Multidimensional Tensor Library

primary use in perturbation methods for Stochastic Dynamic General Equilibrium (SDGE) models

1. Library overview.

The design of the library was driven by the needs of perturbation methods for solving Stochastic Dynamic General Equilibrium models. The aim of the library is not to provide an exhaustive interface to multidimensional linear algebra. The tensor library's main purposes include:

- Define types for tensors, for a multidimensional index of a tensor, and types for folded and unfolded tensors. The tensors defined here have only one multidimensional index and one reserved one-dimensional index. The tensors should allow modelling of higher order derivatives with respect to a few vectors with different sizes (for example $\left[g_{y^2u^3}\right]$). The tensors should allow folded and unfolded storage modes and conversion between them. A folded tensor stores symmetric elements only once, while an unfolded stores data as a whole multidimensional cube.
- Define both sparse and dense tensors. We need only one particular type of sparse tensor. This in contrast to dense tensors, where we need much wider family of types.
- Implement the Faa Di Bruno multidimensional formula. So, the main purpose of the library is to implement the following step of Faa Di Bruno:

$$\left[B_{s^k}\right]_{\alpha_1\ldots\alpha_k} = \left[h_{y^l}\right]_{\gamma_1\ldots\gamma_l}\left(\sum_{c\in M_{l,k}}\prod_{m=1}^{l}\left[g_{c_m}\right]^{\gamma_m}_{c_m(\alpha)}\right)$$

  where s can be a compound vector of variables, $M_{l,k}$ is a set of all equivalences of k element set having l classes, c_m is m-th class of equivalence c, and $c_m(\alpha)$ is a tuple of picked indices from α by class c_m.

  Note that the sparse tensors play a role of h in the Faa Di Bruno, not of B nor g.

The following table is a road-map to various abstractions in the library.

Class defined in	Purpose
⟨ **Tensor** class declaration 218 ⟩ `tensor.hweb`	Virtual base class for all dense tensors, defines *index* as the multidimensonal iterator
⟨ **FTensor** class declaration 220 ⟩ `tensor.hweb`	Virtual base class for all folded tensors
⟨ **UTensor** class declaration 219 ⟩ `tensor.hweb`	Virtual base class for all unfolded tensors
⟨ **FFSTensor** class declaration 235 ⟩ `fs_tensor.hweb`	Class representing folded full symmetry dense tensor, for instance $\left[g_{y^3}\right]$
⟨ **FGSTensor** class declaration 261 ⟩ `gs_tensor.hweb`	Class representing folded general symmetry dense tensor, for instance $\left[g_{y^2u^3}\right]$
⟨ **UFSTensor** class declaration 237 ⟩ `fs_tensor.hweb`	Class representing unfolded full symmetry dense tensor, for instance $\left[g_{y^3}\right]$
⟨ **UGSTensor** class declaration 263 ⟩ `gs_tensor.hweb`	Class representing unfolded general symmetry dense tensor, for instance $\left[g_{y^2u^3}\right]$
⟨ **URTensor** class declaration 294 ⟩ ⟨ **FRTensor** class declaration 296 ⟩ `rfs_tensor.hweb`	Class representing unfolded/folded full symmetry, row-oriented, dense tensor. Row-oriented tensors are used in the Faa Di Bruno above as some part (few or one column) of a product of g's. Their fold/unfold conversions are special in such a way, that they must yield equivalent results if multiplied with folded/unfolded column-oriented counterparts.
⟨ **URSingleTensor** class declaration 298 ⟩ ⟨ **FRSingleTensor** class declaration 299 ⟩ `rfs_tensor.hweb`	Class representing unfolded/folded full symmetry, row-oriented, single column, dense tensor. Besides use in the Faa Di Bruno, the single column row oriented tensor models also higher moments of normal distribution.

⟨ **UPSTensor** class declaration 318 ⟩ `ps_tensor.hweb`	Class representing unfolded, column-oriented tensor whose symmetry is not that of the $\left[B_{y^2u^3}\right]$ but rather of something as $[B_{yuuyu}]$. This tensor evolves during the product operation for unfolded tensors and its basic operation is to add itself to a tensor with nicer symmetry, here $\left[B_{y^2u^3}\right]$.
⟨ **FPSTensor** class declaration 321 ⟩ `ps_tensor.hweb`	Class representing partially folded, column-oriented tensor whose symmetry is not that of the $\left[B_{y^3u^4}\right]$ but rather something as $\left[B_{yu\vert y^3u\vert u^4}\right]$, where the portions of symmetries represent folded dimensions which are combined in unfolded manner. This tensor evolves during the Faa Di Bruno for folded tensors and its basic operation is to add itself to a tensor with nicer symmetry, here folded $\left[B_{y^3u^4}\right]$.
⟨ **USubTensor** class declaration 475 ⟩ `pyramid_prod.hweb`	Class representing unfolded full symmetry, row-oriented tensor which contains a few columns of huge product $\prod_{m=1}^{l}\left[g_{c_m}\right]^{\gamma_m}_{c_m(\alpha)}$. This is needed during the Faa Di Bruno for folded matrices.
⟨ **IrregTensor** class declaration 483 ⟩ `pyramid2_prod.hweb`	Class representing a product of columns of derivatives $\left[z_{y^ku^l}\right]$, where $z = [y^T, v^T, w^T]^T$. Since the first part of z is y, the derivatives contain many zeros, which are not stored, hence the tensor's irregularity. The tensor is used when calculating one step of Faa Di Bruno formula, i.e. $[f_{z^l}]\sum\prod_{m=1}^{l}\left[z_{c_m}\right]^{\gamma_m}_{c_m(\alpha)}$.
⟨ **FSSparseTensor** class declaration 348 ⟩ `sparse_tensor.hweb`	Class representing full symmetry, column-oriented, sparse tensor. It is able to store elements keyed by the multidimensional index, and multiply itself with one column of row-oriented tensor.
⟨ **FGSContainer** class declaration 388 ⟩ `t_container.hweb`	Container of **FGSTensor**s. It implements the Faa Di Bruno with unfolded or folded tensor h yielding folded B. The methods are **FGSContainer** :: *multAndAdd*.
⟨ **UGSContainer** class declaration 387 ⟩ `t_container.hweb`	Container of **FGSTensor**s. It implements the Faa Di Bruno with unfolded tensor h yielding unfolded B. The method is **UGSContainer** :: *multAndAdd*.
⟨ **StackContainerInterface** class declaration 399 ⟩ `stack_container.hweb`	Virtual pure interface describing all logic of stacked containers for which we will do the Faa Di Bruno operation.
⟨ **UnfoldedStackContainer** class declaration 408 ⟩ `stack_container.hweb`	Implements the Faa Di Bruno operation for stack of containers of unfolded tensors.
⟨ **FoldedStackContainer** class declaration 407 ⟩ `stack_container.hweb`	Implements the Faa Di Bruno for stack of containers of folded tensors.
⟨ **ZContainer** class declaration 409 ⟩ `stack_container.hweb`	The class implements the interface **StackContainerInterface** according to z appearing in context of SDGE models. By a simple inheritance, we obtain ⟨ **UnfoldedZContainer** class declaration 412 ⟩ and also ⟨ **FoldedZContainer** class declaration 411 ⟩.

⟨ **GContainer** class declaration 413 ⟩ `stack_container.hweb`	The class implements the interface **StackContainerInterface** according to G appearing in context of SDGE models. By a simple inheritance, we obtain ⟨ **UnfoldedGContainer** class declaration 416 ⟩ and also ⟨ **FoldedGContainer** class declaration 415 ⟩.
⟨ **Equivalence** class declaration 158 ⟩ `equivalence.hweb`	The class represents an equivalence on n-element set. Useful in the Faa Di Bruno.
⟨ **EquivalenceSet** class declaration 159 ⟩ `equivalence.hweb`	The class representing all equivalences on n-element set. Useful in the Faa Di Bruno.
⟨ **Symmetry** class declaration 139 ⟩ `symmetry.hweb`	The class defines a symmetry of general symmetry tensor. This is it defines a basic shape of the tensor. For $[B_{y^2u^3}]$, the symmetry is y^2u^3.
⟨ **Permutation** class declaration 197 ⟩ `permutation.hweb`	The class represents a permutation of n indices. Useful in the Faa Di Bruno.
⟨ **IntSequence** class declaration 50 ⟩ `int_sequence.hweb`	The class represents a sequence of integers. Useful everywhere.
⟨ **TwoDMatrix** class declaration 82 ⟩ ⟨ **ConstTwoDMatrix** class declaration 81 ⟩ `twod_matrix.hweb`	The class provides an interface to a code handling two-dimensional matrices. The code resides in Sylvester module, in directory `sylv/cc`. The object files from that directory need to be linked: `GeneralMatrix.o`, `Vector.o` and `SylvException.o`. There is no similar interface to **Vector** and **ConstVector** classes from the Sylvester module and they are used directly.
⟨ **KronProdAll** class declaration 103 ⟩ `kron_prod.hweb`	The class represents a Kronecker product of a sequence of arbitrary matrices and is able to multiply a matrix from the right without storing the Kronecker product in memory.
⟨ **KronProdAllOptim** class declaration 104 ⟩ `kron_prod.hweb`	The same as **KronProdAll** but it optimizes the order of matrices in the product to minimize the used memory during the Faa Di Bruno operation. Note that it is close to optimal flops.
⟨ **FTensorPolynomial** class declaration 508 ⟩ ⟨ **UTensorPolynomial** class declaration 507 ⟩ `t_polynomial.hweb`	Abstractions representing a polynomial whose coefficients are folded/unfolded tensors and variable is a column vector. The classes provide methods for traditional and horner-like polynomial evaluation. This is useful in simulation code.
⟨ **FNormalMoments** class declaration 524 ⟩ ⟨ **UNormalMoments** class declaration 523 ⟩ `normal_moments.hweb`	These are containers for folded/unfolded single column tensors for higher moments of normal distribution. The code contains an algorithm for generating the moments for arbitrary covariance matrix.
⟨ **TLStatic** class declaration 535 ⟩ `tl_static.hweb`	The class encapsulates all static information needed for the library. It includes a Pascal triangle (for quick computation of binomial coefficients), and precalculated equivalence sets.
⟨ **TLException** class definition 47 ⟩ `tl_exception.hweb`	Simple class thrown as an exception.

2. The tensor library is multi-threaded. This means, if appropriate compilation options were set, some codes are launched concurrently. This boosts the performance on SMP machines or single processors with hyper-threading support. The basic property of the thread implementation in the library is that we do not allow running more concurrent threads than the preset limit. This prevents threads from competing for memory in such a way that the OS constantly switches among threads with frequent I/O for swaps. This may occur since one thread might need much own memory. The threading support allows for detached threads, the synchronization points during the Faa Di Bruno operation are relatively short, so the resulting load is close to the preset maximum number parallel threads.

3. A few words to the library's test suite. The suite resides in directory `tl/testing`. There is a file `tests.cpp` which contains all tests and `main()` function. Also there are files `factory.h` and `factory.cpp` implementing random generation of various objects. The important property of these random objects is that they are the same for all object's invocations. This is very important in testing and debugging. Further, one can find files `monoms.h` and `monoms.cpp`. See below for their explanation.

There are a few types of tests:

1) We test for tensor indices. We go through various tensors with various symmetries, convert indices from folded to unfolded and vice-versa. We test whether their coordinates are as expected.
2) We test the Faa Di Bruno by comparison of the results of **FGSContainer** :: *multAndAdd* against the results of **UGSContainer** :: *multAndAdd*. The two implementations are pretty different, so this is a good test.
3) We use a code in `monoms.h` and `monoms.cpp` to generate a random vector function $f(x(y,u))$ along with derivatives of $[f_x]$, $\left[x_{y^k u^l}\right]$, and $\left[f_{y^k u^l}\right]$. Then we calculate the resulting derivatives $\left[f_{y^k u^l}\right]$ using *multAndAdd* method of **UGSContainer** or **FGSContainer** and compare the derivatives provided by `monoms`. The functions generated in `monoms` are monomials with integer exponents, so the implementation of `monoms` is quite easy.
4) We do a similar thing for sparse tensors. In this case the `monoms` generate a function $f(y,v(y,u),w(y,u))$, provide all the derivatives and the result $\left[f_{y^k u^l}\right]$. Then we calculate the derivatives with *multAndAdd* of **ZContainer** and compare.
5) We test the polynomial evaluation by evaluating a folded and unfolded polynomial in traditional and horner-like fashion. This gives four methods in total. The four results are compared.

4. Utilities.

5. Simple threads. Start of `sthreads.h` file.

This file defines types making a simple interface to multi-threading. It follows the classical C++ idioms for traits. We have three sorts of traits. The first is a **thread_traits**, which make interface to thread functions (run, exit, create and join), the second is **mutex_traits**, which make interface to mutexes (create, lock, unlock), and third is **cond_traits**, which make interface to conditions (create, wait, broadcast, and destroy). At present, there are two implementations. The first are POSIX threads, mutexes, and conditions, the second is serial (no parallelization).

The file provides the following interfaces templated by the types implementing the threading (like types **pthread_t**, and **pthread_mutex_t** for POSIX thread and mutex):

- **thread** is a pure virtual class, which must be inherited and a method **operator**( )( ) be implemented as the running code of the thread. This code is run as a new thread by calling *run* method.
- **thread_group** allows insertion of **thread**s and running all of them simultaneously joining them. The number of maximum parallel threads can be controlled. See below.
- **synchro** object locks a piece of code to be executed only serially for a given data and specified entry-point. It locks the code until it is destructed. So, the typical use is to create the **synchro** object on the stack of a function which is to be synchronized. The synchronization can be subjected to specific data (then a pointer can be passed to **synchro**'s constructor), and can be subjected to specific entry-point (then **const char** ∗ is passed to the constructor).
- **detach_thread** inherits from **thread** and models a detached thread in contrast to **thread** which models the joinable thread.
- **detach_thread_group** groups the detached threads and runs them. They are not joined, they are synchronized by means of a counter counting running threads. A change of the counter is checked by waiting on an associated condition.

What implementation is selected is governed (at present) by `HAVE_PTHREAD`. If it is defined, then POSIX threads are linked. If it is not defined, then serial implementation is taken. In accordance with this, the header file defines macros `THREAD`, `THREAD_GROUP`, and `SYNCHRO` as the picked specialization of **thread** (or **detach_thread**), **thread_group** (or **detach_thread_group**), and **synchro**.

The type of implementation is controlled by **thread_impl** integer template parameter, this can be *posix* or *empty*.

The number of maximum parallel threads is controlled via a static member of **thread_group** and **detach_thread_group** classes.

```
#ifndef STHREAD_H
#define STHREAD_H
#ifdef HAVE_PTHREAD
#include <pthread.h>
#else
    /* Give valid types for POSIX thread types, otherwise the templates fail in empty mode. Don't use
       typedefs because on some systems pthread_t and friends are typedefs even without the include. */
#define pthread_t void *
#define pthread_mutex_t void *
#define pthread_cond_t void *
#endif
#include <cstdio>
#include <list>
#include <map>
  namespace sthread {
    using namespace std;
    class Empty { };
    ⟨ classical IF template 6 ⟩;
    enum { posix, empty };
```

```
    template⟨int⟩ class thread_traits;
    template⟨int⟩ class detach_thread;
    ⟨ thread template class declaration 7 ⟩;
    ⟨ thread_group template class declaration 8 ⟩;
    ⟨ thread_traits template class declaration 12 ⟩;
    ⟨ mutex_traits template class declaration 13 ⟩;
    ⟨ mutex_map template class declaration 14 ⟩;
    ⟨ synchro template class declaration 17 ⟩;
    ⟨ cond_traits template class declaration 20 ⟩;
    ⟨ condition_counter template class declaration 21 ⟩;
    ⟨ detach_thread template class declaration 27 ⟩;
    ⟨ detach_thread_group template class declaration 28 ⟩;
#ifdef HAVE_PTHREAD
    ⟨ POSIX thread specializations 32 ⟩;
#else
    ⟨ No threading specializations 33 ⟩;
#endif
  };
#endif
```

6. Here is the classical IF template.

```
⟨ classical IF template 6 ⟩ ≡
  template⟨bool condition, class Then, class Else⟩ struct IF {
    typedef Then RET;
  };
  template⟨class Then, class Else⟩ struct IF⟨false, Then, Else⟩ {
    typedef Else RET;
  };
```

This code is used in section 5.

7. The class of **thread** is clear. The user implements **operator**( )( ), the method *run* runs the user's code as joinable thread, *exit* kills the execution.

```
⟨ thread template class declaration 7 ⟩ ≡
  template⟨int thread_impl⟩ class thread {
    typedef thread_traits⟨thread_impl⟩ _Ttraits;
    typedef typename _Ttraits::_Tthread _Tthread;
    _Tthread th;
  public:
    virtual ~thread()
    { }
    _Tthread &getThreadIden()
    { return th; }
    const _Tthread &getThreadIden() const
    { return th; }
    virtual void operator()() = 0;
    void run()
    { _Ttraits::run(this); }
    void detach_run()
    { _Ttraits::detach_run(this); }
    void exit()
    { _Ttraits::exit(); }
  };
```

This code is used in section 5.

8. The **thread_group** is also clear. We allow a user to insert the **thread**s, and then launch *run*, which will run all the threads not allowing more than *max_parallel_threads* joining them at the end. This static member can be set from outside.

```
⟨ thread_group template class declaration 8 ⟩ ≡
  template⟨int thread_impl⟩ class thread_group {
    typedef thread_traits⟨thread_impl⟩ _Ttraits;
    typedef thread⟨thread_impl⟩ _Ctype;
    list⟨_Ctype *⟩ tlist;
    typedef typename list⟨_Ctype *⟩::iterator iterator;
  public:
    static int max_parallel_threads;
    void insert(_Ctype *c)
    { tlist.push_back(c); }
    ⟨ thread_group destructor code 9 ⟩;
    ⟨ thread_group::run code 11 ⟩;
  private:
    ⟨ thread_group::run_portion code 10 ⟩;
  };
```

This code is used in section 5.

9. The thread group class maintains list of pointers to threads. It takes responsibility of deallocating the threads. So we implement the destructor.

⟨ **thread_group** destructor code 9 ⟩ ≡

```
~thread_group()
{
  while (¬tlist.empty()) {
    delete tlist.front();
    tlist.pop_front();
  }
}
```

This code is used in section 8.

10. This runs a given number of threads in parallel starting from the given iterator. It returns the first iterator not run.

⟨ **thread_group** :: *run_portion* code 10 ⟩ ≡

```
iterator run_portion(iterator start, int n)
{
  int c = 0;
  for (iterator i = start; c < n; ++i, c++) {
    (*i)→run();
  }
  iterator ret;
  c = 0;
  for (ret = start; c < n; ++ret, c++) {
    _Ttraits::join(*ret);
  }
  return ret;
}
```

This code is used in section 8.

11. Here we run the threads ensuring that not more than *max_parallel_threads* are run in parallel. More over, we do not want to run a too low number of threads, since it is wasting with resource (if there are). Therefore, we run in parallel *max_parallel_threads* batches as long as the remaining threads are greater than the double number. And then the remaining batch (less than $2 * max\_parallel\_threads$) is run half by half.

⟨ **thread_group** :: *run* code 11 ⟩ ≡

```
void run()
{
  int rem = tlist.size();
  iterator pfirst = tlist.begin();
  while (rem > 2 * max_parallel_threads) {
    pfirst = run_portion(pfirst, max_parallel_threads);
    rem −= max_parallel_threads;
  }
  if (rem > max_parallel_threads) {
    pfirst = run_portion(pfirst, rem/2);
    rem −= rem/2;
  }
  run_portion(pfirst, rem);
}
```

This code is used in section 8.

12. Clear. We have only *run*, *detach_run*, *exit* and *join*, since this is only a simple interface.

⟨ **thread_traits** template class declaration 12 ⟩ ≡

```
template<int thread_impl> struct thread_traits {
  typedef typename IF<thread_impl ≡ posix, pthread_t, Empty>::RET _Tthread;
  typedef thread<thread_impl> _Ctype;
  typedef detach_thread<thread_impl> _Dtype;
  static void run(_Ctype *c);
  static void detach_run(_Dtype *c);
  static void exit();
  static void join(_Ctype *c);
};
```

This code is used in section 5.

13. Clear. We have only *init*, *lock*, and *unlock*.

⟨ **mutex_traits** template class declaration 13 ⟩ ≡

```
struct ltmmkey;
typedef pair<const void *, const char *> mmkey;
template<int thread_impl> struct mutex_traits {
  typedef typename IF<thread_impl ≡ posix, pthread_mutex_t, Empty>::RET _Tmutex;
  typedef map<mmkey, pair<_Tmutex, int>, ltmmkey> mutex_int_map;
  static void init(_Tmutex &m);
  static void lock(_Tmutex &m);
  static void unlock(_Tmutex &m);
};
```

This code is used in section 5.

14. Here we define a map of mutexes keyed by a pair of address, and a string. A purpose of the map of mutexes is that, if synchronizing, we need to publish mutexes locking some piece of codes (characterized by the string) accessing the data (characterized by the pointer). So, if any thread needs to pass a **synchro** object, it creates its own with the same address and string, and must look to some public storage to unlock the mutex. If the **synchro** object is created for the first time, the mutex is created and inserted to the map. We count the references to the mutex (number of waiting threads) to know, when it is save to remove the mutex from the map. This is the only purpose of counting the references. Recall, that the mutex is keyed by an address of the data, and without removing, the number of mutexes would only grow.

The map itself needs its own mutex to avoid concurrent insertions and deletions.

⟨ **mutex_map** template class declaration 14 ⟩ ≡

```
struct ltmmkey {
  bool operator()(const mmkey &k1, const mmkey &k2) const
  {
    return k1.first < k2.first ∨ (k1.first ≡ k2.first ∧ strcmp(k1.second, k2.second) < 0);
  }
};
template⟨int thread_impl⟩ class mutex_map : public
      mutex_traits⟨thread_impl⟩::mutex_int_map {
  typedef typename mutex_traits⟨thread_impl⟩::_Tmutex _Tmutex;
  typedef mutex_traits⟨thread_impl⟩ _Mtraits;
  typedef pair⟨_Tmutex, int⟩ mmval;
  typedef map⟨mmkey, mmval, ltmmkey⟩ _Tparent;
  typedef typename _Tparent::iterator iterator;
  typedef typename _Tparent::value_type _mvtype;
  _Tmutex m;
public:
  mutex_map()
  { _Mtraits::init(m); }
  void insert(const void *c, const char *id, const _Tmutex &m)
  { _Tparent::insert(_mvtype(mmkey(c, id), mmval(m, 0))); }
  bool check(const void *c, const char *id) const
  { return _Tparent::find(mmkey(c, id)) ≠ _Tparent::end(); }
  ⟨ mutex_map::get code 15 ⟩;
  ⟨ mutex_map::remove code 16 ⟩;
  void lock_map()
  { _Mtraits::lock(m); }
  void unlock_map()
  { _Mtraits::unlock(m); }
};
```

This code is used in section 5.

15. This returns a pointer to the pair of mutex and count reference number.

⟨ **mutex_map** :: *get* code 15 ⟩ ≡

```
mmval *get(const void *c, const char *id)
{
  iterator it = _Tparent::find(mmkey(c, id));
  if (it ≡ _Tparent::end()) return Λ;
  return &((*it).second);
}
```

This code is used in section 14.

16. This removes unconditionally the mutex from the map regardless its number of references. The only user of this class should be **synchro** class, it implementation must not remove referenced mutex.

⟨ **mutex_map** :: *remove* code 16 ⟩ ≡

```
void remove(const void *c, const char *id)
{
  iterator it = _Tparent::find(mmkey(c, id));
  if (it ≠ _Tparent::end()) erase(it);
}
```

This code is used in section 14.

17. This is the **synchro** class. The constructor of this class tries to lock a mutex for a particular address (identification of data) and string (identification of entry-point). If the mutex is already locked, it waits until it is unlocked and then returns. The destructor releases the lock. The typical use is to construct the object on the stacked of the code being synchronized.

⟨ **synchro** template class declaration 17 ⟩ ≡

```
template⟨int thread_impl⟩ class synchro {
  typedef typename mutex_traits⟨thread_impl⟩::_Tmutex _Tmutex;
  typedef mutex_traits⟨thread_impl⟩ _Mtraits;
public:
  typedef mutex_map⟨thread_impl⟩ mutex_map_t;
private:
  const void *caller;
  const char *iden;
  mutex_map_t &mutmap;
public:
  synchro(const void *c, const char *id, mutex_map_t &mmap)
  : caller(c), iden(id), mutmap(mmap) { lock(); }
  ~synchro()
  { unlock(); }
private:
  ⟨ synchro::lock code 18 ⟩;
  ⟨ synchro::unlock code 19 ⟩;
};
```

This code is used in section 5.

18. The *lock* function acquires the mutex in the map. First it tries to get an exclusive access to the map. Then it increases a number of references of the mutex (if it does not exists, it inserts it). Then unlocks the map, and finally tries to lock the mutex of the map.

⟨ **synchro**::*lock* code 18 ⟩ ≡

```
void lock()
{
  mutmap.lock_map();
  if (¬mutmap.check(caller, iden)) {
    _Tmutex mut;
    _Mtraits::init(mut);
    mutmap.insert(caller, iden, mut);
  }
  mutmap.get(caller, iden)→second++;
  mutmap.unlock_map();
  _Mtraits::lock(mutmap.get(caller, iden)→first);
}
```

This code is used in section 17.

19. The *unlock* function first locks the map. Then releases the lock, and decreases a number of references. If it is zero, it removes the mutex.

⟨ **synchro**::*unlock* code 19 ⟩ ≡

```
void unlock()
{
  mutmap.lock_map();
  if (mutmap.check(caller, iden)) {
    _Mtraits::unlock(mutmap.get(caller, iden)→first);
    mutmap.get(caller, iden)→second−−;
    if (mutmap.get(caller, iden)→second ≡ 0) mutmap.remove(caller, iden);
  }
  mutmap.unlock_map();
}
```

This code is used in section 17.

20. These are traits for conditions. We need *init*, *broadcast*, *wait* and *destroy*.

⟨ **cond_traits** template class declaration 20 ⟩ ≡

```
template⟨int thread_impl⟩ struct cond_traits {
  typedef typename IF⟨thread_impl ≡ posix, pthread_cond_t, Empty⟩::RET _Tcond;
  typedef typename mutex_traits⟨thread_impl⟩::_Tmutex _Tmutex;
  static void init(_Tcond &cond);
  static void broadcast(_Tcond &cond);
  static void wait(_Tcond &cond, _Tmutex &mutex);
  static void destroy(_Tcond &cond);
};
```

This code is used in section 5.

21. Here is the condition counter. It is a counter which starts at 0, and can be increased and decreased. A thread can wait until the counter is changed, this is implemented by condition. After the wait is done, another (or the same) thread, by calling *waitForChange* waits for another change. This can be dangerous, since it is possible to wait for a change which will not happen, because all the threads which can cause the change (by increase of decrease) might had finished.

⟨ **condition_counter** template class declaration 21 ⟩ ≡

```
template⟨int thread_impl⟩ class condition_counter {
    typedef typename mutex_traits⟨thread_impl⟩::_Tmutex _Tmutex;
    typedef typename cond_traits⟨thread_impl⟩::_Tcond _Tcond;
    int counter;
    _Tmutex mut;
    _Tcond cond;
    bool changed;
public:
    ⟨ condition_counter constructor code 22 ⟩;
    ⟨ condition_counter destructor code 23 ⟩;
    ⟨ condition_counter::increase code 24 ⟩;
    ⟨ condition_counter::decrease code 25 ⟩;
    ⟨ condition_counter::waitForChange code 26 ⟩;
};
```

This code is used in section 5.

22. We initialize the counter to 0, and *changed* flag to *true*, since the counter was change from undefined value to 0.

⟨ **condition_counter** constructor code 22 ⟩ ≡

```
condition_counter()
: counter(0), changed(true) {
    mutex_traits⟨thread_impl⟩::init(mut);
    cond_traits⟨thread_impl⟩::init(cond);
}
```

This code is used in section 21.

23. In destructor, we only release the resources associated with the condition.

⟨ **condition_counter** destructor code 23 ⟩ ≡

```
~condition_counter()
{
    cond_traits⟨thread_impl⟩::destroy(cond);
}
```

This code is used in section 21.

24. When increasing, we lock the mutex, advance the counter, remember it is changed, broadcast, and release the mutex.

⟨ **condition_counter** :: *increase* code 24 ⟩ ≡

```
void increase()
{
  mutex_traits<thread_impl>::lock(mut);
  counter++;
  changed = true;
  cond_traits<thread_impl>::broadcast(cond);
  mutex_traits<thread_impl>::unlock(mut);
}
```

This code is used in section 21.

25. Same as increase.

⟨ **condition_counter** :: *decrease* code 25 ⟩ ≡

```
void decrease()
{
  mutex_traits<thread_impl>::lock(mut);
  counter--;
  changed = true;
  cond_traits<thread_impl>::broadcast(cond);
  mutex_traits<thread_impl>::unlock(mut);
}
```

This code is used in section 21.

26. We lock the mutex, and if there was a change since the last call of *waitForChange*, we return immediately, otherwise we wait for the change. The mutex is released.

⟨ **condition_counter** :: *waitForChange* code 26 ⟩ ≡

```
int waitForChange()
{
  mutex_traits<thread_impl>::lock(mut);
  if (¬changed) {
    cond_traits<thread_impl>::wait(cond, mut);
  }
  changed = false;
  int res = counter;
  mutex_traits<thread_impl>::unlock(mut);
  return res;
}
```

This code is used in section 21.

27. The detached thread is the same as joinable **thread**. We only re-implement *run* method to call **thread_traits**::*detach_run*, and add a method which installs a counter. The counter is increased and decreased on the body of the new thread.

⟨ **detach_thread** template class declaration 27 ⟩ ≡

```
template<int thread_impl> class detach_thread : public thread<thread_impl> {
public:
    condition_counter<thread_impl> *counter;
    detach_thread()
    : counter(Λ) { }
    void installCounter(condition_counter<thread_impl> *c)
    { counter = c; }
    void run()
    { thread_traits<thread_impl>::detach_run(this); }
};
```

This code is used in section 5.

28. The detach thread group is (by interface) the same as **thread_group**. The extra thing we have here is the *counter*. The implementation of *insert* and *run* is different.

⟨ **detach_thread_group** template class declaration 28 ⟩ ≡

```
template<int thread_impl> class detach_thread_group {
    typedef thread_traits<thread_impl> _Ttraits;
    typedef cond_traits<thread_impl> _Ctraits;
    typedef detach_thread<thread_impl> _Ctype;
    list<_Ctype *> tlist;
    typedef typename list<_Ctype *>::iterator iterator;
    condition_counter<thread_impl> counter;
public:
    static int max_parallel_threads;
    ⟨ detach_thread_group::insert code 29 ⟩;
    ⟨ detach_thread_group destructor code 30 ⟩;
    ⟨ detach_thread_group::run code 31 ⟩;
};
```

This code is used in section 5.

29. When inserting, the counter is installed to the thread.

⟨ **detach_thread_group**::*insert* code 29 ⟩ ≡

```
void insert(_Ctype *c)
{
    tlist.push_back(c);
    c→installCounter(&counter);
}
```

This code is used in section 28.

30. The destructor is clear.

⟨ **detach_thread_group** destructor code 30 ⟩ ≡

```
~detach_thread_group()
{
  while (¬tlist.empty()) {
    delete tlist.front();
    tlist.pop_front();
  }
}
```

This code is used in section 28.

31. We cycle through all threads in the group, and in each cycle we wait for the change in the *counter*. If the counter indicates less than maximum parallel threads running, then a new thread is run, and the iterator in the list is moved.

At the end we have to wait for all thread to finish.

⟨ **detach_thread_group** :: *run* code 31 ⟩ ≡

```
void run()
{
  int mpt = max_parallel_threads;
  iterator it = tlist.begin();
  while (it ≠ tlist.end()) {
    if (counter.waitForChange() < mpt) {
      counter.increase();
      (*it)→run();
      ++it;
    }
  }
  while (counter.waitForChange() > 0) { }
}
```

This code is used in section 28.

32. Here we only define the specializations for POSIX threads. Then we define the macros. Note that the *PosixSynchro* class construct itself from the static map defined in `sthreads.cpp`.

⟨ POSIX thread specializations 32 ⟩ ≡

```
typedef detach_thread⟨posix⟩ PosixThread;
typedef detach_thread_group⟨posix⟩ PosixThreadGroup;
typedef synchro⟨posix⟩ posix_synchro;
class PosixSynchro : public posix_synchro {
public:
  PosixSynchro(const void *c, const char *id);
};
#define THREAD sthread::PosixThread
#define THREAD_GROUP sthread::PosixThreadGroup
#define SYNCHRO sthread::PosixSynchro
```

This code is used in section 5.

33. Here we define an empty class and use it as thread and mutex. *NoSynchro* class is also empty, but an empty constructor is declared. The empty destructor is declared only to avoid "unused variable warning".

⟨ No threading specializations 33 ⟩ ≡

```
typedef thread⟨empty⟩ NoThread;
typedef thread_group⟨empty⟩ NoThreadGroup;
typedef synchro⟨empty⟩ no_synchro;
class NoSynchro {
public:
  NoSynchro(const void *c, const char *id)
  { }
  ~NoSynchro( )
  { }
};
#define THREAD sthread::NoThread
#define THREAD_GROUP sthread::NoThreadGroup
#define SYNCHRO sthread::NoSynchro
```

This code is used in section 5.

34. End of `sthreads.h` file.

35. Start of `sthreads.h` file. We set the default values for *max_parallel_threads* for both *posix* and *empty* implementation and both joinable and detach group. For *posix* this defaults to uniprocessor machine with hyper-threading, this is 2.

```
#include <cstring>
#include "sthread.h"
#ifdef HAVE_PTHREAD
  namespace sthread {
    template⟨ ⟩ int thread_group⟨posix⟩::max_parallel_threads = 2;
    template⟨ ⟩ int detach_thread_group⟨posix⟩::max_parallel_threads = 2;
    ⟨ POSIX specializations methods 36 ⟩;
  }
#else
  namespace sthread {
    template⟨ ⟩ int thread_group⟨empty⟩::max_parallel_threads = 1;
    template⟨ ⟩ int detach_thread_group⟨empty⟩::max_parallel_threads = 1;
    ⟨ non-threading specialization methods 43 ⟩;
  }
#endif
```

36.

⟨ POSIX specializations methods 36 ⟩ ≡

```
⟨ thread_traits method codes 37 ⟩;
⟨ mutex_traits method codes 38 ⟩;
⟨ cond_traits method codes 39 ⟩;
⟨ PosixSynchro constructor 40 ⟩;
⟨ posix_thread_function code 41 ⟩;
⟨ posix_detach_thread_function code 42 ⟩;
```

This code is used in section 35.

37.

⟨ **thread_traits** method codes 37 ⟩ ≡

```
void *posix_thread_function(void *c);
template⟨ ⟩ void thread_traits⟨posix⟩::run(_Ctype *c)
{
  pthread_create(&(c→getThreadIden( )), Λ, posix_thread_function, (void *) c);
}
void *posix_detach_thread_function(void *c);
template⟨ ⟩ void thread_traits⟨posix⟩::detach_run(_Dtype *c)
{
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_create(&(c→getThreadIden( )), &attr, posix_detach_thread_function, (void *) c);
  pthread_attr_destroy(&attr);
}
template⟨ ⟩ void thread_traits⟨posix⟩::exit( )
{
  pthread_exit(Λ);
}
template⟨ ⟩ void thread_traits⟨posix⟩::join(_Ctype *c)
{
  pthread_join(c→getThreadIden( ), Λ);
}
```

This code is used in section 36.

38.

⟨ **mutex_traits** method codes 38 ⟩ ≡

```
template⟨ ⟩ void mutex_traits⟨posix⟩::init(pthread_mutex_t &m)
{
  pthread_mutex_init(&m, Λ);
}
template⟨ ⟩ void mutex_traits⟨posix⟩::lock(pthread_mutex_t &m)
{
  pthread_mutex_lock(&m);
}
template⟨ ⟩ void mutex_traits⟨posix⟩::unlock(pthread_mutex_t &m)
{
  pthread_mutex_unlock(&m);
}
```

This code is used in section 36.

39.

⟨ **cond_traits** method codes 39 ⟩ ≡

```
template⟨ ⟩ void cond_traits⟨posix⟩::init(_Tcond &cond)
{
  pthread_cond_init(&cond, Λ);
}
template⟨ ⟩ void cond_traits⟨posix⟩::broadcast(_Tcond &cond)
{
  pthread_cond_broadcast(&cond);
}
template⟨ ⟩ void cond_traits⟨posix⟩::wait(_Tcond &cond, _Tmutex &mutex)
{
  pthread_cond_wait(&cond, &mutex);
}
template⟨ ⟩ void cond_traits⟨posix⟩::destroy(_Tcond &cond)
{
  pthread_cond_destroy(&cond);
}
```

This code is used in section 36.

40. Here we instantiate the static map, and construct **PosixSynchro** using that map.

⟨ **PosixSynchro** constructor 40 ⟩ ≡

```
static posix_synchro::mutex_map_t posix_mm;
PosixSynchro::PosixSynchro(const void *c, const char *id)
: posix_synchro(c, id, posix_mm) { }
```

This code is used in section 36.

41. This function is of the type **void** ∗*function*(**void** ∗) as required by POSIX, but it typecasts its argument and runs **operator**( )( ).

⟨ *posix_thread_function* code 41 ⟩ ≡

```
void *posix_thread_function(void *c)
{
  thread_traits⟨posix⟩::_Ctype *ct = (thread_traits⟨posix⟩::_Ctype *) c;
  try {
    ct→operator( )( );
  }
  catch(...)
  {
    ct→exit( );
  }
  return Λ;
}
```

This code is used in section 36.

42.

⟨ *posix_detach_thread_function* code 42 ⟩ ≡

```
void *posix_detach_thread_function(void *c)
{
  thread_traits<posix>::_Dtype *ct = (thread_traits<posix>::_Dtype *) c;
  condition_counter<posix> *counter = ct->counter;
  try {
    ct->operator()();
  }
  catch(...)
  {
    ct->exit();
  }
  if (counter) counter->decrease();
  return Λ;
}
```

This code is used in section 36.

43. The only trait methods we need to work are **thread_traits**::*run* and **thread_traits**::*detach_run*, which directly call **operator**()(). Anything other is empty.

⟨ non-threading specialization methods 43 ⟩ ≡

```
template<> void thread_traits<empty>::run(_Ctype *c)
{
  c->operator()();
}
template<> void thread_traits<empty>::detach_run(_Dtype *c)
{
  c->operator()();
}
template<> void thread_traits<empty>::exit()
{}
template<> void thread_traits<empty>::join(_Ctype *c)
{}
template<> void mutex_traits<empty>::init(Empty &m)
{}
template<> void mutex_traits<empty>::lock(Empty &m)
{}
template<> void mutex_traits<empty>::unlock(Empty &m)
{}
template<> void cond_traits<empty>::init(_Tcond &cond)
{}
template<> void cond_traits<empty>::broadcast(_Tcond &cond)
{}
template<> void cond_traits<empty>::wait(_Tcond &cond, _Tmutex &mutex)
{}
template<> void cond_traits<empty>::destroy(_Tcond &cond)
{}
```

This code is used in section 35.

44. End of `sthreads.h` file.

45. Exception. Start of `tl_exception.h` file.

Within the code we often check some state of variables, typically preconditions or postconditions. If the state is not as required, it is worthless to continue, since this means some fatal error in algorithms. In this case we raise an exception which can be caught at some higher level. This header file defines a simple infrastructure for this.

```
#ifndef TL_EXCEPTION_H
#define TL_EXCEPTION_H
#include <cstring>
#include <cstdio>
  ⟨ body of tl_exception header 46 ⟩;
#endif
```

46. The basic idea of raising an exception if some condition fails is that the conditions is checked only if required. We define global `TL_DEBUG` macro which is integer and says, how many debug messages the programm has to emit. We also define `TL_DEBUG_EXCEPTION` which says, for what values of `TL_DEBUG` we will check for conditions of the exceptions. If the `TL_DEBUG` is equal or higher than `TL_DEBUG_EXCEPTION`, the exception conditions are checked.

We define `TL_RAISE`, and `TL_RAISE_IF` macros which throw an instance of **TLException** if `TL_DEBUG` ≥ `TL_DEBUG_EXCEPTION`. The first is unconditional throw, the second is conditioned by a given expression. Note that if `TL_DEBUG` < `TL_DEBUG_EXCEPTION` then the code is compiled but evaluation of the condition is passed. If code is optimized, the optimizer also passes evaluation of `TL_DEBUG` and `TL_DEBUG_EXCEPTION` comparison (I hope).

We provide default values for `TL_DEBUG` and `TL_DEBUG_EXCEPTION`.

```
⟨ body of tl_exception header 46 ⟩ ≡
#ifndef TL_DEBUG_EXCEPTION
#define TL_DEBUG_EXCEPTION 1
#endif
#ifndef TL_DEBUG
#define TL_DEBUG 0
#endif
#define TL_RAISE(mes)
  if (TL_DEBUG ≥ TL_DEBUG_EXCEPTION) throw TLException(__FILE__, __LINE__, mes);
#define TL_RAISE_IF(expr, mes)
  if (TL_DEBUG ≥ TL_DEBUG_EXCEPTION ∧ (expr)) throw TLException(__FILE__, __LINE__, mes);
  ⟨ TLException class definition 47 ⟩;
```

This code is used in section 45.

47. Primitive exception class containing file name, line number and message.

```
⟨ TLException class definition 47 ⟩ ≡
  class TLException {
    char fname[50];
    int lnum;
    char message[500];
  public:
    TLException(const char *f, int l, const char *mes)
    {
      strncpy(fname, f, 50); fname[49] = '\0';
      strncpy(message, mes, 500); message[499] = '\0';
      lnum = l;
    }
    virtual ~TLException() { }
    virtual void print() const
    { printf("At␣%s:%d:%s\n", fname, lnum, message); }
  };
```

This code is cited in section 1.

This code is used in section 46.

48. End of `tl_exception.h` file.

49. Integer sequence. Start of `int_sequence.h` file.

Here we define an auxiliary abstraction for a sequence of integers. The basic functionality is to hold an ordered sequence of integers with constant length. We prefer using this simple class before STL **vector**⟨**int**⟩ since it is more efficient for our purposes.

The class is used in index of a tensor, in symmetry definition, in Kronecker product dimensions, or as a class of an equivalence. The latter case is not ordered, but we always order equivalence classes in order to ensure unique representativeness. For almost all cases we need the integer sequence to be ordered (sort), or monotonize (indices of folded tensors), or partially monotonize (indices of folded tensors not fully symmetric), or calculate a product of all members or only of a part (used in Kronecker product dimensions). When we calculate offsets in folded tensors, we need to obtain a number of the same items in the front (*getPrefixLength*), and also to add some integer number to all items.

Also, we need to construct a subsequence of a sequence, so some instances do destroy the underlying data, and some not.

```
#ifndef INT_SEQUENCE_H
#define INT_SEQUENCE_H
#include <cstring>
#include <vector>
  using namespace std;
  ⟨ IntSequence class declaration 50 ⟩;
#endif
```

50. The implementation of **IntSequence** is straightforward. It has a pointer *data*, a *length* of the data, and a flag *destroy*, whether the instance must destroy the underlying data.

⟨ **IntSequence** class declaration 50 ⟩ ≡

```
class Symmetry;
class IntSequence {
  int *data;
  int length;
  bool destroy;
public:
  ⟨ IntSequence constructors 51 ⟩;
  ⟨ IntSequence inlines and operators 52 ⟩;
  ⟨ IntSequence orderings 53 ⟩;
  void sort( );
  void monotone( );
  void pmonotone(const Symmetry &s);
  int sum( ) const;
  int mult(int i1, int i2) const;
  int mult( ) const
  { return mult(0, length); }
  void add(int i);
  void add(int f, const IntSequence &s);
  int getPrefixLength( ) const;
  int getNumDistinct( ) const;
  int getMax( ) const;
  bool isPositive( ) const;
  bool isConstant( ) const;
  bool isSorted( ) const;
  void print( ) const;
};
```

This code is cited in section 1.

This code is used in section 49.

51. We have a constructor allocating a given length of data, constructor allocating and then initializing all members to a given number, a copy constructor, a conversion from **vector**⟨**int**⟩, a subsequence constructor, a constructor used for calculating implied symmetry from a more general symmetry and one equivalence class (see **Symmetry** class). Finally we have a constructor which unfolds a sequence with respect to a given symmetry and constructor which inserts a given number to the ordered sequence or given number to a given position.

⟨**IntSequence** constructors 51⟩ ≡

```
IntSequence(int l)
: data(new int[l]), length(l), destroy(true) { }
IntSequence(int l, int n)
: data(new int[l]), length(l), destroy(true) { for (int i = 0; i < length; i++) data[i] = n; }
IntSequence(const IntSequence &s)
: data(new int[s.length]), length(s.length), destroy(true) { memcpy(data, s.data, length * sizeof(int));
  }
IntSequence(IntSequence &s, int i1, int i2)
: data(s.data + i1), length(i2 − i1), destroy(false) { }
IntSequence(const IntSequence &s, int i1, int i2)
: data(new int[i2 − i1]), length(i2 − i1), destroy(true) {
  memcpy(data, s.data + i1, sizeof(int) * length); }
IntSequence(const Symmetry &sy, const vector⟨int⟩ &se);
IntSequence(const Symmetry &sy, const IntSequence &se);
IntSequence(int i, const IntSequence &s);
IntSequence(int i, const IntSequence &s, int pos);
IntSequence(int l, const int *d)
: data(new int[l]), length(l), destroy(true) { memcpy(data, d, sizeof(int) * length); }
```

This code is used in section 50.

52. These are clear inlines and operators.

⟨**IntSequence** inlines and operators 52⟩ ≡

```
const IntSequence &operator=(const IntSequence &s);
virtual ~IntSequence()
{ if (destroy) delete[] data; }
bool operator≡(const IntSequence &s) const;
bool operator≠(const IntSequence &s) const
{ return ¬operator≡(s); }
int &operator[](int i)
{ return data[i]; }
int operator[](int i) const
{ return data[i]; }
int size() const
{ return length; }
```

This code is used in section 50.

53. We provide two orderings. The first **operator** $<$ is the linear lexicographic ordering, the second *less* is the non-linear Cartesian ordering.

⟨ **IntSequence** orderings 53 ⟩ ≡
  bool operator $<$ (**const IntSequence** &s) **const**;
  bool operator$\le$(**const IntSequence** &s) **const**
  { **return** (**operator**$\equiv$(s) $\vee$ **operator** $<$ (s)); }
  bool *lessEq*(**const IntSequence** &s) **const**;
  bool *less*(**const IntSequence** &s) **const**;

This code is used in section 50.

54. End of `int_sequence.h` file.

55. Start of `int_sequence.cpp` file.

#**include** "int_sequence.h"
#**include** "symmetry.h"
#**include** "tl_exception.h"
#**include** <cstdio>
#**include** <climits>
  ⟨ **IntSequence** constructor code 1 56 ⟩;
  ⟨ **IntSequence** constructor code 2 57 ⟩;
  ⟨ **IntSequence** constructor code 3 58 ⟩;
  ⟨ **IntSequence** constructor code 4 59 ⟩;
  ⟨ **IntSequence** :: **operator**= code 60 ⟩;
  ⟨ **IntSequence** :: **operator**$\equiv$ code 61 ⟩;
  ⟨ **IntSequence** :: **operator** $<$ code 62 ⟩;
  ⟨ **IntSequence** :: *lessEq* code 63 ⟩;
  ⟨ **IntSequence** :: *less* code 64 ⟩;
  ⟨ **IntSequence** :: *sort* code 65 ⟩;
  ⟨ **IntSequence** :: *monotone* code 66 ⟩;
  ⟨ **IntSequence** :: *pmonotone* code 67 ⟩;
  ⟨ **IntSequence** :: *sum* code 68 ⟩;
  ⟨ **IntSequence** :: *mult* code 69 ⟩;
  ⟨ **IntSequence** :: *getPrefixLength* code 70 ⟩;
  ⟨ **IntSequence** :: *getNumDistinct* code 71 ⟩;
  ⟨ **IntSequence** :: *getMax* code 72 ⟩;
  ⟨ **IntSequence** :: *add* code 1 73 ⟩;
  ⟨ **IntSequence** :: *add* code 2 74 ⟩;
  ⟨ **IntSequence** :: *isPositive* code 75 ⟩;
  ⟨ **IntSequence** :: *isConstant* code 76 ⟩;
  ⟨ **IntSequence** :: *isSorted* code 77 ⟩;
  ⟨ **IntSequence** :: *print* code 78 ⟩;

56. This unfolds a given integer sequence with respect to the given symmetry. If for example the symmetry is $(2,3)$, and the sequence is (a,b), then the result is (a,a,b,b,b).

⟨ **IntSequence** constructor code 1 56 ⟩ ≡

```
IntSequence::IntSequence(const Symmetry &sy, const IntSequence &se)
  : data(new int[sy.dimen()]), length(sy.dimen()), destroy(true) {
    int k = 0;
    for (int i = 0; i < sy.num(); i++)
      for (int j = 0; j < sy[i]; j++, k++) operator[](k) = se[i];
}
```

This code is used in section 55.

57. This constructs an implied symmetry (implemented as **IntSequence** from a more general symmetry and equivalence class (implemented as **vector**⟨**int**⟩). For example, let the general symmetry be y^3u^2 and the equivalence class is $\{0,4\}$ picking up first and fifth variable, we calculate symmetry (at this point only **IntSequence**) corresponding to the picked variables. These are yu. Thus the constructed sequence must be $(1,1)$, meaning that we picked one y and one u.

⟨ **IntSequence** constructor code 2 57 ⟩ ≡

```
IntSequence::IntSequence(const Symmetry &sy, const vector<int> &se)
  : data(new int[sy.num()]), length(sy.num()), destroy(true) {
    TL_RAISE_IF(sy.dimen() ≤ se[se.size() − 1],
        "Sequence is not reachable by symmetry in IntSequence()");
    for (int i = 0; i < length; i++)
      operator[](i) = 0;
    for (unsigned int i = 0; i < se.size(); i++)
      operator[](sy.findClass(se[i]))++;
}
```

This code is used in section 55.

58. This constructs an ordered integer sequence from the given ordered sequence inserting the given number to the sequence.

⟨ **IntSequence** constructor code 3 58 ⟩ ≡

```
IntSequence::IntSequence(int i, const IntSequence &s)
  : data(new int[s.size() + 1]), length(s.size() + 1), destroy(true) {
    int j = 0;
    while (j < s.size() ∧ s[j] < i) j++;
    for (int jj = 0; jj < j; jj++) operator[](jj) = s[jj];
    operator[](j) = i;
    for (int jj = j; jj < s.size(); jj++) operator[](jj + 1) = s[jj];
}
```

This code is used in section 55.

59.

⟨ **IntSequence** constructor code 4 59 ⟩ ≡

```
IntSequence::IntSequence(int i, const IntSequence &s, int pos)
: data(new int[s.size() + 1]), length(s.size() + 1), destroy(true) {
  TL_RAISE_IF(pos < 0 ∨ pos > s.size(),
      "Wrong␣position␣for␣insertion␣IntSequence␣constructor");
  for (int jj = 0; jj < pos; jj++) operator[](jj) = s[jj];
  operator[](pos) = i;
  for (int jj = pos; jj < s.size(); jj++) operator[](jj + 1) = s[jj];
}
```

This code is used in section 55.

60.

⟨ **IntSequence**::**operator**= code 60 ⟩ ≡

```
const IntSequence &IntSequence::operator=(const IntSequence &s)
{
  TL_RAISE_IF(¬destroy ∧ length ≠ s.length, "Wrong␣length␣for␣in-place␣IntSequence::operator=");
  if (destroy ∧ length ≠ s.length) {
    delete[] data;
    data = new int[s.length];
    destroy = true;
    length = s.length;
  }
  memcpy(data, s.data, sizeof(int) ∗ length);
  return ∗this;
}
```

This code is used in section 55.

