0.1 Time Step Targetting with Krylov Method

This class implements the time-step algorithm as described by Manmana et al. in `http://de.arxiv.org/abs/cond-mat/0606018v1`, in particular Eqs. (55) to (60) on page 26.

"`TimeStepTargetting.h`" 1≡

```
#ifndef TIMESTEPTARGETTING_H
#define TIMESTEPTARGETTING_H
#include <iostream>
#include "ProgressIndicator.h"
#include "BLAS.h"
#include "ApplyOperatorLocal.h"
#include "TimeSerializer.h"
#include "TimeStepParams.h"
```

This class is templated on 7 templates, which are:

1. `LanczosSolverTemplate`, being usually the `LanczosSolver` class.
2. `InternalProductTemplate`, being usually the `InternalProductOnTheFly` class. This is a very short class that allows to compute the superblock matrix either on-the-fly or to store it. (by using `InternalProductStored`). The latter option is limited to small systems due to memory constraints.
3. `WaveFunctionTransformationType` is usually the `WaveFunctionTransformation` class. The wave function transformation is too long to explain here but it is a standard computational trick in DMRG, and was introduce in 1996 by S. White (need to write here the corresponding PRB article FIXME).
4. `ModelType` is the model in question. These are classes under the directory Models.
5. `ConcurrenyType` is the type to deal with parallelization or lack thereof.
6. `IoType` is usually the `IoSimple` class, and deals with writing to disk the time-vectors produced by this class.
7. `VectorWithOffsetTemplate` is usually the `VectorWithOffsets` class that encapsulates the functionality of a vector that is mostly zero except for chunks of non-zero numbers at certain offsets. Note that there is (for efficiency reasons) a `VectorWithOffset` class that encapsulates the functionality of a vector with a single chunk. That class is used in `GroundStateTargetting` but not here. Why do vectors in chunks appear here (you might be wondering)? Well, because of symmetries the vectors are zero mostly everywhere except on the (targetted) symmetry sector(s).

"TimeStepTargetting.h" 2a≡

```
namespace Dmrg {
 template<
   template<typename,typename,typename> class LanczosSolverTemplate,
      template<typename,typename> class InternalProductTemplate,
    typename WaveFunctionTransformationType_,
       typename ModelType_,
    typename ConcurrencyType_,
       typename IoType_,
          template<typename> class VectorWithOffsetTemplate>
 class TimeStepTargetting  {
```

A long series of typedefs follow. Need to explain these maybe (FIXME).

"TimeStepTargetting.h" 2b≡

```
   public:
    typedef WaveFunctionTransformationType_ WaveFunctionTransformationType;
    typedef ModelType_ ModelType;
    typedef ConcurrencyType_ ConcurrencyType;
    typedef IoType_ IoType;
    typedef typename ModelType::RealType RealType;
    typedef std::complex<RealType> ComplexType;
    typedef InternalProductTemplate<ComplexType,ModelType> InternalProductType;
    typedef typename ModelType::OperatorsType OperatorsType;
    typedef typename ModelType::MyBasisWithOperators BasisWithOperatorsType;
    //typedef BasisWithOperators<OperatorsType,ConcurrencyType> BasisWithOperatorsType;
    typedef std::vector<ComplexType> ComplexVectorType;
    //typedef std::VectorWithOffset<ComplexType> VectorWithOffsetType;
    typedef LanczosSolverTemplate<RealType,InternalProductType,ComplexVectorType> LanczosSolverType;
    typedef std::vector<RealType> VectorType;
    //typedef typename BasisWithOperatorsType::SparseMatrixType SparseMatrixType;
    typedef psimag::Matrix<ComplexType> ComplexMatrixType;
    typedef typename LanczosSolverType::TridiagonalMatrixType TridiagonalMatrixType;
    typedef typename BasisWithOperatorsType::OperatorType OperatorType;
    typedef typename BasisWithOperatorsType::BasisType BasisType;
    typedef TimeStepParams<ModelType> TargettingParamsType;
    typedef typename BasisType::BlockType BlockType;
    typedef VectorWithOffsetTemplate<ComplexType> VectorWithOffsetType;
    typedef ComplexVectorType TargetVectorType;
    typedef BlockMatrix<ComplexType,ComplexMatrixType> ComplexBlockMatrixType;
    typedef ApplyOperatorLocal<BasisWithOperatorsType,VectorWithOffsetType,TargetVectorType> ApplyOperatorType;
    typedef TimeSerializer<RealType,VectorWithOffsetType> TimeSerializerType;
```

And now a few enums and other constants. The first refers to the 4 steps in which the time-step algorithm can be.

