

0.1 Dynamic Targetting To get Frequency Dependent Observables

This class implements the dynamic DMRG algorithm as described in [1]. Note that this class implements the Targetting interface also used by GroundStateTargetting (static DMRG) and TimeStepTargetting (for time dependent DMRG)

Following paper reference [1], the name *dyn-vectors* will be used for the four vector: (i) the ground state $|\psi_{gs}\rangle$, (ii) the vector $A|\psi_{gs}\rangle$, (iii) the “correction vector” $|Y_A\rangle$, and (iv) the “correction vector” $|X_A\rangle$, as defined in the paper. The last 3 vectors will be stored in the private member `targetVectors_`, whereas the first one will be stored in the private member `psi_`.

```
"DynamicTargetting.h" 1≡
#ifndef DYNAMICTARGETTING_H
#define DYNAMICTARGETTING_H

#include "ProgressIndicator.h"
#include "BLAS.h"
#include "ApplyOperatorLocal.h"
#include "TimeSerializer.h"
#include "DynamicDmrgParams.h"
#include "DynamicFunctional.h"
#include "Minimizer.h"

namespace Dmrg {
    theClassHere2
} // namespace
#endif // DYNAMICTARGETTING_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 introduced 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. `ConcurenyType` 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.

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

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).

`< theClassHere 2 > ≡`

```
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 DynamicTargetting {
public:
  publicTypeDefs3a
  enumsAndConstants3b
  constructor4a
  publicFunctions6a
private:
  privateFunctions9c
  privateData4b
}; // class DynamicTargetting
```

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

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

`< publicTypedefs 3a > ≡`

```

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 OperatorsType::SparseMatrixType SparseMatrixType;
typedef typename ModelType::MyBasisWithOperators BasisWithOperatorsType;
typedef std::vector<ComplexType> ComplexVectorType;
typedef LanczosSolverTemplate<RealType, InternalProductType, ComplexVectorType> LanczosSolverType;
//typedef std::vector<RealType> VectorType;
typedef psimag::Matrix<ComplexType> ComplexMatrixType;
//typedef typename LanczosSolverType::TridiagonalMatrixType TridiagonalMatrixType;
typedef typename BasisWithOperatorsType::OperatorType OperatorType;
typedef typename BasisWithOperatorsType::BasisType BasisType;
typedef DynamicDmrgParams<ModelType> TargettingParamsType;
typedef typename BasisType::BlockType BlockType;
typedef VectorWithOffsetTemplate<ComplexType> VectorWithOffsetType;
typedef typename VectorWithOffsetType::VectorType VectorType;
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 advancing in space with the wave function transformation (WFT) but not advancing in frequency.
4. WFT_NOADVANCE In this stage we're advancing in space and frequency

`< enumsAndConstants 3b > ≡`

```

enum {DISABLED, OPERATOR, CONVERGING};
enum { EXPAND_ENVIRON=WaveFunctionTransformationType::EXPAND_ENVIRON,
  EXPAND_SYSTEM=WaveFunctionTransformationType::EXPAND_SYSTEM,
  INFINITE=WaveFunctionTransformationType::INFINITE };
static const size_t parallelRank_ = 0; // DYNt needs to support concurrency FIXME

```

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

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 TargettingStructureParams object. 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.

⟨ *constructor 4a* ⟩ ≡

```
DynamicTargetting(  
  const BasisWithOperatorsType& basisS,  
  const BasisWithOperatorsType& basisE,  
  const BasisType& basisSE,  
  const ModelType& model,  
  const TargettingParamsType& tstStruct,  
  const WaveFunctionTransformationType& wft)  
: stackInitialization5a  
  constructorBody5b
```

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

⟨ *privateData 4b* ⟩ ≡

```
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 currentOmega_  
std::vector<VectorWithOffsetType> targetVectors_  
std::vector<RealType> weight_  
RealType gsWeight_  
typename IoType::Out io_  
ApplyOperatorType applyOpLocal_
```

Now we get to the stack initialization of this object. 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 frequency that we are calculating here needs to be

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

described FIXME. We will think this to do one frequency at a time, as suggested by the reference above. Multiple frequencies should be parallelized, and we need to provide a frequency range and stepping in the input file (FIXME) The weight, which is a vector of weights, for each target state (except possibly the ground state) is set to 4 vectors. 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. All right, we may do something *in situ* just to check. The `applyLocal` operator described before is also initialized on the stack.

