return (not data) value of `self`.

4.10.2.3 getGroup – get PermGroup

`getGroup(self) → PermGroup`

Return **PermGroup** to which `self` belongs.

4.10.2.4 order – order of the element

`order(self) → int/long`

Return order as the element of group.

This method is faster than general group method.

4.10.2.5 ToNormal – represent as normal style

`ToNormal(self) → Permute`

Represent `self` as an instance of **Permute**.

4.10.2.6 `simplify` – use simple value

`simplify(self) → ExPermute`

Return the more simple cyclic element.

†This method uses **ToNormal** and **ToCyclic**.

4.10.2.7 `sgn` – sign of the permutation

`sgn(self) → int`

Return the sign of permutation group element.

If `self` is even permutation, that is, `self` can be written as a composition of an even number of transpositions, it returns 1. Otherwise, that is, for odd permutation, it returns -1 .

Examples

```
>>> p = permute.ExPermute(5, [(1, 2, 3), (4, 5)])
>>> p.order()
6
>>> print p.ToNormal()
[1, 2, 3, 4, 5] -> [2, 3, 1, 5, 4]
>>> p * p
[(1, 2, 3), (4, 5), (1, 2, 3), (4, 5)] <[1, 2, 3, 4, 5]>
>>> (p * p).simplify()
[(1, 3, 2)] <[1, 2, 3, 4, 5]>
```

4.10.3 PermGroup – permutation group

Initialize (Constructor)

PermGroup(key: *int/long*) → PermGroup

PermGroup(key: *list/tuple*) → PermGroup

Create a permutation group.

Normally, input list as **key**. If you input some integer N , **key** is set as $[1, 2, \dots, N]$.

Attribute

key :
It expresses **key**.

Operations

operator	explanation
A==B	Check equality for A's value and B's one and A's key and B's one.
card(A)	same as grouporder
str(A)	simple representation
repr(A)	representation

Examples

```
>>> p1 = permute.PermGroup(['a','b','c','d','e'])
>>> print p1
['a','b','c','d','e']
>>> card(p1)
120L
```


Methods

4.10.3.1 createElement – create an element from seed

`createElement(self, seed: list/tuple/dict) → Permute`
`createElement(self, seed: list) → ExPermute`

Create new element in `self`.

`seed` must be the form of “value” on **Permute** or **ExPermute**

4.10.3.2 identity – group identity

`identity(self) → Permute`

Return the identity of `self` as normal type.

For cyclic type, use **identity_c**.

4.10.3.3 identity_c – group identity as cyclic type

`identity_c(self) → ExPermute`

Return permutation group identity as cyclic type.

For normal type, use **identity**.

4.10.3.4 grouporder – order as group

`grouporder(self) → int/long`

Compute the order of `self` as group.

4.10.3.5 randElement – random permute element

`randElement(self) → Permute`

Create random new element as normal type in `self`.

Examples

```
>>> p = permute.PermGroup(5)
>>> print p.createElement([3, 4, 5, 1, 2])
[1, 2, 3, 4, 5] -> [3, 4, 5, 1, 2]
>>> print p.createElement([(1, 2), (3, 4)])
[(1, 2), (3, 4)] <[1, 2, 3, 4, 5]>
>>> print p.identity()
[1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
>>> print p.identity_c()
[] <[1, 2, 3, 4, 5]>
>>> p.grouporder()
120L
>>> print p.randElement()
[1, 2, 3, 4, 5] -> [3, 4, 5, 2, 1]
```

4.11 rational – integer and rational number

rational module provides integer and rational numbers, as class Rational, Integer, RationalField, and IntegerRing.

- **Classes**
 - **Integer**
 - **IntegerRing**
 - **Rational**
 - **RationalField**

This module also provides following constants:

theIntegerRing :

theIntegerRing is represents the ring of rational integers. An instance of **IntegerRing**.

theRationalField :

theRationalField is represents the field of rational numbers. An instance of **RationalField**.

4.11.1 Integer – integer

Integer is a class of integer. Since 'int' and 'long' do not return rational for division, it is needed to create a new class.

This class is a subclass of **CommutativeRingElement** and long.

Initialize (Constructor)

Integer(integer: *integer*) → *Integer*

Construct a Integer object. If argument is omitted, the value becomes 0.

Methods

4.11.1.1 `getRing` – get ring object

`getRing(self) → IntegerRing`

Return an IntegerRing object.

4.11.1.2 `actAdditive` – addition of binary addition chain

`actAdditive(self, other: integer) → Integer`

Act on other additively, i.e. `n` is expanded to `n` time additions of `other`. Naively, it is:

```
return sum([+other for _ in range(self)])
```

but, here we use a binary addition chain.

4.11.1.3 `actMultiplicative` – multiplication of binary addition chain

`actMultiplicative(self, other: integer) → Integer`

Act on other multiplicatively, i.e. `n` is expanded to `n` time multiplications of `other`. Naively, it is:

```
return reduce(lambda x,y: x*y, [+other for _ in range(self)])
```

but, here we use a binary addition chain.

4.11.2 IntegerRing – integer ring

The class is for the ring of rational integers.

This class is a subclass of **CommutativeRing**.

Initialize (Constructor)

IntegerRing() \rightarrow *IntegerRing*

Create an instance of IntegerRing. You may not want to create an instance, since there is already theIntegerRing.

Attribute

zero :

It expresses the additive unit 0. (read only)

one :

It expresses the multiplicative unit 1. (read only)

Operations

operator	explanation
x in Z	return whether an element is in or not.
repr(Z)	return representation string.
str(Z)	return string.

Methods

4.11.2.1 createElement – create Integer object

createElement(self, seed: *integer*) → *Integer*

Return an Integer object with **seed**.
seed must be int, long or rational.Integer.

4.11.2.2 gcd – greatest common divisor

gcd(self, n: *integer*, m: *integer*) → *Integer*

Return the greatest common divisor of given 2 integers.

4.11.2.3 extgcd – extended GCD

extgcd(self, n: *integer*, m: *integer*) → *Integer*

Return a tuple (u, v, d) ; they are the greatest common divisor d of two given integers **n** and **m** and u, v such that $d = nu + mv$.

4.11.2.4 lcm – lowest common multiplier

lcm(self, n: *integer*, m: *integer*) → *Integer*

Return the lowest common multiple of given 2 integers. If both are zero, it raises an exception.

4.11.2.5 getQuotientField – get rational field object

getQuotientField(self) → *RationalField*

Return the rational field (**RationalField**).

4.11.2.6 issubring – subring test

issubring(self, other: **Ring**) → *bool*

Report whether another ring contains the integer ring as subring.

If other is also the integer ring, the output is True. In other cases it depends on the implementation of another ring's `issuperring` method.

4.11.2.7 `issuperring` – superring test

`issuperring(self, other: Ring) → bool`

Report whether the integer ring contains another ring as subring.

If other is also the integer ring, the output is True. In other cases it depends on the implementation of another ring's `issubring` method.

4.11.3 Rational – rational number

The class of rational numbers.

Initialize (Constructor)

```
Rational(numerator: numbers, denominator: numbers=1)  
    → Integer
```

Construct a rational number from:

- integers,
- float, or
- Rational.

Other objects can be converted if they have `toRational` methods. Otherwise raise `TypeError`.

Methods

4.11.3.1 `getRing` – get ring object

`getRing(self)` → *RationalField*

Return a `RationalField` object.

4.11.3.2 `decimalString` – represent decimal

`decimalString(self, N: integer)` → *string*

Return a string of the number to `N` decimal places.

4.11.3.3 `expand` – continued-fraction representation

`expand(self, base: integer, limit: integer)` → *string*

Return the nearest rational number whose denominator is a power of `base` and at most `limit` if `base` is positive integer.

Otherwise, i.e. `base=0`, returns the nearest rational number whose denominator is at most `limit`.

`base` must be non-negative integer.

4.11.4 RationalField – the rational field

RationalField is a class of field of rationals. The class has the single instance **theRationalField**.

This class is a subclass of **QuotientField**.

Initialize (Constructor)

RationalField() \rightarrow *RationalField*

Create an instance of RationalField. You may not want to create an instance, since there is already theRationalField.

Attribute

zero :

It expresses the additive unit 0, namely Rational(0, 1). (read only)

one :

It expresses the multiplicative unit 1, namely Rational(1, 1). (read only)

Operations

operator	explanation
x in Q	return whether an element is in or not.
str(Q)	return string.

Methods

4.11.4.1 createElement – create Rational object

```
createElement(self, numerator: integer or Rational, denominator: integer=1 )  
    → Rational
```

Create a Rational object.

4.11.4.2 classNumber – get class number

```
classNumber(self) → integer
```

Return 1, since the class number of the rational field is one.

4.11.4.3 getQuotientField – get rational field object

```
getQuotientField(self) → RationalField
```

Return the rational field itself.

4.11.4.4 issubring – subring test

```
issubring(self, other: Ring) → bool
```

Report whether another ring contains the rational field as subring.

If other is also the rational field, the output is True. In other cases it depends on the implementation of another ring's issuperring method.

4.11.4.5 issuperring – superring test

```
issuperring(self, other: Ring) → bool
```

Report whether the rational field contains another ring as subring.

If other is also the rational field, the output is True. In other cases it depends on the implementation of another ring's issubring method.

4.12 real – real numbers and its functions

The module `real` provides arbitrary precision real numbers and their utilities. The functions provided are corresponding to the `math` standard module.

- **Classes**

- `RealField`
- `Real`
- `†Constant`
- `†ExponentialPowerSeries`
- `†AbsoluteError`
- `†RelativeError`

- **Functions**

- `exp`
- `sqrt`
- `log`
- `log1piter`
- `piGaussLegendre`
- `eContinuedFraction`
- `floor`
- `ceil`
- `trunc`
- `sin`
- `cos`
- `tan`
- `sinh`
- `cosh`
- `tanh`
- `asin`
- `acos`
- `atan`
- `atan2`
- `hypot`
- `pow`
- `degrees`
- `radians`

- **fabs**
- **fmod**
- **frexp**
- **ldexp**
- **EulerTransform**

This module also provides following constants:

- e** :
This constant is obsolete (Ver 1.1.0).
- pi** :
This constant is obsolete (Ver 1.1.0).
- Log2** :
This constant is obsolete (Ver 1.1.0).
- theRealField** :
theRealField is the instance of **RealField**.

4.12.1 RealField – field of real numbers

The class is for the field of real numbers. The class has the single instance **theRealField**.

This class is a subclass of **Field**.

Initialize (Constructor)

RealField() \rightarrow *RealField*

Create an instance of RealField. You may not want to create an instance, since there is already **theRealField**.

Attribute

zero :

It expresses the additive unit 0. (read only)

one :

It expresses the multiplicative unit 1. (read only)

Operations

operator	explanation
x in R	membership test; return whether an element is in or not.
repr(R)	return representation string.
str(R)	return string.

Methods

4.12.1.1 `getCharacteristic` – get characteristic

`getCharacteristic(self)` → *integer*

Return the characteristic, zero.

4.12.1.2 `issubring` – subring test

`issubring(self, aRing: Ring)` → *bool*

Report whether another ring contains the real field as subring.

4.12.1.3 `issuperring` – superring test

`issuperring(self, aRing: Ring)` → *bool*

Report whether the real field contains another ring as subring.

4.12.2 Real – a Real number

Real is a class of real number. This class is only for consistency for other **Ring** object.

This class is a subclass of **CommutativeRingElement**.

All implemented operators in this class are delegated to Float type.

Initialize (Constructor)

Real(value: *number*) \rightarrow *Real*

Construct a Real object.

value must be int, long, Float or **Rational**.

Methods

4.12.2.1 `getRing` – get ring object

`getRing(self)` \rightarrow *RealField*

Return the real field instance.

4.12.3 Constant – real number with error correction

This class is obsolete (Ver 1.1.0).

4.12.4 ExponentialPowerSeries – exponential power series

This class is obsolete (Ver 1.1.0).

4.12.5 AbsoluteError – absolute error

This class is obsolete (Ver 1.1.0).