1. DISABLED Time-step-targetting is disabled if we are not computing any time-dependent operator yet (like when we're in the infinite algorithm)or if the user specified TSTLoops with numbers greater than zero, those numbers indicate the loops that must pass before time-dependent observables are computed.

2. OPERATOR In this stage we're applying an operator

3. WFT_NOADVANCE In this stage we're adavancing in space with the wave function transformation (WFT) but not advancing in time.

4. WFT_NOADVANCE In this stage we're adavancing in space and time

"TimeStepTargetting.h" 3a≡

```
enum {DISABLED,OPERATOR,WFT_NOADVANCE,WFT_ADVANCE};
enum {EXPAND_ENVIRON=WaveFunctionTransformationType::EXPAND_ENVIRON,
EXPAND_SYSTEM=WaveFunctionTransformationType::EXPAND_SYSTEM,
INFINITE=WaveFunctionTransformationType::INFINITE};


static const size_t parallelRank_ = 0; // TST needs to support concurrency FIXME
```

Now comes the constructor which takes 6 arguments. The first 3 arguments are the system (left-block), environment (right-block), and superblock (system + environment). As usual, the first 2 are heavy objects—with operators—, and the superblock is light. The 4th argument is the model object. The 5th argument is a `TargettingStructureType` object which is a `TargettingStructureParms` object (as you can see in). A structure is just a bunch of data bundled together, and you can see this in the file `TargetStructureParams.h`. The last argument is a `WaveFunctionTransformation` object. More info about this class is in `WaveFunctionTransformation.h`.

"TimeStepTargetting.h" 3b≡

```
TimeStepTargetting(
    const BasisWithOperatorsType& basisS,
        const BasisWithOperatorsType& basisE,
      const BasisType& basisSE,
   const ModelType& model,
  const TargettingParamsType& tstStruct,
  const WaveFunctionTransformationType& wft)
```

Now let us look at the private data of this class:

⟨ *privatedata* 3c ⟩ ≡

```
std::vector<size_t> stage_;
VectorWithOffsetType psi_;
const BasisWithOperatorsType& basisS_;
const BasisWithOperatorsType& basisE_;
const BasisType& basisSE_;
const ModelType& model_;
const TargettingParamsType& tstStruct_;
const WaveFunctionTransformationType& waveFunctionTransformation_;
ProgressIndicator progress_;
RealType currentTime_;
std::vector<RealType> times_,weight_;
std::vector<VectorWithOffsetType> targetVectors_;
RealType gsWeight_;
typename IoType::Out io_;
ApplyOperatorType applyOpLocal_;
```

Now we get to the stack initialization of this object (note the colon). We said before that the algorithm could be in 4 stages. In reality, there is not a stage for the full algorithm but there's a stage for each operator to be applied (like holon and then doublon).These operators are specified by the user in the input file in TSPSites. All stages are set to DISABLED at the beginning. We make reference copies to the bases for system (`basisS`), environment (`basisE`), and superblock (`basisSE`). We also make a reference

copy for the model and the tst t(ime)s(tep)t(argetting)Struct(ure), and of the `waveFunctionTransformation` object. We initialize the `progress` object that helps with printing progress to the terminal. The currentTime is initially set to 0. The number of time steps to perform for the Runge-Kutta procedure is taken from `tstStruct_.timeSteps`, as provided by the user. The `weight`, which is a vector of weights, for each target state (except possibly the ground state) is set to the number of time steps. Next an `io` or input/output object is constructed. This is needed to dump the time-vectors to disk since we don't do computations *in-situ* here.The `applyLocal` operator described before is also initalized on the stack.

"TimeStepTargetting.h" 4a≡

```
: stage_(tstStruct.sites.size(),DISABLED),basisS_(basisS),basisE_(basisE),basisSE_(basisSE),
 model_(model),tstStruct_(tstStruct),waveFunctionTransformation_(wft),
 progress_("TimeStepTargetting",0),currentTime_(0),
 times_(tstStruct_.timeSteps),weight_(tstStruct_.timeSteps),
 targetVectors_(tstStruct_.timeSteps),
 io_(tstStruct_.filename,parallelRank_),applyOpLocal_(basisS,basisE,basisSE)
```

The body of the constructor follows:

"TimeStepTargetting.h" 4b≡

```
{
 if (!wft.isEnabled()) throw std::runtime_error("␣TimeStepTargetting␣"
    "needs␣an␣enabled␣wft\n");

 RealType tau =tstStruct_.tau;
 RealType sum = 0;
 RealType factor = 1.0;
 size_t n = times_.size();
 for (size_t i=0;i<n;i++) {
  times_[i] = i*tau/(n-1);
  weight_[i] = factor/(n+2);
  sum += weight_[i];
 }
 sum -= weight_[0];
 sum -= weight_[n-1];
 weight_[0] = weight_[n-1] = 2*factor/(n+2);
 sum += weight_[n-1];
 sum += weight_[0];

 gsWeight_=1.0-sum;
 sum += gsWeight_;
 //for (size_t i=0;i<weight_.size();i++) sum += weight_[i];
 if (fabs(sum-1.0)>1e-5) throw std::runtime_error("Weights␣don't␣amount␣to␣one\n");
 printHeader();
}
```

The public member function `weight` returns the weight of target state i. This is important for the `DensityMatrix` class to be able weight the states properly. Note that it throws if you ask for weights of time-vectors when all stages are disabled, since this would be an error.

