Documentation

Authors: Simão João, Hanwen Jin, Johannes Lischner

February 23, 2023

Contents

1 Introduction and overview

Metallic nanoparticles have a very special interaction with light. When light shines on a metal, conduction electrons oscillate back and forth driven by the electric field. This non-equilibrium configuration excites electrons into more energetic unoccupied states creating a very energetic electron-hole pair. These carriers can be very energetic (hot carriers) and can be harnessed for many applications. Plasmonic photocatalysis is one such example. This document contains the documentation of our hot carrier generation code, as well as an explanation of the methods involved and physical interpretation of the objects being used.

In a nutshell, the workflow is divided into four section:

1. Defining the atomic positions
2. Finding the Hamiltonian matrix
3. Finding the electric potential
4. Hot carrier generation rate / distribution, density of states or bandstructure

Not all steps are required for each functionality. All of these points are discussed in more detail in the following sections.

2 Mathematical structure

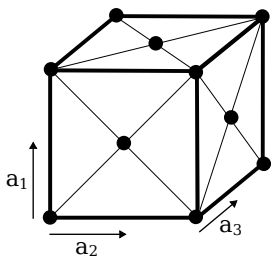
In order to understand these functions, it is important to understand the underlying structure common to all of them.

2.1 Lattice structure

The starting point is the definition of the atomic positions. Internally, these are not specified in terms of cartesian coordinates, but in terms of lattice coordinates. The FCC lattice points are defined in terms of the vectors $\mathbf{a}_i$:

$$\mathbf{R}_n = x_n\mathbf{a}_1 + y_n\mathbf{a}_2 + z_n\mathbf{a}_3$$

and so each point $\mathbf{R}_n$ is represented by the set of three numbers $(x_n, y_n, z_n)_\mathbf{a}$ (where the subscript $\mathbf{a}$ denotes that this set of numbers is a vector in the basis $\mathbf{a}_i$). With this parameterization, the position of every point in the lattice can be described entirely with integer numbers, which makes computations simpler. The lattice is composed of all sets of points such that $x_n + y_n + z_n$ is an even number. For each of these points, the twelve nearest neighbours are defined as $\mathbf{R}_n + \mathbf{d}$ where $\mathbf{d} = (\pm1, \pm1, 0)_\mathbf{a}$, $(\pm1, 0, \pm1)_\mathbf{a}$ or $(0, \pm1, \pm1)_\mathbf{a}$. The six second nearest neighbours are defined with $\mathbf{d} = (\pm2, 0, 0)_\mathbf{a}$, $\mathbf{d} = (0, \pm2, 0)_\mathbf{a}$ and $\mathbf{d} = (0, 0, \pm2)_\mathbf{a}$. The volume of the unit cell is given by $V_c = 8\mathbf{a}_1 \times \mathbf{a}_2 \cdot \mathbf{a}_3 = 8\left|\mathbf{a}_1\right|^3$.



FCC

2.2 Defining the shape

In order to build atomistic nanoparticles of a certain shape, the code first defines the shape and then fills it with atoms following the underlying lattice structure above. Two categories of shapes are supported right now: spheres and polyhedra. The sphere is defined by its radius R, such that $|\mathbf{R}_n| < R$ for every atomic position $\mathbf{R}_n$.

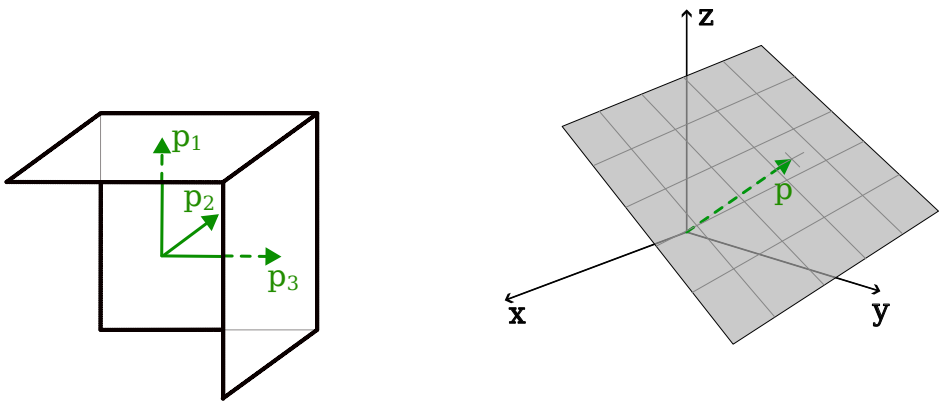


Figure 1: Left: set of three planes defining some boundaries of a shape. Right: detail of a plane defined from a vector.

The polyhedra are defined in terms of planes, by specifying the plane's position and which side of the plane should be considered the inside. Let $\mathbf{p}$ be a vector orthogonal to the plane and passing through the origin (see fig 1). The plane is defined through the equation $\mathbf{r} \cdot \mathbf{p} = |\mathbf{p}|^2$ for every position $\mathbf{r}$. This vector $\mathbf{p}$ uniquely defines any plane in space (except the planes containing the coordinate axis). Let's now assume that the origin is always on the inside of the nanoparticle. Let $\mathbf{r}$ be an arbitrary vector in space. Then $\mathbf{r}$ can be expressed as $\mathbf{r} = \alpha\mathbf{p} + \mathbf{q}$ where $\mathbf{q}$ is a vector orthogonal to $\mathbf{p}$. The vector $\mathbf{r}$ is in the correct side of the plane if $\alpha < 1$, that is, when

$$\mathbf{r} \cdot \mathbf{p} < \mathbf{p} \cdot \mathbf{p} \tag{1}$$

Since the polyhedron is composed of planes, $\mathbf{r}$ is considered to be inside it if it is in the correct side of all the planes that compose it.

