HElib (Detailed Description)

by

1. Basheir khan(FALL-16/RIS-007)
2. Sahibzada Shuja(FALL-16/RIS-015)

HElib

Implementing Homomorphic Encryption

AltCRT Class Reference

Alternative implementation of integer polynomials.

`#include <AltCRT.h>`

Public Member Functions

	AltCRT (const AltCRT &other)
	AltCRT (const ZZX &poly, const FHEcontext &_context, const IndexSet &indexSet)
	AltCRT (const ZZX &poly, const FHEcontext &_context)
	AltCRT (const ZZX &poly)
	AltCRT (const FHEcontext &_context, const IndexSet &indexSet)
	AltCRT (const FHEcontext &_context)
AltCRT &	**operator=** (const AltCRT &other)
AltCRT &	**operator=** (const ZZX &poly)
AltCRT &	**operator=** (const ZZ &num)
AltCRT &	**operator=** (const long num)

long	**getOneRow** (Vec< long > &row, long idx, bool positive=false) const
	Get one row of a polynomial.
long	**getOneRow** (zz_pX &row, long idx) const
void	**toPoly** (ZZX &p, const **IndexSet** &s, bool positive=false) const
void	**toPoly** (ZZX &p, bool positive=false) const
bool	**operator==** (const **AltCRT** &other) const
bool	**operator!=** (const **AltCRT** &other) const
AltCRT &	**SetZero** ()
AltCRT &	**SetOne** ()
void	**breakIntoDigits** (vector< **AltCRT** > &dgts, long n) const
void	**addPrimes** (const **IndexSet** &s1)
double	**addPrimesAndScale** (const **IndexSet** &s1)
void	**removePrimes** (const **IndexSet** &s1)
AltCRT &	**Negate** (const **AltCRT** &other)
AltCRT &	**Negate** ()
AltCRT &	**operator+=** (const **AltCRT** &other)

AltCRT &	**operator+=** (const ZZX &poly)
AltCRT &	**operator+=** (const ZZ &num)
AltCRT &	**operator+=** (long num)
AltCRT &	**operator-=** (const AltCRT &other)
AltCRT &	**operator-=** (const ZZX &poly)
AltCRT &	**operator-=** (const ZZ &num)
AltCRT &	**operator-=** (long num)
AltCRT &	**operator++** ()
AltCRT &	**operator--** ()
void	**operator++** (int)
void	**operator--** (int)
AltCRT &	**operator*=** (const AltCRT &other)
AltCRT &	**operator*=** (const ZZX &poly)
AltCRT &	**operator*=** (const ZZ &num)
AltCRT &	**operator*=** (long num)
void	**Add** (const AltCRT &other, bool matchIndexSets=true)

void	**Sub** (const AltCRT &other, bool matchIndexSets=true)
void	**Mul** (const AltCRT &other, bool matchIndexSets=true)
AltCRT &	**operator/=** (const ZZ &num)
AltCRT &	**operator/=** (long num)
void	**Exp** (long k)
void	**automorph** (long k)
AltCRT &	**operator>>=** (long k)
const FHEcontext &	**getContext** () const
const IndexMap< zz_pX > &	**getMap** () const
const IndexSet &	**getIndexSet** () const
void	**randomize** (const ZZ *seed=NULL)
void	**sampleSmall** ()
void	**sampleHWt** (long Hwt)
void	**sampleGaussian** (double stdev=0.0)
void	**sampleUniform** (const ZZ &B)
void	**scaleDownToSet** (const IndexSet &s, long ptxtSpace)

void	**reduce** () const

Friends

ostream &	**operator<<** (ostream &s, const **AltCRT** &d)
istream &	**operator>>** (istream &s, **AltCRT** &d)

Detailed Description

Alternative implementation of integer polynomials.

The documentation for this class was generated from the following files:

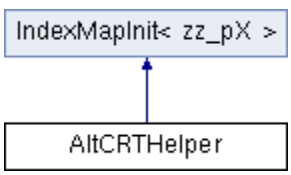
- src/**AltCRT.h**
- src/AltCRT.cpp

AltCRTHelper Class Reference

A helper class to enforce consistency within an **AltCRTHelper** object

`#include <AltCRT.h>`

Inheritance diagram for AltCRTHelper:



Public Member Functions

	AltCRTHelper (const **FHEcontext** &context)
virtual void	**init** (zz_pX &v)
	Initialization function, override with initialization code.
virtual **IndexMapInit**< zz_pX > *	**clone** () const
	Cloning a pointer, override with code to create a fresh copy.

Detailed Description

A helper class to enforce consistency within an **AltCRTHelper** object.

See Section 2.6.2 of the design document (**IndexMap**)

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

- src/**AltCRT.h**

BipartitleGraph Class Reference

A bipartite flow graph.

`#include <matching.h>`

Public Member Functions

void	**addEdge** (long from, long to, long label, long color=0)
void	**partitionToMatchings** ()
void	**printout** ()

Public Attributes

vector< LabeledVertex >	**left**

Detailed Description

A bipartite flow graph.

The documentation for this class was generated from the following files:

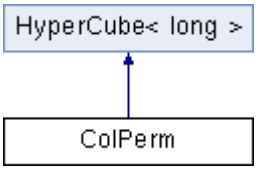
- src/matching.h
- src/matching.cpp

ColPerm Class Reference

Permuting a single dimension (column) of a hypercube. More...

`#include <permutations.h>`

Inheritance diagram for ColPerm:



Public Member Functions

	ColPerm (const **CubeSignature** &_sig)
long	**getPermDim** () const
void	**setPermDim** (long _dim)
void	**makeExplicit** (**Permut** &out) const
long	**getShiftAmounts** (**Permut** &out) const
void	**getBenesShiftAmounts** (Vec< **Permut** > &out, Vec< bool > &idID, const Vec< long > &benesLvls) const
void	**printout** (ostream &s)
	A test/debugging method.

Public Member Functions inherited from HyperCube< long >

Detailed Description

Permuting a single dimension (column) of a hypercube.

ColPerm is derived from a **HyperCube<long>**, and it uses the cube object to store the actual permutation data. The interpretation of this data, however, depends on the data member dim.

The cube is partitioned into columns of size n = getDim(dim): a single column consists of the n entries whose indices i have the same coordinates in all dimensions other than dim. The entries in any such column foem a permutation on [0..n).

For a given ColPerm perm, one way to access each column is as follows: for slice_index = [0..perm.getProd(0, dim)) CubeSlice slice(perm, slice_index, dim) for col_index = [0..perm.getProd(dim+1)) getHyperColumn(column, slice, col_index)

Another way is to use the getCoord and addCoord methods.

For example, permuting a 2x3x2 cube along dim=1 (the 2nd dimention), we could have the data vector as [1 1 0 2 2 0 2 0 1 1 0 2]. This means the four columns are permuted by the permutations [1 0 2] [1 2 0] [2 1 0] [0 1 2]. Written explicitly, we get: [2 3 0 5 4 1 10 7 8 9 6 11].

Another representation that we provide is by "shift amount": how many slots each element needs to move inside its small permutation. For the example above, this will be: [1 -1 0] [2 -1 -1] [2 0 -2] [0 0 0] so we write the permutatation as [1 1 -1 1 0 -2 2 0 0 0 -2 0].

Member Function Documentation

void ColPerm::getBenesShiftAmounts (Vec< Permut > & out,
Vec< bool > & idID,
const Vec< long > & benesLvls
) const

Get multiple layers of a Benes permutation network. Returns in out[i][j] the shift amount to move item j in the i'th layer. Also isID[i]=true if the i'th layer is the identity (i.e., contains only 0 shift amounts).

long ColPerm::getShiftAmounts (Permut & out) const

For each position in the data vector, compute how many slots it should be shifted inside its small permutation. Returns zero if all the shift amount are zero, nonzero values otherwise.

The documentation for this class was generated from the following files:

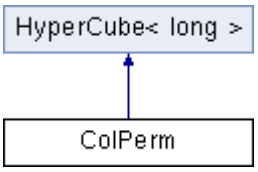
- src/permutations.h
- src/permutations.cpp

ColPerm Class Reference

Permuting a single dimension (column) of a hypercube.

`#include <permutations.h>`

Inheritance diagram for ColPerm:



Public Member Functions

	ColPerm (const CubeSignature &_sig)
long	**getPermDim** () const
void	**setPermDim** (long _dim)
void	**makeExplicit** (Permut &out) const
long	getShiftAmounts (Permut &out) const
void	getBenesShiftAmounts (Vec< Permut > &out, Vec< bool > &idID, const Vec< long > &benesLvls) const
void	printout (ostream &s)
	A test/debugging method.

Public Member Functions inherited from HyperCube< long >

Detailed Description

Permuting a single dimension (column) of a hypercube.

ColPerm is derived from a HyperCube<long>, and it uses the cube object to store the actual permutation data. The interpretation of this data, however, depends on the data member dim.

The cube is partitioned into columns of size n = getDim(dim): a single column consists of the n entries whose indices i have the same coordinates in all dimensions other than dim. The entries in any such column foem a permutation on [0..n).

For a given **ColPerm** perm, one way to access each column is as follows: for slice_index = [0..perm.getProd(0, dim)) **CubeSlice** slice(perm, slice_index, dim) for col_index = [0..perm.getProd(dim+1)) getHyperColumn(column, slice, col_index)

Another way is to use the getCoord and addCoord methods.

For example, permuting a 2x3x2 cube along dim=1 (the 2nd dimention), we could have the data vector as [1 1 0 2 2 0 2 0 1 1 0 2]. This means the four columns are permuted by the permutations [1 0 2] [1 2 0] [2 1 0] [0 1 2]. Written explicitly, we get: [2 3 0 5 4 1 10 7 8 9 6 11].

Another representation that we provide is by "shift amount": how many slots each element needs to move inside its small permutation. For the example above, this will be: [1 -1 0] [2 -1 -1] [2 0 -2] [0 0 0] so we write the permutatation as [1 1 -1 1 0 -2 2 0 0 0 -2 0].

Member Function Documentation

void ColPerm::getBenesShiftAmounts (Vec< Permut > & out,
Vec< bool > & idID,
const Vec< long > & benesLvls
) const

Get multiple layers of a Benes permutation network. Returns in out[i][j] the shift amount to move item j in the i'th layer. Also isID[i]=true if the i'th layer is the identity (i.e., contains only 0 shift amounts).

long ColPerm::getShiftAmounts (Permut & out) const

For each position in the data vector, compute how many slots it should be shifted inside its small permutation. Returns zero if all the shift amount are zero, nonzero values otherwise.

The documentation for this class was generated from the following files:

- src/**permutations.h**
- src/permutations.cpp

Ctxt Class Reference

A **Ctxt** object holds a single cipehrtext. More...

`#include <Ctxt.h>`

Public Member Functions

	Ctxt (const **FHEPubKey** &newPubKey, long newPtxtSpace=0)
	Ctxt (ZeroCtxtLike_type, const **Ctxt** &ctxt)
void	**DummyEncrypt** (const ZZX &ptxt, double size=-1.0)
	Dummy encryption, just encodes the plaintext in a **Ctxt** object.
Ctxt &	**operator=** (const **Ctxt** &other)
bool	**operator==** (const **Ctxt** &other) const
bool	**operator!=** (const **Ctxt** &other) const
bool	**equalsTo** (const **Ctxt** &other, bool comparePkeys=true) const

Ciphertext arithmetic

void	**negate** ()
Ctxt &	**operator+=** (const **Ctxt** &other)
Ctxt &	**operator-=** (const **Ctxt** &other)
void	**addCtxt** (const **Ctxt** &other, bool negative=false)
Ctxt &	**operator*=** (const **Ctxt** &other)

void	**automorph** (long k)
Ctxt &	**operator>>=** (long k)
void	**smartAutomorph** (long k) automorphism with re-lienarization
void	**frobeniusAutomorph** (long j) applies the automorphsim p^j using smartAutomorphism
void	**addConstant** (const DoubleCRT &dcrt, double size=-1.0)
void	**addConstant** (const ZZX &poly, double size=-1.0)
void	**addConstant** (const ZZ &c)
void	**multByConstant** (const DoubleCRT &dcrt, double size=-1.0)
void	**multByConstant** (const ZZX &poly, double size=-1.0)
void	**multByConstant** (const ZZ &c)
void	**divideByP** ()
void	**multByP** (long e=1)
void	**divideBy2** ()
void	**extractBits** (vector< Ctxt > &bits, long nBits2extract=0)
void	**multiplyBy** (const Ctxt &other)

void	**multiplyBy2** (const **Ctxt** &other1, const **Ctxt** &other2)
void	**square** ()
void	**cube** ()

Ciphertext maintenance

Reduce plaintext space to a divisor of the original plaintext space

void	**reducePtxtSpace** (long newPtxtSpace)
void	**reLinearize** (long keyIdx=0)
void	**cleanUp** ()
void	**reduce** () const
void	**blindCtxt** (const ZZX &poly)
	Add a high-noise encryption of the given constant.
xdouble	**modSwitchAddedNoiseVar** () const
	Estimate the added noise variance.
long	**findBaseLevel** () const
	Find the "natural level" of a cipehrtext. More...
void	**modUpToSet** (const **IndexSet** &s)
	Modulus-switching up (to a larger modulus). Must have primeSet <= s, and s must contain either all the special primes or none of them.
void	**modDownToSet** (const **IndexSet** &s)

	Modulus-switching down (to a smaller modulus). mod-switch down to primeSet s, after this call we have primeSet<=s. s must contain either all special primes or none of them.
void	**modDownToLevel** (long lvl)
	Modulus-switching down.
double	**rawModSwitch** (vector< ZZX > &zzParts, long toModulus) const
	Special-purpose modulus-switching for bootstrapping.
void	**findBaseSet** (**IndexSet** &s) const
	Find the "natural prime-set" of a cipehrtext. Find the highest **IndexSet** so that mod-switching down to that set results in the dominant noise term being the additive term due to rounding.
void	**evalPoly** (const ZZX &poly)
	compute the power X,X^2,...,X^n

Utility methods

void	**clear** ()
bool	**isEmpty** () const
	Is this an empty cipehrtext without any parts.
bool	**inCanonicalForm** (long keyID=0) const
	A canonical ciphertext has (at most) handles pointing to (1,s)
bool	**isCorrect** () const
	Would this ciphertext be decrypted without errors?

const FHEcontext &	**getContext** () const
const FHEPubKey &	**getPubKey** () const
const IndexSet &	**getPrimeSet** () const
const xdouble &	**getNoiseVar** () const
const long	**getPtxtSpace** () const
const long	**getKeyID** () const
const long	**effectiveR** () const
double	log_of_ratio () const Returns log(noise-variance)/2 - log(q)

Friends

class	**FHEPubKey**
class	**FHESecKey**
istream &	**operator>>** (istream &str, Ctxt &ctxt)
ostream &	**operator<<** (ostream &str, const Ctxt &ctxt)

Detailed Description

A Ctxt object holds a single cipehrtext.