61.

⟨ **IntSequence**::**operator**≡ code 61 ⟩ ≡

```
bool IntSequence::operator≡(const IntSequence &s) const
{
  if (size() ≠ s.size()) return false;
  int i = 0;
  while (i < size() ∧ operator[](i) ≡ s[i]) i++;
  return i ≡ size();
}
```

This code is used in section 55.

62. We need some linear irreflexive ordering, we implement it as lexicographic ordering without identity.

⟨ **IntSequence**::**operator** < code 62 ⟩ ≡

```
bool IntSequence::operator < (const IntSequence &s) const
{
  int len = min(size(), s.size());
  int i = 0;
  while (i < len ∧ operator[](i) ≡ s[i]) i++;
  return (i < s.size() ∧ (i ≡ size() ∨ operator[](i) < s[i]));
}
```

This code is used in section 55.

63.

⟨ **IntSequence**::*lessEq* code 63 ⟩ ≡

```
bool IntSequence::lessEq(const IntSequence &s) const
{
  TL_RAISE_IF(size() != s.size(), "Sequence with different lengths in IntSequence::lessEq");
  int i = 0;
  while (i < size() && operator[](i) <= s[i]) i++;
  return (i == size());
}
```

This code is used in section 55.

64.

⟨ **IntSequence**::*less* code 64 ⟩ ≡

```
bool IntSequence::less(const IntSequence &s) const
{
  TL_RAISE_IF(size() != s.size(), "Sequence with different lengths in IntSequence::less");
  int i = 0;
  while (i < size() && operator[](i) < s[i]) i++;
  return (i == size());
}
```

This code is used in section 55.

65. This is a bubble sort, all sequences are usually very short, so this sin might be forgiven.

⟨ **IntSequence**::*sort* code 65 ⟩ ≡

```
void IntSequence::sort()
{
  for (int i = 0; i < length; i++) {
    int swaps = 0;
    for (int j = 0; j < length - 1; j++) {
      if (data[j] > data[j + 1]) {
        int s = data[j + 1];
        data[j + 1] = data[j];
        data[j] = s;
        swaps++;
      }
    }
    if (swaps == 0) return;
  }
}
```

This code is used in section 55.

66. Here we monotonize the sequence. If an item is less then its predecessor, it is equalized.

⟨ **IntSequence**::*monotone* code 66 ⟩ ≡

```
void IntSequence::monotone()
{
  for (int i = 1; i < length; i++)
    if (data[i - 1] > data[i])
      data[i] = data[i - 1];
}
```

This code is used in section 55.

67. This partially monotones the sequence. The partitioning is done by a symmetry. So the subsequence given by the symmetry classes are monotonized. For example, if the symmetry is y^2u^3, and the **IntSequence** is $(5, 3, 1, 6, 4)$, the result is $(5, 5, 1, 6, 6)$.

⟨ **IntSequence**::*pmonotone* code 67 ⟩ ≡

```
void IntSequence::pmonotone(const Symmetry &s)
{
  int cum = 0;
  for (int i = 0; i < s.num(); i++) {
    for (int j = cum + 1; j < cum + s[i]; j++)
      if (data[j - 1] > data[j])
        data[j] = data[j - 1];
    cum += s[i];
  }
}
```

This code is used in section 55.

68. This returns sum of all elements. Useful for symmetries.

⟨ **IntSequence**::*sum* code 68 ⟩ ≡

```
int IntSequence::sum() const
{
  int res = 0;
  for (int i = 0; i < length; i++)
    res += operator[](i);
  return res;
}
```

This code is used in section 55.

69. This returns product of subsequent items. Useful for Kronecker product dimensions.

⟨ **IntSequence**::*mult* code 69 ⟩ ≡

```
int IntSequence::mult(int i1, int i2) const
{
  int res = 1;
  for (int i = i1; i < i2; i++)
    res *= operator[](i);
  return res;
}
```

This code is used in section 55.

70. Return a number of the same items in the beginning of the sequence.

⟨ **IntSequence**::*getPrefixLength* code 70 ⟩ ≡

```
int IntSequence::getPrefixLength() const
{
  int i = 0;
  while (i + 1 < size() ∧ operator[](i + 1) ≡ operator[](0)) i++;
  return i + 1;
}
```

This code is used in section 55.

71. This returns a number of distinct items in the sequence. It supposes that the sequence is ordered. For the empty sequence it returns zero.

⟨ **IntSequence** :: *getNumDistinct* code 71 ⟩ ≡
```
int IntSequence::getNumDistinct() const
{
  int res = 0;
  if (size() > 0) res++;
  for (int i = 1; i < size(); i++)
    if (operator[](i) ≠ operator[](i - 1)) res++;
  return res;
}
```
This code is used in section 55.

72. This returns a maximum of the sequence. If the sequence is empty, it returns the least possible **int** value.

⟨ **IntSequence** :: *getMax* code 72 ⟩ ≡
```
int IntSequence::getMax() const
{
  int res = INT_MIN;
  for (int i = 0; i < size(); i++)
    if (operator[](i) > res) res = operator[](i);
  return res;
}
```
This code is used in section 55.

73.

⟨ **IntSequence** :: *add* code 1 73 ⟩ ≡
```
void IntSequence::add(int i)
{
  for (int j = 0; j < size(); j++) operator[](j) += i;
}
```
This code is used in section 55.

74.

⟨ **IntSequence** :: *add* code 2 74 ⟩ ≡
```
void IntSequence::add(int f, const IntSequence &s)
{
  TL_RAISE_IF(size() ≠ s.size(), "Wrong␣sequence␣length␣in␣IntSequence::add");
  for (int j = 0; j < size(); j++) operator[](j) += f * s[j];
}
```
This code is used in section 55.

75.

⟨ **IntSequence** :: *isPositive* code 75 ⟩ ≡

```
bool IntSequence::isPositive() const
{
  int i = 0;
  while (i < size() ∧ operator[](i) ≥ 0) i++;
  return (i ≡ size());
}
```

This code is used in section 55.

76.

⟨ **IntSequence** :: *isConstant* code 76 ⟩ ≡

```
bool IntSequence::isConstant() const
{
  bool res = true;
  int i = 1;
  while (res ∧ i < size()) {
    res = res ∧ operator[](0) ≡ operator[](i);
    i++;
  }
  return res;
}
```

This code is used in section 55.

77.

⟨ **IntSequence** :: *isSorted* code 77 ⟩ ≡

```
bool IntSequence::isSorted() const
{
  bool res = true;
  int i = 1;
  while (res ∧ i < size()) {
    res = res ∧ operator[](i − 1) ≤ operator[](i);
    i++;
  }
  return res;
}
```

This code is used in section 55.

78. Debug print.

⟨ **IntSequence** :: *print* code 78 ⟩ ≡

```
void IntSequence::print() const
{
  printf("[");
  for (int i = 0; i < size(); i++)
    printf("%2d␣", operator[](i));
  printf("]\n");
}
```

This code is used in section 55.

79. End of `int_sequence.cpp` file.

80. Matrix interface. Start of `twod_matrix.h` file.

Here we make an interface to 2-dimensional matrix defined in the Sylvester module. That abstraction provides an interface to BLAS. The main purpose of this file is to only make its subclass in order to keep the tensor library and Sylvester module independent. So here is mainly renaming of methods.

Similarly as in the Sylvester module we declare two classes **TwoDMatrix** and **ConstTwoDMatrix**. The only purpose of the latter is to allow submatrix construction from const reference arguments.

```
#ifndef TWOD_MATRIX_H
#define TWOD_MATRIX_H
#include "GeneralMatrix.h"
#include <cstdio>
  class TwoDMatrix;
  ⟨ConstTwoDMatrix class declaration 81⟩;
  ⟨TwoDMatrix class declaration 82⟩;
  ⟨Mat4Header class declaration 85⟩;
#endif
```

81. We make two obvious constructors, and then a constructor making submatrix of subsequent columns. We also rename **GeneralMatrix** :: *numRows*( ) and **GeneralMatrix** :: *numCols*( ).

```
⟨ConstTwoDMatrix class declaration 81⟩ ≡
  class ConstTwoDMatrix : public ConstGeneralMatrix {
  public:
    ConstTwoDMatrix(int m, int n, const double *d)
    : ConstGeneralMatrix(d, m, n) { }
    ConstTwoDMatrix(const TwoDMatrix &m);
    ConstTwoDMatrix(const TwoDMatrix &m, int first_col, int num);
    ConstTwoDMatrix(const ConstTwoDMatrix &m, int first_col, int num);
    ConstTwoDMatrix(int first_row, int num, const TwoDMatrix &m);
    ConstTwoDMatrix(int first_row, int num, const ConstTwoDMatrix &m);
    ConstTwoDMatrix(const ConstTwoDMatrix &m, int first_row, int first_col, int rows, int cols)
    : ConstGeneralMatrix(m, first_row, first_col, rows, cols) { }
    virtual ~ConstTwoDMatrix() { }
    int nrows() const
    { return numRows(); }
    int ncols() const
    { return numCols(); }
    void writeMat4(FILE *fd, const char *vname) const;
  };
```

This code is cited in section 1.

This code is used in section 80.

82. Here we do the same as for **ConstTwoDMatrix** plus define methods for copying and adding rows and columns.

Also we have *save* method which dumps the matrix to a file with a given name. The file can be read by Scilab `fscanfMat` function.

⟨ **TwoDMatrix** class declaration 82 ⟩ ≡

```
class TwoDMatrix : public GeneralMatrix {
public:
  TwoDMatrix(int r, int c)
  : GeneralMatrix(r, c) { }
  TwoDMatrix(int r, int c, double *d)
  : GeneralMatrix(d, r, c) { }
  TwoDMatrix(int r, int c, const double *d)
  : GeneralMatrix(d, r, c) { }
  TwoDMatrix(const GeneralMatrix &m)
  : GeneralMatrix(m) { }
  TwoDMatrix(const GeneralMatrix &m, const char *dummy)
  : GeneralMatrix(m, dummy) { }
  TwoDMatrix(const TwoDMatrix &m, int first_col, int num)
  : GeneralMatrix(m, 0, first_col, m.numRows(), num) { }
  TwoDMatrix(TwoDMatrix &m, int first_col, int num)
  : GeneralMatrix(m, 0, first_col, m.numRows(), num) { }
  TwoDMatrix(int first_row, int num, const TwoDMatrix &m)
  : GeneralMatrix(m, first_row, 0, num, m.ncols()) { }
  TwoDMatrix(int first_row, int num, TwoDMatrix &m)
  : GeneralMatrix(m, first_row, 0, num, m.ncols()) { }
  TwoDMatrix(TwoDMatrix &m, int first_row, int first_col, int rows, int cols)
  : GeneralMatrix(m, first_row, first_col, rows, cols) { }
  TwoDMatrix(const TwoDMatrix &m, int first_row, int first_col, int rows, int cols)
  : GeneralMatrix(m, first_row, first_col, rows, cols) { }
  TwoDMatrix(const ConstTwoDMatrix &a, const ConstTwoDMatrix &b)
  : GeneralMatrix(a, b) { }
  virtual ~TwoDMatrix() { }
  int nrows() const
  { return numRows(); }
  int ncols() const
  { return numCols(); }
  ⟨ TwoDMatrix row methods declarations 83 ⟩;
  ⟨ TwoDMatrix column methods declarations 84 ⟩;
  void save(const char *fname) const;
  void writeMat4(FILE *fd, const char *vname) const
  { ConstTwoDMatrix(*this).writeMat4(fd, vname); }
};
```

This code is cited in section 1.
This code is used in section 80.

83.

⟨ **TwoDMatrix** row methods declarations 83 ⟩ ≡

```
void copyRow(int from, int to);
void copyRow(const ConstTwoDMatrix &m, int from, int to);
void copyRow(const TwoDMatrix &m, int from, int to)
{ copyRow(ConstTwoDMatrix(m), from, to); }
void addRow(const ConstTwoDMatrix &m, int from, int to)
{ addRow(1.0, m, from, to); }
void addRow(const TwoDMatrix &m, int from, int to)
{ addRow(1.0, ConstTwoDMatrix(m), from, to); }
void addRow(double d, const ConstTwoDMatrix &m, int from, int to);
void addRow(double d, const TwoDMatrix &m, int from, int to)
{ addRow(d, ConstTwoDMatrix(m), from, to); }
```

This code is used in section 82.

84.

⟨ **TwoDMatrix** column methods declarations 84 ⟩ ≡

```
void copyColumn(int from, int to);
void copyColumn(const ConstTwoDMatrix &m, int from, int to);
void copyColumn(const TwoDMatrix &m, int from, int to)
{ copyColumn(ConstTwoDMatrix(m), from, to); }
void addColumn(const ConstTwoDMatrix &m, int from, int to)
{ addColumn(1.0, ConstTwoDMatrix(m), from, to); }
void addColumn(const TwoDMatrix &m, int from, int to)
{ addColumn(1.0, ConstTwoDMatrix(m), from, to); }
void addColumn(double d, const ConstTwoDMatrix &m, int from, int to);
void addColumn(double d, const TwoDMatrix &m, int from, int to)
{ addColumn(d, ConstTwoDMatrix(m), from, to); }
```

This code is used in section 82.

85.

⟨ **Mat4Header** class declaration 85 ⟩ ≡

```
class Mat4Header {
  int type;
  int rows;
  int cols;
  int imagf;
  int namelen;
  const char *vname;
public:
  Mat4Header(const ConstTwoDMatrix &m, const char *vname);
  Mat4Header(const ConstTwoDMatrix &m, const char *vname, const char *dummy);
  void write(FILE *fd) const;
};
```

This code is used in section 80.

86. End of `twod_matrix.h` file.

87. Start of `twod_matrix.cpp` file.

```
#include "twod_matrix.h"
#include "tl_exception.h"
  ⟨ ConstTwoDMatrix constructors 88 ⟩;
  ⟨ ConstTwoDMatrix::writeMat4 code 89 ⟩;
  ⟨ TwoDMatrix row methods code 90 ⟩;
  ⟨ TwoDMatrix column methods code 91 ⟩;
  ⟨ TwoDMatrix::save code 92 ⟩;
  ⟨ Mat4Header constructor 1 code 93 ⟩;
  ⟨ Mat4Header constructor 2 code 94 ⟩;
  ⟨ Mat4Header::write code 95 ⟩;
```

88.

```
⟨ ConstTwoDMatrix constructors 88 ⟩ ≡
  ConstTwoDMatrix::ConstTwoDMatrix(const TwoDMatrix &m)
  : ConstGeneralMatrix(m) { }
  ConstTwoDMatrix::ConstTwoDMatrix(const TwoDMatrix &m, int first_col, int num)
  : ConstGeneralMatrix(m, 0, first_col, m.nrows(), num) { }
  ConstTwoDMatrix::ConstTwoDMatrix(const ConstTwoDMatrix &m, int first_col, int num)
  : ConstGeneralMatrix(m, 0, first_col, m.nrows(), num) { }
  ConstTwoDMatrix::ConstTwoDMatrix(int first_row, int num, const TwoDMatrix &m)
  : ConstGeneralMatrix(m, first_row, 0, num, m.ncols()) { }
  ConstTwoDMatrix::ConstTwoDMatrix(int first_row, int num, const ConstTwoDMatrix &m)
  : ConstGeneralMatrix(m, first_row, 0, num, m.ncols()) { }
```

This code is used in section 87.

89.

```
⟨ ConstTwoDMatrix::writeMat4 code 89 ⟩ ≡
  void ConstTwoDMatrix::writeMat4(FILE *fd, const char *vname) const
  {
    Mat4Header header(*this, vname);
    header.write(fd);
    for (int j = 0; j < ncols(); j++)
      for (int i = 0; i < nrows(); i++) fwrite(&(get(i, j)), sizeof(double), 1, fd);
  }
```

This code is used in section 87.

90.

⟨ **TwoDMatrix** row methods code 90 ⟩ ≡

```
void TwoDMatrix::copyRow(int from, int to)
{
  if (from ≠ to) copyRow(ConstTwoDMatrix(*this), from, to);
}
void TwoDMatrix::copyRow(const ConstTwoDMatrix &m, int from, int to)
{
  ConstVector fr_row(from, m);
  Vector to_row(to, *this);
  to_row = fr_row;
}
void TwoDMatrix::addRow(double d, const ConstTwoDMatrix &m, int from, int to)
{
  ConstVector fr_row(from, m);
  Vector to_row(to, *this);
  to_row.add(d, fr_row);
}
```

This code is used in section 87.

91.

⟨ **TwoDMatrix** column methods code 91 ⟩ ≡

```
void TwoDMatrix::copyColumn(int from, int to)
{
  if (from ≠ to) copyColumn(ConstTwoDMatrix(*this), from, to);
}
void TwoDMatrix::copyColumn(const ConstTwoDMatrix &m, int from, int to)
{
  ConstVector fr_col(m, from);
  Vector to_col(*this, to);
  to_col = fr_col;
}
void TwoDMatrix::addColumn(double d, const ConstTwoDMatrix &m, int from, int to)
{
  ConstVector fr_col(m, from);
  Vector to_col(*this, to);
  to_col.add(d, fr_col);
}
```

This code is used in section 87.

92.

⟨ **TwoDMatrix** :: *save* code 92 ⟩ ≡

```
void TwoDMatrix::save(const char *fname) const
{
  FILE *fd;
  if (Λ ≡ (fd = fopen(fname, "w"))) {
    TL_RAISE("Cannot␣open␣file␣for␣writing␣in␣TwoDMatrix::save");
  }
  for (int row = 0; row < nrows(); row++) {
    for (int col = 0; col < ncols(); col++) fprintf(fd, "␣%20.10g", get(row, col));
    fprintf(fd, "\n");
  }
  fclose(fd);
}
```

This code is used in section 87.

93. This constructs a MAT-4 header for Little Endian dense real double matrix.

⟨ **Mat4Header** constructor 1 code 93 ⟩ ≡

```
Mat4Header::Mat4Header(const ConstTwoDMatrix &m, const char *vn)
: type(0), rows(m.nrows()), cols(m.ncols()), imagf(0), namelen(strlen(vn) + 1), vname(vn) { }
```

This code is used in section 87.

94. This constructs a MAT-4 header for text matrix.

⟨ **Mat4Header** constructor 2 code 94 ⟩ ≡

```
Mat4Header::Mat4Header(const ConstTwoDMatrix &m, const char *vn, const char *dummy)
: type(1), rows(m.nrows()), cols(m.ncols()), imagf(0), namelen(strlen(vn) + 1), vname(vn) { }
```

This code is used in section 87.

95.

⟨ **Mat4Header** :: *write* code 95 ⟩ ≡

```
void Mat4Header::write(FILE *fd) const
{
  fwrite(&type, sizeof(int), 1, fd);
  fwrite(&rows, sizeof(int), 1, fd);
  fwrite(&cols, sizeof(int), 1, fd);
  fwrite(&imagf, sizeof(int), 1, fd);
  fwrite(&namelen, sizeof(int), 1, fd);
  fwrite(vname, 1, namelen, fd);
}
```

This code is used in section 87.

96. End of `twod_matrix.cpp` file.

97. Kronecker product. Start of `kron_prod.h` file.

Here we define an abstraction for a Kronecker product of a sequence of matrices. This is $A_1 \otimes \ldots \otimes A_n$. Obviously we do not store the product in memory. First we need to represent a dimension of the Kronecker product. Then we represent the Kronecker product, simply it is the Kronecker product dimension with a vector of references to the matrices $A_1, \ldots, A_n$.

The main task of this class is to calculate a matrix product $B \cdot (A_1 \otimes A_2 \otimes \ldots \otimes A_n)$ which in our application has much more moderate dimensions than $A_1 \otimes A_2 \otimes \ldots \otimes A_n$. We calculate it as

$$B \cdot (A_1 \otimes I) \cdot \ldots \cdot (I \otimes A_i \otimes I) \cdot \ldots \cdot (I \otimes A_n)$$

where dimensions of identity matrices differ and are given by the chosen order. One can naturally ask, whether there is some optimal order minimizing maximum storage needed for intermediate results. The optimal ordering is implemented by class **KronProdAllOptim**.

For this multiplication, we also need to represent products of type $A \otimes I$, $I \otimes A \otimes I$, and $I \otimes A$.

```
#ifndef KRON_PROD_H
#define KRON_PROD_H
#include "twod_matrix.h"
#include "permutation.h"
#include "int_sequence.h"
  class KronProdAll;
  class KronProdAllOptim;
  class KronProdIA;
  class KronProdIAI;
  class KronProdAI;
  ⟨ KronProdDimens class declaration 98 ⟩;
  ⟨ KronProd class declaration 102 ⟩;
  ⟨ KronProdAll class declaration 103 ⟩;
  ⟨ KronProdAllOptim class declaration 104 ⟩;
  ⟨ KronProdIA class declaration 105 ⟩;
  ⟨ KronProdAI class declaration 106 ⟩;
  ⟨ KronProdIAI class declaration 107 ⟩;
#endif
```

98. **KronProdDimens** maintains a dimension of the Kronecker product. So, it maintains two sequences, one for rows, and one for columns.

```
⟨ KronProdDimens class declaration 98 ⟩ ≡
  class KronProdDimens {
    friend class KronProdAll;
    friend class KronProdAllOptim;
    friend class KronProdIA;
    friend class KronProdIAI;
    friend class KronProdAI;
  private:
    IntSequence rows;
    IntSequence cols;
  public:
    ⟨ KronProdDimens constructors 99 ⟩;
    ⟨ KronProdDimens inline operators 100 ⟩;
    ⟨ KronProdDimens inline methods 101 ⟩;
  };
```

This code is used in section 97.

99. We define three constructors. First initializes to a given dimension, and all rows and cols are set to zeros. Second is a copy constructor. The third constructor takes dimensions of $A_1 \otimes A_2 \otimes \ldots \otimes A_n$, and makes dimensions of $I \otimes A_i \otimes I$, or $I \otimes A_n$, or $A_1 \otimes I$ for a given i. The dimensions of identity matrices are such that

$$A_1 \otimes A_2 \otimes \ldots \otimes A_n = (A_1 \otimes I) \cdot \ldots \cdot (I \otimes A_i \otimes I) \cdot \ldots \cdot (I \otimes A_n)$$

Note that the matrices on the right do not commute only because sizes of identity matrices which are then given by this ordering.

⟨ **KronProdDimens** constructors 99 ⟩ ≡

```
KronProdDimens(int dim)
: rows(dim, 0), cols(dim, 0) { }
KronProdDimens(const KronProdDimens &kd)
: rows(kd.rows), cols(kd.cols) { }
KronProdDimens(const KronProdDimens &kd, int i);
```

This code is used in section 98.

100.

⟨ **KronProdDimens** inline operators 100 ⟩ ≡

```
const KronProdDimens &operator=(const KronProdDimens &kd)
{ rows = kd.rows; cols = kd.cols; return *this; }
bool operator==(const KronProdDimens &kd) const
{ return rows == kd.rows && cols == kd.cols; }
```

This code is used in section 98.

101.

⟨ **KronProdDimens** inline methods 101 ⟩ ≡

```
int dimen() const
{ return rows.size(); }
void setRC(int i, int r, int c)
{ rows[i] = r; cols[i] = c; }
void getRC(int i, int &r, int &c) const
{ r = rows[i]; c = cols[i]; }
void getRC(int &r, int &c) const
{ r = rows.mult(); c = cols.mult(); }
int nrows() const
{ return rows.mult(); }
int ncols() const
{ return cols.mult(); }
int nrows(int i) const
{ return rows[i]; }
int ncols(int i) const
{ return cols[i]; }
```

This code is used in section 98.

102. Here we define an abstract class for all Kronecker product classes, which are **KronProdAll** (the most general), **KronProdIA** (for $I \otimes A$), **KronProdAI** (for $A \otimes I$), and **KronProdIAI** (for $I \otimes A \otimes I$). The purpose of the super class is to only define some common methods and common member *kpd* for dimensions and declare pure virtual *mult* which is implemented by the subclasses.

The class also contains a static method *kronMult*, which calculates a Kronecker product of two vectors and stores it in the provided vector. It is useful at a few points of the library.

⟨**KronProd** class declaration 102⟩ ≡

```
class KronProd {
protected:
  KronProdDimens kpd;
public:
  KronProd(int dim)
  : kpd(dim) { }
  KronProd(const KronProdDimens &kd)
  : kpd(kd) { }
  KronProd(const KronProd &kp)
  : kpd(kp.kpd) { }
  virtual ~KronProd() { }
  int dimen() const
  { return kpd.dimen(); }
  virtual void mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const = 0;
  void mult(const TwoDMatrix &in, TwoDMatrix &out) const
  { mult(ConstTwoDMatrix(in), out); }
  void checkDimForMult(const ConstTwoDMatrix &in, const TwoDMatrix &out) const;
  void checkDimForMult(const TwoDMatrix &in, const TwoDMatrix &out) const
  { checkDimForMult(ConstTwoDMatrix(in), out); }
  static void kronMult(const ConstVector &v1, const ConstVector &v2, Vector &res);
  int nrows() const
  { return kpd.nrows(); }
  int ncols() const
  { return kpd.ncols(); }
  int nrows(int i) const
  { return kpd.nrows(i); }
  int ncols(int i) const
  { return kpd.ncols(i); }
};
```

This code is used in section 97.

103. **KronProdAll** is a main class of this file. It represents the Kronecker product $A_1 \otimes A_2 \otimes \ldots \otimes A_n$. Besides dimensions, it stores pointers to matrices in *matlist* array. If a pointer is null, then the matrix is considered to be unit. The array is set by calls to *setMat* method (for real matrices) or *setUnit* method (for unit matrices).

The object is constructed by a constructor, which allocates the *matlist* and initializes dimensions to zeros. Then a caller must feed the object with matrices by calling *setMat* and *setUnit* repeatedly for different indices.

We implement the *mult* method of **KronProd**, and a new method *multRows*, which creates a vector of kronecker product of all rows of matrices in the object. The rows are given by the **IntSequence**.

⟨ **KronProdAll** class declaration 103 ⟩ ≡

```
class KronProdAll : public KronProd {
  friend class KronProdIA;
  friend class KronProdIAI;
  friend class KronProdAI;
protected:
  const TwoDMatrix **const matlist;
public:
  KronProdAll(int dim)
  : KronProd(dim), matlist(new const TwoDMatrix*[dim]) { }
  virtual ~KronProdAll()
  { delete[] matlist; }
  void setMat(int i, const TwoDMatrix &m);
  void setUnit(int i, int n);
  const TwoDMatrix &getMat(int i) const
  { return *(matlist[i]); }
  void mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const;
  Vector *multRows(const IntSequence &irows) const;
private:
  bool isUnit() const;
};
```

This code is cited in section 1.

This code is used in section 97.

104. The class **KronProdAllOptim** minimizes memory consumption of the product $B\cdot(A_1\otimes A_2\otimes\ldots\otimes A_k)$. The optimization is done by reordering of the matrices $A_1,\ldots,A_k$, in order to minimize a sum of all storages needed for intermediate results. The optimal ordering is also nearly optimal with respect to number of flops.

Let (m_i,n_i) be dimensions of A_i. It is easy to observe, that for i-th step we need storage of $r\cdot n_1\cdot\ldots\cdot n_i\cdot m_{i+1}\cdot\ldots\cdot m_k$, where r is a number of rows of B. To minimize the sum through all i over all permutations of matrices, it is equivalent to minimize the sum $\sum_{i=1}^{k}\frac{m_{i+1}\cdot\ldots\cdot m_k}{n_{i+1}\cdot\ldots\cdot n_k}$. The optimal ordering will yield $\frac{m_k}{n_k}\leq\frac{m_{k-1}}{n_{k-1}}\ldots\leq\frac{m_1}{n_1}$.

Now observe, that the number of flops for i-th step is $r\cdot n_1\cdot\ldots\cdot n_i\cdot m_i\cdot\ldots\cdot m_k$. In order to minimize a number of flops, it is equivalent to minimize $\sum_{i=1}^{k}m_i\frac{m_{i+1}\cdot\ldots\cdot m_k}{n_{i+1}\cdot\ldots\cdot n_k}$. Note that, normally, the m_i does not change as much as $n_{j+1},\ldots,n_k$, so the ordering minimizing the memory will be nearly optimal with respect to number of flops.

The class **KronProdAllOptim** inherits from **KronProdAll**. A public method *optimizeOrder* does the reordering. The permutation is stored in *oper*. So, as long as *optimizeOrder* is not called, the class is equivalent to **KronProdAll**.

⟨ **KronProdAllOptim** class declaration 104 ⟩ ≡

```
class KronProdAllOptim : public KronProdAll {
protected:
  Permutation oper;
public:
  KronProdAllOptim(int dim)
  : KronProdAll(dim), oper(dim) { }
  void optimizeOrder();
  const Permutation &getPer() const
  { return oper; }
};
```

This code is cited in sections 1 and 133.

This code is used in section 97.

105. This class represents $I\otimes A$. We have only one reference to the matrix, which is set by constructor.

⟨ **KronProdIA** class declaration 105 ⟩ ≡

```
class KronProdIA : public KronProd {
  friend class KronProdAll;
  const TwoDMatrix &mat;
public:
  KronProdIA(const KronProdAll &kpa)
  : KronProd(KronProdDimens(kpa.kpd, kpa.dimen() - 1)), mat(kpa.getMat(kpa.dimen() - 1)) { }
  void mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const;
};
```

This code is used in section 97.

106. This class represents $A \otimes I$. We have only one reference to the matrix, which is set by constructor.

⟨ **KronProdAI** class declaration 106 ⟩ ≡

```
class KronProdAI : public KronProd {
  friend class KronProdIAI;
  friend class KronProdAll;
  const TwoDMatrix &mat;
public:
  KronProdAI(const KronProdAll &kpa)
  : KronProd(KronProdDimens(kpa.kpd, 0)), mat(kpa.getMat(0)) { }
  KronProdAI(const KronProdIAI &kpiai);
  void mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const;
};
```

This code is used in section 97.

107. This class represents $I \otimes A \otimes I$. We have only one reference to the matrix, which is set by constructor.

⟨ **KronProdIAI** class declaration 107 ⟩ ≡

```
class KronProdIAI : public KronProd {
  friend class KronProdAI;
  friend class KronProdAll;
  const TwoDMatrix &mat;
public:
  KronProdIAI(const KronProdAll &kpa, int i)
  : KronProd(KronProdDimens(kpa.kpd, i)), mat(kpa.getMat(i)) { }
  void mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const;
};
```

This code is used in section 97.

108. End of `kron_prod.h` file.

109. Start of `kron_prod.cpp` file.

```
#include "kron_prod.h"
#include "tl_exception.h"
#include <cstdio>
  ⟨ KronProdDimens constructor code 110 ⟩;
  ⟨ KronProd::checkDimForMult code 114 ⟩;
  ⟨ KronProd::kronMult code 115 ⟩;
  ⟨ KronProdAll::setMat code 116 ⟩;
  ⟨ KronProdAll::setUnit code 117 ⟩;
  ⟨ KronProdAll::isUnit code 118 ⟩;
  ⟨ KronProdAll::multRows code 130 ⟩;
  ⟨ KronProdIA::mult code 119 ⟩;
  ⟨ KronProdAI constructor code 120 ⟩;
  ⟨ KronProdAI::mult code 121 ⟩;
  ⟨ KronProdIAI::mult code 122 ⟩;
  ⟨ KronProdAll::mult code 123 ⟩;
  ⟨ KronProdAllOptim::optimizeOrder code 133 ⟩;
```

110. Here we construct Kronecker product dimensions from Kronecker product dimensions by picking a given matrix and all other set to identity. The constructor takes dimensions of $A_1 \otimes A_2 \otimes \ldots \otimes A_n$, and makes dimensions of $I \otimes A_i \otimes I$, or $I \otimes A_n$, or $A_1 \otimes I$ for a given i. The identity matrices must fit into the described order. See header file.

We first decide what is a length of the resulting dimensions. Possible length is three for $I \otimes A \otimes I$, and two for $I \otimes A$, or $A \otimes I$.

Then we fork according to i.

⟨ **KronProdDimens** constructor code 110 ⟩ ≡

```
KronProdDimens::KronProdDimens(const KronProdDimens &kd, int i)
: rows((i ≡ 0 ∨ i ≡ kd.dimen() − 1) ? (2) : (3)), cols((i ≡ 0 ∨ i ≡ kd.dimen() − 1) ? (2) : (3)) {
  TL_RAISE_IF(i < 0 ∨ i ≥ kd.dimen(), "Wrong␣index␣for␣pickup␣in␣KronProdDimens␣constructor");
  int kdim = kd.dimen();
  if (i ≡ 0) {
    ⟨ set AI dimensions 111 ⟩;
  }
  else if (i ≡ kdim − 1) {
    ⟨ set IA dimensions 112 ⟩;
  }
  else {
    ⟨ set IAI dimensions 113 ⟩;
  }
}
```

This code is used in section 109.

111. The first rows and cols are taken from kd. The dimensions of identity matrix is a number of rows in $A_2 \otimes \ldots \otimes A_n$ since the matrix $A_1 \otimes I$ is the first.

⟨ set AI dimensions 111 ⟩ ≡

```
rows[0] = kd.rows[0];
rows[1] = kd.rows.mult(1, kdim);
cols[0] = kd.cols[0];
cols[1] = rows[1];
```

This code is used in section 110.

112. The second dimension is taken from kd. The dimensions of identity matrix is a number of columns of $A_1 \otimes \ldots A_{n-1}$, since the matrix $I \otimes A_n$ is the last.

⟨ set IA dimensions 112 ⟩ ≡

```
rows[0] = kd.cols.mult(0, kdim − 1);
rows[1] = kd.rows[kdim − 1];
cols[0] = rows[0];
cols[1] = kd.cols[kdim − 1];
```

This code is used in section 110.

113. The dimensions of the middle matrix are taken from *kd*. The dimensions of the first identity matrix are a number of columns of $A_1 \otimes \ldots \otimes A_{i-1}$, and the dimensions of the last identity matrix are a number of rows of $A_{i+1} \otimes \ldots \otimes A_n$.

⟨ set IAI dimensions 113 ⟩ ≡

```
rows[0] = kd.cols.mult(0, i);
cols[0] = rows[0];
rows[1] = kd.rows[i];
cols[1] = kd.cols[i];
cols[2] = kd.rows.mult(i + 1, kdim);
rows[2] = cols[2];
```

This code is used in section 110.

114. This raises an exception if dimensions are bad for multiplication $out = in * \mathbf{this}$.

⟨ **KronProd**::*checkDimForMult* code 114 ⟩ ≡

```
void KronProd::checkDimForMult(const ConstTwoDMatrix &in, const TwoDMatrix &out)
      const
  {
    int my_rows;
    int my_cols;
    kpd.getRC(my_rows, my_cols);
    TL_RAISE_IF(in.nrows() ≠ out.nrows() ∨ in.ncols() ≠ my_rows,
        "Wrong␣dimensions␣for␣KronProd␣in␣KronProd::checkDimForMult");
  }
```

This code is used in section 109.

115. Here we Kronecker multiply two given vectors *v1* and *v2* and store the result in preallocated *res*.

⟨ **KronProd**::*kronMult* code 115 ⟩ ≡

```
void KronProd::kronMult(const ConstVector &v1, const ConstVector &v2, Vector &res)
  {
    TL_RAISE_IF(res.length() ≠ v1.length() * v2.length(),
        "Wrong␣vector␣lengths␣in␣KronProd::kronMult");
    res.zeros();
    for (int i = 0; i < v1.length(); i++) {
      Vector sub(res, i * v2.length(), v2.length());
      sub.add(v1[i], v2);
    }
  }
```

This code is used in section 109.

116.

⟨ **KronProdAll**::*setMat* code 116 ⟩ ≡

```
void KronProdAll::setMat(int i, const TwoDMatrix &m)
  {
    matlist[i] = &m;
    kpd.setRC(i, m.nrows(), m.ncols());
  }
```

This code is used in section 109.

117.

⟨ **KronProdAll** :: *setUnit* code 117 ⟩ ≡

```
void KronProdAll::setUnit(int i, int n)
{
  matlist[i] = Λ;
  kpd.setRC(i, n, n);
}
```

This code is used in section 109.

118.

⟨ **KronProdAll** :: *isUnit* code 118 ⟩ ≡

```
bool KronProdAll::isUnit() const
{
  int i = 0;
  while (i < dimen() ∧ matlist[i] ≡ Λ) i++;
  return i ≡ dimen();
}
```

This code is used in section 109.

119. Here we multiply $B \cdot (I \otimes A)$. If m is a dimension of the identity matrix, then the product is equal to $B \cdot \mathrm{diag}_m(A)$. If B is partitioned accordingly, then the result is $[B_1 A, B_2 A, \ldots B_m A]$.

Here, *outi* are partitions of *out*, *ini* are const partitions of *in*, and *id_cols* is m. We employ level-2 BLAS.

⟨ **KronProdIA** :: *mult* code 119 ⟩ ≡

```
void KronProdIA::mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const
{
  checkDimForMult(in, out);
  int id_cols = kpd.cols[0];
  ConstTwoDMatrix a(mat);
  for (int i = 0; i < id_cols; i++) {
    TwoDMatrix outi(out, i * a.ncols(), a.ncols());
    ConstTwoDMatrix ini(in, i * a.nrows(), a.nrows());
    outi.mult(ini, a);
  }
}
```

This code is used in section 109.

120. Here we construct **KronProdAI** from **KronProdIAI**. It is clear.

⟨ **KronProdAI** constructor code 120 ⟩ ≡

```
KronProdAI::KronProdAI(const KronProdIAI &kpiai)
: KronProd(KronProdDimens(2)), mat(kpiai.mat) {
  kpd.rows[0] = mat.nrows();
  kpd.cols[0] = mat.ncols();
  kpd.rows[1] = kpiai.kpd.rows[2];
  kpd.cols[1] = kpiai.kpd.cols[2];
}
```

This code is used in section 109.

121. Here we multiply $B \cdot (A \otimes I)$. Let the dimension of the matrix A be $m \times n$, the dimension of I be p, and a number of rows of B be q. We use the fact that $B \cdot (A \otimes I) = \mathrm{reshape}(\mathrm{reshape}(B, q, mp) \cdot A, q, np)$. This works only for matrix B, whose storage has leading dimension equal to number of rows.

For cases where the leading dimension is not equal to the number of rows, we partition the matrix $A \otimes I$ to $m \times n$ square partitions $a_{ij}I$. Therefore, we partition B to m partitions $[B_1, B_2, \ldots, B_m]$. Each partition of B has the same number of columns as the identity matrix. If R denotes the resulting matrix, then it can be partitioned to n partitions $[R_1, R_2, \ldots, R_n]$. Each partition of R has the same number of columns as the identity matrix. Then we have $R_i = \sum a_{ji} B_j$.

In code, *outi* is R_i, *ini* is B_j, and *id_cols* is a dimension of the identity matrix

⟨ **KronProdAI** :: *mult* code 121 ⟩ ≡

```
void KronProdAI::mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const
{
  checkDimForMult(in, out);
  int id_cols = kpd.cols[1];
  ConstTwoDMatrix a(mat);
  if (in.getLD() ≡ in.nrows()) {
    ConstTwoDMatrix in_resh(in.nrows() * id_cols, a.nrows(), in.getData().base());
    TwoDMatrix out_resh(in.nrows() * id_cols, a.ncols(), out.getData().base());
    out_resh.mult(in_resh, a);
  }
  else {
    out.zeros();
    for (int i = 0; i < a.ncols(); i++) {
      TwoDMatrix outi(out, i * id_cols, id_cols);
      for (int j = 0; j < a.nrows(); j++) {
        ConstTwoDMatrix ini(in, j * id_cols, id_cols);
        outi.add(a.get(j, i), ini);
      }
    }
  }
}
```

This code is used in section 109.

122. Here we multiply $B \cdot (I \otimes A \otimes I)$. If n is a dimension of the first identity matrix, then we multiply $B \cdot \mathrm{diag}_n(A \otimes I)$. So we partition B and result R accordingly, and multiply $B_i \cdot (A \otimes I)$, which is in fact **KronProdAI**::*mult*. Note that number of columns of partitions of B are number of rows of $A \otimes I$, and number of columns of R are number of columns of $A \otimes I$.

In code, *id_cols* is n, *akronid* is a Kronecker product object of $A \otimes I$, and *in_bl_width*, and *out_bl_width* are rows and cols of $A \otimes I$.

⟨ **KronProdIAI**::*mult* code 122 ⟩ ≡

```
void KronProdIAI::mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const
{
  checkDimForMult(in, out);
  int id_cols = kpd.cols[0];
  KronProdAI akronid(*this);
  int in_bl_width;
  int out_bl_width;
  akronid.kpd.getRC(in_bl_width, out_bl_width);
  for (int i = 0; i < id_cols; i++) {
    TwoDMatrix outi(out, i * out_bl_width, out_bl_width);
    ConstTwoDMatrix ini(in, i * in_bl_width, in_bl_width);
    akronid.mult(ini, outi);
  }
}
```

This code is used in section 109.

123. Here we multiply $B \cdot (A_1 \otimes \ldots \otimes A_n)$. First we multiply $B \cdot (A_1 \otimes)$, then this is multiplied by all $I \otimes A_i \otimes I$, and finally by $I \otimes A_n$.

If the dimension of the Kronecker product is only 1, then we multiply two matrices in straight way and return.

The intermediate results are stored on heap pointed by *last*. A new result is allocated, and then the former storage is deallocated.

We have to be careful in cases when last or first matrix is unit and no calculations are performed in corresponding codes. The codes should handle *last* safely also if no calcs are done.

⟨ **KronProdAll**::*mult* code 123 ⟩ ≡

```
void KronProdAll::mult(const ConstTwoDMatrix &in, TwoDMatrix &out) const
{
  ⟨ quick copy if product is unit 124 ⟩;
  ⟨ quick zero if one of the matrices is zero 125 ⟩;
  ⟨ quick multiplication if dimension is 1 126 ⟩;
  int c;
  TwoDMatrix *last = Λ;
  ⟨ perform first multiplication AI 127 ⟩;
  ⟨ perform intermediate multiplications IAI 128 ⟩;
  ⟨ perform last multiplication IA 129 ⟩;
}
```

This code is used in section 109.

124.

⟨ quick copy if product is unit 124 ⟩ ≡

```
if (isUnit()) {
  out.zeros();
  out.add(1.0, in);
  return;
}
```

This code is used in section 123.

125. If one of the matrices is exactly zero or the *in* matrix is zero, set out to zero and return

⟨ quick zero if one of the matrices is zero 125 ⟩ ≡

```
bool is_zero = false;
for (int i = 0; i < dimen() ∧ ¬is_zero; i++) is_zero = matlist[i] ∧ matlist[i]→isZero();
if (is_zero ∨ in.isZero()) {
  out.zeros();
  return;
}
```

This code is used in section 123.

126.

⟨ quick multiplication if dimension is 1 126 ⟩ ≡

```
if (dimen() ≡ 1) {
  if (matlist[0])  /* always true */
    out.mult(in, ConstTwoDMatrix(*(matlist[0])));
  return;
}
```

This code is used in section 123.

127. Here we have to construct $A_1 \otimes I$, allocate intermediate result *last*, and perform the multiplication.

⟨ perform first multiplication AI 127 ⟩ ≡

```
if (matlist[0]) {
  KronProdAI akronid(*this);
  c = akronid.kpd.ncols();
  last = new TwoDMatrix(in.nrows(), c);
  akronid.mult(in, *last);
}
else {
  last = new TwoDMatrix(in.nrows(), in.ncols(), in.getData().base());
}
```

This code is used in section 123.

128. Here we go through all $I\otimes A_i\otimes I$, construct the product, allocate new storage for result *newlast*, perform the multiplication, deallocate old *last*, and set *last* to *newlast*.

⟨ perform intermediate multiplications IAI 128 ⟩ ≡

```
for (int i = 1; i < dimen() - 1; i++) {
  if (matlist[i]) {
    KronProdIAI interkron(*this, i);
    c = interkron.kpd.ncols();
    TwoDMatrix *newlast = new TwoDMatrix(in.nrows(), c);
    interkron.mult(*last, *newlast);
    delete last;
    last = newlast;
  }
}
```

This code is used in section 123.

129. Here just construct $I\otimes A_n$ and perform multiplication and deallocate *last*.

⟨ perform last multiplication IA 129 ⟩ ≡

```
if (matlist[dimen() - 1]) {
  KronProdIA idkrona(*this);
  idkrona.mult(*last, out);
}
else {
  out = *last;
}
delete last;
```

This code is used in section 123.

130. This calculates a Kornecker product of rows of matrices, the row indices are given by the integer sequence. The result is allocated and returned. The caller is repsonsible for its deallocation.

⟨ KronProdAll::multRows code 130 ⟩ ≡

```
Vector *KronProdAll::multRows(const IntSequence &irows) const
{
  TL_RAISE_IF(irows.size() ≠ dimen(),
      "Wrong␣length␣of␣row␣indices␣in␣KronProdAll::multRows");
  Vector *last = Λ;
  ConstVector *row;
  vector⟨Vector *⟩ to_delete;
  for (int i = 0; i < dimen(); i++) {
    int j = dimen() - 1 - i;
    ⟨ set row to the row of j-th matrix 131 ⟩;
    ⟨ set last to product of row and last 132 ⟩;
    delete row;
  }
  for (unsigned int i = 0; i < to_delete.size(); i++) delete to_delete[i];
  return last;
}
```

This code is used in section 109.

131. If the j-th matrix is real matrix, then the row is constructed from the matrix. It the matrix is unit, we construct a new vector, fill it with zeros, than set the unit to appropriate place, and make the *row* as ConstVector of this vector, which sheduled for deallocation.

⟨ set *row* to the row of j-th matrix 131 ⟩ ≡

```
if (matlist[j]) row = new ConstVector(irows[j], *(matlist[j]));
else {
  Vector *aux = new Vector(ncols(j));
  aux->zeros();
  (*aux)[irows[j]] = 1.0;
  to_delete.push_back(aux);
  row = new ConstVector(*aux);
}
```

This code is used in section 130.

132. If the *last* is exists, we allocate new storage, Kronecker multiply, deallocate the old storage. If the *last* does not exist, then we only make *last* equal to *row*.

⟨ set *last* to product of *row* and *last* 132 ⟩ ≡

```
if (last) {
  Vector *newlast;
  newlast = new Vector(last->length() * row->length());
  kronMult(*row, ConstVector(*last), *newlast);
  delete last;
  last = newlast;
}
else {
  last = new Vector(*row);
}
```

This code is used in section 130.

133. This permutes the matrices so that the new ordering would minimize memory consumption. As shown in ⟨ **KronProdAllOptim** class declaration 104 ⟩, we want $\frac{m_k}{n_k} \leq \frac{m_{k-1}}{n_{k-1}} \ldots \leq \frac{m_1}{n_1}$, where (m_i, n_i) is the dimension of A_i. So we implement the bubble sort.

⟨ **KronProdAllOptim** :: *optimizeOrder* code 133 ⟩ ≡

```
void KronProdAllOptim::optimizeOrder()
{
  for (int i = 0; i < dimen(); i++) {
    int swaps = 0;
    for (int j = 0; j < dimen() - 1; j++) {
      if (((double) kpd.rows[j])/kpd.cols[j] < ((double) kpd.rows[j + 1])/kpd.cols[j + 1]) {
        ⟨ swap dimensions and matrices at j and j + 1 134 ⟩;
        ⟨ project the swap to the permutation oper 135 ⟩;
      }
    }
    if (swaps == 0) {
      return;
    }
  }
}
```

This code is used in section 109.

134.

⟨ swap dimensions and matrices at j and $j+1$ 134 ⟩ ≡

```
int s = kpd.rows[j + 1];
kpd.rows[j + 1] = kpd.rows[j];
kpd.rows[j] = s;
s = kpd.cols[j + 1];
kpd.cols[j + 1] = kpd.cols[j];
kpd.cols[j] = s;
const TwoDMatrix *m = matlist[j + 1];
matlist[j + 1] = matlist[j];
matlist[j] = m;
```

This code is used in section 133.

135.

⟨ project the swap to the permutation *oper* 135 ⟩ ≡

```
s = oper.getMap()[j + 1];
oper.getMap()[j + 1] = oper.getMap()[j];
oper.getMap()[j] = s;
swaps++;
```

This code is used in section 133.

136. End of `kron_prod.cpp` file.

137. Combinatorics.

138. Symmetry. This is `symmetry.h` file

Symmetry is an abstraction for a term of the form y^3u^2. It manages only indices, not the variable names. So if one uses this abstraction, he must keep in mind that y is the first, and u is the second.

In fact, the symmetry is a special case of equivalence, but its implementation is much simpler. We do not need an abstraction for the term $yyuyu$ but due to Green theorem we can have term y^3u^2. That is why the equivalence is too general for our purposes.

One of a main purposes of the tensor library is to calculate something like:

$$\left[B_{y^2u^3}\right]_{\alpha_1\alpha_2\beta_1\beta_2\beta_3} = \left[g_{y^l}\right]_{\gamma_1\ldots\gamma_l}\left(\sum_{c\in M_{l,5}}\prod_{m=1}^{l}\left[g_{c_m}\right]^{\gamma_m}_{c_m(\alpha,\beta)}\right)$$

If, for instance, $l=3$, and $c=\{\{0,4\},\{1,2\},\{3\}\}$, then we have to calculate

$$\left[g_{y^3}\right]_{\gamma_1\gamma_2\gamma_3}\left[g_{yu}\right]^{\gamma_1}_{\alpha_1\beta_3}\left[g_{yu}\right]^{\gamma_2}_{\alpha_2\beta_1}\left[g_u\right]^{\gamma_3}_{\beta_2}$$

We must be able to calculate a symmetry induced by symmetry y^2u^3 and by an equivalence class from equivalence c. For equivalence class $\{0,4\}$ the induced symmetry is yu, since we pick first and fifth variable from y^2u^3. For a given outer symmetry, the class **InducedSymmetries** does this for all classes of a given equivalence.

We need also to cycle through all possible symmetries yielding the given dimension. For this purpose we define classes **SymmetrySet** and **symiterator**.

The symmetry is implemented as **IntSequence**, in fact, it inherits from it.

```
#ifndef SYMMETRY_H
#define SYMMETRY_H
#include "equivalence.h"
#include "int_sequence.h"
#include <list>
#include <vector>
  ⟨ Symmetry class declaration 139 ⟩;
  ⟨ SymmetrySet class declaration 141 ⟩;
  ⟨ symiterator class declaration 142 ⟩;
  ⟨ InducedSymmetries class declaration 143 ⟩;
#endif
```

139. Clear. The method *isFull* returns true if and only if the symmetry allows for any permutation of indices.

⟨ **Symmetry** class declaration 139 ⟩ ≡
```
class Symmetry : public IntSequence {
public:
  ⟨ Symmetry constructors 140 ⟩;
  int num() const
  { return size(); }
  int dimen() const
  { return sum(); }
  int findClass(int i) const;
  bool isFull() const;
};
```
This code is cited in section 1.

This code is used in section 138.

140. We provide three constructors for symmetries of the form y^n, $y^n u^m$, $y^n u^m \sigma^k$. Also a copy constructor, and finally a constructor of implied symmetry for a symmetry and an equivalence class. It is already implemented in **IntSequence** so we only call appropriate constructor of **IntSequence**. We also provide the subsymmetry, which takes the given length of symmetry from the end.

The last constructor constructs a symmetry from an integer sequence (supposed to be ordered) as a symmetry counting successively equal items. For instance the sequence $(a, a, a, b, c, c, d, d, d, d)$ produces symmetry $(3, 1, 2, 4)$.