`< stackInitialization 5a > ≡`

```
stage_(tstStruct.sites.size(),DISABLED),
basis_(basisS),
basisE_(basisE),
basisSE_(basisSE),
model_(model),
tstStruct_(tstStruct),
waveFunctionTransformation_(wft),
progress_("DynamicTargetting",0),
currentOmega_(tstStruct_.omega),
targetVectors_(3),
weight_(targetVectors_.size()),
io_(tstStruct_.filename,parallelRank_),
applyOpLocal_(basisS,basisE,basisSE)
```

The body of the constructor follows:

`< constructorBody 5b > ≡`

```
{
  if (!wft.isEnabled()) throw std::runtime_error("_DynamicTargetting_"
    "needs_an_enabled_wft\n");
  RealType sum = 0;
  size_t n = weight_.size();
  for (size_t i=0;i<n;i++) {
    weight_[i] = 1.0/(n+1);
    sum += weight_[i];
  }

  gsWeight =1.0-sum;
  sum += gsWeight;
  if (fabs(sum-1.0)>1e-5) throw std::runtime_error("Weights_don't_amount_to_one\n");
  printHeader();
}
```

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

⟨ *publicFunctions 6a* ⟩ ≡

```
weight6b
gsWeight6c
normSquared6d
setGs7a
operatorBracket7b
gs7c
includeGroundStage7d
size7e
operatorParens8a
evolve8b
initialGuess9a
BasisGetFunctions9b
```

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

⟨ *weight 6b* ⟩ ≡

```
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 state. During the disabled stages it is 1 since there are no other vectors to target.

⟨ *gsWeight 6c* ⟩ ≡

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

This member function returns the squared norm of dynamic vector number i . (Do we really need this function?? (FIXME))

⟨ *normSquared 6d* ⟩ ≡

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

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

The function below 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.

$\langle setGs\ 7a \rangle \equiv$

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

The functions below returns the i -th element of the ground state psi .

$\langle operatorBracket\ 7b \rangle \equiv$

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

The function below returns the full ground state vector as a vector with offset:

$\langle gs\ 7c \rangle \equiv$

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

The function below tells if the ground state will be included in the density matrix. If using this class it will always be.

$\langle includeGroundStage\ 7d \rangle \equiv$

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

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

$\langle size\ 7e \rangle \equiv$

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

The function below returns the full time vector number i as a vector with offsets:

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

⟨ *operatorParens 8a* ⟩ ≡

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

This function provides a hook to (possibly) start the computation of dynamic observables. 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 dyn-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.

This function call other functions. We'll continue linearly describing each one in order of appearance.

⟨ *evolve 8b* ⟩ ≡

```
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;
    }

    ComplexType val = calcDynVectors(Eg, phiNew, direction);
    cocoon(val, direction, block); // in-situ

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

The function below provides an initial guess for the Lanczos vector. Traditionally, when DMRG is only targetting the ground state this is a standard

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

procedure (see file `GroundStateTargetting`). Here, when `DynamicTargetting`, 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::setInitialVector` function. When stages are advancing we need to weight each target wave-function-transformed state with the appropriate weights:

⟨ *initialGuess 9a* ⟩ ≡

```
void initialGuess(VectorWithOffsetType& v) const
{
    waveFunctionTransformation_.setInitialVector(v, psi_, basisS_, basisE_, basisSE_);
    if (!allStages(CONVERGING)) 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.

⟨ *BasisGetFunctions 9b* ⟩ ≡

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

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

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

This completes the list of public functions. What remains are private (i.e. non-exported) code used only by this class. We'll visit one function at a time.

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

`<privateFunctions 9c> ≡`

