User Guide

Xatu 1.3.2

A. J. Uría Álvarez[1], J. J. Esteve-Paredes[1], M. A. García-Blázquez[1], and J. J. Palacios[1,2]

[1]Universidad Autónoma de Madrid
[2]Condensed Matter Physics Center (IFIMAC), Instituto Nicolás Cabrera

June 17, 2024

Contents

1 COMPILATION

Xatu is built upon the Armadillo C++ library for linear algebra, which is also based on the standard libraries por linear algebra, namely BLAS, LAPACK and ARPACK. Next we show some example installation instructions for different operating systems.

- **Ubuntu, Debian or WSL**

Install the required libraries:

```
$ apt-get install libopenblas-dev liblapack-dev libarpack2-dev
    libarmadillo-dev
```

The library (`libxatu.a`) can be automatically built running the following command:

```
$ make build
```

Then, the Xatu binary is built running:

```
$ make xatu
```

Alternatively, one can define scripts that make use of the functions defined in the library. To compile them, it suffices to put the script in the `/main` folder and run

```
$ make [script]
```

- **MacOS**

For MacOS the dependencies can be installed via `brew`. To build the library it is recommended to use brew's `gcc` compiler instead of `clang`.

```
$ brew install gcc openblas lapack arpack armadillo
```

Then, specify in the Makefile the new compiler as well as the location of the libraries:

```
CC = g++-13
INCLUDE = -I$(PWD)/include -I/opt/homebrew/include -I/opt/homebrew/opt/
    openblas/include
LIBS = -DARMA_DONT_USE_WRAPPER -L$(PWD) -L/opt/homebrew/lib -L/opt/
    homebrew/opt/openblas/lib -lxatu -larmadillo -lopenblas -llapack
    -fopenmp -lgfortran -larpack
```

- **General**

In case that the libraries are not available through repositories, it is always possible to manually download and compile them. For specific instructions on how to install each library we refer to the documentation provided by each of these libraries. For instance, for Armadillo clone the its repository:

```
$ git clone https://gitlab.com/conradsnicta/armadillo-code.git
```

To install Armadillo run:

```
$ cd armadillo-code
$ cmake .
$ make install
```

Once they are compiled, to link the libraries we have to modify the Makefile to specify the directories where they are installed (e.g. Armadillo and OpenBLAS):

```
INCLUDE = -I/dir/armadillo/include -I/another_dir/OpenBLAS/include/
LIBS = -L/another_dir/OpenBLAS/lib
```

1.1 HDF5

To be able to use HDF5 files with `Xatu`, both the library and the executable must have been compiled with support for it. This requires having installed `libhdf5`. For instance, in Ubuntu it is already packaged and can be installed with:

```
$ apt-get install libhdf5-dev
```

Or in MacOS:

```
$ brew install hdf5
```

To do so, one must provide an additional flag during compilation:

```
$ make build HDF5=1
```

And likewise for the binary:

```
$ make xatu HDF5=1
```

As a rule, anything that requires HDF5 must be compiled using this flag, for instance for scripts that use the functions defined in `libxatu.a`:

```
$ make [script_name] HDF5=1
```

1.2 DEBUG

There is an additional flag to remove the compiler optimizations and activate the debugging mode. Compile the library with:

```
$ make build DEBUG=1
```

And likewise for the binary:

```
$ make xatu DEBUG=1
```

The same can be done for debugging of scripts written using the API:

```
$ make [script_name] DEBUG=1
```

Finally, all flags are compatible between them, meaning that it is possible to compile the library with HDF5 support and in debug mode as well:

```
$ make build HDF5=1 DEBUG=1
```

2 CLI USAGE

The code has been developed with a hybrid approach in mind: one can resort to configuration files to run the program, in analogy with DFT codes, or instead program both the non-interacting system and run the exciton simulation using the provided API. First we are going to discuss its usage with configuration files. The basic usage as a CLI program is described by:

```
xatu [OPTIONS] systemfile [excitonfile]
```

The executable always expects one file describing the system where we want to compute the excitons, and then another file specifying the parameters of the simulation. Their content is addressed in the next sections. The executable can also take optional flags, generally to tune the output of the simulation. By default, running the program without additional flags prints the exciton energies, without writing the results to any file.