⟨ **Symmetry** constructors 140 ⟩ ≡
```
Symmetry(int len, const char *dummy)
: IntSequence(len, 0) { }
Symmetry(int i1)
: IntSequence(1, i1) { }
Symmetry(int i1, int i2)
: IntSequence(2) { operator[](0) = i1; operator[](1) = i2; }
Symmetry(int i1, int i2, int i3)
: IntSequence(3) { operator[](0) = i1; operator[](1) = i2; operator[](2) = i3; }
Symmetry(int i1, int i2, int i3, int i4)
: IntSequence(4) { operator[](0) = i1; operator[](1) = i2; operator[](2) = i3;
  operator[](3) = i4; }
Symmetry(const Symmetry &s)
: IntSequence(s) { }
Symmetry(const Symmetry &s, const OrdSequence &cl)
: IntSequence(s, cl.getData()) { }
Symmetry(Symmetry &s, int len)
: IntSequence(s, s.size() - len, s.size()) { }
Symmetry(const IntSequence &s);
```
This code is used in section 139.

141. The class **SymmetrySet** defines a set of symmetries of the given length having given dimension. It does not store all the symmetries, rather it provides a storage for one symmetry, which is changed as an adjoint iterator moves.

The iterator class is **symiterator**. It is implemented recursively. The iterator object, when created, creates subordinal iterator, which iterates over a symmetry set whose length is one less, and dimension is the former dimension. When the subordinal iterator goes to its end, the superordinal iterator increases left most index in the symmetry, resets the subordinal symmetry set with different dimension, and iterates through the subordinal symmetry set until its end, and so on. That's why we provide also **SymmetrySet** constructor for construction of a subordinal symmetry set.

The typical usage of the abstractions for **SymmetrySet** and **symiterator** is as follows:

$$\textbf{for}\ (\textbf{symiterator}\ si(\textbf{SymmetrySet}(6,4));\ \neg si.isEnd(\,);\ {+}{+}si)\ \{body\}$$

It goes through all symmetries of size 4 having dimension 6. One can use $*si$ as the symmetry in the body.

⟨ **SymmetrySet** class declaration 141 ⟩ ≡

```
class SymmetrySet {
  Symmetry run;
  int dim;
public:
  SymmetrySet(int d, int length)
  : run(length, ""), dim(d) { }
  SymmetrySet(SymmetrySet &s, int d)
  : run(s.run, s.size() - 1), dim(d) { }
  int dimen() const
  { return dim; }
  const Symmetry &sym() const
  { return run; }
  Symmetry &sym()
  { return run; }
  int size() const
  { return run.size(); }
};
```

This code is cited in section 142.

This code is used in section 138.

142. The logic of **symiterator** was described in ⟨ **SymmetrySet** class declaration 141 ⟩. Here we only comment that: the class has a reference to the **SymmetrySet** only to know dimension and for access of its symmetry storage. Further we have pointers to subordinal **symiterator** and its **SymmetrySet**. These are pointers, since the recursion ends at length equal to 2, in which case these pointers are Λ.

The constructor creates the iterator which initializes to the first symmetry (beginning).

⟨ **symiterator** class declaration 142 ⟩ ≡

```
class symiterator {
  SymmetrySet &s;
  symiterator *subit;
  SymmetrySet *subs;
  bool end_flag;
public:
  symiterator(SymmetrySet &ss);
  ~symiterator();
  symiterator &operator++();
  bool isEnd() const
  { return end_flag; }
  const Symmetry &operator*() const
  { return s.sym(); }
};
```

This code is used in section 138.

143. This simple abstraction just constructs a vector of induced symmetries from the given equivalence and outer symmetry. A permutation might optionally permute the classes of the equivalence.

⟨ **InducedSymmetries** class declaration 143 ⟩ ≡

```
class InducedSymmetries : public vector<Symmetry> {
public:
  InducedSymmetries(const Equivalence &e, const Symmetry &s);
  InducedSymmetries(const Equivalence &e, const Permutation &p, const Symmetry &s);
  void print() const;
};
```

This code is used in section 138.

144. End of `symmetry.h` file.

145. Start of `symmetry.cpp` file.

```
#include "symmetry.h"
#include "permutation.h"
#include <cstdio>
  ⟨ Symmetry constructor code 146 ⟩;
  ⟨ Symmetry::findClass code 147 ⟩;
  ⟨ Symmetry::isFull code 148 ⟩;
  ⟨ symiterator constructor code 149 ⟩;
  ⟨ symiterator destructor code 150 ⟩;
  ⟨ symiterator::operator++ code 151 ⟩;
  ⟨ InducedSymmetries constructor code 152 ⟩;
  ⟨ InducedSymmetries permuted constructor code 153 ⟩;
  ⟨ InducedSymmetries::print code 154 ⟩;
```

146. Construct symmetry as numbers of successively equal items in the sequence.

⟨ **Symmetry** constructor code 146 ⟩ ≡
```
Symmetry::Symmetry(const IntSequence &s)
  : IntSequence(s.getNumDistinct(), 0) {
    int p = 0;
    if (s.size() > 0) operator[](p) = 1;
    for (int i = 1; i < s.size(); i++) {
      if (s[i] ≠ s[i − 1]) p++;
      operator[](p)++;
    }
  }
```
This code is used in section 145.

147. Find a class of the symmetry containing a given index.

⟨ **Symmetry** :: *findClass* code 147 ⟩ ≡
```
int Symmetry::findClass(int i) const
{
  int j = 0;
  int sum = 0;
  do {
    sum += operator[](j);
    j++;
  } while (j < size() ∧ sum ≤ i);
  return j − 1;
}
```
This code is used in section 145.

148. The symmetry is full if it allows for any permutation of indices. It means, that there is at most one non-zero index.

⟨ **Symmetry** :: *isFull* code 148 ⟩ ≡
```
bool Symmetry::isFull() const
{
  int count = 0;
  for (int i = 0; i < num(); i++)
    if (operator[](i) ≠ 0) count++;
  return count ≤ 1;
}
```
This code is used in section 145.

149. Here we construct the beginning of the **symiterator**. The first symmetry index is 0. If length is 2, the second index is the dimension, otherwise we create the subordinal symmetry set and its beginning as subordinal **symiterator**.

⟨ **symiterator** constructor code 149 ⟩ ≡

```
symiterator::symiterator(SymmetrySet &ss)
: s(ss), subit(Λ), subs(Λ), end_flag(false) {
  s.sym()[0] = 0;
  if (s.size() ≡ 2) {
    s.sym()[1] = s.dimen();
  }
  else {
    subs = new SymmetrySet(s, s.dimen());
    subit = new symiterator(*subs);
  }
}
```

This code is used in section 145.

150.

⟨ **symiterator** destructor code 150 ⟩ ≡

```
symiterator::~symiterator()
{
  if (subit) delete subit;
  if (subs) delete subs;
}
```

This code is used in section 145.

151. Here we move to the next symmetry. We do so only, if we are not at the end. If length is 2, we increase lower index and decrease upper index, otherwise we increase the subordinal symmetry. If we got to the end, we recreate the subordinal symmetry set and set the subordinal iterator to the beginning. At the end we test, if we are not at the end. This is recognized if the lowest index exceeded the dimension.

⟨ **symiterator** :: **operator** ++ code 151 ⟩ ≡

```
symiterator &symiterator::operator++()
{
  if (¬end_flag) {
    if (s.size() ≡ 2) {
      s.sym()[0]++;
      s.sym()[1]−−;
    }
    else {
      ++(∗subit);
      if (subit→isEnd()) {
        delete subit;
        delete subs;
        s.sym()[0]++;
        subs = new SymmetrySet(s, s.dimen() − s.sym()[0]);
        subit = new symiterator(∗subs);
      }
    }
    if (s.sym()[0] ≡ s.dimen() + 1) end_flag = true;
  }
  return ∗this;
}
```

This code is used in section 145.

152.

⟨ **InducedSymmetries** constructor code 152 ⟩ ≡

```
InducedSymmetries::InducedSymmetries(const Equivalence &e, const Symmetry &s)
{
  for (Equivalence::const_seqit i = e.begin(); i ≠ e.end(); ++i) {
    push_back(Symmetry(s, ∗i));
  }
}
```

This code is used in section 145.

153.

⟨ **InducedSymmetries** permuted constructor code 153 ⟩ ≡

```
InducedSymmetries::InducedSymmetries(const Equivalence &e, const Permutation &p, const
        Symmetry &s)
{
  for (int i = 0; i < e.numClasses(); i++) {
    Equivalence::const_seqit it = e.find(p.getMap()[i]);
    push_back(Symmetry(s, ∗it));
  }
}
```

This code is used in section 145.

154. Debug print.

⟨ **InducedSymmetries** :: *print* code 154 ⟩ ≡

```
void InducedSymmetries::print() const
{
  printf("Induced␣symmetries:␣%lu\n", (unsigned long) size());
  for (unsigned int i = 0; i < size(); i++) operator[](i).print();
}
```

This code is used in section 145.

155. End of `symmetry.cpp` file.

156. Equivalences. Start of `equivalence.h` file

Here we define an equivalence of a set of integers $\{0,1,\ldots,k-1\}$. The purpose is clear, in the tensor library we often iterate through all equivalences and sum matrices. We need an abstraction for an equivalence class, equivalence and a set of all equivalences.

The equivalence class (which is basically a set of integers) is here implemented as ordered integer sequence. The ordered sequence is not implemented via **IntSequence**, but via **vector**⟨**int**⟩ since we need insertions. The equivalence is implemented as an ordered list of equivalence classes, and equivalence set is a list of equivalences.

The ordering of the equivalence classes within an equivalence is very important. For instance, if we iterate through equivalences for $k=5$ and pickup some equivalence class, say $\{\{0,4\},\{1,2\},\{3\}\}$, we then evaluate something like:

$$\left[B_{y^2u^3}\right]_{\alpha_1\alpha_2\beta_1\beta_2\beta_3}=\cdots+\left[g_{y^3}\right]_{\gamma_1\gamma_2\gamma_3}\left[g_{yu}\right]^{\gamma_1}_{\alpha_1\beta_3}\left[g_{yu}\right]^{\gamma_2}_{\alpha_2\beta_1}\left[g_u\right]^{\gamma_3}_{\beta_2}+\cdots$$

If the tensors are unfolded, we can evaluate this expression as

$$g_{y^3}\cdot(g_{yu}\otimes g_{yu}\otimes g_u)\cdot P,$$

where P is a suitable permutation of columns of the expressions, which permutes them so that the index $(\alpha_1,\beta_3,\alpha_2,\beta_1,\beta_2)$ would go to $(\alpha_1,\alpha_2,\beta_1,\beta_2,\beta_3)$. The permutation P can be very ineffective (copying great amount of small chunks of data) if the equivalence class ordering is chosen badly. However, we do not provide any heuristic minimizing a total time spent in all permutations. We choose an ordering which orders the classes according to their averages, and according to the smallest equivalence class element if the averages are the same.

```
#ifndef EQUIVALENCE_H
#define EQUIVALENCE_H
#include "int_sequence.h"
#include <vector>
#include <list>
  using namespace std;
  ⟨ OrdSequence class declaration 157 ⟩;
  ⟨ Equivalence class declaration 158 ⟩;
  ⟨ EquivalenceSet class declaration 159 ⟩;
  ⟨ EquivalenceBundle class declaration 160 ⟩;
#endif
```

157. Here is the abstraction for an equivalence class. We implement it as **vector**⟨**int**⟩. We have a constructor for empty class, copy constructor. What is important here is the ordering operator **operator** < and methods for addition of an integer, and addition of another sequence. Also we provide method *has* which returns true if a given integer is contained.

⟨ **OrdSequence** class declaration 157 ⟩ ≡

```
  class OrdSequence { vector⟨int⟩ data;
public:
  OrdSequence( )
  : data( ) { }
  OrdSequence(const OrdSequence &s)
  : data(s.data) { }
  const OrdSequence &operator=(const OrdSequence &s)
  { data = s.data; return ∗this; }
  bool operator≡(const OrdSequence &s) const;
  int operator[ ](int i) const; bool operator < (const OrdSequence &s) const;
  const vector⟨int⟩ &getData( ) const
  { return data; }
  int length( ) const
  { return data.size( ); }
  void add(int i);
  void add(const OrdSequence &s);
  bool has(int i) const;
  void print(const char ∗prefix) const;
private:
  double average( ) const; } ;
```

This code is used in section 156.

158. Here is the abstraction for the equivalence. It is a list of equivalence classes. Also we remember n, which is a size of underlying set $\{0, 1, \ldots, n-1\}$.

Method *trace* "prints" the equivalence into the integer sequence.

⟨ **Equivalence** class declaration 158 ⟩ ≡

```
class Permutation;
class Equivalence {
private: int n;
  list⟨OrdSequence⟩ classes;
public:
  typedef list⟨OrdSequence⟩::const_iterator const_seqit;
  typedef list⟨OrdSequence⟩::iterator seqit;
  ⟨ Equivalence constructors 161 ⟩;
  const Equivalence &operator=(const Equivalence &e);
  bool operator≡(const Equivalence &e) const;
  bool operator≠(const Equivalence &e) const
  { return ¬operator≡(e); }
  int getN( ) const
  { return n; }
  int numClasses( ) const
  { return classes.size( ); }
  void trace(IntSequence &out, int n) const;
  void trace(IntSequence &out) const
  { trace(out, numClasses( )); }
  void trace(IntSequence &out, const Permutation &per) const;
  void print(const char *prefix) const;
  ⟨ Equivalence begin and end methods 162 ⟩;
  const_seqit find(int i) const;
  seqit find(int i);
protected:
  ⟨ Equivalence protected methods 163 ⟩;
};
```

This code is cited in section 1.

This code is used in section 156.

159. The **EquivalenceSet** is a list of equivalences. The unique constructor constructs a set of all equivalences over n-element set. The equivalences are sorted in the list so that equivalences with fewer number of classes are in the end.

The two methods *has* and *addParents* are useful in the constructor.

⟨ **EquivalenceSet** class declaration 159 ⟩ ≡

```
class EquivalenceSet {
  int n;
  list<Equivalence> equis;
public:
  typedef list<Equivalence>::const_iterator const_iterator;
  EquivalenceSet(int num);
  void print(const char *prefix) const;
  const_iterator begin() const
  { return equis.begin(); }
  const_iterator end() const
  { return equis.end(); }
private:
  bool has(const Equivalence &e) const;
  void addParents(const Equivalence &e, list<Equivalence> &added);
};
```

This code is cited in section 1.

This code is used in section 156.

160. The equivalence bundle class only encapsulates **EquivalenceSet**s from 1 up to a given number. It is able to retrieve the equivalence set over n-element set for a given n, and also it can generate some more sets on request.

It is fully responsible for storage needed for **EquivalenceSet**s.

⟨ **EquivalenceBundle** class declaration 160 ⟩ ≡

```
class EquivalenceBundle {
  vector<EquivalenceSet *> bundle;
public:
  EquivalenceBundle(int nmax);
  ~EquivalenceBundle();
  const EquivalenceSet &get(int n) const;
  void generateUpTo(int nmax);
};
```

This code is used in section 156.

161. The first constructor constructs $\{\{0\},\{1\},\ldots,\{n-1\}\}$.

The second constructor constructs $\{\{0,1,\ldots,n-1\}\}$.

The third is the copy constructor. And the fourth is the copy constructor plus gluing *i1* and *i2* in one class.

⟨ **Equivalence** constructors 161 ⟩ ≡

```
Equivalence(int num);
Equivalence(int num, const char *dummy);
Equivalence(const Equivalence &e);
Equivalence(const Equivalence &e, int i1, int i2);
```

See also section 177.

This code is used in sections 158 and 167.

162.

⟨ **Equivalence** begin and end methods 162 ⟩ ≡

```
seqit begin()
{ return classes.begin(); }
const_seqit begin() const
{ return classes.begin(); }
seqit end()
{ return classes.end(); }
const_seqit end() const
{ return classes.end(); }
```

This code is used in section 158.

163. Here we have find methods. We can find an equivalence class having a given number or we can find an equivalence class of a given index within the ordering.

We have also an *insert* method which inserts a given class according to the class ordering.

⟨ **Equivalence** protected methods 163 ⟩ ≡

```
const_seqit findHaving(int i) const;
seqit findHaving(int i);
void insert(const OrdSequence &s);
```

This code is used in section 158.

164. End of `equivalence.h` file.

165. Start of `equivalence.cpp` file.

```
#include "equivalence.h"
#include "permutation.h"
#include "tl_exception.h"
#include <cstring>
  ⟨ OrdSequence method codes 166 ⟩;
  ⟨ Equivalence method codes 167 ⟩;
  ⟨ EquivalenceSet method codes 168 ⟩;
  ⟨ EquivalenceBundle method codes 169 ⟩;
```

166.

⟨ **OrdSequence** method codes 166 ⟩ ≡

```
  ⟨ OrdSequence::operator[] code 170 ⟩;
  ⟨ OrdSequence::operator < code 171 ⟩;
  ⟨ OrdSequence::operator≡ code 172 ⟩;
  ⟨ OrdSequence::add codes 173 ⟩;
  ⟨ OrdSequence::has code 174 ⟩;
  ⟨ OrdSequence::average() code 175 ⟩;
  ⟨ OrdSequence::print code 176 ⟩;
```

This code is used in section 165.

167.

⟨ **Equivalence** method codes 167 ⟩ ≡
  ⟨ **Equivalence** constructors 161 ⟩;
  ⟨ **Equivalence** copy constructors 178 ⟩;
  ⟨ **Equivalence** :: *findHaving* codes 181 ⟩;
  ⟨ **Equivalence** :: *find* codes 182 ⟩;
  ⟨ **Equivalence** :: *insert* code 183 ⟩;
  ⟨ **Equivalence** :: **operator**= code 179 ⟩;
  ⟨ **Equivalence** :: **operator**≡ code 180 ⟩;
  ⟨ **Equivalence** :: *trace* code 184 ⟩;
  ⟨ **Equivalence** :: *trace* permuted code 185 ⟩;
  ⟨ **Equivalence** :: *print* code 186 ⟩;

This code is used in section 165.

168.

⟨ **EquivalenceSet** method codes 168 ⟩ ≡
  ⟨ **EquivalenceSet** constructor code 187 ⟩;
  ⟨ **EquivalenceSet** :: *has* code 188 ⟩;
  ⟨ **EquivalenceSet** :: *addParents* code 189 ⟩;
  ⟨ **EquivalenceSet** :: *print* code 190 ⟩;

This code is used in section 165.

169.

⟨ **EquivalenceBundle** method codes 169 ⟩ ≡
  ⟨ **EquivalenceBundle** constructor code 191 ⟩;
  ⟨ **EquivalenceBundle** destructor code 192 ⟩;
  ⟨ **EquivalenceBundle** :: *get* code 193 ⟩;
  ⟨ **EquivalenceBundle** :: *generateUpTo* code 194 ⟩;

This code is used in section 165.

170.

⟨ **OrdSequence** :: **operator**[ ] code 170 ⟩ ≡

```
int OrdSequence::operator[](int i) const
{
  TL_RAISE_IF((i < 0 ∨ i ≥ length()), "Index␣out␣of␣range␣in␣OrdSequence::operator[]");
  return data[i];
}
```

This code is used in section 166.

171. Here we implement the ordering. It can be changed, or various orderings can be used for different problem sizes. We order them according to the average, and then according to the first item.

⟨ **OrdSequence** :: **operator** < code 171 ⟩ ≡

```
bool OrdSequence::operator < (const OrdSequence &s) const
{
  double ta = average();
  double sa = s.average();
  return (ta < sa ∨ ((ta ≡ sa) ∧ (operator[](0) > s[0])));
}
```

This code is used in section 166.

172.

⟨ **OrdSequence** :: **operator**≡ code 172 ⟩ ≡

```
bool OrdSequence::operator==(const OrdSequence &s) const
{
  if (length() != s.length()) return false;
  int i = 0;
  while (i < length() && operator[](i) == s[i]) i++;
  return (i == length());
}
```

This code is used in section 166.

173. The first *add* adds a given integer to the class, the second iterates through a given sequence and adds everything found in the given class.

⟨ **OrdSequence** :: *add* codes 173 ⟩ ≡

```
void OrdSequence::add(int i)
{
  vector<int>::iterator vit = data.begin();
  while (vit != data.end() && *vit < i) ++vit;
  if (vit != data.end() && *vit == i) return;
  data.insert(vit, i);
}
void OrdSequence::add(const OrdSequence &s)
{
  vector<int>::const_iterator vit = s.data.begin();
  while (vit != s.data.end()) {
    add(*vit);
    ++vit;
  }
}
```

This code is used in section 166.

174. Answers *true* if a given number is in the class.

⟨ **OrdSequence** :: *has* code 174 ⟩ ≡

```
bool OrdSequence::has(int i) const
{
  vector<int>::const_iterator vit = data.begin();
  while (vit != data.end()) {
    if (*vit == i) return true;
    ++vit;
  }
  return false;
}
```

This code is used in section 166.

175. Return an average of the class.

```
⟨ OrdSequence::average() code 175 ⟩ ≡
  double OrdSequence::average() const
  {
    double res = 0;
    for (unsigned int i = 0; i < data.size(); i++) res += data[i];
    TL_RAISE_IF(data.size() ≡ 0,
        "Attempt␣to␣take␣average␣of␣empty␣class␣in␣OrdSequence::average");
    return res/data.size();
  }
```

This code is used in section 166.

176. Debug print.

```
⟨ OrdSequence::print code 176 ⟩ ≡
  void OrdSequence::print(const char *prefix) const
  {
    printf("%s", prefix);
    for (unsigned int i = 0; i < data.size(); i++) printf("%d␣", data[i]);
    printf("\n");
  }
```

This code is used in section 166.

177.

```
⟨ Equivalence constructors 161 ⟩ +≡
  Equivalence::Equivalence(int num)
  : n(num) {
    for (int i = 0; i < num; i++) {
      OrdSequence s;
      s.add(i);
      classes.push_back(s);
    }
  }
  Equivalence::Equivalence(int num, const char *dummy)
  : n(num) {
    OrdSequence s;
    for (int i = 0; i < num; i++) s.add(i);
    classes.push_back(s);
  }
```

178. Copy constructors. The second also glues a given couple.

```
⟨ Equivalence copy constructors 178 ⟩ ≡
  Equivalence::Equivalence(const Equivalence &e)
  : n(e.n), classes(e.classes) { }
  Equivalence::Equivalence(const Equivalence &e, int i1, int i2)
  : n(e.n), classes(e.classes) {
    seqit s1 = find(i1);
    seqit s2 = find(i2);
    if (s1 ≠ s2) {
      OrdSequence ns(*s1);
      ns.add(*s2);
      classes.erase(s1);
      classes.erase(s2);
      insert(ns);
    }
  }
```

This code is used in section 167.

179.

```
⟨ Equivalence::operator= code 179 ⟩ ≡
  const Equivalence &Equivalence::operator=(const Equivalence &e)
  {
    classes.clear();
    n = e.n;
    classes = e.classes;
    return *this;
  }
```

This code is used in section 167.

180.

```
⟨ Equivalence::operator≡ code 180 ⟩ ≡
  bool Equivalence::operator≡(const Equivalence &e) const
  {
    if (¬std::operator≡(classes, e.classes)) return false;
    if (n ≠ e.n) return false;
    return true;
  }
```

This code is used in section 167.

181. Return an iterator pointing to a class having a given integer.

⟨ **Equivalence** :: *findHaving* codes 181 ⟩ ≡

```
Equivalence::const_seqit Equivalence::findHaving(int i) const
{
  const_seqit si = classes.begin();
  while (si ≠ classes.end()) {
    if ((*si).has(i)) return si;
    ++si;
  }
  TL_RAISE_IF(si ≡ classes.end(),
      "Couldn't␣find␣equivalence␣class␣in␣Equivalence::findHaving");
  return si;
}
Equivalence::seqit Equivalence::findHaving(int i)
{
  seqit si = classes.begin();
  while (si ≠ classes.end()) {
    if ((*si).has(i)) return si;
    ++si;
  }
  TL_RAISE_IF(si ≡ classes.end(),
      "Couldn't␣find␣equivalence␣class␣in␣Equivalence::findHaving");
  return si;
}
```

This code is used in section 167.

182. Find j-th class for a given j.

⟨ **Equivalence** :: *find* codes 182 ⟩ ≡

```
Equivalence::const_seqit Equivalence::find(int j) const
{
  const_seqit si = classes.begin();
  int i = 0;
  while (si ≠ classes.end() ∧ i < j) {
    ++si;
    i++;
  }
  TL_RAISE_IF(si ≡ classes.end(), "Couldn't find equivalence class in Equivalence::find");
  return si;
}
Equivalence::seqit Equivalence::find(int j)
{
  seqit si = classes.begin();
  int i = 0;
  while (si ≠ classes.end() ∧ i < j) {
    ++si;
    i++;
  }
  TL_RAISE_IF(si ≡ classes.end(), "Couldn't find equivalence class in Equivalence::find");
  return si;
}
```

This code is used in section 167.

183. Insert a new class yielding the ordering.

⟨ **Equivalence** :: *insert* code 183 ⟩ ≡

```
void Equivalence::insert(const OrdSequence &s)
{
  seqit si = classes.begin();
  while (si ≠ classes.end() ∧ *si < s) ++si;
  classes.insert(si, s);
}
```

This code is used in section 167.

184. Trace the equivalence into the integer sequence. The classes are in some order (described earlier), and items within classes are ordered, so this implies, that the data can be linearized. This method "prints" them to the sequence. We allow for tracing only a given number of classes from the beginning.

⟨ **Equivalence**::*trace* code 184 ⟩ ≡

```
void Equivalence::trace(IntSequence &out, int num) const
{
  int i = 0;
  int nc = 0;
  for (const_seqit it = begin(); it ≠ end() ∧ nc < num; ++it, ++nc)
    for (int j = 0; j < (*it).length(); j++, i++) {
      TL_RAISE_IF(i ≥ out.size(), "Wrong size of output sequence in Equivalence::trace");
      out[i] = (*it)[j];
    }
}
```

This code is used in section 167.

185.

⟨ **Equivalence**::*trace* permuted code 185 ⟩ ≡

```
void Equivalence::trace(IntSequence &out, const Permutation &per) const
{
  TL_RAISE_IF(out.size() ≠ n, "Wrong size of output sequence in Equivalence::trace");
  TL_RAISE_IF(per.size() ≠ numClasses(),
      "Wrong permutation for permuted Equivalence::trace");
  int i = 0;
  for (int iclass = 0; iclass < numClasses(); iclass++) {
    const_seqit itper = find(per.getMap()[iclass]);
    for (int j = 0; j < (*itper).length(); j++, i++) out[i] = (*itper)[j];
  }
}
```

This code is used in section 167.

186. Debug print.

⟨ **Equivalence**::*print* code 186 ⟩ ≡

```
void Equivalence::print(const char *prefix) const
{
  int i = 0;
  for (const_seqit it = classes.begin(); it ≠ classes.end(); ++it, i++) {
    printf("%sclass %d: ", prefix, i);
    (*it).print("");
  }
}
```

This code is used in section 167.

187. Here we construct a set of all equivalences over n-element set. The construction proceeds as follows. We maintain a list of added equivalences. At each iteration we pop front of the list, try to add all parents of the popped equivalence. This action adds new equivalences to the object and also to the added list. We finish the iterations when the added list is empty.

In the beginning we start with $\{\{0\},\{1\},\ldots,\{n-1\}\}$. Adding of parents is an action which for a given equivalence tries to glue all possible couples and checks whether a new equivalence is already in the equivalence set. This is not effective, but we will do the construction only ones.

In this way we breath-first search a lattice of all equivalences. Note that the lattice is modular, that is why the result of a construction is a list with a property that between two equivalences with the same number of classes there are only equivalences with that number of classes. Obviously, the list is decreasing in a number of classes (since it is constructed by gluing attempts).

⟨ **EquivalenceSet** constructor code 187 ⟩ ≡

```
EquivalenceSet::EquivalenceSet(int num)
: n(num), equis() {
  list⟨Equivalence⟩ added;
  Equivalence first(n);
  equis.push_back(first);
  addParents(first, added);
  while (¬added.empty()) {
    addParents(added.front(), added);
    added.pop_front();
  }
  if (n > 1) {
    Equivalence last(n, "");
    equis.push_back(last);
  }
}
```

This code is used in section 168.

188. This method is used in *addParents* and returns *true* if the object already has that equivalence. We trace list of equivalences in reverse order since equivalences are ordered in the list from the most primitive (nothing equivalent) to maximal (all is equivalent). Since we will have much more results of *has* method as *true*, and **operator**≡ between equivalences is quick if number of classes differ, and in time we will compare with equivalences with less classes, then it is more efficient to trace the equivalences from less classes to more classes. hence the reverse order.

⟨ **EquivalenceSet**::*has* code 188 ⟩ ≡

```
bool EquivalenceSet::has(const Equivalence &e) const
{
  list⟨Equivalence⟩::const_reverse_iterator rit = equis.rbegin();
  while (rit ≠ equis.rend() ∧ *rit ≠ e) ++rit;
  if (rit ≠ equis.rend()) return true;
  return false;
}
```

This code is used in section 168.

189. Responsibility of this methods is to try to glue all possible couples within a given equivalence and add those which are not in the list yet. These are added also to the *added* list.

If number of classes is 2 or 1, we exit, because there is nothing to be added.

⟨ **EquivalenceSet** :: *addParents* code 189 ⟩ ≡

```
void EquivalenceSet::addParents(const Equivalence &e, list<Equivalence> &added)
{
  if (e.numClasses() ≡ 2 ∨ e.numClasses() ≡ 1) return;
  for (int i1 = 0; i1 < e.numClasses(); i1++)
    for (int i2 = i1 + 1; i2 < e.numClasses(); i2++) {
      Equivalence ns(e, i1, i2);
      if (¬has(ns)) {
        added.push_back(ns);
        equis.push_back(ns);
      }
    }
}
```

This code is used in section 168.

190. Debug print.

⟨ **EquivalenceSet** :: *print* code 190 ⟩ ≡

```
void EquivalenceSet::print(const char *prefix) const
{
  char tmp[100];
  strcpy(tmp, prefix);
  strcat(tmp, "␣␣␣␣");
  int i = 0;
  for (list<Equivalence>::const_iterator it = equis.begin(); it ≠ equis.end(); ++it, i++) {
    printf("%sequivalence␣%d:(classes␣%d)\n", prefix, i, (*it).numClasses());
    (*it).print(tmp);
  }
}
```

This code is used in section 168.

191. Construct the bundle. *nmax* is a maximum size of underlying set.

⟨ **EquivalenceBundle** constructor code 191 ⟩ ≡

```
EquivalenceBundle::EquivalenceBundle(int nmax)
{
  nmax = max(nmax, 1);
  generateUpTo(nmax);
}
```

This code is used in section 169.

192. Destruct bundle. Just free all pointers.

⟨ **EquivalenceBundle** destructor code 192 ⟩ ≡

```
EquivalenceBundle::~EquivalenceBundle()
{
  for (unsigned int i = 0; i < bundle.size(); i++) delete bundle[i];
}
```

This code is used in section 169.

193. Remember, that the first item is **EquivalenceSet**(1).

⟨ **EquivalenceBundle** :: *get* code 193 ⟩ ≡

```
const EquivalenceSet &EquivalenceBundle::get(int n) const
{
  if (n > (int)(bundle.size()) ∨ n < 1) {
    TL_RAISE("Equivalence␣set␣not␣found␣in␣EquivalenceBundle::get");
    return *(bundle[0]);
  }
  else {
    return *(bundle[n − 1]);
  }
}
```

This code is used in section 169.

194. Get *curmax* which is a maximum size in the bundle, and generate for all sizes from $curmax + 1$ up to *nmax*.

⟨ **EquivalenceBundle** :: *generateUpTo* code 194 ⟩ ≡

```
void EquivalenceBundle::generateUpTo(int nmax)
{
  int curmax = bundle.size();
  for (int i = curmax + 1; i ≤ nmax; i++) bundle.push_back(new EquivalenceSet(i));
}
```

This code is used in section 169.

195. End of `equivalence.cpp` file.

196. Permutations. Start of `permutation.h` file.

The permutation class is useful when describing a permutation of indices in permuted symmetry tensor. This tensor comes to existence, for instance, as a result of the following tensor multiplication:

$$\left[g_{y^3}\right]_{\gamma_1\gamma_2\gamma_3}\left[g_{yu}\right]^{\gamma_1}_{\alpha_1\beta_3}\left[g_{yu}\right]^{\gamma_2}_{\alpha_2\beta_1}\left[g_u\right]^{\gamma_3}_{\beta_2}$$

If this operation is done by a Kronecker product of unfolded tensors, the resulting tensor has permuted indices. So, in this case the permutation is implied by the equivalence: $\{\{0,4\},\{1,3\},\{2\}\}$. This results in a permutation which maps indices $(0,1,2,3,4)\mapsto(0,2,4,3,1)$.

The other application of **Permutation** class is to permute indices with the same permutation as done during sorting.

Here we only define an abstraction for the permutation defined by an equivalence. Its basic operation is to apply the permutation to the integer sequence. The application is right (or inner), in sense that it works on indices of the sequence not items of the sequence. More formally $s\circ m\neq m\circ s$. In here, the application of the permutation defined by map m is $s\circ m$.

Also, we need **PermutationSet** class which contains all permutations of n element set, and a bundle of permutations **PermutationBundle** which contains all permutation sets up to a given number.

```
#ifndef PERMUTATION_H
#define PERMUTATION_H
#include "int_sequence.h"
#include "equivalence.h"
#include <vector>
  ⟨ Permutation class declaration 197 ⟩;
  ⟨ PermutationSet class declaration 198 ⟩;
  ⟨ PermutationBundle class declaration 199 ⟩;
#endif
```

197. The permutation object will have a map, which defines mapping of indices $(0, 1, \ldots, n-1) \mapsto (m_0, m_1, \ldots, m_{n-1})$. The map is the sequence $(m_0, m_1, \ldots, m_{n-1}$. When the permutation with the map m is applied on sequence s, it permutes its indices: $s \circ \mathrm{id} \mapsto s \circ m$.

So we have one constructor from equivalence, then a method *apply*, and finally a method *tailIdentity* which returns a number of trailing indices which yield identity. Also we have a constructor calculating map, which corresponds to permutation in sort. This is, we want (sorted s) $\circ\, m = s$.

⟨ **Permutation** class declaration 197 ⟩ ≡

```
class Permutation {
protected:
  IntSequence permap;
public:
  Permutation(int len)
  : permap(len) { for (int i = 0; i < len; i++) permap[i] = i; }
  Permutation(const Equivalence &e)
  : permap(e.getN()) { e.trace(permap); }
  Permutation(const Equivalence &e, const Permutation &per)
  : permap(e.getN()) { e.trace(permap, per); }
  Permutation(const IntSequence &s)
  : permap(s.size()) { computeSortingMap(s); }
  ;
  Permutation(const Permutation &p)
  : permap(p.permap) { }
  Permutation(const Permutation &p1, const Permutation &p2)
  : permap(p2.permap) { p1.apply(permap); }
  Permutation(const Permutation &p, int i)
  : permap(p.size(), p.permap, i) { }
  const Permutation &operator=(const Permutation &p)
  { permap = p.permap; return *this; }
  bool operator==(const Permutation &p)
  { return permap == p.permap; }
  int size() const
  { return permap.size(); }
  void print() const
  { permap.print(); }
  void apply(const IntSequence &src, IntSequence &tar) const;
  void apply(IntSequence &tar) const;
  void inverse();
  int tailIdentity() const;
  const IntSequence &getMap() const
  { return permap; }
  IntSequence &getMap()
  { return permap; }
protected:
  void computeSortingMap(const IntSequence &s);
};
```

This code is cited in section 1.

This code is used in section 196.

198. The **PermutationSet** maintains an array of of all permutations. The default constructor constructs one element permutation set of one element sets. The second constructor constructs a new permutation set over n from all permutations over $n-1$. The parameter n need not to be provided, but it serves to distinguish the constructor from copy constructor, which is not provided.

The method *getPreserving* returns a factor subgroup of permutations, which are invariants with respect to the given sequence. This are all permutations p yielding $p \circ s = s$, where s is the given sequence.

⟨ **PermutationSet** class declaration 198 ⟩ ≡

```
class PermutationSet {
  int order;
  int size;
  const Permutation **const pers;
public:
  PermutationSet();
  PermutationSet(const PermutationSet &ps, int n);
  ~PermutationSet();
  int getNum() const
  { return size; }
  const Permutation &get(int i) const
  { return *(pers[i]); }
  vector<const Permutation *> getPreserving(const IntSequence &s) const;
};
```

This code is used in section 196.

199. The permutation bundle encapsulates all permutations sets up to some given dimension.

⟨ **PermutationBundle** class declaration 199 ⟩ ≡

```
class PermutationBundle {
  vector<PermutationSet *> bundle;
public:
  PermutationBundle(int nmax);
  ~PermutationBundle();
  const PermutationSet &get(int n) const;
  void generateUpTo(int nmax);
};
```

This code is used in section 196.

200. End of `permutation.h` file.

201. Start of `permutation.cweb` file.

```
#include "permutation.h"
#include "tl_exception.h"
  ⟨ Permutation::apply code 202 ⟩;
  ⟨ Permutation::inverse code 203 ⟩;
  ⟨ Permutation::tailIdentity code 204 ⟩;
  ⟨ Permutation::computeSortingMap code 205 ⟩;
  ⟨ PermutationSet constructor code 1 206 ⟩;
  ⟨ PermutationSet constructor code 2 207 ⟩;
  ⟨ PermutationSet destructor code 208 ⟩;
  ⟨ PermutationSet::getPreserving code 209 ⟩;
  ⟨ PermutationBundle constructor code 210 ⟩;
  ⟨ PermutationBundle destructor code 211 ⟩;
  ⟨ PermutationBundle::get code 212 ⟩;
  ⟨ PermutationBundle::generateUpTo code 213 ⟩;
```

202. This is easy, we simply apply the map in the fashion $s \circ m$..

```
⟨ Permutation::apply code 202 ⟩ ≡
  void Permutation::apply(const IntSequence &src, IntSequence &tar) const
  {
    TL_RAISE_IF(src.size() ≠ permap.size() ∨ tar.size() ≠ permap.size(),
        "Wrong␣sizes␣of␣input␣or␣output␣in␣Permutation::apply");
    for (int i = 0; i < permap.size(); i++) tar[i] = src[permap[i]];
  }
  void Permutation::apply(IntSequence &tar) const
  {
    IntSequence tmp(tar);
    apply(tmp, tar);
  }
```

This code is used in section 201.

203.

```
⟨ Permutation::inverse code 203 ⟩ ≡
  void Permutation::inverse()
  {
    IntSequence former(permap);
    for (int i = 0; i < size(); i++) permap[former[i]] = i;
  }
```

This code is used in section 201.

204. Here we find a number of trailing indices which are identical with the permutation.

```
⟨ Permutation::tailIdentity code 204 ⟩ ≡
  int Permutation::tailIdentity() const
  {
    int i = permap.size();
    while (i > 0 ∧ permap[i − 1] ≡ i − 1) i−−;
    return permap.size() − i;
  }
```

This code is used in section 201.

205. This calculates a map which corresponds to sorting in the following sense: (sorted s) $\circ\, m = s$, where s is a given sequence.

We go through s and find an the same item in sorted s. We construct the *permap* from the found pair of indices. We have to be careful, to not assign to two positions in s the same position in sorted s, so we maintain a bitmap *flag*, in which we remember indices from the sorted s already assigned.

⟨ **Permutation** :: *computeSortingMap* code 205 ⟩ ≡

```
void Permutation::computeSortingMap(const IntSequence &s)
{
  IntSequence srt(s);
  srt.sort();
  IntSequence flags(s.size(), 0);
  for (int i = 0; i < s.size(); i++) {
    int j = 0;
    while (j < s.size() ∧ (flags[j] ∨ srt[j] ≠ s[i])) j++;
    TL_RAISE_IF(j ≡ s.size(), "Internal␣algorithm␣error␣in␣Permutation::computeSortingMap");
    flags[j] = 1;
    permap[i] = j;
  }
}
```

This code is used in section 201.

206.

⟨ **PermutationSet** constructor code 1 206 ⟩ ≡

```
PermutationSet::PermutationSet()
: order(1), size(1), pers(new const Permutation*[size]) {
  pers[0] = new Permutation(1);
}
```

This code is used in section 201.

207.

⟨ **PermutationSet** constructor code 2 207 ⟩ ≡

```
PermutationSet::PermutationSet(const PermutationSet &sp, int n)
: order(n), size(n * sp.size), pers(new const Permutation*[size]) {
  for (int i = 0; i < size; i++) pers[i] = Λ;
  TL_RAISE_IF(n ≠ sp.order + 1, "Wrong␣new␣order␣in␣PermutationSet␣constructor");
  int k = 0;
  for (int i = 0; i < sp.size; i++) {
    for (int j = 0; j < order; j++, k++) {
      pers[k] = new Permutation(*(sp.pers[i]), j);
    }
  }
}
```

This code is used in section 201.

208.

⟨ **PermutationSet** destructor code 208 ⟩ ≡

```
PermutationSet::~PermutationSet()
{
  for (int i = 0; i < size; i++)
    if (pers[i]) delete pers[i];
  delete[] pers;
}
```

This code is used in section 201.

209.

⟨ **PermutationSet**::*getPreserving* code 209 ⟩ ≡

```
vector<const Permutation *> PermutationSet::getPreserving(const IntSequence &s) const
{
  TL_RAISE_IF(s.size() != order, "Wrong sequence length in PermutationSet::getPreserving");
  vector<const Permutation *> res;
  IntSequence tmp(s.size());
  for (int i = 0; i < size; i++) {
    pers[i]->apply(s, tmp);
    if (s == tmp) {
      res.push_back(pers[i]);
    }
  }
  return res;
}
```

This code is used in section 201.

210.

⟨ **PermutationBundle** constructor code 210 ⟩ ≡

```
PermutationBundle::PermutationBundle(int nmax)
{
  nmax = max(nmax, 1);
  generateUpTo(nmax);
}
```

This code is used in section 201.

211.

⟨ **PermutationBundle** destructor code 211 ⟩ ≡

```
PermutationBundle::~PermutationBundle()
{
  for (unsigned int i = 0; i < bundle.size(); i++) delete bundle[i];
}
```

This code is used in section 201.

212.

⟨ **PermutationBundle** :: *get* code 212 ⟩ ≡

```
const PermutationSet &PermutationBundle::get(int n) const
{
  if (n > (int)(bundle.size()) ∨ n < 1) {
    TL_RAISE("Permutation␣set␣not␣found␣in␣PermutationSet::get");
    return *(bundle[0]);
  }
  else {
    return *(bundle[n − 1]);
  }
}
```

This code is used in section 201.

213.

⟨ **PermutationBundle** :: *generateUpTo* code 213 ⟩ ≡

```
void PermutationBundle::generateUpTo(int nmax)
{
  if (bundle.size() ≡ 0) bundle.push_back(new PermutationSet());
  int curmax = bundle.size();
  for (int n = curmax + 1; n ≤ nmax; n++) {
    bundle.push_back(new PermutationSet(*(bundle.back()), n));
  }
}
```

This code is used in section 201.

214. End of `permutation.cweb` file.

215. Tensors.

216. Tensor concept. Start of `tensor.h` file.

Here we define a tensor class. Tensor is a mathematical object corresponding to a $(n+1)$-dimensional array. An element of such array is denoted $[B]^{\beta}_{\alpha_1\ldots\alpha_n}$, where β is a special index and $\alpha_1\ldots\alpha_n$ are other indices. The class **Tensor** and its subclasses view such array as a 2D matrix, where β corresponds to one dimension, and $\alpha_1\ldots\alpha_2$ unfold to the other dimension. Whether β correspond to rows or columns is decided by tensor subclasses, however, most of our tensors will have rows indexed by β, and $\alpha_1\ldots\alpha_n$ will unfold column-wise.

There might be some symmetries in the tensor data. For instance, if α_1 is interchanged with α_3 and the both elements equal for all possible α_i, and β, then there is a symmetry of α_1 and α_3.

For any symmetry, there are basically two possible storages of the data. The first is unfolded storage, which stores all elements regardless the symmetry. The other storage type is folded, which stores only elements which do not repeat. We declare abstract classes for unfolded tensor, and folded tensor.

Also, here we also define a concept of tensor index which is the n-tuple $\alpha_1\ldots\alpha_n$. It is an iterator, which iterates in dependence of symmetry and storage of the underlying tensor.

Although we do not decide about possible symmetries at this point, it is worth noting that we implement two kinds of symmetries. The first one is a full symmetry where all indices are interchangeable. The second one is a generalization of the first. We define tensor of a symmetry, where there are a few groups of indices interchangeable within a group and not across. Moreover, the groups are required to be consequent partitions of the index n-tuple. This is, we do not allow α_1 be interchangeable with α_3 and not with α_2 at the same time.

However, some intermediate results are, in fact, tensors of a symmetry not fitting to our concept. We develop the tensor abstraction for it, but these objects are not used very often. They have limited usage due to their specialized constructor.

```
#ifndef TENSOR_H
#define TENSOR_H
#include "int_sequence.h"
#include "twod_matrix.h"
  ⟨ index class definition 217 ⟩;
  ⟨ Tensor class declaration 218 ⟩;
  ⟨ UTensor class declaration 219 ⟩;
  ⟨ FTensor class declaration 220 ⟩;
#endif
```

217. The index represents n-tuple $\alpha_1 \ldots \alpha_n$. Since its movement is dependent on the underlying tensor (with storage and symmetry), we maintain a pointer to that tensor, we maintain the n-tuple (or coordinates) as **IntSequence** and also we maintain the offset number (column, or row) of the index in the tensor. The pointer is const, since we do not need to change data through the index.

Here we require the *tensor* to implement *increment* and *decrement* methods, which calculate following and preceding n-tuple. Also, we need to calculate offset number from the given coordinates, so the tensor must implement method *getOffset*. This method is used only in construction of the index from the given coordinates. As the index is created, the offset is automatically incremented, and decremented together with index. The*getOffset* method can be relatively computationally complex. This must be kept in mind. Also we generally suppose that n-tuple of all zeros is the first offset (first columns or row).

What follows is a definition of index class, the only interesting point is **operator**≡ which decides only according to offset, not according to the coordinates. This is useful since there can be more than one of coordinate representations of past-the-end index.

⟨ index class definition 217 ⟩ ≡
```
template⟨class _Tptr⟩ class _index {
  typedef _index⟨_Tptr⟩ _Self;
  _Tptr tensor;
  int offset;
  IntSequence coor;
public:
  _index(_Tptr t, int n)
  : tensor(t), offset(0), coor(n, 0) { }
  _index(_Tptr t, const IntSequence &cr, int c)
  : tensor(t), offset(c), coor(cr) { }
  _index(_Tptr t, const IntSequence &cr)
  : tensor(t), offset(tensor→getOffset(cr)), coor(cr) { }
  _index(const _index &ind)
  : tensor(ind.tensor), offset(ind.offset), coor(ind.coor) { }
  const _Self &operator=(const _Self &in)
  { tensor = in.tensor; offset = in.offset; coor = in.coor;
    return *this; }
  _Self &operator++()
  { tensor→increment(coor); offset++; return *this; }
  _Self &operator--()
  { tensor→decrement(coor); offset--; return *this; }
  int operator*() const
  { return offset; }
  bool operator≡(const _index &n) const
  { return offset ≡ n.offset; }
  bool operator≠(const _index &n) const
  { return offset ≠ n.offset; }
  const IntSequence &getCoor() const
  { return coor; }
  void print() const
  { printf("%4d:␣", offset); coor.print(); }
};
```
This code is used in section 216.

218. Here is the **Tensor** class, which is nothing else than a simple subclass of **TwoDMatrix**. The unique semantically new member is *dim* which is tensor dimension (length of $\alpha_1 \dots \alpha_n$). We also declare *increment*, *decrement* and *getOffset* methods as pure virtual.

We also add members for index begin and index end. This is useful, since *begin* and *end* methods do not return instance but only references, which prevent making additional copy of index (for example in for cycles as $in \neq end()$ which would do a copy of index for each cycle). The index begin *in_beg* is constructed as a sequence of all zeros, and *in_end* is constructed from the sequence *last* passed to the constructor, since it depends on subclasses. Also we have to say, along what coordinate is the multidimensional index. This is used only for initialization of *in_end*.

Also, we declare static auxiliary functions for $\binom{n}{k}$ which is *noverk* and a^b, which is *power*.

⟨ **Tensor** class declaration 218 ⟩ ≡

```
class Tensor : public TwoDMatrix {
public:
  enum indor {
    along_row, along_col
  };
  typedef _index<const Tensor *> index;
protected:
  const index in_beg;
  const index in_end;
  int dim;
public:
  Tensor(indor io, const IntSequence &last, int r, int c, int d)
  : TwoDMatrix(r, c), in_beg(this, d), in_end(this, last, (io ≡ along_row) ? r : c), dim(d) { }
  Tensor(indor io, const IntSequence &first, const IntSequence &last, int r, int c, int d)
  : TwoDMatrix(r, c), in_beg(this, first, 0), in_end(this, last, (io ≡ along_row) ? r : c), dim(d) { }
  Tensor(int first_row, int num, Tensor &t)
  : TwoDMatrix(first_row, num, t), in_beg(t.in_beg), in_end(t.in_end), dim(t.dim) { }
  Tensor(const Tensor &t)
  : TwoDMatrix(t), in_beg(this, t.in_beg.getCoor(), *(t.in_beg)), in_end(this, t.in_end.getCoor(),
      *(t.in_end)), dim(t.dim) { }
  virtual ~Tensor() { }
  virtual void increment(IntSequence &v) const = 0;
  virtual void decrement(IntSequence &v) const = 0;
  virtual int getOffset(const IntSequence &v) const = 0;
  int dimen() const
  { return dim; }
  const index &begin() const
  { return in_beg; }
  const index &end() const
  { return in_end; }
  static int noverk(int n, int k);
  static int power(int a, int b);
  static int noverseq(const IntSequence &s)
  {
    IntSequence seq(s);
    return noverseq_ip((IntSequence &) s);
```

```
    }
  private:
    static int noverseq_ip(IntSequence &s);
  };
```

This code is cited in section 1.

This code is used in section 216.

219. Here is an abstraction for unfolded tensor. We provide a pure virtual method *fold* which returns a new instance of folded tensor of the same symmetry. Also we provide static methods for incrementing and decrementing an index with full symmetry and general symmetry as defined above.

⟨ **UTensor** class declaration 219 ⟩ ≡

```
  class FTensor;
  class UTensor : public Tensor {
  public:
    UTensor(indor io, const IntSequence &last, int r, int c, int d)
    : Tensor(io, last, r, c, d) { }
    UTensor(const UTensor &ut)
    : Tensor(ut) { }
    UTensor(int first_row, int num, UTensor &t)
    : Tensor(first_row, num, t) { }
    virtual ~UTensor( ) { }
    virtual FTensor &fold( ) const = 0;
    static void increment(IntSequence &v, int nv);
    static void decrement(IntSequence &v, int nv);
    static void increment(IntSequence &v, const IntSequence &nvmx);
    static void decrement(IntSequence &v, const IntSequence &nvmx);
    static int getOffset(const IntSequence &v, int nv);
    static int getOffset(const IntSequence &v, const IntSequence &nvmx);
  };
```

This code is cited in section 1.

This code is used in section 216.

220. This is an abstraction for folded tensor. It only provides a method *unfold*, which returns the unfolded version of the same symmetry, and static methods for decrementing indices.

We also provide static methods for decrementing the **IntSequence** in folded fashion and also calculating an offset for a given **IntSequence**. However, this is relatively complex calculation, so this should be avoided if possible.

⟨ **FTensor** class declaration 220 ⟩ ≡

```
class FTensor : public Tensor {
public:
    FTensor(indor io, const IntSequence &last, int r, int c, int d)
    : Tensor(io, last, r, c, d) { }
    FTensor(const FTensor &ft)
    : Tensor(ft) { }
    FTensor(int first_row, int num, FTensor &t)
    : Tensor(first_row, num, t) { }
    virtual ~FTensor() { }
    virtual UTensor &unfold() const = 0;
    static void decrement(IntSequence &v, int nv);
    static int getOffset(const IntSequence &v, int nv)
    { IntSequence vtmp(v); return getOffsetRecurse(vtmp, nv); }
private:
    static int getOffsetRecurse(IntSequence &v, int nv);
};
```

This code is cited in section 1.