The class **Ctxt** includes a vector<CtxtPart>: For a **Ctxt** c, c[i] is the i'th ciphertext part, which can be used also as a **DoubleCRT** object (since **CtxtPart** is derived from **DoubleCRT**). By convention, c[0], the first **CtxtPart** object in the vector, has skHndl that points to 1 (i.e., it is just added in upon decryption, without being multiplied by anything). We maintain the invariance that all the parts of a ciphertext are defined relative to the same set of primes.

A ciphertext contains also pointers to the general parameters of this FHE instance and the public key, and an estimate of the noise variance. The noise variance is determined by the norm of the canonical embedding of the noise polynomials, namely their evaluations in roots of the ring polynomial (which are the complex primitive roots of unity). We consider each such evaluation point as a random variable, and estimate the variances of these variables. This estimate is heuristic, assuming that various quantities "behave like independent random variables". The variance is added on addition, multiplied on multiplications, remains unchanged for automorphism, and is roughly scaled down by mod-switching with some added factor, and similarly scaled up by key-switching with some added factor. The noiseVar data member of the class keeps the esitmated variance.

Member Function Documentation

void Ctxt::divideByP ( )

Divide a cipehrtext by p, for plaintext space p^r, r>1. It is assumed that the ciphertext encrypts a polynomial which is zero mod p. If this is not the case then the result will not be a valid ciphertext anymore. As a side-effect, the plaintext space is reduced from p^r to p^{r-1}.

void Ctxt::evalPoly (const ZZX & poly)

compute the power X,X^2,...,X^n

Evaluate the cleartext poly on the encrypted ciphertext

long Ctxt::findBaseLevel ( ) const

Find the "natural level" of a cipehrtext.

How many levels in the "base-set" for that ciphertext.

void Ctxt::multByP (long e = 1) inline

Mulitply ciphretext by p^e, for plaintext space p^r. This also has the side-effect of increasing the plaintext space to p^{r+e}.

double Ctxt::rawModSwitch (vector< ZZX > & zzParts,
long toModulus
) Const

Special-purpose modulus-switching for bootstrapping.

Mod-switch to an externally-supplied modulus. The modulus need not be in the moduli-chain in the context, and does not even need to be a prime. The ciphertext *this is not affected, instead the result is returned in the zzParts vector, as a vector of ZZX'es. Returns an extimate for the noise variance after mod-switching.

The documentation for this class was generated from the following files:

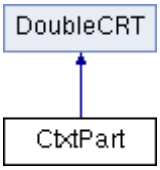
- src/Ctxt.h
- src/Ctxt.bak.cpp
- src/Ctxt.cpp

CtxtPart Class Reference

One entry in a ciphertext vector.

`#include <Ctxt.h>`

Inheritance diagram for CtxtPart:



Public Member Functions

bool	**operator==** (const CtxtPart &other) const
bool	**operator!=** (const CtxtPart &other) const

	CtxtPart (const FHEcontext &_context)
	CtxtPart (const FHEcontext &_context, const IndexSet &s)
	CtxtPart (const FHEcontext &_context, const IndexSet &s, const SKHandle &otherHandle)
	CtxtPart (const DoubleCRT &other)
	CtxtPart (const DoubleCRT &other, const SKHandle &otherHandle)

Public Member Functions inherited from DoubleCRT

Public Attributes

SKHandle	skHandle
	The handle is a public data member.

Detailed Description

One entry in a ciphertext vector.

A cipehrtext part consists of a polynomial (element of the ring R_Q) and a handle to the corresponding secret-key polynomial.

The documentation for this class was generated from the following files:

- src/Ctxt.h
- src/Ctxt.bak.cpp
- src/Ctxt.cpp

CubeSignature Class Reference

Holds a vector of dimensions for a hypercube and some additional data.

`#include <hypercube.h>`

Public Member Functions

	CubeSignature (const Vec< long > &_dims)
	CubeSignature (const **PAlgebra** &alg)
	Build a **CubeSignature** to reflect the hypercube structure of Zm* /(p)
void	**initSignature** (const Vec< long > &_dims)
long	**getSize** () const
	total size of cube
long	**getNumDims** () const
	number of dimensions
long	**getDim** (long d) const
	size of dimension d
long	**getProd** (long d) const
	product of sizes of dimensions d, d+1, ...
long	**getProd** (long from, long to) const
	product of sizes of dimensions from, from+1, ..., to-1
long	**getCoord** (long i, long d) const
	get coordinate in dimension d of index i

long	**addCoord** (long i, long d, long offset) const
	add offset to coordinate in dimension d of index i

long	**numSlices** (long d=1) const
	number of slices

long	**sliceSize** (long d=1) const
	size of one slice

long	**numCols** () const
	number of columns

Friends

ostream &	**operator<<** (ostream &s, const **CubeSignature** &sig)

Detailed Description

Holds a vector of dimensions for a hypercube and some additional data.

The documentation for this class was generated from the following files:

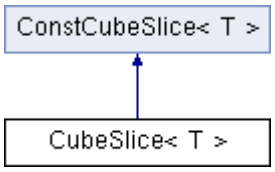
- src/**hypercube.h**
- src/PAlgebra.cpp

CubeSlice< T > Class Template Reference

A lower-dimension slice of a hypercube.

`#include <hypercube.h>`

Inheritance diagram for CubeSlice< T >:



Public Member Functions

	CubeSlice (HyperCube< T > &_cube)
	CubeSlice (Vec< T > &_data, const CubeSignature &_sig)
	CubeSlice (const CubeSlice< T > &bigger, long i, long _dimOffset=1)
	CubeSlice (HyperCube< T > &_cube, long i, long _dimOffset=1)
void	**copy** (const ConstCubeSlice< T > &other) const
T &	**at** (long i) const
T &	**operator[]** (long i) const

Public Member Functions inherited from ConstCubeSlice< T >

Detailed Description

template<class T>
class CubeSlice< T >

A lower-dimension slice of a hypercube.

The documentation for this class was generated from the following files:

- src/**hypercube.h**
- src/hypercube.cpp

deep_clone< X > Class Template Reference

Deep copy: initialize with clone.

`#include <cloned_ptr.h>`

Static Public Member Functions

static X *	**apply** (const X *x)

Detailed Description

template<class X>
class deep_clone< X >

Deep copy: initialize with clone.

Template Parameters

X The class to which this points

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

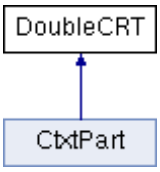
- src/**cloned_ptr.h**

DoubleCRT Class Reference

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

`#include <DoubleCRT.h>`

Inheritance diagram for DoubleCRT:



Public Member Functions

	DoubleCRT (const ZZX &poly, const **FHEcontext** &_context, const **IndexSet** &indexSet) Initializing **DoubleCRT** from a ZZX polynomial.
	DoubleCRT (const ZZX &poly, const **FHEcontext** &_context)
	DoubleCRT (const ZZX &poly) Context is not specified, use the "active context".
	DoubleCRT (const **FHEcontext** &_context)
	DoubleCRT (const **FHEcontext** &_context, const **IndexSet** &indexSet) Also specify the **IndexSet** explicitly.
DoubleCRT &	**operator=** (const **DoubleCRT** &other)
DoubleCRT &	**operator=** (const ZZX &poly)
DoubleCRT &	**operator=** (const ZZ &num)
DoubleCRT &	**operator=** (const long num)
long	**getOneRow** (Vec< long > &row, long idx, bool positive=false)

	const
	Get one row of a polynomial.
long	**getOneRow** (zz_pX &row, long idx) const
void	**toPoly** (ZZX &p, const **IndexSet** &s, bool positive=false) const
	Recovering the polynomial in coefficient representation. This yields an integer polynomial with coefficients in [-P/2,P/2], unless the positive flag is set to true, in which case we get coefficients in [0,P-1] (P is the product of all moduli used). Using the optional **IndexSet** param we compute the polynomial reduced modulo the product of only the ptimes in that set.
void	**toPoly** (ZZX &p, bool positive=false) const
bool	**operator==** (const **DoubleCRT** &other) const
bool	**operator!=** (const **DoubleCRT** &other) const
DoubleCRT &	**SetZero** ()
DoubleCRT &	**SetOne** ()
void	**breakIntoDigits** (vector< **DoubleCRT** > &dgts, long n) const
	Break into n digits,according to the primeSets in context.digits. See Section 3.1.6 of the design document (re-linearization)
void	**addPrimes** (const **IndexSet** &s1)
	Expand the index set by s1. It is assumed that s1 is disjoint from the current index set.

double	**addPrimesAndScale** (const **IndexSet** &s1)
	Expand index set by s1, and multiply by Prod_{q in s1}. s1 is disjoint from the current index set, returns log(product).
void	**removePrimes** (const **IndexSet** &s1)
	Remove s1 from the index set.
const **FHEcontext** &	**getContext** () const
const **IndexMap**< vec_long > &	**getMap** () const
const **IndexSet** &	**getIndexSet** () const
void	**randomize** (const ZZ *seed=NULL)
	Fills each row i with random ints mod pi, uses NTL's PRG.
void	**sampleSmall** ()
	Coefficients are -1/0/1, Prob[0]=1/2.
void	**sampleHWt** (long Hwt)
	Coefficients are -1/0/1 with pre-specified number of nonzeros.
void	**sampleGaussian** (double stdev=0.0)
	Coefficients are Gaussians.
void	**sampleUniform** (const ZZ &B)
	Coefficients are uniform in [-B..B].
void	**scaleDownToSet** (const **IndexSet** &s, long ptxtSpace)

void	**FFT** (const ZZX &poly, const **IndexSet** &s)
void	**reduce** () const

Arithmetic operation

Only the "destructive" versions are used, i.e., a += b is implemented but not a + b.

DoubleCRT &	**Negate** (const **DoubleCRT** &other)
DoubleCRT &	**Negate** ()
DoubleCRT &	**operator+=** (const **DoubleCRT** &other)
DoubleCRT &	**operator+=** (const ZZX &poly)
DoubleCRT &	**operator+=** (const ZZ &num)
DoubleCRT &	**operator+=** (long num)
DoubleCRT &	**operator-=** (const **DoubleCRT** &other)
DoubleCRT &	**operator-=** (const ZZX &poly)
DoubleCRT &	**operator-=** (const ZZ &num)
DoubleCRT &	**operator-=** (long num)
DoubleCRT &	**operator++** ()
DoubleCRT &	**operator--** ()
void	**operator++** (int)

void	**operator--** (int)
DoubleCRT &	**operator*=** (const **DoubleCRT** &other)
DoubleCRT &	**operator*=** (const ZZX &poly)
DoubleCRT &	**operator*=** (const ZZ &num)
DoubleCRT &	**operator*=** (long num)
void	**Add** (const **DoubleCRT** &other, bool matchIndexSets=true)
void	**Sub** (const **DoubleCRT** &other, bool matchIndexSets=true)
void	**Mul** (const **DoubleCRT** &other, bool matchIndexSets=true)
DoubleCRT &	**operator/=** (const ZZ &num)
DoubleCRT &	**operator/=** (long num)
void	**Exp** (long k) Small-exponent polynomial exponentiation.
void	**automorph** (long k)
DoubleCRT &	**operator>>=** (long k)

Friends

ostream &	**operator<<** (ostream &s, const **DoubleCRT** &d)
istream &	**operator>>** (istream &s, **DoubleCRT** &d)

Detailed Description

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

Double-CRT form is a matrix of L rows and phi(m) columns. The i'th row contains the FFT of the element wrt the ith prime, i.e. the evaluations of the polynomial at the primitive mth roots of unity mod the ith prime. The polynomial thus represented is defined modulo the product of all the primes in use.

The list of primes is defined by the data member indexMap. indexMap.getIndexSet() defines the set of indices of primes associated with this **DoubleCRT** object: they index the primes stored in the associated FHEContext.

Arithmetic operations are computed modulo the product of the primes in use and also modulo Phi_m(X). Arithmetic operations can only be applied to **DoubleCRT** objects relative to the same context, trying to add/multiply objects that have different FHEContext objects will raise an error.

Constructor & Destructor Documentation

```
DoubleCRT::DoubleCRT ( const ZZX &          poly,
                       const FHEcontext &   _context,
                       const IndexSet &     indexSet
                     )
```

Initializing **DoubleCRT** from a ZZX polynomial.

Parameters

poly	The ring element itself, zero if not specified
_context	The context for this **DoubleCRT** object, use "current active context" if not specified
indexSet	Which primes to use for this object, if not specified then use all of them

---

The documentation for this class was generated from the following files:

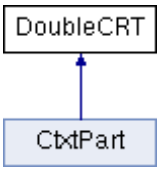
- src/**DoubleCRT.h**
- src/DoubleCRT.cpp

DoubleCRT Class Reference

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form. More...

`#include <DoubleCRT.h>`

Inheritance diagram for DoubleCRT:



Public Member Functions

	DoubleCRT (const ZZX &poly, const **FHEcontext** &_context, const **IndexSet** &indexSet) Initializing **DoubleCRT** from a ZZX polynomial. More...
	DoubleCRT (const ZZX &poly, const **FHEcontext** &_context)
	DoubleCRT (const ZZX &poly) Context is not specified, use the "active context".
	DoubleCRT (const **FHEcontext** &_context)
	DoubleCRT (const **FHEcontext** &_context, const **IndexSet** &indexSet) Also specify the **IndexSet** explicitly.
DoubleCRT &	**operator=** (const **DoubleCRT** &other)
DoubleCRT &	**operator=** (const ZZX &poly)
DoubleCRT &	**operator=** (const ZZ &num)

DoubleCRT &	**operator=** (const long num)
long	getOneRow (Vec< long > &row, long idx, bool positive=false) const
	Get one row of a polynomial.
long	**getOneRow** (zz_pX &row, long idx) const
void	toPoly (ZZX &p, const IndexSet &s, bool positive=false) const
	Recovering the polynomial in coefficient representation. This yields an integer polynomial with coefficients in [-P/2,P/2], unless the positive flag is set to true, in which case we get coefficients in [0,P-1] (P is the product of all moduli used). Using the optional **IndexSet** param we compute the polynomial reduced modulo the product of only the ptimes in that set.
void	**toPoly** (ZZX &p, bool positive=false) const
bool	**operator==** (const DoubleCRT &other) const
bool	**operator!=** (const DoubleCRT &other) const
DoubleCRT &	**SetZero** ()
DoubleCRT &	**SetOne** ()
void	breakIntoDigits (vector< DoubleCRT > &dgts, long n) const
	Break into n digits,according to the primeSets in context.digits. See Section 3.1.6 of the design document (re-linearization)

void	**addPrimes** (const **IndexSet** &s1) Expand the index set by s1. It is assumed that s1 is disjoint from the current index set.
double	**addPrimesAndScale** (const **IndexSet** &s1) Expand index set by s1, and multiply by Prod_{q in s1}. s1 is disjoint from the current index set, returns log(product).
void	**removePrimes** (const **IndexSet** &s1) Remove s1 from the index set.
const **FHEcontext** &	**getContext** () const
const **IndexMap**< vec_long > &	**getMap** () const
const **IndexSet** &	**getIndexSet** () const
void	**randomize** (const ZZ *seed=NULL) Fills each row i with random ints mod pi, uses NTL's PRG.
void	**sampleSmall** () Coefficients are -1/0/1, Prob[0]=1/2.
void	**sampleHWt** (long Hwt) Coefficients are -1/0/1 with pre-specified number of nonzeros.
void	**sampleGaussian** (double stdev=0.0) Coefficients are Gaussians.

void	**sampleUniform** (const ZZ &B)
	Coefficients are uniform in [-B..B].
void	**scaleDownToSet** (const **IndexSet** &s, long ptxtSpace)
void	**FFT** (const ZZX &poly, const **IndexSet** &s)
void	**reduce** () const

Arithmetic operation

Only the "destructive" versions are used, i.e., a += b is implemented but not a + b.