4.12.6 RelativeError – relative error

This class is obsolete (Ver 1.1.0).

4.12.7 exp(function) – exponential value

This function is obsolete (Ver 1.1.0).

4.12.8 sqrt(function) – square root

This function is obsolete (Ver 1.1.0).

4.12.9 log(function) – logarithm

This function is obsolete (Ver 1.1.0).

4.12.10 log1piter(function) – iterator of $\log(1+x)$

log1piter(xx: *number*) → *iterator*

Return iterator for $\log(1+x)$.

4.12.11 piGaussLegendre(function) – pi by Gauss-Legendre

This function is obsolete (Ver 1.1.0).

4.12.12 eContinuedFraction(function) – Napier’s Constant by continued fraction expansion

This function is obsolete (Ver 1.1.0).

4.12.13 **floor(function)** – floor the number

floor(*x*: *number*) → *integer*

Return the biggest integer not more than *x*.

4.12.14 **ceil(function)** – ceil the number

ceil(*x*: *number*) → *integer*

Return the smallest integer not less than *x*.

4.12.15 **tranc(function)** – round-off the number

tranc(*x*: *number*) → *integer*

Return the number of rounded off *x*.

4.12.16 **sin(function)** – sine function

This function is obsolete (Ver 1.1.0).

4.12.17 **cos(function)** – cosine function

This function is obsolete (Ver 1.1.0).

4.12.18 **tan(function)** – tangent function

This function is obsolete (Ver 1.1.0).

4.12.19 **sinh(function)** – hyperbolic sine function

This function is obsolete (Ver 1.1.0).

4.12.20 **cosh(function)** – hyperbolic cosine function

This function is obsolete (Ver 1.1.0).

4.12.21 **tanh(function)** – hyperbolic tangent function

This function is obsolete (Ver 1.1.0).

4.12.22 `asin(function)` – arc sine function

This function is obsolete (Ver 1.1.0).

4.12.23 `acos(function)` – arc cosine function

This function is obsolete (Ver 1.1.0).

4.12.24 `atan(function)` – arc tangent function

This function is obsolete (Ver 1.1.0).

4.12.25 `atan2(function)` – arc tangent function

This function is obsolete (Ver 1.1.0).

4.12.26 `hypot(function)` – Euclidean distance function

This function is obsolete (Ver 1.1.0).

4.12.27 `pow(function)` – power function

This function is obsolete (Ver 1.1.0).

4.12.28 `degrees(function)` – convert angle to degree

This function is obsolete (Ver 1.1.0).

4.12.29 `radians(function)` – convert angle to radian

This function is obsolete (Ver 1.1.0).

4.12.30 `fabs(function)` – absolute value

`fabs(x: number) → number`

Return absolute value of `x`

4.12.31 `fmod(function)` – modulo function over real

`fmod(x: number, y: number) → number`

Return $x - ny$, where `n` is the quotient of `x` / `y`, rounded towards zero to an integer.

4.12.32 **frexp(function) – expression with base and binary exponent**

frexp(*x*: *number*) → (*m*,*e*)

Return a tuple (*m*,*e*), where $x = m \times 2^e$, $1/2 \leq \text{abs}(m) < 1$ and *e* is an integer.

†This function is provided as the counter-part of `math.frexp`, but it might not be useful.

4.12.33 **ldexp(function) – construct number from base and binary exponent**

ldexp(*x*: *number*, *i*: *number*) → *number*

Return $x \times 2^i$.

4.12.34 **EulerTransform(function) – iterator yields terms of Euler transform**

EulerTransform(*iterator*: *iterator*) → *iterator*

Return an iterator which yields terms of Euler transform of the given *iterator*.

†

4.13 ring – for ring object

- **Classes**
 - **Ring**
 - **CommutativeRing**
 - **Field**
 - **QuotientField**
 - **RingElement**
 - **CommutativeRingElement**
 - **FieldElement**
 - **QuotientFieldElement**
 - **Ideal**
 - **ResidueClassRing**
 - **ResidueClass**
 - **CommutativeRingProperties**
- **Functions**
 - **getRingInstance**
 - **getRing**
 - **inverse**
 - **exact_division**

4.13.1 †Ring – abstract ring

Ring is an abstract class which expresses that the derived classes are (in mathematical meaning) rings.

Definition of ring (in mathematical meaning) is as follows: Ring is a structure with addition and multiplication. It is an abelian group with addition, and a monoid with multiplication. The multiplication obeys the distributive law.

This class is abstract and cannot be instantiated.

Attribute

zero additive unit

one multiplicative unit

Operations

operator	explanation
A==B	Return whether M and N are equal or not.

Methods

4.13.1.1 createElement – create an element

createElement(self, seed: (*undefined*)) → *RingElement*

Return an element of the ring with seed.

This is an abstract method.

4.13.1.2 getCharacteristic – characteristic as ring

getCharacteristic(self) → *integer*

Return the characteristic of the ring.

The Characteristic of a ring is the smallest positive integer n s.t. $na = 0$ for any element a of the ring, or 0 if there is no such natural number.
This is an abstract method.

4.13.1.3 issubring – check subring

issubring(self, other: *RingElement*) → *True/False*

Report whether another ring contains the ring as a subring.

This is an abstract method.

4.13.1.4 issuperring – check superring

issuperring(self, other: *RingElement*) → *True/False*

Report whether the ring is a superring of another ring.

This is an abstract method.

4.13.1.5 getCommonSuperring – get common ring

`getCommonSuperring(self, other: RingElement) → RingElement`

Return common super ring of self and another ring.

This method uses **issubring**, **issuperring**.

4.13.2 †CommutativeRing – abstract commutative ring

CommutativeRing is an abstract subclass of **Ring** whose multiplication is commutative.

CommutativeRing is subclass of **Ring**.

There are some properties of commutative rings, algorithms should be chosen accordingly. To express such properties, there is a class **CommutativeRingProperties**.

This class is abstract and cannot be instantiated.

Attribute

properties an instance of **CommutativeRingProperties**

Methods

4.13.2.1 `getQuotientField` – create quotient field

`getQuotientField(self)` → *QuotientField*

Return the quotient field of the ring.

This is an abstract method.

If quotient field of `self` is not available, it should raise exception.

4.13.2.2 `isdomain` – check domain

`isdomain(self)` → *True/False/None*

Return True if the ring is actually a domain, False if not, or None if uncertain.

4.13.2.3 `isnoetherian` – check Noetherian domain

`isnoetherian(self)` → *True/False/None*

Return True if the ring is actually a Noetherian domain, False if not, or None if uncertain.

4.13.2.4 `isufd` – check UFD

`isufd(self)` → *True/False/None*

Return True if the ring is actually a unique factorization domain (UFD), False if not, or None if uncertain.

4.13.2.5 `ispid` – check PID

`ispid(self)` → *True/False/None*

Return True if the ring is actually a principal ideal domain (PID), False if not, or None if uncertain.

4.13.2.6 iseuclidean – check Euclidean domain

iseuclidean(self) → *True/False/None*

Return True if the ring is actually a Euclidean domain, False if not, or None if uncertain.

4.13.2.7 isfield – check field

isfield(self) → *True/False/None*

Return True if the ring is actually a field, False if not, or None if uncertain.

4.13.2.8 registerModuleAction – register action as ring

registerModuleAction(self, action_ring: *RingElement*, action: *function*)
→ (*None*)

Register a ring `action_ring`, which act on the ring through `action` so the ring be an `action_ring` module.

See **hasaction**, **getaction**.

4.13.2.9 hasaction – check if the action has

hasaction(self, action_ring: *RingElement*) → *True/False*

Return True if `action_ring` is registered to provide action.

See **registerModuleAction**, **getaction**.

4.13.2.10 getaction – get the registered action

hasaction(self, action_ring: *RingElement*) → *function*

Return the registered action for `action_ring`.

See **registerModuleAction**, **hasaction**.

4.13.3 †Field – abstract field

Field is an abstract class which expresses that the derived classes are (in mathematical meaning) fields, i.e., a commutative ring whose multiplicative monoid is a group.

Field is subclass of **CommutativeRing**. **getQuotientField** and **isfield** are not abstract (trivial methods).

This class is abstract and cannot be instantiated.

Methods

4.13.3.1 gcd – gcd

`gcd(self, a: FieldElement, b: FieldElement) → FieldElement`

Return the greatest common divisor of `a` and `b`.

A field is trivially a UFD and should provide `gcd`. If we can implement an algorithm for computing `gcd` in an Euclidean domain, we should provide the method corresponding to the algorithm.

4.13.4 †QuotientField – abstract quotient field

QuotientField is an abstract class which expresses that the derived classes are (in mathematical meaning) quotient fields.

`self` is the quotient field of `domain`.

QuotientField is subclass of **Field**.

In the initialize step, it registers trivial action named as `baseaction`; i.e. it expresses that an element of a domain acts an element of the quotient field by using the multiplication in the domain.

This class is abstract and cannot be instantiated.

Attribute

basedomain domain which generates the quotient field `self`

4.13.5 †**RingElement** – abstract element of ring

RingElement is an abstract class for elements of rings.

This class is abstract and cannot be instantiated.

Operations

operator	explanation
A==B	equality (abstract)

Methods

4.13.5.1 `getRing` – `getRing`

`getRing(self) → Ring`

Return an object of a subclass of `Ring`, to which the element belongs.

This is an abstract method.

4.13.6 †CommutativeRingElement – abstract element of commutative ring

CommutativeRingElement is an abstract class for elements of commutative rings.

This class is abstract and cannot be instantiated.

Methods

4.13.6.1 `mul_module_action` – apply a module action

`mul_module_action(self, other: RingElement) → (undefined)`

Return the result of a module action. `other` must be in one of the action rings of `self`'s ring.

This method uses `getRing`, `CommutativeRing` and `getaction`. We should consider that the method is abstract.

4.13.6.2 `exact_division` – division exactly

`exact_division(self, other: CommutativeRingElement)
→ CommutativeRingElement`

In UFD, if `other` divides `self`, return the quotient as a UFD element.

The main difference with `/` is that `/` may return the quotient as an element of quotient field.

Simple cases:

- in a Euclidean domain, if remainder of euclidean division is zero, the division `//` is exact.
- in a field, there's no difference with `/`.

If `other` doesn't divide `self`, raise `ValueError`. Though `__divmod__` can be used automatically, we should consider that the method is abstract.

4.13.7 †FieldElement – abstract element of field

FieldElement is an abstract class for elements of fields.

FieldElement is subclass of **CommutativeRingElement**. **exact_division** are not abstract (trivial methods).

This class is abstract and cannot be instantiated.

4.13.8 †QuotientFieldElement – abstract element of quotient field

QuotientFieldElement class is an abstract class to be used as a super class of concrete quotient field element classes.

QuotientFieldElement is subclass of **FieldElement**.
`self` expresses $\frac{\text{numerator}}{\text{denominator}}$ in the quotient field.

This class is abstract and should not be instantiated.
`denominator` should not be 0.

Attribute

numerator numerator of `self`

denominator denominator of `self`

Operations

operator	explanation
A+B	addition
A-B	subtraction
A*B	multiplication
A**B	powering
A/B	division
-A	sign reversion (additive inversion)
inverse(A)	multiplicative inversion
A==B	equality

4.13.9 †Ideal – abstract ideal

Ideal class is an abstract class to represent the finitely generated ideals.

†Because the finitely-generatedness is not a restriction for Noetherian rings and in the most cases only Noetherian rings are used, it is general enough.

This class is abstract and should not be instantiated.
`generators` must be an element of the `aring` or a list of elements of the `aring`.
If `generators` is an element of the `aring`, we consider `self` is the principal ideal generated by `generators`.

Attribute

`ring` the ring belonged to by `self`

`generators` generators of the ideal `self`

Operations

operator	explanation
<code>I+J</code>	addition $\{i + j \mid i \in I, j \in J\}$
<code>I*J</code>	multiplication $IJ = \{\sum_{i,j} ij \mid i \in I, j \in J\}$
<code>I==J</code>	equality
<code>e in I</code>	For <code>e</code> in the ring, to which the ideal <code>I</code> belongs.