This code is used in section 216.

221. End of `tensor.h` file.

222. Start of `tensor.cpp` file.

```
#include "tensor.h"
#include "tl_exception.h"
#include "tl_static.h"
    ⟨ Tensor static methods 223 ⟩;
    ⟨ Tensor::noverseq_ip static method 224 ⟩;
    ⟨ UTensor::increment code 1 225 ⟩;
    ⟨ UTensor::decrement code 1 226 ⟩;
    ⟨ UTensor::increment code 2 227 ⟩;
    ⟨ UTensor::decrement code 2 228 ⟩;
    ⟨ UTensor::getOffset code 1 229 ⟩;
    ⟨ UTensor::getOffset code 2 230 ⟩;
    ⟨ FTensor::decrement code 231 ⟩;
    ⟨ FTensor::getOffsetRecurse code 232 ⟩;
```

223. Here we implement calculation of $\binom{n}{k}$ where $n-k$ is usually bigger than k.

Also we implement a^b.

⟨ **Tensor** static methods 223 ⟩ ≡

```
int Tensor::noverk(int n, int k)
{
  return tls.ptriang→noverk(n, k);
}
int Tensor::power(int a, int b)
{
  int res = 1;
  for (int i = 0; i < b; i++) res *= a;
  return res;
}
```

This code is used in section 222.

224. Here we calculate a generalized combination number $\binom{a}{b_1,\ldots,b_n}$, where $a = b_1 + \ldots + b_n$. We use the identity

$$\binom{a}{b_1,\ldots,b_n} = \binom{b_1+b_2}{b_1} \cdot \binom{a}{b_1+b_2, b_3, \ldots, b_n}$$

This number is exactly a number of unfolded indices corresponding to one folded index, where the sequence $b_1, \ldots, b_n$ is the symmetry of the index.

⟨ **Tensor** :: *noverseq_ip* static method 224 ⟩ ≡

```
int Tensor::noverseq_ip(IntSequence &s)
{
  if (s.size() ≡ 0 ∨ s.size() ≡ 1) return 1;
  s[1] += s[0];
  return noverk(s[1], s[0]) * noverseq(IntSequence(s, 1, s.size()));
}
```

This code is used in section 222.

225. Here we increment a given sequence within full symmetry given by *nv*, which is number of variables in each dimension. The underlying tensor is unfolded, so we increase the rightmost by one, and if it is *nv* we zero it and increase the next one to the left.

⟨ **UTensor** :: *increment* code 1 225 ⟩ ≡

```
void UTensor::increment(IntSequence &v, int nv)
{
  if (v.size() ≡ 0) return;
  int i = v.size() − 1;
  v[i]++;
  while (i > 0 ∧ v[i] ≡ nv) {
    v[i] = 0;
    v[−−i]++;
  }
}
```

This code is used in section 222.

226. This is dual to **UTensor** :: *increment*(**IntSequence** &v, **int** nv).

⟨ **UTensor** :: *decrement* code 1 226 ⟩ ≡

```
void UTensor::decrement(IntSequence &v, int nv)
{
  if (v.size() ≡ 0) return;
  int i = v.size() − 1;
  v[i]−−;
  while (i > 0 ∧ v[i] ≡ −1) {
    v[i] = nv − 1;
    v[−−i]−−;
  }
}
```

This code is used in section 222.

227. Here we increment index for general symmetry for unfolded storage. The sequence *nvmx* assigns for each coordinate a number of variables. Since the storage is unfolded, we do not need information about what variables are symmetric, everything necessary is given by *nvmx*.

⟨ **UTensor** :: *increment* code 2 227 ⟩ ≡

```
void UTensor::increment(IntSequence &v, const IntSequence &nvmx)
{
  if (v.size() ≡ 0) return;
  int i = v.size() − 1;
  v[i]++;
  while (i > 0 ∧ v[i] ≡ nvmx[i]) {
    v[i] = 0;
    v[−−i]++;
  }
}
```

This code is used in section 222.

228. This is a dual code to **UTensor** :: *increment*(**IntSequence** &v, **const IntSequence** &*nvmx*).

⟨ **UTensor** :: *decrement* code 2 228 ⟩ ≡

```
void UTensor::decrement(IntSequence &v, const IntSequence &nvmx)
{
  if (v.size() ≡ 0) return;
  int i = v.size() − 1;
  v[i]−−;
  while (i > 0 ∧ v[i] ≡ −1) {
    v[i] = nvmx[i] − 1;
    v[−−i]−−;
  }
}
```

This code is used in section 222.

229. Here we return an offset for a given coordinates of unfolded full symmetry tensor. This is easy.

⟨ **UTensor** :: *getOffset* code 1 229 ⟩ ≡

```
int UTensor::getOffset(const IntSequence &v, int nv)
{
  int pow = 1;
  int res = 0;
  for (int i = v.size() - 1; i >= 0; i--) {
    res += v[i] * pow;
    pow *= nv;
  }
  return res;
}
```

This code is used in section 222.

230. Also easy.

⟨ **UTensor** :: *getOffset* code 2 230 ⟩ ≡

```
int UTensor::getOffset(const IntSequence &v, const IntSequence &nvmx)
{
  int pow = 1;
  int res = 0;
  for (int i = v.size() - 1; i >= 0; i--) {
    res += v[i] * pow;
    pow *= nvmx[i];
  }
  return res;
}
```

This code is used in section 222.

231. Decrementing of coordinates of folded index is not that easy. Note that if a trailing part of coordinates is (b, a, a, a) (for instance) with $b < a$, then a preceding coordinates are $(b, a - 1, n - 1, n - 1)$, where n is a number of variables nv. So we find the left most element which is equal to the last element, decrease it by one, and then set all elements to the right to $n - 1$.

⟨ **FTensor** :: *decrement* code 231 ⟩ ≡

```
void FTensor::decrement(IntSequence &v, int nv)
{
  int i = v.size() - 1;
  while (i > 0 && v[i - 1] == v[i]) i--;
  v[i]--;
  for (int j = i + 1; j < v.size(); j++) v[j] = nv - 1;
}
```

This code is used in section 222.

232. This calculates order of the given index of our ordering of indices. In order to understand how it works, let us take number of variables n and dimension k, and write down all the possible combinations of indices in our ordering. For example for $n = 4$ and $k = 3$, the sequence looks as:

0 0 0	1 1 1	2 2 2	3 3 3
0 0 1	1 1 2	2 2 3	
0 0 2	1 1 3	2 3 3	
0 0 3	1 2 2		
0 1 1	1 2 3		
0 1 2	1 3 3		
0 1 3			
0 2 2			
0 2 3			
0 3 3			

Now observe, that a number of sequences starting with zero is the same as total number of sequences with the same number of variables but with dimension minus one. More generally, if $S_{n,k}$ denotes number of indices of n variables and dimension k, then the number of indices beginning with m is exactly $S_{n-m,k-1}$. This is because m can be subtracted from all items, and we obtain sequence of indices of $n - m$ variables. So we have formula:

$$S_{n,k} = S_{n,k-1} + S_{n-1,k-1} + \ldots + S_{1,k-1}$$

Now it is easy to calculate offset of index of the form $(m, \ldots, m)$. It is a sum of all above it, this is $S_{n,k-1} + \ldots + S_{n-m,k-1}$. We know that $S_{n,k} = \binom{n+k-1}{k}$. Using above formula, we can calculate offset of $(m, \ldots, m)$ as

$$\binom{n+k-1}{k} - \binom{n-m+k-1}{k}$$

The offset of general index $(m_1, m_2, \ldots, m_k)$ is calculated recursively, since it is offset of $(m_1, \ldots, m_1)$ for n variables plus offset of $(m_2 - m_1, m_3 - m_1, \ldots, m_k - m_1)$ for $n - m_1$ variables.

⟨ **FTensor** :: *getOffsetRecurse* code 232 ⟩ ≡

```
int FTensor::getOffsetRecurse(IntSequence &v, int nv)
{
  if (v.size() ≡ 0) return 0;
  int prefix = v.getPrefixLength();
  int m = v[0];
  int k = v.size();
  int s1 = noverk(nv + k − 1, k) − noverk(nv − m + k − 1, k);
  IntSequence subv(v, prefix, k);
  subv.add(−m);
  int s2 = getOffsetRecurse(subv, nv − m);
  return s1 + s2;
}
```

This code is used in section 222.

233. End of `tensor.cpp` file.

234. Full symmetry tensor. Start of `fs_tensor.h` file.

Here we define folded and unfolded tensors for full symmetry. All tensors from here are identifying the multidimensional index with columns.

```
#ifndef FS_TENSOR_H
#define FS_TENSOR_H
#include "tensor.h"
#include "symmetry.h"
  class FGSTensor;
  class UGSTensor;
  class FRSingleTensor;
  class FSSparseTensor;
  ⟨ FFSTensor class declaration 235 ⟩;
  ⟨ UFSTensor class declaration 237 ⟩;
#endif
```

235. Folded tensor with full symmetry maintains only information about number of symmetrical variables *nv*. Further, we implement what is left from the super class **FTensor**.

We implement *getOffset* which should be used with care since its complexity.

We implement a method adding a given general symmetry tensor to the full symmetry tensor supposing the variables of the general symmetry tensor are stacked giving only one variable of the full symmetry tensor. For instance, if $x = [y^T, u^T]^T$, then we can add tensor $\left[g_{y^2u}\right]$ to tensor g_{x^3}. This is done in method *addSubTensor*. Consult ⟨ **FGSTensor** class declaration 261 ⟩ to know what is general symmetry tensor.

```
⟨ FFSTensor class declaration 235 ⟩ ≡
  class UFSTensor;
  class FFSTensor : public FTensor {
    int nv;
  public:
    ⟨ FFSTensor constructor declaration 236 ⟩;
    void increment(IntSequence &v) const;
    void decrement(IntSequence &v) const;
    UTensor &unfold() const;
    Symmetry getSym() const
    { return Symmetry(dimen()); }
    int getOffset(const IntSequence &v) const;
    void addSubTensor(const FGSTensor &t);
    int nvar() const
    { return nv; }
    static int calcMaxOffset(int nvar, int d);
  };
```

This code is cited in section 1.

This code is used in section 234.

236. Here are the constructors. The second constructor constructs a tensor by one-dimensional contraction from the higher dimensional tensor t. This is, it constructs a tensor

$$[g_{y^n}]_{\alpha_1\ldots\alpha_n} = \left[t_{y^{n+1}}\right]_{\alpha_1\ldots\alpha_n\beta} [x]^\beta$$

See implementation ⟨ **FFSTensor** contraction constructor 241 ⟩ for details.

The next constructor converts from sparse tensor (which is fully symmetric and folded by nature).

The fourth constructs object from unfolded fully symmetric.

The fifth constructs a subtensor of selected rows.

⟨ **FFSTensor** constructor declaration 236 ⟩ ≡

```
FFSTensor(int r, int nvar, int d)
: FTensor(along_col, IntSequence(d, nvar), r, calcMaxOffset(nvar, d), d), nv(nvar) { }
FFSTensor(const FFSTensor &t, const ConstVector &x);
FFSTensor(const FSSparseTensor &t);
FFSTensor(const FFSTensor &ft)
: FTensor(ft), nv(ft.nv) { }
FFSTensor(const UFSTensor &ut);
FFSTensor(int first_row, int num, FFSTensor &t)
: FTensor(first_row, num, t), nv(t.nv) { }
```

This code is used in section 235.

237. Unfolded fully symmetric tensor is almost the same in structure as **FFSTensor**, but the method *unfoldData*. It takes columns which also exist in folded version and copies them to all their symmetrical locations. This is useful when constructing unfolded tensor from folded one.

⟨ **UFSTensor** class declaration 237 ⟩ ≡

```
class UFSTensor : public UTensor {
  int nv;
public:
  ⟨ UFSTensor constructor declaration 238 ⟩;
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  FTensor &fold() const;
  Symmetry getSym() const
  { return Symmetry(dimen()); }
  int getOffset(const IntSequence &v) const;
  void addSubTensor(const UGSTensor &t);
  int nvar() const
  { return nv; }
  static int calcMaxOffset(int nvar, int d)
  { return power(nvar, d); }
private:
  void unfoldData();
};
```

This code is cited in section 1.

This code is used in section 234.

238.

⟨ **UFSTensor** constructor declaration 238 ⟩ ≡
  UFSTensor(**int** r, **int** $nvar$, **int** d)
  : **UTensor**($along\_col$, **IntSequence**(d, $nvar$), r, $calcMaxOffset$($nvar$, d), d), nv($nvar$) { }
  UFSTensor(**const UFSTensor** &t, **const ConstVector** &x);
  UFSTensor(**const UFSTensor** &ut)
  : **UTensor**(ut), nv($ut.nv$) { }
  UFSTensor(**const FFSTensor** &ft);
  UFSTensor(**int** $first\_row$, **int** num, **UFSTensor** &t)
  : **UTensor**($first\_row$, num, t), nv($t.nv$) { }

This code is used in section 237.

239. End of `fs_tensor.h` file.

240. Start of `fs_tensor.cpp` file.

\#**include** "fs_tensor.h"
\#**include** "gs_tensor.h"
\#**include** "sparse_tensor.h"
\#**include** "rfs_tensor.h"
\#**include** "tl_exception.h"
  ⟨ **FFSTensor** contraction constructor 241 ⟩;
  ⟨ **FFSTensor** :: $calcMaxOffset$ code 242 ⟩;
  ⟨ **FFSTensor** conversion from sparse 243 ⟩;
  ⟨ **FFSTensor** conversion from unfolded 244 ⟩;
  ⟨ **FFSTensor** :: $unfold$ code 245 ⟩;
  ⟨ **FFSTensor** :: $increment$ code 246 ⟩;
  ⟨ **FFSTensor** :: $decrement$ code 247 ⟩;
  ⟨ **FFSTensor** :: $getOffset$ code 248 ⟩;
  ⟨ **FFSTensor** :: $addSubTensor$ code 249 ⟩;
  ⟨ **UFSTensor** contraction constructor 251 ⟩;
  ⟨ **UFSTensor** conversion from folded 252 ⟩;
  ⟨ **UFSTensor** :: $fold$ code 253 ⟩;
  ⟨ **UFSTensor** increment and decrement 254 ⟩;
  ⟨ **UFSTensor** :: $getOffset$ code 255 ⟩;
  ⟨ **UFSTensor** :: $addSubTensor$ code 256 ⟩;
  ⟨ **UFSTensor** :: $unfoldData$ code 257 ⟩;

241. This constructs a fully symmetric tensor as given by the contraction:

$$\left[g_{y^n}\right]_{\alpha_1\ldots\alpha_n} = \left[t_{y^{n+1}}\right]_{\alpha_1\ldots\alpha_n\beta}\left[x\right]^\beta$$

We go through all columns of output tensor $[g]$ and for each column we cycle through all variables, insert a variable to the column coordinates obtaining a column of tensor $[t]$. the column is multiplied by an appropriate item of x and added to the column of $[g]$ tensor.

⟨ **FFSTensor** contraction constructor 241 ⟩ ≡

```
FFSTensor::FFSTensor(const FFSTensor &t, const ConstVector &x)
: FTensor(along_col, IntSequence(t.dimen() − 1, t.nvar()), t.nrows(), calcMaxOffset(t.nvar(),
      t.dimen() − 1), t.dimen() − 1), nv(t.nvar()) {
  TL_RAISE_IF(t.dimen() < 1, "Wrong␣dimension␣for␣tensor␣contraction␣of␣FFSTensor");
  TL_RAISE_IF(t.nvar() ≠ x.length(),
      "Wrong␣number␣of␣variables␣for␣tensor␣contraction␣of␣FFSTensor");
  zeros();
  for (Tensor::index to = begin(); to ≠ end(); ++to) {
    for (int i = 0; i < nvar(); i++) {
      IntSequence from_ind(i, to.getCoor());
      Tensor::index from(&t, from_ind);
      addColumn(x[i], t, *from, *to);
    }
  }
}
```

This code is cited in sections 236 and 251.

This code is used in section 240.

242. This returns number of indices for folded tensor with full symmetry. Let n be a number of variables *nvar* and d the dimension *dim*. Then the number of indices is $\binom{n+d-1}{d}$.

⟨ **FFSTensor** :: *calcMaxOffset* code 242 ⟩ ≡

```
int FFSTensor::calcMaxOffset(int nvar, int d)
{
  if (nvar ≡ 0 ∧ d ≡ 0) return 1;
  if (nvar ≡ 0 ∧ d > 0) return 0;
  return noverk(nvar + d − 1, d);
}
```

This code is used in section 240.

243. The conversion from sparse tensor is clear. We go through all the tensor and write to the dense what is found.

⟨ **FFSTensor** conversion from sparse 243 ⟩ ≡

```
FFSTensor::FFSTensor(const FSSparseTensor &t)
: FTensor(along_col, IntSequence(t.dimen(), t.nvar()), t.nrows(), calcMaxOffset(t.nvar(), t.dimen()),
      t.dimen()), nv(t.nvar()) {
  zeros();
  for (FSSparseTensor::const_iterator it = t.getMap().begin(); it ≠ t.getMap().end(); ++it) {
    index ind(this, (*it).first);
    get((*it).second.first, *ind) = (*it).second.second;
  }
}
```

This code is used in section 240.

244. The conversion from unfolded copies only columns of respective coordinates. So we go through all the columns in the folded tensor (this), make an index of the unfolded vector from coordinates, and copy the column.

```
⟨ FFSTensor conversion from unfolded 244 ⟩ ≡
  FFSTensor::FFSTensor(const UFSTensor &ut)
  : FTensor(along_col, IntSequence(ut.dimen(), ut.nvar()), ut.nrows(), calcMaxOffset(ut.nvar(),
        ut.dimen()), ut.dimen()), nv(ut.nvar()) {
    for (index in = begin(); in ≠ end(); ++in) {
      index src(&ut, in.getCoor());
      copyColumn(ut, *src, *in);
    }
  }
```

This code is used in section 240.

245. Here just make a new instance and return the reference.

```
⟨ FFSTensor::unfold code 245 ⟩ ≡
  UTensor &FFSTensor::unfold() const
  {
    return *(new UFSTensor(*this));
  }
```

This code is used in section 240.

246. Incrementing is easy. We have to increment by calling static method **UTensor** :: *increment* first. In this way, we have coordinates of unfolded tensor. Then we have to skip to the closest folded index which corresponds to monotonizeing the integer sequence.

```
⟨ FFSTensor::increment code 246 ⟩ ≡
  void FFSTensor::increment(IntSequence &v) const
  {
    TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣FFSTensor::increment");
    UTensor::increment(v, nv);
    v.monotone();
  }
```

This code is used in section 240.

247. Decrement calls static **FTensor** :: *decrement*.

```
⟨ FFSTensor::decrement code 247 ⟩ ≡
  void FFSTensor::decrement(IntSequence &v) const
  {
    TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣FFSTensor::decrement");
    FTensor::decrement(v, nv);
  }
```

This code is used in section 240.

248.

⟨ **FFSTensor** :: *getOffset* code 248 ⟩ ≡

```
int FFSTensor::getOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input␣vector␣size␣in␣FFSTensor::getOffset");
  return FTensor::getOffset(v, nv);
}
```

This code is used in section 240.

249. Here we add a general symmetry tensor to the (part of) full symmetry tensor provided that the unique variable of the full symmetry tensor is a stack of variables from the general symmetry tensor.

We check for the dimensions and number of variables. Then we calculate a shift of coordinates when going from the general symmetry tensor to full symmetry (it corresponds to shift of coordinates induces by stacking the variables). Then we add the appropriate columns by going through the columns in general symmetry, adding the shift and sorting.

⟨ **FFSTensor** :: *addSubTensor* code 249 ⟩ ≡

```
void FFSTensor::addSubTensor(const FGSTensor &t)
{
  TL_RAISE_IF(dimen() ≠ t.getDims().dimen(),
      "Wrong␣dimensions␣for␣FFSTensor::addSubTensor");
  TL_RAISE_IF(nvar() ≠ t.getDims().getNVS().sum(), "Wrong␣nvs␣for␣FFSTensor::addSubTensor");
  ⟨ set shift for addSubTensor 250 ⟩;
  for (Tensor::index ind = t.begin(); ind ≠ t.end(); ++ind) {
    IntSequence c(ind.getCoor());
    c.add(1, shift);
    c.sort();
    Tensor::index tar(this, c);
    addColumn(t, *ind, *tar);
  }
}
```

This code is cited in section 256.

This code is used in section 240.

250.

⟨ set shift for *addSubTensor* 250 ⟩ ≡

```
IntSequence shift_pre(t.getSym().num(), 0);
for (int i = 1; i < t.getSym().num(); i++) shift_pre[i] = shift_pre[i - 1] + t.getDims().getNVS()[i - 1];
IntSequence shift(t.getSym(), shift_pre);
```

This code is used in sections 249 and 256.

251. This is a bit more straightforward than ⟨ **FFSTensor** contraction constructor 241 ⟩. We do not add column by column but we do it by submatrices due to regularity of the unfolded tensor.

⟨ **UFSTensor** contraction constructor 251 ⟩ ≡

```
UFSTensor::UFSTensor(const UFSTensor &t, const ConstVector &x)
  : UTensor(along_col, IntSequence(t.dimen() − 1, t.nvar()), t.nrows(), calcMaxOffset(t.nvar(),
        t.dimen() − 1), t.dimen() − 1), nv(t.nvar()) {
    TL_RAISE_IF(t.dimen() < 1, "Wrong dimension for tensor contraction of UFSTensor");
    TL_RAISE_IF(t.nvar() ≠ x.length(),
        "Wrong number of variables for tensor contraction of UFSTensor");
    zeros();
    for (int i = 0; i < ncols(); i++) {
      ConstTwoDMatrix tpart(t, i * nvar(), nvar());
      Vector outcol(*this, i);
      tpart.multaVec(outcol, x);
    }
  }
```

This code is used in section 240.

252. Here we convert folded full symmetry tensor to unfolded. We copy all columns of folded tensor, and then call *unfoldData*( ).

⟨ **UFSTensor** conversion from folded 252 ⟩ ≡

```
UFSTensor::UFSTensor(const FFSTensor &ft)
  : UTensor(along_col, IntSequence(ft.dimen(), ft.nvar()), ft.nrows(), calcMaxOffset(ft.nvar(),
        ft.dimen()), ft.dimen()), nv(ft.nvar()) {
    for (index src = ft.begin(); src ≠ ft.end(); ++src) {
      index in(this, src.getCoor());
      copyColumn(ft, *src, *in);
    }
    unfoldData();
  }
```

This code is used in section 240.

253. Here we just return a reference to new instance of folded tensor.

⟨ **UFSTensor**::*fold* code 253 ⟩ ≡

```
FTensor &UFSTensor::fold() const
{
  return *(new FFSTensor(*this));
}
```

This code is used in section 240.

254. Here we just call **UTensor** respective static methods.

⟨ **UFSTensor** increment and decrement 254 ⟩ ≡

```
void UFSTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong input/output vector size in UFSTensor::increment");
  UTensor::increment(v, nv);
}
void UFSTensor::decrement(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong input/output vector size in UFSTensor::decrement");
  UTensor::decrement(v, nv);
}
```

This code is used in section 240.

255.

⟨ **UFSTensor** :: *getOffset* code 255 ⟩ ≡

```
int UFSTensor::getOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong input vector size in UFSTensor::getOffset");
  return UTensor::getOffset(v, nv);
}
```

This code is used in section 240.

256. This is very similar to ⟨ **FFSTensor** :: *addSubTensor* code 249 ⟩. The only difference is the addition. We go through all columns in the full symmetry tensor and cancel the shift. If the coordinates after the cancellation are positive, we find the column in the general symmetry tensor, and add it.

⟨ **UFSTensor** :: *addSubTensor* code 256 ⟩ ≡

```
void UFSTensor::addSubTensor(const UGSTensor &t)
{
  TL_RAISE_IF(dimen() ≠ t.getDims().dimen(),
      "Wrong dimensions for UFSTensor::addSubTensor");
  TL_RAISE_IF(nvar() ≠ t.getDims().getNVS().sum(), "Wrong nvs for UFSTensor::addSubTensor");
  ⟨set shift for addSubTensor 250⟩;
  for (Tensor::index tar = begin(); tar ≠ end(); ++tar) {
    IntSequence c(tar.getCoor());
    c.sort();
    c.add(−1, shift);
    if (c.isPositive() ∧ c.less(t.getDims().getNVX())) {
      Tensor::index from(&t, c);
      addColumn(t, ∗from, ∗tar);
    }
  }
}
```

This code is used in section 240.

257. Here we go through all columns, find a column of folded index, and then copy the column data. Finding the index is done by sorting the integer sequence.

⟨ **UFSTensor** :: *unfoldData* code 257 ⟩ ≡

```
void UFSTensor::unfoldData()
{
  for (index in = begin(); in ≠ end(); ++in) {
    IntSequence v(in.getCoor());
    v.sort();
    index tmp(this, v);
    copyColumn(*tmp, *in);
  }
}
```

This code is used in section 240.

258. End of `fs_tensor.cpp` file.

259. General symmetry tensor. Start of `gs_tensor.h` file.

Here we define tensors for general symmetry. All tensors from here are identifying the multidimensional index with columns. Thus all symmetries regard to columns. The general symmetry here is not the most general. It captures all symmetries of indices which are given by continuous partitioning of indices. Two items are symmetric if they belong to the same group. The continuity implies that if two items belong to one group, then all items between them belong to that group. This continuous partitioning of indices is described by **Symmetry** class.

The dimension of the tensors here are described (besides the symmetry) also by number of variables for each group. This is dealt in the class for tensor dimensions defined also here.

```
#ifndef GS_TENSOR_H
#define GS_TENSOR_H
#include "tensor.h"
#include "fs_tensor.h"
#include "symmetry.h"
#include "rfs_tensor.h"
  class FGSTensor;
  class UGSTensor;
  class FSSparseTensor;
  ⟨ TensorDimens class declaration 260 ⟩;
  ⟨ FGSTensor class declaration 261 ⟩;
  ⟨ UGSTensor class declaration 263 ⟩;
#endif
```

260. This class encapsulates symmetry information for the general symmetry tensor. It maintains a vector of variable numbers *nvs*, and symmetry *sym*. For example, let the symmetry be y^2u^3, and variable numbers be 10 for y, and 5 for u. Then the *nvs* is $(10,5)$, and *sym* is $(2,3)$. Also it maintains *nvmax* unfolded *nvs* with respect to the symmetry, this is $(10,10,5,5,5)$.

The constructors of **TensorDimens** are clear and pretty intuitive but the constructor which is used for slicing fully symmetric tensor. It constructs the dimensions from the partitioning of variables of fully symmetric tensor. Let the partitioning be, for instance, (a,b,c,d), where (n_a,n_b,n_c,n_d) are lengths of the partitions. Let one want to get a slice only of the part of the fully symmetric tensor corresponding to indices of the form b^2d^3. This corresponds to the symmetry $a^0b^2c^0d^3$. So, the dimension of the slice would be also (n_a,n_b,n_c,n_d) for number of variables and $(0,2,0,3)$ for the symmetry. So we provide the constructor which takes sizes of partitions (n_a,n_b,n_c,n_d) as **IntSequence**, and indices of picked partitions, in our case $(1,1,3,3,3)$, as **IntSequence**.

The class is able to calculate number of offsets (columns or rows depending what matrix coordinate we describe) in unfolded and folded tensors with the given symmetry.

⟨ **TensorDimens** class declaration 260 ⟩ ≡

```
class TensorDimens {
protected:
  IntSequence nvs;
  Symmetry sym;
  IntSequence nvmax;
public:
  TensorDimens(const Symmetry &s, const IntSequence &nvars)
  : nvs(nvars), sym(s), nvmax(sym, nvs) { }
  TensorDimens(int nvar, int dimen)
  : nvs(1), sym(dimen), nvmax(dimen, nvar) { nvs[0] = nvar; }
  TensorDimens(const TensorDimens &td)
  : nvs(td.nvs), sym(td.sym), nvmax(td.nvmax) { }
  virtual ~TensorDimens() { }
  TensorDimens(const IntSequence &ss, const IntSequence &coor);
  const TensorDimens &operator=(const TensorDimens &td)
  { nvs = td.nvs; sym = td.sym; nvmax = td.nvmax; return *this; }
  bool operator≡(const TensorDimens &td) const
  { return nvs ≡ td.nvs ∧ sym ≡ td.sym; }
  bool operator≠(const TensorDimens &td) const
  { return ¬operator≡(td); }
  int dimen() const
  { return sym.dimen(); }
  int getNVX(int i) const
  { return nvmax[i]; }
  const IntSequence &getNVS() const
  { return nvs; }
  const IntSequence &getNVX() const
  { return nvmax; }
  const Symmetry &getSym() const
  { return sym; }
  int calcUnfoldMaxOffset() const;
  int calcFoldMaxOffset() const;
  int calcFoldOffset(const IntSequence &v) const;
```

```
    void decrement(IntSequence &v) const;
  };
```

This code is cited in section 267.

This code is used in section 259.

261. Here is a class for folded general symmetry tensor. It only contains tensor dimensions, it defines types for indices, implement virtual methods of super class **FTensor**.

We add a method *contractAndAdd* which performs a contraction of one variable in the tensor. This is, for instance

$$[r_{x^i z^k}]_{\alpha_1\ldots\alpha_i\gamma_1\ldots\gamma_k} = \left[t_{x^i y^j z^k}\right]_{\alpha_1\ldots\alpha_i\beta_1\ldots\beta_j\gamma_1\ldots\gamma_k} [c]^{\beta_1\ldots\beta_j}$$

Also we add *getOffset* which should be used with care.

⟨ **FGSTensor** class declaration 261 ⟩ ≡

```
  class GSSparseTensor;
  class FGSTensor : public FTensor {
    friend class UGSTensor;
    const TensorDimens tdims;
  public:
    ⟨ FGSTensor constructor declarations 262 ⟩;
    virtual ~FGSTensor() { }
    void increment(IntSequence &v) const;
    void decrement(IntSequence &v) const
    { tdims.decrement(v); }
    UTensor &unfold() const;
    const TensorDimens &getDims() const
    { return tdims; }
    const Symmetry &getSym() const
    { return getDims().getSym(); }
    void contractAndAdd(int i, FGSTensor &out, const FRSingleTensor &col) const;
    int getOffset(const IntSequence &v) const
    { return tdims.calcFoldOffset(v); }
  };
```

This code is cited in sections 1 and 235.

This code is used in section 259.

262. These are standard constructors followed by two slicing. The first constructs a slice from the sparse, the second from the dense (both fully symmetric). Next constructor is just a conversion from *GSSParseTensor*. The last constructor allows for in-place conversion from **FFSTensor** to **FGSTensor**.

⟨ **FGSTensor** constructor declarations 262 ⟩ ≡

```
FGSTensor(int r, const TensorDimens &td)
: FTensor(along_col, td.getNVX(), r, td.calcFoldMaxOffset(), td.dimen()), tdims(td) { }
FGSTensor(const FGSTensor &ft)
: FTensor(ft), tdims(ft.tdims) { }
FGSTensor(const UGSTensor &ut);
FGSTensor(int first_row, int num, FGSTensor &t)
: FTensor(first_row, num, t), tdims(t.tdims) { }
FGSTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence &coor, const
    TensorDimens &td);
FGSTensor(const FFSTensor &t, const IntSequence &ss, const IntSequence &coor, const
    TensorDimens &td);
FGSTensor(const GSSparseTensor &sp);
FGSTensor(FFSTensor &t)
: FTensor(0, t.nrows(), t), tdims(t.nvar(), t.dimen()) { }
```

This code is used in section 261.

263. Besides similar things that has **FGSTensor**, we have here also method *unfoldData*, and helper method *getFirstIndexOf* which corresponds to sorting coordinates in fully symmetric case (here the action is more complicated, so we put it to the method).

⟨ **UGSTensor** class declaration 263 ⟩ ≡

```
class UGSTensor : public UTensor {
  friend class FGSTensor;
  const TensorDimens tdims;
public:
  ⟨ UGSTensor constructor declarations 264 ⟩;
  virtual ~UGSTensor() { }
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  FTensor &fold() const;
  const TensorDimens &getDims() const
  { return tdims; }
  const Symmetry &getSym() const
  { return getDims().getSym(); }
  void contractAndAdd(int i, UGSTensor &out, const URSingleTensor &col) const;
  int getOffset(const IntSequence &v) const;
private:
  void unfoldData();
public:
  index getFirstIndexOf(const index &in) const;
};
```

This code is cited in section 1.

This code is used in section 259.

264. These are standard constructors. The last two constructors are slicing. The first makes a slice from fully symmetric sparse, the second from fully symmetric dense unfolded tensor. The last constructor allows for in-place conversion from **UFSTensor** to **UGSTensor**.

⟨ **UGSTensor** constructor declarations 264 ⟩ ≡
UGSTensor(**int** *r*, **const TensorDimens** &*td*)
: **UTensor**(*along_col*, *td.getNVX*( ), *r*, *td.calcUnfoldMaxOffset*( ), *td.dimen*( )), *tdims*(*td*) { }
UGSTensor(**const UGSTensor** &*ut*)
: **UTensor**(*ut*), *tdims*(*ut.tdims*) { }
UGSTensor(**const FGSTensor** &*ft*);
UGSTensor(**int** *first_row*, **int** *num*, **UGSTensor** &*t*)
: **UTensor**(*first_row*, *num*, *t*), *tdims*(*t.tdims*) { }
UGSTensor(**const FSSparseTensor** &*t*, **const IntSequence** &*ss*, **const IntSequence** &*coor*, **const TensorDimens** &*td*);
UGSTensor(**const UFSTensor** &*t*, **const IntSequence** &*ss*, **const IntSequence** &*coor*, **const TensorDimens** &*td*);
UGSTensor(**UFSTensor** &*t*)
: **UTensor**(0, *t.nrows*( ), *t*), *tdims*(*t.nvar*( ), *t.dimen*( )) { }

This code is used in section 263.

265. End of `gs_tensor.h` file.

266. Start of `gs_tensor.cpp` file.

#**include** "gs_tensor.h"
#**include** "sparse_tensor.h"
#**include** "tl_exception.h"
#**include** "kron_prod.h"
⟨ **TensorDimens** constructor code 267 ⟩;
⟨ **TensorDimens** :: *calcUnfoldMaxOffset* code 268 ⟩;
⟨ **TensorDimens** :: *calcFoldMaxOffset* code 269 ⟩;
⟨ **TensorDimens** :: *calcFoldOffset* code 270 ⟩;
⟨ **TensorDimens** :: *decrement* code 271 ⟩;
⟨ **FGSTensor** conversion from **UGSTensor** 274 ⟩;
⟨ **FGSTensor** slicing from **FSSparseTensor** 275 ⟩;
⟨ **FGSTensor** slicing from **FFSTensor** 277 ⟩;
⟨ **FGSTensor** conversion from **GSSparseTensor** 278 ⟩;
⟨ **FGSTensor** :: *increment* code 279 ⟩;
⟨ **FGSTensor** :: *unfold* code 280 ⟩;
⟨ **FGSTensor** :: *contractAndAdd* code 281 ⟩;
⟨ **UGSTensor** conversion from **FGSTensor** 283 ⟩;
⟨ **UGSTensor** slicing from **FSSparseTensor** 284 ⟩;
⟨ **UGSTensor** slicing from **UFSTensor** 285 ⟩;
⟨ **UGSTensor** increment and decrement codes 286 ⟩;
⟨ **UGSTensor** :: *fold* code 287 ⟩;
⟨ **UGSTensor** :: *getOffset* code 288 ⟩;
⟨ **UGSTensor** :: *unfoldData* code 289 ⟩;
⟨ **UGSTensor** :: *getFirstIndexOf* code 290 ⟩;
⟨ **UGSTensor** :: *contractAndAdd* code 291 ⟩;

267. This constructs the tensor dimensions for slicing. See ⟨ **TensorDimens** class declaration 260 ⟩ for details.

⟨ **TensorDimens** constructor code 267 ⟩ ≡

```
TensorDimens::TensorDimens(const IntSequence &ss, const IntSequence &coor)
  : nvs(ss), sym(ss.size(), ""), nvmax(coor.size(), 0) {
    TL_RAISE_IF(¬coor.isSorted(),
        "Coordinates␣not␣sorted␣in␣TensorDimens␣slicing␣constructor");
    TL_RAISE_IF(coor[0] < 0 ∨ coor[coor.size() − 1] ≥ ss.size(),
        "A␣coordinate␣out␣of␣stack␣range␣in␣TensorDimens␣slicing␣constructor");
    for (int i = 0; i < coor.size(); i++) {
      sym[coor[i]]++;
      nvmax[i] = ss[coor[i]];
    }
}
```

This code is used in section 266.

268. Number of unfold offsets is a product of all members of *nvmax*.

⟨ **TensorDimens** :: *calcUnfoldMaxOffset* code 268 ⟩ ≡

```
int TensorDimens::calcUnfoldMaxOffset() const
{
  return nvmax.mult();
}
```

This code is used in section 266.

269. Number of folded offsets is a product of all unfold offsets within each equivalence class of the symmetry.

⟨ **TensorDimens** :: *calcFoldMaxOffset* code 269 ⟩ ≡

```
int TensorDimens::calcFoldMaxOffset() const
{
  int res = 1;
  for (int i = 0; i < nvs.size(); i++) {
    if (nvs[i] ≡ 0 ∧ sym[i] > 0) return 0;
    if (sym[i] > 0) res *= Tensor::noverk(nvs[i] + sym[i] − 1, sym[i]);
  }
  return res;
}
```

This code is used in section 266.

270. Here we implement offset calculation for folded general symmetry tensor. The offset of a given sequence is calculated by breaking the sequence to subsequences according to the symmetry. The offset is orthogonal with respect to the blocks, this means that indexing within the blocks is independent. If there are two blocks, for instance, then the offset will be an offset within the outer block (the first) multiplied with all offsets of the inner block (last) plus an offset within the second block.

Generally, the resulting offset r will be

$$\sum_{i=1}^{s} r_i \cdot \left(\prod_{j=i+1}^{s} n_j\right),$$

where s is a number of blocks (*getSym*( ).*num*( )), r_i is an offset within i-th block, and n_j is a number of all offsets in j-th block.

In the code, we go from the innermost to the outermost, maintaining the product in *pow*.

⟨ **TensorDimens** :: *calcFoldOffset* code 270 ⟩ ≡

```
int TensorDimens::calcFoldOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input␣vector␣size␣in␣TensorDimens::getFoldOffset");
  int res = 0;
  int pow = 1;
  int blstart = v.size();
  for (int ibl = getSym().num() − 1; ibl ≥ 0; ibl−−) {
    int bldim = getSym()[ibl];
    if (bldim > 0) {
      blstart −= bldim;
      int blnvar = getNVX()[blstart];
      IntSequence subv(v, blstart, blstart + bldim);
      res += FTensor::getOffset(subv, blnvar) ∗ pow;
      pow ∗= FFSTensor::calcMaxOffset(blnvar, bldim);
    }
  }
  TL_RAISE_IF(blstart ≠ 0, "Error␣in␣tracing␣symmetry␣in␣TensorDimens::getFoldOffset");
  return res;
}
```

This code is used in section 266.

271. In order to find the predecessor of index within folded generally symmetric tensor, note, that a decrease action in i-th partition of symmetric indices can happen only if all indices in all subsequent partitions are zero. Then the decrease action of whole the index consists of decrease action of the first nonzero partition from the right, and setting these trailing zero partitions to their maximum indices.

So we set *iblock* to the number of last partitions. During the execution, *block_first*, and *block_last* will point to the first element of *iblock* and, first element of following block.

Then we check for all trailing zero partitions, set them to their maximums and return *iblock* to point to the first non-zero partition (or the first partition). Then for this partition, we decrease the index (fully symmetric within that partition).

⟨ **TensorDimens** :: *decrement* code 271 ⟩ ≡

```
void TensorDimens::decrement(IntSequence &v) const
{
  TL_RAISE_IF(getNVX().size() ≠ v.size(),
      "Wrong size of input/output sequence in TensorDimens::decrement");
  int iblock = getSym().num() − 1;
  int block_last = v.size();
  int block_first = block_last − getSym()[iblock];
  ⟨ check for zero trailing blocks 272 ⟩;
  ⟨ decrease the non-zero block 273 ⟩;
}
```

This code is used in section 266.

272.

⟨ check for zero trailing blocks 272 ⟩ ≡

```
while (iblock > 0 ∧ v[block_last − 1] ≡ 0) {
  for (int i = block_first; i < block_last; i++) v[i] = getNVX(i);   /* equivalent to nvs[iblock] */
  iblock−−;
  block_last = block_first;
  block_first −= getSym()[iblock];
}
```

This code is used in section 271.

273.

⟨ decrease the non-zero block 273 ⟩ ≡

```
IntSequence vtmp(v, block_first, block_last);
FTensor::decrement(vtmp, getNVX(block_first));
```

This code is used in section 271.

274. Here we go through columns of folded, calculate column of unfolded, and copy data.

⟨ **FGSTensor** conversion from **UGSTensor** 274 ⟩ ≡

```
FGSTensor::FGSTensor(const UGSTensor &ut)
: FTensor(along_col, ut.tdims.getNVX(), ut.nrows(), ut.tdims.calcFoldMaxOffset(), ut.dimen()),
      tdims(ut.tdims) {
  for (index ti = begin(); ti ≠ end(); ++ti) {
    index ui(&ut, ti.getCoor());
    copyColumn(ut, *ui, *ti);
  }
}
```

This code is used in section 266.

275. Here is the code of slicing constructor from the sparse tensor. We first calculate coordinates of first and last index of the slice within the sparse tensor (these are *lb* and *ub*), and then we iterate through all items between them (in lexicographical ordering of sparse tensor), and check whether an item is between the *lb* and *ub* in Cartesian ordering (this corresponds to belonging to the slices). If it belongs, then we subtract the lower bound *lb* to obtain coordinates in the **this** tensor and we copy the item.

⟨ **FGSTensor** slicing from **FSSparseTensor** 275 ⟩ ≡

```
FGSTensor::FGSTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
        &coor, const TensorDimens &td)
  : FTensor(along_col, td.getNVX(), t.nrows(), td.calcFoldMaxOffset(), td.dimen()), tdims(td) {
    ⟨ set lb and ub to lower and upper bounds of indices 276 ⟩;
    zeros();
    FSSparseTensor::const_iterator lbi = t.getMap().lower_bound(lb);
    FSSparseTensor::const_iterator ubi = t.getMap().upper_bound(ub);
    for (FSSparseTensor::const_iterator run = lbi; run ≠ ubi; ++run) {
      if (lb.lessEq((*run).first) ∧ (*run).first.lessEq(ub)) {
        IntSequence c((*run).first);
        c.add(−1, lb);
        Tensor::index ind(this, c);
        TL_RAISE_IF(*ind < 0 ∨ *ind ≥ ncols(),
            "Internal error in slicing constructor of FGSTensor");
        get((*run).second.first, *ind) = (*run).second.second;
      }
    }
  }
```

This code is cited in sections 277 and 365.

This code is used in section 266.

276. Here we first set *s_offsets* to offsets of partitions whose lengths are given by *ss*. So *s_offsets* is a cumulative sum of *ss*.

Then we create *lb* to be coordinates of the possibly first index from the slice, and *ub* to be coordinates of possibly last index of the slice.

⟨ set *lb* and *ub* to lower and upper bounds of indices 276 ⟩ ≡

```
IntSequence s_offsets(ss.size(), 0);
for (int i = 1; i < ss.size(); i++) s_offsets[i] = s_offsets[i − 1] + ss[i − 1];
IntSequence lb(coor.size());
IntSequence ub(coor.size());
for (int i = 0; i < coor.size(); i++) {
  lb[i] = s_offsets[coor[i]];
  ub[i] = s_offsets[coor[i]] + ss[coor[i]] − 1;
}
```

This code is cited in section 366.

This code is used in sections 275 and 277.

277. The code is similar to ⟨ **FGSTensor** slicing from **FSSparseTensor** 275 ⟩.

⟨ **FGSTensor** slicing from **FFSTensor** 277 ⟩ ≡

```
FGSTensor::FGSTensor(const FFSTensor &t, const IntSequence &ss, const IntSequence
        &coor, const TensorDimens &td)
  : FTensor(along_col, td.getNVX(), t.nrows(), td.calcFoldMaxOffset(), td.dimen()), tdims(td) {
    if (ncols() ≡ 0) return;
    ⟨ set lb and ub to lower and upper bounds of indices 276 ⟩;
    zeros();
    Tensor::index lbi(&t, lb);
    Tensor::index ubi(&t, ub);
    ++ubi;
    for (Tensor::index run = lbi; run ≠ ubi; ++run) {
      if (lb.lessEq(run.getCoor()) ∧ run.getCoor().lessEq(ub)) {
        IntSequence c(run.getCoor());
        c.add(−1, lb);
        Tensor::index ind(this, c);
        TL_RAISE_IF(∗ind < 0 ∨ ∗ind ≥ ncols(),
            "Internal␣error␣in␣slicing␣constructor␣of␣FGSTensor");
        copyColumn(t, ∗run, ∗ind);
      }
    }
  }
```

This code is used in section 266.

278.

⟨ **FGSTensor** conversion from **GSSparseTensor** 278 ⟩ ≡

```
FGSTensor::FGSTensor(const GSSparseTensor &t)
  : FTensor(along_col, t.getDims().getNVX(), t.nrows(), t.getDims().calcFoldMaxOffset(), t.dimen()),
        tdims(t.getDims()) {
    zeros();
    for (FSSparseTensor::const_iterator it = t.getMap().begin(); it ≠ t.getMap().end(); ++it) {
      index ind(this, (∗it).first);
      get((∗it).second.first, ∗ind) = (∗it).second.second;
    }
  }
```

This code is used in section 266.

279. First we increment as unfolded, then we must monotonize within partitions defined by the symmetry. This is done by **IntSequence**::*pmonotone*.

⟨ **FGSTensor**::*increment* code 279 ⟩ ≡

```
void FGSTensor::increment(IntSequence &v) const
  {
    TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣FGSTensor::increment");
    UTensor::increment(v, tdims.getNVX());
    v.pmonotone(tdims.getSym());
  }
```

This code is used in section 266.

280. Return unfolded version of the tensor.

⟨ **FGSTensor** :: *unfold* code 280 ⟩ ≡

```
UTensor &FGSTensor::unfold() const
{
  return *(new UGSTensor(*this));
}
```

This code is used in section 266.

281. Here we implement the contraction

$$[r_{x^iz^k}]_{\alpha_1\ldots\alpha_i\gamma_1\ldots\gamma_k} = \left[t_{x^iy^jz^k}\right]_{\alpha_1\ldots\alpha_i\beta_1\ldots\beta_j\gamma_1\ldots\gamma_k} [c]^{\beta_1\ldots\beta_j}$$

More generally, x^i and z^k can represent also general symmetries.

The operation can be rewritten as a matrix product

$$\left[t_{x^iy^jz^k}\right] \cdot (I_l \otimes c \otimes I_r)$$

where l is a number of columns in tensor with symmetry on the left (i.e. x^i), and r is a number of columns in tensor with a symmetry on the right (i.e. z^k). The code proceeds accordingly. We first form two symmetries *sym_left* and *sym_right*, then calculate the number of columns *dleft*$= l$ and *dright*$= r$, form the Kronecker product and multiply and add.

The input parameter i is the order of a variable being contracted starting from 0.

⟨ **FGSTensor** :: *contractAndAdd* code 281 ⟩ ≡

```
void FGSTensor::contractAndAdd(int i, FGSTensor &out, const FRSingleTensor &col) const
{
  TL_RAISE_IF(i < 0 ∨ i ≥ getSym().num(), "Wrong␣index␣for␣FGSTensor::contractAndAdd");
  TL_RAISE_IF(getSym()[i] ≠ col.dimen() ∨ tdims.getNVS()[i] ≠ col.nvar(),
      "Wrong␣dimensions␣for␣FGSTensor::contractAndAdd");
  ⟨ set sym_left and sym_right to symmetries around i 282 ⟩;
  int dleft = TensorDimens(sym_left, tdims.getNVS()).calcFoldMaxOffset();
  int dright = TensorDimens(sym_right, tdims.getNVS()).calcFoldMaxOffset();
  KronProdAll kp(3);
  kp.setUnit(0, dleft);
  kp.setMat(1, col);
  kp.setUnit(2, dright);
  FGSTensor tmp(out.nrows(), out.getDims());
  kp.mult(*this, tmp);
  out.add(1.0, tmp);
}
```

This code is cited in section 291.

This code is used in section 266.

282. Here we have a symmetry of **this** tensor and we have to set *sym_left* to the subsymmetry left from the i-th variable and *sym_right* to the subsymmetry right from the i-th variable. So we copy first all the symmetry and then put zeros to the left for *sym_right* and to the right for *sym_left*.

⟨ set *sym_left* and *sym_right* to symmetries around i 282 ⟩ ≡
```
Symmetry sym_left(getSym());
Symmetry sym_right(getSym());
for (int j = 0; j < getSym().num(); j++) {
  if (j ≤ i) sym_right[j] = 0;
  if (j ≥ i) sym_left[j] = 0;
}
```
This code is used in sections 281 and 291.

283. Here we go through folded tensor, and each index we convert to index of the unfolded tensor and copy the data to the unfolded. Then we unfold data within the unfolded tensor.

⟨ **UGSTensor** conversion from **FGSTensor** 283 ⟩ ≡
```
UGSTensor::UGSTensor(const FGSTensor &ft)
: UTensor(along_col, ft.tdims.getNVX(), ft.nrows(), ft.tdims.calcUnfoldMaxOffset(), ft.dimen()),
      tdims(ft.tdims) {
  for (index fi = ft.begin(); fi ≠ ft.end(); ++fi) {
    index ui(this, fi.getCoor());
    copyColumn(ft, *fi, *ui);
  }
  unfoldData();
}
```
This code is used in section 266.

284. This makes a folded slice from the sparse tensor and unfolds it.

⟨ **UGSTensor** slicing from **FSSparseTensor** 284 ⟩ ≡
```
UGSTensor::UGSTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
        &coor, const TensorDimens &td)
: UTensor(along_col, td.getNVX(), t.nrows(), td.calcUnfoldMaxOffset(), td.dimen()), tdims(td) {
  if (ncols() ≡ 0) return;
  FGSTensor ft(t, ss, coor, td);
  for (index fi = ft.begin(); fi ≠ ft.end(); ++fi) {
    index ui(this, fi.getCoor());
    copyColumn(ft, *fi, *ui);
  }
  unfoldData();
}
```
This code is used in section 266.

285. This makes a folded slice from dense and unfolds it.

⟨ **UGSTensor** slicing from **UFSTensor** 285 ⟩ ≡

```
UGSTensor::UGSTensor(const UFSTensor &t, const IntSequence &ss, const IntSequence
        &coor, const TensorDimens &td)
  : UTensor(along_col, td.getNVX(), t.nrows(), td.calcUnfoldMaxOffset(), td.dimen()), tdims(td) {
    FFSTensor folded(t);
    FGSTensor ft(folded, ss, coor, td);
    for (index fi = ft.begin(); fi ≠ ft.end(); ++fi) {
      index ui(this, fi.getCoor());
      copyColumn(ft, *fi, *ui);
    }
    unfoldData();
  }
```

This code is used in section 266.

286. Clear, just call **UTensor** static methods.

⟨ **UGSTensor** increment and decrement codes 286 ⟩ ≡

```
void UGSTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣UGSTensor::increment");
  UTensor::increment(v, tdims.getNVX());
}
void UGSTensor::decrement(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣UGSTensor::decrement");
  UTensor::decrement(v, tdims.getNVX());
}
```

This code is used in section 266.