"TimeStepTargetting.h" 4c≡

```
RealType weight(size_t i) const
{
 if (allStages(DISABLED)) throw std::runtime_error("TST: What are you doing here?\n");
 return weight_[i];
 //return 1.0;
}
```

The public member function `gsWeight` returns the weight of the ground states. During the disabled stages it is 1 since there are no other vectors to target.

"TimeStepTargetting.h" 5a≡

```
RealType gsWeight() const
{
 if (allStages(DISABLED)) return 1.0;
 return gsWeight_;
 //return 1.0;
}
```

This member function returns the squared norm of time vector number i.

"TimeStepTargetting.h" 5b≡

```
RealType normSquared(size_t i) const
{
 // call to mult will conjugate one of the vector
 return real(multiply(targetVectors_[i],targetVectors_[i]));
}
```

Sets the ground state to whatever is passed in `v`. The basis to which this state belongs need be passed in `someBasis` because of the chunking of this vector.

"TimeStepTargetting.h" 5c≡

```
template<typename SomeBasisType>
void setGs(const std::vector<TargetVectorType>& v,
    const SomeBasisType& someBasis)
{
 psi_.set(v,someBasis);
}
```

Returns the $i-$th element of the ground state `psi`.

"TimeStepTargetting.h" 5d≡

```
const ComplexType& operator[](size_t i) const { return psi_[i]; }

ComplexType& operator[](size_t i) { return psi_[i]; }
```

Returns the full ground state vector as a vector with offset:

"TimeStepTargetting.h" 5e≡

```
const VectorWithOffsetType& gs() const { return psi_; }
```

Will the ground state be included in the density matrix? If using this class it will always be.

"TimeStepTargetting.h" 5f≡

```
bool includeGroundStage() const {return true; }
```

How many time vectors does the `DensityMatrix` need to include, excepting the ground state? Note that when all stages are disabled no time vectors are included.

"TimeStepTargetting.h" 6a≡

```
size_t size() const
{
 if (allStages(DISABLED)) return 0;
 return targetVectors_.size();
}
```

Returns the full time vector number i as a vector with offsets:

"TimeStepTargetting.h" 6b≡

```
const VectorWithOffsetType& operator()(size_t i) const
{
 return targetVectors_[i];
}
```

This function provides a hook to (possibly) start the computation of time-evolution. Five arguments are passed. First Eg, the ground state energy, then the `direction` of expansion (system or environment), then the `block` being currently grown or shrunk, then the `loopNumber` of the finite algorithm, and finally a flag `needsPrinting` that indicates if time-vectors need to be printed to disk for post-processing or not.

Here the main work is done two functions, a different function `evolve` is called to either WFT transform the vector or to apply the operators to the ground state. The next big function is `calcTimeVectors` that apply $exp(iHt)$ to the resulting vector for different times $t + x\Delta t$, where if the Runge-Kutta procedure is used then $x = 0, 1/3, 2/3$ and 1.

These two functions call other functions. We'll continue linearly describing each one in turnin order of appearance.

"TimeStepTargetting.h" 7a≡

```
void evolve(RealType Eg,size_t direction,const BlockType& block,
 size_t loopNumber, bool needsPrinting)
{
 size_t count =0;
 VectorWithOffsetType phiOld = psi_;
 VectorWithOffsetType phiNew;
 size_t max = tstStruct_.sites.size();

 if (noStageIs(DISABLED)) max = 1;

 // Loop over each operator that needs to be applied
 // in turn to the g.s.
 for (size_t i=0;i<max;i++) {
  count += evolve(i,phiNew,phiOld,Eg,direction,block,loopNumber,max-1);
  phiOld = phiNew;
 }

 if (count==0) {
  // always print to keep observer driver in sync
  if (needsPrinting) {
   zeroOutVectors();
   printVectors(block);
  }
  return;
 }

 calcTimeVectors(Eg,phiNew,direction);

 cocoon(direction,block); // in-situ

 if (needsPrinting) printVectors(block); // for post-processing
}
```

The below function is called from the `evolve` above and, if appropriate, applies operator i to `phiOld` storing the result in `phiNew`. In some cases it just advances, through the WFT, state `phiOld` into `phiNew`. Let's look at the algorithm in detail.