Currently, three shapes are supported: octahedra, cubes and rhombic dodecahedra. The are particularly interesting shapes because their planes fit nicely with the FCC crystal planes.

- The cube is defined by the set of six planes $(\pm c, 0, 0)_{\mathbf{a}}$, $(0, \pm c, 0)_{\mathbf{a}}$ and $(0, 0, \pm c)_{\mathbf{a}}$. The variable c parametrizes the size of the cube and is equal to half its side length ℓ (in units of $\mathbf{a}_i$). Therefore, the cube's real side length is $2c\,|\mathbf{a}_1|$ and its volume is $8V_c c^3$. If c is an odd integer, then all the cube's vertices will contain an FCC lattice point.

- The octahedron is defined by the set of eight planes $(\pm c, \pm c, \pm c)_{\mathbf{a}}$. The octahedron is contained in a cube of side ℓ such that its vertices are at the center of the cube's faces. Setting $x = y = 0$ and $z = \ell/2$ in the plane equation for $(c, c, c)_{\mathbf{a}}$ we get the relation between the plane parameter c and the cube's side ℓ: $\ell = 6c$. The vertices are placed in positions $(\pm\ell/2, 0, 0)_{\mathbf{a}}$, $(0, \pm\ell/2, 0)_{\mathbf{a}}$ and $(0, 0, \pm\ell/2)_{\mathbf{a}}$, so they contain an FCC lattice point if $\ell/2$ is odd. The octahedron's edges have length $3\sqrt{2}c\,|\mathbf{a}_1|$ and its volume is $V = 108V_c c^3$. As c is increased, new atomic planes are introduced when c is an odd integer.

- The rhombic dodecahedron is defined by the set of twelve planes $(\pm c, \pm c, 0)_{\mathbf{a}}$, $(\pm c, 0, \pm c)_{\mathbf{a}}$ and $(0, \pm c, \pm c)_{\mathbf{a}}$. It is constructed by starting with a cube of side length ℓ and adding a square pyramid of height $\ell/2$ to each of its faces. Setting $x = y = 0$ and $z = 3\ell/2$ in the plane $(c, 0, c)$, we find $\ell = 4c/3$. Each edge has length $\left(2/\sqrt{3}\right) c\,|\mathbf{a}_1|$ and its volume is $2\ell^3 = (128/27)\,V_c c^3$. The vertices connecting three facets are located at $(\pm\ell/2, \pm\ell/2, \pm\ell/2)$ and the vertices connecting four facets are located at $(\pm\ell, 0, 0)$, $(0, \pm\ell, 0)$ and $(0, 0, \pm\ell)$. It is not possible for all the vertices to contain FCC lattice points

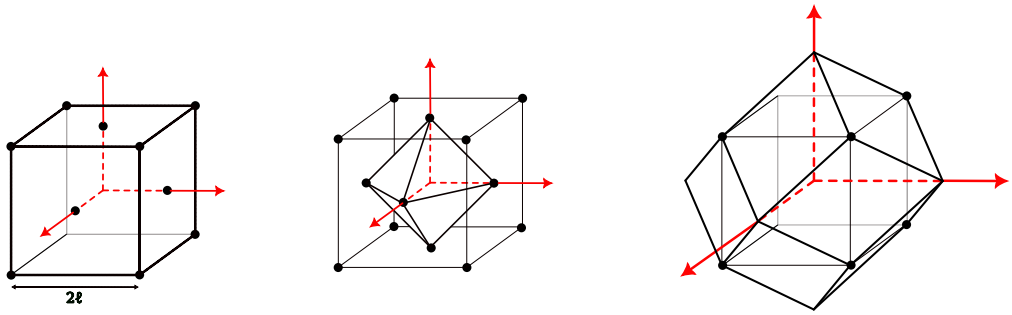


Figure 2: Nanoparticle shapes compatible with the FCC lattice structure: cube, octahedron and rhombic dodecahedron.

2.3 Filling the shape with atoms

The shapes in the previous section define the boundaries of the nanoparticle but now we need to fill it with atoms. To do that, we begin with an atom which we know to be inside of the nanoparticle and recursively find the neighbours using the underlying lattice structure. If the neighbour is inside the nanoparticle, it is added to the list of atoms. This is done with the `explore` algorithm in the `shape_lib.jl` library.