287. Return a new instance of folded version.

⟨ **UGSTensor**::*fold* code 287 ⟩ ≡

```
FTensor &UGSTensor::fold() const
{
  return *(new FGSTensor(*this));
}
```

This code is used in section 266.

288. Return an offset of a given index.

⟨ **UGSTensor**::*getOffset* code 288 ⟩ ≡

```
int UGSTensor::getOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input␣vector␣size␣in␣UGSTensor::getOffset");
  return UTensor::getOffset(v, tdims.getNVX());
}
```

This code is used in section 266.

289. Unfold all data. We go through all the columns and for each we obtain an index of the first equivalent, and copy the data.

⟨ **UGSTensor** :: *unfoldData* code 289 ⟩ ≡

```
void UGSTensor::unfoldData()
{
  for (index in = begin(); in ≠ end(); ++in) copyColumn(*(getFirstIndexOf(in)), *in);
}
```

This code is used in section 266.

290. Here we return the first index which is equivalent in the symmetry to the given index. It is a matter of sorting all the symmetry partitions of the index.

⟨ **UGSTensor** :: *getFirstIndexOf* code 290 ⟩ ≡

```
Tensor::index UGSTensor::getFirstIndexOf(const index &in) const
{
  IntSequence v(in.getCoor());
  int last = 0;
  for (int i = 0; i < tdims.getSym().num(); i++) {
    IntSequence vtmp(v, last, last + tdims.getSym()[i]);
    vtmp.sort();
    last += tdims.getSym()[i];
  }
  return index(this, v);
}
```

This code is used in section 266.

291. Here is perfectly same code with the same semantics as in ⟨ **FGSTensor** :: *contractAndAdd* code 281 ⟩.

⟨ **UGSTensor** :: *contractAndAdd* code 291 ⟩ ≡

```
void UGSTensor::contractAndAdd(int i, UGSTensor &out, const URSingleTensor &col) const
{
  TL_RAISE_IF(i < 0 ∨ i ≥ getSym().num(), "Wrong␣index␣for␣UGSTensor::contractAndAdd");
  TL_RAISE_IF(getSym()[i] ≠ col.dimen() ∨ tdims.getNVS()[i] ≠ col.nvar(),
      "Wrong␣dimensions␣for␣UGSTensor::contractAndAdd");
  ⟨ set sym_left and sym_right to symmetries around i 282 ⟩;
  int dleft = TensorDimens(sym_left, tdims.getNVS()).calcUnfoldMaxOffset();
  int dright = TensorDimens(sym_right, tdims.getNVS()).calcUnfoldMaxOffset();
  KronProdAll kp(3);
  kp.setUnit(0, dleft);
  kp.setMat(1, col);
  kp.setUnit(2, dright);
  UGSTensor tmp(out.nrows(), out.getDims());
  kp.mult(*this, tmp);
  out.add(1.0, tmp);
}
```

This code is used in section 266.

292. End of `gs_tensor.cpp` file.

293. Row-wise full symmetry tensor. Start of `rfs_tensor.h` file.

Here we define classes for full symmetry tensors with the multidimensional index identified with rows. The primary usage is for storage of data coming from (or from a sum of)

$$\prod_{m=1}^{l}\left[g_{s^{|c_m|}}\right]^{\gamma_m}_{c_m(\alpha)}$$

where α coming from a multidimensional index go through some set S and c is some equivalence. So we model a tensor of the form:

$$\left[\prod_{m=1}^{l}\left[g_{s^{|c_m|}}\right]^{\gamma_m}_{c_m(\alpha)}\right]_S^{\gamma_1\cdots\gamma_l}$$

Since all $\gamma_1,\ldots,\gamma_l$ correspond to the same variable, the tensor is fully symmetric. The set of indices S cannot be very large and sometimes it is only one element. This case is handled in a special subclass.

We provide both folded and unfolded versions. Their logic is perfectly the same as in **UFSTensor** and **FFSTensor** with two exceptions. One has been already mentioned, the multidimensional index is along the rows. The second are conversions between the two types. Since this kind of tensor is used to multiply (from the right) a tensor whose multidimensional index is identified with columns, we will need a different way of a conversion. If the multiplication of two folded tensors is to be equivalent with multiplication of two unfolded, the folding of the right tensor must sum all equivalent elements since they are multiplied with the same number from the folded tensor. (Equivalent here means all elements of unfolded tensor corresponding to one element in folded tensor.) For this reason, it is necessary to calculate a column number from the given sequence, so we implement *getOffset*. Process of unfolding is not used, so we implemented it so that unfolding and then folding a tensor would yield the same data.

```
#ifndef RFS_TENSOR_H
#define RFS_TENSOR_H
#include "tensor.h"
#include "fs_tensor.h"
#include "symmetry.h"
  ⟨ URTensor class declaration 294 ⟩;
  ⟨ FRTensor class declaration 296 ⟩;
  ⟨ URSingleTensor class declaration 298 ⟩;
  ⟨ FRSingleTensor class declaration 299 ⟩;
#endif
```

294. This is straightforward and very similar to **UFSTensor**.

⟨ **URTensor** class declaration 294 ⟩ ≡

```
class FRTensor;
class URTensor : public UTensor {
  int nv;
public:
  ⟨ URTensor constructor declaration 295 ⟩;
  virtual ~URTensor() { }
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  FTensor &fold() const;
  int getOffset(const IntSequence &v) const;
  int nvar() const
  { return nv; }
  Symmetry getSym() const
  { return Symmetry(dimen()); }
};
```

This code is cited in section 1.

This code is used in section 293.

295.

⟨ **URTensor** constructor declaration 295 ⟩ ≡

```
URTensor(int c, int nvar, int d)
: UTensor(along_row, IntSequence(d, nvar), UFSTensor::calcMaxOffset(nvar, d), c, d), nv(nvar) { }
URTensor(const URTensor &ut)
: UTensor(ut), nv(ut.nv) { }
URTensor(const FRTensor &ft);
```

This code is used in section 294.

296. This is straightforward and very similar to **FFSTensor**.

⟨ **FRTensor** class declaration 296 ⟩ ≡

```
class FRTensor : public FTensor {
  int nv;
public:
  ⟨ FRTensor constructor declaration 297 ⟩;
  virtual ~FRTensor() { }
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  UTensor &unfold() const;
  int nvar() const
  { return nv; }
  int getOffset(const IntSequence &v) const
  { return FTensor::getOffset(v, nv); }
  Symmetry getSym() const
  { return Symmetry(dimen()); }
};
```

This code is cited in section 1.

This code is used in section 293.

297.

⟨ **FRTensor** constructor declaration 297 ⟩ ≡

```
FRTensor(int c, int nvar, int d)
: FTensor(along_row, IntSequence(d, nvar), FFSTensor::calcMaxOffset(nvar, d), c, d), nv(nvar) { }
FRTensor(const FRTensor &ft)
: FTensor(ft), nv(ft.nv) { }
FRTensor(const URTensor &ut);
```

This code is used in section 296.

298. The following class represents specialization of **URTensor** coming from Kronecker multiplication of a few vectors. So the resulting row-oriented tensor has one column. We provide two constructors, one constructs the tensor from a few vectors stored as **vector**⟨**ConstVector**⟩. The second makes the Kronecker power of one given vector.

⟨ **URSingleTensor** class declaration 298 ⟩ ≡

```
class URSingleTensor : public URTensor {
public:
  URSingleTensor(int nvar, int d)
  : URTensor(1, nvar, d) { }
  URSingleTensor(const vector<ConstVector> &cols);
  URSingleTensor(const ConstVector &v, int d);
  URSingleTensor(const URSingleTensor &ut)
  : URTensor(ut) { }
  virtual ~URSingleTensor() { }
  FTensor &fold() const;
};
```

This code is cited in section 1.

This code is used in section 293.

299. This class represents one column row-oriented tensor. The only way how to construct it is from the **URSingleTensor** or from the scratch. The folding algorithm is the same as folding of general **URTensor**. Only its implementation is different, since we do not copy rows, but only elements.

⟨ **FRSingleTensor** class declaration 299 ⟩ ≡

```
class FRSingleTensor : public FRTensor {
public:
  FRSingleTensor(int nvar, int d)
  : FRTensor(1, nvar, d) { }
  FRSingleTensor(const URSingleTensor &ut);
  FRSingleTensor(const FRSingleTensor &ft)
  : FRTensor(ft) { }
  virtual ~FRSingleTensor() { }
};
```

This code is cited in section 1.

This code is used in section 293.

300. End of `rfs_tensor.h` file.

301. Start of `rfs_tensor.cpp` file.

```
#include "rfs_tensor.h"
#include "kron_prod.h"
#include "tl_exception.h"
  ⟨ FRTensor conversion from unfolded 302 ⟩;
  ⟨ FRTensor::unfold code 303 ⟩;
  ⟨ FRTensor::increment code 304 ⟩;
  ⟨ FRTensor::decrement code 305 ⟩;
  ⟨ URTensor conversion from folded 306 ⟩;
  ⟨ URTensor::fold code 307 ⟩;
  ⟨ URTensor increment and decrement 308 ⟩;
  ⟨ URTensor::getOffset code 309 ⟩;
  ⟨ URSingleTensor constructor 1 code 310 ⟩;
  ⟨ URSingleTensor constructor 2 code 311 ⟩;
  ⟨ URSingleTensor::fold code 312 ⟩;
  ⟨ FRSingleTensor conversion from unfolded 313 ⟩;
```

302. The conversion from unfolded to folded sums up all data from unfolded corresponding to one folded index. So we go through all the rows in the unfolded tensor *ut*, make an index of the folded tensor by sorting the coordinates, and add the row.

```
⟨ FRTensor conversion from unfolded 302 ⟩ ≡
  FRTensor::FRTensor(const URTensor &ut)
  : FTensor(along_row, IntSequence(ut.dimen(), ut.nvar()), FFSTensor::calcMaxOffset(ut.nvar(),
        ut.dimen()), ut.ncols(), ut.dimen()), nv(ut.nvar()) {
    zeros();
    for (index in = ut.begin(); in ≠ ut.end(); ++in) {
      IntSequence vtmp(in.getCoor());
      vtmp.sort();
      index tar(this, vtmp);
      addRow(ut, *in, *tar);
    }
  }
```

This code is used in section 301.

303. Here just make a new instance and return the reference.

```
⟨ FRTensor::unfold code 303 ⟩ ≡
  UTensor &FRTensor::unfold() const
  {
    return *(new URTensor(*this));
  }
```

This code is used in section 301.

304. Incrementing is easy. The same as for **FFSTensor**.

⟨ **FRTensor** :: *increment* code 304 ⟩ ≡

```
void FRTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣FRTensor::increment");
  UTensor::increment(v, nv);
  v.monotone();
}
```

This code is used in section 301.

305. Decrement calls static **FTensor** :: *decrement*.

⟨ **FRTensor** :: *decrement* code 305 ⟩ ≡

```
void FRTensor::decrement(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣input/output␣vector␣size␣in␣FRTensor::decrement");
  FTensor::decrement(v, nv);
}
```

This code is used in section 301.

306. Here we convert folded full symmetry tensor to unfolded. We copy all columns of folded tensor to unfolded and leave other columns (duplicates) zero. In this way, if the unfolded tensor is folded back, we should get the same data.

⟨ **URTensor** conversion from folded 306 ⟩ ≡

```
URTensor::URTensor(const FRTensor &ft)
: UTensor(along_row, IntSequence(ft.dimen(), ft.nvar()), UFSTensor::calcMaxOffset(ft.nvar(),
      ft.dimen()), ft.ncols(), ft.dimen()), nv(ft.nvar()) {
  zeros();
  for (index src = ft.begin(); src ≠ ft.end(); ++src) {
    index in(this, src.getCoor());
    copyRow(ft, *src, *in);
  }
}
```

This code is used in section 301.

307. Here we just return a reference to new instance of folded tensor.

⟨ **URTensor** :: *fold* code 307 ⟩ ≡

```
FTensor &URTensor::fold() const
{
  return *(new FRTensor(*this));
}
```

This code is used in section 301.

308. Here we just call **UTensor** respective static methods.

⟨ **URTensor** increment and decrement 308 ⟩ ≡

```
void URTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input/output vector size in URTensor::increment");
  UTensor::increment(v, nv);
}
void URTensor::decrement(IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input/output vector size in URTensor::decrement");
  UTensor::decrement(v, nv);
}
```

This code is used in section 301.

309.

⟨ **URTensor**::*getOffset* code 309 ⟩ ≡

```
int URTensor::getOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input vector size in URTensor::getOffset");
  return UTensor::getOffset(v, nv);
}
```

This code is used in section 301.

310. Here we construct $v_1 \otimes v_2 \otimes \ldots \otimes v_n$, where $v_1, v_2, \ldots, v_n$ are stored in **vector**⟨**ConstVector**⟩.

⟨ **URSingleTensor** constructor 1 code 310 ⟩ ≡

```
URSingleTensor::URSingleTensor(const vector<ConstVector> &cols)
: URTensor(1, cols[0].length(), cols.size()) {
  if (dimen() == 1) {
    getData() = cols[0];
    return;
  }
  Vector *last = new Vector(cols[cols.size() - 1]);
  for (int i = cols.size() - 2; i > 0; i--) {
    Vector *newlast = new Vector(Tensor::power(nvar(), cols.size() - i));
    KronProd::kronMult(cols[i], ConstVector(*last), *newlast);
    delete last;
    last = newlast;
  }
  KronProd::kronMult(cols[0], ConstVector(*last), getData());
  delete last;
}
```

This code is used in section 301.

311. Here we construct $v \otimes \ldots \otimes v$, where the number of v copies is d.

⟨ **URSingleTensor** constructor 2 code 311 ⟩ ≡

```
URSingleTensor::URSingleTensor(const ConstVector &v, int d)
: URTensor(1, v.length(), d) {
  if (d ≡ 1) {
    getData() = v;
    return;
  }
  Vector *last = new Vector(v);
  for (int i = d − 2; i > 0; i−−) {
    Vector *newlast = new Vector(last→length() * v.length());
    KronProd::kronMult(v, ConstVector(*last), *newlast);
    delete last;
    last = newlast;
  }
  KronProd::kronMult(v, ConstVector(*last), getData());
  delete last;
}
```

This code is used in section 301.

312. Here we construct **FRSingleTensor** from **URSingleTensor** and return its reference.

⟨ **URSingleTensor**::*fold* code 312 ⟩ ≡

```
FTensor &URSingleTensor::fold() const
{
  return *(new FRSingleTensor(*this));
}
```

This code is used in section 301.

313. The conversion from unfolded **URSingleTensor** to folded **FRSingleTensor** is completely the same as conversion from **URTensor** to **FRTensor**, only we do not copy rows but elements.

⟨ **FRSingleTensor** conversion from unfolded 313 ⟩ ≡

```
FRSingleTensor::FRSingleTensor(const URSingleTensor &ut)
: FRTensor(1, ut.nvar(), ut.dimen()) {
  zeros();
  for (index in = ut.begin(); in ≠ ut.end(); ++in) {
    IntSequence vtmp(in.getCoor());
    vtmp.sort();
    index tar(this, vtmp);
    get(*tar, 0) += ut.get(*in, 0);
  }
}
```

This code is used in section 301.

314. End of `rfs_tensor.cpp` file.

315. Even more general symmetry tensor. Start of `ps_tensor.h` file.

Here we define an abstraction for a tensor, which has a general symmetry, but the symmetry is not of what is modelled by **Symmetry**. This kind of tensor comes to existence when we evaluate something like:

$$\left[B_{y^2u^3}\right]_{\alpha_1\alpha_2\beta_1\beta_2\beta_3} = \cdots + \left[g_{y^3}\right]_{\gamma_1\gamma_2\gamma_3} \left[g_{yu}\right]^{\gamma_1}_{\alpha_1\beta_3} \left[g_{yu}\right]^{\gamma_2}_{\alpha_2\beta_1} \left[g_u\right]^{\gamma_3}_{\beta_2} + \cdots$$

If the tensors are unfolded, we obtain a tensor

$$g_{y^3} \cdot (g_{yu} \otimes g_{yu} \otimes g_u)$$

Obviously, this tensor can have a symmetry not compatible with ordering $\alpha_1\alpha_2\beta_1\beta_2\beta_3$, (in other words, not compatible with symmetry y^2u^3). In fact, the indices are permuted.

This kind of tensor must be added to $\left[B_{y^2u^3}\right]$. Its dimensions are the same as of $\left[B_{y^2u^3}\right]$, but some coordinates are permuted. The addition is the only action we need to do with the tensor.

Another application where this permuted symmetry tensor appears is a slice of a fully symmetric tensor. If the symmetric dimension of the tensor is partitioned to continuous parts, and we are interested only in data with a given symmetry (permuted) of the partitions, then we have the permuted symmetry tensor. For instance, if x is partitioned $x = [a, b, c, d]$, and having tensor $[f_{x^3}]$, one can d a slice (subtensor) $[f_{aca}]$. The data of this tensor are permuted of $[f_{a^c}]$.

Here we also define the folded version of permuted symmetry tensor. It has permuted symmetry and is partially folded. One can imagine it as a product of a few dimensions, each of them is folded and having a few variables. The underlying variables are permuted. The product of such dimensions is described by **PerTensorDimens2**. The tensor holding the underlying data is **FPSTensor**.

```
#ifndef PS_TENSOR_H
#define PS_TENSOR_H
#include "tensor.h"
#include "gs_tensor.h"
#include "equivalence.h"
#include "permutation.h"
#include "kron_prod.h"
#include "sparse_tensor.h"
  ⟨ SortIntSequence class declaration 316 ⟩;
  ⟨ PerTensorDimens class declaration 317 ⟩;
  ⟨ UPSTensor class declaration 318 ⟩;
  ⟨ PerTensorDimens2 class declaration 320 ⟩;
  ⟨ FPSTensor class declaration 321 ⟩;
#endif
```

316. This is just a helper class for ordering a sequence on call stack.

```
⟨ SortIntSequence class declaration 316 ⟩ ≡
  class SortIntSequence : public IntSequence {
  public:
    SortIntSequence(const IntSequence &s)
    : IntSequence(s) { sort(); }
  };
```

This code is used in section 315.

317. Here we declare a class describing dimensions of permuted symmetry tensor. It inherits from **TensorDimens** and adds a permutation which permutes *nvmax*. It has two constructors, each corresponds to a context where the tensor appears.

The first constructor calculates the permutation from a given equivalence.

The second constructor corresponds to dimensions of a slice. Let us take $[f_{aca}]$ as an example. First it calculates **TensorDimens** of $[f_{a^c}]$, then it calculates a permutation corresponding to ordering of *aca* to a^2c, and applies this permutation on the dimensions as the first constructor. The constructor takes only stack sizes (lengths of a, b, c, and d), and coordinates of picked partitions.

Note that inherited methods *calcUnfoldColumns* and *calcFoldColumns* work, since number of columns is independent on the permutation, and *calcFoldColumns* does not use changed *nvmax*, it uses *nvs*, so it is OK.

⟨ **PerTensorDimens** class declaration 317 ⟩ ≡

```
class PerTensorDimens : public TensorDimens {
protected:
  Permutation per;
public:
  PerTensorDimens(const Symmetry &s, const IntSequence &nvars, const Equivalence &e)
  : TensorDimens(s, nvars), per(e) { per.apply(nvmax); }
  PerTensorDimens(const TensorDimens &td, const Equivalence &e)
  : TensorDimens(td), per(e) { per.apply(nvmax); }
  PerTensorDimens(const TensorDimens &td, const Permutation &p)
  : TensorDimens(td), per(p) { per.apply(nvmax); }
  PerTensorDimens(const IntSequence &ss, const IntSequence &coor)
  : TensorDimens(ss, SortIntSequence(coor)), per(coor) { per.apply(nvmax); }
  PerTensorDimens(const PerTensorDimens &td)
  : TensorDimens(td), per(td.per) { }
  const PerTensorDimens &operator=(const PerTensorDimens &td)
  { TensorDimens::operator=(td); per = td.per; return *this; }
  bool operator≡(const PerTensorDimens &td)
  { return TensorDimens::operator≡(td) ∧ per ≡ td.per; }
  int tailIdentity() const
  { return per.tailIdentity(); }
  const Permutation &getPer() const
  { return per; }
};
```

This code is used in section 315.

318. Here we declare the permuted symmetry unfolded tensor. It has **PerTensorDimens** as a member. It inherits from **UTensor** which requires to implement *fold* method. There is no folded counterpart, so in our implementation we raise unconditional exception, and return some dummy object (just to make it compilable without warnings).

The class has two sorts of constructors corresponding to a context where it appears. The first constructs object from a given matrix, and Kronecker product. Within the constructor, all the calculations are performed. Also we need to define dimensions, these are the same of the resulting matrix (in our example $\left[B_{y^2u^3}\right]$) but permuted. The permutation is done in **PerTensorDimens** constructor.

The second type of constructor is slicing. It makes a slice from **FSSparseTensor**. The slice is given by stack sizes, and coordinates of picked stacks.

There are two algorithms for filling a slice of a sparse tensor. The first *fillFromSparseOne* works well for more dense tensors, the second *fillFromSparseTwo* is better for very sparse tensors. We provide a static method, which decides what of the two algorithms is better.

⟨ **UPSTensor** class declaration 318 ⟩ ≡

```
class UPSTensor : public UTensor {
  const PerTensorDimens tdims;
public:
  ⟨ UPSTensor constructors from Kronecker product 319 ⟩;
  UPSTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
      &coor, const PerTensorDimens &ptd);
  UPSTensor(const UPSTensor &ut)
  : UTensor(ut), tdims(ut.tdims) { }
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  FTensor &fold() const;
  int getOffset(const IntSequence &v) const;
  void addTo(FGSTensor &out) const;
  void addTo(UGSTensor &out) const;
  enum fill_method {
    first, second
  };
  static fill_method decideFillMethod(const FSSparseTensor &t);
private:
  int tailIdentitySize() const;
  void fillFromSparseOne(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
      &coor);
  void fillFromSparseTwo(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
      &coor);
};
```

This code is cited in section 1.

This code is used in section 315.

319. Here we have four constructors making an **UPSTensor** from a product of matrix and Kronecker product. The first constructs the tensor from equivalence classes of the given equivalence in an order given by the equivalence. The second does the same but with optimized **KronProdAllOptim**, which has a different order of matrices than given by the classes in the equivalence. This permutation is projected to the permutation of the **UPSTensor**. The third, is the same as the first, but the classes of the equivalence are permuted by the given permutation. Finally, the fourth is the most general combination. It allows for a permutation of equivalence classes, and for optimized **KronProdAllOptim**, which permutes the permuted equivalence classes.

⟨ **UPSTensor** constructors from Kronecker product 319 ⟩ ≡

```
UPSTensor(const TensorDimens &td, const Equivalence &e, const ConstTwoDMatrix
        &a, const KronProdAll &kp)
  : UTensor(along_col, PerTensorDimens(td, e).getNVX(), a.nrows(), kp.ncols(), td.dimen()),
      tdims(td, e) { kp.mult(a, *this); }
UPSTensor(const TensorDimens &td, const Equivalence &e, const ConstTwoDMatrix
        &a, const KronProdAllOptim &kp)
  : UTensor(along_col, PerTensorDimens(td, Permutation(e, kp.getPer())).getNVX(), a.nrows(),
      kp.ncols(), td.dimen()), tdims(td, Permutation(e, kp.getPer())) { kp.mult(a, *this); }
UPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation &p, const
        ConstTwoDMatrix &a, const KronProdAll &kp)
  : UTensor(along_col, PerTensorDimens(td, Permutation(e, p)).getNVX(), a.nrows(), kp.ncols(),
      td.dimen()), tdims(td, Permutation(e, p)) { kp.mult(a, *this); }
UPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation &p, const
        ConstTwoDMatrix &a, const KronProdAllOptim &kp)
  : UTensor(along_col, PerTensorDimens(td,
      Permutation(e, Permutation(p, kp.getPer()))).getNVX(), a.nrows(), kp.ncols(), td.dimen()),
      tdims(td, Permutation(e, Permutation(p, kp.getPer()))) { kp.mult(a, *this); }
```

This code is used in section 318.

320. Here we define an abstraction for the tensor dimension with the symmetry like $xuv|uv|xu|y|y|x|x|y$. These symmetries come as induces symmetries of equivalence and some outer symmetry. Thus the underlying variables are permuted. One can imagine the dimensions as an unfolded product of dimensions which consist of folded products of variables.

We inherit from **PerTensorDimens** since we need the permutation implied by the equivalence. The new member are the induced symmetries (symmetries of each folded dimensions) and *ds* which are sizes of the dimensions. The number of folded dimensions is return by *numSyms*.

The object is constructed from outer tensor dimensions and from equivalence with optionally permuted classes.

⟨ **PerTensorDimens2** class declaration 320 ⟩ ≡

```
class PerTensorDimens2 : public PerTensorDimens {
  InducedSymmetries syms;
  IntSequence ds;
public:
  PerTensorDimens2(const TensorDimens &td, const Equivalence &e, const Permutation &p)
  : PerTensorDimens(td, Permutation(e, p)), syms(e, p, td.getSym()), ds(syms.size()) {
    setDimensionSizes(); }
  PerTensorDimens2(const TensorDimens &td, const Equivalence &e)
  : PerTensorDimens(td, e), syms(e, td.getSym()), ds(syms.size()) { setDimensionSizes(); }
  int numSyms() const
  { return (int) syms.size(); }
  const Symmetry &getSym(int i) const
  { return syms[i]; }
  int calcMaxOffset() const
  { return ds.mult(); }
  int calcOffset(const IntSequence &coor) const;
  void print() const;
protected:
  void setDimensionSizes();
};
```

This code is used in section 315.

321. Here we define an abstraction of the permuted symmetry folded tensor. It is needed in context of the Faa Di Bruno formula for folded stack container multiplied with container of dense folded tensors, or multiplied by one full symmetry sparse tensor.

For example, if we perform the Faa Di Bruno for $F = f(z)$, where $z = [g(x, y, u, v), h(x, y, u), x, y]^T$, we get for one concrete equivalence:

$$\left[F_{x^4y^3u^3v^2}\right] = \ldots + \left[f_{g^2h^2x^2y}\right]\left([g]_{xv} \otimes [g]_{u^2v} \otimes [h]_{xu} \otimes [h]_{y^2} \otimes \left[[I]_x \otimes [I]_x\right] \otimes \left[[I]_y\right]\right) + \ldots$$

The class **FPSTensor** represents the tensor at the right. Its dimension corresponds to a product of 7 dimensions with the following symmetries: $xv|u^v|xu|y^2|x|x|y$. Such the dimension is described by **PerTensorDimens2**.

The tensor is constructed in a context of stack container multiplication, so, it is constructed from dimensions *td* (dimensions of the output tensor), stack product *sp* (implied symmetries picking tensors from a stack container, here it is z), then a sorted integer sequence of the picked stacks of the stack product (it is always sorted, here it is $(0, 0, 1, 1, 2, 2, 3)$), then the tensor $\left[f_{g^2h^2x^2y}\right]$ (its symmetry must be the same as symmetry given by the *istacks*), and finally from the equivalence with permuted classes.

We implement *increment* and *getOffset* methods, *decrement* and *unfold* raise an exception. Also, we implement *addTo* method, which adds the tensor data (partially unfolded) to folded general symmetry tensor.

⟨ **FPSTensor** class declaration 321 ⟩ ≡

```
template⟨typename _Ttype⟩ class StackProduct;
class FPSTensor : public FTensor {
  const PerTensorDimens2 tdims;
public:
  ⟨ FPSTensor constructors 322 ⟩;
  void increment(IntSequence &v) const;
  void decrement(IntSequence &v) const;
  UTensor &unfold( ) const;
  int getOffset(const IntSequence &v) const;
  void addTo(FGSTensor &out) const;
};
```

This code is cited in section 1.

This code is used in section 315.

322. As for **UPSTensor**, we provide four constructors allowing for combinations of permuting equivalence classes, and optimization of **KronProdAllOptim**. These constructors multiply with dense general symmetry tensor (coming from the dense container, or as a dense slice of the full symmetry sparse tensor). In addition to these 4 constructors, we have one constructor multiplying with general symmetry sparse tensor (coming as a sparse slice of the full symmetry sparse tensor).

⟨ **FPSTensor** constructors 322 ⟩ ≡

```
FPSTensor(const TensorDimens &td, const Equivalence &e, const ConstTwoDMatrix
        &a, const KronProdAll &kp)
  : FTensor(along_col, PerTensorDimens(td, e).getNVX(), a.nrows(), kp.ncols(), td.dimen()),
      tdims(td, e) { kp.mult(a, *this); }

FPSTensor(const TensorDimens &td, const Equivalence &e, const ConstTwoDMatrix
        &a, const KronProdAllOptim &kp)
  : FTensor(along_col, PerTensorDimens(td, Permutation(e, kp.getPer())).getNVX(), a.nrows(),
      kp.ncols(), td.dimen()), tdims(td, e, kp.getPer()) { kp.mult(a, *this); }

FPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation &p, const
        ConstTwoDMatrix &a, const KronProdAll &kp)
  : FTensor(along_col, PerTensorDimens(td, Permutation(e, p)).getNVX(), a.nrows(), kp.ncols(),
      td.dimen()), tdims(td, e, p) { kp.mult(a, *this); }

FPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation &p, const
        ConstTwoDMatrix &a, const KronProdAllOptim &kp)
  : FTensor(along_col, PerTensorDimens(td,
      Permutation(e, Permutation(p, kp.getPer()))).getNVX(), a.nrows(), kp.ncols(), td.dimen()),
      tdims(td, e, Permutation(p, kp.getPer())) { kp.mult(a, *this); }

FPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation &p, const
    GSSparseTensor &t, const KronProdAll &kp);

FPSTensor(const FPSTensor &ft)
  : FTensor(ft), tdims(ft.tdims) { }
```

This code is used in section 321.

323. End of `ps_tensor.h` file.

324. Start of `ps_tensor.cpp` file.

```
#include "ps_tensor.h"
#include "fs_tensor.h"
#include "tl_exception.h"
#include "tl_static.h"
#include "stack_container.h"
  ⟨ UPSTensor :: decideFillMethod code 325 ⟩;
  ⟨ UPSTensor slicing constructor code 326 ⟩;
  ⟨ UPSTensor increment and decrement 327 ⟩;
  ⟨ UPSTensor :: fold code 328 ⟩;
  ⟨ UPSTensor :: getOffset code 329 ⟩;
  ⟨ UPSTensor :: addTo folded code 330 ⟩;
  ⟨ UPSTensor :: addTo unfolded code 331 ⟩;
  ⟨ UPSTensor :: tailIdentitySize code 332 ⟩;
  ⟨ UPSTensor :: fillFromSparseOne code 333 ⟩;
  ⟨ UPSTensor :: fillFromSparseTwo code 334 ⟩;
  ⟨ PerTensorDimens2 :: setDimensionSizes code 335 ⟩;
  ⟨ PerTensorDimens2 :: calcOffset code 336 ⟩;
  ⟨ PerTensorDimens2 :: print code 337 ⟩;
  ⟨ FPSTensor :: increment code 338 ⟩;
  ⟨ FPSTensor :: decrement code 339 ⟩;
  ⟨ FPSTensor :: unfold code 340 ⟩;
  ⟨ FPSTensor :: getOffset code 341 ⟩;
  ⟨ FPSTensor :: addTo code 342 ⟩;
  ⟨ FPSTensor sparse constructor 343 ⟩;
```

325. Here we decide, what method for filling a slice in slicing constructor to use. A few experiments suggest, that if the tensor is more than 8% filled, the first method (*fillFromSparseOne*) is better. For fill factors less than 1%, the second can be 3 times quicker.

```
⟨ UPSTensor :: decideFillMethod code 325 ⟩ ≡
  UPSTensor :: fill_method UPSTensor :: decideFillMethod(const FSSparseTensor &t)
  {
    if (t.getFillFactor() > 0.08) return first;
    else return second;
  }
```

This code is used in section 324.

326. Here we make a slice. We decide what fill method to use and set it.

```
⟨ UPSTensor slicing constructor code 326 ⟩ ≡
  UPSTensor :: UPSTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
        &coor, const PerTensorDimens &ptd)
  : UTensor(along_col, ptd.getNVX(), t.nrows(), ptd.calcUnfoldMaxOffset(), ptd.dimen()), tdims(ptd) {
    TL_RAISE_IF(coor.size() ≠ t.dimen(),
        "Wrong␣coordinates␣length␣of␣stacks␣for␣UPSTensor␣slicing␣constructor");
    TL_RAISE_IF(ss.sum() ≠ t.nvar(),
        "Wrong␣length␣of␣stacks␣for␣UPSTensor␣slicing␣constructor");
    if (first ≡ decideFillMethod(t)) fillFromSparseOne(t, ss, coor);
    else fillFromSparseTwo(t, ss, coor);
  }
```

This code is used in section 324.

327.

⟨ **UPSTensor** increment and decrement 327 ⟩ ≡

```
void UPSTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input/output vector size in UPSTensor::increment");
  UTensor::increment(v, tdims.getNVX());
}
void UPSTensor::decrement(IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input/output vector size in UPSTensor::decrement");
  UTensor::decrement(v, tdims.getNVX());
}
```

This code is used in section 324.

328.

⟨ **UPSTensor**::*fold* code 328 ⟩ ≡

```
FTensor &UPSTensor::fold() const
{
  TL_RAISE("Never should come to this place in UPSTensor::fold");
  FFSTensor *nothing = new FFSTensor(0, 0, 0);
  return *nothing;
}
```

This code is used in section 324.

329.

⟨ **UPSTensor**::*getOffset* code 329 ⟩ ≡

```
int UPSTensor::getOffset(const IntSequence &v) const
{
  TL_RAISE_IF(v.size() != dimen(), "Wrong input vector size in UPSTensor::getOffset");
  return UTensor::getOffset(v, tdims.getNVX());
}
```

This code is used in section 324.

330.

⟨ **UPSTensor**::*addTo* folded code 330 ⟩ ≡

```
void UPSTensor::addTo(FGSTensor &out) const
{
  TL_RAISE_IF(out.getDims() != tdims,
      "Tensors have incompatible dimens in UPSTensor::addTo");
  for (index in = out.begin(); in != out.end(); ++in) {
    IntSequence vtmp(dimen());
    tdims.getPer().apply(in.getCoor(), vtmp);
    index tin(this, vtmp);
    out.addColumn(*this, *tin, *in);
  }
}
```

This code is used in section 324.

331. In here, we have to add this permuted symmetry unfolded tensor to an unfolded not permuted tensor. One easy way would be to go through the target tensor, permute each index, and add the column.

However, it may happen, that the permutation has some non-empty identity tail. In this case, we can add not only individual columns, but much bigger data chunks, which is usually more efficient. Therefore, the code is quite dirty, because we have not an iterator, which iterates over tensor at some higher levels. So we simulate it by the following code.

First we set *cols* to the length of the data chunk and *off* to its dimension. Then we need a front part of *nvmax* of *out*, which is *nvmax_part*. Our iterator here is an integer sequence *outrun* with full length, and *outrun_part* its front part. The *outrun* is initialized to zeros. In each step we need to increment *outrun* *cols*-times, this is done by incrementing its prefix *outrun_part*.

So we loop over all *cols*wide partitions of *out*, permute *outrun* to obtain *perrun* to obtain column of this matrix. (note that the trailing part of *perrun* is the same as of *outrun*. Then we construct submatrices, add them, and increment *outrun*.

⟨ **UPSTensor** :: *addTo* unfolded code 331 ⟩ ≡

```
void UPSTensor::addTo(UGSTensor &out) const
{
  TL_RAISE_IF(out.getDims() ≠ tdims,
      "Tensors have incompatible dimens in UPSTensor::addTo");
  int cols = tailIdentitySize();
  int off = tdims.tailIdentity();
  IntSequence outrun(out.dimen(), 0);
  IntSequence outrun_part(outrun, 0, out.dimen() − off);
  IntSequence nvmax_part(out.getDims().getNVX(), 0, out.dimen() − off);
  for (int out_col = 0; out_col < out.ncols(); out_col += cols) {   /* permute outrun */
    IntSequence perrun(out.dimen());
    tdims.getPer().apply(outrun, perrun);
    index from(this, perrun);   /* construct submatrices */
    ConstTwoDMatrix subfrom(*this, *from, cols);
    TwoDMatrix subout(out, out_col, cols);   /* add */
    subout.add(1, subfrom);   /* increment outrun by cols */
    UTensor::increment(outrun_part, nvmax_part);
  }
}
```

This code is used in section 324.

332. This returns a product of all items in *nvmax* which make up the trailing identity part.

⟨ **UPSTensor** :: *tailIdentitySize* code 332 ⟩ ≡

```
int UPSTensor::tailIdentitySize() const
{
  return tdims.getNVX().mult(dimen() − tdims.tailIdentity(), dimen());
}
```

This code is used in section 324.

333. This fill method is pretty dumb. We go through all columns in **this** tensor, translate coordinates to sparse tensor, sort them and find an item in the sparse tensor. There are many not successful lookups for really sparse tensor, that is why the second method works better for really sparse tensors.

⟨ **UPSTensor** :: *fillFromSparseOne* code 333 ⟩ ≡

```
void UPSTensor::fillFromSparseOne(const FSSparseTensor &t, const IntSequence &ss, const
        IntSequence &coor)
{
  IntSequence cumtmp(ss.size());
  cumtmp[0] = 0;
  for (int i = 1; i < ss.size(); i++) cumtmp[i] = cumtmp[i - 1] + ss[i - 1];
  IntSequence cum(coor.size());
  for (int i = 0; i < coor.size(); i++) cum[i] = cumtmp[coor[i]];
  zeros();
  for (Tensor::index run = begin(); run ≠ end(); ++run) {
    IntSequence c(run.getCoor());
    c.add(1, cum);
    c.sort();
    FSSparseTensor::const_iterator sl = t.getMap().lower_bound(c);
    if (sl ≠ t.getMap().end()) {
      FSSparseTensor::const_iterator su = t.getMap().upper_bound(c);
      for (FSSparseTensor::const_iterator srun = sl; srun ≠ su; ++srun)
        get((*srun).second.first, *run) = (*srun).second.second;
    }
  }
}
```

This code is used in section 324.

334. This is the second way of filling the slice. For instance, let the slice correspond to partitions *abac*. In here we first calculate lower and upper bounds for index of the sparse tensor for the slice. These are *lb_srt* and *ub_srt* respectively. They corresponds to ordering *aabc*. Then we go through that interval, and select items which are really between the bounds. Then we take the index, subtract the lower bound to get it to coordinates of the slice. We get something like (i_a, j_a, k_b, l_c). Then we apply the inverse permutation as of the sorting form $abac \mapsto aabc$ to get index (i_a, k_b, j_a, l_c). Recall that the slice is unfolded, so we have to apply all permutations preserving the stack coordinates *abac*. In our case we get list of indices (i_a, k_b, j_a, l_c) and (j_a, k_b, i_a, l_c). For all we copy the item of the sparse tensor to the appropriate column.

⟨ **UPSTensor**::*fillFromSparseTwo* code 334 ⟩ ≡

```
void UPSTensor::fillFromSparseTwo(const FSSparseTensor &t, const IntSequence &ss, const
        IntSequence &coor)
{
  IntSequence coor_srt(coor);
  coor_srt.sort();
  IntSequence cum(ss.size());
  cum[0] = 0;
  for (int i = 1; i < ss.size(); i++) cum[i] = cum[i - 1] + ss[i - 1];
  IntSequence lb_srt(coor.size());
  IntSequence ub_srt(coor.size());
  for (int i = 0; i < coor.size(); i++) {
    lb_srt[i] = cum[coor_srt[i]];
    ub_srt[i] = cum[coor_srt[i]] + ss[coor_srt[i]] - 1;
  }
  const PermutationSet &pset = tls.pbundle→get(coor.size());
  vector⟨const Permutation *⟩ pp = pset.getPreserving(coor);
  Permutation unsort(coor);
  zeros();
  FSSparseTensor::const_iterator lbi = t.getMap().lower_bound(lb_srt);
  FSSparseTensor::const_iterator ubi = t.getMap().upper_bound(ub_srt);
  for (FSSparseTensor::const_iterator run = lbi; run ≠ ubi; ++run) {
    if (lb_srt.lessEq((*run).first) ∧ (*run).first.lessEq(ub_srt)) {
      IntSequence c((*run).first);
      c.add(−1, lb_srt);
      unsort.apply(c);
      for (unsigned int i = 0; i < pp.size(); i++) {
        IntSequence cp(coor.size());
        pp[i]→apply(c, cp);
        Tensor::index ind(this, cp);
        TL_RAISE_IF(*ind < 0 ∨ *ind ≥ ncols(),
            "Internal␣error␣in␣slicing␣constructor␣of␣UPSTensor");
        get((*run).second.first, *ind) = (*run).second.second;
      }
    }
  }
}
```

This code is used in section 324.

335. Here we calculate the maximum offsets in each folded dimension (dimension sizes, hence *ds*).

⟨ **PerTensorDimens2** :: *setDimensionSizes* code 335 ⟩ ≡

```
void PerTensorDimens2::setDimensionSizes()
{
  const IntSequence &nvs = getNVS();
  for (int i = 0; i < numSyms(); i++) {
    TensorDimens td(syms[i], nvs);
    ds[i] = td.calcFoldMaxOffset();
  }
}
```

This code is used in section 324.

336. If there are two folded dimensions, the offset in such a dimension is offset of the second plus offset of the first times the maximum offset of the second. If there are $n+1$ dimensions, the offset is a sum of offsets of the last dimension plus the offset in the first n dimensions multiplied by the maximum offset of the last dimension. This is exactly what the following code does.

⟨ **PerTensorDimens2** :: *calcOffset* code 336 ⟩ ≡

```
int PerTensorDimens2::calcOffset(const IntSequence &coor) const
{
  TL_RAISE_IF(coor.size() ≠ dimen(),
      "Wrong length of coordinates in PerTensorDimens2::calcOffset");
  IntSequence cc(coor);
  int ret = 0;
  int off = 0;
  for (int i = 0; i < numSyms(); i++) {
    TensorDimens td(syms[i], getNVS());
    IntSequence c(cc, off, off + syms[i].dimen());
    int a = td.calcFoldOffset(c);
    ret = ret * ds[i] + a;
    off += syms[i].dimen();
  }
  return ret;
}
```

This code is used in section 324.

337.

⟨ **PerTensorDimens2** :: *print* code 337 ⟩ ≡

```
void PerTensorDimens2::print() const
{
  printf("nvmax: ");
  nvmax.print();
  printf("per:   ");
  per.print();
  printf("syms:  ");
  syms.print();
  printf("dims:  ");
  ds.print();
}
```

This code is used in section 324.

338. Here we increment the given integer sequence. It corresponds to **UTensor** :: *increment* of the whole sequence, and then partial monotonizing of the subsequences with respect to the symmetries of each dimension.

⟨ **FPSTensor** :: *increment* code 338 ⟩ ≡

```
void FPSTensor::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(), "Wrong␣length␣of␣coordinates␣in␣FPSTensor::increment");
  UTensor::increment(v, tdims.getNVX());
  int off = 0;
  for (int i = 0; i < tdims.numSyms(); i++) {
    IntSequence c(v, off, off + tdims.getSym(i).dimen());
    c.pmonotone(tdims.getSym(i));
    off += tdims.getSym(i).dimen();
  }
}
```

This code is used in section 324.

339.

⟨ **FPSTensor** :: *decrement* code 339 ⟩ ≡

```
void FPSTensor::decrement(IntSequence &v) const
{
  TL_RAISE("FPSTensor::decrement␣not␣implemented");
}
```

This code is used in section 324.

340.

⟨ **FPSTensor** :: *unfold* code 340 ⟩ ≡

```
UTensor &FPSTensor::unfold() const
{
  TL_RAISE("Unfolding␣of␣FPSTensor␣not␣implemented");
  UFSTensor *nothing = new UFSTensor(0, 0, 0);
  return *nothing;
}
```

This code is used in section 324.

341. We only call *calcOffset* of the **PerTensorDimens2**.

⟨ **FPSTensor** :: *getOffset* code 341 ⟩ ≡

```
int FPSTensor::getOffset(const IntSequence &v) const
{
  return tdims.calcOffset(v);
}
```

This code is used in section 324.

342. Here we add the tensor to *out*. We go through all columns of the *out*, apply the permutation to get index in the tensor, and add the column. Note that if the permutation is identity, then the dimensions of the tensors might not be the same (since this tensor is partially folded).

⟨ **FPSTensor** :: *addTo* code 342 ⟩ ≡

```
void FPSTensor::addTo(FGSTensor &out) const
{
  for (index tar = out.begin(); tar ≠ out.end(); ++tar) {
    IntSequence coor(dimen());
    tdims.getPer().apply(tar.getCoor(), coor);
    index src(this, coor);
    out.addColumn(*this, *src, *tar);
  }
}
```

This code is used in section 324.

343. Here is the constructor which multiplies the Kronecker product with the general symmetry sparse tensor **GSSparseTensor**. The main idea is to go through items in the sparse tensor (each item selects rows in the matrices form the Kornecker product), then to Kronecker multiply the rows and multiply with the item, and to add the resulting row to the appropriate row of the resulting **FPSTensor**.

The realization of this idea is a bit more complicated since we have to go through all items, and each item must be added as many times as it has its symmetric elements. Moreover, the permutations shuffle order of rows in their Kronecker product.

So, we through all unfolded indices in a tensor with the same dimensions as the **GSSparseTensor** (sparse slice). For each such index we calculate its folded version (corresponds to ordering of subsequences within symmetries), we test if there is an item in the sparse slice with such coordinates, and if there is, we construct the Kronecker product of the rows, and go through all of items with the coordinates, and add to appropriate rows of **this** tensor.

⟨ **FPSTensor** sparse constructor 343 ⟩ ≡

```
FPSTensor::FPSTensor(const TensorDimens &td, const Equivalence &e, const Permutation
        &p, const GSSparseTensor &a, const KronProdAll &kp)
  : FTensor(along_col, PerTensorDimens(td, Permutation(e, p)).getNVX(), a.nrows(), kp.ncols(),
      td.dimen()), tdims(td, e, p) {
    zeros();
    UGSTensor dummy(0, a.getDims());
    for (Tensor::index run = dummy.begin(); run ≠ dummy.end(); ++run) {
      Tensor::index fold_ind = dummy.getFirstIndexOf(run);
      const IntSequence &c = fold_ind.getCoor();
      GSSparseTensor::const_iterator sl = a.getMap().lower_bound(c);
      if (sl ≠ a.getMap().end()) {
        Vector *row_prod = kp.multRows(run.getCoor());
        GSSparseTensor::const_iterator su = a.getMap().upper_bound(c);
        for (GSSparseTensor::const_iterator srun = sl; srun ≠ su; ++srun) {
          Vector out_row((*srun).second.first, *this);
          out_row.add((*srun).second.second, *row_prod);
        }
        delete row_prod;
      }
    }
  }
```

This code is cited in sections 446 and 450.

This code is used in section 324.

344. End of `ps_tensor.cpp` file.

345. Sparse tensor. Start of `sparse_tensor.h` file.

Here we declare a sparse full and general symmetry tensors with the multidimensional index along columns. We implement them as a **multimap** associating to each sequence of coordinates **IntSequence** a set of pairs (row, number). This is very convenient but not optimal in terms of memory consumption. So the implementation can be changed.

The current **multimap** implementation allows insertions. Another advantage of this approach is that we do not need to calculate column numbers from the **IntSequence**, since the column is accessed directly via the key which is **IntSequence**.

The only operation we need to do with the full symmetry sparse tensor is a left multiplication of a row oriented single column tensor. The result of such operation is a column of the same size as the sparse tensor. Other important operations are slicing operations. We need to do sparse and dense slices of full symmetry sparse tensors. In fact, the only constructor of general symmetry sparse tensor is slicing from the full symmetry sparse.

```
#ifndef SPARSE_TENSOR_H
#define SPARSE_TENSOR_H
#include "symmetry.h"
#include "tensor.h"
#include "gs_tensor.h"
#include "Vector.h"
#include <map>
  using namespace std;
  ⟨ ltseq predicate 346 ⟩;
  ⟨ SparseTensor class declaration 347 ⟩;
  ⟨ FSSparseTensor class declaration 348 ⟩;
  ⟨ GSSparseTensor class declaration 349 ⟩;
#endif
```

346.

```
⟨ ltseq predicate 346 ⟩ ≡
  struct ltseq {
    bool operator()(const IntSequence &s1, const IntSequence &s2) const
    { return s1 < s2; }
  };
```

This code is used in section 345.

347. This is a super class of both full symmetry and general symmetry sparse tensors. It contains a **multimap** and implements insertions. It tracks maximum and minimum row, for which there is an item.

⟨ **SparseTensor** class declaration 347 ⟩ ≡

```
class SparseTensor {
public:
    typedef pair<int, double> Item;
    typedef multimap<IntSequence, Item, ltseq> Map;
    typedef Map::const_iterator const_iterator;
protected:
    typedef Map::iterator iterator;
    Map m;
    const int dim;
    const int nr;
    const int nc;
    int first_nz_row;
    int last_nz_row;
public:
    SparseTensor(int d, int nnr, int nnc)
    : dim(d), nr(nnr), nc(nnc), first_nz_row(nr), last_nz_row(-1) { }
    SparseTensor(const SparseTensor &t)
    : m(t.m), dim(t.dim), nr(t.nr), nc(t.nc) { }
    virtual ~SparseTensor() { }
    void insert(const IntSequence &s, int r, double c);
    const Map &getMap() const
    { return m; }
    int dimen() const
    { return dim; }
    int nrows() const
    { return nr; }
    int ncols() const
    { return nc; }
    double getFillFactor() const
    { return ((double) m.size())/(nrows() * ncols()); }
    double getFoldIndexFillFactor() const;
    double getUnfoldIndexFillFactor() const;
    int getNumNonZero() const
    { return m.size(); }
    int getFirstNonZeroRow() const
    { return first_nz_row; }
    int getLastNonZeroRow() const
    { return last_nz_row; }
    virtual const Symmetry &getSym() const = 0;
    void print() const;
    bool isFinite() const;
}
```

This code is used in section 345.

348. This is a full symmetry sparse tensor. It implements *multColumnAndAdd* and in addition to *sparseTensor*, it has *nv* (number of variables), and symmetry (basically it is a dimension).

⟨ **FSSparseTensor** class declaration 348 ⟩ ≡

```
class FSSparseTensor : public SparseTensor {
public:
  typedef SparseTensor::const_iterator const_iterator;
private:
  const int nv;
  const Symmetry sym;
public:
  FSSparseTensor(int d, int nvar, int r);
  FSSparseTensor(const FSSparseTensor &t);
  void insert(const IntSequence &s, int r, double c);
  void multColumnAndAdd(const Tensor &t, Vector &v) const;
  const Symmetry &getSym() const
  { return sym; }
  int nvar() const
  { return nv; }
  void print() const;
};
```

This code is cited in section 1.

This code is used in section 345.

349. This is a general symmetry sparse tensor. It has **TensorDimens** and can be constructed as a slice of the full symmetry sparse tensor. The slicing constructor takes the same form as the slicing **FGSTensor** constructor from full symmetry sparse tensor.

⟨ **GSSparseTensor** class declaration 349 ⟩ ≡

```
class GSSparseTensor : public SparseTensor {
public:
  typedef SparseTensor::const_iterator const_iterator;
private:
  const TensorDimens tdims;
public:
  GSSparseTensor(const FSSparseTensor &t, const IntSequence &ss, const IntSequence
      &coor, const TensorDimens &td);
  GSSparseTensor(const GSSparseTensor &t)
  : SparseTensor(t), tdims(t.tdims) { }
  void insert(const IntSequence &s, int r, double c);
  const Symmetry &getSym() const
  { return tdims.getSym(); }
  const TensorDimens &getDims() const
  { return tdims; }
  void print() const;
};
```

This code is used in section 345.