`-h (--help)`

Used to print a help message with the usage of the executable and a list of all possible flags that may be passed. The simulation is not performed (even in presence of configuration files).

`-s (--states) nstates`

The number of states specified with this flag is also used for any of the output flags. By default, the number of states is 8 (i.e. if the flag is not present).

`-p (--precision) decimals`

One can specify the number of decimals used when printing the exciton energies. This is relevant to detect state degeneracy without inspecting manually the states. Defaults to 6 decimals if not present.

`-d (--dft) [ncells]`

This flag is used to indicate that the `systemfile` provided corresponds to a CRYSTAL output file, instead of following the standarized format. DFT calculations usually involve several unit cells to determine the Bloch Hamiltonian, so the optional value `ncells` can be passed to specify how many we want to take into account. Otherwise all of them are read and used.

`-eck (--energy, --eigenstates, --kwf)`

The optional flags `-e`, `-c`, `-k`, `-r` are used to specify which exciton output is written to file. `-e` writes the energies, `-c` writes the eigenvectors, `-k` writes the reciprocal density. Note that they can be combined instead of being written separately (e.g. `-ek` instead of `-e -k`).

`-r (--rswf) [holeIndex] [-r ncells]`

Used to write the real-space probability densities to a file. One can give the index of the atom where the hole is located (defaults to first atom of the motif). It can be used a second time to specify the number of unit cells where we want to compute the amplitude (e.g. `-r 1 -r 10` fixes the hole at the second atom of the motif, and uses 10 unit cells along each axis).

`-s (--spin)`

Computes the total spin of the excitons, and writes it to a file. This assumes that the single-particle basis includes spin without performing any check, so incorrect usage could result in wrong results or runtime errors.

`-a (--absorption)`

Computes the optical conductivity (which reflects the absorption of light up to a constant factor) as a function of frequency using the exciton spectrum, and saves the result to a file. A file named "kubo_w.in" with the adequate format (shown below) must be present in the working directory.

`-m (--method) diag | davidson | sparse`

Choose method to obtain the eigenstates of the BSE. By default, full diagonalization is used. If the Davidson or sparse (Lanczos) method is selected, then it is used to compute the number of states specified before.

`-b (--bands) kpointsfile`

To check that the system file was written correctly, one can use this option to diagonalize the Bloch Hamiltonian on the **k** points specified on a file, and write the energy bands to a file. No exciton calculation is performed.

`-f (--format) model | hdf5`

Flag used to specify the type of system file in the calculation. If not present it defaults to 'model', which is the native format implemented in the code. Note that to be able to use the hdf5 format, the code must have been compiled with HDF5 support (see compilation section for more details).

3 API

So far we have discussed a more streamlined usage of the package. In some cases, however, the user could benefit from accessing directly the results of an exciton calculation, instead of having to dump it to a file to postprocess it later. To enable this possibility, the package has been also designed as a library, meaning that one can import the classes and functions defined in the API, and use them to build some extra functionality.

To do so, the package provides a header file which defines a namespace. Within the namespace we have access to all the exciton functionality, which is completely documented. For instructions on how to build the documentation, we refer to the project repository where the most up-to-date information will be present. Additionally, some usage examples can be found under the root directory in the folder `/main`.

The outline for a general exciton simulation is the following: one first has to create a System object, which can be done with a system file. Alternatively, one can define a subclass that inherits from System, and use it to implement the desired behaviour (namely the Bloch Hamiltonian). Then this System is passed on to the Exciton class, which we configure with the desired parameters. The interacting Hamiltonian is initialized and solved, returning a Result object which contains the eigenvalues and eigenvectors. With this, now we can compute some observables, or instead use these states to perform some other calculations out of the scope of the code.