```
evolvePrivate10a
computePhi11c
cocoon12b
checkOrderOfSites13a
allStages13b
noStages13c
getStage14a
calcDynVectors14b
minimizeFunctional15a
minimizeFunctional215b
obtainXA15c
obtainXA216a
guessPhiSectors16b
zeroOutVectors16c
printVectors17a
printHeader17b
test?
areAllTargetsSensible?
isThisTargetSensible?
```

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.

`<evolvePrivate 10a> ≡`

```
size_t evolve(
    size_t i,
    VectorWithOffsetType& phiNew,
    VectorWithOffsetType& phiOld,
    RealType Eg,
    size_t direction,
    const BlockType& block,
    size_t loopNumber,
    size_t lastI)
{
    checkIfWeAreInTheRightLoop10b
    checkIfAddedBlockIsSizeOne10c
    checkStage10d
    checkOperator11a
    computeAtimesPsi11b
    return 1;
}
```

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:

`<checkIfWeAreInTheRightLoop 10b> ≡`

```
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:

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

`< checkIfAddedBlockIsSizeOne 10c > ≡`

```
if (block.size()!=1) throw
std::runtime_error("DynamicTargetting::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:

`< checkStage 10d > ≡`

```
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 are presented to us by the DMRG sweeping. This is explained below under function `checkOrder`

`< checkOperator 11a > ≡`

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

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

`< computeAtimesPsi 11b > ≡`

```
std::ostringstream msg;
msg<<" Evolving ,_stage="<<getStage(i)<<"_site="<<site<<"_loopNumber="<<loopNumber;
msg<<"_Eg="<<Eg;
progress_.println(msg, std::cout);

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

Let us look at `computephi`, the next private function.

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

⟨ *computePhi 11c* ⟩ ≡

```
void computePhi(size_t i, VectorWithOffsetType& phiNew,
  VectorWithOffsetType& phiOld, size_t systemOrEnviron)
{
  if (stage_[i]==OPERATOR) {
    computePhiOperator11d
  } else if (stage_[i]== CONVERGING) {
    computePhiAdvance12a
  } else {
    throw std::runtime_error("It's_5_am,_do_you_know_what_line_"
      "_your_code_is_exec-ing?\n");
  }
}
```

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`.

⟨ *computePhiOperator 11d* ⟩ ≡

```
std::ostream msg;
msg<<"I'm_applying_a_local_operator_now";
progress_.println(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.

⟨ *computePhiAdvance 12a* ⟩ ≡

```
std::ostream msg;
msg<<"I'm_calling_the_WFT_now";
progress_.println(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_[0],
  basisS_, basisE_, basisSE_); // generalize for su(2)
phiNew.collapseSectors();
```

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

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

⟨ `cocoon` 12b ⟩ ≡

```
void cocoon(ComplexType& val, 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);
    std::cerr<<"OMEGA_"<<currentOmega<<"_"<<imag(val)<<"_"<<real(val)<<"_"<<site<<"\n";
    for (size_t j=0;j<targetVectors_.size();j++) {
        std::string s = "<P"+utils::ttos(j)+"|A|P"+utils::ttos(j)+">";
        test(targetVectors_[j], targetVectors_[0], 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 needs to be the same as the order in which the DMRG sweeping process encounters those sites. Else we throw.

⟨ `checkOrderOfSites` 13a ⟩ ≡

```
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_dynamic_site_" + utils::ttos(stage_.sites[i]);
            s = s + "_before_having_seen";
            s = s + "_site_" + utils::ttos(j);
            s = s + "_Please_order_your_dynamic_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`).

⟨ `allStages` 13b ⟩ ≡

```
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.

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

⟨noStageIs 13c⟩ ≡

```
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 .