350. End of `sparse_tensor.h` file.

351. Start of `sparse_tensor.cpp` file.

```
#include "sparse_tensor.h"
#include "fs_tensor.h"
#include "tl_exception.h"
#include <cmath>
  ⟨ SparseTensor :: insert code 352 ⟩;
  ⟨ SparseTensor :: isFinite code 354 ⟩;
  ⟨ SparseTensor :: getFoldIndexFillFactor code 355 ⟩;
  ⟨ SparseTensor :: getUnfoldIndexFillFactor code 356 ⟩;
  ⟨ SparseTensor :: print code 357 ⟩;
  ⟨ FSSparseTensor constructor code 358 ⟩;
  ⟨ FSSparseTensor copy constructor code 359 ⟩;
  ⟨ FSSparseTensor :: insert code 360 ⟩;
  ⟨ FSSparseTensor :: multColumnAndAdd code 361 ⟩;
  ⟨ FSSparseTensor :: print code 364 ⟩;
  ⟨ GSSparseTensor slicing constructor 365 ⟩;
  ⟨ GSSparseTensor :: insert code 367 ⟩;
  ⟨ GSSparseTensor :: print code 368 ⟩;
```

352. This is straightforward. Before we insert anything, we do a few checks. Then we reset *first_nz_row* and *last_nz_row* if necessary.

```
⟨ SparseTensor :: insert code 352 ⟩ ≡
  void SparseTensor :: insert(const IntSequence &key, int r, double c)
  {
    TL_RAISE_IF(r < 0 ∨ r ≥ nr,
        "Row␣number␣out␣of␣dimension␣of␣tensor␣in␣SparseTensor::insert");
    TL_RAISE_IF(key.size() ≠ dimen(), "Wrong␣length␣of␣key␣in␣SparseTensor::insert");
    TL_RAISE_IF(¬std::isfinite(c), "Insertion␣of␣non-finite␣value␣in␣SparseTensor::insert");
    iterator first_pos = m.lower_bound(key);
    ⟨ check that pair key and r is unique 353 ⟩;
    m.insert(first_pos, Map::value_type(key, Item(r, c)));
    if (first_nz_row > r) first_nz_row = r;
    if (last_nz_row < r) last_nz_row = r;
  }
```

This code is used in section 351.

353.

```
⟨ check that pair key and r is unique 353 ⟩ ≡
  iterator last_pos = m.upper_bound(key);
  for (iterator it = first_pos; it ≠ last_pos; ++it)
    if ((*it).second.first ≡ r) {
      TL_RAISE("Duplicate␣<key,␣r>␣insertion␣in␣SparseTensor::insert");
      return;
    }
```

This code is used in section 352.

354. This returns true if all items are finite (not Nan nor Inf).

⟨ **SparseTensor** :: *isFinite* code 354 ⟩ ≡
```
bool SparseTensor::isFinite() const
{
  bool res = true;
  const_iterator run = m.begin();
  while (res ∧ run ≠ m.end()) {
    if (¬std::isfinite((*run).second.second)) res = false;
    ++run;
  }
  return res;
}
```
This code is used in section 351.

355. This returns a ratio of a number of non-zero columns in folded tensor to the total number of columns.

⟨ **SparseTensor** :: *getFoldIndexFillFactor* code 355 ⟩ ≡
```
double SparseTensor::getFoldIndexFillFactor() const
{
  int cnt = 0;
  const_iterator start_col = m.begin();
  while (start_col ≠ m.end()) {
    cnt++;
    const IntSequence &key = (*start_col).first;
    start_col = m.upper_bound(key);
  }
  return ((double) cnt)/ncols();
}
```
This code is used in section 351.

356. This returns a ratio of a number of non-zero columns in unfolded tensor to the total number of columns.

⟨ **SparseTensor** :: *getUnfoldIndexFillFactor* code 356 ⟩ ≡
```
double SparseTensor::getUnfoldIndexFillFactor() const
{
  int cnt = 0;
  const_iterator start_col = m.begin();
  while (start_col ≠ m.end()) {
    const IntSequence &key = (*start_col).first;
    Symmetry s(key);
    cnt += Tensor::noverseq(s);
    start_col = m.upper_bound(key);
  }
  return ((double) cnt)/ncols();
}
```
This code is used in section 351.

357. This prints the fill factor and all items.

⟨ **SparseTensor** :: *print* code 357 ⟩ ≡

```
void SparseTensor::print() const
{
  printf("Fill: %3.2f %%\n", 100 * getFillFactor());
  const_iterator start_col = m.begin();
  while (start_col != m.end()) {
    const IntSequence &key = (*start_col).first;
    printf("Column: ");
    key.print();
    const_iterator end_col = m.upper_bound(key);
    int cnt = 1;
    for (const_iterator run = start_col; run != end_col; ++run, cnt++) {
      if ((cnt/7) * 7 == cnt) printf("\n");
      printf("%d(%6.2g)  ", (*run).second.first, (*run).second.second);
    }
    printf("\n");
    start_col = end_col;
  }
}
```

This code is used in section 351.

358.

⟨ **FSSparseTensor** constructor code 358 ⟩ ≡

```
FSSparseTensor::FSSparseTensor(int d, int nvar, int r)
: SparseTensor(d, r, FFSTensor::calcMaxOffset(nvar, d)), nv(nvar), sym(d) { }
```

This code is used in section 351.

359.

⟨ **FSSparseTensor** copy constructor code 359 ⟩ ≡

```
FSSparseTensor::FSSparseTensor(const FSSparseTensor &t)
: SparseTensor(t), nv(t.nvar()), sym(t.sym) { }
```

This code is used in section 351.

360.

⟨ **FSSparseTensor** :: *insert* code 360 ⟩ ≡

```
void FSSparseTensor::insert(const IntSequence &key, int r, double c)
{
  TL_RAISE_IF(!key.isSorted(), "Key is not sorted in FSSparseTensor::insert");
  TL_RAISE_IF(key[key.size() - 1] >= nv || key[0] < 0,
      "Wrong value of the key in FSSparseTensor::insert");
  SparseTensor::insert(key, r, c);
}
```

This code is used in section 351.

361. We go through the tensor t which is supposed to have single column. If the item of t is nonzero, we make a key by sorting the index, and then we go through all items having the same key (it is its column), obtain the row number and the element, and do the multiplication.

The test for non-zero is $a \neq 0.0$, since there will be items which are exact zeros.

I have also tried to make the loop through the sparse tensor outer, and find index of tensor t within the loop. Surprisingly, it is little slower (for monomial tests with probability of zeros equal 0.3). But everything depends how filled is the sparse tensor.

```
⟨ FSSparseTensor::multColumnAndAdd code 361 ⟩ ≡
  void FSSparseTensor::multColumnAndAdd(const Tensor &t, Vector &v) const
  {
    ⟨ check compatibility of input parameters 362 ⟩;
    for (Tensor::index it = t.begin(); it ≠ t.end(); ++it) {
      int ind = *it;
      double a = t.get(ind, 0);
      if (a ≠ 0.0) {
        IntSequence key(it.getCoor());
        key.sort();
        ⟨ check that key is within the range 363 ⟩;
        const_iterator first_pos = m.lower_bound(key);
        const_iterator last_pos = m.upper_bound(key);
        for (const_iterator cit = first_pos; cit ≠ last_pos; ++cit) {
          int r = (*cit).second.first;
          double c = (*cit).second.second;
          v[r] += c * a;
        }
      }
    }
  }
```

This code is used in section 351.

362.

```
⟨ check compatibility of input parameters 362 ⟩ ≡
  TL_RAISE_IF(v.length() ≠ nrows(),
      "Wrong␣size␣of␣output␣vector␣in␣FSSparseTensor::multColumnAndAdd");
  TL_RAISE_IF(t.dimen() ≠ dimen(),
      "Wrong␣dimension␣of␣tensor␣in␣FSSparseTensor::multColumnAndAdd");
  TL_RAISE_IF(t.ncols() ≠ 1,
      "The␣input␣tensor␣is␣not␣single-column␣in␣FSSparseTensor::multColumnAndAdd");
```

This code is used in section 361.

363.

```
⟨ check that key is within the range 363 ⟩ ≡
  TL_RAISE_IF(key[0] < 0 ∨ key[key.size() − 1] ≥ nv,
      "Wrong␣coordinates␣of␣index␣in␣FSSparseTensor::multColumnAndAdd");
```

This code is used in section 361.

364.

⟨ **FSSparseTensor** :: *print* code 364 ⟩ ≡

```
void FSSparseTensor::print() const
{
  printf("FS␣Sparse␣tensor:␣dim=%d,␣nv=%d,␣(%dx%d)\n", dim, nv, nr, nc);
  SparseTensor::print();
}
```

This code is used in section 351.

365. This is the same as ⟨ **FGSTensor** slicing from **FSSparseTensor** 275 ⟩.

⟨ **GSSparseTensor** slicing constructor 365 ⟩ ≡

```
GSSparseTensor::GSSparseTensor(const FSSparseTensor &t, const IntSequence &ss, const
        IntSequence &coor, const TensorDimens &td)
: SparseTensor(td.dimen(), t.nrows(), td.calcFoldMaxOffset()), tdims(td) {
  ⟨ set lb and ub to lower and upper bounds of slice indices 366 ⟩;
  FSSparseTensor::const_iterator lbi = t.getMap().lower_bound(lb);
  FSSparseTensor::const_iterator ubi = t.getMap().upper_bound(ub);
  for (FSSparseTensor::const_iterator run = lbi; run ≠ ubi; ++run) {
    if (lb.lessEq((*run).first) ∧ (*run).first.lessEq(ub)) {
      IntSequence c((*run).first);
      c.add(−1, lb);
      insert(c, (*run).second.first, (*run).second.second);
    }
  }
}
```

This code is used in section 351.

366. This is the same as ⟨ set *lb* and *ub* to lower and upper bounds of indices 276 ⟩ in `gs_tensor.cpp`, see that file for details.

⟨ set *lb* and *ub* to lower and upper bounds of slice indices 366 ⟩ ≡

```
IntSequence s_offsets(ss.size(), 0);
for (int i = 1; i < ss.size(); i++) s_offsets[i] = s_offsets[i − 1] + ss[i − 1];
IntSequence lb(coor.size());
IntSequence ub(coor.size());
for (int i = 0; i < coor.size(); i++) {
  lb[i] = s_offsets[coor[i]];
  ub[i] = s_offsets[coor[i]] + ss[coor[i]] − 1;
}
```

This code is used in section 365.

367.

⟨ **GSSparseTensor** :: *insert* code 367 ⟩ ≡

```
void GSSparseTensor::insert(const IntSequence &s, int r, double c)
{
  TL_RAISE_IF(¬s.less(tdims.getNVX()),
      "Wrong␣coordinates␣of␣index␣in␣GSSparseTensor::insert");
  SparseTensor::insert(s, r, c);
}
```

This code is used in section 351.

368.

⟨ **GSSparseTensor** :: *print* code 368 ⟩ ≡

```
void GSSparseTensor::print() const
{
  printf("GS␣Sparse␣tensor:␣(%dx%d)\nSymmetry:␣", nr, nc);
  tdims.getSym().print();
  printf("NVS:␣");
  tdims.getNVS().print();
  SparseTensor::print();
}
```

This code is used in section 351.

369. End of `sparse_tensor.cpp` file.

370. The Faa Di Bruno formula.

371. Tensor containers. Start of `t_container.h` file.

One of primary purposes of the tensor library is to perform one step of the Faa Di Bruno formula:

$$[B_{s^k}]_{\alpha_1\ldots\alpha_k} = [h_{y^l}]_{\gamma_1\ldots\gamma_l} \sum_{c\in M_{l,k}} \prod_{m=1}^{l} [g_{s^{|c_m|}}]^{\gamma_m}_{c_m(\alpha)}$$

where h_{y^l} and g_{s^i} are tensors, $M_{l,k}$ is a set of all equivalences with l classes of k element set, c_m is m-the class of equivalence c, and $|c_m|$ is its cardinality. Further, $c_m(\alpha)$ is a sequence of αs picked by equivalence class c_m.

In order to accomplish this operation, we basically need some storage of all tensors of the form $[g_{s^i}]$. Note that s can be compound, for instance $s = [y, u]$. Then we need storage for $\left[g_{y^3}\right]$, $\left[g_{y^2u}\right]$, $\left[g_{yu^5}\right]$, etc.

We need an object holding all tensors of the same type. Here type means an information, that coordinates of the tensors can be of type y, or u. We will group only tensors, whose symmetry is described by **Symmetry** class. These are only y^2u^3, not $yuyu^2$. So, we are going to define a class which will hold tensors whose symmetries are of type **Symmetry** and have the same symmetry length (number of different coordinate types). Also, for each symmetry there will be at most one tensor.

The class has two purposes: The first is to provide storage (insert and retrieve). The second is to perform the above step of Faa Di Bruno. This is going through all equivalences with l classes, perform the tensor product and add to the result.

We define a template class *TensorContainer*. From different instantiations of the template class we will inherit to create concrete classes, for example container of unfolded general symmetric tensors. The one step of the Faa Di Bruno (we call it *multAndAdd*) is implemented in the concrete subclasses, because the implementation depends on storage. Note even, that *multAndAdd* has not a template common declaration. This is because sparse tensor h is multiplied by folded tensors g yielding folded tensor B, but unfolded tensor h is multiplied by unfolded tensors g yielding unfolded tensor B.

```
#ifndef T_CONTAINER_H
#define T_CONTAINER_H
#include "symmetry.h"
#include "gs_tensor.h"
#include "tl_exception.h"
#include "tl_static.h"
#include "sparse_tensor.h"
#include "equivalence.h"
#include "rfs_tensor.h"
#include "Vector.h"
#include <map>
#include <string>
#include <sstream>
  ⟨ ltsym predicate 372 ⟩;
  ⟨ TensorContainer class definition 373 ⟩;
  ⟨ UGSContainer class declaration 387 ⟩;
  ⟨ FGSContainer class declaration 388 ⟩;
#endif
```

372. We need a predicate on strict weak ordering of symmetries.

⟨ **ltsym** predicate 372 ⟩ ≡

```
struct ltsym {
  bool operator()(const Symmetry &s1, const Symmetry &s2) const
  { return s1 < s2; }
};
```

This code is used in section 371.

373. Here we define the template class for tensor container. We implement it as *stl* ::**map**. It is a unique container, no two tensors with same symmetries can coexist. Keys of the map are symmetries, values are pointers to tensor. The class is responsible for deallocating all tensors. Creation of the tensors is done outside.

The class has integer n as its member. It is a number of different coordinate types of all contained tensors. Besides intuitive insert and retrieve interface, we define a method *fetchTensors*, which for a given symmetry and given equivalence calculates symmetries implied by the symmetry and all equivalence classes, and fetches corresponding tensors in a vector.

Also, each instance of the container has a reference to **EquivalenceBundle** which allows an access to equivalences.

⟨ **TensorContainer** class definition 373 ⟩ ≡

```
template⟨class _Ttype⟩ class TensorContainer {
protected:
  typedef const _Ttype *_const_ptr;
  typedef _Ttype *_ptr;
  typedef map⟨Symmetry, _ptr, ltsym⟩ _Map;
  typedef typename _Map::value_type _mvtype;
public:
  typedef typename _Map::iterator iterator;
  typedef typename _Map::const_iterator const_iterator;
private:
  int n;
  _Map m;
protected:
  const EquivalenceBundle &ebundle;
public:
  TensorContainer(int nn)
  : n(nn), ebundle(*(tls.ebundle)) { }
  ⟨ TensorContainer copy constructor 375 ⟩;
  ⟨ TensorContainer subtensor constructor 376 ⟩;
  ⟨ TensorContainer : get code 377 ⟩;
  ⟨ TensorContainer::check code 378 ⟩;
  ⟨ TensorContainer::insert code 379 ⟩;
  ⟨ TensorContainer::remove code 380 ⟩;
  ⟨ TensorContainer::clear code 381 ⟩;
  ⟨ TensorContainer::fetchTensors code 386 ⟩;
  ⟨ TensorContainer::getMaxDim code 382 ⟩;
  ⟨ TensorContainer::print code 383 ⟩;
  ⟨ TensorContainer::writeMat4 code 384 ⟩;
  ⟨ TensorContainer::writeMMap code 385 ⟩;
  virtual ~TensorContainer()
  { clear(); }
  ⟨ TensorContainer inline methods 374 ⟩;
};
```

This code is used in section 371.

374.

⟨ **TensorContainer** inline methods 374 ⟩ ≡

```
int num() const
{ return n; }
const EquivalenceBundle &getEqBundle() const
{ return ebundle; }
const_iterator begin() const
{ return m.begin(); }
const_iterator end() const
{ return m.end(); }
iterator begin()
{ return m.begin(); }
iterator end()
{ return m.end(); }
```

This code is used in section 373.

375. This is just a copy constructor. This makes a hard copy of all tensors.

⟨ **TensorContainer** copy constructor 375 ⟩ ≡

```
TensorContainer(const TensorContainer<_Ttype> &c)
: n(c.n), m(), ebundle(c.ebundle) {
  for (const_iterator it = c.m.begin(); it != c.m.end(); ++it) {
    _Ttype *ten = new _Ttype(*((*it).second));
    insert(ten);
  }
}
```

This code is used in section 373.

376. This constructor constructs a new tensor container, whose tensors are in-place subtensors of the given container.

⟨ **TensorContainer** subtensor constructor 376 ⟩ ≡

```
TensorContainer(int first_row, int num, TensorContainer<_Ttype> &c)
: n(c.n), ebundle(*(tls.ebundle)) {
  for (iterator it = c.m.begin(); it != c.m.end(); ++it) {
    _Ttype *t = new _Ttype(first_row, num, *((*it).second));
    insert(t);
  }
}
```

This code is used in section 373.

377.

⟨ **TensorContainer** : *get* code 377 ⟩ ≡

```
_const_ptr get(const Symmetry &s) const
{
  TL_RAISE_IF(s.num() ≠ num(), "Incompatible␣symmetry␣lookup␣in␣TensorContainer::get");
  const_iterator it = m.find(s);
  if (it ≡ m.end()) {
    TL_RAISE("Symmetry␣not␣found␣in␣TensorContainer::get");
    return Λ;
  }
  else {
    return (*it).second;
  }
}
_ptr get(const Symmetry &s)
{
  TL_RAISE_IF(s.num() ≠ num(), "Incompatible␣symmetry␣lookup␣in␣TensorContainer::get");
  iterator it = m.find(s);
  if (it ≡ m.end()) {
    TL_RAISE("Symmetry␣not␣found␣in␣TensorContainer::get");
    return Λ;
  }
  else {
    return (*it).second;
  }
}
```

This code is used in section 373.

378.

⟨ **TensorContainer** :: *check* code 378 ⟩ ≡

```
bool check(const Symmetry &s) const
{
  TL_RAISE_IF(s.num() ≠ num(), "Incompatible␣symmetry␣lookup␣in␣TensorContainer::check");
  const_iterator it = m.find(s);
  return it ≠ m.end();
}
```

This code is used in section 373.

379.

⟨ **TensorContainer** :: *insert* code 379 ⟩ ≡

```
void insert(_ptr t)
{
  TL_RAISE_IF(t→getSym().num() ≠ num(),
      "Incompatible␣symmetry␣insertion␣in␣TensorContainer::insert");
  TL_RAISE_IF(check(t→getSym()), "Tensor␣already␣in␣container␣in␣TensorContainer::insert");
  m.insert(_mvtype(t→getSym(), t));
  if (¬t→isFinite()) {
    throw TLException(__FILE__, __LINE__,
        "NaN␣or␣Inf␣asserted␣in␣TensorContainer::insert");
  }
}
```

This code is used in section 373.

380.

⟨ **TensorContainer** :: *remove* code 380 ⟩ ≡

```
void remove(const Symmetry &s)
{
  iterator it = m.find(s);
  if (it ≠ m.end()) {
    _ptr t = (*it).second;
    m.erase(it);
    delete t;
  }
}
```

This code is used in section 373.

381.

⟨ **TensorContainer** :: *clear* code 381 ⟩ ≡

```
void clear()
{
  while (¬m.empty()) {
    delete (*(m.begin())).second;
    m.erase(m.begin());
  }
}
```

This code is used in section 373.

382.

⟨ **TensorContainer** :: *getMaxDim* code 382 ⟩ ≡

```
int getMaxDim() const
{
  int res = −1;
  for (const_iterator run = m.begin(); run ≠ m.end(); ++run) {
    int dim = (∗run).first.dimen();
    if (dim > res) res = dim;
  }
  return res;
}
```

This code is used in section 373.

383. Debug print.

⟨ **TensorContainer** :: *print* code 383 ⟩ ≡

```
void print() const
{
  printf("Tensor␣container:␣nvars=%d,␣tensors=%d\n", n, m.size());
  for (const_iterator it = m.begin(); it ≠ m.end(); ++it) {
    printf("Symmetry:␣");
    (∗it).first.print();
    ((∗it).second)→print();
  }
}
```

This code is used in section 373.

384. Output to the MAT–4 file.

⟨ **TensorContainer** :: *writeMat4* code 384 ⟩ ≡

```
void writeMat4(FILE ∗fd, const char ∗prefix) const
{
  for (const_iterator it = begin(); it ≠ end(); ++it) {
    char lname[100];
    sprintf(lname, "%s_g", prefix);
    const Symmetry &sym = (∗it).first;
    for (int i = 0; i < sym.num(); i++) {
      char tmp[10];
      sprintf(tmp, "_%d", sym[i]);
      strcat(lname, tmp);
    }
    ConstTwoDMatrix m(∗((∗it).second));
    m.writeMat4(fd, lname);
  }
}
```

This code is used in section 373.

385. Output to the Memory Map.

⟨ **TensorContainer** :: *writeMMap* code 385 ⟩ ≡

```
void writeMMap(map<string, ConstTwoDMatrix> &mm, const string &prefix) const
{
  ostringstream lname;
  for (const_iterator it = begin(); it != end(); ++it) {
    lname.str(prefix);
    lname << "_g";
    const Symmetry &sym = (*it).first;
    for (int i = 0; i < sym.num(); i++) lname << "_" << sym[i];
    mm.insert(make_pair(lname.str(), ConstTwoDMatrix(*((*it).second))));
  }
}
```

This code is used in section 373.

386. Here we fetch all tensors given by symmetry and equivalence. We go through all equivalence classes, calculate implied symmetry, and fetch its tensor storing it in the same order to the vector.

⟨ **TensorContainer** :: *fetchTensors* code 386 ⟩ ≡

```
vector<_const_ptr> fetchTensors(const Symmetry &rsym, const Equivalence &e) const
{
  vector<_const_ptr> res(e.numClasses());
  int i = 0;
  for (Equivalence::const_seqit it = e.begin(); it != e.end(); ++it, i++) {
    Symmetry s(rsym, *it);
    res[i] = get(s);
  }
  return res;
}
```

This code is used in section 373.

387. Here is a container storing **UGSTensor**s. We declare *multAndAdd* method.

⟨ **UGSContainer** class declaration 387 ⟩ ≡

```
class FGSContainer;
class UGSContainer : public TensorContainer<UGSTensor> {
public:
  UGSContainer(int nn)
  : TensorContainer<UGSTensor>(nn) { }
  UGSContainer(const UGSContainer &uc)
  : TensorContainer<UGSTensor>(uc) { }
  UGSContainer(const FGSContainer &c);
  void multAndAdd(const UGSTensor &t, UGSTensor &out) const;
};
```

This code is cited in section 1.

This code is used in section 371.

388. Here is a container storing **FGSTensor**s. We declare two versions of *multAndAdd* method. The first works for folded B and folded h tensors, the second works for folded B and unfolded h. There is no point to do it for unfolded B since the algorithm go through all the indices of B and calculates corresponding columns. So, if B is needed unfolded, it is more effective to calculate its folded version and then unfold by conversion.

The static member *num_one_time* is a number of columns formed from product of g tensors at one time. This is subject to change, probably we will have to do some tuning and decide about this number based on symmetries, and dimensions in the runtime.

⟨ **FGSContainer** class declaration 388 ⟩ ≡

```
class FGSContainer : public TensorContainer⟨FGSTensor⟩ {
  static const int num_one_time;
public:
  FGSContainer(int nn)
  : TensorContainer⟨FGSTensor⟩(nn) { }
  FGSContainer(const FGSContainer &fc)
  : TensorContainer⟨FGSTensor⟩(fc) { }
  FGSContainer(const UGSContainer &c);
  void multAndAdd(const FGSTensor &t, FGSTensor &out) const;
  void multAndAdd(const UGSTensor &t, FGSTensor &out) const;
private:
  static Tensor::index getIndices(int num, vector⟨IntSequence⟩ &out, const Tensor::index
      &start, const Tensor::index &end);
};
```

This code is cited in section 1.

This code is used in section 371.

389. End of `t_container.h` file.

390. Start of `t_container.cpp` file.

```
#include "t_container.h"
#include "kron_prod.h"
#include "ps_tensor.h"
#include "pyramid_prod.h"
  const int FGSContainer::num_one_time = 10;
  ⟨ UGSContainer conversion from FGSContainer 391 ⟩;
  ⟨ UGSContainer::multAndAdd code 392 ⟩;
  ⟨ FGSContainer conversion from UGSContainer 393 ⟩;
  ⟨ FGSContainer::multAndAdd folded code 394 ⟩;
  ⟨ FGSContainer::multAndAdd unfolded code 395 ⟩;
  ⟨ FGSContainer::getIndices code 396 ⟩;
```

391.

⟨ **UGSContainer** conversion from **FGSContainer** 391 ⟩ ≡

```
UGSContainer::UGSContainer(const FGSContainer &c)
  : TensorContainer⟨UGSTensor⟩(c.num()) {
    for (FGSContainer::const_iterator it = c.begin(); it ≠ c.end(); ++it) {
      UGSTensor *unfolded = new UGSTensor(*((*it).second));
      insert(unfolded);
    }
}
```

This code is used in section 390.

392. We set l to dimension of t, this is a tensor which multiplies tensors from the container from the left. Also we set k to a dimension of the resulting tensor. We go through all equivalences on k element set and pickup only those which have l classes.

In each loop, we fetch all necessary tensors for the product to the vector *ts*. Then we form Kronecker product **KronProdAll** and feed it with tensors from *ts*. Then we form unfolded permuted symmetry tensor **UPSTensor** as matrix product of t and Kronecker product *kp*. Then we add the permuted data to *out*. This is done by **UPSTensor** method *addTo*.

⟨ **UGSContainer**::*multAndAdd* code 392 ⟩ ≡

```
void UGSContainer::multAndAdd(const UGSTensor &t, UGSTensor &out) const
{
  int l = t.dimen();
  int k = out.dimen();
  const EquivalenceSet &eset = ebundle.get(k);
  for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
    if ((*it).numClasses() ≡ l) {
      vector⟨const UGSTensor *⟩ ts = fetchTensors(out.getSym(), *it);
      KronProdAllOptim kp(l);
      for (int i = 0; i < l; i++) kp.setMat(i, *(ts[i]));
      kp.optimizeOrder();
      UPSTensor ups(out.getDims(), *it, t, kp);
      ups.addTo(out);
    }
  }
}
```

This code is cited in section 395.

This code is used in section 390.

393.

⟨ **FGSContainer** conversion from **UGSContainer** 393 ⟩ ≡

```
FGSContainer::FGSContainer(const UGSContainer &c)
  : TensorContainer⟨FGSTensor⟩(c.num()) {
    for (UGSContainer::const_iterator it = c.begin(); it ≠ c.end(); ++it) {
      FGSTensor *folded = new FGSTensor(*((*it).second));
      insert(folded);
    }
}
```

This code is used in section 390.

394. Here we perform one step of the Faa Di Bruno operation. We call the *multAndAdd* for unfolded tensor.

⟨ **FGSContainer** :: *multAndAdd* folded code 394 ⟩ ≡

```
void FGSContainer::multAndAdd(const FGSTensor &t, FGSTensor &out) const
{
  UGSTensor ut(t);
  multAndAdd(ut, out);
}
```

This code is cited in section 474.

This code is used in section 390.

395. This is the same as ⟨ **UGSContainer** :: *multAndAdd* code 392 ⟩ but we do not construct **UPSTensor** from the Kronecker product, but **FPSTensor**.

⟨ **FGSContainer** :: *multAndAdd* unfolded code 395 ⟩ ≡

```
void FGSContainer::multAndAdd(const UGSTensor &t, FGSTensor &out) const
{
  int l = t.dimen();
  int k = out.dimen();
  const EquivalenceSet &eset = ebundle.get(k);
  for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
    if ((*it).numClasses() ≡ l) {
      vector<const FGSTensor *> ts = fetchTensors(out.getSym(), *it);
      KronProdAllOptim kp(l);
      for (int i = 0; i < l; i++) kp.setMat(i, *(ts[i]));
      kp.optimizeOrder();
      FPSTensor fps(out.getDims(), *it, t, kp);
      fps.addTo(out);
    }
  }
}
```

This code is cited in section 474.

This code is used in section 390.

396. This fills a given vector with integer sequences corresponding to first *num* indices from interval *start* (including) to *end* (excluding). If there are not *num* of such indices, the shorter vector is returned.

⟨ **FGSContainer** :: *getIndices* code 396 ⟩ ≡

```
Tensor::index FGSContainer::getIndices(int num, vector<IntSequence> &out, const
        Tensor::index &start, const Tensor::index &end)
{
  out.clear();
  int i = 0;
  Tensor::index run = start;
  while (i < num && run != end) {
    out.push_back(run.getCoor());
    i++;
    ++run;
  }
  return run;
}
```

This code is used in section 390.

397. End of `t_container.cpp` file.

398. Stack of containers. Start of `stack_container.h` file.

Here we develop abstractions for stacked containers of tensors. For instance, in perturbation methods for SDGE we need function

$$z(y,u,u',\sigma)=\begin{bmatrix}G(y,u,u',\sigma)\\ g(y,u,\sigma)\\ y\\ u\end{bmatrix}$$

and we need to calculate one step of Faa Di Bruno formula

$$[B_{s^k}]_{\alpha_1\ldots\alpha_l}=[f_{z^l}]_{\beta_1\ldots\beta_l}\sum_{c\in M_{l,k}}\prod_{m=1}^{l}\left[z_{s^k(c_m)}\right]^{\beta_m}_{c_m(\alpha)}$$

where we have containers for derivatives of G and g.

The main purpose of this file is to define abstractions for stack of containers and possibly raw variables, and code *multAndAdd* method calculating (one step of) the Faa Di Bruno formula for folded and unfolded tensors. Note also, that tensors $[f_{z^l}]$ are sparse.

The abstractions are built as follows. At the top, there is an interface describing stack of columns. It contains pure virtual methods needed for manipulating the container stack. For technical reasons it is a template. Both versions (folded, and unfolded) provide all interface necessary for implementation of *multAndAdd*. The second way of inheritance is first general implementation of the interface **StackContainer**, and then specific (**ZContainer** for our specific z). The only method which is virtual also after **StackContainer** is *getType*, which is implemented in the specialization and determines behaviour of the stack. The complete classes are obtained by inheriting from the both branches, as it is drawn below:

$$\begin{tikzcd}
& \textbf{StackContainerInterface}\langle\textbf{FGSTensor}\rangle \arrow[dl, "\text{virtual}"'] \arrow[dr, "\text{virtual}"] & \\
\textbf{StackContainer}\langle\textbf{FGSTensor}\rangle \arrow[d] & & \textbf{FoldedStackContainer} \arrow[ddl] \\
\textbf{ZContainer}\langle\textbf{FGSTensor}\rangle \arrow[dr] & & \\
& \textbf{FoldedZContainer} &
\end{tikzcd}$$

$$\begin{tikzcd}
& \textbf{StackContainerInterface}\langle\textbf{UGSTensor}\rangle \arrow[dl, "\text{virtual}"'] \arrow[dr, "\text{virtual}"] & \\
\textbf{StackContainer}\langle\textbf{UGSTensor}\rangle \arrow[d] & & \textbf{UnfoldedStackContainer} \arrow[ddl] \\
\textbf{ZContainer}\langle\textbf{UGSTensor}\rangle \arrow[dr] & & \\
& \textbf{UnfoldedZContainer} &
\end{tikzcd}$$

We have also two supporting classes **StackProduct** and **KronProdStack** and a number of worker classes used as threads.

```
#ifndef STACK_CONTAINER_H
```

```
#define STACK_CONTAINER_H
#include "int_sequence.h"
#include "equivalence.h"
#include "tl_static.h"
#include "t_container.h"
#include "kron_prod.h"
#include "permutation.h"
#include "sthread.h"
  ⟨ StackContainerInterface class declaration 399 ⟩;
  ⟨ StackContainer class declaration 400 ⟩;
  ⟨ FoldedStackContainer class declaration 407 ⟩;
  ⟨ UnfoldedStackContainer class declaration 408 ⟩;
  ⟨ ZContainer class declaration 409 ⟩;
  ⟨ FoldedZContainer class declaration 411 ⟩;
  ⟨ UnfoldedZContainer class declaration 412 ⟩;
  ⟨ GContainer class declaration 413 ⟩;
  ⟨ FoldedGContainer class declaration 415 ⟩;
  ⟨ UnfoldedGContainer class declaration 416 ⟩;
  ⟨ StackProduct class declaration 417 ⟩;
  ⟨ KronProdStack class declaration 424 ⟩;
  ⟨ WorkerFoldMAADense class declaration 426 ⟩;
  ⟨ WorkerFoldMAASparse1 class declaration 427 ⟩;
  ⟨ WorkerFoldMAASparse2 class declaration 428 ⟩;
  ⟨ WorkerFoldMAASparse4 class declaration 429 ⟩;
  ⟨ WorkerUnfoldMAADense class declaration 430 ⟩;
  ⟨ WorkerUnfoldMAASparse1 class declaration 431 ⟩;
  ⟨ WorkerUnfoldMAASparse2 class declaration 432 ⟩;
#endif
```

399. Here is the general interface to stack container. The subclasses maintain **IntSequence** of stack sizes, i.e. size of G, g, y, and u. Then a convenience **IntSequence** of stack offsets. Then vector of pointers to containers, in our example G, and g.

A non-virtual subclass must implement *getType* which determines dependency of stack items on symmetries. There are three possible types for a symmetry. Either the stack item derivative wrt. the symmetry is a matrix, or a unit matrix, or zero.

Method *isZero* returns true if the derivative of a given stack item wrt. to given symmetry is zero as defined by *getType* or the derivative is not present in the container. In this way, we can implement the formula conditional some of the tensors are zero, which is not true (they are only missing).

Method *createPackedColumn* returns a vector of stack derivatives with respect to the given symmetry and of the given column, where all zeros from zero types, or unit matrices are deleted. See `kron_prod2.hweb` for explanation.

⟨ **StackContainerInterface** class declaration 399 ⟩ ≡

```
template⟨class _Ttype⟩
class StackContainerInterface {
public:
  typedef TensorContainer⟨_Ttype⟩ _Ctype;
  typedef enum { matrix, unit, zero } itype;
protected:
  const EquivalenceBundle &ebundle;
public:
  StackContainerInterface()
  : ebundle(*(tls.ebundle)) { }
  virtual ~StackContainerInterface() { }
  virtual const IntSequence &getStackSizes() const = 0;
  virtual IntSequence &getStackSizes() = 0;
  virtual const IntSequence &getStackOffsets() const = 0;
  virtual IntSequence &getStackOffsets() = 0;
  virtual int numConts() const = 0;
  virtual const _Ctype *getCont(int i) const = 0;
  virtual itype getType(int i, const Symmetry &s) const = 0;
  virtual int numStacks() const = 0;
  virtual bool isZero(int i, const Symmetry &s) const = 0;
  virtual const _Ttype *getMatrix(int i, const Symmetry &s) const = 0;
  virtual int getLengthOfMatrixStacks(const Symmetry &s) const = 0;
  virtual int getUnitPos(const Symmetry &s) const = 0;
  virtual Vector *createPackedColumn(const Symmetry &s, const IntSequence &coor, int &iu)
      const = 0;
  int getAllSize() const
  { return getStackOffsets()[numStacks() − 1] + getStackSizes()[numStacks() − 1]; }
};
```

This code is cited in section 1.

This code is used in section 398.

400. Here is **StackContainer**, which implements almost all interface **StackContainerInterface** but one method *getType* which is left for implementation to specializations.

⟨ **StackContainer** class declaration 400 ⟩ ≡

```
template⟨class _Ttype⟩
class StackContainer : virtual public StackContainerInterface⟨_Ttype⟩ {
public:
  typedef StackContainerInterface⟨_Ttype⟩ _Stype;
  typedef typename StackContainerInterface⟨_Ttype⟩::_Ctype _Ctype;
  typedef typename StackContainerInterface⟨_Ttype⟩::itype itype;
protected:
  int num_conts;
  IntSequence stack_sizes;
  IntSequence stack_offsets;
  const _Ctype **const conts;
public:
  StackContainer(int ns, int nc)
  : num_conts(nc), stack_sizes(ns, 0), stack_offsets(ns, 0), conts(new const _Ctype*[nc]) { }
  virtual ~StackContainer( ) {
    delete[] conts;
  }
  const IntSequence &getStackSizes( ) const
  { return stack_sizes; }
  IntSequence &getStackSizes( )
  { return stack_sizes; }
  const IntSequence &getStackOffsets( ) const
  { return stack_offsets; }
  IntSequence &getStackOffsets( )
  { return stack_offsets; }
  int numConts( ) const
  { return num_conts;
  }
  const _Ctype *getCont(int i) const
  { return conts[i]; }
  virtual itype getType(int i, const Symmetry &s) const = 0;
  int numStacks( ) const
  { return stack_sizes.size( ); }
  ⟨ StackContainer::isZero code 401 ⟩;
  ⟨ StackContainer::getMatrix code 402 ⟩;
  ⟨ StackContainer::getLengthOfMatrixStacks code 403 ⟩;
  ⟨ StackContainer::getUnitPos code 404 ⟩;
  ⟨ StackContainer::createPackedColumn code 405 ⟩;
protected:
  ⟨ StackContainer::calculateOffsets code 406 ⟩;
};
```

This code is used in section 398.

401.

⟨ **StackContainer** :: *isZero* code 401 ⟩ ≡

```
bool isZero(int i, const Symmetry &s) const
{
  TL_RAISE_IF(i < 0 ∨ i ≥ numStacks( ), "Wrong␣index␣to␣stack␣in␣StackContainer::isZero.");
  return (getType(i, s) ≡ _Stype::zero ∨ (getType(i, s) ≡ _Stype::matrix ∧ ¬conts[i]→check(s)));
}
```

This code is used in section 400.

402.

⟨ **StackContainer** :: *getMatrix* code 402 ⟩ ≡

```
const _Ttype *getMatrix(int i, const Symmetry &s) const
{
  TL_RAISE_IF(isZero(i, s) ∨ getType(i, s) ≡ _Stype::unit,
      "Matrix␣is␣not␣returned␣in␣StackContainer::getMatrix");
  return conts[i]→get(s);
}
```

This code is used in section 400.

403.

⟨ **StackContainer** :: *getLengthOfMatrixStacks* code 403 ⟩ ≡

```
int getLengthOfMatrixStacks(const Symmetry &s) const
{
  int res = 0;
  int i = 0;
  while (i < numStacks( ) ∧ getType(i, s) ≡ _Stype::matrix) res += stack_sizes[i++];
  return res;
}
```

This code is used in section 400.

404.

⟨ **StackContainer** :: *getUnitPos* code 404 ⟩ ≡

```
int getUnitPos(const Symmetry &s) const
{
  if (s.dimen( ) ≠ 1) return −1;
  int i = numStacks( ) − 1;
  while (i ≥ 0 ∧ getType(i, s) ≠ _Stype::unit) i−−;
  return i;
}
```

This code is used in section 400.

405.

⟨ **StackContainer** :: *createPackedColumn* code 405 ⟩ ≡

```
Vector *createPackedColumn(const Symmetry &s, const IntSequence &coor, int &iu) const
{
  TL_RAISE_IF(s.dimen() != coor.size(),
      "Incompatible coordinates for symmetry in StackContainer::createPackedColumn");
  int len = getLengthOfMatrixStacks(s);
  iu = -1;
  int i = 0;
  if (-1 != (i = getUnitPos(s))) {
    iu = stack_offsets[i] + coor[0];
    len++;
  }
  Vector *res = new Vector(len);
  i = 0;
  while (i < numStacks() && getType(i, s) == _Stype::matrix) {
    const _Ttype *t = getMatrix(i, s);
    Tensor::index ind(t, coor);
    Vector subres(*res, stack_offsets[i], stack_sizes[i]);
    subres = ConstVector(ConstGeneralMatrix(*t), *ind);
    i++;
  }
  if (iu != -1) (*res)[len - 1] = 1;
  return res;
}
```

This code is used in section 400.

406.

⟨ **StackContainer** :: *calculateOffsets* code 406 ⟩ ≡

```
void calculateOffsets()
{
  stack_offsets[0] = 0;
  for (int i = 1; i < stack_offsets.size(); i++) stack_offsets[i] = stack_offsets[i - 1] + stack_sizes[i - 1];
}
```

This code is used in section 400.

407.

⟨ **FoldedStackContainer** class declaration 407 ⟩ ≡

```
class WorkerFoldMAADense;
class WorkerFoldMAASparse1;
class WorkerFoldMAASparse2;
class WorkerFoldMAASparse4;
class FoldedStackContainer : virtual public StackContainerInterface⟨FGSTensor⟩ {
  friend class WorkerFoldMAADense;
  friend class WorkerFoldMAASparse1;
  friend class WorkerFoldMAASparse2;
  friend class WorkerFoldMAASparse4;
public:
  static double fill_threshold;
  void multAndAdd(int dim, const TensorContainer⟨FSSparseTensor⟩ &c, FGSTensor &out)
      const
  { if (c.check(Symmetry(dim))) multAndAdd(*(c.get(Symmetry(dim))), out); }
  void multAndAdd(const FSSparseTensor &t, FGSTensor &out) const;
  void multAndAdd(int dim, const FGSContainer &c, FGSTensor &out) const;
protected:
  void multAndAddSparse1(const FSSparseTensor &t, FGSTensor &out) const;
  void multAndAddSparse2(const FSSparseTensor &t, FGSTensor &out) const;
  void multAndAddSparse3(const FSSparseTensor &t, FGSTensor &out) const;
  void multAndAddSparse4(const FSSparseTensor &t, FGSTensor &out) const;
  void multAndAddStacks(const IntSequence &fi, const FGSTensor &g, FGSTensor &out, const
      void *ad) const;
  void multAndAddStacks(const IntSequence &fi, const GSSparseTensor &g, FGSTensor
      &out, const void *ad) const;
};
```

This code is cited in section 1.

This code is used in section 398.

408.

⟨ **UnfoldedStackContainer** class declaration 408 ⟩ ≡

```
class WorkerUnfoldMAADense;
class WorkerUnfoldMAASparse1;
class WorkerUnfoldMAASparse2;
class UnfoldedStackContainer : virtual public StackContainerInterface<UGSTensor> {
  friend class WorkerUnfoldMAADense;
  friend class WorkerUnfoldMAASparse1;
  friend class WorkerUnfoldMAASparse2;
public:
  static double fill_threshold;
  void multAndAdd(int dim, const TensorContainer<FSSparseTensor> &c, UGSTensor &out)
      const
  { if (c.check(Symmetry(dim))) multAndAdd(*(c.get(Symmetry(dim))), out); }
  void multAndAdd(const FSSparseTensor &t, UGSTensor &out) const;
  void multAndAdd(int dim, const UGSContainer &c, UGSTensor &out) const;
protected:
  void multAndAddSparse1(const FSSparseTensor &t, UGSTensor &out) const;
  void multAndAddSparse2(const FSSparseTensor &t, UGSTensor &out) const;
  void multAndAddStacks(const IntSequence &fi, const UGSTensor &g, UGSTensor &out, const
    void *ad) const;
};
```

This code is cited in section 1.

This code is used in section 398.

409. Here is the specialization of the **StackContainer**. We implement here the z needed in SDGE context. We implement *getType* and define a constructor feeding the data and sizes.

Note that it has two containers, the first is dependent on four variables $G(y^*, u, u', \sigma)$, and the second dependent on three variables $g(y^*, u, \sigma)$. So that we would be able to stack them, we make the second container g be dependent on four variables, the third being u' a dummy and always returning zero if dimension of u' is positive.

⟨ **ZContainer** class declaration 409 ⟩ ≡

```
template<class _Ttype>
class ZContainer : public StackContainer<_Ttype> {
public:
  typedef StackContainer<_Ttype> _Tparent;
  typedef StackContainerInterface<_Ttype> _Stype;
  typedef typename _Tparent::_Ctype _Ctype;
  typedef typename _Tparent::itype itype;
  ZContainer(const _Ctype *gss, int ngss, const _Ctype *g, int ng, int ny, int nu)
  : _Tparent(4, 2) {
    _Tparent::stack_sizes[0] = ngss;
    _Tparent::stack_sizes[1] = ng;
    _Tparent::stack_sizes[2] = ny;
    _Tparent::stack_sizes[3] = nu;
    _Tparent::conts[0] = gss;
    _Tparent::conts[1] = g;
    _Tparent::calculateOffsets();
  }
  ⟨ ZContainer::getType code 410 ⟩;
};
```

This code is cited in sections 1 and 413.

This code is used in section 398.

410. Here we say, what happens if we derive z. recall the top of the file, how z looks, and code is clear.

⟨ **ZContainer** :: *getType* code 410 ⟩ ≡

```
itype getType(int i, const Symmetry &s) const
{
  if (i ≡ 0) return _Stype::matrix;
  if (i ≡ 1)
    if (s[2] > 0) return _Stype::zero;
    else return _Stype::matrix;
  if (i ≡ 2)
    if (s ≡ Symmetry(1, 0, 0, 0)) return _Stype::unit;
    else return _Stype::zero;
  if (i ≡ 3)
    if (s ≡ Symmetry(0, 1, 0, 0)) return _Stype::unit;
    else return _Stype::zero;
  TL_RAISE("Wrong␣stack␣index␣in␣ZContainer::getType");
  return _Stype::zero;
}
```

This code is used in section 409.

411.

⟨ **FoldedZContainer** class declaration 411 ⟩ ≡
class FoldedZContainer : **public ZContainer**⟨**FGSTensor**⟩, **public FoldedStackContainer** {
public:
typedef TensorContainer⟨**FGSTensor**⟩ **_Ctype**;
FoldedZContainer(**const _Ctype** $*gss$, **int** $ngss$, **const _Ctype** $*g$, **int** ng, **int** ny, **int** nu)
: **ZContainer**⟨**FGSTensor**⟩($gss, ngss, g, ng, ny, nu$) { }
};

This code is cited in section 1.

This code is used in section 398.

412.

⟨ **UnfoldedZContainer** class declaration 412 ⟩ ≡
class UnfoldedZContainer : **public ZContainer**⟨**UGSTensor**⟩, **public UnfoldedStackContainer**
{
public:
typedef TensorContainer⟨**UGSTensor**⟩ **_Ctype**;
UnfoldedZContainer(**const _Ctype** $*gss$, **int** $ngss$, **const _Ctype** $*g$, **int** ng, **int** ny, **int** nu)
: **ZContainer**⟨**UGSTensor**⟩($gss, ngss, g, ng, ny, nu$) { }
};

This code is cited in section 1.

This code is used in section 398.

413. Here we have another specialization of container used in context of SDGE. We define a container for

$$G(y, u, u', \sigma) = g^{**}(g^*(y, u, \sigma), u', \sigma)$$

For some reason, the symmetry of g^{**} has length 4 although it is really dependent on three variables. (To now the reason, consult ⟨ **ZContainer** class declaration 409 ⟩.) So, it has four stack, the third one is dummy, and always returns zero. The first stack corresponds to a container of g^*.

⟨ **GContainer** class declaration 413 ⟩ ≡

```
template⟨class _Ttype⟩
class GContainer : public StackContainer⟨_Ttype⟩ {
public:
  typedef StackContainer⟨_Ttype⟩ _Tparent;
  typedef StackContainerInterface⟨_Ttype⟩ _Stype;
  typedef typename StackContainer⟨_Ttype⟩::_Ctype _Ctype;
  typedef typename StackContainer⟨_Ttype⟩::itype itype;
  GContainer(const _Ctype *gs, int ngs, int nu)
  : StackContainer⟨_Ttype⟩(4, 1) {
    _Tparent::stack_sizes[0] = ngs;
    _Tparent::stack_sizes[1] = nu;
    _Tparent::stack_sizes[2] = nu;
    _Tparent::stack_sizes[3] = 1;
    _Tparent::conts[0] = gs;
    _Tparent::calculateOffsets( );
  }
  ⟨ GContainer::getType code 414 ⟩;
};
```

This code is cited in section 1.

This code is used in section 398.

414. Here we define the dependencies in $g^{**}(g^*(y, u, \sigma), u', \sigma)$. Also note, that first derivative of g^* wrt σ is always zero, so we also add this information.

⟨ **GContainer** :: *getType* code 414 ⟩ ≡

```
itype getType(int i, const Symmetry &s) const
{
  if (i ≡ 0)
    if (s[2] > 0 ∨ s ≡ Symmetry(0, 0, 0, 1)) return _Stype::zero;
    else return _Stype::matrix;
  if (i ≡ 1)
    if (s ≡ Symmetry(0, 0, 1, 0)) return _Stype::unit;
    else return _Stype::zero;
  if (i ≡ 2) return _Stype::zero;
  if (i ≡ 3)
    if (s ≡ Symmetry(0, 0, 0, 1)) return _Stype::unit;
    else return _Stype::zero;
  TL_RAISE("Wrong stack index in GContainer::getType");
  return _Stype::zero;
}
```

This code is used in section 413.

415.

⟨ **FoldedGContainer** class declaration 415 ⟩ ≡

```
class FoldedGContainer : public GContainer<FGSTensor>, public FoldedStackContainer {
public:
    typedef TensorContainer<FGSTensor> _Ctype;
    FoldedGContainer(const _Ctype *gs, int ngs, int nu)
    : GContainer<FGSTensor>(gs, ngs, nu) { }
};
```

This code is cited in section 1.

This code is used in section 398.

416.

⟨ **UnfoldedGContainer** class declaration 416 ⟩ ≡

```
class UnfoldedGContainer : public GContainer<UGSTensor>, public UnfoldedStackContainer
        {
public:
    typedef TensorContainer<UGSTensor> _Ctype;
    UnfoldedGContainer(const _Ctype *gs, int ngs, int nu)
    : GContainer<UGSTensor>(gs, ngs, nu) { }
};
```

This code is cited in section 1.

This code is used in section 398.

417. Here we have a support class for product of **StackContainer**s. It only adds a dimension to **StackContainer**. It selects the symmetries according to equivalence classes passed to the constructor. The equivalence can have permuted classes by some given permutation. Nothing else is interesting.

```
⟨ StackProduct class declaration 417 ⟩ ≡
  template⟨class _Ttype⟩
  class StackProduct {
  public:
    typedef StackContainerInterface⟨_Ttype⟩ _Stype;
    typedef typename _Stype::_Ctype _Ctype;
    typedef typename _Stype::itype itype;
  protected:
    const _Stype &stack_cont;
    InducedSymmetries syms;
    Permutation per;
  public:
    StackProduct(const _Stype &sc, const Equivalence &e, const Symmetry &os)
    : stack_cont(sc), syms(e, os), per(e) { }
    StackProduct(const _Stype &sc, const Equivalence &e, const Permutation &p, const
            Symmetry &os)
    : stack_cont(sc), syms(e, p, os), per(e, p) { }
    int dimen() const
    { return syms.size(); }
    int getAllSize() const
    { return stack_cont.getAllSize(); }
    const Symmetry &getProdSym(int ip) const
    { return syms[ip]; }
    ⟨ StackProduct::isZero code 418 ⟩;
    ⟨ StackProduct::getType code 419 ⟩;
    ⟨ StackProduct::getMatrix code 420 ⟩;
    ⟨ StackProduct::createPackedColumns code 421 ⟩;
    ⟨ StackProduct::getSize code 422 ⟩;
    ⟨ StackProduct::numMatrices code 423 ⟩;
  };
```

This code is used in section 398.