The documentation for the library is generated using Doxygen. To build it, we have to install Doxygen and then run from the `/docs` folder:

```
$ doxygen docs.cfg
```

4 INPUT

4.1 Structure of a model file

The system configuration files contain all the information needed to characterize completely the material of study: it provides the lattice vectors and motif positions, which is required for the real-space evaluation of the excitons. Then, we have the number of orbitals of each unique chemical species, which is needed to compute the matrix elements correctly, and the filling, which determines which bands participate in the formation of the exciton. Finally, the file contains the matrices needed to build the Bloch Hamiltonian, this is, the Fock matrices $H(\mathbf{R})$ and their corresponding Bravais vectors $\mathbf{R}$. The Bloch Hamiltonian is then reconstructed as:

$$H(\mathbf{k}) = \sum_{\mathbf{R}} H(\mathbf{R}) e^{i\mathbf{k}\cdot\mathbf{R}} \tag{1}$$

Note that even though one has to provide the orbitals of each species, the specific type of orbital is not needed since the interaction is computed using the point-like approximation. A system file is specified using labels for each block. Blocks always begin with the block delimiter `#`, followed by a label. A block is then defined as all the content between two consecutive block delimiters.

The expected content for each label will be discussed next. Any line containing `!` is regarded as a comment, and empty lines are skipped.

`BravaisLattice`: Basis vectors of the Bravais lattice. The number of vectors present is also used to determine the dimensionality of the system. The expected format is one vector per line, `x y z`.

`Motif`: List with the positions and chemical species of all atoms of the motif (unit cell). The chemical species are specified with an integer index, used later to retrieve the number of orbitals of that species. The expected format is one atom per line, `x y z index`.

`Orbitals`: Number of orbitals of each chemical species present. The position of the number of orbitals for each species follows the indexing used in the motif block. This block expects one or more numbers of orbitals, the same as the number of different species present, `n1 [n2 ...]`.

`Filling`: Total number of electrons in the unit cell. Required to identify the Fermi level, which is the reference point in the construction of the excitons. Must be an integer number.

`BravaisVectors`: List of Bravais vectors $\mathbf{R}$ that participate in the construction of the Bloch Hamiltonian (1). Expected one per line, in format `x y z`.

`FockMatrices`: Matrices $H(\mathbf{R})$ that construct the Bloch Hamiltonian $H(\mathbf{k})$. The matrices must be fully defined, i.e., they cannot be triangular, since the code does not use hermiticity to generate the Bloch Hamiltonian. The Fock matrices given must follow the ordering given in the block `BravaisVectors`. The matrices can be real or complex, and each one must be separated from the next using the delimiter `&`. In case the matrices are complex, the real and imaginary parts must be separated by a space, and the complex part must carry the imaginary number symbol (e.g. $1.5\ -2.1j$). Both i and j can be used.

`[OverlapMatrices]`: In case that the orbitals used are not orthonormal, one can optionally provide the overlap matrices $S(\mathbf{R})$. The overlap in $\mathbf{k}$ space is given by:

$$S(\mathbf{k}) = \sum_{\mathbf{R}} S(\mathbf{R}) e^{i\mathbf{k}\cdot\mathbf{R}}$$

This is necessary to be able to reproduce the bands, which come from solving the generalized eigenvalue problem $H(\mathbf{k})S(\mathbf{k})\Psi = ES(\mathbf{k})\Psi$. This will be specially necessary if the system was determined using DFT, since in tight-binding we usually assume orthonormality. This block follows the same rules as `FockMatrices`: each matrix $S(\mathbf{R})$ must be separated with the delimiter `&`, and they must follow the order given in `BravaisVectors`.

Several examples of valid system files are provided in the code repository, under the folder `/models`.