"TimeStepTargetting.h" 7b≡

```
size_t evolve(
  size_t i,
  VectorWithOffsetType& phiNew,
  VectorWithOffsetType& phiOld,
  RealType Eg,
  size_t direction,
  const BlockType& block,
  size_t loopNumber,
       size_t lastI)
{
 static size_t  timesWithoutAdvancement=0;
```

If we have not yet reached the finite loop that the user specified as a starting loop, or if we are in the infinite algorithm phase, then we do nothing:

"TimeStepTargetting.h" 7c≡

```
 if (tstStruct_.startingLoops[i]>loopNumber || direction==INFINITE) return 0;
```

Currently this class can only deal with a DMRG algorithm that uses single site blocks for growth and shrinkage:

"TimeStepTargetting.h" 8a≡

```
    if (block.size()!=1) throw
     std::runtime_error("TimeStepTargetting::evolve(...):"
       "␣blocks␣of␣size␣!=␣1␣are␣unsupported␣(sorry)\n");
    size_t site = block[0];
```

If the stage is disabled and this is not the site on which the user specified, through `tstStruct_.sites`, to apply the operator, then do nothing:

"TimeStepTargetting.h" 8b≡

```
    if (site != tstStruct_.sites[i] && stage_[i]==DISABLED) return 0;
```

If we are on the site specified by the user to apply the operator, and we were disabled before, change the stage to `OPERATOR`. Otherwise, do not apply the operator, just advance in space one site.using the WFT: We also check the order in which sites were specified by the user against the order in which sites appear through the DMRG sweeping. This is explained below under function `checkOrder`

"TimeStepTargetting.h" 8c≡

```
    if (site == tstStruct_.sites[i] && stage_[i]==DISABLED) stage_[i]=OPERATOR;
    else stage_[i]=WFT_NOADVANCE;
    if (stage_[i] == OPERATOR) checkOrder(i);
```

If we have visited this site more than the times the user specified via `tstStruct_.advanceEach`, then set the stage to advance in time. Time only advances after the last (`lastI`) operators has been applied. In this case reset `timesWithoutAdvancement`, otherwise increase it.

"TimeStepTargetting.h" 8d≡

```
    if (timesWithoutAdvancement >= tstStruct_.advanceEach) {
     stage_[i] = WFT_ADVANCE;
     if (i==lastI) {
      currentTime_ += tstStruct_.tau;
      timesWithoutAdvancement=0;
     }
    } else {
     if (i==lastI && stage_[i]==WFT_NOADVANCE)
      timesWithoutAdvancement++;
    }
```

We now print some progress. Up to now, we simply set the stage but we are yet to apply the operator to the state `phiOld`. We delegate that to function `computePhi` that will be explain below.

"TimeStepTargetting.h" 9a≡

```
    std::ostringstream msg2;
    msg2<<"Steps without advance: "<<timesWithoutAdvancement;
    if (timesWithoutAdvancement>0) progress_.printline(msg2,std::cout);

    std::ostringstream msg;
    msg<<"Evolving, stage="<<getStage(i)<<" site="<<site<<" loopNumber="<<loopNumber;
    msg<<" Eg="<<Eg;
    progress_.printline(msg,std::cout);

    // phi = A|psi>
    computePhi(i,phiNew,phiOld,direction);

    return 1;
  }
```

Let us look at `computephi`. If we're in the stage of applying operator i, then we call `applyLocal` (see function `operator()` in file `ApplyLocalOperator.h`) to apply this operator to state `phiOld` and store the result in `phiNew`.

⟨ *computephi1* 9b ⟩ ≡

```
  void computePhi(size_t i,VectorWithOffsetType& phiNew,
    VectorWithOffsetType& phiOld,size_t systemOrEnviron)
  {
    size_t indexAdvance = times_.size()-1;
    size_t indexNoAdvance = 0;
    if (stage_[i]==OPERATOR) {
      std::ostringstream msg;
      msg<<"I'm applying a local operator now";
      progress_.printline(msg,std::cout);
      FermionSign fs(basisS_,tstStruct_.electrons);
      applyOpLocal_(phiNew,phiOld,tstStruct_.aOperators[i],fs,systemOrEnviron);
      RealType norma = norm(phiNew);
      if (norma==0) throw std::runtime_error("Norm of phi is zero\n");
      //std::cerr<<"Norm of phi="<<norma<<" when i="<<i<<"\n";
```

Else we need to advance in space with the WFT. In principle, to do this we just call function `setInitialVector` in file `WaveFunctionTransformation.h` as you can see below. There is, however, a slight complication, in that the `WaveFunctionTransformation` class expects to know which sectors in the resulting vector (`phiNew`) will turn out to be non-zero. So, we need to either guess which sectors will be non-zero by calling `guessPhiSectors` as described below, or just populate all sectors with and then "collapse" the non-zero sectors for efficiency.