DoubleCRT &	**Negate** (const **DoubleCRT** &other)
DoubleCRT &	**Negate** ()
DoubleCRT &	**operator+=** (const **DoubleCRT** &other)
DoubleCRT &	**operator+=** (const ZZX &poly)
DoubleCRT &	**operator+=** (const ZZ &num)
DoubleCRT &	**operator+=** (long num)
DoubleCRT &	**operator-=** (const **DoubleCRT** &other)
DoubleCRT &	**operator-=** (const ZZX &poly)
DoubleCRT &	**operator-=** (const ZZ &num)
DoubleCRT &	**operator-=** (long num)
DoubleCRT &	**operator++** ()

DoubleCRT &	**operator--** ()
void	**operator++** (int)
void	**operator--** (int)
DoubleCRT &	**operator*=** (const DoubleCRT &other)
DoubleCRT &	**operator*=** (const ZZX &poly)
DoubleCRT &	**operator*=** (const ZZ &num)
DoubleCRT &	**operator*=** (long num)
void	**Add** (const DoubleCRT &other, bool matchIndexSets=true)
void	**Sub** (const DoubleCRT &other, bool matchIndexSets=true)
void	**Mul** (const DoubleCRT &other, bool matchIndexSets=true)
DoubleCRT &	**operator/=** (const ZZ &num)
DoubleCRT &	**operator/=** (long num)
void	Exp (long k) Small-exponent polynomial exponentiation.
void	**automorph** (long k)
DoubleCRT &	**operator>>=** (long k)

Friends

ostream &	**operator<<** (ostream &s, const DoubleCRT &d)
istream &	**operator>>** (istream &s, DoubleCRT &d)

Detailed Description

Implementatigs polynomials (elements in the ring R_Q) in double-CRT form.

Double-CRT form is a matrix of L rows and phi(m) columns. The i'th row contains the FFT of the element wrt the ith prime, i.e. the evaluations of the polynomial at the primitive mth roots of unity mod the ith prime. The polynomial thus represented is defined modulo the product of all the primes in use.

The list of primes is defined by the data member indexMap. indexMap.getIndexSet() defines the set of indices of primes associated with this DoubleCRT object: they index the primes stored in the associated FHEContext.

Arithmetic operations are computed modulo the product of the primes in use and also modulo Phi_m(X). Arithmetic operations can only be applied to DoubleCRT objects relative to the same context, trying to add/multiply objects that have different FHEContext objects will raise an error.

Constructor & Destructor Documentation

```
DoubleCRT::DoubleCRT ( const ZZX &          poly,
                       const FHEcontext &   _context,
                       const IndexSet &     indexSet
                     )
```

Initializing DoubleCRT from a ZZX polynomial.

Parameters

poly	The ring element itself, zero if not specified
_context	The context for this DoubleCRT object, use "current active context" if not specified

indexSet Which primes to use for this object, if not specified then use all of them

---

The documentation for this class was generated from the following files:

- src/DoubleCRT.h
- src/DoubleCRT.cpp

DynamicCtxtPowers Class Reference

Store powers of X, compute them synamically as needed. More...

`#include <polyEval.h>`

Public Member Functions

	DynamicCtxtPowers (const Ctxt &c, long nPowers)
Ctxt &	**getPower** (long e)
	Returns the e'th power, computing it as needed.
Ctxt &	**at** (long i)
	dp.at(i) and dp[i] both return the i+1st power
Ctxt &	**operator[]** (long i)
const vector< Ctxt > &	**getVector** () const
long	**size** () const
bool	**isPowerComputed** (long i)

Detailed Description

Store powers of X, compute them synamically as needed.

The documentation for this class was generated from the following files:

- src/**polyEval.h**
- src/polyEval.cpp

EncryptedArray Class Reference

A simple wrapper for a smart pointer to an **EncryptedArrayBase**. This is the interface that higher-level code should use. More...

`#include <EncryptedArray.h>`

Public Member Functions

	EncryptedArray (const **FHEcontext** &context, const ZZX &G=ZZX(1, 1))
	constructor: G defaults to the monomial X, **PAlgebraMod** from context
	EncryptedArray (const **FHEcontext** &context, const **PAlgebraMod** &_alMod)
	constructor: G defaults to F0, **PAlgebraMod** explicitly given
EncryptedArray &	**operator=** (const **EncryptedArray** &other)
template<class type >	
const **EncryptedArrayDerived** < type > &	**getDerived** (type) const
	downcast operator example: const EncryptedArrayDerived<PA_GF2>& rep = ea.getDerived(PA_GF2());

Direct access to EncryptedArrayBase methods

const FHEcontext &	**getContext** () const
const PAlgebraMod &	**getAlMod** () const
const long	**getDegree** () const
void	**rotate** (Ctxt &ctxt, long k) const
void	**shift** (Ctxt &ctxt, long k) const
void	**rotate1D** (Ctxt &ctxt, long i, long k, bool dc=false) const
void	**shift1D** (Ctxt &ctxt, long i, long k) const
void	**mat_mul_dense** (Ctxt &ctxt, const PlaintextMatrixBaseInterface &mat) const
void	**mat_mul** (Ctxt &ctxt, const PlaintextMatrixBaseInterface &mat) const
void	**mat_mul1D** (Ctxt &ctxt, const PlaintextMatrixBaseInterface &mat, long dim) const
void	**mat_mul1D** (Ctxt &ctxt, const PlaintextBlockMatrixBaseInterface &mat, long dim) const
void	**mat_mul** (Ctxt &ctxt, const PlaintextBlockMatrixBaseInterface &mat) const
void	**compMat** (CachedPtxtMatrix &cmat, const PlaintextMatrixBaseInterface &mat) const

void	**compMat** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const
void	**compMat** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const
void	**compMat** (CachedDCRTPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const
void	**compMat1D** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const
void	**compMat1D** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const
void	**compMat1D** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const
void	**compMat1D** (CachedDCRTPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const
void	**encode** (ZZX &ptxt, const vector< long > &array) const
void	**encode** (ZZX &ptxt, const vector< ZZX > &array) const
void	**encode** (ZZX &ptxt, const **PlaintextArray** &array) const
void	**encodeUnitSelector** (ZZX &ptxt, long i) const
void	**decode** (vector< long > &array, const ZZX &ptxt) const

void	**decode** (vector< ZZX > &array, const ZZX &ptxt) const
void	**decode** (PlaintextArray &array, const ZZX &ptxt) const
void	**random** (vector< long > &array) const
void	**random** (vector< ZZX > &array) const
void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< long > &ptxt) const
void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< ZZX > &ptxt) const
void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const PlaintextArray &ptxt) const
void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< long > &ptxt) const
void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< ZZX > &ptxt) const
void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, PlaintextArray &ptxt) const
void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const vector< long > &ptxt, long skIdx=0) const
void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const vector< ZZX > &ptxt, long skIdx=0) const
void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const PlaintextArray &ptxt, long skIdx=0) const

void	**select** (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< long > &selector) const
void	**select** (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< ZZX > &selector) const
void	**select** (Ctxt &ctxt1, const Ctxt &ctxt2, const PlaintextArray &selector) const
void	**buildLinPolyCoeffs** (vector< ZZX > &C, const vector< ZZX > &L) const
void	**restoreContext** () const
void	**restoreContextForG** () const
long	**size** () const
long	**dimension** () const
long	**sizeOfDimension** (long i) const
long	**nativeDimension** (long i) const
long	**coordinate** (long i, long k) const
long	**addCoord** (long i, long k, long offset) const

template<class U >

void	rotate1D (vector< U > &out, const vector< U > &in, long i, long offset) const
	rotate an array by offset in the i'th dimension (output should

	not alias input)

Detailed Description

A simple wrapper for a smart pointer to an **EncryptedArrayBase**. This is the interface that higher-level code should use.

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

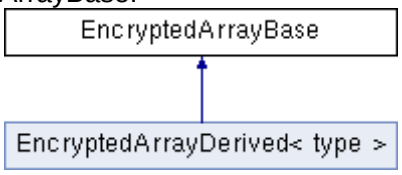
- src/**EncryptedArray.h**

EncryptedArrayBase Class Reference abstract

virtual class for data-movement operations on arrays of slots More...

```
#include <EncryptedArray.h>
```

Inheritance diagram for EncryptedArrayBase:



Public Member Functions

virtual **EncryptedArrayBase** *	**clone** () const =0
virtual const **FHEcontext** &	**getContext** () const =0
virtual const long	**getDegree** () const =0
virtual void	**rotate** (**Ctxt** &ctxt, long k) const =0
	Right rotation as a linear array. E.g., rotating ctxt=Enc(1 2 3 ... n) by k=1 gives Enc(n 1 2 ... n-1)

virtual void	**shift** (**Ctxt** &ctxt, long k) const =0
	Non-cyclic right shift with zero fill E.g., shifting ctxt=Enc(1 2 3 ... n) by k=1 gives Enc(0 1 2... n-1)
virtual void	**rotate1D** (**Ctxt** &ctxt, long i, long k, bool dc=false) const =0
	right-rotate k positions along the i'th dimension More...
virtual void	**shift1D** (**Ctxt** &ctxt, long i, long k) const =0
	Right shift k positions along the i'th dimension with zero fill.
virtual void	**buildLinPolyCoeffs** (vector< ZZX > &C, const vector< ZZX > &L) const =0
	Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..d-1. The result is a coefficient vector C for the linearized polynomial representing M: a polynoamial h in Z/(p^r)[X] of degree < d is sent to .
virtual void	**restoreContext** () const =0
virtual void	**restoreContextForG** () const =0
long	**size** () const
	Total size (# of slots) of hypercube.
long	**dimension** () const
	Number of dimensions of hypercube.

long	**sizeOfDimension** (long i) const
	Size of given dimension.

bool	**nativeDimension** (long i) const
	Is rotations in given dimension a "native" operation?

long	**coordinate** (long i, long k) const
	returns coordinate of index k along the i'th dimension

long	**addCoord** (long i, long k, long offset) const
	adds offset to index k in the i'th dimension

template<class U >

void	**rotate1D** (vector< U > &out, const vector< U > &in, long i, long offset) const
	rotate an array by offset in the i'th dimension (output should not alias input)

Matrix multiplication routines

virtual void	**mat_mul_dense** (**Ctxt** &ctxt, const **PlaintextMatrixBaseInterface** &mat) const =0
	Multiply ctx by plaintext matrix. **Ctxt** is treated as a row matrix v, and replaced by an encryption of v * mat. Optimized for dense matrices.

virtual void	**compMat_dense** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const =0

virtual void	**compMat_dense** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const =0

virtual void	**mat_mul** (**Ctxt** &ctxt, const **PlaintextMatrixBaseInterface** &mat) const =0
	Multiply ctx by plaintext matrix. **Ctxt** is treated as a row matrix v, and replaced by an encryption of v * mat. Optimized for sparse diagonals.
virtual void	**compMat** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const =0
virtual void	**compMat** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const =0
virtual void	**mat_mul** (**Ctxt** &ctxt, const **PlaintextBlockMatrixBaseInterface** &mat) const =0
	Multiply ctx by plaintext block matrix (over the base field/ring). **Ctxt** is treated as a row matrix v, and replaced by an encryption of v*mat. Optimized for sparse diagonals.
virtual void	**compMat** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const =0
virtual void	**compMat** (CachedDCRTPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const =0
virtual void	**mat_mul1D** (**Ctxt** &ctxt, const **PlaintextMatrixBaseInterface** &mat, long dim) const =0
	Multiply ctx by plaintext matrix. **Ctxt** is treated as a row matrix v, and replaced by en encryption of v * mat' where mat' is the block-diagonal matrix defined by mat in dimension dim. Here, mat should represent a D x D matrix, where D is the order of generator dim. We also allow dim to be one greater than the number of generators in zMStar, as if there were an implicit generator of order 1, this is convenient in some applications.

virtual void	**compMat1D** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const =0
virtual void	**compMat1D** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const =0
virtual void	**mat_mul1D** (**Ctxt** &ctxt, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const =0
virtual void	**compMat1D** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const =0
virtual void	**compMat1D** (CachedDCRTPtxtBlockMatrix &smat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const =0

Encoding/decoding methods

virtual void	**encode** (ZZX &ptxt, const vector< long > &array) const =0
virtual void	**encode** (ZZX &ptxt, const vector< ZZX > &array) const =0
virtual void	**encode** (ZZX &ptxt, const **PlaintextArray** &array) const =0
virtual void	**decode** (vector< long > &array, const ZZX &ptxt) const =0
virtual void	**decode** (vector< ZZX > &array, const ZZX &ptxt) const =0
virtual void	**decode** (**PlaintextArray** &array, const ZZX &ptxt) const =0
virtual void	**random** (vector< long > &array) const =0
virtual void	**random** (vector< ZZX > &array) const =0

long	**decode1Slot** (const ZZX &ptxt, long i) const
void	**decode1Slot** (ZZX &slot, const ZZX &ptxt, long i) const
virtual void	**encodeUnitSelector** (ZZX &ptxt, long i) const =0
	Encodes a vector with 1 at position i and 0 everywhere else.

Encoding+encryption/decryption+decoding

virtual void	**encrypt** (**Ctxt** &ctxt, const **FHEPubKey** &pKey, const vector< long > &ptxt) const =0
virtual void	**encrypt** (**Ctxt** &ctxt, const **FHEPubKey** &pKey, const vector< ZZX > &ptxt) const =0
virtual void	**encrypt** (**Ctxt** &ctxt, const **FHEPubKey** &pKey, const **PlaintextArray** &ptxt) const =0
virtual void	**decrypt** (const **Ctxt** &ctxt, const **FHESecKey** &sKey, vector< long > &ptxt) const =0
virtual void	**decrypt** (const **Ctxt** &ctxt, const **FHESecKey** &sKey, vector< ZZX > &ptxt) const =0
virtual void	**decrypt** (const **Ctxt** &ctxt, const **FHESecKey** &sKey, **PlaintextArray** &ptxt) const =0
virtual void	**skEncrypt** (**Ctxt** &ctxt, const **FHESecKey** &sKey, const vector< long > &ptxt, long skIdx=0) const =0
virtual void	**skEncrypt** (**Ctxt** &ctxt, const **FHESecKey** &sKey, const vector< ZZX > &ptxt, long skIdx=0) const =0
virtual void	**skEncrypt** (**Ctxt** &ctxt, const **FHESecKey** &sKey,

	const **PlaintextArray** &ptxt, long skIdx=0) const =0
long	**decrypt1Slot** (const **Ctxt** &ctxt, const **FHESecKey** &sKey, long i) const
void	**decrypt1Slot** (ZZX &slot, const **Ctxt** &ctxt, const **FHESecKey** &sKey, long i) const
virtual void	**select** (**Ctxt** &ctxt1, const **Ctxt** &ctxt2, const vector< long > &selector) const =0
	MUX: ctxt1 = ctxt1*selector + ctxt2*(1-selector)
virtual void	**select** (**Ctxt** &ctxt1, const **Ctxt** &ctxt2, const vector< ZZX > &selector) const =0
virtual void	**select** (**Ctxt** &ctxt1, const **Ctxt** &ctxt2, const **PlaintextArray** &selector) const =0

Detailed Description

virtual class for data-movement operations on arrays of slots

An object ea of type **EncryptedArray** stores information about an **FHEcontext** context, and a monic polynomial G. If context defines parameters m, p, and r, then ea is a helper abject that supports encoding/decoding and encryption/decryption of vectors of plaintext slots over the ring (Z/(p^r)[X])/(G).