4.2 HDF5 files

HDF5 have been introduced as a standardized alternative to the modelfiles. As a hierarchical data format, they are structured in the same way as the modelfiles, namely all the data fields are contained in the root group. The name of each dataset must be the same as those used for the modelfile. Note however that while the modelfiles are not sensitive to upper or lower case, the fields defined in the HDF5 file are. The specific

The main difference comes from the usage of complex numbers, which is not supported by the HDF5 format. To allow complex Hamiltonians (i.e. with spin-orbit coupling), in addition to the fields present in the modelfile one can also define the following dataset:

`[hamiltonian.imag]`: Optional dataset used to specify the imaginary part of the matrices that form the Bloch Hamiltonian. If present, its shape must be equal to that of `[hamiltonian]`.

4.3 Structure of an exciton file

The purpose of the system file was to specify completely the system where we want to compute the excitons. Then, the exciton file is used to describe the excitons themselves: number of points in the mesh or submesh, bands that participate and center-of-mass momentum for example, as well as some additional flags. The idea is to keep the functionality as orthogonal as possible between files. With one system file, we can test for the convergence of the excitons with the number of kpoints, or with the number of bands modifying the exciton file only. Finally, we have the runtime options of the program, which in general do not affect the energy and modify the output exclusively. The philosophy is to maximize the reproducibility and facilitate tracking of the experiments.

The exciton files are built following the rules of the system files. They are composed of blocks, starting with `#`. Each block has a label, which determines the expected content of the block. Next we provide a list of the possible parameters used in the construction of an exciton file:

`Label`: Used to specify the name of the files containing the output of the program. The files will be named `[Label].eigval`, `[Label].kwf`, etc.

`Bands`: Number of bands above and below the Fermi level. The minimum value is 1, to describe one conduction band and one valence band (i.e. only one combination of bands).

`[BandList]`: As an alternative to `Bands`, one can specify a list with the indices of the bands that compose the exciton. 0 is taken as the last valence band, meaning that 1 would be the first conduction band, -1 is the second valence band and so on. This option can be used to generate asymmetric combinations of bands. It overrides the `Bands` block.

`Ncells`: Number of points in one direction of the Brillouin zone, or equivalently number of unit cells along one axis. The same number of points is taken along all directions.

`[Submesh]`: Used to specify a submesh of the Brillouin zone. Takes a positive integer m, which divides the BZ along each axis by that factor. The resulting area is meshed with the number of points specified in the `Ncells` block. This option can become memory intensive (it scales as $\mathcal{O}(m^d)$, d the dimension).

`[ShiftMesh]`: In case that we are using a submesh, then probably we also want to shift the meshed area to center it at the gap, where the exciton peaks. Takes a vector with its components, `kx ky kz`.

`Dielectric`: The Keldysh interaction requires setting the dielectric constants of substrate ϵ_s, the medium ϵ_m and the screening length r_0, which involves the dielectric constant of the material. This block expects three values, `es em r0`.

`[TotalMomentum]`: One can optionally specify the total or center-of-mass momentum $\mathbf{Q}$ of the exciton. By default, it is taken to be zero, unless this block is specified. It expects a vector in form `qx qy qz`.

`[Reciprocal]`: If present, the interaction matrix elements are computed in reciprocal space instead of direct space, which is the default. It takes an integer argument to specify the number of reciprocal cells to sum over, `nG`.

`[Potential]` - New in v1.3.0!: Used to specify the potential function used in the direct term of the kernel of the BSE. Current possible values are `keldysh` or `coulomb`. It defaults to `keldysh` if unspecified.

`[Exchange]`: Flag to turn on the exchange interaction. By default computations neglect the exchange, and use only the direct term. It has to be set to `true` or `false`.

`[Exchange.potential]` - New in v1.3.0!: Used to specify the potential function used in the exchange term of the kernel of the BSE. Current possible values are `keldysh` or `coulomb`. It defaults to `keldysh` if unspecified. Note that is used only if `[Exchange]` was set to `true`.