⟨ *computephi2* 10a ⟩ ≡

```
  } else if (stage_[i]== WFT_NOADVANCE || stage_[i]== WFT_ADVANCE) {
   size_t advance = indexNoAdvance;
   if (stage_[i] == WFT_ADVANCE) advance = indexAdvance;
   std::ostringstream msg;
   msg<<"I'm calling the WFT now";
   progress_.printline(msg,std::cout);

   if (tstStruct_.aOperators.size()==1) guessPhiSectors(phiNew,i,systemOrEnviron);
   else phiNew.populateSectors(basisSE_);

   // OK, now that we got the partition number right, let's wft:
   waveFunctionTransformation_.setInitialVector(phiNew,targetVectors_[advance],
     basisS_,basisE_,basisSE_); // generalize for su(2)
   phiNew.collapseSectors();

  } else {
   throw std::runtime_error("It's 5 am, do you know what line "
    " your code is exec-ing?\n");
  }
 }
```

This function provides an initial guess for the Lanczos vector. Traditionally, when DMRG is only targetting the ground state this is a standard procedure (see file `GroundStateTargetting`). Here, when `TimeStepTargetting`, we need be concerned with all target states and we the stages of the application of operators. When all stages are disabled then the initial guess is just delegated to one call of the WFT's`setInitialVector` function. When stages are advancing we need to weight each target WFtransformed state with the appropriate weights:

`"TimeStepTargetting.h"` 10b≡

```
 void initialGuess(VectorWithOffsetType& v) const
 {
  waveFunctionTransformation_.setInitialVector(v,psi_,basisS_,basisE_,basisSE_);
  bool b = allStages(WFT_ADVANCE) || allStages(WFT_NOADVANCE);
  if (!b) return;
  std::vector<VectorWithOffsetType> vv(targetVectors_.size());
  for (size_t i=0;i<targetVectors_.size();i++) {
   waveFunctionTransformation_.setInitialVector(vv[i],
    targetVectors_[i],basisS_,basisE_,basisSE_);
   if (norm(vv[i])<1e-6) continue;
   VectorWithOffsetType w= weight_[i]*vv[i];
   v += w;
  }
 }
```

Finally, the following 3 member public functions return the superblock object, the system (left-block) or the environment objects, or rather, the references held by this class.

`"TimeStepTargetting.h"` 10c≡

```
 const BasisType& basisSE() const { return basisSE_; }

 const BasisWithOperatorsType& basisS() const { return basisS_; }

 const BasisWithOperatorsType& basisE() const { return basisE_; }
```

What remains are private (i.e. non-exported) code used only by this class. We'll visit one function at a time.

First the `cocoon` function does measure the density of all time vectors *in situ*. This is done only for debugging purposes, and uses the function `test`.

"TimeStepTargetting.h" 11a≡

```
private:

 // in situ computation:
 void cocoon(size_t direction,const BlockType& block) const
 {
  size_t site = block[0];
  std::cerr<<"————————————&&&&␣Cocoon␣output␣starts\n";
  test(psi_,psi_,direction,"<PSI|A|PSI>",site);

  for (size_t j=0;j<targetVectors_.size();j++) {
   std::string s = "<P"+utils::ttos(j)+"|A|P"+utils::ttos(j)+">";
   test(targetVectors_[j],psi_,direction,s,site);
  }
  std::cerr<<"————————————&&&&␣Cocoon␣output␣ends\n";
 }
```

If we see $site[i]$ then we need to make sure we've seen all sites $site[j]$ for $j \leq i$.In other words, the order in which the user specifies the affected sites for the application of operators need to be the same as the order in which the DMRG sweeping process encounters those sites. Else we throw.

"TimeStepTargetting.h" 11b≡

```
 void checkOrder(size_t i) const
 {
  if (i==0) return;
  for (size_t j=0;j<i;j++) {
   if (stage_[j] == DISABLED) {
    std::string s ="TST::␣Seeing␣tst␣site␣"+utils::ttos(tstStruct_.sites[i]);
    s =s + "␣before␣having␣seen";
    s = s + "␣site␣"+utils::ttos(j);
    s = s +".␣Please␣order␣your␣tst␣sites␣in␣order␣of␣appearance.\n";
    throw std::runtime_error(s);
   }
  }
 }
```

The little function below returns true if the stages of all the operators to be applied (or of all the sites on which those operators are to be applied) is equal to x. Else it returns false. Valid stages were noted before (cross reference here FIXME).