418.

```
⟨ StackProduct::isZero code 418 ⟩ ≡
  bool isZero(const IntSequence &istacks) const
  {
    TL_RAISE_IF(istacks.size() ≠ dimen(), "Wrong␣istacks␣coordinates␣for␣StackProduct::isZero");
    bool res = false;
    int i = 0;
    while (i < dimen() ∧ ¬(res = stack_cont.isZero(istacks[i], syms[i]))) i++;
    return res;
  }
```

This code is used in section 417.

419.

⟨ **StackProduct** :: *getType* code 419 ⟩ ≡

```
itype getType(int is, int ip) const
{
  TL_RAISE_IF(is < 0 ∨ is ≥ stack_cont.numStacks(),
      "Wrong␣index␣to␣stack␣in␣StackProduct::getType");
  TL_RAISE_IF(ip < 0 ∨ ip ≥ dimen(),
      "Wrong␣index␣to␣stack␣container␣in␣StackProduct::getType");
  return stack_cont.getType(is, syms[ip]);
}
```

This code is used in section 417.

420.

⟨ **StackProduct** :: *getMatrix* code 420 ⟩ ≡

```
const _Ttype *getMatrix(int is, int ip) const
{
  return stack_cont.getMatrix(is, syms[ip]);
}
```

This code is used in section 417.

421.

⟨ **StackProduct** :: *createPackedColumns* code 421 ⟩ ≡

```
void createPackedColumns(const IntSequence &coor, Vector **vs, IntSequence &iu) const
{
  TL_RAISE_IF(iu.size() ≠ dimen(),
      "Wrong␣storage␣length␣for␣unit␣flags␣in␣StackProduct::createPackedColumn");
  TL_RAISE_IF(coor.size() ≠ per.size(),
      "Wrong␣size␣of␣index␣coor␣in␣StackProduct::createPackedColumn");
  IntSequence perindex(coor.size());
  per.apply(coor, perindex);
  int off = 0;
  for (int i = 0; i < dimen(); i++) {
    IntSequence percoor(perindex, off, syms[i].dimen() + off);
    vs[i] = stack_cont.createPackedColumn(syms[i], percoor, iu[i]);
    off += syms[i].dimen();
  }
}
```

This code is used in section 417.

422.

⟨ **StackProduct** :: *getSize* code 422 ⟩ ≡

```
int getSize(int is) const
{
  return stack_cont.getStackSizes()[is];
}
```

This code is used in section 417.

423.

⟨ **StackProduct** :: *numMatrices* code 423 ⟩ ≡

```
int numMatrices(const IntSequence &istacks) const
{
  TL_RAISE_IF(istacks.size() ≠ dimen(),
      "Wrong␣size␣of␣stack␣coordinates␣in␣StackContainer::numMatrices");
  int ret = 0;
  int ip = 0;
  while (ip < dimen() ∧ getType(istacks[ip], ip) ≡ _Stype::matrix) {
    ret++;
    ip++;
  }
  return ret;
}
```

This code is used in section 417.

424. Here we only inherit from Kronecker product **KronProdAllOptim**, only to allow for a constructor constructing from **StackProduct**.

⟨ **KronProdStack** class declaration 424 ⟩ ≡

```
template⟨class _Ttype⟩ class KronProdStack : public KronProdAllOptim {
public:
  typedef StackProduct⟨_Ttype⟩ _Ptype;
  typedef StackContainerInterface⟨_Ttype⟩ _Stype;
  ⟨ KronProdStack constructor code 425 ⟩;
};
```

This code is used in section 398.

425. Here we construct **KronProdAllOptim** from **StackContainer** and given selections of stack items from stack containers in the product. We only decide whether to insert matrix, or unit matrix.

At this point, we do not call **KronProdAllOptim** :: *optimizeOrder*, so the **KronProdStack** behaves like **KronProdAll** (i.e. no optimization is done).

⟨ **KronProdStack** constructor code 425 ⟩ ≡

```
KronProdStack(const _Ptype &sp, const IntSequence &istack)
: KronProdAllOptim(sp.dimen()) {
  TL_RAISE_IF(sp.dimen() ≠ istack.size(),
      "Wrong␣stack␣product␣dimension␣for␣KronProdStack␣constructor");
  for (int i = 0; i < sp.dimen(); i++) {
    TL_RAISE_IF(sp.getType(istack[i], i) ≡ _Stype::zero,
        "Attempt␣to␣construct␣KronProdStack␣from␣zero␣matrix");
    if (sp.getType(istack[i], i) ≡ _Stype::unit) setUnit(i, sp.getSize(istack[i]));
    if (sp.getType(istack[i], i) ≡ _Stype::matrix) {
      const TwoDMatrix *m = sp.getMatrix(istack[i], i);
      TL_RAISE_IF(m→nrows() ≠ sp.getSize(istack[i]),
          "Wrong␣size␣of␣returned␣matrix␣in␣KronProdStack␣constructor");
      setMat(i, *m);
    }
  }
}
```

This code is used in section 424.

426.

⟨ **WorkerFoldMAADense** class declaration 426 ⟩ ≡

```
class WorkerFoldMAADense : public THREAD
{
  const FoldedStackContainer &cont;
  Symmetry sym;
  const FGSContainer &dense_cont;
  FGSTensor &out;
public:
  WorkerFoldMAADense(const FoldedStackContainer &container, const Symmetry &s, const
      FGSContainer &dcontainer, FGSTensor &outten);
  void operator()();
}
;
```

This code is used in section 398.

427.

⟨ **WorkerFoldMAASparse1** class declaration 427 ⟩ ≡

```
class WorkerFoldMAASparse1 : public THREAD
{
  const FoldedStackContainer &cont;
  const FSSparseTensor &t;
  FGSTensor &out;
  IntSequence coor;
  const EquivalenceBundle &ebundle;
public:
  WorkerFoldMAASparse1(const FoldedStackContainer &container, const FSSparseTensor
      &ten, FGSTensor &outten, const IntSequence &c);
  void operator()();
}
;
```

This code is used in section 398.

428.

⟨ **WorkerFoldMAASparse2** class declaration 428 ⟩ ≡

```
class WorkerFoldMAASparse2 : public THREAD
{
  const FoldedStackContainer &cont;
  const FSSparseTensor &t;
  FGSTensor &out;
  IntSequence coor;
public:
  WorkerFoldMAASparse2(const FoldedStackContainer &container, const FSSparseTensor
      &ten, FGSTensor &outten, const IntSequence &c);
  void operator()();
}
;
```

This code is used in section 398.

429.

⟨ **WorkerFoldMAASparse4** class declaration 429 ⟩ ≡

```
class WorkerFoldMAASparse4 : public THREAD
{
  const FoldedStackContainer &cont;
  const FSSparseTensor &t;
  FGSTensor &out;
  IntSequence coor;
public:
  WorkerFoldMAASparse4(const FoldedStackContainer &container, const FSSparseTensor
      &ten, FGSTensor &outten, const IntSequence &c);
  void operator()();
}
;
```

This code is used in section 398.

430.

⟨ **WorkerUnfoldMAADense** class declaration 430 ⟩ ≡

```
class WorkerUnfoldMAADense : public THREAD
{
  const UnfoldedStackContainer &cont;
  Symmetry sym;
  const UGSContainer &dense_cont;
  UGSTensor &out;
public:
  WorkerUnfoldMAADense(const UnfoldedStackContainer &container, const Symmetry
      &s, const UGSContainer &dcontainer, UGSTensor &outten);
  void operator()();
}
;
```

This code is used in section 398.

431.

⟨ **WorkerUnfoldMAASparse1** class declaration 431 ⟩ ≡

```
class WorkerUnfoldMAASparse1 : public THREAD
{
  const UnfoldedStackContainer &cont;
  const FSSparseTensor &t;
  UGSTensor &out;
  IntSequence coor;
  const EquivalenceBundle &ebundle;
public:
  WorkerUnfoldMAASparse1(const UnfoldedStackContainer &container, const
      FSSparseTensor &ten, UGSTensor &outten, const IntSequence &c);
  void operator()();
}
;
```

This code is used in section 398.

432.

⟨ **WorkerUnfoldMAASparse2** class declaration 432 ⟩ ≡

```
class WorkerUnfoldMAASparse2 : public THREAD
{
  const UnfoldedStackContainer &cont;
  const FSSparseTensor &t;
  UGSTensor &out;
  IntSequence coor;
public:
  WorkerUnfoldMAASparse2(const UnfoldedStackContainer &container, const
      FSSparseTensor &ten, UGSTensor &outten, const IntSequence &c);
  void operator()();
}
;
```

This code is used in section 398.

433. End of `stack_container.h` file.

434. Start of `stack_container.cpp` file.

```
#include "stack_container.h"
#include "pyramid_prod2.h"
#include "ps_tensor.h"
  double FoldedStackContainer::fill_threshold = 0.00005;
  double UnfoldedStackContainer::fill_threshold = 0.00005;
  ⟨ FoldedStackContainer::multAndAdd sparse code 435 ⟩;
  ⟨ FoldedStackContainer::multAndAdd dense code 436 ⟩;
  ⟨ WorkerFoldMAADense::operator()() code 437 ⟩;
  ⟨ WorkerFoldMAADense constructor code 438 ⟩;
  ⟨ FoldedStackContainer::multAndAddSparse1 code 439 ⟩;
  ⟨ WorkerFoldMAASparse1::operator()() code 440 ⟩;
  ⟨ WorkerFoldMAASparse1 constructor code 441 ⟩;
  ⟨ FoldedStackContainer::multAndAddSparse2 code 442 ⟩;
  ⟨ WorkerFoldMAASparse2::operator()() code 443 ⟩;
  ⟨ WorkerFoldMAASparse2 constructor code 444 ⟩;
  ⟨ FoldedStackContainer::multAndAddSparse3 code 445 ⟩;
  ⟨ FoldedStackContainer::multAndAddSparse4 code 446 ⟩;
  ⟨ WorkerFoldMAASparse4::operator()() code 447 ⟩;
  ⟨ WorkerFoldMAASparse4 constructor code 448 ⟩;
  ⟨ FoldedStackContainer::multAndAddStacks dense code 449 ⟩;
  ⟨ FoldedStackContainer::multAndAddStacks sparse code 450 ⟩;
  ⟨ UnfoldedStackContainer::multAndAdd sparse code 451 ⟩;
  ⟨ UnfoldedStackContainer::multAndAdd dense code 452 ⟩;
  ⟨ WorkerUnfoldMAADense::operator()() code 453 ⟩;
  ⟨ WorkerUnfoldMAADense constructor code 454 ⟩;
  ⟨ UnfoldedStackContainer::multAndAddSparse1 code 455 ⟩;
  ⟨ WorkerUnfoldMAASparse1::operator()() code 456 ⟩;
  ⟨ WorkerUnfoldMAASparse1 constructor code 457 ⟩;
  ⟨ UnfoldedStackContainer::multAndAddSparse2 code 458 ⟩;
  ⟨ WorkerUnfoldMAASparse2::operator()() code 459 ⟩;
  ⟨ WorkerUnfoldMAASparse2 constructor code 460 ⟩;
  ⟨ UnfoldedStackContainer::multAndAddStacks code 461 ⟩;
```

435. Here we multiply the sparse tensor with the **FoldedStackContainer**. We have four implementations, *multAndAddSparse1*, *multAndAddSparse2*, *multAndAddSparse3*, and *multAndAddSparse4*. The third is not threaded yet and I expect that it is certainly the slowest. The *multAndAddSparse4* exploits the sparsity, however, it seems to be still worse than *multAndAddSparse2* even for really sparse matrices. On the other hand, it can be more efficient than *multAndAddSparse2* for large problems, since it does not need that much of memory and can avoid much swapping. Very preliminary examination shows that *multAndAddSparse2* is the best in terms of time.

```
⟨ FoldedStackContainer::multAndAdd sparse code 435 ⟩ ≡
  void FoldedStackContainer::multAndAdd(const FSSparseTensor &t, FGSTensor &out) const
  {
    TL_RAISE_IF(t.nvar() ≠ getAllSize(),
        "Wrong␣number␣of␣variables␣of␣tensor␣for␣FoldedStackContainer::multAndAdd");
    multAndAddSparse2(t, out);
  }
```

This code is used in section 434.

436. Here we perform the Faa Di Bruno step for a given dimension *dim*, and for the dense fully symmetric tensor which is scattered in the container of general symmetric tensors. The implementation is pretty the same as ⟨ **UnfoldedStackContainer** :: *multAndAdd* dense code 452 ⟩.

⟨ **FoldedStackContainer** :: *multAndAdd* dense code 436 ⟩ ≡

```
void FoldedStackContainer::multAndAdd(int dim, const FGSContainer &c, FGSTensor &out)
        const
{
  TL_RAISE_IF(c.num() != numStacks(),
      "Wrong symmetry length of container for FoldedStackContainer::multAndAdd");
  THREAD_GROUP gr;
  SymmetrySet ss(dim, c.num());
  for (symiterator si(ss); !si.isEnd(); ++si) {
    if (c.check(*si)) {
      THREAD *worker = new WorkerFoldMAADense(*this, *si, c, out);
      gr.insert(worker);
    }
  }
  gr.run();
}
```

This code is used in section 434.

437. This is analogous to ⟨ **WorkerUnfoldMAADense** :: **operator**( )( ) code 453 ⟩.

⟨ **WorkerFoldMAADense** :: **operator**( )( ) code 437 ⟩ ≡

```
void WorkerFoldMAADense::operator()()
{
  Permutation iden(dense_cont.num());
  IntSequence coor(sym, iden.getMap());
  const FGSTensor *g = dense_cont.get(sym);
  cont.multAndAddStacks(coor, *g, out, &out);
}
```

This code is used in section 434.

438.

⟨ **WorkerFoldMAADense** constructor code 438 ⟩ ≡

```
WorkerFoldMAADense::WorkerFoldMAADense(const FoldedStackContainer
        &container, const Symmetry &s, const FGSContainer &dcontainer, FGSTensor &outten)
  : cont(container), sym(s), dense_cont(dcontainer), out(outten) { }
```

This code is used in section 434.

439. This is analogous to ⟨ **UnfoldedStackContainer** :: *multAndAddSparse1* code 455 ⟩.

⟨ **FoldedStackContainer** :: *multAndAddSparse1* code 439 ⟩ ≡

```
void FoldedStackContainer::multAndAddSparse1(const FSSparseTensor &t, FGSTensor &out)
    const
{
  THREAD_GROUP gr;
  UFSTensor dummy(0, numStacks(), t.dimen());
  for (Tensor::index ui = dummy.begin(); ui ≠ dummy.end(); ++ui) {
    THREAD *worker = new WorkerFoldMAASparse1(*this, t, out, ui.getCoor());
    gr.insert(worker);
  }
  gr.run();
}
```

This code is used in section 434.

440. This is analogous to ⟨ **WorkerUnfoldMAASparse1** :: **operator**( )( ) code 456 ⟩. The only difference is that instead of **UPSTensor** as a result of multiplication of unfolded tensor and tensors from containers, we have **FPSTensor** with partially folded permuted symmetry.

todo: make slice vertically narrowed according to the fill of t, vertically narrow out accordingly.

⟨ **WorkerFoldMAASparse1** :: **operator**( )( ) code 440 ⟩ ≡

```
void WorkerFoldMAASparse1::operator()()
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  const PermutationSet &pset = tls.pbundle->get(t.dimen());
  Permutation iden(t.dimen());
  UPSTensor slice(t, cont.getStackSizes(), coor, PerTensorDimens(cont.getStackSizes(), coor));
  for (int iper = 0; iper < pset.getNum(); iper++) {
    const Permutation &per = pset.get(iper);
    IntSequence percoor(coor.size());
    per.apply(coor, percoor);
    for (EquivalenceSet::const_iterator it = eset.begin(); it != eset.end(); ++it) {
      if ((*it).numClasses() == t.dimen()) {
        StackProduct<FGSTensor> sp(cont, *it, out.getSym());
        if (!sp.isZero(percoor)) {
          KronProdStack<FGSTensor> kp(sp, percoor);
          kp.optimizeOrder();
          const Permutation &oper = kp.getPer();
          if (Permutation(oper, per) == iden) {
            FPSTensor fps(out.getDims(), *it, slice, kp);
            {
              SYNCHRO syn(&out, "WorkerUnfoldMAASparse1");
              fps.addTo(out);
            }
          }
        }
      }
    }
  }
}
```

This code is used in section 434.

441.

⟨ **WorkerFoldMAASparse1** constructor code 441 ⟩ ≡

```
WorkerFoldMAASparse1::WorkerFoldMAASparse1(const FoldedStackContainer
      &container, const FSSparseTensor &ten, FGSTensor &outten, const IntSequence &c)
  : cont(container), t(ten), out(outten), coor(c), ebundle(*(tls.ebundle)) { }
```

This code is used in section 434.

442. Here is the second implementation of sparse folded *multAndAdd*. It is pretty similar to implementation of ⟨ **UnfoldedStackContainer** :: *multAndAddSparse2* code 458 ⟩. We make a dense folded *slice*, and then call folded *multAndAddStacks*, which multiplies all the combinations compatible with the slice.

⟨ **FoldedStackContainer** :: *multAndAddSparse2* code 442 ⟩ ≡

```
void FoldedStackContainer::multAndAddSparse2(const FSSparseTensor &t, FGSTensor &out)
        const
{
  THREAD_GROUP gr;
  FFSTensor dummy_f(0, numStacks(), t.dimen());
  for (Tensor::index fi = dummy_f.begin(); fi ≠ dummy_f.end(); ++fi) {
    THREAD *worker = new WorkerFoldMAASparse2(*this, t, out, fi.getCoor());
    gr.insert(worker);
  }
  gr.run();
}
```

This code is used in section 434.

443. Here we make a sparse slice first and then call *multAndAddStacks* if the slice is not empty. If the slice is really sparse, we call sparse version of *multAndAddStacks*. What means "really sparse" is given by *fill_threshold*. It is not tuned yet, a practice shows that it must be a really low number, since sparse *multAndAddStacks* is much slower than the dense version.

Further, we take only nonzero rows of the slice, and accordingly of the out tensor. We jump over zero initial rows and drop zero tailing rows.

⟨ **WorkerFoldMAASparse2** :: **operator**( )( ) code 443 ⟩ ≡

```
void WorkerFoldMAASparse2::operator()()
{
  GSSparseTensor slice(t, cont.getStackSizes(), coor, TensorDimens(cont.getStackSizes(), coor));
  if (slice.getNumNonZero()) {
    if (slice.getUnfoldIndexFillFactor() > FoldedStackContainer::fill_threshold) {
      FGSTensor dense_slice(slice);
      int r1 = slice.getFirstNonZeroRow();
      int r2 = slice.getLastNonZeroRow();
      FGSTensor dense_slice1(r1, r2 − r1 + 1, dense_slice);
      FGSTensor out1(r1, r2 − r1 + 1, out);
      cont.multAndAddStacks(coor, dense_slice1, out1, &out);
    }
    else cont.multAndAddStacks(coor, slice, out, &out);
  }
}
```

This code is cited in section 459.

This code is used in section 434.

444.

⟨ **WorkerFoldMAASparse2** constructor code 444 ⟩ ≡

```
WorkerFoldMAASparse2::WorkerFoldMAASparse2(const FoldedStackContainer
        &container, const FSSparseTensor &ten, FGSTensor &outten, const IntSequence &c)
  : cont(container), t(ten), out(outten), coor(c) { }
```

This code is used in section 434.

445. Here is the third implementation of the sparse folded *multAndAdd*. It is column-wise implementation, and thus is not a good candidate for the best performer.

We go through all columns from the output. For each column we calculate folded *sumcol* which is a sum of all appropriate columns for all suitable equivalences. So we go through all suitable equivalences, for each we construct a **StackProduct** object and construct **IrregTensor** for a corresponding column of z. The **IrregTensor** is an abstraction for Kronecker multiplication of stacked columns of the two containers without zeros. Then the column is added to *sumcol*. Finally, the *sumcol* is multiplied by the sparse tensor.

⟨ **FoldedStackContainer** :: *multAndAddSparse3* code 445 ⟩ ≡

```
void FoldedStackContainer::multAndAddSparse3(const FSSparseTensor &t, FGSTensor &out)
        const
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  for (Tensor::index run = out.begin(); run ≠ out.end(); ++run) {
    Vector outcol(out, *run);
    FRSingleTensor sumcol(t.nvar(), t.dimen());
    sumcol.zeros();
    for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
      if ((*it).numClasses() ≡ t.dimen()) {
        StackProduct⟨FGSTensor⟩ sp(*this, *it, out.getSym());
        IrregTensorHeader header(sp, run.getCoor());
        IrregTensor irten(header);
        irten.addTo(sumcol);
      }
    }
    t.multColumnAndAdd(sumcol, outcol);
  }
}
```

This code is used in section 434.

446. Here is the fourth implementation of sparse **FoldedStackContainer** :: *multAndAdd*. It is almost equivalent to *multAndAddSparse2* with the exception that the **FPSTensor** as a result of a product of a slice and Kronecker product of the stack derivatives is calculated in the sparse fashion. For further details, see ⟨ **FoldedStackContainer** :: *multAndAddStacks* sparse code 450 ⟩ and ⟨ **FPSTensor** sparse constructor 343 ⟩.

⟨ **FoldedStackContainer** :: *multAndAddSparse4* code 446 ⟩ ≡

```
void FoldedStackContainer::multAndAddSparse4(const FSSparseTensor &t, FGSTensor &out)
        const
{
  THREAD_GROUP gr;
  FFSTensor dummy_f(0, numStacks(), t.dimen());
  for (Tensor::index fi = dummy_f.begin(); fi ≠ dummy_f.end(); ++fi) {
    THREAD *worker = new WorkerFoldMAASparse4(*this, t, out, fi.getCoor());
    gr.insert(worker);
  }
  gr.run();
}
```

This code is used in section 434.

447. The **WorkerFoldMAASparse4** is the same as **WorkerFoldMAASparse2** with the exception that we call a sparse version of *multAndAddStacks*.

⟨ **WorkerFoldMAASparse4** :: **operator**( )( ) code 447 ⟩ ≡

```
void WorkerFoldMAASparse4::operator()()
{
  GSSparseTensor slice(t, cont.getStackSizes(), coor, TensorDimens(cont.getStackSizes(), coor));
  if (slice.getNumNonZero()) cont.multAndAddStacks(coor, slice, out, &out);
}
```

This code is used in section 434.

448.

⟨ **WorkerFoldMAASparse4** constructor code 448 ⟩ ≡

```
WorkerFoldMAASparse4::WorkerFoldMAASparse4(const FoldedStackContainer
    &container, const FSSparseTensor &ten, FGSTensor &outten, const IntSequence &c)
  : cont(container), t(ten), out(outten), coor(c) { }
```

This code is used in section 434.

449. This is almost the same as ⟨ **UnfoldedStackContainer** :: *multAndAddStacks* code 461 ⟩. The only difference is that we do not construct a **UPSTensor** from **KronProdStack**, but we construct partially folded permuted symmetry **FPSTensor**. Note that the tensor g must be unfolded in order to be able to multiply with unfolded rows of Kronecker product. However, columns of such a product are partially folded giving a rise to the **FPSTensor**.

⟨ **FoldedStackContainer** :: *multAndAddStacks* dense code 449 ⟩ ≡

```
void FoldedStackContainer::multAndAddStacks(const IntSequence &coor, const FGSTensor
        &g, FGSTensor &out, const void *ad) const
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  UGSTensor ug(g);
  UFSTensor dummy_u(0, numStacks(), g.dimen());
  for (Tensor::index ui = dummy_u.begin(); ui ≠ dummy_u.end(); ++ui) {
    IntSequence tmp(ui.getCoor());
    tmp.sort();
    if (tmp ≡ coor) {
      Permutation sort_per(ui.getCoor());
      sort_per.inverse();
      for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
        if ((*it).numClasses() ≡ g.dimen()) {
          StackProduct⟨FGSTensor⟩ sp(*this, *it, sort_per, out.getSym());
          if (¬sp.isZero(coor)) {
            KronProdStack⟨FGSTensor⟩ kp(sp, coor);
            if (ug.getSym().isFull()) kp.optimizeOrder();
            FPSTensor fps(out.getDims(), *it, sort_per, ug, kp);
            {
              SYNCHRO syn(ad, "multAndAddStacks");
              fps.addTo(out);
            }
          }
        }
      }
    }
  }
}
```

This code is cited in section 450.

This code is used in section 434.

450. This is almost the same as ⟨ **FoldedStackContainer** :: *multAndAddStacks* dense code 449 ⟩. The only difference is that the Kronecker product of the stacks is multiplied with sparse slice **GSSparseTensor** (not dense slice **FGSTensor**). The multiplication is done in ⟨ **FPSTensor** sparse constructor 343 ⟩.

⟨ **FoldedStackContainer** :: *multAndAddStacks* sparse code 450 ⟩ ≡

```
void FoldedStackContainer::multAndAddStacks(const IntSequence &coor, const
        GSSparseTensor &g, FGSTensor &out, const void *ad) const
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  UFSTensor dummy_u(0, numStacks(), g.dimen());
  for (Tensor::index ui = dummy_u.begin(); ui ≠ dummy_u.end(); ++ui) {
    IntSequence tmp(ui.getCoor());
    tmp.sort();
    if (tmp ≡ coor) {
      Permutation sort_per(ui.getCoor());
      sort_per.inverse();
      for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
        if ((*it).numClasses() ≡ g.dimen()) {
          StackProduct⟨FGSTensor⟩ sp(*this, *it, sort_per, out.getSym());
          if (¬sp.isZero(coor)) {
            KronProdStack⟨FGSTensor⟩ kp(sp, coor);
            FPSTensor fps(out.getDims(), *it, sort_per, g, kp);
            {
              SYNCHRO syn(ad, "multAndAddStacks");
              fps.addTo(out);
            }
          }
        }
      }
    }
  }
}
```

This code is cited in sections 446 and 459.

This code is used in section 434.

451. Here we simply call either *multAndAddSparse1* or *multAndAddSparse2*. The first one allows for optimization of Kronecker products, so it seems to be more efficient.

⟨ **UnfoldedStackContainer** :: *multAndAdd* sparse code 451 ⟩ ≡

```
void UnfoldedStackContainer::multAndAdd(const FSSparseTensor &t, UGSTensor &out) const
{
  TL_RAISE_IF(t.nvar() ≠ getAllSize(),
      "Wrong␣number␣of␣variables␣of␣tensor␣for␣UnfoldedStackContainer::multAndAdd");
  multAndAddSparse2(t, out);
}
```

This code is used in section 434.

452. Here we implement the formula for stacks for fully symmetric tensor scattered in a number of general symmetry tensors contained in a given container. The implementations is pretty the same as in *multAndAddSparse2* but we do not do the slices of sparse tensor, but only a lookup to the container.

This means that we do not iterate through a dummy folded tensor to obtain folded coordinates of stacks, rather we iterate through all symmetries contained in the container and the coordinates of stacks are obtained as unfolded identity sequence via the symmetry. The reason of doing this is that we are unable to calculate symmetry from stack coordinates as easily as stack coordinates from the symmetry.

⟨ **UnfoldedStackContainer** :: *multAndAdd* dense code 452 ⟩ ≡

```
void UnfoldedStackContainer::multAndAdd(int dim, const UGSContainer &c, UGSTensor
        &out) const
{
  TL_RAISE_IF(c.num() ≠ numStacks(),
      "Wrong␣symmetry␣length␣of␣container␣for␣UnfoldedStackContainer::multAndAdd");
  THREAD_GROUP gr;
  SymmetrySet ss(dim, c.num());
  for (symiterator si(ss); ¬si.isEnd(); ++si) {
    if (c.check(*si)) {
      THREAD * worker = new WorkerUnfoldMAADense(*this, *si, c, out);
      gr.insert(worker);
    }
  }
  gr.run();
}
```

This code is cited in section 436.

This code is used in section 434.

453.

⟨ **WorkerUnfoldMAADense** :: **operator**( )( ) code 453 ⟩ ≡

```
void WorkerUnfoldMAADense::operator()()
{
  Permutation iden(dense_cont.num());
  IntSequence coor(sym, iden.getMap());
  const UGSTensor *g = dense_cont.get(sym);
  cont.multAndAddStacks(coor, *g, out, &out);
}
```

This code is cited in section 437.

This code is used in section 434.

454.

⟨ **WorkerUnfoldMAADense** constructor code 454 ⟩ ≡

```
WorkerUnfoldMAADense::WorkerUnfoldMAADense(const UnfoldedStackContainer
        &container, const Symmetry &s, const UGSContainer &dcontainer, UGSTensor
        &outten)
  : cont(container), sym(s), dense_cont(dcontainer), out(outten) { }
```

This code is used in section 434.

455. Here we implement the formula for unfolded tensors. If, for instance, a coordinate z of a tensor $[f_{z^2}]$ is partitioned as $z=[a,b]$, then we perform the following:

$$[f_{z^2}]\left(\sum_c \begin{bmatrix}a_{c(x)}\\ b_{c(y)}\end{bmatrix}\otimes\begin{bmatrix}a_{c(y)}\\ b_{c(y)}\end{bmatrix}\right)=[f_{aa}]\left(\sum_c a_{c(x)}\otimes a_{c(y)}\right)+[f_{ab}]\left(\sum_c a_{c(x)}\otimes b_{c(y)}\right)+ \\ [f_{ba}]\left(\sum_c b_{c(x)}\otimes a_{c(y)}\right)+[f_{bb}]\left(\sum_c b_{c(x)}\otimes b_{c(y)}\right)$$

This is exactly what happens here. The code is clear. It goes through all combinations of stacks, and each thread is responsible for operation for the slice corresponding to the combination of the stacks.

⟨ **UnfoldedStackContainer** :: *multAndAddSparse1* code 455 ⟩ ≡

```
void UnfoldedStackContainer::multAndAddSparse1(const FSSparseTensor &t, UGSTensor
        &out) const
{
  THREAD_GROUP gr;
  UFSTensor dummy(0, numStacks(), t.dimen());
  for (Tensor::index ui = dummy.begin(); ui ≠ dummy.end(); ++ui) {
    THREAD *worker = new WorkerUnfoldMAASparse1(*this, t, out, ui.getCoor());
    gr.insert(worker);
  }
  gr.run();
}
```

This code is cited in sections 439 and 456.

This code is used in section 434.

456. This does a step of ⟨ **UnfoldedStackContainer** :: *multAndAddSparse1* code 455 ⟩ for a given coordinates. First it makes the slice of the given stack coordinates. Then it multiplies everything what should be multiplied with the slice. That is it goes through all equivalences, creates **StackProduct**, then **KronProdStack**, which is added to *out*. So far everything is clear.

However, we want to use optimized **KronProdAllOptim** to minimize a number of flops and memory needed in the Kronecker product. So we go through all permutations *per*, permute the coordinates to get *percoor*, go through all equivalences, and make **KronProdStack** and optimize it. The result of optimization is a permutation *oper*. Now, we multiply the Kronecker product with the slice, only if the slice has the same ordering of coordinates as the Kronecker product **KronProdStack**. However, it is not perfectly true. Since we go through **all** permutations *per*, there might be two different permutations leading to the same ordering in **KronProdStack** and thus the same ordering in the optimized **KronProdStack**. The two cases would be counted twice, which is wrong. That is why we do not condition on $\mathrm{coor} \circ \mathrm{oper} \circ \mathrm{per} = \mathrm{coor}$, but we condition on $\mathrm{oper} \circ \mathrm{per} = \mathrm{id}$. In this way, we rule out permutations *per* leading to the same ordering of stacks when applied on *coor*.

todo: vertically narrow slice and out according to the fill in t.

⟨ **WorkerUnfoldMAASparse1** :: **operator** ( )( ) code 456 ⟩ ≡

```
void WorkerUnfoldMAASparse1::operator()()
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  const PermutationSet &pset = tls.pbundle→get(t.dimen());
  Permutation iden(t.dimen());
  UPSTensor slice(t, cont.getStackSizes(), coor, PerTensorDimens(cont.getStackSizes(), coor));
  for (int iper = 0; iper < pset.getNum(); iper++) {
    const Permutation &per = pset.get(iper);
    IntSequence percoor(coor.size());
    per.apply(coor, percoor);
    for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
      if ((*it).numClasses() ≡ t.dimen()) {
        StackProduct⟨UGSTensor⟩ sp(cont, *it, out.getSym());
        if (¬sp.isZero(percoor)) {
          KronProdStack⟨UGSTensor⟩ kp(sp, percoor);
          kp.optimizeOrder();
          const Permutation &oper = kp.getPer();
          if (Permutation(oper, per) ≡ iden) {
            UPSTensor ups(out.getDims(), *it, slice, kp);
            {
              SYNCHRO syn(&out, "WorkerUnfoldMAASparse1");
              ups.addTo(out);
            }
          }
        }
      }
    }
  }
}
```

This code is cited in section 440.

This code is used in section 434.

457.

⟨ **WorkerUnfoldMAASparse1** constructor code 457 ⟩ ≡

```
WorkerUnfoldMAASparse1::WorkerUnfoldMAASparse1(const UnfoldedStackContainer
        &container, const FSSparseTensor &ten, UGSTensor &outten, const IntSequence &c)
  : cont(container), t(ten), out(outten), coor(c), ebundle(*(tls.ebundle)) { }
```

This code is used in section 434.

458. In here we implement the formula by a bit different way. We use the fact, using notation of ⟨ **UnfoldedStackContainer** :: *multAndAddSparse2* code 458 ⟩, that

$$[f_{ba}]\left(\sum_c b_{c(x)}\otimes a_{c(y)}\right)=[f_{ab}]\left(\sum_c a_{c(y)}\otimes b_{c(b)}\right)\cdot P$$

where P is a suitable permutation of columns. The permutation corresponds to (in this example) a swap of a and b. An advantage of this approach is that we do not need **UPSTensor** for f_{ba}, and thus we decrease the number of needed slices.

So we go through all folded indices of stack coordinates, then for each such index *fi* we make a slice and call *multAndAddStacks*. This goes through all corresponding unfolded indices to perform the formula. Each unsorted (unfold) index implies a sorting permutation *sort_per* which must be used to permute stacks in **StackProduct**, and permute equivalence classes when **UPSTensor** is formed. In this way the column permutation P from the formula is factored to the permutation of **UPSTensor**.

⟨ **UnfoldedStackContainer** :: *multAndAddSparse2* code 458 ⟩ ≡

```
void UnfoldedStackContainer::multAndAddSparse2(const FSSparseTensor &t, UGSTensor
        &out) const
{
  THREAD_GROUP gr;
  FFSTensor dummy_f(0, numStacks(), t.dimen());
  for (Tensor::index fi = dummy_f.begin(); fi ≠ dummy_f.end(); ++fi) {
    THREAD *worker = new WorkerUnfoldMAASparse2(*this, t, out, fi.getCoor());
    gr.insert(worker);
  }
  gr.run();
}
```

This code is cited in sections 442, 458, and 459.

This code is used in section 434.

459. This does a step of ⟨ **UnfoldedStackContainer** :: *multAndAddSparse2* code 458 ⟩ for a given coordinates.

todo: implement *multAndAddStacks* for sparse slice as ⟨ **FoldedStackContainer** :: *multAndAddStacks* sparse code 450 ⟩ and do this method as ⟨ **WorkerFoldMAASparse2** :: **operator** ( )( ) code 443 ⟩.

⟨ **WorkerUnfoldMAASparse2** :: **operator** ( )( ) code 459 ⟩ ≡

```
void WorkerUnfoldMAASparse2::operator()()
{
  GSSparseTensor slice(t, cont.getStackSizes(), coor, TensorDimens(cont.getStackSizes(), coor));
  if (slice.getNumNonZero()) {
    FGSTensor fslice(slice);
    UGSTensor dense_slice(fslice);
    int r1 = slice.getFirstNonZeroRow();
    int r2 = slice.getLastNonZeroRow();
    UGSTensor dense_slice1(r1, r2 - r1 + 1, dense_slice);
    UGSTensor out1(r1, r2 - r1 + 1, out);
    cont.multAndAddStacks(coor, dense_slice1, out1, &out);
  }
}
```

This code is used in section 434.

460.

⟨ **WorkerUnfoldMAASparse2** constructor code 460 ⟩ ≡

```
WorkerUnfoldMAASparse2::WorkerUnfoldMAASparse2(const UnfoldedStackContainer
    &container, const FSSparseTensor &ten, UGSTensor &outten, const IntSequence &c)
: cont(container), t(ten), out(outten), coor(c) { }
```

This code is used in section 434.

461. For a given unfolded coordinates of stacks *fi*, and appropriate tensor g, whose symmetry is a symmetry of *fi*, the method contributes to *out* all tensors in unfolded stack formula involving stacks chosen by *fi*.

We go through all *ui* coordinates which yield *fi* after sorting. We construct a permutation *sort_per* which sorts *ui* to *fi*. We go through all appropriate equivalences, and construct **StackProduct** from equivalence classes permuted by *sort_per*, then **UPSTensor** with implied permutation of columns by the permuted equivalence by *sort_per*. The **UPSTensor** is then added to *out*.

We cannot use here the optimized **KronProdStack**, since the symmetry of **UGSTensor** &*g* prescribes the ordering of the stacks. However, if g is fully symmetric, we can do the optimization harmlessly.

⟨ **UnfoldedStackContainer** :: *multAndAddStacks* code 461 ⟩ ≡

```
void UnfoldedStackContainer::multAndAddStacks(const IntSequence &fi, const UGSTensor
        &g, UGSTensor &out, const void *ad) const
{
  const EquivalenceSet &eset = ebundle.get(out.dimen());
  UFSTensor dummy_u(0, numStacks(), g.dimen());
  for (Tensor::index ui = dummy_u.begin(); ui ≠ dummy_u.end(); ++ui) {
    IntSequence tmp(ui.getCoor());
    tmp.sort();
    if (tmp ≡ fi) {
      Permutation sort_per(ui.getCoor());
      sort_per.inverse();
      for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
        if ((*it).numClasses() ≡ g.dimen()) {
          StackProduct⟨UGSTensor⟩ sp(*this, *it, sort_per, out.getSym());
          if (¬sp.isZero(fi)) {
            KronProdStack⟨UGSTensor⟩ kp(sp, fi);
            if (g.getSym().isFull()) kp.optimizeOrder();
            UPSTensor ups(out.getDims(), *it, sort_per, g, kp);
            {
              SYNCHRO syn(ad, "multAndAddStacks");
              ups.addTo(out);
            }
          }
        }
      }
    }
  }
}
```

This code is cited in section 449.

This code is used in section 434.

462. End of `stack_container.cpp` file.

463. Refined stack of containers. Start of `fine_container.h` file.

This file defines a refinement of the stack container. It makes a vertical refinement of a given stack container, it refines only matrix items, the items which are always zero, or can be identity matrices are not refined.

The refinement is done by a simple construction from the stack container being refined. A parameter is passed meaning a maximum size of each stack in the refined container. The resulting object is stack container, so everything works seamlessly.

We define here a class for refinement of sizes **SizeRefinement**, this is purely an auxiliary class allowing us to write a code more concisely. The main class of this file is **FineContainer**, which corresponds to refining. The two more classes **FoldedFineContainer** and **UnfoldedFineContainer** are its specializations.

NOTE: This code was implemented with a hope that it will help to cut down memory allocations during the Faa Di Bruno formula evaluation. However, it seems that this needs to be accompanied with a similar thing for tensor multidimensional index. Thus, the abstraction is not currently used, but it might be useful in future.

```
#ifndef FINE_CONTAINER_H
#define FINE_CONTAINER_H
#include "stack_container.h"
#include <vector>
  ⟨ SizeRefinement class declaration 464 ⟩;
  ⟨ FineContainer class declaration 465 ⟩;
  ⟨ FoldedFineContainer class declaration 468 ⟩;
  ⟨ UnfoldedFineContainer class declaration 469 ⟩;
#endif
```

464. This class splits the first *nc* elements of the given sequence *s* to a sequence not having items greater than given *max*. The remaining elements (those behind *nc*) are left untouched. It also remembers the mapping, i.e. for a given index in a new sequence, it is able to return a corresponding index in old sequence.

```
⟨ SizeRefinement class declaration 464 ⟩ ≡
  class SizeRefinement {
    vector⟨int⟩ rsizes;
    vector⟨int⟩ ind_map;
    int new_nc;
  public:
    SizeRefinement(const IntSequence &s, int nc, int max);
    int getRefSize(int i) const
    { return rsizes[i]; }
    int numRefinements() const
    { return rsizes.size(); }
    int getOldIndex(int i) const
    { return ind_map[i]; }
    int getNC() const
    { return new_nc; }
  };
```

This code is used in section 463.

465. This main class of this class refines a given stack container, and inherits from the stack container. It also defines the *getType* method, which returns a type for a given stack as the type of the corresponding (old) stack of the former stack container.

⟨ **FineContainer** class declaration 465 ⟩ ≡

```
template⟨class _Ttype⟩
class FineContainer : public SizeRefinement, public StackContainer⟨_Ttype⟩ {
protected:
  typedef StackContainer⟨_Ttype⟩ _Stype;
  typedef typename StackContainerInterface⟨_Ttype⟩::_Ctype _Ctype;
  typedef typename StackContainerInterface⟨_Ttype⟩::itype itype;
  _Ctype **const ref_conts;
  const _Stype &stack_cont;
public:
  ⟨ FineContainer constructor 466 ⟩;
  ⟨ FineContainer destructor 467 ⟩;
  itype getType(int i, const Symmetry &s) const
  { return stack_cont.getType(getOldIndex(i), s); }
};
```

This code is used in section 463.

466. Here we construct the **SizeRefinement** and allocate space for the refined containers. Then, the containers are created and put to *conts* array. Note that the containers do not claim any further space, since all the tensors of the created containers are in-place submatrices.

Here we use a dirty trick of converting **const** pointer to non-**const** pointer and passing it to a subtensor container constructor. The containers are stored in *ref_conts* and then in *conts* from **StackContainer**. However, this is safe since neither *ref_conts* nor *conts* are used in non-**const** contexts. For example, **StackContainer** has only a **const** method to return a member of *conts*.

```
⟨ FineContainer constructor 466 ⟩ ≡
  FineContainer(const _Stype &sc, int max)
  : SizeRefinement(sc.getStackSizes(), sc.numConts(), max),
        StackContainer⟨_Ttype⟩(numRefinements(), getNC()), ref_conts(new _Ctype*[getNC()]),
        stack_cont(sc) {
    for (int i = 0; i < numRefinements(); i++) _Stype::stack_sizes[i] = getRefSize(i);
    _Stype::calculateOffsets();
    int last_cont = −1;
    int last_row = 0;
    for (int i = 0; i < getNC(); i++) {
      if (getOldIndex(i) ≠ last_cont) {
        last_cont = getOldIndex(i);
        last_row = 0;
      }
      union {
        const _Ctype *c;
        _Ctype *n;
      } convert;
      convert.c = stack_cont.getCont(last_cont);
      ref_conts[i] = new _Ctype(last_row, _Stype::stack_sizes[i], *(convert.n));
      _Stype::conts[i] = ref_conts[i];
      last_row += _Stype::stack_sizes[i];
    }
  }
```

This code is used in section 465.

467. Here we deallocate the refined containers, and deallocate the array of refined containers.

```
⟨ FineContainer destructor 467 ⟩ ≡
  virtual ~FineContainer()
  {
    for (int i = 0; i < _Stype::numConts(); i++) delete ref_conts[i];
    delete[] ref_conts;
  }
```

This code is used in section 465.

468. Here is **FineContainer** specialization for folded tensors.

⟨ **FoldedFineContainer** class declaration 468 ⟩ ≡
class FoldedFineContainer : **public FineContainer**⟨**FGSTensor**⟩, **public FoldedStackContainer** {
public:
FoldedFineContainer(**const StackContainer**⟨**FGSTensor**⟩ &*sc*, **int** *max*)
: **FineContainer**⟨**FGSTensor**⟩(*sc*, *max*) { }
};

This code is used in section 463.

469. Here is **FineContainer** specialization for unfolded tensors.

⟨ **UnfoldedFineContainer** class declaration 469 ⟩ ≡
class UnfoldedFineContainer : **public FineContainer**⟨**UGSTensor**⟩, **public UnfoldedStackContainer** {
public:
UnfoldedFineContainer(**const StackContainer**⟨**UGSTensor**⟩ &*sc*, **int** *max*)
: **FineContainer**⟨**UGSTensor**⟩(*sc*, *max*) { }
};

This code is used in section 463.

470. End of `fine_container.h` file.

471. Start of `stack_container.cpp` file.

#**include** "fine_container.h"
#**include** <cmath>
⟨ **SizeRefinement** constructor code 472 ⟩;

472. Here we construct the vector of new sizes of containers (before nc) and copy all remaining sizes behind nc.

⟨ **SizeRefinement** constructor code 472 ⟩ ≡

```
SizeRefinement::SizeRefinement(const IntSequence &s, int nc, int max)
{
  new_nc = 0;
  for (int i = 0; i < nc; i++) {
    int nr = s[i]/max;
    if (s[i] % max ≠ 0) nr++;
    int ss = (nr > 0) ? (int) round(((double) s[i])/nr) : 0;
    for (int j = 0; j < nr − 1; j++) {
      rsizes.push_back(ss);
      ind_map.push_back(i);
      new_nc++;
    }
    rsizes.push_back(s[i] − (nr − 1) ∗ ss);
    ind_map.push_back(i);
    new_nc++;
  }
  for (int i = nc; i < s.size(); i++) {
    rsizes.push_back(s[i]);
    ind_map.push_back(i);
  }
}
```

This code is used in section 471.

473. End of `stack_container.cpp` file.

474. Multiplying tensor columns. Start of `pyramid_prod.h` file.

In here, we implement the Faa Di Bruno for folded tensors. Recall, that one step of the Faa Di Bruno is a formula:

$$[B_{s^k}]_{\alpha_1\ldots\alpha_k} = [h_{y^l}]_{\gamma_1\ldots\gamma_l}\prod_{m=1}^{l}\left[g_{s^{|c_m|}}\right]^{\gamma_m}_{c_m(\alpha)}$$

In contrast to unfolded implementation of **UGSContainer**::*multAndAdd* with help of **KronProdAll** and **UPSTensor**, we take a completely different strategy. We cannot afford full instantiation of

$$\sum_{c\in M_{l,k}}\prod_{m=1}^{l}\left[g_{s^{|c_m|}}\right]^{\gamma_m}_{c_m(\alpha)}$$

and therefore we do it per partes. We select some number of columns, for instance 10, calculate 10 continuous iterators of tensor B. Then we form unfolded tensor

$$[G]_S^{\gamma_1\ldots\gamma_l} = \left[\sum_{c\in M_{l,k}}\prod_{m=1}^{l}\left[g_{s^{|c_m|}}\right]^{\gamma_m}_{c_m(\alpha)}\right]_S$$

where S is the selected set of 10 indices. This is done as Kronecker product of vectors corresponding to selected columns. Note that, in general, there is no symmetry in G, its type is special class for this purpose.

If g is folded, then we have to form folded version of G. There is no symmetry in G data, so we sum all unfolded indices corresponding to folded index together. This is perfectly OK, since we multiply these groups of (equivalent) items with the same number in fully symmetric g.

After this, we perform ordinary matrix multiplication to obtain a selected set of columns of B.

In here, we define a class for forming and representing $[G]_S^{\gamma_1\ldots\gamma_l}$. Basically, this tensor is row-oriented (multidimensional index is along rows), and it is fully symmetric. So we inherit from **URTensor**. If we need its folded version, we simply use a suitable conversion. The new abstraction will have only a new constructor allowing a construction from the given set of indices S, and given set of tensors g. The rest of the process is implemented in ⟨**FGSContainer**::*multAndAdd* unfolded code 395⟩ or ⟨**FGSContainer**::*multAndAdd* folded code 394⟩.

```
#ifndef PYRAMID_PROD_H
#define PYRAMID_PROD_H
#include "int_sequence.h"
#include "rfs_tensor.h"
#include "gs_tensor.h"
#include "t_container.h"
#include <vector>
  using namespace std;
  ⟨USubTensor class declaration 475⟩;
#endif
```

475. Here we define the new tensor for representing $[G]_S^{\gamma_1\ldots\gamma_l}$. It allows a construction from container of folded general symmetry tensors *cont*, and set of indices *ts*. Also we have to supply dimensions of resulting tensor B, and dimensions of tensor h.

⟨ **USubTensor** class declaration 475 ⟩ ≡

```
class USubTensor : public URTensor {
public:
  USubTensor(const TensorDimens &bdims, const TensorDimens &hdims, const FGSContainer
      &cont, const vector⟨IntSequence⟩ &lst);
  void addKronColumn(int i, const vector⟨const FGSTensor *⟩ &ts, const IntSequence &pindex);
};
```

This code is cited in section 1.

This code is used in section 474.

476. End of `pyramid_prod.h` file.

477. Start of `pyramid_prod.cpp` file.

```
#include "pyramid_prod.h"
#include "permutation.h"
#include "tl_exception.h"
  ⟨ USubTensor constructor code 478 ⟩;
  ⟨ USubTensor::addKronColumn code 479 ⟩;
```

478. Here we construct the **USubTensor** object. We allocate space via the parent **URTensor**. Number of columns is a length of the list of indices *lst*, number of variables and dimensions are of the tensor h, this is given by *hdims*.

We go through all equivalences with number of classes equal to dimension of B. For each equivalence we make a permutation *per*. Then we fetch all the necessary tensors g with symmetries implied by symmetry of B and the equivalence. Then we go through the list of indices, permute them by the permutation and add the Kronecker product of the selected columns. This is done by *addKronColumn*.

⟨ **USubTensor** constructor code 478 ⟩ ≡

```
USubTensor::USubTensor(const TensorDimens &bdims, const TensorDimens &hdims, const
        FGSContainer &cont, const vector⟨IntSequence⟩ &lst)
: URTensor(lst.size(), hdims.getNVX()[0], hdims.dimen()) {
  TL_RAISE_IF(¬hdims.getNVX().isConstant(),
      "Tensor␣has␣not␣full␣symmetry␣in␣USubTensor()");
  const EquivalenceSet &eset = cont.getEqBundle().get(bdims.dimen());
  zeros();
  for (EquivalenceSet::const_iterator it = eset.begin(); it ≠ eset.end(); ++it) {
    if ((*it).numClasses() ≡ hdims.dimen()) {
      Permutation per(*it);
      vector⟨const FGSTensor *⟩ ts = cont.fetchTensors(bdims.getSym(), *it);
      for (int i = 0; i < (int) lst.size(); i++) {
        IntSequence perindex(lst[i].size());
        per.apply(lst[i], perindex);
        addKronColumn(i, ts, perindex);
      }
    }
  }
}
```

This code is used in section 477.

479. This makes a Kronecker product of appropriate columns from tensors in *fs* and adds such data to *i*-th column of this matrix. The appropriate columns are defined by *pindex* sequence. A column of a tensor has index created from a corresponding part of *pindex*. The sizes of these parts are given by dimensions of the tensors in *ts*.

Here we break the given index *pindex* according to the dimensions of the tensors in *ts*, and for each subsequence of the *pindex* we find an index of the folded tensor, which involves calling *getOffset* for folded tensor, which might be costly. We gather all columns to a vector *tmpcols* which are Kronecker multiplied in constructor of **URSingleTensor**. Finally we add data of **URSingleTensor** to the *i*-th column.

⟨ **USubTensor** :: *addKronColumn* code 479 ⟩ ≡

```
void USubTensor::addKronColumn(int i, const vector⟨const FGSTensor *⟩ &ts, const
        IntSequence &pindex)
  {
    vector⟨ConstVector⟩ tmpcols;
    int lastdim = 0;
    for (unsigned int j = 0; j < ts.size(); j++) {
      IntSequence ind(pindex, lastdim, lastdim + ts[j]→dimen());
      lastdim += ts[j]→dimen();
      index in(ts[j], ind);
      tmpcols.push_back(ConstVector(*(ts[j]), *in));
    }
    URSingleTensor kronmult(tmpcols);
    Vector coli(*this, i);
    coli.add(1.0, kronmult.getData());
  }
```

This code is used in section 477.