The polynomial G should be irreducble over Z/(p^r) (this is not checked). The degree of G should divide the multiplicative order of p modulo m (this is checked). Currently, the following restriction is imposed:

either r == 1 or deg(G) == 1 or G == factors[0].

ea stores objects in the polynomial ring Z/(p^r)[X].

Just as for the class PAlegebraMod, if p == 2 and r == 1, then these polynomials are represented as GF2X's, and otherwise as zz_pX's. Thus, the types of these objects are not determined until run time. As such, we need to use a class heirarchy, which mirrors that of **PAlgebraMod**, as follows.

EncryptedArrayBase is a virtual class

EncryptedArrayDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.

The class **EncryptedArray** is a simple wrapper around a smart pointer to an **EncryptedArrayBase** object: copying an **EncryptedArray** object results is a "deep copy" of the underlying object of the derived class.

Member Function Documentation

virtual void EncryptedArrayBase::rotate1D (**Ctxt** **& ctxt,**
long i,
long k,
bool dc = `false`
) const pure virtual

right-rotate k positions along the i'th dimension

Parameters

dc means "don't care", which means that the caller guarantees that only zero elements rotate off the end – this allows for some optimizations that would not otherwise be possible

Implemented in **EncryptedArrayDerived< type >**.

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

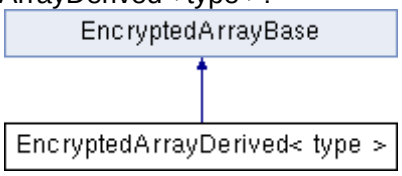
- src/**EncryptedArray.h**

EncryptedArrayDerived< type > Class Template Reference

Derived concrete implementation of **EncryptedArrayBase**. More...

`#include <EncryptedArray.h>`

Inheritance diagram for EncryptedArrayDerived< type >:



Public Member Functions

	EncryptedArrayDerived (const FHEcontext &_context, const RX &_G, const PAlgebraMod &_tab)
	EncryptedArrayDerived (const EncryptedArrayDerived &other)
EncryptedArrayDerived &	**operator=** (const EncryptedArrayDerived &other)
virtual EncryptedArrayBase *	**clone** () const
const RX &	**getG** () const
const Mat< R > &	**getNormalBasisMatrix** () const
const Mat< R > &	**getNormalBasisMatrixInverse** () const
void	**initNormalBasisMatrix** () const
virtual void	**restoreContext** () const
virtual void	**restoreContextForG** () const
virtual const FHEcontext &	**getContext** () const
virtual const long	**getDegree** () const

const PAlgebraModDerived< type > &	**getTab** () const
virtual void	rotate (Ctxt &ctxt, long k) const
	Right rotation as a linear array. E.g., rotating ctxt=Enc(1 2 3 ... n) by k=1 gives Enc(n 1 2 ... n-1)
virtual void	shift (Ctxt &ctxt, long k) const
	Non-cyclic right shift with zero fill E.g., shifting ctxt=Enc(1 2 3 ... n) by k=1 gives Enc(0 1 2... n-1)
virtual void	rotate1D (Ctxt &ctxt, long i, long k, bool dc=false) const
	right-rotate k positions along the i'th dimension More...
virtual void	shift1D (Ctxt &ctxt, long i, long k) const
	Right shift k positions along the i'th dimension with zero fill.
void	**rec_mul** (long dim, Ctxt &res, const Ctxt &pdata, const vector< long > &idx, const PlaintextMatrixInterface< type > &mat, const vector< long > &dimx) const
virtual void	mat_mul_dense (Ctxt &ctxt, const PlaintextMatrixBaseInterface &mat) const
	Multiply ctx by plaintext matrix. Ctxt is treated as a row matrix v, and replaced by an encryption of v * mat. Optimized for dense matrices.
virtual void	**compMat_dense** (CachedPtxtMatrix &cmat, const PlaintextMatrixBaseInterface &mat) const
virtual void	**compMat_dense** (CachedDCRTPtxtMatrix &cmat,

	const **PlaintextMatrixBaseInterface** &mat) const
virtual void	**mat_mul** (**Ctxt** &ctxt, const **PlaintextMatrixBaseInterface** &mat) const
	Multiply ctx by plaintext matrix. **Ctxt** is treated as a row matrix v, and replaced by an encryption of v * mat. Optimized for sparse diagonals.
virtual void	**compMat** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const
virtual void	**compMat** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat) const
virtual void	**mat_mul** (**Ctxt** &ctxt, const **PlaintextBlockMatrixBaseInterface** &mat) const
	Multiply ctx by plaintext block matrix (over the base field/ring). **Ctxt** is treated as a row matrix v, and replaced by an encryption of v*mat. Optimized for sparse diagonals.
virtual void	**compMat** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const
virtual void	**compMat** (CachedDCRTPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat) const
virtual void	**mat_mul1D** (**Ctxt** &ctxt, const **PlaintextMatrixBaseInterface** &mat, long dim) const
	Multiply ctx by plaintext matrix. **Ctxt** is treated as a row matrix v, and replaced by en encryption of v * mat' where mat' is the block-diagonal matrix defined by mat in dimension dim. Here, mat should represent a D x D

	matrix, where D is the order of generator dim. We also allow dim to be one greater than the number of generators in zMStar, as if there were an implicit generator of order 1, this is convenient in some applications.
virtual void	**compMat1D** (CachedPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const
virtual void	**compMat1D** (CachedDCRTPtxtMatrix &cmat, const **PlaintextMatrixBaseInterface** &mat, long dim) const
virtual void	**mat_mul1D** (**Ctxt** &ctxt, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const
virtual void	**compMat1D** (CachedPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const
virtual void	**compMat1D** (CachedDCRTPtxtBlockMatrix &cmat, const **PlaintextBlockMatrixBaseInterface** &mat, long dim) const
virtual void	**encode** (ZZX &ptxt, const vector< long > &array) const
virtual void	**encode** (ZZX &ptxt, const vector< ZZX > &array) const
virtual void	**encode** (ZZX &ptxt, const **PlaintextArray** &array) const
virtual void	**encodeUnitSelector** (ZZX &ptxt, long i) const
	Encodes a vector with 1 at position i and 0 everywhere

	else.
virtual void	**decode** (vector< long > &array, const ZZX &ptxt) const
virtual void	**decode** (vector< ZZX > &array, const ZZX &ptxt) const
virtual void	**decode** (PlaintextArray &array, const ZZX &ptxt) const
virtual void	**random** (vector< long > &array) const
virtual void	**random** (vector< ZZX > &array) const
virtual void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< long > &ptxt) const
virtual void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< ZZX > &ptxt) const
virtual void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const PlaintextArray &ptxt) const
virtual void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< long > &ptxt) const
virtual void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< ZZX > &ptxt) const
virtual void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, PlaintextArray &ptxt) const
virtual void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const vector< long > &ptxt, long skIdx=0) const
virtual void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const

	vector< ZZX > &ptxt, long skIdx=0) const
virtual void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const PlaintextArray &ptxt, long skIdx=0) const
virtual void	select (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< long > &selector) const
	MUX: ctxt1 = ctxt1*selector + ctxt2*(1-selector)
virtual void	**select** (Ctxt &ctxt1, const Ctxt &ctxt2, const vector< ZZX > &selector) const
virtual void	**select** (Ctxt &ctxt1, const Ctxt &ctxt2, const PlaintextArray &selector) const
virtual void	buildLinPolyCoeffs (vector< ZZX > &C, const vector< ZZX > &L) const
	Linearized polynomials. L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..d-1. The result is a coefficient vector C for the linearized polynomial representing M: a polynoamial h in Z/(p^r)[X] of degree < d is sent to .
void	**encode** (ZZX &ptxt, const vector< RX > &array) const
void	**decode** (vector< RX > &array, const ZZX &ptxt) const
void	**random** (vector< RX > &array) const

void	**encrypt** (Ctxt &ctxt, const FHEPubKey &pKey, const vector< RX > &ptxt) const
void	**decrypt** (const Ctxt &ctxt, const FHESecKey &sKey, vector< RX > &ptxt) const
void	**skEncrypt** (Ctxt &ctxt, const FHESecKey &sKey, const vector< RX > &ptxt, long skIdx=0) const
virtual void	**buildLinPolyCoeffs** (vector< RX > &C, const vector< RX > &L) const

Public Member Functions inherited from EncryptedArrayBase

Detailed Description

template<class type>
class EncryptedArrayDerived< type >

Derived concrete implementation of EncryptedArrayBase.

Member Function Documentation

template<class type >

void EncryptedArrayDerived< type >::rotate1D (Ctxt & ctxt,
long i,
long k,
bool dc = `false`
) const virtual

right-rotate k positions along the i'th dimension

Parameters

dc means "don't care", which means that the caller guarantees that only zero elements rotate

off the end – this allows for some optimizations that would not otherwise be possible

Implements **EncryptedArrayBase**.

The documentation for this class was generated from the following files:

- src/**EncryptedArray.h**
- src/EncryptedArray.cpp

EvalMap Class Reference

Class that provides the functionality for the linear transforms used in boostrapping. The constructor is invoked with three arguments: More...

`#include <EvalMap.h>`

Public Member Functions

	EvalMap (const **EncryptedArray** &_ea, const Vec< long > &mvec, bool _invert, bool normal_basis=true)
void	**apply** (**Ctxt** &ctxt) const

Detailed Description

Class that provides the functionality for the linear transforms used in boostrapping. The constructor is invoked with three arguments:

- an **EncryptedArray** object ea
- an integer vector mvec
- a boolean flag invert The mvec vector specifies the factorization of m to use in the "powerful basis" decomposition.

If the invert flag is false, the forward transformation is used. This transformation views the slots as being packed with powerful-basis coefficients and performs a multi-point polynomial evaluation. This is the second transformation used in bootstrapping.

If invert flag is true, the inverse transformation is used. In addition, the current implementation folds into the inverse transformation a transformation that moves the coefficients in each slot into a normal-basis representation, which helps with the unpacking procedure.

The constructor precomputes certain values, but the linear transformation itself is effected using the apply method.

Note that the factorization in mvec must correspond to the generators used in **PAlgebra**. The best way to ensure this is to used directly the output of the params program, which will supply values for mvec (to be used here), and gens and ords (to be used in initialize the **FHEcontext**).

The documentation for this class was generated from the following files:

- src/**EvalMap.h**
- src/EvalMap.cpp

EvalMap Class Reference

Class that provides the functionality for the linear transforms used in boostrapping. The constructor is invoked with three arguments: More...

`#include <EvalMap.h>`

Public Member Functions

	EvalMap (const **EncryptedArray** &_ea, const Vec< long > &mvec, bool _invert, bool normal_basis=true)
void	**apply** (**Ctxt** &ctxt) const

Detailed Description

Class that provides the functionality for the linear transforms used in boostrapping. The constructor is invoked with three arguments:

- an **EncryptedArray** object ea
- an integer vector mvec
- a boolean flag invert The mvec vector specifies the factorization of m to use in the "powerful basis" decomposition.

If the invert flag is false, the forward transformation is used. This transformation views the slots as being packed with powerful-basis coefficients and performs a multi-point polynomial evaluation. This is the second transformation used in bootstrapping.

If invert flag is true, the inverse transformation is used. In addition, the current implementation folds into the inverse transformation a transformation that moves the coefficients in each slot into a normal-basis representation, which helps with the unpacking procedure.

The constructor precomputes certain values, but the linear transformation itself is effected using the apply method.

Note that the factorization in mvec must correspond to the generators used in **PAlgebra**. The best way to ensure this is to used directly the output of the params program, which will supply values for mvec (to be used here), and gens and ords (to be used in initialize the **FHEcontext**).

The documentation for this class was generated from the following files:

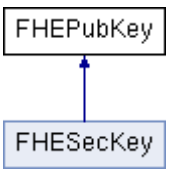
- src/**EvalMap.h**
- src/EvalMap.cpp

FHEPubKey Class Reference

The public key. More...

`#include <FHE.h>`

Inheritance diagram for FHEPubKey:



Public Member Functions

	FHEPubKey (const FHEcontext &_context)
	FHEPubKey (const FHEPubKey &other)
void	**clear** ()
bool	**operator==** (const FHEPubKey &other) const
bool	**operator!=** (const FHEPubKey &other) const
const FHEcontext &	**getContext** () const
long	**getPtxtSpace** () const
bool	**keyExists** (long keyID)
long	getSKeyWeight (long keyID=0) const The Hamming weight of the secret key.
bool	isReachable (long k, long keyID=0) const Is it possible to re-linearize the automorphism X -> X^k See Section 3.2.2 in the design document (KeySwitchMap)
void	setKeySwitchMap (long keyId=0) Compute the reachability graph of key-switching matrices See Section 3.2.2 in the design document (KeySwitchMap)
long	Encrypt (Ctxt &ciphertxt, const ZZX &plaintxt, long ptxtSpace=0, bool highNoise=false) const

	Encrypts plaintext, result returned in the ciphertext argument. The returned value is the plaintext-space for that ciphertext. When called with highNoise=true, returns a ciphertext with noise level~q/8.
bool	**isBootstrappable** () const
void	**reCrypt** (Ctxt &ctxt)

Find key-switching matrices

const KeySwitch &	getKeySWmatrix (const SKHandle &from, long toID=0) const
	Find a key-switching matrix by its indexes. If no such matrix exists it returns a dummy matrix with toKeyID==-1.
const KeySwitch &	**getKeySWmatrix** (long fromSPower, long fromXPower, long fromID=0, long toID=0) const
bool	**haveKeySWmatrix** (const SKHandle &from, long toID=0) const
bool	**haveKeySWmatrix** (long fromSPower, long fromXPower, long fromID=0, long toID=0) const
const KeySwitch &	getAnyKeySWmatrix (const SKHandle &from) const
	Is there a matrix from this key to *any* base key?
bool	**haveAnyKeySWmatrix** (const SKHandle &from) const
const KeySwitch &	getNextKSWmatrix (long fromXPower, long fromID=0) const
	Get the next matrix to use for multi-hop automorphism See Section 3.2.2 in the design document.

Static Public Member Functions

static long	**ePlusR** (long p)

Friends

class	**FHESecKey**
ostream &	**operator<<** (ostream &str, const **FHEPubKey** &pk)
istream &	**operator>>** (istream &str, **FHEPubKey** &pk)

Detailed Description

The public key.