"TimeStepTargetting.h" 11c≡

```
 bool allStages(size_t x) const
 {
  for (size_t i=0;i<stage_.size();i++)
   if (stage_[i]!=x) return false;
  return true;
 }
```

The function below returns true if no stage is x, else false.

"TimeStepTargetting.h" 12a≡

```
bool noStageIs(size_t x) const
{
 for (size_t i=0;i<stage_.size();i++)
  if (stage_[i]==x) return false;
 return true;
}
```

This function returns a string (human-readable) representation of the stage given by i.

"TimeStepTargetting.h" 12b≡

```
std::string getStage(size_t i) const
{
 switch (stage_[i]) {
  case DISABLED:
   return "Disabled";
   break;
  case OPERATOR:
   return "Applying operator for the first time";
   break;
  case WFT_ADVANCE:
   return "WFT with time stepping";
   break;
  case WFT_NOADVANCE:
   return "WFT without time change";
   break;
 }
 return "undefined";
}
```

"TimeStepTargetting.h" 12c≡

```
computephi19b
computephi210a
```

The function below is a crucial one. It applies $exp(iHt)$ to vector `phi`. Results will be stored in the private data member `targetVectors_`, which is a vector of vectors. First, `triDiag` decomposes the Hamiltonian matrix into a tridiagonal matrix T, and the computed Lanczos vectors are put in the matrix V. Then T is diagonalized in place, and its eigenvalues are put in `eigs`.or ϵ_k Finally we are ready to $V^\dagger T^d agger \exp(i\epsilon t) TV$ in function `calcTargetVectors`.

"TimeStepTargetting.h" 13a≡

```
void calcTimeVectors(
    RealType Eg,
        const VectorWithOffsetType& phi,
    size_t systemOrEnviron)
{
 std::vector<ComplexMatrixType> V(phi.sectors());
 std::vector<ComplexMatrixType> T(phi.sectors());

 std::vector<size_t> steps(phi.sectors());

 triDiag(phi,T,V,steps);

 std::vector<std::vector<RealType> > eigs(phi.sectors());

 for (size_t ii=0;ii<phi.sectors();ii++)
  utils::diag(T[ii],eigs[ii],'V');

 calcTargetVectors(phi,T,V,Eg,eigs,steps,systemOrEnviron);
}
```

As promised, `calcTargetVectors` below applies $exp(iHt)$ to vector `phi` to create the time vectors. This function simply loops over all times t, storing the result in `targetVectors_[i]` for each time index i.

"TimeStepTargetting.h" 13b≡

```
void calcTargetVectors(
    const VectorWithOffsetType& phi,
    const std::vector<ComplexMatrixType>& T,
    const std::vector<ComplexMatrixType>& V,
    RealType Eg,
        const std::vector<VectorType>& eigs,
      std::vector<size_t> steps,
        size_t systemOrEnviron)
{
 targetVectors_[0] = phi;
 for (size_t i=1;i<times_.size();i++) {
  // Only time differences here (i.e. times_[i] not times_[i]+currentTime_)
  calcTargetVector(targetVectors_[i],phi,T,V,Eg,eigs,times_[i],steps);
  //normalize(targetVectors_[i]);
 }
}
```

And this one does a single time t. What we do here is loop over all symmetry sectors of vector `phi`. For each symmetry sector $i0$ we store the result is a chunk r that is `setDataInSector` into vector with offsets `v`.

"TimeStepTargetting.h" 14a≡

```
    void calcTargetVector(
        VectorWithOffsetType& v,
              const VectorWithOffsetType& phi,
        const std::vector<ComplexMatrixType>& T,
        const std::vector<ComplexMatrixType>& V,
        RealType Eg,
              const std::vector<VectorType>& eigs,
            RealType t,
        std::vector<size_t> steps)
    {
     v = phi;
     for (size_t ii=0;ii<phi.sectors();ii++) {
      size_t i0 = phi.sector(ii);
      ComplexVectorType r;
      calcTargetVector(r,phi,T[ii],V[ii],Eg,eigs[ii],t,steps[ii],i0);
      v.setDataInSector(r,i0);
     }
    }
```

And this function below does just one symmetry sector, that with index $i0$. The procedure computes a vector r, and then does $tmp = Tr$, and finaly $r = V.tmp$. The end result is equal to VTr. So, what is r? It is the vector calculated by `calcR` as explained below.