480. End of `pyramid_prod.cpp` file.

481. Multiplying stacked tensor columns. Start of `pyramid_prod2.h` file.

We need to calculate the following tensor product:

$$[f_{s^j}]_{\alpha_1\ldots\alpha_j} = \sum_{l=1}^{j} [f_{z^l}]_{\beta_1\ldots\beta_l} \sum_{c\in M_{l,j}} \prod_{m=1}^{l} [z_{c_m}]^{\beta_m}_{c_m(\alpha)}$$

where $s=[y,u,u',\sigma]$, and z is a composition of four variables, say $[v,w,y,u]$. Note that z ends with y and u, and the only non-zero derivative of the trailing part of z involving y or u is the first derivative and is the unit matrix $y_y=[1]$ or $u_u=[1]$. Also, we suppose that the dependence of v, and w on s is such that whenever derivative of w is nonzero, then also of v. This means that there for any derivative and any index there is a continuous part of derivatives of v and optionally of w followed by column of zeros containing at most one 1.

This structure can be modelled and exploited with some costs at programming. For example, let us consider the following product:

$$\left[B_{y^2u^3}\right]_{\alpha_1\alpha_2\beta_1\beta_2\beta_3} = \ldots [f_{z^3}]_{\gamma_1\gamma_2\gamma_3} [z_{yu}]^{\gamma_1}_{\alpha_1\beta_1} [z_y]^{\gamma_2}_{\alpha_2} [z_{uu}]^{\gamma_3}_{\beta_2\beta_3} \ldots$$

The term corresponds to equivalence $\{\{0,2\},\{1\},\{3,4\}\}$. For the fixed index $\alpha_1\alpha_2\beta_1\beta_2\beta_3$ we have to make a Kronecker product of the columns

$$[z_{yu}]_{\alpha_1\beta_1} \otimes [z_y]_{\alpha_2} \otimes [z_{uu}]_{\beta_2\beta_3}$$

which can be written as

$$\begin{bmatrix} [v_{yu}]_{\alpha_1\beta_1} \\ [w_{yu}]_{\alpha_1\beta_1} \\ 0 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} [v_y]_{\alpha_2} \\ [w_y]_{\alpha_2} \\ 1_{\alpha_2} \\ 0 \end{bmatrix} \otimes \begin{bmatrix} [v_{uu}]_{\beta_2\beta_3} \\ [w_{uu}]_{\beta_2\beta_3} \\ 0 \\ 0 \end{bmatrix}$$

where 1_{α_2} is a column of zeros having the only 1 at α_2 index.

This file develops the abstraction for this Kronecker product column without multiplication of the zeros at the top. Basically, it will be a column which is a Kronecker product of the columns without the zeros:

$$\begin{bmatrix} [v_{yu}]_{\alpha_1\beta_1} \\ [w_{yu}]_{\alpha_1\beta_1} \end{bmatrix} \otimes \begin{bmatrix} [v_y]_{\alpha_2} \\ [w_y]_{\alpha_2} \\ 1 \end{bmatrix} \otimes \begin{bmatrix} [v_{uu}]_{\beta_2\beta_3} \\ [w_{uu}]_{\beta_2\beta_3} \end{bmatrix}$$

The class will have a tensor infrastructure introducing **index** which iterates over all items in the column with $\gamma_1\gamma_2\gamma_3$ as coordinates in $[f_{z^3}]$. The data of such a tensor is not suitable for any matrix operation and will have to be accessed only through the **index**. Note that this does not matter, since $[f_{z^l}]$ are sparse.

```
#ifndef PYRAMID_PROD2_H
#define PYRAMID_PROD2_H
#include "permutation.h"
#include "tensor.h"
#include "tl_exception.h"
#include "rfs_tensor.h"
#include "stack_container.h"
#include "Vector.h"
  ⟨IrregTensorHeader class declaration 482⟩;
  ⟨IrregTensor class declaration 483⟩;
#endif
```

482. First we declare a helper class for the tensor. Its purpose is to gather the columns which are going to be Kronecker multiplied. The input of this helper class is **StackProduct⟨FGSTensor⟩** and coordinate c of the column.

It maintains *unit_flag* array which says for what columns we must stack 1 below v and w. In this case, the value of *unit_flag* is an index of the 1, otherwise the value of *unit_flag* is -1.

Also we have storage for the stacked columns *cols*. The object is responsible for memory management associated to this storage. That is why we do not allow any copy constructor, since we need to be sure that no accidental copies take place. We declare the copy constructor as private and not implement it.

⟨ **IrregTensorHeader** class declaration 482 ⟩ ≡

```
class IrregTensor;
class IrregTensorHeader {
  friend class IrregTensor;
  int nv;
  IntSequence unit_flag;
  Vector **const cols;
  IntSequence end_seq;
public:
  IrregTensorHeader(const StackProduct<FGSTensor> &sp, const IntSequence &c);
  ~IrregTensorHeader();
  int dimen() const
  { return unit_flag.size(); }
  void increment(IntSequence &v) const;
  int calcMaxOffset() const;
private:
  IrregTensorHeader(const IrregTensorHeader &);
};
```

This code is used in section 481.

483. Here we declare the irregular tensor. There is no special logic here. We inherit from **Tensor** and we must implement three methods, *increment*, *decrement* and *getOffset*. The last two are not implemented now, since they are not needed, and they raise an exception. The first just calls *increment* of the header. Also we declare a method *addTo* which adds this unfolded irregular single column tensor to folded (regular) single column tensor.

The header **IrregTensorHeader** lives with an object by a reference. This is dangerous. However, we will use this class only in a simple loop and both **IrregTensor** and **IrregTensorHeader** will be destructed at the end of a block. Since the super class **Tensor** must be initialized before any member, we could do either a save copy of **IrregTensorHeader**, or relatively dangerous the reference member. For the reason above we chose the latter.

```
⟨ IrregTensor class declaration 483 ⟩ ≡
  class IrregTensor : public Tensor {
    const IrregTensorHeader &header;
  public:
    IrregTensor(const IrregTensorHeader &h);
    void addTo(FRSingleTensor &out) const;
    void increment(IntSequence &v) const
    { header.increment(v); }
    void decrement(IntSequence &v) const
    { TL_RAISE("Not␣implemented␣error␣in␣IrregTensor::decrement"); }
    int getOffset(const IntSequence &v) const
    { TL_RAISE("Not␣implemented␣error␣in␣IrregTensor::getOffset"); return 0; }
  };
```

This code is cited in section 1.

This code is used in section 481.

484. End of `pyramid_prod2.h` file.

485. Start of `pyramid_prod2.cpp` file.

```
#include "pyramid_prod2.h"
#include "rfs_tensor.h"
  ⟨ IrregTensorHeader constructor code 486 ⟩;
  ⟨ IrregTensorHeader::increment code 487 ⟩;
  ⟨ IrregTensorHeader destructor code 489 ⟩;
  ⟨ IrregTensorHeader::calcMaxOffset code 490 ⟩;
  ⟨ IrregTensor constructor code 491 ⟩;
  ⟨ IrregTensor::addTo code 492 ⟩;
```

486. Here we only call *sp.createPackedColumns*(*c*, *cols*, *unit_flag*) which fills *cols* and *unit_flag* for the given column *c*. Then we set *end_seq* according to *unit_flag* and columns lengths.

⟨ **IrregTensorHeader** constructor code 486 ⟩ ≡

```
IrregTensorHeader::IrregTensorHeader(const StackProduct⟨FGSTensor⟩ &sp, const
        IntSequence &c)
  : nv(sp.getAllSize()), unit_flag(sp.dimen()), cols(new Vector*[sp.dimen()]), end_seq(sp.dimen()) {
    sp.createPackedColumns(c, cols, unit_flag);
    for (int i = 0; i < sp.dimen(); i++) {
      end_seq[i] = cols[i]→length();
      if (unit_flag[i] ≠ −1) end_seq[i] = unit_flag[i] + 1;
    }
  }
```

This code is used in section 485.

487. Here we have to increment the given integer sequence. We do it by the following code, whose pattern is valid for all tensor. The only difference is how we increment item of coordinates.

⟨ **IrregTensorHeader**::*increment* code 487 ⟩ ≡

```
void IrregTensorHeader::increment(IntSequence &v) const
{
  TL_RAISE_IF(v.size() ≠ dimen(),
      "Wrong␣size␣of␣coordinates␣in␣IrregTensorHeader::increment");
  if (v.size() ≡ 0) return;
  int i = v.size() − 1;
  ⟨ increment i-th item in coordinate v 488 ⟩;
  while (i > 0 ∧ v[i] ≡ end_seq[i]) {
    v[i] = 0;
    i−−;
    ⟨ increment i-th item in coordinate v 488 ⟩;
  }
}
```

This code is used in section 485.

488. Here we increment item of coordinates. Whenever we reached end of column coming from matrices, and *unit_flag* is not −1, we have to jump to that *unit_flag*.

⟨ increment i-th item in coordinate v 488 ⟩ ≡

```
v[i]++;
if (unit_flag[i] ≠ −1 ∧ v[i] ≡ cols[i]→length() − 1) v[i] = unit_flag[i];
```

This code is used in section 487.

489.

⟨ **IrregTensorHeader** destructor code 489 ⟩ ≡

```
IrregTensorHeader::~IrregTensorHeader()
{
  for (int i = 0; i < dimen(); i++) delete cols[i];
  delete[] cols;
}
```

This code is used in section 485.

490. It is a product of all column lengths.

⟨ **IrregTensorHeader** :: *calcMaxOffset* code 490 ⟩ ≡

```
int IrregTensorHeader::calcMaxOffset() const
{
  int res = 1;
  for (int i = 0; i < dimen(); i++) res *= cols[i]->length();
  return res;
}
```

This code is used in section 485.

491. Everything is done in **IrregTensorHeader**, only we have to Kronecker multiply all columns of the header.

⟨ **IrregTensor** constructor code 491 ⟩ ≡

```
IrregTensor::IrregTensor(const IrregTensorHeader &h)
: Tensor(along_row, IntSequence(h.dimen(), 0), h.end_seq, h.calcMaxOffset(), 1, h.dimen()), header(h)
    {
  if (header.dimen() == 1) {
    getData() = *(header.cols[0]);
    return;
  }
  Vector *last = new Vector(*(header.cols[header.dimen() - 1]));
  for (int i = header.dimen() - 2; i > 0; i--) {
    Vector *newlast = new Vector(last->length() * header.cols[i]->length());
    KronProd::kronMult(ConstVector(*(header.cols[i])), ConstVector(*last), *newlast);
    delete last;
    last = newlast;
  }
  KronProd::kronMult(ConstVector(*(header.cols[0])), ConstVector(*last), getData());
  delete last;
}
```

This code is used in section 485.

492. Clear.

⟨ **IrregTensor** :: *addTo* code 492 ⟩ ≡

```
void IrregTensor::addTo(FRSingleTensor &out) const
{
  for (index it = begin(); it != end(); ++it) {
    IntSequence tmp(it.getCoor());
    tmp.sort();
    Tensor::index ind(&out, tmp);
    out.get(*ind, 0) += get(*it, 0);
  }
}
```

This code is used in section 485.

493. End of `pyramid_prod2.cpp` file.

494. Miscellany.

495. Tensor polynomial evaluation. Start of `t_polynomial.h` file.

We need to evaluate a tensor polynomial of the form:

$$[g_x]_{\alpha_1}[x]^{\alpha_1} + [g_{x^2}]_{\alpha_1\alpha_2}[x]^{\alpha_1}[x]^{\alpha_2} + \ldots + [g_{x^n}]_{\alpha_1\ldots\alpha_n}\prod_{i=1}^{n}[x]^{\alpha_i}$$

where x is a column vector.

We have basically two options. The first is to use the formula above, the second is to use a Horner-like formula:

$$\left[\cdots\left[\left[[g_{x^{n-1}}] + [g_{x^n}]_{\alpha_1\ldots\alpha_{n-1}\alpha_n}[x]^{\alpha_n}\right]_{\alpha_1\ldots\alpha_{n-2}\alpha_{n-1}}[x]^{\alpha_{n-1}}\right]\cdots\right]_{\alpha_1}[x]^{\alpha_1}$$

Alternativelly, we can put the the polynomial into a more compact form

$$[g_x]_{\alpha_1}[x]^{\alpha_1} + [g_{x^2}]_{\alpha_1\alpha_2}[x]^{\alpha_1}[x]^{\alpha_2} + \ldots + [g_{x^n}]_{\alpha_1\ldots\alpha_n}\prod_{i=1}^{n}[x]^{\alpha_i} = [G]_{\alpha_1\ldots\alpha_n}\prod_{i=1}^{n}\begin{bmatrix}1\\ x\end{bmatrix}^{\alpha_i}$$

Then the polynomial evaluation becomes just a matrix multiplication of the vector power.

Here we define the tensor polynomial as a container of full symmetry tensors and add an evaluation methods. We have two sorts of containers, folded and unfolded. For each type we declare two methods implementing the above formulas. We define classes for the compactification of the polynomial. The class derives from the tensor and has a eval method.

```
#include "t_container.h"
#include "fs_tensor.h"
#include "rfs_tensor.h"
#include "tl_static.h"
  ⟨ PowerProvider class declaration 496 ⟩;
  ⟨ TensorPolynomial class declaration 497 ⟩;
  ⟨ UTensorPolynomial class declaration 507 ⟩;
  ⟨ FTensorPolynomial class declaration 508 ⟩;
  ⟨ CompactPolynomial class declaration 509 ⟩;
  ⟨ UCompactPolynomial class declaration 512 ⟩;
  ⟨ FCompactPolynomial class declaration 513 ⟩;
```

496. Just to make the code nicer, we implement a Kronecker power of a vector encapsulated in the following class. It has *getNext* method which returns either folded or unfolded row-oriented single column Kronecker power of the vector according to the type of a dummy argument. This allows us to use the type dependent code in templates below.

The implementation of the Kronecker power is that we maintain the last unfolded power. If unfolded *getNext* is called, we Kronecker multiply the last power with a vector and return it. If folded *getNext* is called, we do the same plus we fold it.

getNext returns the vector for the first call (first power), the second power is returned on the second call, and so on.

⟨ **PowerProvider** class declaration 496 ⟩ ≡

```
class PowerProvider {
    Vector origv;
    URSingleTensor *ut;
    FRSingleTensor *ft;
    int nv;
  public:
    PowerProvider(const ConstVector &v)
    : origv(v), ut(Λ), ft(Λ), nv(v.length()) { }
    ~PowerProvider();
    const URSingleTensor &getNext(const URSingleTensor *dummy);
    const FRSingleTensor &getNext(const FRSingleTensor *dummy);
};
```

This code is used in section 495.

497. The tensor polynomial is basically a tensor container which is more strict on insertions. It maintains number of rows and number of variables and allows insertions only of those tensors, which yield these properties. The maximum dimension is maintained by *insert* method.

So we re-implement *insert* method and implement *evalTrad* (traditional polynomial evaluation) and horner-like evaluation *evalHorner*.

In addition, we implement derivatives of the polynomial and its evaluation. The evaluation of a derivative is different from the evaluation of the whole polynomial, simply because the evaluation of the derivatives is a tensor, and the evaluation of the polynomial is a vector (zero dimensional tensor). See documentation to ⟨ **TensorPolynomial**::*derivative* code 505 ⟩ and ⟨ **TensorPolynomial**::*evalPartially* code 506 ⟩ for details.

```
⟨ TensorPolynomial class declaration 497 ⟩ ≡
  template⟨class _Ttype, class _TGStype, class _Stype⟩
  class TensorPolynomial : public TensorContainer⟨_Ttype⟩ {
    int nr;
    int nv;
    int maxdim;
    typedef TensorContainer⟨_Ttype⟩ _Tparent;
    typedef typename _Tparent::_ptr _ptr;
  public:
    TensorPolynomial(int rows, int vars)
    : TensorContainer⟨_Ttype⟩(1), nr(rows), nv(vars), maxdim(0) { }
    TensorPolynomial(const TensorPolynomial⟨_Ttype, _TGStype, _Stype⟩ &tp, int k)
    : TensorContainer⟨_Ttype⟩(tp), nr(tp.nr), nv(tp.nv), maxdim(0) { derivative(k); }
    TensorPolynomial(int first_row, int num, TensorPolynomial⟨_Ttype, _TGStype, _Stype⟩ &tp)
    : TensorContainer⟨_Ttype⟩(first_row, num, tp), nr(num), nv(tp.nv), maxdim(tp.maxdim) { }
    ⟨ TensorPolynomial contract constructor code 498 ⟩;
    TensorPolynomial(const TensorPolynomial &tp)
    : TensorContainer⟨_Ttype⟩(tp), nr(tp.nr), nv(tp.nv), maxdim(tp.maxdim) { }
    int nrows() const
    { return nr; }
    int nvars() const
    { return nv; }
    ⟨ TensorPolynomial::evalTrad code 502 ⟩;
    ⟨ TensorPolynomial::evalHorner code 503 ⟩;
    ⟨ TensorPolynomial::insert code 504 ⟩;
    ⟨ TensorPolynomial::derivative code 505 ⟩;
    ⟨ TensorPolynomial::evalPartially code 506 ⟩;
  };
```

This code is used in section 495.

498. This constructor takes a tensor polynomial

$$P(x,y)=\sum_{k=0}^{m}[g_{(xy)^k}]_{\alpha_1\ldots\alpha_k}\begin{bmatrix}x\\ y\end{bmatrix}^{\alpha_1\ldots\alpha_k}$$

and for a given x it makes a polynomial

$$Q(y)=P(x,y).$$

The algorithm for each full symmetry $(xy)^k$ works with subtensors (slices) of symmetry x^iy^j (with $i+j=k$), and contracts these subtensors with respect to x^i to obtain a tensor of full symmetry y^j. Since the column x^i is calculated by **PowerProvider** we cycle for $i=1,...,m$. Then we have to add everything for $i=0$.

The code works as follows: For slicing purposes we need stack sizes *ss* corresponing to lengths of x and y, and then identity *pp* for unfolding a symmetry of the slice to obtain stack coordinates of the slice. Then we do the calculations for $i=1,\ldots,m$ and then for $i=0$.

⟨ **TensorPolynomial** contract constructor code 498 ⟩ ≡

```
TensorPolynomial(const TensorPolynomial<_Ttype, _TGStype, _Stype> &tp, const Vector
        &xval)
  : TensorContainer<_Ttype>(1), nr(tp.nrows()), nv(tp.nvars() − xval.length()), maxdim(0) {
    TL_RAISE_IF(nvars() < 0,
        "Length␣of␣xval␣too␣big␣in␣TensorPolynomial␣contract␣constructor");
    IntSequence ss(2); ss[0] = xval.length(); ss[1] = nvars();
    IntSequence pp(2); pp[0] = 0; pp[1] = 1;
    ⟨ do contraction for all i > 0 499 ⟩;
    ⟨ do contraction for i = 0 500 ⟩;
}
```

This code is used in section 497.

499. Here we setup the **PowerProvider**, and cycle through $i = 1, \dots, m$. Within the loop we cycle through $j = 0, \dots, m - i$. If there is a tensor with symmetry $(xy)^{i+j}$ in the original polynomial, we make its slice with symmetry $x^i y^j$, and *contractAndAdd* it to the tensor *ten* in the **this** polynomial with a symmetry y^j.

Note three things: First, the tensor *ten* is either created and put to **this** container or just got from the container, this is done in ⟨ initialize *ten* of dimension j 501 ⟩. Second, the contribution to the *ten* tensor must be multiplied by $\binom{i+j}{j}$, since there are exactly that number of slices of $(xy)^{i+j}$ of the symmetry $x^i y^j$ and all must be added. Third, the tensor *ten* is fully symmetric and **_TGStype** :: *contractAndAdd* works with general symmetry, that is why we have to in-place convert fully syummetric *ten* to a general symmetry tensor.

⟨ do contraction for all $i > 0$ 499 ⟩ ≡

```
PowerProvider pwp(xval);
for (int i = 1; i ≤ tp.maxdim; i++) {
  const _Stype &xpow = pwp.getNext((const _Stype *) Λ);
  for (int j = 0; j ≤ tp.maxdim − i; j++) {
    if (tp.check(Symmetry(i + j))) {
      ⟨ initialize ten of dimension j 501 ⟩;
      Symmetry sym(i, j);
      IntSequence coor(sym, pp);
      _TGStype slice(*(tp.get(Symmetry(i + j))), ss, coor, TensorDimens(sym, ss));
      slice.mult(Tensor::noverk(i + j, j));
      _TGStype tmp(*ten);
      slice.contractAndAdd(0, tmp, xpow);
    }
  }
}
```

This code is cited in section 500.

This code is used in section 498.

500. This is easy. The code is equivalent to code ⟨ do contraction for all $i > 0$ 499 ⟩ as for $i = 0$. The contraction here takes a form of a simple addition.

⟨ do contraction for $i = 0$ 500 ⟩ ≡

```
for (int j = 0; j ≤ tp.maxdim; j++) {
  if (tp.check(Symmetry(j))) {
    ⟨ initialize ten of dimension j 501 ⟩;
    Symmetry sym(0, j);
    IntSequence coor(sym, pp);
    _TGStype slice(*(tp.get(Symmetry(j))), ss, coor, TensorDimens(sym, ss));
    ten→add(1.0, slice);
  }
}
```

This code is used in section 498.

501. The pointer *ten* is either a new tensor or got from **this** container.

⟨ initialize *ten* of dimension *j* 501 ⟩ ≡

```
_Ttype *ten;
if (_Tparent::check(Symmetry(j))) {
  ten = _Tparent::get(Symmetry(j));
}
else {
  ten = new _Ttype(nrows(), nvars(), j);
  ten->zeros();
  insert(ten);
}
```

This code is cited in section 499.

This code is used in sections 499 and 500.

502. Here we cycle up to the maximum dimension, and if a tensor exists in the container, then we multiply it with the Kronecker power of the vector supplied by **PowerProvider**.

⟨ **TensorPolynomial**::*evalTrad* code 502 ⟩ ≡

```
void evalTrad(Vector &out, const ConstVector &v) const
{
  if (_Tparent::check(Symmetry(0))) out = _Tparent::get(Symmetry(0))->getData();
  else out.zeros();
  PowerProvider pp(v);
  for (int d = 1; d <= maxdim; d++) {
    const _Stype &p = pp.getNext((const _Stype *) Λ);
    Symmetry cs(d);
    if (_Tparent::check(cs)) {
      const _Ttype *t = _Tparent::get(cs);
      t->multaVec(out, p.getData());
    }
  }
}
```

This code is used in section 497.

503. Here we construct by contraction *maxdim* − 1 tensor first, and then cycle. The code is clear, the only messy thing is **new** and **delete**.

⟨ **TensorPolynomial** :: *evalHorner* code 503 ⟩ ≡

```
void evalHorner(Vector &out, const ConstVector &v) const
{
  if (_Tparent::check(Symmetry(0))) out = _Tparent::get(Symmetry(0))→getData();
  else out.zeros();
  if (maxdim ≡ 0) return;
  _Ttype *last;
  if (maxdim ≡ 1) last = new _Ttype(*(_Tparent::get(Symmetry(1))));
  else last = new _Ttype(*(_Tparent::get(Symmetry(maxdim))), v);
  for (int d = maxdim − 1; d ≥ 1; d−−) {
    Symmetry cs(d);
    if (_Tparent::check(cs)) {
      const _Ttype *nt = _Tparent::get(cs);
      last→add(1.0, ConstTwoDMatrix(*nt));
    }
    if (d > 1) {
      _Ttype *new_last = new _Ttype(*last, v);
      delete last;
      last = new_last;
    }
  }
  last→multaVec(out, v);
  delete last;
}
```

This code is used in section 497.

504. Before a tensor is inserted, we check for the number of rows, and number of variables. Then we insert and update the *maxdim*.

⟨ **TensorPolynomial** :: *insert* code 504 ⟩ ≡

```
void insert(_ptr t)
{
  TL_RAISE_IF(t→nrows() ≠ nr, "Wrong number of rows in TensorPolynomial::insert");
  TL_RAISE_IF(t→nvar() ≠ nv, "Wrong number of variables in TensorPolynomial::insert");
  TensorContainer⟨_Ttype⟩::insert(t);
  if (maxdim < t→dimen()) maxdim = t→dimen();
}
```

This code is used in section 497.

505. The polynomial takes the form

$$\sum_{i=0}^{n} \frac{1}{i!} \left[g_{y^i}\right]_{\alpha_1\ldots\alpha_i} [y]^{\alpha_1} \ldots [y]^{\alpha_i},$$

where $\left[g_{y^i}\right]$ are i-order derivatives of the polynomial. We assume that $\frac{1}{i!}\left[g_{y^i}\right]$ are items in the tensor container. This method differentiates the polynomial by one order to yield:

$$\sum_{i=1}^{n} \frac{1}{i!} \left[i \cdot g_{y^i}\right]_{\alpha_1\ldots\alpha_i} [y]^{\alpha_1} \ldots [y]^{\alpha_{i-1}},$$

where $\left[i \cdot \frac{1}{i!} \cdot g_{y^i}\right]$ are put to the container.

A polynomial can be derivative of some order, and the order cannot be recognized from the object. That is why we need to input the order.

⟨ **TensorPolynomial** :: *derivative* code 505 ⟩ ≡

```
void derivative(int k)
{
  for (int d = 1; d ≤ maxdim; d++) {
    if (_Tparent::check(Symmetry(d))) {
      _Ttype *ten = _Tparent::get(Symmetry(d));
      ten→mult((double) max((d − k), 0));
    }
  }
}
```

This code is cited in section 497.

This code is used in section 497.

506. Now let us suppose that we have an s order derivative of a polynomial whose i order derivatives are $\left[g_{y^i}\right]$, so we have

$$\sum_{i=s}^{n}\frac{1}{i!}\left[g_{y^i}\right]_{\alpha_1\ldots\alpha_i}\prod_{k=1}^{i-s}\left[y\right]^{\alpha_k},$$

where $\frac{1}{i!}\left[g_{y^i}\right]$ are tensors in the container.

This methods performs this evaluation. The result is an s dimensional tensor. Note that when combined with the method *derivative*, they evaluate a derivative of some order. For example a sequence of calls $g.derivative(0)$, $g.derivative(1)$ and $der = g.evalPartially(2, v)$ calculates 2! multiple of the second derivative of g at v.

⟨ **TensorPolynomial** :: *evalPartially* code 506 ⟩ ≡

```
_Ttype *evalPartially(int s, const ConstVector &v)
{
  TL_RAISE_IF(v.length() ≠ nvars(),
      "Wrong␣length␣of␣vector␣for␣TensorPolynomial::evalPartially");
  _Ttype *res = new _Ttype(nrows(), nvars(), s);
  res→zeros();
  if (_Tparent::check(Symmetry(s))) res→add(1.0, *(_Tparent::get(Symmetry(s))));
  for (int d = s + 1; d ≤ maxdim; d++) {
    if (_Tparent::check(Symmetry(d))) {
      const _Ttype &ltmp = *(_Tparent::get(Symmetry(d)));
      _Ttype *last = new _Ttype(ltmp);
      for (int j = 0; j < d − s; j++) {
        _Ttype *newlast = new _Ttype(*last, v);
        delete last;
        last = newlast;
      }
      res→add(1.0, *last);
      delete last;
    }
  }
  return res;
}
```

This code is cited in section 497.

This code is used in section 497.

507. This just gives a name to unfolded tensor polynomial.

⟨ **UTensorPolynomial** class declaration 507 ⟩ ≡

```
class FTensorPolynomial;
class UTensorPolynomial : public TensorPolynomial⟨UFSTensor, UGSTensor,
        URSingleTensor⟩ {
public:
    UTensorPolynomial(int rows, int vars)
    : TensorPolynomial⟨UFSTensor, UGSTensor, URSingleTensor⟩(rows, vars) { }
    UTensorPolynomial(const UTensorPolynomial &up, int k)
    : TensorPolynomial⟨UFSTensor, UGSTensor, URSingleTensor⟩(up, k) { }
    UTensorPolynomial(const FTensorPolynomial &fp);
    UTensorPolynomial(const UTensorPolynomial &tp, const Vector &xval)
    : TensorPolynomial⟨UFSTensor, UGSTensor, URSingleTensor⟩(tp, xval) { }
    UTensorPolynomial(int first_row, int num, UTensorPolynomial &tp)
    : TensorPolynomial⟨UFSTensor, UGSTensor, URSingleTensor⟩(first_row, num, tp) { }
};
```

This code is cited in section 1.
This code is used in section 495.

508. This just gives a name to folded tensor polynomial.

⟨ **FTensorPolynomial** class declaration 508 ⟩ ≡

```
class FTensorPolynomial : public TensorPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩
        {
public:
    FTensorPolynomial(int rows, int vars)
    : TensorPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩(rows, vars) { }
    FTensorPolynomial(const FTensorPolynomial &fp, int k)
    : TensorPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩(fp, k) { }
    FTensorPolynomial(const UTensorPolynomial &up);
    FTensorPolynomial(const FTensorPolynomial &tp, const Vector &xval)
    : TensorPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩(tp, xval) { }
    FTensorPolynomial(int first_row, int num, FTensorPolynomial &tp)
    : TensorPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩(first_row, num, tp) { }
};
```

This code is cited in section 1.
This code is used in section 495.

509. The compact form of **TensorPolynomial** is in fact a full symmetry tensor, with the number of variables equal to the number of variables of the polynomial plus 1 for 1.

⟨ **CompactPolynomial** class declaration 509 ⟩ ≡

```
template⟨class _Ttype, class _TGStype, class _Stype⟩
class CompactPolynomial : public _Ttype {
public:
    ⟨ CompactPolynomial constructor code 510 ⟩;
    ⟨ CompactPolynomial::eval method code 511 ⟩;
};
```

This code is used in section 495.

510. This constructor copies matrices from the given tensor polynomial to the appropriate location in this matrix. It creates a dummy tensor *dum* with two variables (one corresponds to 1, the other to x). The index goes through this dummy tensor and the number of columns of the folded/unfolded general symmetry tensor corresponding to the selections of 1 or x given by the index. Length of 1 is one, and length of x is *pol.nvars*( ). This nvs information is stored in *dumnvs*. The symmetry of this general symmetry dummy tensor *dumgs* is given by a number of ones and x's in the index. We then copy the matrix, if it exists in the polynomial and increase *offset* for the following cycle.

⟨ **CompactPolynomial** constructor code 510 ⟩ ≡

```
CompactPolynomial(const TensorPolynomial<_Ttype, _TGStype, _Stype> &pol)
: _Ttype(pol.nrows(), pol.nvars() + 1, pol.getMaxDim()) {
  _Ttype::zeros();
  IntSequence dumnvs(2);
  dumnvs[0] = 1;
  dumnvs[1] = pol.nvars();
  int offset = 0;
  _Ttype dum(0, 2, _Ttype::dimen());
  for (Tensor::index i = dum.begin(); i != dum.end(); ++i) {
    int d = i.getCoor().sum();
    Symmetry symrun(_Ttype::dimen() - d, d);
    _TGStype dumgs(0, TensorDimens(symrun, dumnvs));
    if (pol.check(Symmetry(d))) {
      TwoDMatrix subt(*this, offset, dumgs.ncols());
      subt.add(1.0, *(pol.get(Symmetry(d))));
    }
    offset += dumgs.ncols();
  }
}
```

This code is used in section 509.

511. We create *x1* to be a concatenation of 1 and x, and then create **PowerProvider** to make a corresponding power *xpow* of *x1*, and finally multiply this matrix with the power.

```
⟨ CompactPolynomial::eval method code 511 ⟩ ≡
  void eval(Vector &out, const ConstVector &v) const
  {
    TL_RAISE_IF(v.length() + 1 ≠ _Ttype::nvar(),
        "Wrong input vector length in CompactPolynomial::eval");
    TL_RAISE_IF(out.length() ≠ _Ttype::nrows(),
        "Wrong output vector length in CompactPolynomial::eval");
    Vector x1(v.length() + 1);
    Vector x1p(x1, 1, v.length());
    x1p = v;
    x1[0] = 1.0;
    if (_Ttype::dimen() ≡ 0) out = ConstVector(*this, 0);
    else {
      PowerProvider pp(x1);
      const _Stype &xpow = pp.getNext((const _Stype *) Λ);
      for (int i = 1; i < _Ttype::dimen(); i++) xpow = pp.getNext((const _Stype *) Λ);
      multVec(0.0, out, 1.0, xpow);
    }
  }
```

This code is used in section 509.

512. Specialization of the **CompactPolynomial** for unfolded tensor.

```
⟨ UCompactPolynomial class declaration 512 ⟩ ≡
  class UCompactPolynomial : public CompactPolynomial⟨UFSTensor, UGSTensor,
        URSingleTensor⟩ {
  public:
    UCompactPolynomial(const UTensorPolynomial &upol)
    : CompactPolynomial⟨UFSTensor, UGSTensor, URSingleTensor⟩(upol) { }
  };
```

This code is used in section 495.

513. Specialization of the **CompactPolynomial** for folded tensor.

```
⟨ FCompactPolynomial class declaration 513 ⟩ ≡
  class FCompactPolynomial : public CompactPolynomial⟨FFSTensor, FGSTensor,
        FRSingleTensor⟩ {
  public:
    FCompactPolynomial(const FTensorPolynomial &fpol)
    : CompactPolynomial⟨FFSTensor, FGSTensor, FRSingleTensor⟩(fpol) { }
  };
```

This code is used in section 495.

514. End of `t_polynomial.h` file.

515. Start of `t_polynomial.cpp` file.

```
#include "t_polynomial.h"
#include "kron_prod.h"
  ⟨ PowerProvider::getNext unfolded code 516 ⟩;
  ⟨ PowerProvider::getNext folded code 517 ⟩;
  ⟨ PowerProvider destructor code 518 ⟩;
  ⟨ UTensorPolynomial constructor conversion code 519 ⟩;
  ⟨ FTensorPolynomial constructor conversion code 520 ⟩;
```

516. This method constructs unfolded *ut* of higher dimension, deleting the previous.

```
⟨ PowerProvider::getNext unfolded code 516 ⟩ ≡
  const URSingleTensor &PowerProvider::getNext(const URSingleTensor *dummy)
  {
    if (ut) {
      URSingleTensor *ut_new = new URSingleTensor(nv, ut→dimen() + 1);
      KronProd::kronMult(ConstVector(origv), ConstVector(ut→getData()), ut_new→getData());
      delete ut;
      ut = ut_new;
    }
    else {
      ut = new URSingleTensor(nv, 1);
      ut→getData() = origv;
    }
    return *ut;
  }
```

This code is used in section 515.

517. This method just constructs next unfolded *ut* and creates folded *ft*.

```
⟨ PowerProvider::getNext folded code 517 ⟩ ≡
  const FRSingleTensor &PowerProvider::getNext(const FRSingleTensor *dummy)
  {
    getNext(ut);
    if (ft) delete ft;
    ft = new FRSingleTensor(*ut);
    return *ft;
  }
```

This code is used in section 515.

518.

```
⟨ PowerProvider destructor code 518 ⟩ ≡
  PowerProvider::~PowerProvider()
  {
    if (ut) delete ut;
    if (ft) delete ft;
  }
```

This code is used in section 515.

519. Clear.

⟨ **UTensorPolynomial** constructor conversion code 519 ⟩ ≡

```
UTensorPolynomial::UTensorPolynomial(const FTensorPolynomial &fp)
  : TensorPolynomial<UFSTensor, UGSTensor, URSingleTensor>(fp.nrows(), fp.nvars()) {
    for (FTensorPolynomial::const_iterator it = fp.begin(); it ≠ fp.end(); ++it) {
      insert(new UFSTensor(*((*it).second)));
    }
}
```

This code is used in section 515.

520. Clear.

⟨ **FTensorPolynomial** constructor conversion code 520 ⟩ ≡

```
FTensorPolynomial::FTensorPolynomial(const UTensorPolynomial &up)
  : TensorPolynomial<FFSTensor, FGSTensor, FRSingleTensor>(up.nrows(), up.nvars()) {
    for (UTensorPolynomial::const_iterator it = up.begin(); it ≠ up.end(); ++it) {
      insert(new FFSTensor(*((*it).second)));
    }
}
```

This code is used in section 515.

521. End of `t_polynomial.cpp` file.

522. Moments of normal distribution. Start of `normal_moments.h` file.

Here we calculate the higher order moments of normally distributed random vector u with means equal to zero and given variance–covariance matrix V, this is $u \sim N(0, V)$. The moment generating function for such distribution is $f(t) = e^{\frac{1}{2}t^T V t}$. If we derivate it wrt t and unfold the higher dimensional tensors row-wise, we obtain terms like

$$
\begin{aligned}
\frac{\partial}{\partial t} f(t) =& f(t) \cdot Vt \\
\frac{\partial^2}{\partial t^2} f(t) =& f(t) \cdot (Vt \otimes Vt + v) \\
\frac{\partial^3}{\partial t^3} f(t) =& f(t) \cdot (Vt \otimes Vt \otimes Vt + P_?(v \otimes Vt) + P_?(Vt \otimes v) + v \otimes Vt) \\
\frac{\partial^4}{\partial t^4} f(t) =& f(t) \cdot (Vt \otimes Vt \otimes Vt \otimes Vt + S_?(v \otimes Vt \otimes Vt) + S_?(Vt \otimes v \otimes Vt) + S_?(Vt \otimes Vt \otimes v) + S_?(v \otimes v))
\end{aligned}
$$

where v is vectorized V ($v = \mathrm{vec}(V)$), and $P_?$ is a suitable row permutation (corresponds to permutation of multidimensional indices) which permutes the tensor data, so that the index of a variable being derived would be the last. This ensures that all (permuted) tensors can be summed yielding a tensor whose indices have some order (in here we chose the order that more recent derivating variables are to the right). Finally, $S_?$ is a suitable sum of various $P_?$.

We are interested in $S_?$ multiplying the Kronecker powers $\otimes^n v$. The $S_?$ is a (possibly) multi-set of permutations of even order. Note that we know a number of permutations in $S_?$. The above formulas for $F(t)$ derivatives are valid also for monomial u, and from literature we know that $2n$-th moment is $\frac{(2n!)}{n!2^n}\sigma^2$. So there are $\frac{(2n!)}{n!2^n}$ permutations in $S_?$.

In order to find the $S_?$ we need to define a couple of things. First we define a sort of equivalence between the permutations applicable to even number of indices. We write $P_1 \equiv P_2$ whenever $P_1^{-1} \circ P_2$ permutes only whole pairs, or items within pairs, but not indices across the pairs. For instance the permutations $(0, 1, 2, 3)$ and $(3, 2, 0, 1)$ are equivalent, but $(0, 2, 1, 3)$ is not equivalent with the two. Clearly, the $\equiv$ is an equivalence.

This allows to define a relation $\sqsubseteq$ between the permutation multi-sets S, which is basically the subset relation $\subseteq$ but with respect to the equivalence $\equiv$, more formally:

$$S_1 \sqsubseteq S_2 \quad \text{iff} \quad P \in S_1 \Rightarrow \exists Q \in S_2 : P \equiv Q$$

This induces an equivalence $S_1 \equiv S_2$.

Now let F_n denote a set of permutations on $2n$ indices which is maximal with respect to $\sqsubseteq$, and minimal with respect to $\equiv$. (In other words, it contains everything up to the equivalence $\equiv$.) It is straightforward to calculate a number of permutations in F_n. This is a total number of all permutations of $2n$ divided by permutations of pairs divided by permutations within the pairs. This is $\frac{(2n!)}{n!2^n}$.

We prove that $S_? \equiv F_n$. Clearly $S_? \sqsubseteq F_n$, since F_n is maximal. In order to prove that $F_n \sqsubseteq S_?$, let us assert that for any permutation P and for any (semi)positive definite matrix V we have $PS_? \otimes^n v = S_? \otimes^n v$. Below we show that there is a positive definite matrix V of some dimension that for any two permutation multi-sets S_1, S_2, we have

$$S_1 \not\equiv S_2 \Rightarrow S_1(\otimes^n v) \neq S_2(\otimes^n v)$$

So it follows that for any permutation P, we have $PS_? \equiv S_?$. For a purpose of contradiction let $P \in F_n$ be a permutation which is not equivalent to any permutation from $S_?$. Since $S_?$ is non-empty, let us pick $P_0 \in S_?$. Now assert that $P_0^{-1} S_? \not\equiv P^{-1} S_?$ since the first contains an identity and the second does not contain a permutation equivalent to identity. Thus we have $(P \circ P_0^{-1}) S_? \not\equiv S_?$ which gives the contradiction and we have proved that $F_n \sqsubseteq S_?$. Thus $F_n \equiv S_?$. Moreover, we know that $S_?$ and F_n have the same number of permutations, hence the minimality of $S_?$ with respect to $\equiv$.

Now it suffices to prove that there exists a positive definite V such that for any two permutation multi-sets S_1, and S_2 holds $S_1 \not\equiv S_2 \Rightarrow S_1(\otimes^n v) \neq S_2(\otimes^n v)$. If V is $n \times n$ matrix, then $S_1 \not\equiv S_2$ implies that there is identically nonzero polynomial of elements from V of order n over integers. If $V = A^T A$ then there is identically non-zero polynomial of elements from A of order $2n$. This means, that we have to find $n(n+1)/2$ tuple x of real numbers such that all identically non-zero polynomials p of order $2n$ over integers yield $p(x) \neq 0$.

The x is constructed as follows: $x_i = \pi^{\log r_i}$, where r_i is i-th prime. Let us consider monom $x_1^{j_1} \cdot \ldots \cdot x_k^{j_k}$. When the monom is evaluated, we get

$$\pi^{\log r_1^{j_1} + \ldots + \log r_k^{j_k}} = \pi^{\log \left(r_1^{j_1} \cdot \ldots \cdot r_k^{j_k}\right)}$$

Now it is easy to see that if an integer combination of such terms is zero, then the combination must be either trivial or sum to 0 and all monoms must be equal. Both cases imply a polynomial identically equal to zero. So, any non-trivial integer polynomial evaluated at x must be non-zero.

So, having this result in hand, now it is straightforward to calculate higher moments of normal distribution. Here we define a container, which does the job. In its constructor, we simply calculate Kronecker powers of v and apply F_n to $\otimes^n v$. F_n is, in fact, a set of all equivalences in sense of class **Equivalence** over $2n$ elements, having n classes each of them having exactly 2 elements.

```
#ifndef NORMAL_MOMENTS_H
#define NORMAL_MOMENTS_H
#include "t_container.h"
  ⟨ UNormalMoments class declaration 523 ⟩;
  ⟨ FNormalMoments class declaration 524 ⟩;
#endif
```

523.

```
⟨ UNormalMoments class declaration 523 ⟩ ≡
  class UNormalMoments : public TensorContainer⟨URSingleTensor⟩ {
  public:
    UNormalMoments(int maxdim, const TwoDMatrix &v);
  private:
    void generateMoments(int maxdim, const TwoDMatrix &v);
    static bool selectEquiv(const Equivalence &e);
  };
```

This code is cited in section 1.

This code is used in section 522.

524.

```
⟨ FNormalMoments class declaration 524 ⟩ ≡
  class FNormalMoments : public TensorContainer⟨FRSingleTensor⟩ {
  public:
    FNormalMoments(const UNormalMoments &moms);
  };
```

This code is cited in section 1.

This code is used in section 522.

525. End of `normal_moments.h` file.

526. Start of `normal_moments.cpp` file.

```
#include "normal_moments.h"
#include "permutation.h"
#include "kron_prod.h"
#include "tl_static.h"
  ⟨ UNormalMoments constructor code 527 ⟩;
  ⟨ UNormalMoments::generateMoments code 528 ⟩;
  ⟨ UNormalMoments::selectEquiv code 530 ⟩;
  ⟨ FNormalMoments constructor code 531 ⟩;
```

527.

```
⟨ UNormalMoments constructor code 527 ⟩ ≡
  UNormalMoments::UNormalMoments(int maxdim, const TwoDMatrix &v)
  : TensorContainer⟨URSingleTensor⟩(1) {
    if (maxdim ≥ 2) generateMoments(maxdim, v);
  }
```

This code is used in section 526.

528. Here we fill up the container with the tensors for $d = 2, 4, 6, \ldots$ up to the given dimension. Each tensor of moments is equal to $F_n\left(\otimes^n v\right)$. This has a dimension equal to $2n$. See the header file for proof and details.

Here we sequentially construct the Kronecker power $\otimes^n v$, and apply F_n.

```
⟨ UNormalMoments::generateMoments code 528 ⟩ ≡
  void UNormalMoments::generateMoments(int maxdim, const TwoDMatrix &v)
  {
    TL_RAISE_IF(v.nrows() ≠ v.ncols(),
        "Variance-covariance␣matrix␣is␣not␣square␣in␣UNormalMoments␣constructor");
    int nv = v.nrows();
    URSingleTensor *mom2 = new URSingleTensor(nv, 2);
    mom2→getData() = v.getData();
    insert(mom2);
    URSingleTensor *kronv = new URSingleTensor(nv, 2);
    kronv→getData() = v.getData();
    for (int d = 4; d ≤ maxdim; d += 2) {
      URSingleTensor *newkronv = new URSingleTensor(nv, d);
      KronProd::kronMult(ConstVector(v.getData()), ConstVector(kronv→getData()),
          newkronv→getData());
      delete kronv;
      kronv = newkronv;
      URSingleTensor *mom = new URSingleTensor(nv, d);
      ⟨ apply F_n to kronv 529 ⟩;
      insert(mom);
    }
    delete kronv;
  }
```

This code is used in section 526.

529. Here we go through all equivalences, select only those having 2 elements in each class, then go through all elements in *kronv* and add to permuted location of *mom*.

The permutation must be taken as inverse of the permutation implied by the equivalence, since we need a permutation which after application to identity of indices yileds indices in the equivalence classes. Note how the **Equivalence** :: *apply* method works.

⟨ apply F_n to *kronv* 529 ⟩ ≡

```
mom→zeros();
const EquivalenceSet eset = ebundle.get(d);
for (EquivalenceSet::const_iterator cit = eset.begin(); cit ≠ eset.end(); cit++) {
  if (selectEquiv(*cit)) {
    Permutation per(*cit);
    per.inverse();
    for (Tensor::index it = kronv→begin(); it ≠ kronv→end(); ++it) {
      IntSequence ind(kronv→dimen());
      per.apply(it.getCoor(), ind);
      Tensor::index it2(mom, ind);
      mom→get(*it2, 0) += kronv→get(*it, 0);
    }
  }
}
```

This code is used in section 528.

530. We return *true* for an equivalence whose each class has 2 elements.

⟨ **UNormalMoments** :: *selectEquiv* code 530 ⟩ ≡

```
bool UNormalMoments::selectEquiv(const Equivalence &e)
{
  if (2 * e.numClasses() ≠ e.getN()) return false;
  for (Equivalence::const_seqit si = e.begin(); si ≠ e.end(); ++si) {
    if ((*si).length() ≠ 2) return false;
  }
  return true;
}
```

This code is used in section 526.

531. Here we go through all the unfolded container, fold each tensor and insert it.

⟨ **FNormalMoments** constructor code 531 ⟩ ≡

```
FNormalMoments::FNormalMoments(const UNormalMoments &moms)
: TensorContainer⟨FRSingleTensor⟩(1) {
  for (UNormalMoments::const_iterator it = moms.begin(); it ≠ moms.end(); ++it) {
    FRSingleTensor *fm = new FRSingleTensor(*((*it).second));
    insert(fm);
  }
}
```

This code is used in section 526.

532. End of `normal_moments.cpp` file.

533. Tensor library static data. Start of `tl_static.h` file.

The purpose of this file is to make a unique static variable which would contain all other static variables and be responsible for their correct initialization and destruction. The variables include an equivalence bundle and a Pascal triangle for binomial coefficients. Both depend on dimension of the problem, and maximum number of variables.

So we declare static *tls* variable of type **TLStatic** encapsulating the variables. The *tls* must be initialized at the beginning of the program, as dimension and number of variables is known.

Also we define a class for Pascal triangle.

```
#ifndef TL_STATIC_H
#define TL_STATIC_H
#include "equivalence.h"
#include "permutation.h"
  ⟨ PascalTriangle class declaration 534 ⟩;
  ⟨ TLStatic class declaration 535 ⟩;
  extern TLStatic tls;
#endif
```

534. Pascal triangle is a storage for binomial coefficients. We store in *data* array the coefficients of rectangle starting at $\binom{0}{0}$, and ending $\binom{nmax+kmax}{kmax}$.

```
⟨ PascalTriangle class declaration 534 ⟩ ≡
  class PascalTriangle {
    int *data;
    int kmax;
    int nmax;
  public:
    PascalTriangle(int n, int k);
    ~PascalTriangle()
    { delete[] data; }
    int noverk(int n, int k) const;
  };
```

This code is used in section 533.

535.

```
⟨ TLStatic class declaration 535 ⟩ ≡
  struct TLStatic {
    EquivalenceBundle *ebundle;
    PermutationBundle *pbundle;
    PascalTriangle *ptriang;
    TLStatic();
    ~TLStatic();
    void init(int dim, int nvar);
  };
```

This code is cited in section 1.

This code is used in section 533.

536. End of `tl_static.h` file.

537. Start of `tl_static.cpp` file.

```
#include "tl_static.h"
#include "tl_exception.h"
  TLStatic tls;
  ⟨ TLStatic methods 538 ⟩;
  ⟨ PascalTriangle constructor code 539 ⟩;
  ⟨ PascalTriangle :: noverk code 540 ⟩;
```

538. Note that we allow for repeated calls of *init*. This is not normal and the only purpose of allowing this is the test suite.

```
⟨ TLStatic methods 538 ⟩ ≡
  TLStatic::TLStatic()
  {
    ebundle = Λ;
    pbundle = Λ;
    ptriang = Λ;
  }
  TLStatic::~TLStatic()
  {
    if (ebundle) delete ebundle;
    if (pbundle) delete pbundle;
    if (ptriang) delete ptriang;
  }
  void TLStatic::init(int dim, int nvar)
  {
    if (ebundle) ebundle→generateUpTo(dim);
    else ebundle = new EquivalenceBundle(dim);
    if (pbundle) pbundle→generateUpTo(dim);
    else pbundle = new PermutationBundle(dim);
    if (ptriang) delete ptriang;
    ptriang = new PascalTriangle(nvar, dim);
  }
```

This code is used in section 537.

539. The coefficients are stored in *data* row by row where a row are coeffs with the same k.

We first initialize the first row with ones. Then for each other row we initialize the first item to one, and other items are a sum of coefficients of $n-1$ which is in code $i+j-1$.

```
⟨ PascalTriangle constructor code 539 ⟩ ≡
  PascalTriangle::PascalTriangle(int n, int k)
  : data(new int[(n + 1) * (k + 1)]), kmax(k), nmax(n) {
    for (int i = 0; i ≤ n; i++) data[i] = 1;
    for (int j = 1; j ≤ k; j++) {
      data[j * (nmax + 1)] = 1;
      for (int i = 1; i ≤ n; i++) data[j * (nmax + 1) + i] = noverk(i + j − 1, j) + noverk(i + j − 1, j − 1);
    }
  }
```

This code is used in section 537.

540. Clear. Recall, that there are $nmax + 1$ items in a row.

⟨ **PascalTriangle**::*noverk* code 540 ⟩ ≡

```
int PascalTriangle::noverk(int n, int k) const
{
  TL_RAISE_IF(k > n ∨ n < 0, "Wrong␣arguments␣for␣PascalTriangle::noverk");
  if (k ≤ kmax ∧ n − k ≤ nmax) return data[k ∗ (nmax + 1) + n − k];
  if (n − k ≤ kmax ∧ k ≤ nmax) return data[(n − k) ∗ (nmax + 1) + k];
  TL_RAISE("n␣or␣k␣out␣of␣range␣in␣PascalTriangle::noverk");
  return 0;
}
```

This code is used in section 537.

541. End of `tl_static.cpp` file.

542. Index.

Tensor Library