Methods

4.13.9.1 `issubset` – check subset

`issubset(self, other: Ideal) → True/False`

Report whether another ideal contains this ideal.

We should consider that the method is abstract.

4.13.9.2 `issuperset` – check superset

`issuperset(self, other: Ideal) → True/False`

Report whether this ideal contains another ideal.

We should consider that the method is abstract.

4.13.9.3 `reduce` – reduction with the ideal

`issuperset(self, other: Ideal) → True/False`

Reduce an element with the ideal to simpler representative.

This method is abstract.

4.13.10 †ResidueClassRing – abstract residue class ring

Initialize (Constructor)

```
ResidueClassRing(ring: CommutativeRing, ideal: Ideal)  
    → ResidueClassRing
```

A residue class ring R/I , where R is a commutative ring and I is its ideal.

ResidueClassRing is subclass of **CommutativeRing**.
one, **zero** are not abstract (trivial methods).

Because we assume that **ring** is Noetherian, so is **ring**.

This class is abstract and should not be instantiated.
ring should be an instance of **CommutativeRing**, and **ideal** must be an instance of **Ideal**, whose **ring** attribute points the same ring with the given **ring**.

Attribute

ring the base ring R

ideal the ideal I

Operations

operator	explanation
A==B	equality
e in A	report whether e is in the residue ring self .

4.13.11 †ResidueClass – abstract an element of residue class ring

Initialize (Constructor)

```
ResidueClass(x: CommutativeRingElement, ideal: Ideal)  
    → ResidueClass
```

Element of residue class ring $x + I$, where I is the modulus ideal and x is a representative element.

ResidueClass is subclass of **CommutativeRingElement**.

This class is abstract and should not be instantiated.
`ideal` corresponds to the ideal I .

Operations

These operations uses **reduce**.

operator	explanation
<code>x+y</code>	addition
<code>x-y</code>	subtraction
<code>x*y</code>	multiplication
<code>A==B</code>	equality

4.13.12 †CommutativeRingProperties – properties for CommutativeRingProperties

Initialize (Constructor)

CommutativeRingProperties((None)) → CommutativeRingProperties

Boolean properties of ring.

Each property can have one of three values; *True*, *False*, or *None*. Of course *True* is true and *False* is false, and *None* means that the property is not set neither directly nor indirectly.

CommutativeRingProperties class treats

- Euclidean (Euclidean domain),
- PID (Principal Ideal Domain),
- UFD (Unique Factorization Domain),
- Noetherian (Noetherian ring (domain)),
- field (Field)

Methods

4.13.12.1 isfield – check field

isfield(self) → *True/False/None*

Return True/False according to the field flag value being set, otherwise return None.

4.13.12.2 setIsfield – set field

isfield(self, value: *True/False*) → (*None*)

Set True/False value to the field flag.
Propagation:

- True → euclidean

4.13.12.3 iseclidean – check euclidean

iseclidean(self) → *True/False/None*

Return True/False according to the euclidean flag value being set, otherwise return None.

4.13.12.4 setIseclidean – set euclidean

isfield(self, value: *True/False*) → (*None*)

Set True/False value to the euclidean flag.
Propagation:

- True → PID
- False → field

4.13.12.5 ispid – check PID

ispid(self) → *True/False/None*

Return True/False according to the PID flag value being set, otherwise return None.

4.13.12.6 setIspid – set PID

ispid(self, value: *True/False*) → (*None*)

Set True/False value to the euclidean flag.
Propagation:

- True → UFD, Noetherian
- False → euclidean

4.13.12.7 isufd – check UFD

isufd(self) → *True/False/None*

Return True/False according to the UFD flag value being set, otherwise return None.

4.13.12.8 setIsufd – set UFD

isufd(self, value: *True/False*) → (*None*)

Set True/False value to the UFD flag.
Propagation:

- True → domain

- False \rightarrow PID

4.13.12.9 isnoetherian – check Noetherian

isnoetherian(self) \rightarrow *True/False/None*

Return True/False according to the Noetherian flag value being set, otherwise return None.

4.13.12.10 setIsnoetherian – set Noetherian

isnoetherian(self, value: *True/False*) \rightarrow (*None*)

Set True/False value to the Noetherian flag.
Propagation:

- True \rightarrow domain
- False \rightarrow PID

4.13.12.11 isdomain – check domain

isdomain(self) \rightarrow *True/False/None*

Return True/False according to the domain flag value being set, otherwise return None.

4.13.12.12 setIsdomain – set domain

isdomain(self, value: *True/False*) \rightarrow (*None*)

Set True/False value to the domain flag.
Propagation:

- False \rightarrow UFD, Noetherian

4.13.13 `getRingInstance(function)`

`getRingInstance(obj: RingElement) → RingElement`

Return a `RingElement` instance which equals `obj`.

Mainly for python built-in objects such as `int` or `float`.

4.13.14 `getRing(function)`

`getRing(obj: RingElement) → Ring`

Return a ring to which `obj` belongs.

Mainly for python built-in objects such as `int` or `float`.

4.13.15 `inverse(function)`

`inverse(obj: CommutativeRingElement) → QuotientFieldElement`

Return the inverse of `obj`. The inverse can be in the quotient field, if the `obj` is an element of non-field domain.

Mainly for python built-in objects such as `int` or `float`.

4.13.16 `exact_division(function)`

**`exact_division(self: RingElement, other: RingElement)
→ RingElement`**

Return the division of `self` by `other` if the division is exact.

Mainly for python built-in objects such as `int` or `float`.

Examples

```
>>> print ring.getRing(3)
Z
```

```
>>> print ring.exact_division(6, 3)
2L
```

4.14 vector – vector object and arithmetic

- **Classes**
 - **Vector**
- **Functions**
 - **innerProduct**

This module provides an exception class.

VectorSizeError : Report vector size is invalid. (Mainly for operations with two vectors.)

4.14.1 Vector – vector class

Vector is a class for vector.

Initialize (Constructor)

Vector(*compo: list*) \rightarrow *Vector*

Create Vector object from *compo*. *compo* must be a list of elements which are an integer or an instance of **RingElement**.

Attribute

compo :

It expresses component of vector.

Operations

Note that index is 1-origin, which is standard in mathematics field.

operator	explanation
u+v	Vector sum.
u-v	Vector subtraction.
A*v	Multiplication vector with matrix
a*v	or scalar multiplication.
v//a	Scalar division.
v%n	Reduction each elements of compo
-v	element negation.
u==v	equality.
u!=v	inequality.
v[i]	Return the coefficient of i-th element of Vector.
v[i] = c	Replace the coefficient of i-th element of Vector by c.
len(v)	return length of compo .
repr(v)	return representation string.
str(v)	return string of compo .

Examples

```
>>> A = vector.Vector([1, 2])
>>> A
Vector([1, 2])
>>> A.compo
[1, 2]
```

```
>>> B = vector.Vector([2, 1])
>>> A + B
Vector([3, 3])
>>> A % 2
Vector([1, 0])
>>> A[1]
1
>>> len(B)
2
```

Methods

4.14.1.1 copy – copy itself

`copy(self) → Vector`

Return copy of `self`.

4.14.1.2 set – set other compo

`set(self, compo: list) → (None)`

Substitute **compo** with `compo`.

4.14.1.3 indexOfNoneZero – first non-zero coordinate

`indexOfNoneZero(self) → integer`

Return the first index of non-zero element of `self.compo`.

†Raise `ValueError` if all elements of **compo** are zero.

4.14.1.4 toMatrix – convert to Matrix object

`toMatrix(self, as_column: bool=False) → Matrix`

Return **Matrix** object using **createMatrix** function.

If `as_column` is `True`, create the column matrix with `self`. Otherwise, create the row matrix.

Examples

```
>>> A = vector.Vector([0, 4, 5])
>>> A.indexOfNoneZero()
2
>>> print A.toMatrix()
0 4 5
>>> print A.toMatrix()
```

0
4
5

4.14.2 innerProduct(function) – inner product

innerProduct(bra: *Vector*, ket: *Vector*) → *RingElement*

Return the inner product of bra and ket.

The function supports Hermitian inner product for elements in the complex number field.

†Note that the returned value depends on type of elements.

Examples

```
>>> A = vector.Vector([1, 2, 3])
>>> B = vector.Vector([2, 1, 0])
>>> vector.innerProduct(A, B)
4
>>> C = vector.Vector([1+1j, 2+2j, 3+3j])
>>> vector.innerProduct(C, C)
(28+0j)
```


4.15 factor.ecm – ECM factorization

This module has curve type constants:

S : aka SUYAMA. Suyama’s parameter selection strategy.

B : aka BERNSTEIN. Bernstein’s parameter selection strategy.

A1 : aka ASUNCION1. Asuncion’s parameter selection strategy variant 1.

A2 : aka ASUNCION2. ditto 2.

A3 : aka ASUNCION3. ditto 3.

A4 : aka ASUNCION4. ditto 4.

A5 : aka ASUNCION5. ditto 5.

See J.S.Asuncion’s master thesis [?] for details of each family.

4.15.1 ecm – elliptic curve method

```
ecm(n: integer, curve_type: curvetype=A1, incs: integer=3, trials:
integer=20, verbose: bool=False)
    → integer
```

Find a factor of n by elliptic curve method.

If it cannot find non-trivial factor of n , then it returns 1.

`curve_type` should be chosen from **curvetype** constants above.

The second optional argument `incs` specifies a number of changes of bounds. The function repeats factorization trials several times changing curves with a fixed bounds.

Optional argument `trials` can control how quickly move on to the next higher bounds.

`verbose` toggles verbosity.

4.16 factor.find – find a factor

All methods in this module return one of a factor of given integer. If it fails to find a non-trivial factor, it returns 1. Note that 1 is a factor anyway.

`verbose` boolean flag can be specified for verbose reports. To receive these messages, you have to prepare a logger (see [logging](#)).

4.16.1 trialDivision – trial division

trialDivision(*n*: *integer*, ****options**) → *integer*

Return a factor of *n* by trial divisions.

options can be either one of the following:

1. **start** and **stop** as range parameters. In addition to these, **step** is also available.
2. **iterator** as an iterator of primes.

If **options** is not given, the function divides *n* by primes from 2 to the floor of the square root of *n* until a non-trivial factor is found.

`verbose` boolean flag can be specified for verbose reports.

4.16.2 pmom – $p - 1$ method

pmom(*n*: *integer*, ****options**) → *integer*

Return a factor of *n* by the $p - 1$ method.

The function tries to find a non-trivial factor of *n* using Algorithm 8.8.2 ($p - 1$ first stage) of [?]. In the case of $n = 2^i$, the function will not terminate. Due to the nature of the method, the method may return the trivial factor only.

`verbose` Boolean flag can be specified for verbose reports, though it is not so verbose indeed.

4.16.3 rhomethod – ρ method

rhomethod(*n*: *integer*, ****options**) → *integer*

Return a factor of *n* by Pollard's ρ method.

The implementation refers the explanation in [?]. Due to the nature of the

method, a factorization may return the trivial factor only.

`verbose` Boolean flag can be specified for verbose reports.

Examples

```
>>> factor.find.trialDivision(1001)
7
>>> factor.find.trialDivision(1001, start=10, stop=32)
11
>>> factor.find.pmom(1001)
91
>>> import logging
>>> logging.basicConfig()
>>> factor.find.rhomethod(1001, verbose=True)
INFO:nzmath.factor.find:887 748
13
```

4.17 factor.methods – factoring methods

It uses methods of **factor.find** module or some heavier methods of related modules to find a factor. Also, classes of **factor.util** module is used to track the factorization process. **options** are normally passed to the underlying function without modification.

This module uses the following type:

factorlist :

factorlist is a list which consists of pairs (**base**, **index**). Each pair means $base^{index}$. The product of these terms expresses prime factorization.

4.17.1 factor – easiest way to factor

```
factor(n: integer, method: string='default', **options )  
→ factorlist
```

Factor the given positive integer **n**.

By default, use several methods internally.