"TimeStepTargetting.h" 14b≡

```
    void calcTargetVector(
        ComplexVectorType& r,
              const VectorWithOffsetType& phi,
        const ComplexMatrixType& T,
        const ComplexMatrixType& V,
        RealType Eg,
              const VectorType& eigs,
            RealType t,
          size_t steps,
        size_t i0)
    {
     size_t n2 = steps;
     size_t n = V.n_row();
     if (T.n_col()!=T.n_row()) throw std::runtime_error("T␣is␣not␣square\n");
     if (V.n_col()!=T.n_col()) throw std::runtime_error("V␣is␣not␣nxn2\n");
     // for (size_t j=0;j<v.size();j++) v[j] = 0; <-- harmful if v is sparse
     ComplexType zone = 1.0;
     ComplexType zzero = 0.0;

     ComplexVectorType tmp(n2);
     r.resize(n2);
     calcR(r,T,V,phi,Eg,eigs,t,steps,i0);
     psimag::BLAS::GEMV('N', n2, n2, zone, &(T(0,0)), n2, &(r[0]), 1, zzero, &(tmp[0]), 1 );
     r.resize(n);
     psimag::BLAS::GEMV('N', n,  n2, zone, &(V(0,0)), n, &(tmp[0]),1, zzero, &(r[0]),
1 );
    }
```

This does one more step in the formula $\psi(t) = V^\dagger T^\dagger exp(iDt)TV\phi$. Note that D is diagonal, its diagonal values are `eigs` below, but we need to substract the ground state energy `Eg`. This returns the vector $exp(iDt)TV\phi$.

"TimeStepTargetting.h" 15a≡

```
void calcR(
 ComplexVectorType& r,
      const ComplexMatrixType& T,
 const ComplexMatrixType& V,
      const VectorWithOffsetType& phi,
      RealType Eg,
 const VectorType& eigs,
      RealType t,
 size_t n2,
 size_t i0)
{
 for (size_t k=0;k<n2;k++) {
  ComplexType sum = 0.0;
  for (size_t kprime=0;kprime<n2;kprime++) {
   ComplexType tmpV = calcVTimesPhi(kprime,V,phi,i0);
   sum += conj(T(kprime,k))*tmpV;
  }
  RealType tmp = (eigs[k]-Eg)*t;
  ComplexType c(cos(tmp),sin(tmp));
  r[k] = c * sum;
 }
}
```

This function does $\sum_x V_{k',x} phi_x$, that is $V\phi$

"TimeStepTargetting.h" 15b≡

```
ComplexType calcVTimesPhi(size_t kprime,const ComplexMatrixType& V,const VectorWithOffsetType& phi,
     size_t i0)
{
 ComplexType ret = 0;
 size_t total = phi.effectiveSize(i0);

 for (size_t j=0;j<total;j++)
  ret += conj(V(j,kprime))*phi.fastAccess(i0,j);
 return ret;
}
```

Here is tridiag, we loop over (non-zero) symmetry sectors:

"TimeStepTargetting.h" 15c≡

```
void triDiag(
  const VectorWithOffsetType& phi,
  std::vector<ComplexMatrixType>& T,
   std::vector<ComplexMatrixType>& V,
  std::vector<size_t>& steps)
{
 for (size_t ii=0;ii<phi.sectors();ii++) {
  size_t i = phi.sector(ii);
  steps[ii] = triDiag(phi,T[ii],V[ii],i);
 }
}
```

And now for each symmetry sector:

"TimeStepTargetting.h" 16a≡

```
size_t triDiag(const VectorWithOffsetType& phi,ComplexMatrixType& T,ComplexMatrixType& V,size_t i0)
{
 size_t p = basisSE_.findPartitionNumber(phi.offset(i0));
 typename ModelType::ModelHelperType modelHelper(p,basisSE_,basisS_,basisE_,model_.orbitals());
    //,useReflection_);
 typename LanczosSolverType::LanczosMatrixType lanczosHelper(&model_,&modelHelper);

 RealType eps= 0.01*ProgramGlobals::LanczosTolerance;
 size_t iter= ProgramGlobals::LanczosSteps;

 //srand48(3243447);
 LanczosSolverType lanczosSolver(lanczosHelper,iter,eps,parallelRank_);

 TridiagonalMatrixType ab;
 size_t total = phi.effectiveSize(i0);
 TargetVectorType phi2(total);
 phi.extract(phi2,i0);
 /* std::ostringstream msg;
 msg<<"Calling tridiagonalDecomposition...\n";
 progress_.printline(msg,std::cerr);*/
 lanczosSolver.tridiagonalDecomposition(phi2,ab,V);
 ab.buildDenseMatrix(T);
 //check1(V,phi2);
 return lanczosSolver.steps();
}
```

