TBTK - Tight Binding ToolKit

A c++ library for solving tight-binding models

Model

Designed to solve any bilinear Hamiltonian

$$H = \sum_{ij} a_{ij} c_i^\dagger c_j .$$

Model: Indices and HoppingAmplitudes

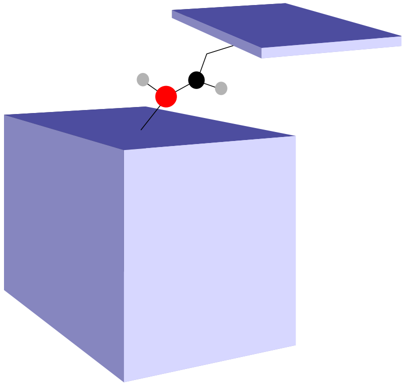
Flexible indices allows for general geometries to be defined that are not necessarily indexable by a single type of index. Theses are generally reffered to as 'to' and 'from' indices, and are used to specify 'HoppingAmplitudes' a_{ij}.

$$H = \sum_{ij} a_{ij} c_i^{\dagger} c_j.$$

Index	Also known as	Example
i	to	{subsystem, x, y, spin}
j	from	{subsystem, x, y, z, orbital, spin}

Model: Example

Consider for example a system consisting of three subsystems of different dimension.



Model: Example

A possible indexing scheme might look like this.

$$\begin{aligned} \text{3D bulk: } & \{0, x, y, z, \text{orbital}, \text{spin}\}, \\ \text{1D molecule: } & \{1, x, \text{orbital}, \text{spin}\}, \\ \text{2D sheet: } & \{2, x, y, \text{spin}\}. \end{aligned}$$

Model: Important functions

Setup model

```
    Model model;
```

Add hopping amplitude

```
    model.addHA(HoppingAmplitude(-mu, {x, y, s}, {x, y, s}));
```

Add hopping amplitude and Hermitian conjugate

```
    model.addHAAndHC(HoppingAmplitude(-t,
                                      {x+1, y, s},
                                      {x, y, s}));
```

Construct model. Hilbert space is created.

```
    model.construct();
```

Model: Example

```
Model model;
for(int x = 0; x < SIZE_X; x++){
    for(int y = 0; y < SIZE_Y; y++){
        for(int s = 0; s < 2; s++){
            model.addHA(-mu, {x, y, s}, {x, y, s});
            if(x+1 < SIZE_X)
                model.addHAAndHC(-t, {x+1, y, s}, {x, y, s});
        }
    }
}
model.construct();
```

Solvers

- DiagonalizationSolver (Eigenvalues and eigenvectors)
- ChebyshevSolver (Green's function)

DiagonalizatioSolver

The DiagonalizationSolver can be used to

- Calculate eigenvalues and eigenvectors.
- Self-consistently or non self-consistently.
- Calculate custom physical quantities by accessing the wave function directly using physical indices...
- ...or use a "property extractor" to extract common properties such as eigenvalues, DOS, spin-polarized LDOS, magnetization, etc.
- Have been tested for Hamiltonians with a basis size of up to 14000

Diagonalization solver: Important functions

```
Create diagonalization solver
    DiagonalizationSolver dSolver;

Set model
    dSolver.setModel(&model);

Diagonalize Hamiltonian. Is done self-consistently if a
self-consistenct callback has been set
    dSolver.run();

Set self-consisteny callback, where scCallback is a user
defined callback function
    dSolver.setSCCallabck(scCallback);
```

Diagonalization solver: Important functions

```
Set maximum number of self-consistency iterations
    dSolver.setMaxIterations(MAX_ITERATIONS);

Get eigenvalues
    dSolver.getEigenValues();

Get amplitude of eigenstate n at index (x, y, s)
    dSolver.getAmplitude(n, {x, y, s});

Create property extractor
    PropertyExtractor pe(&dSolver);

Get eigenvalues
    double *ev = pe.getEV();
```