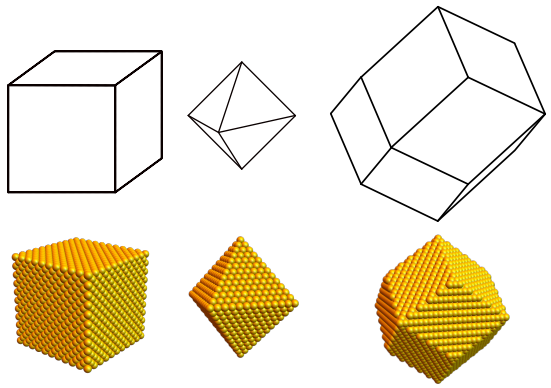


Figure 3:

3 Finding the Hamiltonian

Once the set of atomic positions have all been found, the tight-binding Hamiltonian between them is obtained via a Slater-Koster parametrization. This gives rise to a sparse matrix but assumes a perfectly regular lattice. More complicated lattices give rise to more complicated tight-binding Hamiltonians, which can be imported explicitly from a file instead of relying on a Slater-Koster parametrization.

4 Finding the electric potential

The electrons inside the nanoparticle feel not only the external electric field coming directly from the light source, but also the induced field arising from the polarizable material.

4.1 Quasi-static approximation

The nanoparticle is assumed to be much smaller than the wavelength of the incoming electric field, warranting the use of the quasi-static approximation (QSA). Under the QSA, the incident electric field is assumed to be uniform across the nanoparticle but parametrized by the frequency ω, which changes the dielectric constant of the medium.

4.2 Expression for the dielectric sphere

Simple shapes like the sphere have a known exact expression for the electric field inside the nanoparticle (in Fourier space)

$$E_{\text{tot}}(\omega) = \frac{3\varepsilon_m}{\varepsilon(\omega) + 2\varepsilon_m} E_0$$

where ε_m is the dielectric constant of the surrounding medium and E_0 is the incident electric field strength. From this, the electric potential energy can easily be found

$$\Phi_{\text{tot}}(\omega) = \frac{3\varepsilon_m}{\varepsilon(\omega) + 2\varepsilon_m} (-e) E_0 z$$

More complicated shapes like the octahedron require a numerical solution of Maxwell's equations and this is done using COMSOL (see our COMSOL tutorial).

5 Fermi Golden Rule

Once the Hamiltonian and the potential have been found, we have all the ingredients to compute the hot carrier generation rate using Fermi's Golden Rule (FGR). FGR provides the rate of transitions to states with energy E due to a periodic perturbation:

$$N_e(E, \omega) = \frac{2}{V} \sum_{if} \Gamma_{if}(\omega) \delta(E - E_f)$$

where

$$\Gamma_{if} = \frac{2\pi}{\hbar} \left| \langle f | \phi_{\text{tot}}(\omega) | i \rangle \right|^2 \delta(E_f - E_i - \hbar\omega) f(E_i) (1 - f(E_f)).$$

$N_e(E, \omega) dE$ is interpreted as the number of transitions per unit time with energy between E and $E + dE$ due to a perturbation of frequency ω. The interpretation of each of the terms is the following:

1. The electric field does not affect every pair of states in the same manner. The number $\langle f | \phi_{\text{tot}}(\omega) | i \rangle$ codifies this effect.
2. The product of Fermi functions $f(E_i)(1 - f(E_f))$ ensures that only initial states below the Fermi energy and final states above the Fermi energy contribute to this integral. A finite temperature relaxes this restriction slightly.
3. The second Dirac delta $\delta(E_f - E_i - \hbar\omega)$ is a consequence of the interaction with the electric field of frequency ω. The electric field only causes transitions which change the energy by $\hbar\omega$. In practice, there are many mechanisms going on inside the nanoparticle, which slightly violate this restriction, so the Dirac delta is instead approximated by a gaussian of a certain width γ, denoted by δ_γ
4. The first Dirac delta $\delta(E - E_f)$ means that only final states which have energy E are to be considered. Like before, this Dirac delta can also be approximated by a gaussian of a certain (different) width λ, representing the limited resolution of the measuring apparatus. If this is ignored, we have infinite resolution, which gives rise to sharp peaks.

For reference, the electric field E_{ref} used in experiments is such that the illumination intensity is $I = 1\text{mW}/\mu\text{m}^2$, so $E_{\text{ref}} = 8.7 \times 10^5 \text{Volt/m} = 8.7 \times 10^{-4} \text{Volt/nm}$. For comparison, the average solar intensity on Earth's surface is 1360W/m^2, which is about 10^6 times weaker!

Calculation of the Fermi Golden Rule

In practice, for large nanoparticles, the form used for the Fermi golden rule is the following:

$$N_e(E,\omega) = \frac{4\pi}{\hbar V}(1 - f(E)) \int_{-\infty}^{\infty} d\varepsilon f(\varepsilon) \delta_\gamma(E - \varepsilon - \hbar\omega) \Phi_\omega(\varepsilon, E) \tag{2}$$

where

$$\Phi_\omega(\varepsilon, \varepsilon') = \text{Tr}\left[\phi_{\text{tot}}(\omega) \delta(\varepsilon - H) \phi_{\text{tot}}^\dagger(\omega) \delta(\varepsilon' - H)\right]$$

represents the optical matrix in energy space rather than the eigenbasis of the Hamiltonian. It is calculated by decomposing it into a double Chebyshev expansion

$$\Phi_\omega(\varepsilon, \varepsilon') = \sum_{n=0}^{M-1} \sum_{m=0}^{M-1} \Delta_n(\varepsilon) \Delta_m(\varepsilon') \mu_{nm} \tag{3}$$

where

$$\mu_{nm} = \text{Tr}\left[\phi_{\text{tot}}(\omega) T_n(H) \phi_{\text{tot}}^\dagger(\omega) T_m(H)\right] \tag{4}$$

is the matrix of Chebyshev moments. $T_n(H)$ is the n-th Chebyshev polynomial with a matrix argument and $\Delta_n(\varepsilon)$ is the Chebyshev coefficient defined as

$$\Delta_n(\varepsilon) = \frac{1}{\pi\sqrt{1-\varepsilon^2}} \frac{T_n(\varepsilon)}{1 + \delta_{n,0}} g_{n,J}^M$$

The last term $g_{n,J}^M$ doesn't come from the Chebyshev expansion, but from the Kernel Polynomial Method. It is the weight associated with the Jackson kernel and is useful to get rid of the Gibbs oscillations phenomenon.