A validity check

"TimeStepTargetting.h" 16b≡

```
//! This check is invalid if there are more than one sector
void check1(const ComplexMatrixType& V,const TargetVectorType& phi2)
{
 if (V.n_col()>V.n_row()) throw std::runtime_error("cols > rows\n");
 TargetVectorType r(V.n_col());
 for (size_t k=0;k<V.n_col();k++) {
  r[k] = 0.0;
  for (size_t j=0;j<V.n_row();j++)
   r[k] += conj(V(j,k))*phi2[j];
  // is r(k) == \delta(k,0)
  if (k==0 && std::norm(r[k]-1.0)>1e-5)
   std::cerr<<"WARNING: r[0]="<<r[0]<<" != 1\n";
  if (k>0 && std::norm(r[k])>1e-5)
   std::cerr<<"WARNING: r["<<k<<"]="<<r[k]<<" !=0\n";
 }
}
```

As explained above (cross reference here), we need know before applying an operator were the non-zero sectors are going to be. The operator (think “easy” exciton) does not necessarily have the symmetry of the Hamiltonian, so non-zero sectors of the original vector are not going to coincide with the non-zero sectors of the result vector, neither do the empty sectors be the same. Note that using simpy

:

```
size_t partition = targetVectors_[0].findPartition(basisSE_);
```

doesn’t work, since `targetVectors_[0]` is stale at this pointThis function should not be called one more than one operators will be applied.

"TimeStepTargetting.h" 17a≡

```
void guessPhiSectors(VectorWithOffsetType& phi,size_t i,size_t systemOrEnviron)
{
 FermionSign fs(basisS_,tstStruct_.electrons);
 if (allStages(WFT_NOADVANCE)) {
  VectorWithOffsetType tmpVector = psi_;
  for (size_t j=0;j<tstStruct_.aOperators.size();j++) {
   applyOpLocal_(phi,tmpVector,tstStruct_.aOperators[j],fs,
    systemOrEnviron);
   tmpVector = phi;
  }
  return;
 }
 applyOpLocal_(phi,psi_,tstStruct_.aOperators[i],fs,
     systemOrEnviron);
}
```

The function below makes all target vectors empty:

"TimeStepTargetting.h" 17b≡

```
void zeroOutVectors()
{
 for (size_t i=0;i<targetVectors_.size();i++)
  targetVectors_[i].resize(basisSE_.size());
}
```

The function below prints all target vectors to disk, using the `TimeSerializer` class.

"TimeStepTargetting.h" 17c≡

```
void printVectors(const std::vector<size_t>& block)
{
 if (block.size()!=1) throw std::runtime_error(
  "TST␣only␣supports␣blocks␣of␣size␣1\n");

 TimeSerializerType ts(currentTime_,block[0],targetVectors_);
 ts.save(io_);
}
```

Print header to disk to index the time vectors. This indexing wil lbe used at postprocessing.

"TimeStepTargetting.h" 17d≡

```
void printHeader()
{
 io_.print(tstStruct_);
 std::string label = "times";
 io_.printVector(times_,label);
 label = "weights";
 io_.printVector(weight_,label);
 std::string s = "GsWeight="+utils::ttos(gsWeight_);
 io_.printline(s);
}
```

The `test` function below performs a measurement *in situ*. This is mainly for testing purposes, since measurements are better done, post-processing.

"TimeStepTargetting.h" 18a≡

```
    void test(
      const VectorWithOffsetType& src1,
      const VectorWithOffsetType& src2,
      size_t systemOrEnviron,
       const std::string& label,
      size_t site) const
    {
     VectorWithOffsetType dest;
     OperatorType A = tstStruct_.aOperators[0];
     CrsMatrix<ComplexType> tmpC(model_.getOperator("c",0,0));
     /*CrsMatrix<ComplexType> tmpCt;
     transposeConjugate(tmpCt,tmpC);
     multiply(A.data,tmpCt,tmpC);*/
     A.fermionSign = -1;
     A.data = tmpC;
     FermionSign fs(basisS_,tstStruct_.electrons);
     applyOpLocal_(dest,src1,A,fs,systemOrEnviron);

     ComplexType sum = 0;
     for (size_t ii=0;ii<dest.sectors();ii++) {
      size_t i = dest.sector(ii);
      size_t offset1 = dest.offset(i);
      for (size_t jj=0;jj<src2.sectors();jj++) {
       size_t j = src2.sector(jj);
       size_t offset2 = src2.offset(j);
       if (i!=j) continue; //throw std::runtime_error("Not same sector\n");
       for (size_t k=0;k<dest.effectiveSize(i);k++)
        sum+= dest[k+offset1] * conj(src2[k+offset2]);
      }
     }
     std::cerr<<site<<" "<<sum<<" "<<" "<<currentTime_;
     std::cerr<<" "<<label<<std::norm(src1)<<" "<<std::norm(src2)<<" "<<std::norm(dest)<<"\n";
    }
```

"TimeStepTargetting.h" 18b≡

```
  privatedata3c
  };      //class TimeStepTargetting
 } // namespace Dmrg

 #endif
```