The documentation for this class was generated from the following files:

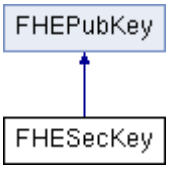
- src/**FHE.h**
- src/FHE.cpp
- src/recryption.cpp

FHESecKey Class Reference

The secret key. More...

`#include <FHE.h>`

Inheritance diagram for FHESecKey:



Public Member Functions

	FHESecKey (const **FHEcontext** &_context)

bool	**operator==** (const **FHESecKey** &other) const
bool	**operator!=** (const **FHESecKey** &other) const
void	**clear** ()
long	**ImportSecKey** (const **DoubleCRT** &sKey, long hwt, long ptxtSpace=0, bool onlyLinear=false)
long	**GenSecKey** (long hwt, long ptxtSpace=0, bool onlyLinear=false)
void	**GenKeySWmatrix** (long fromSPower, long fromXPower, long fromKeyIdx=0, long toKeyIdx=0, long ptxtSpace=0)
void	**Decrypt** (ZZX &plaintxt, const **Ctxt** &ciphertxt) const
void	**Decrypt** (ZZX &plaintxt, const **Ctxt** &ciphertxt, ZZX &f) const
	Debugging version, returns in f the polynomial before reduction modulo the ptxtSpace.
long	**Encrypt** (**Ctxt** &ctxt, const ZZX &ptxt, long ptxtSpace=0, long skIdx=0) const
	Symmetric encryption using the secret key.
long	**genRecryptData** ()
	Generate bootstrapping data if needed, returns index of key.

Public Member Functions inherited from FHEPubKey

Public Attributes

vector< **DoubleCRT** >	**sKeys**

Friends

ostream &	**operator<<** (ostream &str, const FHESecKey &sk)
istream &	**operator>>** (istream &str, FHESecKey &sk)

Additional Inherited Members

Static Public Member Functions inherited from FHEPubKey

Detailed Description

The secret key.

Member Function Documentation

void FHESecKey::GenKeySWmatrix (long fromSPower,
long fromXPower,
long fromKeyIdx = 0,
long toKeyIdx = 0,
long ptxtSpace = 0
)

Generate a key-switching matrix and store it in the public key. The i'th column of the matrix encrypts fromKey*B1*B2*...*B{i-1}*Q under toKey, relative to the largest modulus (i.e., all primes) and plaintext space p. Q is the product of special primes, and the Bi's are the products of primes in the i'th digit. The plaintext space defaults to 2^r, as defined by context.mod2r.

long FHESecKey::GenSecKey (long hwt,
long ptxtSpace = 0,
bool onlyLinear = false
) inline

Key generation: This procedure generates a single secret key, pushes it onto the sKeys list using ImportSecKey from above.

```
long FHESecKey::ImportSecKey ( const DoubleCRT &  sKey,
                               long                hwt,
                               long                ptxtSpace = 0,
                               bool                onlyLinear = false
                             )
```

We allow the calling application to choose a secret-key polynomial by itself, then insert it into the **FHESecKey** object, getting the index of that secret key in the sKeys list. If this is the first secret-key for this object then the procedure below also generates a corresponding public encryption key. It is assumed that the context already contains all parameters.

The documentation for this class was generated from the following files:

- src/**FHE.h**
- src/FHE.cpp

FHEtimer Class Reference

A simple class to accumulate time. More...

`#include <timing.h>`

Public Member Functions

	FHEtimer (const char *_name, const char *_loc)
void	**reset** ()
double	**getTime** () const
long	**getNumCalls** () const

Public Attributes

const char *	**name**
const char *	**loc**
FHE_atomic_ulong	**counter**
FHE_atomic_long	**numCalls**

Detailed Description

A simple class to accumulate time.

The documentation for this class was generated from the following files:

- src/**timing.h**
- src/timing.cpp

FlowEdge Class Reference

An edge in a flow graph. More...

`#include <matching.h>`

Public Member Functions

	FlowEdge (long c=0, long f=0)

Public Attributes

long	**capacity**

long	**flow**

Detailed Description

An edge in a flow graph.

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

- src/**matching.h**

FullBinaryTree< T > Class Template Reference

A simple implementation of full binary trees (each non-leaf has 2 children) More...

`#include <permutations.h>`

Public Member Functions

	FullBinaryTree (long _aux=0)
	FullBinaryTree (const T &d, long _aux=0)
void	**putDataInRoot** (const T &d)
long	**size** ()
TreeNode< T > &	**operator[]** (long i)
const TreeNode< T > &	**operator[]** (long i) const

TreeNode< T > &	**at** (long i)
const TreeNode< T > &	**at** (long i) const
T &	**DataOfNode** (long i)
const T &	**DataOfNode** (long i) const
long	**getAuxKey** () const
void	**setAuxKey** (long _aux)
long	**getNleaves** () const
long	**firstLeaf** () const
long	**nextLeaf** (long i) const
long	**prevLeaf** (long i) const
long	**lastLeaf** () const
long	**rootIdx** () const
long	**parentIdx** (long i) const

long	**leftChildIdx** (long i) const
long	**rightChildIdx** (long i) const
void	**printout** (ostream &s, long idx=0) const
long	**addChildren** (long prntIdx, const T &leftData, const T &rightData)
void	**collapseToRoot** () Remove all nodes in the tree except for the root.

Detailed Description

template<class T>
class FullBinaryTree< T >

A simple implementation of full binary trees (each non-leaf has 2 children)

Member Function Documentation

template<class T >

long FullBinaryTree< T >::addChildren (long prntIdx,
const T & leftData,
const T & rightData
)

If the parent is a leaf, add to it tho children with the given data, else just update the data of the two children of this parent. Returns the index of the left child, the right-child index is one more than the left-child index.

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

- src/**permutations.h**

GenDescriptor Class Reference

A minimal description of a generator for the purpose of building tree. More...

`#include <permutations.h>`

Public Member Functions

	GenDescriptor (long _order, bool _good, long gen=0)

Public Attributes

long	**genIdx**
long	**order**
bool	**good**

Detailed Description

A minimal description of a generator for the purpose of building tree.

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

- src/**permutations.h**

GeneralBenesNetwork Class Reference

Implementation of generalized Benes Permutation Network. More...

`#include <permutations.h>`

Public Member Functions

long	**getDepth** () const
long	**getSize** () const
long	**getNumLevels** () const
const Vec< short > &	**getLevel** (long i) const
long	**levelToDepthMap** (long i) const
long	**shamt** (long i) const
	GeneralBenesNetwork (const **Permut** &perm)
bool	**testNetwork** (const **Permut** &perm) const

Static Public Member Functions

static long	**depth** (long n)
static long	**levelToDepthMap** (long n, long k, long i)
static long	**shamt** (long n, long k, long i)

Detailed Description

Implementation of generalized Benes Permutation Network.

Member Function Documentation

static long GeneralBenesNetwork::depth (long n) inlinestatic

computes recursion depth k for generalized Benes network of size n. the actual number of levels in the network is 2*k-1

static long GeneralBenesNetwork::levelToDepthMap (long n,
long k,
long i
) inlinestatic

maps a level number i = 0..2*k-2 to a recursion depth d = 0..k-1 using the formula d = (k-1)-|(k-1)-i|

static long GeneralBenesNetwork::shamt (long n,
long k,
long i
) inlinestatic

shift amount for level number i=0..2*k-2 using the formula ceil(floor(n/2^d) / 2), where d = levelToDepthMap(i)

The documentation for this class was generated from the following files:

- src/**permutations.h**
- src/BenesNetwork.cpp

GeneratorTrees Class Reference

A vector of generator trees, one per generator in Zm*/(p) More...

```
#include <permutations.h>
```

Public Member Functions

long	**numLayers** () const
long	**numTrees** () const
long	**getSize** () const
OneGeneratorTree &	**operator[]** (long i)
const OneGeneratorTree &	**operator[]** (long i) const
OneGeneratorTree &	**at** (long i)
const OneGeneratorTree &	**at** (long i) const
OneGeneratorTree &	**getGenTree** (long i)
const OneGeneratorTree &	**getGenTree** (long i) const
const Permut &	**mapToCube** () const
const Permut &	**mapToArray** () const
Permut &	**mapToCube** ()
Permut &	**mapToArray** ()
long	**mapToCube** (long i) const
long	**mapToArray** (long i) const

void	**getCubeDims** (Vec< long > &dims) const
void	**getCubeSubDims** (Vec< long > &dims) const
long	**buildOptimalTrees** (const Vec< **GenDescriptor** > &vec, long depthBound)
void	**ComputeCubeMapping** () Computes permutations mapping between linear array and the cube. More...

Friends

ostream &	**operator<<** (ostream &s, const **GeneratorTrees** &t)

Detailed Description

A vector of generator trees, one per generator in Zm*/(p)

Member Function Documentation

long GeneratorTrees::buildOptimalTrees (const Vec< GenDescriptor > & vec,
long depthBound
)

Compute the trees corresponding to the "optimal" way of breaking a permutation into dimensions, subject to some constraints. Returns the cost (# of 1D shifts) of this colution. Returns NTL_MAX_LONG if no solution

void GeneratorTrees::ComputeCubeMapping ( )

Computes permutations mapping between linear array and the cube.

If the cube dimensions (i.e., leaves of tree) are n1,n2,...,nt and N= n_j is the size of the cube, then an integer i can be represented in either the mixed base of the n_j's or in "CRT basis" relative to the leaves: Namely either i = {j<=t} i_j * {k>j} n_k, or i = i'_leaf * leaf.e mod N.

The breakPermByDim procedure expects its input in the mixed-base representation, and the maps are used to convert back and forth. Specifically, let (i'_1,...,i'_t) be the CRT representation of i in this cube, and j = {j=1}^t i'_j * {k>j} n_k, then we have map2cube[i]=j and map2array[j]=i.

void GeneratorTrees::getCubeDims (Vec< long > & dims) const

Get the "crude" cube dimensions corresponding to the vector of trees, the ordered vector with one dimension per tree

void GeneratorTrees::getCubeSubDims (Vec< long > & dims) const

Get the "fine" cube dimensions corresponding to the vector of trees, the ordered vector with one dimension per leaf in all the trees.

The documentation for this class was generated from the following files:

- src/**permutations.h**
- src/OptimizePermutations.cpp
- src/permutations.cpp

HyperCube< T > Class Template Reference

A multi-dimensional cube. More...

`#include <hypercube.h>`

Public Member Functions

	HyperCube (const **CubeSignature** &_sig)
	initialzie a **HyperCube** with a **CubeSignature**

HyperCube &	operator= (const HyperCube< T > &other)
	assignment: signatures must be the same
bool	operator== (const HyperCube< T > &other) const
	equality testing: signaturees must be the same
bool	**operator!=** (const HyperCube< T > &other) const
const CubeSignature &	getSig () const
	const ref to signature
Vec< T > &	getData ()
const Vec< T > &	getData () const
	read-only ref to data vector
long	getSize () const
	total size of cube
long	getNumDims () const
	number of dimensions
long	getDim (long d) const
	size of dimension d

long	**getProd** (long d) const
	product of sizes of dimensions d, d+1, ...
long	**getProd** (long from, long to) const
	product of sizes of dimensions from, from+1, ..., to-1
long	**getCoord** (long i, long d) const
	get coordinate in dimension d of index i
long	**addCoord** (long i, long d, long offset) const
	add offset to coordinate in dimension d of index i
long	**numSlices** (long d=1) const
	number of slices
long	**sliceSize** (long d=1) const
	size of one slice
long	**numCols** () const
	number of columns
T &	**at** (long i)
	reference to element at position i, with bounds check
T &	**operator[]** (long i)

	reference to element at position i, without bounds check
const T &	**at** (long i) const
	read-only reference to element at position i, with bounds check
const T &	**operator[]** (long i) const
	read-only reference to element at position i, without bounds check
void	**rotate1D** (long i, long k)
	rotate k positions along the i'th dimension
void	**shift1D** (long i, long k)
	Shift k positions along the i'th dimension with zero fill.

Detailed Description

template<class T>
class HyperCube< T >

A multi-dimensional cube.

Such an object is initialzied with a **CubeSignature**: a reference to the signature is stored with the cube, and so the signature must remain alive during the lifetime of the cube, to prevent dangling pointers.

Member Function Documentation

template<class T>

Vec<T>& HyperCube< T >::getData ( ) inline

read/write ref to the data vector. Note that the length of data is fixed upon construction, so it cannot be changed through this ref.

The documentation for this class was generated from the following files:

- src/hypercube.h
- src/hypercube.cpp

IndexMap< T > Class Template Reference

IndexMap<T> implements a generic map indexed by a dynamic index set. More...

`#include <IndexMap.h>`

Public Member Functions

	IndexMap ()
	The empty map.
	IndexMap (**IndexMapInit**< T > *_init)
	A map with an initialization object. This associates a method for initializing new elements in the map. When a new index j is added to the index set, an object t of type T is created using the default constructor for T, after which the function _init->init(t) is called (t is passed by reference). To use this feature, you need to derive a subclass of IndexMapInit<T> that defines the init function. This "helper object" should be created using operator new, and the pointer is "exclusively owned" by the map object.
const **IndexSet** &	**getIndexSet** () const

	Get the underlying index set.
T &	**operator[]** (long j)
	Access functions: will raise an error if j does not belong to the current index set.
const T &	**operator[]** (long j) const
void	**insert** (long j)
	Insert indexes to the **IndexSet**. Insertion will cause new T objects to be created, using the default constructor, and possibly initilized via the IndexMapInit<T> pointer.
void	**insert** (const **IndexSet** &s)
void	**remove** (long j)
	Delete indexes from **IndexSet**, may cause objects to be destroyed.
void	**remove** (const **IndexSet** &s)
void	**clear** ()

Detailed Description

template<class T>
class IndexMap< T >

IndexMap<T> implements a generic map indexed by a dynamic index set.

Additionally, it allows new elements of the map to be initialized in a flexible manner.

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

- src/IndexMap.h

IndexMapInit< T > Class Template Reference abstract

Initializing elements in an **IndexMap**. More...

`#include <IndexMap.h>`

Public Member Functions

virtual void	**init** (T &)=0
	Initialization function, override with initialization code.
virtual **IndexMapInit**< T > *	**clone** () const =0
	Cloning a pointer, override with code to create a fresh copy.

Detailed Description

template<class T>
class IndexMapInit< T >

Initializing elements in an **IndexMap**.

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

- src/IndexMap.h

IndexSet Class Reference

A dynamic set of non-negative integers. More...