6 Density of states

The density of states is defined as the histogram of energy levels:

$$\rho(E) = \frac{1}{N} \sum_i \delta(E - E_i)$$

Just like for the Fermi golden rule, the density of states of large nanoparticles can be calculated using the Kernel polynomial method, by first expressing it as a trace

$$\rho(E) = \frac{1}{N} \text{Tr}\left[\delta(E - H)\right]$$

and then expanding the Dirac delta operator as a series of Chebyshev polynomials

$$\rho(E) = \frac{1}{N} \sum_{n=0}^{M-1} \Delta_n(E) \mu_n$$

where $\Delta_n(E)$ is defined just like before and

$$\mu_n = \text{Tr}\left[T_n(H)\right]$$

are the Chebyshev moments.

7 Band structure

The band structure is calculated along the path $\Gamma \to X \to W \to L \to \Gamma \to K$ in the FCC Brillouin zone

8 How to use the code

We have two main ways of interfacing with the code:

1. **High-level interface** give the user a lot flexibility in defining the nanoparticle geometries, materials and calculation parameters, but automatically take care of the little calculation details. They give the user no access to the internal objects being calculated. This interface is the file `prompt.jl`.

2. **Low-level interfaces** provide a direct connection to the libraries used to run the algorithms. They require detailed knowledge of the methods being used, but give the user complete control over the simulation, including access to internal objects like the atomic positions and Hamiltonian. There are four low level interfaces, one for each functionality of the code: hot carrier generation (`hcg_ll.jl`), distribution (`dist_ll.jl`), density of states (`dos_ll.jl`) and band structure (`bands_ll.jl`)

8.1 High-level interface

This approach has a direct correspondence with the workflow specified in the introduction. The atomic position generation and Hamiltonian creation are bundled into one single step, via the function `hamiltonian_builder_predefined`. This function allows the user to select from four different shapes (cube, octahedron, rhombic dodecahedron and sphere) according to the prescription detailed earlier. `l_nm` is the edge length in the case of polyhedra or the radius in the case of the sphere. It is expected to be in nanometers. Selecting the material automatically defines the bulk dielectric function, the Slater-Koster parameters, the Fermi energy and the KPM rescaling factors.

```
1 shape = "cube"
2 mater = "gold"
3 l_nm  = 1.5 # [nm] dimension of nanoparticle
4 hb    = hamiltonian_builder_predefined(shape, l_nm, mater)
```

The next step of defining the electric potential is required for the hot carrier generation and population distribution functionalities. The frequency has to be specified, but the nanoparticle's dielectric function and the environment dielectric function can be ommitted. If the environment's dielectric function is not specified, the default value of 1 is used. If the nanoparticle's dielectric function is not specified, the function specified in the material file will be used.

```
1 freq_eV = 2.4 # [eV] frequency of optical potential
2 eps_m   = 2.0 # relative dielectric constant of the environment (default 1)
3 eps     = -200+3im # relative dielectric constant of the nanoparticle (default prebuild)
4
5 pb = potential_builder_sphere(freq_eV, eps=eps, eps_m=eps_m)
```

Alternatively, the user can tell the program to import the potential from a file using the COMSOL format.

```
1 freq_eV = 2.4 # [eV] frequency of optical potential
2 filename = "pot.dat"
3 pb = potential_builder_import(freq_eV, filename)
```

The hot carrier generation rate can be requested by calling `calculation_builder_hcg`. This function can be called without any arguments, in which case the default values in the following code comments will be used. If the convergence parameter is set to true, then convergence tests will be printed out.

```
T     = 298.0 # [K] temperature (default 298 K)
gph   = 0.01  # [eV] lifetime (default 10meV)
conv  = true  # include convergence data in the output (default false)
N     = 200   # number of Chebyshev polynomials (default 200)
NR    = 5     # number of random vectors (default 5)
NB    = 10    # number of Chebyshev polynomial blocks (default 10)
NE    = 1000  # number of output energies (default 1000)
hname = "hcg.dat" # hot carrier generation output filename (default hcg.dat)

gb = calculation_builder_hcg(temp=T, gph=gph, N=N, NR=NR,
                         NB=NB, NE=NE, conv=conv, name=hname)
```