The optional argument **method** can be:

- **'ecm'**: use elliptic curve method.
- **'mpqs'**: use MPQS method.
- **'pmom'**: use $p - 1$ method.
- **'rhomethod'**: use Pollard's ρ method.
- **'trialDivision'**: use trial division.

(†In fact, the initial letter of method name suffices to specify.)

4.17.2 ecm – elliptic curve method

```
ecm(n: integer, **options ) → factorlist
```

Factor the given integer **n** by elliptic curve method.

(See **ecm** of **factor.ecm** module.)

4.17.3 mpqs – multi-polynomial quadratic sieve method

`mpqs(n: integer, **options) → factorlist`

Factor the given integer `n` by multi-polynomial quadratic sieve method.

(See `mpqsfind` of `factor.mpqs` module.)

4.17.4 pmom – $p - 1$ method

`pmom(n: integer, **options) → factorlist`

Factor the given integer `n` by $p - 1$ method.

The method may fail unless `n` has an appropriate factor for the method.
(See `pmom` of `factor.find` module.)

4.17.5 rhomethod – ρ method

`rhomethod(n: integer, **options) → factorlist`

Factor the given integer `n` by Pollard's ρ method.

The method is a probabilistic method, possibly fails in factorizations.
(See `rhomethod` of `factor.find` module.)

4.17.6 trialDivision – trial division

`trialDivision(n: integer, **options) → factorlist`

Factor the given integer `n` by trial division.

`options` for the trial sequence can be either:

1. `start` and `stop` as range parameters.
2. `iterator` as an iterator of primes.
3. `eratosthenes` as an upper bound to make prime sequence by sieve.

If none of the options above are given, the function divides `n` by primes from 2 to the floor of the square root of `n` until a non-trivial factor is found.
(See `trialDivision` of `factor.find` module.)

Examples

```
>>> factor.methods.factor(10001)
[(73, 1), (137, 1)]
>>> factor.methods.ecm(1000001)
[(101L, 1), (9901L, 1)]
```

4.18 factor.misc – miscellaneous functions related factoring

- Functions
 - **allDivisors**
 - **primeDivisors**
 - **primePowerTest**
 - **squarePart**
- Classes
 - **FactoredInteger**

4.18.1 allDivisors – all divisors

allDivisors(*n*: *integer*) → *list*

Return all factors divide *n* as a list.

4.18.2 primeDivisors – prime divisors

primeDivisors(*n*: *integer*) → *list*

Return all prime factors divide *n* as a list.

4.18.3 primePowerTest – prime power test

primePowerTest(*n*: *integer*) → (*integer*, *integer*)

Judge whether *n* is of the form p^k with a prime *p* or not. If it is true, then (*p*, *k*) will be returned, otherwise (*n*, 0).

This function is based on Algo. 1.7.5 in [?].

4.18.4 squarePart – square part

squarePart(*n*: *integer*) → *integer*

Return the largest integer whose square divides *n*.

Examples

```
>>> factor.misc.allDivisors(1001)
[1, 7, 11, 13L, 77, 91L, 143L, 1001L]
>>> factor.misc.primeDivisors(100)
[2, 5]
>>> factor.misc.primePowerTest(128)
(2, 7)
>>> factor.misc.squarePart(128)
8L
```


4.18.5 FactoredInteger – integer with its factorization

Initialize (Constructor)

```
FactoredInteger(integer: integer, factors: dict={})  
    → FactoredInteger
```

Integer with its factorization information.

If **factors** is given, it is a dict of type **prime:exponent** and the product of $prime^{exponent}$ is equal to the **integer**. Otherwise, factorization is carried out in initialization.

```
from _partial_factorization(cls, integer: integer, partial: dict)  
    → FactoredInteger
```

A class method to create a new **FactoredInteger** object from partial factorization information **partial**.

Operations

operator	explanation
F * G	multiplication (other operand can be an int)
F ** n	powering
F == G	equal
F != G	not equal
F % G	remainder (the result is an int)
F // G	same as exact_division method
str(F)	string
int(F)	convert to Python integer (forgetting factorization)

Methods

4.18.5.1 `is_divisible_by`

```
is_divisible_by(self, other: integer/FactoredInteger)  
    → bool
```

Return True if `other` divides `self`.

4.18.5.2 `exact_division`

```
exact_division(self, other: integer/FactoredInteger)  
    → FactoredInteger
```

Divide by `other`. The `other` must divide `self`.

4.18.5.3 `divisors`

```
divisors(self) → list
```

Return all divisors as a list.

4.18.5.4 `proper_divisors`

```
proper_divisors(self) → list
```

Return all proper divisors (i.e. divisors excluding 1 and `self`) as a list.

4.18.5.5 `prime_divisors`

```
prime_divisors(self) → list
```

Return all prime divisors as a list.

4.18.5.6 `square_part`

```
square_part(self, asfactored: bool=False) → integer/FactoredInteger object
```

Return the largest integer whose square divides `self`.

If an optional argument `asfactored` is true, then the result is also a **FactoredInteger object**. (default is False)

4.18.5.7 `squarefree_part`

`squarefree_part(self, asfactored: bool=False) → integer/FactoredInteger object`

Return the largest squarefree integer which divides `self`.

If an optional argument `asfactored` is true, then the result is also a **FactoredInteger object**. (default is False)

4.18.5.8 `copy`

`copy(self) → FactoredInteger object`

Return a copy of the object.

4.19 `factor.mpqqs` – MPQS

4.19.1 `mpqsfind`

`mpqsfind(n: integer, s: integer=0, f: integer=0, m: integer=0, verbose: bool=False) → integer`

Find a factor of `n` by MPQS(multiple-polynomial quadratic sieve) method.

MPQS is suitable for factorizing a large number.

Optional arguments `s` is the range of sieve, `f` is the number of factor base, and `m` is multiplier. If these are not specified, the function guesses them from `n`.

4.19.2 `mpqs`

`mpqs(n: integer, s: integer=0, f: integer=0, m: integer=0) → factorlist`

Factorize `n` by MPQS method.

Optional arguments are same as **mpqsfind**.

4.19.3 eratosthenes

eratosthenes(*n*: *integer*) → *list*

Enumerate the primes up to *n*.

4.20 factor.util – utilities for factorization

- Classes
 - **FactoringInteger**
 - **FactoringMethod**

This module uses following type:

factorlist :

factorlist is a list which consists of pairs (**base**, **index**). Each pair means $base^{index}$. The product of those terms expresses whole prime factorization.

4.20.1 FactoringInteger – keeping track of factorization

Initialize (Constructor)

FactoringInteger(number: *integer*) → *FactoringInteger*

This is the base class for factoring integers.

number is stored in the attribute **number**. The factors will be stored in the attribute **factors**, and primality of factors will be tracked in the attribute **primality**.

The given **number** must be a composite number.

Attribute

number :

The composite number.

factors :

Factors known at the time being referred.

primality :

A dictionary of primality information of known factors. **True** if the factor is prime, **False** composite, or **None** undetermined.

Methods

4.20.1.1 getNextTarget – next target

`getNextTarget(self, cond: function=None) → integer`

Return the next target which meets `cond`.

If `cond` is not specified, then the next target is a composite (or undetermined) factor of **number**.

`cond` should be a binary predicate whose arguments are base and index.
If there is no target factor, **LookupError** will be raised.

4.20.1.2 getResult – result of factorization

`getResult(self) → factors`

Return the currently known factorization of the **number**.

4.20.1.3 register – register a new factor

`register(self, divisor: integer, isprime: bool=None)
→`

Register a divisor of the **number** if the `divisor` is a true divisor of the number.

The number is divided by the `divisor` as many times as possible.

The optional argument `isprime` tells the primality of the `divisor` (default to undetermined).

4.20.1.4 sortFactors – sort factors

`sortFactors(self) →`

Sort factors list.

This affects the result of **getResult**.

Examples

```

>>> A = factor.util.FactoringInteger(100)
>>> A.getNextTarget()
100
>>> A.getResult()
[(100, 1)]
>>> A.register(5, True)
>>> A.getResult()
[(5, 2), (4, 1)]
>>> A.sortFactors()
>>> A.getResult()
[(4, 1), (5, 2)]
>>> A.primality
{4: None, 5: True}
>>> A.getNextTarget()
4

```


4.20.2 FactoringMethod – method of factorization

Initialize (Constructor)

FactoringMethod() \rightarrow *FactoringMethod*

Base class of factoring methods.

All methods defined in **factor.methods** are implemented as derived classes of this class. The method which users may call is **factor** only. Other methods are explained for future implementers of a new factoring method.

Methods

4.20.2.1 factor – do factorization

```
factor(self, number: integer, return_type: str='list', need_sort:  
bool=False )  
    → factorlist
```

Return the factorization of the given positive integer `number`.

The default returned type is a **factorlist**.

A keyword option `return_type` can be as the following:

1. 'list' for default type (**factorlist**).
2. 'tracker' for **FactoringInteger**.

Another keyword option `need_sort` is Boolean: `True` to sort the result. This should be specified with `return_type='list'`.

4.20.2.2 †continue_factor – continue factorization

```
continue_factor(self, tracker: FactoringInteger, return_type:  
str='tracker', primeq: func=primeq )  
    → FactoringInteger
```

Continue factoring of the given `tracker` and return the result of factorization.

The default returned type is **FactoringInteger**, but if `return_type` is specified as 'list' then it returns **factorlist**. The primality is judged by a function specified in `primeq` optional keyword argument, which default is **primeq**.

4.20.2.3 †find – find a factor

```
find(self, target: integer, **options ) → integer
```

Find a factor from the `target` number.

This method has to be overridden, or **factor** method should be overridden not to call this method.

4.20.2.4 †generate – generate prime factors

```
generate(self, target: integer, **options ) → integer
```

Generate prime factors of the `target` number with their valuations.

The method may terminate with yielding $(1, 1)$ to indicate the factorization is incomplete.
This method has to be overridden, or **factor** method should be overridden not to call this method.

4.21 poly.factor – polynomial factorization

The factor module is for factorizations of integer coefficient univariate polynomials.

This module using following type:

polynomial :

`polynomial` is the polynomial generated by function `poly.uniutil.polynomial`.

4.21.1 brute_force_search – search factorization by brute force

```
brute_force_search(f: poly.uniutil.IntegerPolynomial, fp_factors:  
list, q: integer)  
→ [factors]
```

Find the factorization of `f` by searching a factor which is a product of some combination in `fp_factors`. The combination is searched by brute force.

The argument `fp_factors` is a list of `poly.uniutil.FinitePrimeFieldPolynomial`.

4.21.2 divisibility_test – divisibility test

```
divisibility_test(f: polynomial, g: polynomial) → bool
```

Return Boolean value whether `f` is divisible by `g` or not, for polynomials.

4.21.3 minimum_absolute_injection – send coefficients to minimum absolute representation

```
minimum_absolute_injection(f: polynomial) → F
```

Return an integer coefficient polynomial `F` by injection of a $\mathbf{Z}/p\mathbf{Z}$ coefficient polynomial `f` with sending each coefficient to minimum absolute representatives.

The coefficient ring of given polynomial `f` must be **IntegerResidueClassRing** or **FinitePrimeField**.

4.21.4 padic_factorization – p-adic factorization

```
padic_factorization(f: polynomial) → p, factors
```

Return a prime p and a p -adic factorization of given integer coefficient square-free polynomial f . The result **factors** have integer coefficients, injected from \mathbb{F}_p to its minimum absolute representation.

†The prime is chosen to be:

1. f is still squarefree mod p ,
2. the number of factors is not greater than with the successive prime.

The given polynomial f must be `poly.uniutil.IntegerPolynomial`.

4.21.5 `upper_bound_of_coefficient` – Landau-Mignotte bound of coefficients

`upper_bound_of_coefficient(f: polynomial) → long`

Compute Landau-Mignotte bound of coefficients of factors, whose degree is no greater than half of the given f .

The given polynomial f must have integer coefficients.

4.21.6 `zassenhaus` – squarefree integer polynomial factorization by Zassenhaus method

`zassenhaus(f: polynomial) → list of factors f`

Factor a squarefree integer coefficient polynomial f with Berlekamp-Zassenhaus method.