`#include <IndexSet.h>`

Public Member Functions

	IndexSet (long low, long high)
	IndexSet (long j)
long	first () const
	Returns the first element, 0 if the set is empty.
long	last () const
	Returns the last element, -1 if the set is empty.
long	next (long j) const
	Returns the next element after j, if any; otherwise j+1.
long	**prev** (long j) const
long	card () const
	The cardinality of the set.
bool	contains (long j) const
	Returns true iff the set contains j.
bool	contains (const IndexSet &s) const
	Returns true iff the set contains s.

bool	**disjointFrom** (const **IndexSet** &s) const
	Returns true iff the set is disjoint from s.
bool	**operator==** (const **IndexSet** &s) const
bool	**operator!=** (const **IndexSet** &s) const
void	**clear** ()
	Set to the empty set.
void	**insert** (long j)
	Add j to the set.
void	**remove** (long j)
	Remove j from the set.
void	**insert** (const **IndexSet** &s)
	Add s to the set (union)
void	**remove** (const **IndexSet** &s)
	Remove s from the set (set minus)
void	**retain** (const **IndexSet** &s)
	Retains only those elements that are also in s (intersection)
bool	**isInterval** () const
	Is this set a contiguous interval?

Static Public Member Functions

static const **IndexSet** &	**emptySet** ()
	Read-only access to an empty set.

Detailed Description

A dynamic set of non-negative integers.

You can iterate through a set as follows:

```
for (long i = s.first(); i <= s.last(); i = s.next(i)) ...
for (long i = s.last(); i >= s.first(); i = s.prev(i)) ...
```

The documentation for this class was generated from the following files:

- src/**IndexSet.h**
- src/IndexSet.cpp

KeySwitch Class Reference

Key-switching matrices. More...

```
#include <FHE.h>
```

Public Member Functions

	KeySwitch (long sPow=0, long xPow=0, long fromID=0, long toID=0, long p=0)
	KeySwitch (const **SKHandle** &_fromKey, long fromID=0, long toID=0, long p=0)
bool	**operator==** (const **KeySwitch** &other) const
bool	**operator!=** (const **KeySwitch** &other) const

unsigned long	**NumCols** () const

void	**verify** (**FHESecKey** &sk)
	A debugging method.

void	**readMatrix** (istream &str, const **FHEcontext** &context)
	Read a key-switching matrix from input.

Static Public Member Functions

static const **KeySwitch** &	**dummy** ()
	returns a dummy static matrix with toKeyId == -1

Public Attributes

SKHandle	**fromKey**
long	**toKeyID**
long	**ptxtSpace**
vector< **DoubleCRT** >	**b**
ZZ	**prgSeed**

Detailed Description

Key-switching matrices.

There are basically two approaches for how to do key-switching: either decompose the mod-q ciphertext into bits (or digits) to make it low-norm, or perform the key-switching operation mod Q>>q. The tradeoff is that when decomposing the (coefficients of the) ciphertext into t digits, we need to

increase the size of the key-switching matrix by a factor of t (and the running time similarly grows). On the other hand if we do not decompose at all then we need to work modulo Q>q^2, which means that the bitsize of our largest modulus q0 more than doubles (and hence also the parameter m more than doubles). In general if we decompose into digits of size B then we need to work with Q>q*B.)

The part of the spectrum where we expect to find the sweet spot is when we decompose the ciphertext into digits of size B=q0^{1/t} for some small constant t (maybe t=2,3 or so). This means that our largest modulus has to be Q>q0^{1+1/t}, which increases also the parameter m by a factor (1+1/t). It also means that for key-switching in the top levels we would break the ciphertext to t digits, hence the key-switching matrix will have t columns.

A key-switch matrix W[s'->s] converts a ciphertext-part with respect to secret-key polynomial s' into a canonical cipehrtext (i.e. a two-part ciphertext with respect to (1,s)). The matrix W is a 2-by-t matrix of **DoubleCRT** objects. The bottom row are just (psudo)random elements. Then for column i, if the bottom element is ai then the top element is set as bi = P*Bi*s' + p*ei - s * ai mod P*q0, where p is the plaintext space (i.e. 2 or 2^r) and Bi is the product of the digits-sizes corresponding to columns 0...i-1. (For example if we have digit sizes 3,5,7 then B0=1, B1=3, B2=15 and B3=105.) Also, q0 is the product of all the "ciphertext primes" and P is roughly the product of all the special primes. (Actually, if Q is the product of all the special primes then P=Q*(Q^{-1} mod p).)

In this implementation we save some space, by keeping only a PRG seed for generating the pseudo-random elements, rather than the elements themselves.

To convert a cipehrtext part R, we break R into digits R = sum_i Bi Ri, then set (q0,q1)^T = sum_i Ri * column-i. Note that we have <(1,s),(q0,q1)> = sum_i Ri*(s*ai - s*ai + p*ei +P*Bi*s') = P * sum_i Bi*Ri * s' + p sum_i Ri*ei = P * R * s' + p*a-small-element (mod P*q0) where the last element is small since the ei's are small and |Ri|<B. Note that if the ciphertext is encrypted relative to plaintext space p' and then key-switched with matrices W relative to plaintext space p, then we get a mew ciphertxt with noise p'*small+p*small, so it is valid relative to plaintext space GCD(p',p).

The matrix W is defined modulo Q>t*B*sigma*q0 (with sigma a bound on the size of the ei's), and Q is the product of all the small primes in our moduli chain. However, if p is much smaller than B then is is enough to use W mod Qi with Qi a smaller modulus, Q>p*sigma*q0. Also note that if p<Br then we will be using only first r columns of the matrix W.

The documentation for this class was generated from the following files:

- src/**FHE.h**
- src/FHE.cpp

LabeledEdge Class Reference

A generic directed edge in a graph with some labels. More...

`#include <matching.h>`

Public Member Functions

	LabeledEdge (long f, long t, long l=0, long c=0)

Public Attributes

long	**from**
long	**to**
long	**label**
long	**color**

Detailed Description

A generic directed edge in a graph with some labels.

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

- src/**matching.h**

LabeledVertex Class Reference

A generic node in a graph with some labels. More...

`#include <matching.h>`

Public Member Functions

	LabeledVertex (long n, long l=0)
void	**addEdge** (long nn, long l=0, long c=0)
void	**addNeighbor** (long nn, long l=0, long c=0)

Public Attributes

long	**name**
long	**label**
LNeighborList	**neighbors**

Detailed Description

A generic node in a graph with some labels.

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

- src/**matching.h**

MappingData< type > Class Template Reference

Auxilliary structure to support encoding/decoding slots. More...

`#include <PAlgebra.h>`

Public Member Functions

const RX &	**getG** () const

long	**getDegG** () const

void	**restoreContextForG** () const

Friends

class	**PAlgebraModDerived< type >**

Detailed Description

template<class type>
class MappingData< type >

Auxilliary structure to support encoding/decoding slots.

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

- src/**PAlgebra.h**

PAlgebra Class Reference

The structure of (Z/mZ)* /(p) More...

`#include <PAlgebra.h>`

Public Member Functions

	PAlgebra (unsigned long mm, unsigned long pp=2, const vector< long > &_gens=vector< long >(), const vector< long > &_ords=vector< long >())

bool	**operator==** (const **PAlgebra** &other) const
bool	**operator!=** (const **PAlgebra** &other) const
void	**printout** () const
	Prints the structure in a readable form.
unsigned long	**getM** () const
	Returns m.
unsigned long	**getP** () const
	Returns p.
unsigned long	**getPhiM** () const
	Returns phi(m)
unsigned long	**getOrdP** () const
	The order of p in (Z/mZ)^*.
unsigned long	**getNSlots** () const
	The number of plaintext slots = phi(m)/ord(p)
const ZZX &	**getPhimX** () const
	The cyclotomix polynomial Phi_m(X)
void	**set_cM** (double c)
	The ring constant cM.
const double	**get_cM** () const

unsigned long	**numOfGens** () const
	The prime-power factorization of m. More...

unsigned long	**ZmStarGen** (unsigned long i) const
	the i'th generator in (Z/mZ)^* /(p) (if any)

unsigned long	**OrderOf** (unsigned long i) const
	The order of i'th generator (if any)

bool	**SameOrd** (unsigned long i) const
	Is ord(i'th generator) the same as its order in (Z/mZ)^*?

Translation between index, represnetatives, and exponents

unsigned long	**ith_rep** (unsigned long i) const
	Returns the i'th element in T.

long	**indexOfRep** (unsigned long t) const
	Returns the index of t in T.

bool	**isRep** (unsigned long t) const
	Is t in T?

long	**indexInZmstar** (unsigned long t) const
	Returns the index of t in (Z/mZ)*.

bool	**inZmStar** (unsigned long t) const
	Is t in [0,m-1] with (t,m)=1?

long	**coordinate** (long i, long k) const

	Returns ith coordinate of index k along the i'th dimension. See Section 2.4 in the design document.

long	**addCoord** (long i, long k, long offset) const
	adds offset to index k in the i'th dimension

unsigned long	**exponentiate** (const vector< unsigned long > &exps, bool onlySameOrd=false) const
	Returns prod_i gi^{exps[i]} mod m. If onlySameOrd=true, use only generators that have the same order as in (Z/mZ)^*.

const long *	**dLog** (unsigned long t) const
	Inverse of exponentiate.

unsigned long	**gGrpOrd** (bool onlySameOrd=false) const

bool	**nextExpVector** (vector< unsigned long > &exps) const

Detailed Description

The structure of (Z/mZ)* /(p)

A **PAlgebra** object is determined by an integer m and a prime p, where p does not divide m. It holds information descrtibing the structure of (Z/mZ)^*, which is isomorphic to the Galois group over A = Z[X]/Phi_m(X)).

We represent (Z/mZ)^* as (Z/mZ)^* = (p) x (g1,g2,...) x (h1,h2,...) where the group generated by g1,g2,... consists of the elements that have the same order in (Z/mZ)^* as in (Z/mZ)^* /(p,g_1,...,g_{i-1}), and h1,h2,... generate the remaining quotient group (Z/mZ)^* /(p,g1,g2,...).

We let T subset (Z/mZ)^* be a set of representatives for the quotient group (Z/mZ)^* /(p), defined as T={ prod_i gi^{ei} * prod_j hj^{ej} } where the ei's range over 0,1,...,ord(gi)-1 and the ej's range over 0,1,...ord(hj)-1 (these last orders are in (Z/mZ)^* /(p,g1,g2,...)).

Phi_m(X) is factored as Phi_m(X)= prod_{t in T} F_t(X) mod p, where the F_t's are irreducible modulo p. An arbitrary factor is chosen as F_1, then for each t in T we associate with the index t the factor F_t(X) = GCD(F_1(X^t), Phi_m(X)).

Note that fixing a representation of the field R=(Z/pZ)[X]/F_1(X) and letting z be a root of F_1 in R (which is a primitive m-th root of unity in R), we get that F_t is the minimal polynomial of z^{1/t}.

Member Function Documentation

bool PAlgebra::nextExpVector (vector< unsigned long > & exps) const

exps is an array of exponents (the dLog of some t in T), this function incerement exps lexicographic order, reutrn false if it cannot be incremented (because it is at its maximum value)

unsigned long PAlgebra::numOfGens ( ) const inline

The prime-power factorization of m.

The number of generators in (Z/mZ)^* /(p)

unsigned long PAlgebra::qGrpOrd (bool onlySameOrd = `false`) const inline

The order of the quoteint group (Z/mZ)^* /(p) (if flag=false), or the subgroup of elements with the same order as in (Z/mZ)^* (if flag=true)

The documentation for this class was generated from the following files:

- src/**PAlgebra.h**
- src/PAlgebra.cpp

PAlgebraMod Class Reference

The structure of Z[X]/(Phi_m(X), p) More...

`#include <PAlgebra.h>`

Public Member Functions

	PAlgebraMod (const PAlgebra &zMStar, long r)
template<class type >	
const PAlgebraModDerived< type > &	getDerived (type) const
bool	**operator==** (const PAlgebraMod &other) const
bool	**operator!=** (const PAlgebraMod &other) const
PA_tag	getTag () const
	Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
const PAlgebra &	getZMStar () const
	Returns reference to underlying PAlgebra object.
const vector< ZZX > &	getFactorsOverZZ () const
	Returns reference to the factorization of Phi_m(X) mod p^r, but as ZZX's.
long	getR () const
	The value r.
long	getPPowR () const
	The value p^r.
void	restoreContext () const
	Restores the NTL context for p^r.

Detailed Description

The structure of Z[X]/(Phi_m(X), p)

An object of type **PAlgebraMod** stores information about a **PAlgebra** object zMStar, and an integer r. It also provides support for encoding and decoding plaintext slots.

the **PAlgebra** object zMStar defines (Z/mZ)^* /(0), and the **PAlgebraMod** object stores various tables related to the polynomial ring Z/(p^r)[X]. To do this most efficiently, if p == 2 and r == 1, then these polynomials are represented as GF2X's, and otherwise as zz_pX's. Thus, the types of these objects are not determined until run time. As such, we need to use a class heirarchy, as follows.

- **PAlgebraModBase** is a virtual class
- PAlegbraModDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.
- The class **PAlgebraMod** is a simple wrapper around a smart pointer to a **PAlgebraModBase** object: copying a **PAlgebra** object results is a "deep copy" of the underlying object of the derived class. It provides dDirect access to the virtual methods of **PAlgebraModBase**, along with a "downcast" operator to get a reference to the object as a derived type, and also == and != operators.

Member Function Documentation

template<class type >

const PAlgebraModDerived<type>& PAlgebraMod::getDerived (type) const inline

Downcast operator example: const PAlgebraModDerived<PA_GF2>& rep = alMod.getDerived(PA_GF2());

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

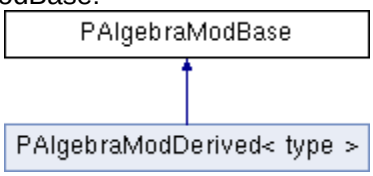
- src/**PAlgebra.h**

PAlgebraModBase Class Reference abstract

Virtual base class for **PAlgebraMod**. More...

`#include <PAlgebra.h>`

Inheritance diagram for PAlgebraModBase:



Public Member Functions

virtual **PAlgebraModBase** *	**clone** () const =0
virtual PA_tag	**getTag** () const =0
	Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
virtual const **PAlgebra** &	**getZMStar** () const =0
	Returns reference to underlying **PAlgebra** object.
virtual const vector< ZZX > &	**getFactorsOverZZ** () const =0
	Returns reference to the factorization of Phi_m(X) mod p^r, but as ZZX's.
virtual long	**getR** () const =0
	The value r.
virtual long	**getPPowR** () const =0
	The value p^r.
virtual void	**restoreContext** () const =0
	Restores the NTL context for p^r.

Detailed Description

Virtual base class for **PAlgebraMod**.

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

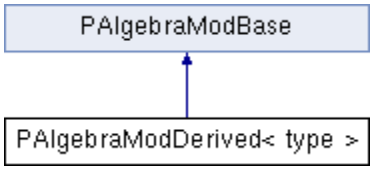
- src/**PAlgebra.h**

PAlgebraModDerived< type > Class Template Reference

A concrete instantiation of the virtual class. More...