⟨getStage 14a⟩ ≡

```
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 CONVERGING:
    return "Converging_DDMRG";
    break;
  }
  return "undefined";
}
```

The below function computes steps 3 and 4 of the algorithm described in page 3 of reference [1]. The incoming arguments are the ground state energy E_g , the vector ϕ or $|\phi\rangle$ which is $|\phi\rangle \equiv A|\psi_{gs}\rangle$, and the direction of growth specified in `systemOrEnviron`. Note that ϕ will be stored in `targetVectors_[0]`, $|Y_A\rangle$ in `targetVectors_[1]`, $|X_A\rangle$ in `targetVectors_[2]`.

⟨calcDynVectors 14b⟩ ≡

```
ComplexType calcDynVectors(
  RealType Eg,
  const VectorWithOffsetType& phi,
  size_t systemOrEnviron)
{
  RealType retIm = minimizeFunctional(targetVectors_[1],Eg,phi,systemOrEnviron);
  obtainXA(targetVectors_[2],targetVectors_[1],Eg);
  RealType retRe = -real(targetVectors_[2]*phi)/M_PI; // Eq.~(12a)
  targetVectors_[0] = phi;
  areAllTargetsSensible();
  return ComplexType(retRe,retIm);
}
```

After `calcDynVectors` is called, `psiMin` contains $|Y_A\rangle$, as explained in Eq. (15). From Eq. (16), `iaw` contains $I_A(\omega)$.

Below we minimize Eq. (14) of reference [1], and obtain ψ_{min} which is stored in `psiMin`.

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

$\langle minimizeFunctional\ 15a \rangle \equiv$

```
RealType minimizeFunctional(
    VectorWithOffsetType& psiMin,
    RealType Eg,
    const VectorWithOffsetType&phi,
    size_t systemOrEnviron)
{
    VectorWithOffsetType phiCopy = phi;
    psiMin = phi;
    RealType ret = 0;
    for (size_t i=0;i<phiCopy.sectors();i++) {
        VectorType sv;
        size_t ii = phiCopy.sector(i);
        psiMin.extract(sv, ii);
        if (sv.size()==0) throw std::runtime_error("Non-zero_sector_is_zero!\n");
        ret += minimizeFunctional(sv,Eg,phi,ii);
        psiMin.setDataInSector(sv, ii);
    }
    return ret;
}
```

The function computes the minimum of the W functional and returns the complex number $Im[G(\omega + i\eta)]$. Note that the return values use (16).

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

$\langle minimizeFunctional2 \ 15b \rangle \equiv$

```

RealType minimizeFunctional(VectorType& sv, RealType Eg, const VectorWithOffsetType& phi, size_t ind)
{
    size_t p = basisSE_.findPartitionNumber(phi.offset(ind));
    typename ModelType::ModelHelperType modelHelper(p, basisSE_, basisS_, basisE_, model_.orbitals());
    typedef typename LanczosSolverType::LanczosMatrixType LanczosMatrixType;
    LanczosMatrixType h(&model_, &modelHelper);
    typedef DynamicFunctional<RealType, LanczosMatrixType, VectorType> DynamicFunctionalType;
    VectorType aVector;
    phi.extract(aVector, ind);
    DynamicFunctionalType wFunctional(h, aVector, currentOmega_, Eg, tstStruct_.eta);
    size_t maxIter = 1000;

    PsmagLite::Minimizer<RealType, DynamicFunctionalType> min(wFunctional, maxIter);
    std::vector<RealType> svReal(2*sv.size());
    //wFunctional.packComplexToReal(svReal, sv);
    for (size_t i=0; i<svReal.size(); i++) svReal[i]=drand48();
    RealType norma = std::norm(svReal);
    for (size_t i=0; i<svReal.size(); i++) svReal[i]/=norma;

    int iter = -1;
    RealType delta = 1e-3;
    RealType tolerance = 1e-3;
    size_t counter = 0;
    while (iter<0 && counter<100) {
        iter = min.simplex(svReal, delta, tolerance);
        delta /= 2;
        tolerance *= 1.2;
        counter++;
    }
    if (iter<0) {
        std::cerr<<"delta="<<delta<<"_tol="<<tolerance<<"\n";
        throw std::runtime_error
            ("DynTargetting::minimizeFunctional(...):No_minimum_found\n");
    }
    wFunctional.packRealToComplex(sv, svReal);
    return -wFunctional(svReal)/(M_PI*tstStruct_.eta);
}

```

The function below implements Eq. (11) of reference [1].