4.21.7 `integerpolynomialfactorization` – Integer polynomial factorization

`integerpolynomialfactorization(f: polynomial) → factor`

Factor an integer coefficient polynomial f with Berlekamp-Zassenhaus method.

factor output by the form of list of tuples that formed (factor, index).

4.22 poly.formalsum – formal sum

- Classes

- †**FormalSumContainerInterface**
- **DictFormalSum**
- †**ListFormalSum**

The formal sum is mathematically a finite sum of terms, A term consists of two parts: coefficient and base. All coefficients in a formal sum are in a common ring, while bases are arbitrary.

Two formal sums can be added in the following way. If there are terms with common base, they are fused into a new term with the same base and coefficients added.

A coefficient can be looked up from the base. If the specified base does not appear in the formal sum, it is null.

We refer the following for convenience as **terminit**:

terminit :

terminit means one of types to initialize **dict**. The dictionary constructed from it will be considered as a mapping from bases to coefficients.

Note for beginner You may need USE only **DictFormalSum**, but may have to READ the description of **FormalSumContainerInterface** because interface (all method names and their semantics) is defined in it.

4.22.1 FormalSumContainerInterface – interface class

Initialize (Constructor)

Since the interface is an abstract class, do not instantiate.

The interface defines what “formal sum” is. Derived classes must provide the following operations and methods.

Operations

operator	explanation
$f + g$	addition
$f - g$	subtraction
$-f$	negation
$+f$	new copy
$f * a, a * f$	multiplication by scalar a
$f == g$	equality
$f != g$	inequality
$f[b]$	get coefficient corresponding to a base b
$b \text{ in } f$	return whether base b is in f
$\text{len}(f)$	number of terms
$\text{hash}(f)$	hash

Methods

4.22.1.1 `construct_with_default` – copy-constructing

```
construct_with_default(self, maindata: terminit)  
→ FormalSumContainerInterface
```

Create a new formal sum of the same class with `self`, with given only the `maindata` and use copy of `self`'s data if necessary.

4.22.1.2 `iterterms` – iterator of terms

```
iterterms(self) → iterator
```

Return an iterator of the terms.

Each term yielded from iterators is a `(base, coefficient)` pair.

4.22.1.3 `itercoefficients` – iterator of coefficients

```
itercoefficients(self) → iterator
```

Return an iterator of the coefficients.

4.22.1.4 `iterbases` – iterator of bases

```
iterbases(self) → iterator
```

Return an iterator of the bases.

4.22.1.5 `terms` – list of terms

```
terms(self) → list
```

Return a list of the terms.

Each term in returned lists is a `(base, coefficient)` pair.

4.22.1.6 `coefficients` – list of coefficients

```
coefficients(self) → list
```

Return a list of the coefficients.

4.22.1.7 bases – list of bases

bases(self) → list

Return a list of the bases.

4.22.1.8 terms_map – list of terms

terms_map(self, func: function) → FormalSumContainerInterface

Map on terms, i.e., create a new formal sum by applying **func** to each term.

func has to accept two parameters **base** and **coefficient**, then return a new term pair.

4.22.1.9 coefficients_map – list of coefficients

coefficients_map(self) → FormalSumContainerInterface

Map on coefficients, i.e., create a new formal sum by applying **func** to each coefficient.

func has to accept one parameters **coefficient**, then return a new coefficient.

4.22.1.10 bases_map – list of bases

bases_map(self) → FormalSumContainerInterface

Map on bases, i.e., create a new formal sum by applying **func** to each base.

func has to accept one parameters **base**, then return a new base.

4.22.2 DictFormalSum – formal sum implemented with dictionary

A formal sum implementation based on dict.

This class inherits **FormalSumContainerInterface**. All methods of the interface are implemented.

Initialize (Constructor)

```
DictFormalSum(args: terminit, defaultvalue: RingElement=None)  
→ DictFormalSum
```

See **terminit** for type of **args**. It makes a mapping from bases to coefficients.

The optional argument **defaultvalue** is the default value for **__getitem__**, i.e., if there is no term with the specified base, a look up attempt returns the **defaultvalue**. It is, thus, an element of the ring to which other coefficients belong.

4.22.3 ListFormalSum – formal sum implemented with list

A formal sum implementation based on list.

This class inherits **FormalSumContainerInterface**. All methods of the interface are implemented.

Initialize (Constructor)

```
ListFormalSum(args: terminit, defaultvalue: RingElement=None)  
→ ListFormalSum
```

See **terminit** for type of **args**. It makes a mapping from bases to coefficients.

The optional argument **defaultvalue** is the default value for **__getitem__**, i.e., if there is no term with the specified base, a look up attempt returns the **defaultvalue**. It is, thus, an element of the ring to which other coefficients belong.

4.23 poly.groebner – Gröbner Basis

The groebner module is for computing Gröbner bases for multivariate polynomial ideals.

This module uses the following types:

polynomial :

`polynomial` is the polynomial generated by function **polynomial**.

order :

`order` is the order on terms of polynomials.

4.23.1 buchberger – naïve algorithm for obtaining Gröbner basis

buchberger(generating: *list*, order: *order*) → [*polynomials*]

Return a Gröbner basis of the ideal generated by given generating set of polynomials with respect to the `order`.

Be careful, this implementation is very naive.

The argument `generating` is a list of **Polynomial**; the argument `order` is an order.

4.23.2 normal_strategy – normal algorithm for obtaining Gröbner basis

normal_strategy(generating: *list*, order: *order*) → [*polynomials*]

Return a Gröbner basis of the ideal generated by given generating set of polynomials with respect to the `order`.

This function uses the ‘normal strategy’.

The argument `generating` is a list of **Polynomial**; the argument `order` is an order.

4.23.3 reduce_groebner – reduce Gröbner basis

reduce_groebner(gbasis: *list*, order: *order*) → [*polynomials*]

Return the reduced Gröbner basis constructed from a Gröbner basis.

The output satisfies that:

- $\text{lb}(f)$ divides $\text{lb}(g) \Rightarrow g$ is not in reduced Gröbner basis, and
- monic.

The argument `gbasis` is a list of polynomials, a Gröbner basis (not merely a generating set).

4.23.4 `s_polynomial` – S-polynomial

`s_polynomial(f: polynomial, g: polynomial, order: order)`
 \rightarrow [*polynomials*]

Return S-polynomial of `f` and `g` with respect to the `order`.

$$S(f, g) = (\text{lc}(g) * T / \text{lb}(f)) * f - (\text{lc}(f) * T / \text{lb}(g)) * g,$$

where $T = \text{lcm}(\text{lb}(f), \text{lb}(g))$.

Examples

```
>>> f = multiutil.polynomial({(1,0):2, (1,1):1},rational.theRationalField, 2)
>>> g = multiutil.polynomial({(0,1):-2, (1,1):1},rational.theRationalField, 2)
>>> lex = termorder.lexicographic_order
>>> groebner.s_polynomial(f, g, lex)
UniqueFactorizationDomainPolynomial({(1, 0): 2, (0, 1): 2})
>>> gb = groebner.normal_strategy([f, g], lex)
>>> for gb_poly in gb:
...     print gb_poly
...
UniqueFactorizationDomainPolynomial({(1, 1): 1, (1, 0): 2})
UniqueFactorizationDomainPolynomial({(1, 1): 1, (0, 1): -2})
UniqueFactorizationDomainPolynomial({(1, 0): 2, (0, 1): 2})
UniqueFactorizationDomainPolynomial({(0, 2): -2, (0, 1): -4.0})
>>> gb_red = groebner.reduce_groebner(gb, lex)
>>> for gb_poly in gb_red:
...     print gb_poly
...
UniqueFactorizationDomainPolynomial({(1, 0): Rational(1, 1), (0, 1): Rational(1, 1)})
UniqueFactorizationDomainPolynomial({(0, 2): Rational(1, 1), (0, 1): 2.0})
```

4.24 `poly.hensel` – Hensel lift

- Classes
 - †**HenselLiftPair**

- †**HenselLiftMulti**
- †**HenselLiftSimultaneously**
- **Functions**
 - **lift_upto**

In this module document, *polynomial* means integer polynomial.

4.24.1 HenselLiftPair – Hensel lift for a pair

Initialize (Constructor)

```
HenselLiftPair(f: polynomial, a1: polynomial, a2: polynomial, u1: polynomial, u2: polynomial, p: integer, q: integer=p)  
→ HenselLiftPair
```

This object keeps integer polynomial pair which will be lifted by Hensel's lemma.

The argument should satisfy the following preconditions:

- f, a1 and a2 are monic
- $f == a1 * a2 \pmod{q}$
- $a1 * u1 + a2 * u2 == 1 \pmod{p}$
- p divides q and both are positive

```
from_factors(f: polynomial, a1: polynomial, a2: polynomial, p: integer)  
→ HenselLiftPair
```

This is a class method to create and return an instance of `HenselLiftPair`. You do not have to precompute u1 and u2 for the default constructor; they will be prepared for you from other arguments.

The argument should satisfy the following preconditions:

- f, a1 and a2 are monic
- $f == a1 * a2 \pmod{p}$
- p is prime

Attribute

point :
factors a1 and a2 as a list.

Methods

4.24.1.1 lift – lift one step

lift(self) →

Lift polynomials by so-called the quadratic method.

4.24.1.2 lift_factors – lift a_1 and a_2

lift_factors(self) →

Update factors by lifted integer coefficient polynomials A_i 's:

- $f == A_1 * A_2 \pmod{p * q}$
- $A_i == a_i \pmod{q} \ (i = 1, 2)$

Moreover, q is updated to $p * q$.

†The preconditions which should be automatically satisfied:

- $f == a_1 * a_2 \pmod{q}$
- $a_1 * u_1 + a_2 * u_2 == 1 \pmod{p}$
- p divides q

4.24.1.3 lift_ladder – lift u_1 and u_2

lift_ladder(self) →

Update u_1 and u_2 with U_1 and U_2 :

- $a_1 * U_1 + a_2 * U_2 == 1 \pmod{p**2}$
- $U_i == u_i \pmod{p} \ (i = 1, 2)$

Then, update p to $p**2$.

†The preconditions which should be automatically satisfied:

- $a_1 * u_1 + a_2 * u_2 == 1 \pmod{p}$

4.24.2 HenselLiftMulti – Hensel lift for multiple polynomials

Initialize (Constructor)

HenselLiftMulti(f: *polynomial*, factors: *list*, ladder: *tuple*, p: *integer*, q: *integer*=p) → *HenselLiftMulti*

This object keeps integer polynomial factors which will be lifted by Hensel's lemma. If the number of factors is just two, then you should use **HenselLiftPair**.

factors is a list of polynomials; we refer those polynomials as **a1**, **a2**, ... **ladder** is a tuple of two lists **sis** and **tis**, both lists consist polynomials. We refer polynomials in **sis** as **s1**, **s2**, ..., and those in **tis** as **t1**, **t2**, ... Moreover, we define **bi** as the product of **aj**'s for $i < j$. The argument should satisfy the following preconditions:

- **f** and all of **factors** are monic
- $f == a_1 * \dots * a_r \pmod{q}$
- $a_i * s_i + b_i * t_i == 1 \pmod{p} \ (i = 1, 2, \dots, r)$
- **p** divides **q** and both are positive

```
from _factors(f: polynomial, factors: list, p: integer)
    → HenselLiftMulti
```

This is a class method to create and return an instance of **HenselLiftMulti**. You do not have to precompute **ladder** for the default constructor; they will be prepared for you from other arguments.

The argument should satisfy the following preconditions:

- **f** and all of **factors** are monic
- $f == a_1 * \dots * a_r \pmod{q}$
- **p** is prime

Attribute

point :

factors **ais** as a list.

Methods

4.24.2.1 lift – lift one step

lift(self) →

Lift polynomials by so-called the quadratic method.

4.24.2.2 lift_factors – lift factors

lift_factors(self) →

Update factors by lifted integer coefficient polynomials A_i s:

- $f == A_1 \dots A_r \pmod{p * q}$
- $A_i == a_i \pmod{q} \ (i = 1, \dots, r)$

Moreover, q is updated to $p * q$.