`[Scissor]`: Used to specify a scissor shift of the bands to correct the gap. This optional field takes a single value, `shift`

`[Regularization]`: Keyword used to set the regularization distance used in the real-space method to avoid the electrostatic divergence at $\mathbf{r} = 0$ by setting $V(0) = V(a)$, where a is the regularization distance. By default this parameter is set to the unit cell lattice parameter. It is advised to be changed only for supercell calculations.

As it can be seen, a minimum exciton simulation only requires specifying the number of bands, the number of $\mathbf{k}$ points and the dielectric constants. The modification of any of the present parameters is expected to result in a variation of the exciton results (energies, wavefunctions, conductivity), which is why all this parameters have been delegated to the same file.

4.4 Absorption file

For the calculation of the Kubo conductivity, one needs to provide a separate input file named `kubo_w.in` in the working folder. This file is used to specify all parameters relative to the conductivity calculation, namely the desired energy interval, the point sampling and the broadening to be used, as well as the output files. Its format is as follows:

```
#initial frequency (eV)
5
#frequency range (eV)
8
#number of frequency points (integer)
300
#broadening parameter (eV)
0.05
#type of broadening
lorentzian
#output kubo name files
kubo_hBN_sp.dat
kubo_hBN_ex.dat
```

Do note that as opposed to the previous configuration files, the name of each section starting with # is not actually relevant for the parsing; the program always expects the same fields to be present

in the file, and in the same order as presented here. For the broadening, three different options are allowed (`'lorentzian', 'exponential', 'gaussian'`)

5 OUTPUT

To conclude the usage section, we will describe briefly the structure of the output files. Here we describe how each file is written so the user can write their own custom routines; we also provide some example Python scripts under the folder `/plot`.

- Energy: The energy file has in the first line the total number of energies written in the file. The second line contains all the energies, separated by a tabulation, `e1 e2 ... en`. All energies are written, including degenerate levels. All the exciton energies are given with respect to the Fermi sea energy. To obtain the binding energy, one must substract the gap from the exciton energy. Units are $[E] = \text{eV}$.

- States: The first line contains the dimension of the BSE matrix n, i.e. the number of different electron-hole pairs. The next n lines specify the valence, conduction bands of each electron-hole pair and their **k** point, `kx ky kz v c`. Afterwards, each line specifies completely the coefficients of each exciton state. The format per line is: `Re(A1) Im(A1) Re(A2) Im(A2) ...`.

- Reciprocal probability density: For the reciprocal density, on each line we specify the coordinates of the k point and the associated probability, `kx ky kz P`. Each state is separated from the next by a delimiter `#`. Units are $[k] = \text{Å}^{-1}$.

- Real-space probability density: The first line has the coordinates of the hole, `hx hy hz`. The following ones each have the coordinates of one atomic position, and the probability of finding the electron: `x y z P`. Densities for different states are separated by `#`. Units are $[x] = \text{Å}$.

- Spin: On each line we write the index of the current exciton, and next the total spin projection, the hole and the electron spin, `n St Sh Se`. Spin units are $[S_z] = \hbar$.

- Absorption: Both conductivities with and without exciton effects are computed and written to two different files ("hBN_sp.dat" and "hBN_ex.dat" for instance). Each row contains the following columns: ω, σ_{xx}, σ_{xy}, σ_{xz}, σ_{yx}, σ_{yy}, σ_{yz}, σ_{zx}, σ_{zy}, σ_{zz}. Units are $[\omega] = \text{eV}$, $[\sigma_{ij}] = e^2/\hbar$. Additionally, the file with optical conductivity for the excitons also incorporates all the exciton oscillator strengths $V^\alpha = \sum_{v,c,\mathbf{k}} A_{vc}(\mathbf{k}) v^\alpha_{vc}(\mathbf{k}), \forall \alpha \in \{x, y, z\}$. This block follows after the conductivities by a blank line, with the following format: `E_X Re(Vx) Im(Vx) Re(Vy) Im(Vy) Re(Vz) Im(Vz)`, each line for each exciton of the spectrum.