`#include <PAlgebra.h>`

Inheritance diagram for PAlgebraModDerived< type >:



Public Member Functions

	PAlgebraModDerived (const **PAlgebra** &zMStar, long r)
	PAlgebraModDerived (const **PAlgebraModDerived** &other)
PAlgebraModDerived &	**operator=** (const **PAlgebraModDerived** &other)
virtual **PAlgebraModBase** *	**clone** () const Returns a pointer to a "clone".
virtual PA_tag	**getTag** () const

	Returns the type tag: PA_GF2_tag or PA_zz_p_tag.
virtual const **PAlgebra** &	**getZMStar** () const
	Returns reference to underlying **PAlgebra** object.
virtual const vector< ZZX > &	**getFactorsOverZZ** () const
	Returns reference to the factorization of Phi_m(X) mod p^r, but as ZZX's.
virtual long	**getR** () const
	The value r.
virtual long	**getPPowR** () const
	The value p^r.
virtual void	**restoreContext** () const
	Restores the NTL context for p^r.
const RXModulus &	**getPhimXMod** () const
	Returns reference to an RXModulus representing Phi_m(X) (mod p^r)
const vec_RX &	**getFactors** () const
	Returns reference to the factors of Phim_m(X) modulo p^r.
const vec_RX &	**getCrtCoeffs** () const

	Returns the CRT coefficients: element i contains (prod_{j!=i} F_j)^{-1} mod F_i, where F_0 F_1 ... is the factorization of Phi_m(X) mod p^r.

const vector< vector< RX > > &	**getMaskTable** () const
	Returns ref to maskTable, which is used to implement rotations (in the **EncryptedArray** module). More...

Embedding in the plaintext slots and decoding back

In all the functions below, G must be irredicible mod p, and the order of G must divide the order of p modulo m (as returned by zMStar.getOrdP()). In addition, when r > 1, G must be the monomial X (RX(1, 1))

void	**CRT_decompose** (vector< RX > &crt, const RX &H) const
	Returns a vector crt[] such that crt[i] = H mod Ft (with t = T[i])

void	**CRT_reconstruct** (RX &H, vector< RX > &crt) const
	Returns H in R[X]/Phi_m(X) s.t. for every i<nSlots and t=T[i], we have H == crt[i] (mod Ft)

void	**mapToSlots** (**MappingData**< type > &mappingData, const RX &G) const
	Compute the maps for all the slots. In the current implementation, we if r > 1, then we must have either deg(G) == 1 or G == factors[0].

void	**embedInAllSlots** (RX &H, const RX &alpha, const **MappingData**< type > &mappingData) const
	Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as

	alpha in R[X]/G(X). More...
void	**embedInSlots** (RX &H, const vector< RX > &alphas, const **MappingData**< type > &mappingData) const
	Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alphas[i] in R[X]/G(X). More...
void	**decodePlaintext** (vector< RX > &alphas, const RX &ptxt, const **MappingData**< type > &mappingData) const
	Return an array such that alphas[i] in R[X]/G(X) represent the same element as rt = (H mod Ft) in R[X]/Ft(X) where t=T[i]. More...
void	**buildLinPolyCoeffs** (vector< RX > &C, const vector< RX > &L, const **MappingData**< type > &mappingData) const
	Returns a coefficient vector C for the linearized polynomial representing M. More...

Detailed Description

template<class type>
class PAlgebraModDerived< type >

A concrete instantiation of the virtual class.

Member Function Documentation

template<class type >

void PAlgebraModDerived< type (vector< RX > & C,

>::buildLinPolyCoeffs

```
                                const vector< RX > &          L,
                                const MappingData< type > &   mappingData
                              )                               const
```

Returns a coefficient vector C for the linearized polynomial representing M.

For h in Z/(p^r)[X] of degree < d,

G is assumed to be defined in mappingData, with d = deg(G). L describes a linear map M by describing its action on the standard power basis: M(x^j mod G) = (L[j] mod G), for j = 0..d-1.

template<class type >

void PAlgebraModDerived< type >::decodePlaintext

```
                              ( vector< RX > &          alphas,
                                const RX &              ptxt,
                                const MappingData< type > &   mappingData
                              )                               const
```

Return an array such that alphas[i] in R[X]/G(X) represent the same element as rt = (H mod Ft) in R[X]/Ft(X) where t=T[i].

The mappingData argument should contain the output of mapToSlots(G).

template<class type >

void PAlgebraModDerived< type >::embedInAllSlots

```
                              ( RX &                    H,
                                const RX &              alpha,
```

const MappingData< type > & mappingData

) const

Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alpha in R[X]/G(X).

Must have deg(alpha)<deg(G). The mappingData argument should contain the output of mapToSlots(G).

template<class type >

void PAlgebraModDerived< type >::embedInSlots (RX & H,

const vector< RX > & alphas,

const MappingData< type > & mappingData

) const

Returns H in R[X]/Phi_m(X) s.t. for every t in T, the element Ht = (H mod Ft) in R[X]/Ft(X) represents the same element as alphas[i] in R[X]/G(X).

Must have deg(alpha[i])<deg(G). The mappingData argument should contain the output of mapToSlots(G).

template<class type>

const vector< vector< RX > >& PAlgebraModDerived< type >::getMaskTable ( ) const inline

Returns ref to maskTable, which is used to implement rotations (in the EncryptedArray module).

maskTable[i][j] is a polynomial representation of a mask that is 1 in all slots whose i'th coordinate is at least j, and 0 elsewhere. We have:

```
maskTable.size() == zMStar.numOfGens()      // # of generators
for i = 0..maskTable.size()-1:
  maskTable[i].size() == zMStar.OrderOf(i) // order of generator i
```

The documentation for this class was generated from the following files:

- src/**PAlgebra.h**
- src/PAlgebra.cpp

PermNetLayer Class Reference

The information needed to apply one layer of a permutation network. More...

`#include <permutations.h>`

Public Member Functions

long	**getGenIdx** () const
long	**getE** () const
const Vec< long > &	**getShifts** () const
bool	**isIdentity** () const

Friends

class	**PermNetwork**
ostream &	**operator<<** (ostream &s, const **PermNetwork** &net)

Detailed Description

The information needed to apply one layer of a permutation network.

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

- src/permutations.h

PermNetwork Class Reference

A full permutation network. More...

`#include <permutations.h>`

Public Member Functions

	PermNetwork (const Permut &pi, const GeneratorTrees &trees)
long	**depth** () const
void	buildNetwork (const Permut &pi, const GeneratorTrees &trees)
void	applyToCtxt (Ctxt &c, const EncryptedArray &ea) const
	Apply network to permute a ciphertext.
void	applyToCube (HyperCube< long > &v) const
	Apply network to array, used mostly for debugging.
void	applyToPtxt (ZZX &p, const EncryptedArray &ea) const
	Apply network to plaintext polynomial, used mostly for debugging.
const PermNetLayer &	**getLayer** (long i) const

Friends

ostream &	**operator<<** (ostream &s, const PermNetwork &net)

Detailed Description

A full permutation network.

Member Function Documentation

void PermNetwork::buildNetwork (const Permut & pi,
const GeneratorTrees & trees
)

Take as input a permutation pi and the trees of all the generators, and prepares the permutation network for this pi

---

The documentation for this class was generated from the following files:

- src/**permutations.h**
- src/PermNetwork.cpp

PlaintextArray Class Reference

A simple wrapper for a pointer to a **PlaintextArrayBase**. This is the interface that higher-level code should use. More...

`#include <EncryptedArray.h>`

Public Member Functions

	PlaintextArray (const **EncryptedArray** &ea)

template<class type >

const **PlaintextArrayDerived** < type > &	**getDerived** (type) const

template<class type >

PlaintextArrayDerived< type > &	**getDerived** (type)
const EncryptedArray &	getEA () const Get the EA object (which is needed for the encoding/decoding routines)
void	rotate (long k) Rotation/shift as a linear array.
void	shift (long k) Non-cyclic shift with zero fill.
void	encode (const vector< long > &array) Encode/decode arrays into plaintext polynomials.
void	**encode** (const vector< ZZX > &array)
void	**decode** (vector< long > &array) const
void	**decode** (vector< ZZX > &array) const
void	encode (long val) Encode with the same value replicated in each slot.
void	**encode** (const ZZX &val)
void	random () Generate a uniformly random element.
bool	equals (const PlaintextArray &other) const

	Equality testing.
bool	**equals** (const vector< long > &other) const
bool	**equals** (const vector< ZZX > &other) const
void	**add** (const **PlaintextArray** &other)
void	**sub** (const **PlaintextArray** &other)
void	**negate** ()
void	**mul** (const **PlaintextArray** &other)
void	**mat_mul** (const **PlaintextMatrixBaseInterface** &mat)
void	**alt_mul** (const **PlaintextMatrixBaseInterface** &mat)
void	**mat_mul** (const **PlaintextBlockMatrixBaseInterface** &mat)
void	**replicate** (long i)
	Replicate coordinate i at all coordinates.
void	**frobeniusAutomorph** (long j)
void	**frobeniusAutomorph** (const Vec< long > &vec)
virtual void	**applyPerm** (const Vec< long > &pi)
void	**print** (ostream &s) const

Detailed Description

A simple wrapper for a pointer to a **PlaintextArrayBase**. This is the interface that higher-level code should use.

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

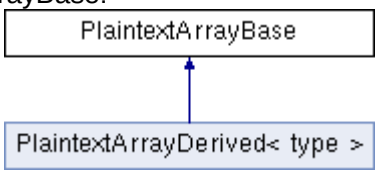
- src/**EncryptedArray.h**

PlaintextArrayBase Class Reference abstract

Virtual class for array of slots, not encrypted. More...

```
#include <EncryptedArray.h>
```

Inheritance diagram for PlaintextArrayBase:



Public Member Functions

virtual **PlaintextArrayBase** *	**clone** () const =0
virtual const **EncryptedArray** &	**getEA** () const =0
	Get the EA object (which is needed for the encoding/decoding routines)
virtual void	**rotate** (long k)=0
	Rotation/shift as a linear array.
virtual void	**shift** (long k)=0
	Non-cyclic shift with zero fill.

virtual void	**encode** (const vector< long > &array)=0 Encode/decode arrays into plaintext polynomials.
virtual void	**encode** (const vector< ZZX > &array)=0
virtual void	**decode** (vector< long > &array) const =0
virtual void	**decode** (vector< ZZX > &array) const =0
virtual void	**encode** (long val)=0 Encode with the same value replicated in each slot.
virtual void	**encode** (const ZZX &val)=0
virtual void	**random** ()=0 Generate a uniformly random element.
virtual bool	**equals** (const **PlaintextArrayBase** &other) const =0 Equality testing.
virtual bool	**equals** (const vector< long > &other) const =0
virtual bool	**equals** (const vector< ZZX > &other) const =0
virtual void	**add** (const **PlaintextArrayBase** &other)=0
virtual void	**sub** (const **PlaintextArrayBase** &other)=0
virtual void	**mul** (const **PlaintextArrayBase** &other)=0

virtual void	**negate** ()=0
virtual void	**mat_mul** (const PlaintextMatrixBaseInterface &mat)=0
virtual void	**alt_mul** (const PlaintextMatrixBaseInterface &mat)=0
virtual void	**mat_mul** (const PlaintextBlockMatrixBaseInterface &mat)=0
virtual void	**replicate** (long i)=0 Replicate coordinate i at all coordinates.
virtual void	**frobeniusAutomorph** (long j)=0 apply x -> x^{p^j} at all coordinates
virtual void	**frobeniusAutomorph** (const Vec< long > &vec)=0 apply x-> x^{p^{vec[i]}} to slot i
virtual void	**applyPerm** (const Vec< long > &pi)=0 apply a permutation: new[i] = old[pi[i]]
virtual void	**print** (ostream &s) const =0

Detailed Description

Virtual class for array of slots, not encrypted.

An object pa of type PlaintextArray stores information about an EncryptedArray object ea. The object pa stores a vector of plaintext slots, where each slot is an element of the polynomial ring (Z/(p^r)[X])/(G), where p, r, and G are as defined in ea. Support for arithemetic on PlaintextArray objects is provided.

Mirroring **PAlgebraMod** and **EncryptedArray**, we have the following class heirarchy:

PlaintextArrayBase is a virtual class

PlaintextArrayDerived<type> is a derived template class, where type is either PA_GF2 or PA_zz_p.

The class **PlaintextArray** is a simple wrapper around a smart pointer to a **PlaintextArray** object: copying a **PlaintextArray** object results is a "deep copy" of the underlying object of the derived class.

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

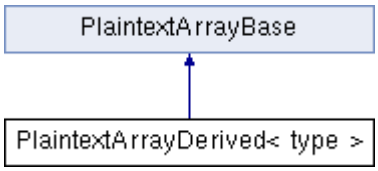
- src/**EncryptedArray.h**

PlaintextArrayDerived< type > Class Template Reference

Derived concrete implementation of **PlaintextArrayBase**. More...

`#include <EncryptedArray.h>`

Inheritance diagram for PlaintextArrayDerived< type >:



Public Member Functions

virtual **PlaintextArrayBase** *	**clone** () const
virtual const **EncryptedArray** &	**getEA** () const Get the EA object (which is needed for the encoding/decoding routines)
	PlaintextArrayDerived (const **EncryptedArray** &_ea)

	PlaintextArrayDerived (const PlaintextArrayDerived &other)
PlaintextArrayDerived &	**operator=** (const PlaintextArrayDerived &other)
virtual void	rotate (long k) Rotation/shift as a linear array.
virtual void	shift (long k) Non-cyclic shift with zero fill.
virtual void	encode (const vector< long > &array) Encode/decode arrays into plaintext polynomials.
virtual void	**encode** (const vector< ZZX > &array)
virtual void	**decode** (vector< long > &array) const
virtual void	**decode** (vector< ZZX > &array) const
virtual void	encode (long val) Encode with the same value replicated in each slot.
virtual void	**encode** (const ZZX &val)
virtual void	random ()

	Generate a uniformly random element.
virtual bool	**equals** (const **PlaintextArrayBase** &other) const
	Equality testing.
virtual bool	**equals** (const vector< long > &other) const
virtual bool	**equals** (const vector< ZZX > &other) const
virtual void	**add** (const **PlaintextArrayBase** &other)
virtual void	**sub** (const **PlaintextArrayBase** &other)
virtual void	**negate** ()
virtual void	**mul** (const **PlaintextArrayBase** &other)
virtual void	**mat_mul** (const **PlaintextMatrixBaseInterface** &mat)
virtual void	**alt_mul** (const **PlaintextMatrixBaseInterface** &mat)
virtual void	**mat_mul** (const **PlaintextBlockMatrixBaseInterface** &mat)
virtual void	**replicate** (long i)
	Replicate coordinate i at all coordinates.

virtual void	**frobeniusAutomorph** (long j)
	apply x -> x^{p^j} at all coordinates
virtual void	**frobeniusAutomorph** (const Vec< long > &vec)
	apply x-> x^{p^{vec[i]}} to slot i
virtual void	**applyPerm** (const Vec< long > &pi)
	apply a permutation: new[i] = old[pi[i]]
virtual void	**print** (ostream &s) const
const vector< RX > &	**getData** () const
void	**setData** (const vector< RX > &_data)

Static Public Member Functions

static void	**rec_mul** (long dim, const **EncryptedArray** &ea, vector< RX > &res, const vector< RX > &pdata, const vector< long > &idx, const **PlaintextMatrixInterface**< type > &mat)

Friends

template<class tt >

ostream &	**operator<<** (ostream &os, const **PlaintextArrayDerived**< tt > &pt)

Detailed Description

template<class type>
class PlaintextArrayDerived< type >

Derived concrete implementation of **PlaintextArrayBase**.