Diagonalization solver: Important functions

```
Calculate DOS with a upper and lower energy cutoff of U_LIM
and L_LIM, respectively, and a resolution of RESOLUTION
number of points between L_LIM and U_LIM.
    double *dos = pe.calculateDOS(U_LIM, L_LIM, RESOLUTION);

Calculate electron density. Will be calculated for all x and
y, and spin-indices will be summed to create a total density.
    double *density = pe.calculateDensity(
                            {IDX_X, IDX_Y, IDX_SUM_ALL},
                            {SIZE_X, SIZE_Y, 2});
```

Diagonalization solver: Important functions

Calculate magnetization. Will be calculated for all x along the line y = SIZE_Y/2.

```
    double *mag = pe.calculateMAG(
                      {IDX_X, SIZE_Y/2, IDX_SPIN},
                      {SIZE_X, 1, 2});
```

Diagonalization solver: Important functions

```
Calculate spin-polarized LDOS. Will be calculated for all y
along the line x = SIZE_X/2. Note that IDX_X is used for the
y-index here. IDX_X, IDX_Y, IDX_Z, does for the property
extractor refer to the first, second, and third loop index,
and the y index is the first loop index when x is fixed. The
upper and lower energy cutoff is set by U_LIM and L_LIM,
respectively, and RESOLUTION number of points are used
between the two limits.
    double *sp_ldos = pe.calculateSP_LDOS(
                            {SIZE_X/2, IDX_X, IDX_SPIN},
                            {1, SIZE_Y, 2}
                            U_LIM, L_LIM, RESOLUTION);
```

DiagonalizationSolver: Example

```
//Non self-consistently
DiagonalizationSolver dSolver;
dSolver.setModel(&model);
dSolver.run()

//Self-consistently (requires scCallback to be defined).
DiagonalizationSolver dSolver;
dSolver.setModel(&model);
dSolver.setSCCallback(scCallback);
dSolver.setMaxIterations(MAX_ITERATIONS);
dSolver.run();
```

Chebyshev solver

$$G_{ij}(E) = \frac{-2i}{\sqrt{1-E^2}} \sum_{n}^{\infty} b_{ij}^{(n)} e^{-in \arccos(E)}.$$

- Calculates the Green's function
- Can be done either on CPU or GPU
- Works for Hamiltonians with a basis size of at least 4 million

Chebyshev solver: Important functions

```
Create Chebyshev solver
    ChebyshevSolver cSolver;

Set model
    cSolver.setModel(&model);

Calculate Chebyshev coefficeints for G_{ij}, where
i = (x, y, 0), j = (x+1, y, 1)
    complex<double> coefficients[NUM_COEFFICIENTS];
    cSolver.calculateCoefficeints({x, y, 0}, {x+1, y, 1},
                                  coefficients,
                                  NUM_COEFFICIENTS);
    cSolver.calculateCoefficeintsGPU({x, y, 0}, {x+1, y, 1},
                                     coefficients,
                                     NUM_COEFFICIENTS);
```

Chebyshev solver: Important functions

```
Calculate Chebyshev coefficeints for G_{ij}, where i is a
range of NUM_I indices and = (x, y, 1)
    complex<double> coefficients[NUM_COEFFICIENTS*NUM_I];
    vector<Index> toIndices;
    for(int x = 0; x < NUM_I; x++)
         toIndices.push_back({x, y, 0});
    cSolver.calculateCoefficeintsGPU(toIndices, {x, y, 1},
                                     coefficients,
                                     NUM_COEFFICIENTS);
```

Chebyshev solver: Important functions

Evaluate Green's function without using lookup table

```
    complex<double> greensFunction[ENERGY_RESOLUTION];
    cSolver.generateGreensFunction(greensFunction,
                                   coefficients,
                                   NUM_COEFFICIENTS,
                                   ENERGY_RESOLUTION);
```

Setup lookup table

```
    cSolver.generateLookupTable(NUM_COEFFICIENTS,
                                ENERGY_RESOLUTION);
```

Load lookup table to GPU

```
    cSolver.loadLookupTableGPU();
```

Chebyshev solver: Important functions

Generate Green's function using lookup table

```
    complex<double> greensFunction[ENERGY_RESOLUTION];
    cSolver.generateGreensFunction(greensFunction,
                                   coefficeints);
    cSolver.generateGreensFunctionGPU(greensFunction,
                                      coefficeints);
```

Free GPU memory from lookup table

```
    cSolver.destroyLookupTableGPU();
```