The population distribution can be requested by calling `calculation_builder_dist`. This function can be called without any arguments, in which case the default values specified in the following code comments will be used. Relaxation rates $\Gamma(\varepsilon, \varepsilon')$ will by default be $\Gamma(\varepsilon, \varepsilon') = 1\text{eV}$.

```
T     = 298.0 # [K] temperature (default 298 K)
conv  = true  # include convergence data in the output (default false)
N     = 200   # number of Chebyshev polynomials (default 200)
NR    = 5     # number of random vectors (default 5)
NB    = 10    # number of Chebyshev polynomial blocks (default 10)
NE    = 1000  # number of output energies (default 1000)
dname = "dist.dat" # hot carrier generation output filename (default dist.dat)

# Relaxation rates in eV
function Gamma(E,Ep)
    return 0.1
end

db = calculation_builder_dist(temp=T, Gamma=Gamma, N=N, NR=NR,
                         NB=NB, NE=NE, conv=conv, name=dname)
```

The density of states can be requested by calling `calculation_builder_dos`. This function can be called without any arguments, in which case the following defaults will be used. If the minimum or maximum energies specified are beyond the bounds of the density of states, they will automatically be redefined.

```
N  = 200        # number of Chebyshev polynomials (default 200)
NK = 4          # number of random vectors (default 5)
NE = 1000       # number of energy points (default 1000)
minE_Ha = -1.0 # minimum energy in Hartree (default -2)
maxE_Ha = +1.0 # maximum energy in Hartree (default +2)
dosname = "dos.dat" # output file name (default "dos.dat")

dosb = calculation_builder_dos(N=N, NE=NE, minE_Ha=minE_Ha,
                         maxE_Ha=maxE_Ha, NK=NK, name=dosname)
```

The band structure can be requested by calling `calculation_builder_bands`.

```
bandname = "bands.dat" # output file name (default "bands.dat")
bb = calculation_builder_bands(name=bandname)
```

After the builders have been set up, the calculation is performed by calling the `nanoparticle_run` function, which takes as argument all the builders.

```
nanoparticle_run(ham_builder=hb, pot_builder=pb, hcg_builder=gb,
                         dist_builder=db, dos_builder=dosb, band_builder=bb)
```

8.2 Advanced usage: low-level interfaces

These are examples on how to use the low-level interfaces for each of the functionalities.

Band structure `interfaces/bands_ll.jl`

The band structure is the simplest object to calculate. It only requires knowledge of the material. The tight-binding parameters are found with the `tightbinding` method and fed into the `get_bands` method which calculates the band structure along a predefined path. Finally, the band structure is saved into a file.

```
mater = "gold"
onsite, first_neighbour, second_neighbour, A, B, fermi_Ha, a0, diel = tightbinding(mater)
bands = get_bands(onsite, first_neighbour, second_neighbour)

open("bands.dat", "w") do io
        writedlm(io, bands)
end
```

Density of states `interfaces/dos_ll.jl`

The density of states requires some more input. First, the material has to be specified in order to provide the tight-binding parameters

```
mater = "gold"
onsite, first_neighbour, second_neighbour, A, B, fermi_Ha, a0, diel = tightbinding(mater)
```

Then, the geometry of the nanoparticle has to be specified. Using "cube", "octahedron", "rhombic" or "sphere" produces the shapes just as described before. The Hamiltonian for that shape is then generated as normal.

```
shape = "cube"
rad = 5.1 # [nm]
Elist, Edict, R = generate_shape_FCC(rad, shape, a0)
H = slater_koster_FCC(Elist, Edict, onsite, first_neighbour, second_neighbour, A, B)
```

If the "periodic" option is used instead, then some more options have to be specified. `L` is the number of crystal planes to consider in the periodic lattice (see fig. 4), `rad` is the maximum length (in nanometers) to keep adding atoms to the lattice. The `slater_koster_FCC` function needs four additional parameters to deal with the periodic lattice. The `periodic` flag has to be set to `true`, and the number of crystal planes has to be specified for each direction (`L1`, `L2`, `L3`)

```
shape = "periodic"
L = 16
rad = (L-1+0.01)*a0/2 # [nm]
Elist, Edict, R = generate_shape_FCC(rad, shape, a0)

H = slater_koster_FCC(Elist, Edict, onsite, first_neighbour,
            second_neighbour, A, B, L1=L, L2=L, L3=L, periodic=true)
```

After the Hamiltonian has been specified, the calculation proceeds by first computing the Chebyshev moments and then resumming the series to obtain the density of states. The Chebyshev moment computation requires specifying the number of moments `N`, the number of random vectors `Nk` and a flag specifying the kind of vectors to be used. For most situations, random vectors will be the desired option, which requires `flag=2`. The series resummation requires the percentage of Chebyshev polynomials to keep `perc` (any number smaller than 100% is mostly useful for convergence analysis purposes), the bounds (in Hartree) within which to evaluate the density of states

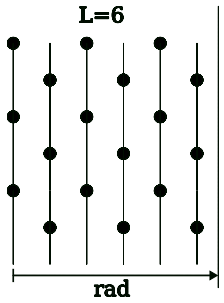


Figure 4: Number of lattice planes L and its relation to `rad`.