†The preconditions which should be automatically satisfied:

- $f == a_1 \dots a_r \pmod{q}$
- $a_i * s_i + b_i * t_i == 1 \pmod{p} \ (i = 1, \dots, r)$
- p divides q

4.24.2.3 lift_ladder – lift u_1 and u_2

lift_ladder(self) →

Update s_i s and t_i s with S_i s and T_i s:

- $a_1 * S_i + b_i * T_i == 1 \pmod{p**2}$
- $S_i == s_i \pmod{p} \ (i = 1, \dots, r)$
- $T_i == t_i \pmod{p} \ (i = 1, \dots, r)$

Then, update p to $p**2$.

†The preconditions which should be automatically satisfied:

- $a_i * s_i + b_i * t_i == 1 \pmod{p} \ (i = 1, \dots, r)$

4.24.3 HenselLiftSimultaneously

The method explained in [?].

†Keep these invariants:

- a_i, p_i and g_i are monic
- $f == g_1 * \dots * g_r \pmod{p}$
- $f == d_0 + d_1 p + d_2 p^{**2} + \dots + d_k p^{**k}$
- $h_i == g_{(i+1)} * \dots * g_r$
- $1 == g_i * s_i + h_i * t_i \pmod{p} \ (i = 1, \dots, r)$
- $\deg(s_i) < \deg(h_i), \deg(t_i) < \deg(g_i) \ (i = 1, \dots, r)$
- p divides q
- $f == l_1 * \dots * l_r \pmod{q/p}$
- $f == a_1 * \dots * a_r \pmod{q}$
- $u_i == a_i * y_i + b_i * z_i \pmod{p} \ (i = 1, \dots, r)$

Initialize (Constructor)

```
HenselLiftSimultaneously(target: polynomial, factors: list, cofactors:
list, bases: list, p: integer)
    → HenselLiftSimultaneously
```

This object keeps integer polynomial factors which will be lifted by Hensel's lemma.

```
f = target, gi in factors, his in cofactors and sis and tis are in bases.
from _factors(target: polynomial, factors: list, p: integer, ubound: in-
teger=sys.maxint)
    → HenselLiftSimultaneously
```

This is a class method to create and return an instance of `HenselLiftSimultaneously`, whose factors are lifted by `HenselLiftMulti` upto `ubound` if it is smaller than `sys.maxint`, or upto `sys.maxint` otherwise. You do not have to precompute auxiliary polynomials for the default constructor; they will be prepared for you from other arguments.

```
f = target, gis in factors.
```

Methods

4.24.3.1 lift – lift one step

lift(self) →

The lift. You should call this method only.

4.24.3.2 first_lift – the first step

first_lift(self) →

Start lifting.

`f == l1*l2*...*lr (mod p**2)`

Initialize `dis`, `uis`, `vis` and `zis`. Update `ais`, `bis`. Then, update `q` with `p**2`.

4.24.3.3 general_lift – next step

general_lift(self) →

Continue lifting.

`f == a1*a2*...*ar (mod p*q)`

Initialize `ais`, `ubis`, `vis` and `zis`. Then, update `q` with `p*q`.

4.24.4 lift_upto – main function

lift_upto(self, target: *polynomial*, factors: *list*, p: *integer*, bound: *integer*)
→ *tuple*

Hensel lift `factors` mod `p` of `target` upto `bound` and return `factors` mod `q` and the `q` itself.

These preconditions should be satisfied:

- `target` is monic.
- `target == product(factors) mod p`

The result (`factors`, `q`) satisfies the following postconditions:

- there exist k s.t. `q == p**k >= bound` and
- `target == product(factors) mod q`

4.25 poly.multiutil – utilities for multivariate polynomials

- **Classes**
 - **RingPolynomial**
 - **DomainPolynomial**
 - **UniqueFactorizationDomainPolynomial**
 - OrderProvider
 - NestProvider
 - PseudoDivisionProvider
 - GcdProvider
 - RingElementProvider
- **Functions**
 - **polynomial**

4.25.1 RingPolynomial

General polynomial with commutative ring coefficients.

Initialize (Constructor)

```
RingPolynomial(coefficients: termint, **keywords: dict)  
    → RingPolynomial
```

The keywords must include:

coeffring a commutative ring (*CommutativeRing*)

number_of_variables the number of variables(*integer*)

order term order (*TermOrder*)

This class inherits **BasicPolynomial**, **OrderProvider**, **NestProvider** and **RingElementProvider**.

Attribute

order :
term order.

Methods

4.25.1.1 `getRing`

`getRing(self) → Ring`

Return an object of a subclass of `Ring`, to which the polynomial belongs.
(This method overrides the definition in `RingElementProvider`)

4.25.1.2 `getCoefficientRing`

`getCoefficientRing(self) → Ring`

Return an object of a subclass of `Ring`, to which the all coefficients belong.
(This method overrides the definition in `RingElementProvider`)

4.25.1.3 `leading_variable`

`leading_variable(self) → integer`

Return the position of the leading variable (the leading term among all total degree one terms).
The leading term varies with term orders, so does the result. The term order can be specified via the attribute `order`.
(This method is inherited from `NestProvider`)

4.25.1.4 `nest`

**`nest(self, outer: integer, coeffring: CommutativeRing)
→ polynomial`**

Nest the polynomial by extracting `outer` variable at the given position.
(This method is inherited from `NestProvider`)

4.25.1.5 `unnest`

**`nest(self, q: polynomial, outer: integer, coeffring: CommutativeRing)
→ polynomial`**

Unnest the nested polynomial `q` by inserting `outer` variable at the given position.
(This method is inherited from `NestProvider`)

4.25.2 `DomainPolynomial`

Polynomial with domain coefficients.

Initialize (Constructor)

```
DomainPolynomial(coefficients: terminit, **keywords: dict)  
→ DomainPolynomial
```

The **keywords** must include:

coeffring a commutative ring (*CommutativeRing*)

number_of_variables the number of variables(*integer*)

order term order (*TermOrder*)

This class inherits **RingPolynomial** and **PseudoDivisionProvider**.

Operations

operator	explanation
f / g	division (result is a rational function)

Methods

4.25.2.1 pseudo_divmod

pseudo_divmod(self, other: *polynomial*) → *polynomial*

Return Q, R polynomials such that:

$$d^{\deg(\text{self})-\deg(\text{other})+1}\text{self} = \text{other} \times Q + R$$

w.r.t. a fixed variable, where d is the leading coefficient of **other**.

The leading coefficient varies with term orders, so does the result. The term order can be specified via the attribute **order**.

(This method is inherited from PseudoDivisionProvider.)

4.25.2.2 pseudo_floordiv

pseudo_floordiv(self, other: *polynomial*) → *polynomial*

Return a polynomial Q such that

$$d^{\deg(\text{self})-\deg(\text{other})+1}\text{self} = \text{other} \times Q + R$$

w.r.t. a fixed variable, where d is the leading coefficient of **other** and R is a polynomial.

The leading coefficient varies with term orders, so does the result. The term order can be specified via the attribute **order**.

(This method is inherited from PseudoDivisionProvider.)

4.25.2.3 pseudo_mod

pseudo_mod(self, other: *polynomial*) → *polynomial*

Return a polynomial R such that

$$d^{\deg(\text{self})-\deg(\text{other})+1} \times \text{self} = \text{other} \times Q + R$$

where d is the leading coefficient of **other** and Q a polynomial.

The leading coefficient varies with term orders, so does the result. The term order can be specified via the attribute **order**.

(This method is inherited from PseudoDivisionProvider.)

4.25.2.4 exact_division

exact_division(self, other: *polynomial*) → *polynomial*

Return quotient of exact division.

(This method is inherited from PseudoDivisionProvider.)

4.25.3 UniqueFactorizationDomainPolynomial

Polynomial with unique factorization domain (UFD) coefficients.

Initialize (Constructor)

```
UniqueFactorizationDomainPolynomial(coefficients: terminit,  
**keywords: dict)  
    → UniqueFactorizationDomainPolynomial
```

The keywords must include:

coeffring a commutative ring (*CommutativeRing*)

number_of_variables the number of variables(*integer*)

order term order (*TermOrder*)

This class inherits **DomainPolynomial** and **GcdProvider**.

Methods

4.25.3.1 gcd

gcd(self, other: *polynomial*) → *polynomial*

Return gcd. The nested polynomials' gcd is used.
(This method is inherited from GcdProvider.)

4.25.3.2 resultant

resultant(self, other: *polynomial*, var: *integer*) → *polynomial*

Return resultant of two polynomials of the same ring, with respect to the variable specified by its position var.

4.25.4 polynomial – factory function for various polynomials

polynomial(coefficients: *terminit*, coeffring: *CommutativeRing*,
number_of_variables: *integer*=None)
→ *polynomial*

Return a polynomial.

†One can override the way to choose a polynomial type from a coefficient ring, by setting:
`special_ring_table[coeffring_type] = polynomial_type`
before the function call.

4.25.5 prepare_indeterminates – simultaneous declarations of indeterminates

prepare_indeterminates(names: *string*, ctx: *dict*, coeffring: *CoefficientRing*=None)
→ *None*

From space separated **names** of indeterminates, prepare variables representing the indeterminates. The result will be stored in **ctx** dictionary.

The variables should be prepared at once, otherwise wrong aliases of variables may confuse you in later calculation.

If an optional **coeffring** is not given, indeterminates will be initialized as integer coefficient polynomials.

Examples

```
>>> prepare_indeterminates("X Y Z", globals())
>>> Y
```

`UniqueFactorizationDomainPolynomial({(0, 1, 0): 1})`

4.26 poly.multivar – multivariate polynomial

- Classes
 - †**PolynomialInterface**
 - †**BasicPolynomial**
 - **TermIndices**

4.26.1 PolynomialInterface – base class for all multivariate polynomials

Since the interface is an abstract class, do not instantiate.

4.26.2 BasicPolynomial – basic implementation of polynomial

Basic polynomial data type.

4.26.3 TermIndices – Indices of terms of multivariate polynomials

It is a tuple-like object.

Initialize (Constructor)

TermIndices(indices: *tuple*) → *TermIndices*

The constructor does not check the validity of indices, such as integerness, nonnegativity, etc.

Operations

operator	explanation
<code>t == u</code>	equality
<code>t != u</code>	inequality
<code>t + u</code>	(componentwise) addition
<code>t - u</code>	(componentwise) subtraction
<code>t * a</code>	(componentwise) multiplication by scalar <code>a</code>
<code>t <= u, t < u, t >= u, t > u</code>	ordering
<code>t[k]</code>	k-th index
<code>len(t)</code>	length
<code>iter(t)</code>	iterator
<code>hash(t)</code>	hash

Methods

4.26.3.1 pop

pop(self, pos: *integer*) → (*integer*, *TermIndices*)

Return the index at pos and a new TermIndices object as the omitting-the-pos indices.

4.26.3.2 gcd

gcd(self, other: *TermIndices*) → *TermIndices*

Return the “gcd” of two indices.

4.26.3.3 lcm

lcm(self, other: *TermIndices*) → *TermIndices*

Return the “lcm” of two indices.

4.27 poly.ratfunc – rational function

- Classes

- **RationalFunction**

A rational function is a ratio of two polynomials.

Please don't expect this module is useful. It just provides an acceptable container for polynomial division.

4.27.1 RationalFunction – rational function class

Initialize (Constructor)

RationalFunction(numerator: *polynomial*, denominator: *polynomial*=1)
→ *RationalFunction*

Make a rational function with the given **numerator** and **denominator**. If the **numerator** is a **RationalFunction** instance and **denominator** is not given, then make a copy. If the **numerator** is a kind of polynomial, then make a rational function whose numerator is the given polynomial. Additionally, if **denominator** is also given, the denominator is set to its values, otherwise the denominator is 1.

Attribute

numerator :
polynomial.

denominator :
polynomial.

Operations

operator	explanation
A==B	Return whether A and B are equal or not.
str(A)	Return readable string.
repr(A)	Return string representing A's structure.