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

- src/**EncryptedArray.h**

PlaintextBlockMatrixBaseInterface Class Reference abstract

An abstract interface for linear transformations. More...

`#include <EncryptedArray.h>`

Inheritance diagram for PlaintextBlockMatrixBaseInterface:

Public Member Functions

virtual const **EncryptedArray** &	**getEA** () const =0

Detailed Description

An abstract interface for linear transformations.

A block matrix implements linear transformation over the base field/ring (e.g., Z_2, Z_3, Z_{2^8}, etc.) Any class implementing this interface should be linked to a specific **EncryptedArray** object, a reference to which is returned by the getEA() method – this method will generally be invoked by an **EncryptedArray** object to verify consistent use.

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

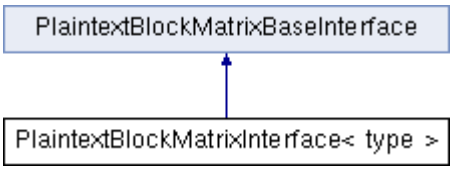
- src/EncryptedArray.h

PlaintextBlockMatrixInterface< type > Class Template Reference abstract

A somewhat less abstract interface for linear transformations. More...

`#include <EncryptedArray.h>`

Inheritance diagram for PlaintextBlockMatrixInterface< type >:



Public Member Functions

virtual bool	**get** (mat_R &out, long i, long j) const =0

Public Member Functions inherited from PlaintextBlockMatrixBaseInterface

Detailed Description

template<class type>
class PlaintextBlockMatrixInterface< type >

A somewhat less abstract interface for linear transformations.

A block matrix implements linear transformation over the base field/ring (e.g., Z_2, Z_3, Z_{2^8}, etc.) The method get(out, i, j) copies the element at row i column j of a matrix into the variable out. The type of out is mat_R (so either mar_GF2 or mat_zz_p. A return value of true means that the entry is zero, and out is not touched.

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

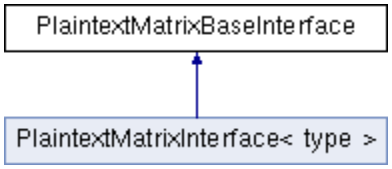
- src/**EncryptedArray.h**

PlaintextMatrixBaseInterface Class Reference abstract

An abstract interface for linear transformations. More...

`#include <EncryptedArray.h>`

Inheritance diagram for PlaintextMatrixBaseInterface:



Public Member Functions

virtual const **EncryptedArray** &	**getEA** () const =0

Detailed Description

An abstract interface for linear transformations.

A matrix implements linear transformation over an extension field/ring (e.g., GF(2^d) or Z_{2^8}[X]/G(X) for irreducible G). Any class implementing this interface should be linked to a specific**EncryptedArray** object, a reference to which is returned by the getEA() method – this method will generally be invoked by an **EncryptedArray** object to verify consistent use.

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

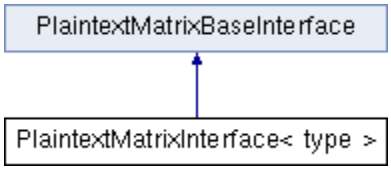
- src/**EncryptedArray.h**

PlaintextMatrixInterface< type > Class Template Reference abstract

A somewhat less abstract interface for linear transformations. More...

`#include <EncryptedArray.h>`

Inheritance diagram for PlaintextMatrixInterface< type >:



Public Member Functions

virtual bool	**get** (RX &out, long i, long j) const =0

Public Member Functions inherited from PlaintextMatrixBaseInterface

Detailed Description

template<class type>
class PlaintextMatrixInterface< type >

A somewhat less abstract interface for linear transformations.

A matrix implements linear transformation over an extension field/ring (e.g., GF(2^d) or Z_{2^8}[X]/G(X) for irreducible G). The method get(out, i, j) copies the element at row i column j of a matrix into the variable out. The type of out is RX, which is GF2X if type is PA_GF2, and zz_pX if type is PA_zz_p. A return value of true means that the entry is zero, and out is not touched.

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

- src/EncryptedArray.h

PowerfulConversion Class Reference

Conversion between powerful representation in R_m/(q) and zz_pX. More...

`#include <powerful.h>`

Public Member Functions

	PowerfulConversion (const PowerfulTranslationIndexes &ind)
void	**initPConv** (const PowerfulTranslationIndexes &ind)
void	**restoreModulus** () const
const CubeSignature &	**getLongSig** () const
const CubeSignature &	**getShortSig** () const
long	powerfulToPoly (zz_pX &poly, const HyperCube< zz_p > &powerful) const
long	**polyToPowerful** (HyperCube< zz_p > &powerful, const zz_pX &poly) const

Detailed Description

Conversion between powerful representation in R_m/(q) and zz_pX.

Usage pattern is as follows:

// compute tables for index translation PowerfulTranslationIndexes ind(mvec); // mvec is some factorization of m

// ... set the current zz_p::modulus to something before initializing PowerfulConversion pConv(ind);

// Alternatively use // **PowerfulConversion** pConv(); pConv.initPConv(ind); // Only the latter call needs zz_p::modulus to be defined

// A powerful basis is defined wrt same modulus and cube signature HyperCube<zz_p> powerful(pConv.getShortSig());

// ... some code here to initialize powerful // code can also do other stuff, perhaps changing zz_p::modulus

pConv.restoreModulus(); // restore zz_p::modulus zz_pX poly; // defined relative to the same modulus pConv.powerfulToPoly(poly, powerful);

// ... some more code here, perhaps changing zz_p::modulus again

pConv.restoreModulus(); // restore zz_p::modulus pConv.polyToPowerful(powerful, poly);

Member Function Documentation

long PowerfulConversion::powerfulToPoly (zz_pX & poly,
const HyperCube< zz_p > & powerful
) const

The conversion routines return the value of the modulus q. It is assumed that the modulus is already set before calling them

---

The documentation for this class was generated from the following files:

- src/**powerful.h**
- src/powerful.cpp

PowerfulDCRT Class Reference

Conversion between powerful representation, **DoubleCRT**, and ZZX. More...

`#include <powerful.h>`

Public Member Functions

	PowerfulDCRT (const FHEcontext &_context, const Vec< long > &mvec)
const PowerfulTranslationIndexes &	**getIndexTranslation** () const
const PowerfulConversion &	**getPConv** (long i) const
void	**dcrtToPowerful** (Vec< ZZ > &powerful, const DoubleCRT &dcrt) const
void	**powerfulToDCRT** (DoubleCRT &dcrt, const Vec< ZZ > &powerful) const
void	**ZZXtoPowerful** (Vec< ZZ > &powerful, const ZZX &poly, IndexSet s=IndexSet::emptySet()) const
void	**powerfulToZZX** (ZZX &poly, const Vec< ZZ > &powerful, IndexSet s=IndexSet::emptySet()) const

Detailed Description

Conversion between powerful representation, DoubleCRT, and ZZX.

The documentation for this class was generated from the following files:

- src/powerful.h
- src/powerful.cpp

PowerfulTranslationIndexes Class Reference

Holds index tables for translation between powerful and zz_pX. More...

`#include <powerful.h>`

Public Member Functions

	PowerfulTranslationIndexes (const Vec< long > &mv)

Public Attributes

long	**M**
long	**Phim**
Vec< long >	**Mvec**
Vec< long >	**Phivec**
Vec< long >	**Divvec**
Vec< long >	**Invvec**
CubeSignature	**longSig**
CubeSignature	**shortSig**
Vec< long >	**polyToCubeMap**
Vec< long >	**cubeToPolyMap**

Vec< long >	**shortToLongMap**
Vec< ZZX >	**cycVec**
ZZX	**phimX**

Detailed Description

Holds index tables for translation between powerful and zz_pX.

The documentation for this class was generated from the following files:

- src/**powerful.h**
- src/powerful.cpp

RandomState Class Reference

Facility for "restoring" the NTL PRG state

`#include <NumbTh.h>`

Public Member Functions

void	**restore** ()
	Restore the PRG state of NTL.

Detailed Description

Facility for "restoring" the NTL PRG state.

NTL's random number generation faciliity is pretty limited, and does not provide a way to save/restore the state of a pseudo-random stream. This class gives us that ability: Constructing a**RandomState** object uses the PRG to generate 512 bits and stores them. Upon destruction (or an explicit call to **restore()**), these bits are used to re-set the seed of the PRG. A typical usage of thie class is as follows:

```
{
RandomState r; // save the random state
SetSeed(something); // set the PRG seed to something
... // more code that uses the new PRG seed
} // The destructor is called implicitly, PRG state is restored
```

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

- src/**NumbTh.h**

RecryptData Class Reference

A structure to hold recryption-related data inside the **FHEcontext**

`#include <recryption.h>`

Public Member Functions

void	**init** (const **FHEcontext** &context, const Vec< long > &mvec_, long t=0, bool consFlag=false)
	Initialize the recryption data in the context.
bool	**operator==** (const **RecryptData** &other) const
bool	**operator!=** (const **RecryptData** &other) const

Public Attributes

Vec< long >	**mvec**
	Some data members that are only used for I/O.
long	**hwt**
	partition of m into co-prime factors
bool	**conservative**
	Hamming weight of recryption secret-key.
long	**e**
	flag for choosing more conservatice parameters More...
long	**ePrime**
long	**skHwt**
	Hamming weight of recryption secret key.
double	**alpha**
	an optimization parameter
PAlgebraMod *	**alMod**
	for plaintext space p^{e-e'+r}
EncryptedArray *	**ea**
	for plaintext space p^{e-e'+r}
EvalMap *	**firstMap**
	linear maps

EvalMap *	**secondMap**

PowerfulDCRT *	p2dConv
	conversion between ZZX and Powerful

vector< ZZX >	unpackSlotEncoding
	linPolys for uppacking the slots

Static Public Attributes

static const long	defSkHwt =56
	default Hamming weight of recryption key

Detailed Description

A structure to hold recryption-related data inside the FHEcontext.

Member Data Documentation

long RecryptData::e

flag for choosing more conservatice parameters

skey encrypted wrt space p^{e-e'+r}

The documentation for this class was generated from the following files:

- src/recryption.h
- src/recryption.cpp

ReplicateHandler Class Reference abstract

A virtual class to handle call-backs to get the output of replicate. More...

`#include <replicate.h>`

Public Member Functions

virtual void	**handle** (const Ctxt &ctxt)=0

Detailed Description

A virtual class to handle call-backs to get the output of replicate.

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

- src/replicate.h

---

shallow_clone< X > Class Template Reference

Shallow copy: initialize with copy constructor.

`#include <cloned_ptr.h>`

Static Public Member Functions

static X *	**apply** (const X *x)

Detailed Description

template<class X>
class shallow_clone< X >

Shallow copy: initialize with copy constructor.

Template Parameters

X The class to which this points

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

- src/**cloned_ptr.h**

SKHandle Class Reference

A handle, describing the secret-key element that "matches" a part, of the form s^r(X^t). More...

`#include <Ctxt.h>`

Public Member Functions

	SKHandle (long newPowerOfS=0, long newPowerOfX=1, long newSecretKeyID=0)
void	**setBase** (long newSecretKeyID=-1)
	Set powerOfS=powerOfX=1.
bool	**isBase** (long ofKeyID=0) const
	Is powerOfS==powerOfX==1?
void	**setOne** (long newSecretKeyID=-1)
	Set powerOfS=0, powerOfX=1.

bool	**isOne** () const
	Is powerOfS==0?
bool	**operator==** (const **SKHandle** &other) const
bool	**operator!=** (const **SKHandle** &other) const
long	**getPowerOfS** () const
long	**getPowerOfX** () const
long	**getSecretKeyID** () const
bool	**mul** (const **SKHandle** &a, const **SKHandle** &b)
	Computes the "product" of two handles. More...

Friends

class	**Ctxt**
istream &	**operator>>** (istream &s, **SKHandle** &handle)

Detailed Description

A handle, describing the secret-key element that "matches" a part, of the form s^r(X^t).

Member Function Documentation

bool SKHandle::mul (const SKHandle & a,
const SKHandle & b inline

)

Computes the "product" of two handles.

The key-ID's and powers of X must match, else an error state arises, which is represented using a key-ID of -1 and returning false. Also, note that inputs may alias outputs.

To detremine if the resulting handle canbe re-liearized using some key-switchingmatrices from the public key, use the method pubKey.haveKeySWmatrix(handle,handle.secretKeyID), from the class **FHEPubKey** in **FHE.h**

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

- src/**Ctxt.h**

SubDimension Class Reference

A node in a tree relative to some generator

`#include <permutations.h>`

Public Member Functions

	SubDimension (long sz=0, bool gd=false, long ee=0, const Vec< long > &bns1=dummyBenes, const Vec< long > &bns2=dummyBenes)
long	**totalLength** () const

Public Attributes

long	**Size**

bool	**Good**
long	**E**
Vec< long >	**frstBenes**
Vec< long >	**scndBenes**

Friends

ostream &	**operator<<** (ostream &s, const **SubDimension** &tree)

Detailed Description

A node in a tree relative to some generator.

The documentation for this class was generated from the following files:

- src/**permutations.h**
- src/permutations.cpp

TreeNode< T > Class Template Reference

A node in a full binary tree.

`#include <permutations.h>`

Public Member Functions

	TreeNode (const T &d)

T &	**getData** ()
const T &	**getData** () const
long	**getParent** () const
long	**getLeftChild** () const
long	**getRightChild** () const
long	**getPrev** () const
long	**getNext** () const

Friends

class	**FullBinaryTree< T >**

Detailed Description

template<class T>
class TreeNode< T >

A node in a full binary tree.

These nodes are in a vector, so we use indexes rather than pointers

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

- src/**permutations.h**

---

zz_pXModulus1 Class Reference

Auxiliary classes to facillitiate faster reduction mod Phi_m(X) when the input has degree less than m. More...

`#include <NumbTh.h>`

Public Member Functions

	zz_pXModulus1 (long _m, const zz_pX &_f)
const zz_pXModulus &	**upcast** () const

Public Attributes

long	**M**
zz_pX	**F**
long	**N**
bool	**specialLogic**
long	**K**
long	**k1**

fftRep	**R0**
fftRep	**R1**
zz_pXModulus	**Fm**

Detailed Description

Auxiliary classes to facillitiate faster reduction mod Phi_m(X) when the input has degree less than m.

The documentation for this class was generated from the following files:

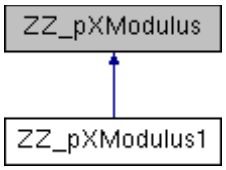
- src/**NumbTh.h**
- src/NumbTh.cpp

ZZ_pXModulus1 Class Reference

placeholder for pXModulus ...no optimizations More...

`#include <NumbTh.h>`

Inheritance diagram for ZZ_pXModulus1:



Public Member Functions

	ZZ_pXModulus1 (long _m, const ZZ_pX &_f)
const ZZ_pXModulus &	**upcast** () const

Detailed Description

placeholder for pXModulus ...no optimizations

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

- src/**NumbTh.h**