`minE_Ha` and `maxE_Ha` and the total number of points `NE` within those bounds. The bounds are automatically adjusted if they fall tto far from the actual DoS limits. Finally, the DoS is saved to a file.

```
N  = 300  # number of total Chebyshev polynomials
Nk = 2    # number of random vectors
flag = 2  # flag=2 means random vectors

mu  = doscompute_mu!(H, N, Nk, flag)

perc = 100   # percentage of Chebyshev moments to keep
minE_Ha = -1 # smallest energy
maxE_Ha = +1 # largest energy
NE = 1000    # number of energies

dos = get_dos(mu, perc, minE_Ha, maxE_Ha, NE, A, B)

open("dos.dat", "w") do io
        writedlm(io, dos_periodic)
end
```

Hot carrier generation rate `interfaces/hcg_ll.jl`

The initial steps for the hot carrier generation rate calculation are identical to the ones for the density of states. First specify the material and the geometry to get the Hamiltonian.

```
mater = "gold"
onsite, first_neighbour, second_neighbour, A, B, fermi_Ha, a0, diel = tightbinding(mater)


shape = "cube"
rad   = 1.5 # [nm] dimension of nanoparticle
Elist, Edict, R = generate_shape_FCC(rad, shape, a0)

# Use list of atomic positions to determine the Hamiltonian. H is in KPM units
H = slater_koster_FCC(Elist, Edict, onsite, first_neighbour, second_neighbour, A, B)
```

If the user wants to export the atomic positions for visualization purposes, or for use with COMSOL, that can be done with resort to the `print_positions_comsol` function, which prints out the atomic positions in a format that is understood by COMSOL.

```
filename = "positions.dat"
print_positions_comsol(R, filename)
```

After that, the potential in Fermi's golden rule has to be specified. The function potential_sphere assumes the nanoparticle has a spherical shape. In this case, there is an exact analytical expression for the potential and it is evaluated at each lattice site. This requires specifying the dielectric constant `eps` of the nanoparticle, which is obtained from the `tightbinding` function and is a function of the frequency `freq`, and the dielectric function of the environment `eps_m`.

```
freq  = 2.4 # [eV] frequency of optical potential
eps_m = 1.0 # relative dielectric constant of the environment
eps = diel(freq)

# get the potential. Phi is in units of eV
Phi = potential_sphere(R, eps, eps_m)
```

Alternatively, the potential can also be read directly from a file, using the COMSOL format.

```
filename = "pot.dat"
Phi = comsol_read(filename)
```

Once the Hamiltonian and the electric potential are known, they are used to get the matrix of Chebyshev moments required to get the hot carrier generation rate:

$$\mu_{nm} = \mathrm{Tr}\left[\phi_{\mathrm{tot}}\left(\omega\right) T_n\left(H\right) \phi_{\mathrm{tot}}^{\dagger}\left(\omega\right) T_m\left(H\right)\right]$$

N is the number of Chebyshev moments used for each Dirac delta operator. Since there are two operators, N^2 total moments will be required. For efficiency purposes, this Chebyshev moment matrix is calculated in blocks of size `(NTx,NTy)`. Larger block sizes use more memory, but decrease the computational time.

```
N = 200 # number of total Chebyshev polynomials

# number of Chebyshev polynomials per block
NTx = 10
NTy = 10

# number of blocks
NNL = trunc(Int,N/NTx) + 1
NNR = trunc(Int,N/NTy) + 1
Nk  = 5 # number of random vectors

# Build the Chebyshev matrix
mumn = compute_mumn!(H, Phi, NNL, NNR, NTx, NTy, Nk)
```

The next step is to resum the Chebyshev matrix into the $\Phi_\omega\left(\varepsilon, \varepsilon'\right)$ function which contains information about the total number of optical transitions between energy ε and ε'

$$\Phi_\omega\left(\varepsilon, \varepsilon'\right) = \sum_{n=0}^{M-1} \sum_{m=0}^{M-1} \Delta_n\left(\varepsilon\right) \Delta_m\left(\varepsilon'\right) \mu_{nm}$$

This function is estimated in a grid of `Nx` $\times$ `Ny` energy points, spanning the whole spectrum (automatically defined). The set of energies for the first (second) entry are `Ei` (`Ej`) and the Φ_ω function is saved in the `opt` matrix. The more polynomials that are used, the more energy points should be used as well. As a rule of thumb, it is useful to use twice as many energy points as Chebyshev polynomials. Just like before, `perc1` and `perc2` control the amount of Chebyshev polynomials to keep in the calculation and are mostly useful for convergence analysis purposes.

```
1
2  # Number of integration points in the energy discretization
3  Nx = N*2
4  Ny = Nx
5
6  # Percentage of Chebyshev polynomials to keep
7  perc1 = 100
8  perc2 = perc1
9
10
11 # Transform the Chebyshev matrix into the energy-resolved optical matrix
12 # Ei,Ej are in eV, opt is dimensionless
13 Ei,Ej,opt = resum_mu_dd(mumn, Nx, Ny, A, B, perc1, perc2)
```