Methods

4.27.1.1 `getRing` – get rational function field

`getRing(self)` → **RationalFunctionField**

Return the rational function field to which the rational function belongs.

4.28 poly.ring – polynomial rings

- Classes
 - **PolynomialRing**
 - **RationalFunctionField**
 - **PolynomialIdeal**

4.28.1 PolynomialRing – ring of polynomials

A class for uni-/multivariate polynomial rings. A subclass of **CommutativeRing**.

Initialize (Constructor)

```
PolynomialRing(coeffring: CommutativeRing, number_of_variables:  
integer=1)  
→ PolynomialRing
```

`coeffring` is the ring of coefficients. `number_of_variables` is the number of variables. If its value is greater than 1, the ring is for multivariate polynomials.

Attribute

zero :
zero of the ring.

one :
one of the ring.

Methods

4.28.1.1 getInstance – classmethod

getInstance(coeffring: *CommutativeRing*, number_of_variables: *integer*)
→ *PolynomialRing*

return the instance of polynomial ring with coefficient ring `coeffring` and number of variables `number_of_variables`.

4.28.1.2 getCoefficientRing

getCoefficientRing() → *CommutativeRing*

4.28.1.3 getQuotientField

getQuotientField() → *Field*

4.28.1.4 issubring

issubring(other: *Ring*) → *bool*

4.28.1.5 issuperring

issuperring(other: *Ring*) → *bool*

4.28.1.6 getCharacteristic

getCharacteristic() → *integer*

4.28.1.7 createElement

createElement(seed) → *polynomial*

Return a polynomial. `seed` can be a polynomial, an element of coefficient ring, or any other data suited for the first argument of uni-/multi-variate polynomials.

4.28.1.8 gcd

gcd(a, b) → *polynomial*

Return the greatest common divisor of given polynomials (if possible). The polynomials must be in the polynomial ring. If the coefficient ring is a field, the result is monic.

4.28.1.9 isdomain

4.28.1.10 iseclidean

4.28.1.11 isnoetherian

4.28.1.12 ispid

4.28.1.13 isufd

Inherited from **CommutativeRing**.

4.28.2 RationalFunctionField – field of rational functions

Initialize (Constructor)

RationalFunctionField(field: *Field*, number_of_variables: *integer*)
→ *RationalFunctionField*

A class for fields of rational functions. It is a subclass of **QuotientField**.

field is the field of coefficients, which should be a **Field** object. number_of_variables is the number of variables.

Attribute

zero :
zero of the field.

one :
one of the field.

Methods

4.28.2.1 getInstance – classmethod

getInstance(coefffield: *Field*, number_of_variables: *integer*)
→ *RationalFunctionField*

return the instance of `RationalFunctionField` with coefficient field `coefffield` and number of variables `number_of_variables`.

4.28.2.2 createElement

createElement(*seedarg: *list*, **seedkwd: *dict*) → *RationalFunction*

4.28.2.3 getQuotientField

getQuotientField() → *Field*

4.28.2.4 issubring

issubring(other: *Ring*) → *bool*

4.28.2.5 issuperring

issuperring(other: *Ring*) → *bool*

4.28.2.6 unnest

unnest() → *RationalFunctionField*

If `self` is a nested `RationalFunctionField` i.e. its coefficient field is also a `RationalFunctionField`, then the method returns one level unnested `RationalFunctionField`.
For example:

Examples

```
>>> RationalFunctionField(RationalFunctionField(Q, 1), 1).unnest()  
RationalFunctionField(Q, 2)
```

4.28.2.7 gcd

gcd(a: *RationalFunction*, b: *RationalFunction*) → *RationalFunction*

Inherited from `Field`.

4.28.2.8 `isdomain`

4.28.2.9 `iseuclidean`

4.28.2.10 `isnoetherian`

4.28.2.11 `ispid`

4.28.2.12 `isufd`

Inherited from **CommutativeRing**.

4.28.3 PolynomialIdeal – ideal of polynomial ring

A subclass of **Ideal** represents ideals of polynomial rings.

Initialize (Constructor)

PolynomialIdeal(generators: *list*, polyring: *PolynomialRing*)
→ *PolynomialIdeal*

Create an object represents an ideal in a polynomial ring `polyring` generated by `generators`.

Operations

operator	explanation
<code>in</code>	membership test
<code>==</code>	same ideal?
<code>!=</code>	different ideal?
<code>+</code>	addition
<code>*</code>	multiplication

Methods

4.28.3.1 reduce

`reduce(element: polynomial) → polynomial`

Modulo `element` by the ideal.

4.28.3.2 issubset

`issubset(other: set) → bool`

4.28.3.3 issuperset

`issuperset(other: set) → bool`

4.29 poly.termorder – term orders

- **Classes**
 - †**TermOrderInterface**
 - †**UnivarTermOrder**
 - **MultivarTermOrder**
- **Functions**
 - **weight_order**

4.29.1 TermOrderInterface – interface of term order

Initialize (Constructor)

TermOrderInterface(comparator: *function*) → *TermOrderInterface*

A term order is primarily a function, which determines precedence between two terms (or monomials). By the precedence, all terms are ordered.

More precisely in terms of `Python`, a term order accepts two tuples of integers, each of which represents power indices of the term, and returns 0, 1 or -1 just like `cmp` built-in function.

A `TermOrder` object provides not only the precedence function, but also a method to format a string for a polynomial, to tell degree, leading coefficients, etc.

`comparator` accepts two tuple-like objects of integers, each of which represents power indices of the term, and returns 0, 1 or -1 just like `cmp` built-in function.

This class is abstract and should not be instantiated. The methods below have to be overridden.

Methods

4.29.1.1 `cmp`

`cmp(self, left: tuple, right: tuple) → integer`

Compare two index tuples `left` and `right` and determine precedence.

4.29.1.2 `format`

**`format(self, polynom: polynomial, **keywords: dict)`
→ *string***

Return the formatted string of the polynomial `polynom`.

4.29.1.3 `leading_coefficient`

`leading_coefficient(self, polynom: polynomial) → CommutativeRingElement`

Return the leading coefficient of polynomial `polynom` with respect to the term order.

4.29.1.4 `leading_term`

`leading_term(self, polynom: polynomial) → tuple`

Return the leading term of polynomial `polynom` as tuple of (degree index, coefficient) with respect to the term order.

4.29.2 `UnivarTermOrder` – term order for univariate polynomials

Initialize (Constructor)

`UnivarTermOrder(comparator: function) → UnivarTermOrder`

There is one unique term order for univariate polynomials. It's known as degree.

One thing special to univariate case is that powers are not tuples but bare integers. According to the fact, method signatures also need be translated from the definitions in `TermOrderInterface`, but its easy, and we omit some explanations.

`comparator` can be any callable that accepts two integers and returns 0, 1 or -1 just like `cmp`, i.e. if they are equal it returns 0, first one is greater 1, and otherwise -1. Theoretically acceptable comparator is only the `cmp` function.

This class inherits **TermOrderInterface**.

Methods

4.29.2.1 `format`

```
format(self, polynom: polynomial, varname: string='X', reverse:  
bool=False)  
    → string
```

Return the formatted string of the polynomial `polynom`.

- `polynom` must be a univariate polynomial.
- `varname` can be set to the name of the variable.
- `reverse` can be either `True` or `False`. If it's `True`, terms appear in reverse (descending) order.

4.29.2.2 `degree`

```
degree(self, polynom: polynomial) → integer
```

Return the degree of the polynomial `polynom`.

4.29.2.3 `tail_degree`

```
tail_degree(self, polynom: polynomial) → integer
```

Return the least degree among all terms of the `polynom`.

This method is *experimental*.

4.29.3 `MultivarTermOrder` – term order for multivariate polynomials

Initialize (Constructor)

```
MultivarTermOrder(comparator: function) → MultivarTermOrder
```

This class inherits **TermOrderInterface**.

Methods

4.29.3.1 `format`

```
format(self, polynom: polynomial, varname: tuple=None, reverse:  
bool=False, **kwds: dict)  
    → string
```

Return the formatted string of the polynomial `polynom`.

An additional argument `varnames` is required to name variables.

- `polynom` is a multivariate polynomial.
- `varnames` is the sequence of the variable names.
- `reverse` can be either `True` or `False`. If it's `True`, terms appear in reverse (descending) order.

4.29.4 `weight_order` – weight order

```
weight_order(weight: sequence, tie_breaker: function=None)  
    → function
```

Return a comparator of weight ordering by `weight`.

Let w denote the `weight`. The weight ordering is defined for arguments x and y that $x < y$ if $w \cdot x < w \cdot y$ or $w \cdot x == w \cdot y$ and tie breaker tells $x < y$.

The option `tie_breaker` is another comparator that will be used if dot products with the weight vector leaves arguments tie. If the option is `None` (default) and a tie breaker is indeed necessary to order given arguments, a `TypeError` is raised.

Examples

```
>>> w = termorder.MultivarTermOrder(  
...     termorder.weight_order((6, 3, 1), cmp))  
>>> w.cmp((1, 0, 0), (0, 1, 2))  
1
```

4.30 poly.uniutil – univariate utilities

- **Classes**
 - **RingPolynomial**
 - **DomainPolynomial**
 - **UniqueFactorizationDomainPolynomial**
 - **IntegerPolynomial**
 - **FieldPolynomial**
 - **FinitePrimeFieldPolynomial**
 - OrderProvider
 - DivisionProvider
 - PseudoDivisionProvider
 - ContentProvider
 - SubresultantGcdProvider
 - PrimeCharacteristicFunctionsProvider
 - VariableProvider
 - RingElementProvider
- **Functions**
 - **polynomial**

4.30.1 RingPolynomial – polynomial over commutative ring

Initialize (Constructor)

```
RingPolynomial(coefficients: terminit, coeffring: CommutativeRing,  
               **keywords: dict)  
    → RingPolynomial object
```

Initialize a polynomial over the given commutative ring `coeffring`.

This class inherits from **SortedPolynomial**, **OrderProvider** and **RingElementProvider**.

The type of the `coefficients` is **terminit**. `coeffring` is an instance of descendant of **CommutativeRing**.

Methods

4.30.1.1 `getRing`

`getRing(self) → Ring`

Return an object of a subclass of `Ring`, to which the polynomial belongs.
(This method overrides the definition in `RingElementProvider`)

4.30.1.2 `getCoefficientRing`

`getCoefficientRing(self) → Ring`

Return an object of a subclass of `Ring`, to which the all coefficients belong.
(This method overrides the definition in `RingElementProvider`)

4.30.1.3 `shift_degree_to`

`shift_degree_to(self, degree: integer) → polynomial`

Return polynomial whose degree is the given `degree`. More precisely, let $f(X) = a_0 + \dots + a_n X^n$, then `f.shift_degree_to(m)` returns:

- zero polynomial, if `f` is zero polynomial
- $a_{n-m} + \dots + a_n X^m$, if $0 \leq m < n$
- $a_0 X^{m-n} + \dots + a_n X^m$, if $m \geq n$

(This method is inherited from `OrderProvider`)

4.30.1.4 `split_at`

`split_at(self, degree: integer) → polynomial`

Return tuple of two polynomials, which are split at the given degree. The term of the given degree, if exists, belongs to the lower degree polynomial.
(This method is inherited from `OrderProvider`)

4.30.2 DomainPolynomial – polynomial over domain

Initialize (Constructor)

`DomainPolynomial(coefficients: terminit, coeffring: CommutativeRing, **keywords: dict)`
`→ DomainPolynomial object`

Initialize a polynomial over the given domain `coeffring`.

In addition to the basic polynomial operations, it has pseudo division methods.

This class inherits **RingPolynomial** and **PseudoDivisionProvider**.

The type of the `coefficients` is **termint**. `coeffring` is an instance of descendant of **CommutativeRing** which satisfies `coeffring.isdomain()`.

Methods

4.30.2.1 pseudo_divmod

pseudo_divmod(self, other: *polynomial*) → tuple

Return a tuple (Q, R), where Q, R are polynomials such that:

$$d^{\deg(f)-\deg(\text{other})+1}f = \text{other} \times Q + R,$$

where d is the leading coefficient of **other**.