$\langle obtainXA \ 15c \rangle \equiv$

```

void obtainXA(
    VectorWithOffsetType& xa,
    const VectorWithOffsetType& ya,
    RealType Eg)
{
    xa = ya;
    for (size_t i=0; i<ya.sectors(); i++) {
        size_t ii = ya.sector(i);
        obtainXA(xa, Eg, ya, ii);
    }
}

```

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

`< obtainXA2 16a > ≡`

```

void obtainXA(VectorWithOffsetType& xa, RealType Eg, const VectorWithOffsetType& ya, size_t i)
{
    size_t p = basisSE_.findPartitionNumber(ya.offset(i));
    typename ModelType::ModelHelperType modelHelper(p, basisSE_, basisS_, basisE_, model_.orbitals());
    typedef typename LanczosSolverType::LanczosMatrixType LanczosMatrixType;
    LanczosMatrixType h(&model_ & modelHelper);
    VectorType yaThisSector;
    ya.extract(yaThisSector, i);
    VectorType sv(yaThisSector.size(), 0.0);
    h.matrixVectorProduct(sv, yaThisSector); // sv = H * yaThisSector
    RealType factor = (Eg + currentOmega_);
    sv -= (yaThisSector * factor);
    sv *= (1/tstStruct_.eta);
    xa.setDataInSector(sv, i);
}

```

`void areAllTargetsSensible`

As explained above (cross reference here), we need to know before applying an operator where the non-zero sectors are going to be. The operator (think c^\dagger) does not necessarily have the symmetry of the Hamiltonian, so non-zero sectors of the original vector are not—in general—going to coincide with the non-zero sectors of the result vector, neither will the empty sectors be the same. Note that using simply

```

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

```

doesn't work, since `targetVectors_[0]` is stale at this point. This function should not be called when more than one operator will be applied.

`< guessPhiSectors 16b > ≡`

```

void guessPhiSectors(VectorWithOffsetType& phi, size_t i, size_t systemOrEnviron)
{
    FermionSign fs(basisS_, tstStruct_.electrons);
    if (allStages(CONVERGING)) {
        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:

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

⟨ *zeroOutVectors* 16c ⟩ ≡

```
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.

⟨ *printVectors* 17a ⟩ ≡

```
void printVectors(const std::vector<size_t>& block)
{
    if (block.size()!=1) throw std::runtime_error(
        "DynamicTargetting_only_supports_blocks_of_size_1\n");

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

Print header to disk to index the time vectors. This indexing will be used at postprocessing.

⟨ *printHeader* 17b ⟩ ≡

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

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

0.1. DYNAMIC TARGETTING TO GET FREQUENCY DEPENDENT OBSERVABLES

`< test ? > ≡`

```

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.makeDiagonal(tmpC.rank(),1.0);
    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<<"_ "<<"_ "<<currentOmega_;
    std::cerr<<"_ "<<label<<std::norm(src1)<<"_ "<<std::norm(src2)<<"_ "<<std::norm(dest)<<"\n";
}

```

The function below is just for checking:

`< areAllTargetsSensible ? > ≡`

```

void areAllTargetsSensible() const
{
    for (size_t i=0;i<targetVectors_.size();i++)
        isThisTargetSensible(i);
}

```

`< isThisTargetSensible ? > ≡`

```

void isThisTargetSensible(size_t i) const
{
    RealType norma = std::norm(targetVectors_[i]);
    if (norma<1e-6) throw std::runtime_error("Norma_is_zero\n");
}

```

Bibliography

- [1] E. Jeckelmann. *Phys. Rev. B*, 66:045114, 2002.