Next, the optical matrix Φ_ω is integrated into the hot carrier generation rate,

$$N_e(E,\omega) = \frac{4\pi}{\hbar V}(1 - f(E)) \int_{-\infty}^{\infty} d\varepsilon f(\varepsilon)\, \delta_\gamma(E - \varepsilon - \hbar\omega)\, \Phi_\omega(\varepsilon, E)$$

which requires knowledge of the temperature `tempr` (in Kelvin), broadening parameters (`kappa`, `gph`) and number of points to use in the integration (`NEp`) and final result (`NE`). These are not necessarily the same as `Nx` and `Ny` because an interpolation is used to get a better sampling for the integration step. Additional debug information can be retreived by setting `write` to `true`. The Fermi energy (`fermi_Ha`) and frequency `freq` are also required for this calculation and are both obtained earlier in the code.

```
1  NE  = N*2+1 # Number of energy points in the integration
2  NEp = N*2+1 # Number of output energy points
3
4  tempr = 298 # [K] temperature
5
6  # hot carrier generation parameters
7  kappa = 0.0
8  gph   = 0.06 # [eV] broadening
9
10 # Temperature
11 kbT  = tempr*boltzmann # eV
12 beta = 1.0/kbT
13
14 write = false # write debug information
15
16 # Get the hot carrier generation rate
17 hcg = sum_hcg( opt, Ei, Ej, NE, NEp, fermi_Ha, freq, beta, kappa, gph, write)
```

The hot carrier generation rate is then saved to a file. The first column is the set of energies at which the hot carrier generation rate is being calculated, the second column is the rate of hot electron generation and the third column is the rate of hot hole generation.

```
1  # Write the data
2  columns = Array{Float64}(undef, NE, 3)
3  columns[:,1] = hcg[1]  # energy list
4  columns[:,2] = hcg[2]  # hot electron
5  columns[:,3] = hcg[3]  # hot hole
6
7  # Write the data
8  writedlm("hcg.dat", columns)
```

Alternatively, the user can skip the optical matrix calculation and integration by using the hcg_conv function, which does it automatically while generating additional data for convergence analysis. This function does not require specification of the percentage and number of energy points used in the integration, since they are estimated automatically and checked for convergence.

```
print_conv = true
hcg = hcg_conv(mumn, A, B, NE, fermi_Ha, freq, beta, kappa, gph, print_conv)
```

Population distribution `interfaces/dist_ll.jl`

Mathematically, the calculation of the population distribution is very similar to the hot carrier generation, and uses many of the same objects. Everything is the same up to and including the resummation of the Chebyshev moments matrix into the Φ_ω matrix. Φ_ω is then used to get the distribution of electrons by integrating it with a relaxation rate function $\Gamma(\varepsilon, \varepsilon')$

$$N_e(E) = -4 \int d\varepsilon \int d\varepsilon' \Phi_\omega(\varepsilon, \varepsilon') \, \delta(E - \varepsilon) \, \frac{f(\varepsilon) - f(\varepsilon')}{\Gamma(\varepsilon, \varepsilon)} \left[\frac{\Gamma(\varepsilon, \varepsilon')}{(\hbar\omega - \varepsilon' + \varepsilon)^2 + \Gamma^2(\varepsilon, \varepsilon')} + \frac{\Gamma(\varepsilon, \varepsilon')}{(-\hbar\omega - \varepsilon' + \varepsilon)^2 + \Gamma^2(\varepsilon, \varepsilon')} \right]$$

which needs to be specified. This function can be both a function of ε and ε', but for simplicity here we set it to a constant. `Gamma` is the implementation of this function and `Gammaf` is a prefactor to `Gamma` which is used for convergence analysis. For most cases, it can be simply set to one.

```
Gammaf = 1

# units: eV
function Gamma(E,Ep)
        return 0.1
end

dist = sum_dist(opt, Ei, Ej, NE, NEp, fermi_Ha, freq, beta, Gammaf, Gamma, write)
```

The output is a an array of two columns, where the first column is the list of energies and the second is the population distribution for each energy.

```
columns = Array{Float64}(undef, NE, 2)
columns[:,1] = dist[1] # energy list
columns[:,2] = dist[2] # population

# Write the data
writedlm("dist.dat", columns)
```

9 Code documentation

The code under src is written in the Julia programming language and offers both high-level and low-level interfaces to the user. It is organized into four main categories:

- Materials library (src/materials)
- Algorithms library (src/libs)
- External visualization tools (src/vis_tools)
- Interfaces - low and high level (src/interfaces)

Next, we look at each of these categories in more detail.

9.1 Materials library

The materials library contains all the parameters required for simulating the most common materials. At this moment, we have an implementation of gold and silver. Each material has its own separate file, containing the Slater-Koster parameters, the dielectric constant as a function of frequency, the Fermi energy and the KPM rescaling parameters.

9.2 Algorithms library

The algorithms library contains all the low-level implementations of the algorithms used throughout the code. This includes the Chebyshev recursion scheme for the hot carrier generation rate/distribution and density of states, the code to generate the band structure, etc.

Slater-Koster tight-binding parametrization `slater_koster.jl`

Contains the implementation of the Slater-Koster tight-binding parametrization for FCC lattices

`slater_koster_FCC(Elist, Edict, onsite, first_neighbour, second_neighbour, A, B)`

Elist is the list of all the positions of atoms inside the nanoparticle Elist[n] returns a 3D vector containing the cartesian coorinates

9.3 External visualization tools

The external visualization tools folder contains several small program written in Python to help visualize the atomic positions, the optical matrix, etc.