(This method is inherited from PseudoDivisionProvider)

4.30.2.2 pseudo_floordiv

pseudo_floordiv(self, other: *polynomial*) → *polynomial*

Return a polynomial Q such that:

$$d^{\deg(f)-\deg(\text{other})+1}f = \text{other} \times Q + R,$$

where d is the leading coefficient of **other**.

(This method is inherited from PseudoDivisionProvider)

4.30.2.3 pseudo_mod

pseudo_mod(self, other: *polynomial*) → *polynomial*

Return a polynomial R such that:

$$d^{\deg(f)-\deg(\text{other})+1}f = \text{other} \times Q + R,$$

where d is the leading coefficient of **other**.

(This method is inherited from PseudoDivisionProvider)

4.30.2.4 exact_division

exact_division(self, other: *polynomial*) → *polynomial*

Return quotient of exact division.

(This method is inherited from PseudoDivisionProvider)

4.30.2.5 scalar_exact_division

**scalar_exact_division(self, scale: *CommutativeRingElement*)
→ *polynomial***

Return quotient by **scale** which can divide each coefficient exactly.

(This method is inherited from PseudoDivisionProvider)

4.30.2.6 discriminant

discriminant(self) → *CommutativeRingElement*

Return discriminant of the polynomial.

4.30.2.7 to_field_polynomial

to_field_polynomial(self) → *FieldPolynomial*

Return a *FieldPolynomial* object obtained by embedding the polynomial ring over the domain D to over the quotient field of D .

4.30.3 UniqueFactorizationDomainPolynomial – polynomial over UFD

Initialize (Constructor)

**UniqueFactorizationDomainPolynomial(coefficients: *terminit*,
coeffring: *CommutativeRing*, **keywords: *dict*)
→ *UniqueFactorizationDomainPolynomial* object**

Initialize a polynomial over the given UFD *coeffring*.

This class inherits from **DomainPolynomial**, **SubresultantGcdProvider** and **ContentProvider**.

The type of the *coefficients* is **terminit**. *coeffring* is an instance of descendant of **CommutativeRing** which satisfies *coeffring*.isufd().

4.30.3.1 content

content(self) → *CommutativeRingElement*

Return content of the polynomial.
(This method is inherited from *ContentProvider*)

4.30.3.2 primitive_part

primitive_part(self) → *UniqueFactorizationDomainPolynomial*

Return the primitive part of the polynomial.
(This method is inherited from *ContentProvider*)

4.30.3.3 subresultant_gcd

subresultant_gcd(self, other: *polynomial*) → *UniqueFactorizationDomainPolynomial*

Return the greatest common divisor of given polynomials. They must be in the polynomial ring and its coefficient ring must be a UFD.

(This method is inherited from SubresultantGcdProvider)

Reference: [?]Algorithm 3.3.1

4.30.3.4 subresultant_extgcd

subresultant_extgcd(self, other: *polynomial*) → *tuple*

Return (A, B, P) s.t. $A \times self + B \times other = P$, where P is the greatest common divisor of given polynomials. They must be in the polynomial ring and its coefficient ring must be a UFD.

Reference: [?]p.18

(This method is inherited from SubresultantGcdProvider)

4.30.3.5 resultant

resultant(self, other: *polynomial*) → *polynomial*

Return the resultant of *self* and *other*.

(This method is inherited from SubresultantGcdProvider)

4.30.4 IntegerPolynomial – polynomial over ring of rational integers

Initialize (Constructor)

IntegerPolynomial(coefficients: *terminit*, coeffring: *CommutativeRing*, **keywords: *dict*)
→ *IntegerPolynomial object*

Initialize a polynomial over the given commutative ring *coeffring*.

This class is required because special initialization must be done for built-in int/long.

This class inherits from **UniqueFactorizationDomainPolynomial**.

The type of the *coefficients* is **terminit**. *coeffring* is an instance of **IntegerRing**. You have to give the rational integer ring, though it seems redundant.

4.30.5 FieldPolynomial – polynomial over field

Initialize (Constructor)

```
FieldPolynomial(coefficients: terminit, coeffring: Field, **keywords:
dict)
    → FieldPolynomial object
```

Initialize a polynomial over the given field `coeffring`.

Since the polynomial ring over field is a Euclidean domain, it provides divisions.

This class inherits from **RingPolynomial**, **DivisionProvider** and **ContentProvider**.

The type of the `coefficients` is **terminit**. `coeffring` is an instance of descendant of **Field**.

Operations

operator	explanation
<code>f // g</code>	quotient of floor division
<code>f % g</code>	remainder
<code>divmod(f, g)</code>	quotient and remainder
<code>f / g</code>	division in rational function field

Methods

4.30.5.1 content

content(self) → *FieldElement*

Return content of the polynomial.
(This method is inherited from ContentProvider)

4.30.5.2 primitive_part

primitive_part(self) → *polynomial*

Return the primitive part of the polynomial.
(This method is inherited from ContentProvider)

4.30.5.3 mod

mod(self, dividend: *polynomial*) → *polynomial*

Return *dividend* mod *self*.
(This method is inherited from DivisionProvider)

4.30.5.4 scalar_exact_division

scalar_exact_division(self, scale: *FieldElement*)
→ *polynomial*

Return quotient by *scale* which can divide each coefficient exactly.
(This method is inherited from DivisionProvider)

4.30.5.5 gcd

gcd(self, other: *polynomial*) → *polynomial*

Return a greatest common divisor of self and other.

Returned polynomial is always monic.
(This method is inherited from DivisionProvider)

4.30.5.6 extgcd

extgcd(self, other: *polynomial*) → *tuple*

Return a tuple (`u`, `v`, `d`); they are the greatest common divisor d of two polynomials `self` and `other` and u, v such that

$$d = self \times u + other \times v$$

See **extgcd**.

(This method is inherited from `DivisionProvider`)

4.30.6 `FinitePrimeFieldPolynomial` – polynomial over finite prime field

Initialize (Constructor)

```
FinitePrimeFieldPolynomial(coefficients: terminit, coeffring:
FinitePrimeField, **keywords: dict)
    → FinitePrimeFieldPolynomial object
```

Initialize a polynomial over the given commutative ring `coeffring`.

This class inherits from **FieldPolynomial** and **PrimeCharacteristicFunctionsProvider**.

The type of the `coefficients` is **terminit**. `coeffring` is an instance of descendant of **FinitePrimeField**.

Methods

4.30.6.1 mod_pow – powering with modulus

```
mod_pow(self, polynom: polynomial, index: integer)  
→ polynomial
```

Return $\text{polynom}^{\text{index}} \bmod \text{self}$.

Note that **self** is the modulus.
(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.2 pthroot

```
pthroot(self) → polynomial
```

Return a polynomial obtained by sending X^p to X , where p is the characteristic. If the polynomial does not consist of p -th powered terms only, result is nonsense.

(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.3 squarefree_decomposition

```
squarefree_decomposition(self) → dict
```

Return the square free decomposition of the polynomial.

The return value is a dict whose keys are integers and values are corresponding powered factors. For example, If

Examples

```
>>> A = A1 * A2**2  
>>> A.squarefree_decomposition()  
{1: A1, 2: A2}.
```

(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.4 distinct_degree_decomposition

```
distinct_degree_decomposition(self) → dict
```

Return the distinct degree factorization of the polynomial.

The return value is a dict whose keys are integers and values are corresponding product of factors of the degree. For example, if $A = A1 \times A2$, and all irreducible

factors of $A1$ having degree 1 and all irreducible factors of $A2$ having degree 2, then the result is: {1: $A1$, 2: $A2$ }.

The given polynomial must be square free, and its coefficient ring must be a finite field.

(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.5 `split_same_degrees`

`split_same_degrees(self, degree:) → list`

Return the irreducible factors of the polynomial.

The polynomial must be a product of irreducible factors of the given degree. (This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.6 `factor`

`factor(self) → list`

Factor the polynomial.

The returned value is a list of tuples whose first component is a factor and second component is its multiplicity.

(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.6.7 `isirreducible`

`isirreducible(self) → bool`

If the polynomial is irreducible return `True`, otherwise `False`.

(This method is inherited from PrimeCharacteristicFunctionsProvider)

4.30.7 `polynomial` – factory function for various polynomials

`polynomial(coefficients: terminit, coeffring: CommutativeRing) → polynomial`

Return a polynomial.

†One can override the way to choose a polynomial type from a coefficient ring, by setting:

```
special_ring_table[coeffring_type] = polynomial_type
```

before the function call.

4.31 poly.univar – univariate polynomial

- Classes
 - †**PolynomialInterface**
 - †**BasicPolynomial**
 - **SortedPolynomial**

This `poly.univar` using following type:

polynomial :

`polynomial` is an instance of some descendant class of **PolynomialInterface** in this context.

4.31.1 PolynomialInterface – base class for all univariate polynomials

Initialize (Constructor)

Since the interface is an abstract class, do not instantiate.
The class is derived from **FormalSumContainerInterface**.

Operations

operator	explanation
$f * g$	multiplication ¹
$f ** i$	powering

Methods

4.31.1.1 differentiate – formal differentiation

differentiate(self) → *polynomial*

Return the formal differentiation of this polynomial.

4.31.1.2 downshift_degree – decreased degree polynomial

downshift_degree(self, slide: *integer*) → *polynomial*

Return the polynomial obtained by shifting downward all terms with degrees of `slide`.

Be careful that if the least degree term has the degree less than `slide` then the result is not mathematically a polynomial. Even in such a case, the method does not raise an exception.

†f.downshift_degree(slide) is equivalent to f.upsift_degree(-slide).

4.31.1.3 upshift_degree – increased degree polynomial

upshift_degree(self, slide: *integer*) → *polynomial*

Return the polynomial obtained by shifting upward all terms with degrees of `slide`.

†f.upshift_degree(slide) is equivalent to f.term_mul((slide, 1)).

4.31.1.4 ring_mul – multiplication in the ring

ring_mul(self, other: *polynomial*) → *polynomial*

Return the result of multiplication with the `other` polynomial.

4.31.1.5 scalar_mul – multiplication with a scalar

scalar_mul(self, scale: *scalar*) → *polynomial*

Return the result of multiplication by scalar `scale`.

4.31.1.6 term_mul – multiplication with a term

term_mul(self, term: *term*) → *polynomial*

Return the result of multiplication with the given `term`. The `term` can be given as a tuple (degree, coeff) or as a polynomial.

4.31.1.7 square – multiplication with itself

```
square(self) → polynomial
```

Return the square of this polynomial.

4.31.2 BasicPolynomial – basic implementation of polynomial

Basic polynomial data type. There are no concept such as variable name and ring.

Initialize (Constructor)

```
BasicPolynomial(coefficients: terminit, **keywords: dict)  
→ BasicPolynomial
```

This class inherits and implements **PolynomialInterface**.

The type of the coefficients is **terminit**.

4.31.3 SortedPolynomial – polynomial keeping terms sorted

Initialize (Constructor)

```
SortedPolynomial(coefficients: terminit, _sorted: bool=False,  
**keywords: dict)  
→ SortedPolynomial
```

The class is derived from **PolynomialInterface**.

The type of the coefficients is **terminit**. Optionally `_sorted` can be True if the coefficients is an already sorted list of terms.

Methods

4.31.3.1 degree – degree

degree(self) → *integer*

Return the degree of this polynomial. If the polynomial is the zero polynomial, the degree is -1 .

4.31.3.2 leading_coefficient – the leading coefficient

leading_coefficient(self) → *object*

Return the coefficient of highest degree term.

4.31.3.3 leading_term – the leading term

leading_term(self) → *tuple*

Return the leading term as a tuple (degree, coefficient).

4.31.3.4 †ring_mul_karatsuba – the leading term

ring_mul_karatsuba(self, other: *polynomial*) → *polynomial*

Multiplication of two polynomials in the same ring. Computation is carried out by Karatsuba method.

This may run faster when degree is higher than 100 or so. It is off by default, if you need to use this, do by yourself.