Part I
Technical information

10 Transforming the Fermi Golden Rule

Since diagonalization is very hard to do for large matrices, we need to express N_e in a different way in order to use a different set of methods. Let's begin by introducing two integrals over the energy in order to capture all the eigen-energies with Dirac deltas:

$$N_e(E,\omega) = \frac{2}{V}\sum_{if}\int_{-\infty}^{\infty} d\varepsilon \int_{-\infty}^{\infty} d\varepsilon' \delta(\varepsilon - E_i)\,\delta(\varepsilon' - E_f)\,\Gamma_{if}(\omega)\,\delta(E - E_f)$$

Now we just need to replace the eigen-energies:

$$N_e(E,\omega) = \frac{4\pi}{\hbar V}\int_{-\infty}^{\infty} d\varepsilon \int_{-\infty}^{\infty} d\varepsilon' f(\varepsilon)\,(1 - f(\varepsilon'))\,\delta(E - \varepsilon')\,\delta(\varepsilon' - \varepsilon - \hbar\omega)\sum_{if}\left|\langle f\left|\Phi_{\text{tot}}(\omega)\right|i\rangle\right|^2 \delta(\varepsilon - E_i)\,\delta(\varepsilon' - E_f)$$

which motivates us to define the new object

$$\Phi_\omega(\varepsilon,\varepsilon') = \sum_{if}\left|\langle f\left|\Phi_{\text{tot}}(\omega)\right|i\rangle\right|^2 \delta(\varepsilon - E_i)\,\delta(\varepsilon' - E_f).$$

This object is a reinterpretation of the potential matrix element in energy space. Instead of telling us the matrix element of the initial state i with final state f, it tells us the matrix element of initial state of energy ε with final state of energy ε'. It can be cast into a basis-independent form suitable for Chebyshev expansions:

$$\Phi_\omega(\varepsilon,\varepsilon') = \text{Tr}\left[\Phi_{\text{tot}}(\omega)\,\delta(\varepsilon - H)\,\Phi_{\text{tot}}^\dagger(\omega)\,\delta(\varepsilon' - H)\right].$$

Plugging this back into the expression for the hot-carrier generation rate, we get

$$N_e(E,\omega) = \frac{4\pi}{\hbar V}\int_{-\infty}^{\infty} d\varepsilon \int_{-\infty}^{\infty} d\varepsilon' f(\varepsilon)\,(1 - f(\varepsilon'))\,\delta(E - \varepsilon')\,\delta(\varepsilon' - \varepsilon - \hbar\omega)\,\Phi_\omega(\varepsilon,\varepsilon').$$

$\Phi_\omega(\varepsilon,\varepsilon')$ is simply a function of two energies, so $N_e(E,\omega)$ can be obtained through a straightfoward numerical double integral. This form also makes it easier to interpret what is happening to $\Phi_\omega(\varepsilon,\varepsilon')$ during the integration process. Using the first Dirac delta to eliminate the integral in ε', the hot carrier generation rate can be simplified to

$$N_e(E,\omega) = \frac{4\pi}{\hbar V}(1 - f(E))\int_{-\infty}^{\infty} d\varepsilon f(\varepsilon)\,\delta_\gamma(E - \varepsilon - \hbar\omega)\,\Phi_\omega(\varepsilon,E). \tag{5}$$

The final ingredient to understand the inner workings of the code is the Chebyshev expansion the Dirac delta operators inside of $\Phi_\omega(\varepsilon,\varepsilon')$. These operators can be expanded in a series of Chebyshev polynomials

$$\delta(\varepsilon - H) = \sum_{n=0}^{M-1}\Delta_n(\varepsilon)\,T_n(H)$$

where

$$\Delta_n(\varepsilon) = \frac{2}{\delta_{n,0}+1}\frac{T_n(\varepsilon)}{\pi\sqrt{1-\varepsilon^2}}g_J^{n,M}$$

is the coefficient of the expansion regularized by the Jackson weight g_J. Thus, $\Phi_\omega(\varepsilon, \varepsilon')$ is expressed as

$$\Phi_\omega(\varepsilon, \varepsilon') = \mathrm{Tr}\left[\Phi_{\mathrm{tot}}(\omega)\left(\sum_{n=0}^{M-1}\Delta_n(\varepsilon)\,T_n(H)\right)\Phi_{\mathrm{tot}}^\dagger(\omega)\left(\sum_{m=0}^{M-1}\Delta_m(\varepsilon')\,T_m(H)\right)\right]$$

and can be calculated in two steps. The first step is to calculate the Chebyshev moments

$$\mu_{nm} = \mathrm{Tr}\left[\Phi_{\mathrm{tot}}(\omega)\,T_n(H)\,\Phi_{\mathrm{tot}}^\dagger(\omega)\,T_m(H)\right]$$

and the second is to resum the matrix:

$$\Phi_\omega(\varepsilon, \varepsilon') = \sum_{n=0}^{M-1}\sum_{m=0}^{M-1}\Delta_n(\varepsilon)\,\Delta_m(\varepsilon')\,\mu_{nm}$$

Part II
Code

11 External dependencies

These are the dependencies required to run the program.

- julia with the following dependencies
  - DelimitedFiles
  - Printf
  - MKL
  - Random
  - Dates
  - MKLSparse
  - HDF5
  - NearestNeighbors
  - DataStructures
  - Interpolations
  - SparseArrays
  - LinearAlgebra

Lines to install them using the Julia prompt

```
using Pkg;
Pkg.add.(["DelimitedFiles", "Printf", "MKL", "Random", "Dates", "MKLSparse", "HDF5", "N
```