OpenBTE

Release 0.1

Giuseppe Romano

Sep 19, 2023

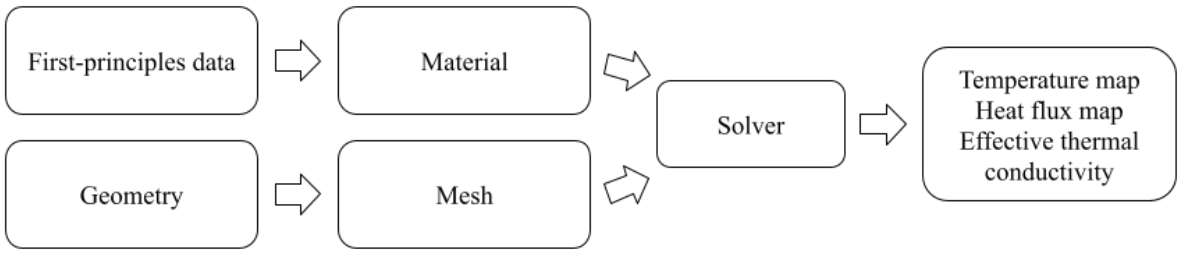
GETTING STARTED

CHAPTER

ONE

INTRODUCTION

OpenBTE is a Python-based tool for modeling particles flux at the nondiffusive level and in arbitrary geometries. Current focus is on thermal transport. The code implements the phonon Boltzmann transport equation, informed by first-principles calculations. Both forward and backward modes are supported, enabling inverse design using direct parameter optimization. A typical OpenBTE simulation is given by the combination of three main blocks: Mesh, Material and Solver.



When possible, OpenBTE automatically exploits parallelism using Python's Multiprocessing module. By default, all the available virtual cores (vCores) are employed. To run your script with a given number of vCores, use the `np` flag, e.g.

```
python run.py -np 4
```

Main features include:

- Vectorial mean-free-path interpolation
- Interface with first-principles solvers
- Arbitrary geometries
- Inverse design
- Interactive temperature and flux maps visualization
- Effective thermal conductivity
- Outputting data in `.vtu` format for advanced visualization

CHAPTER

TWO

INSTALLING OPENBTE

OpenBTE is available to install via the Python Package Index.

```
pip install --upgrade openbte
```

If you want to enable OpenBTE for GPUs, you will have to install the JAX version for your CUDA driver.

CHAPTER

THREE

EXAMPLE 1: POROUS MATERIAL

An OpenBTE simulation is specified by a combination of a material, geometry and solver. We begin with creating a material. To this end, we load previously computed first-principled calculations on Si at room temperature

```
from openbte import load_rta

rta_data = load_rta('Si_rta')
```

Next step is to perform MFP interpolation

```
from openbte import RTA2DSym

mat = RTA2DSym(rta_data)
```

The `RTA2DSym` model is for simulation domains which have translational symmetry along the z axis. To create a geometry, we instantiate an object of the class `Geometry`

```
from openbte import Geometry

G = Geometry(0.1)
```

where 0.1 is the carhacteristic size (in nm) of the mesh. In this example, we create a porous material with porosity 0.2 and rectangular aligned pores. Given the periodicity of the system, we simulate only a unit-cell to which we apply periodic boundary conditions. To define the unit-cell, we use the `add_shape` method

```
from openbte import rectangle

L         = 10 #nm

G.add_shape(rectangle(area = L*L))
```

To add the hole in the middle, we use the `add_hole` method

```
porosity = 0.2

area = porosity*L*L

G.add_hole(rectangle(area = area,x=0,y=0))
```

To apply boundary conditions, we need to assign a name to sides and refer to them in the solver section. Sides are selected with `selector`. In this case, we assign all internal sides the name `Boundary`

```
G.set_boundary_region(selector = 'inner',region = 'Boundary')
```

To apply periodic boundary conditions along both axes, we use the `set_periodicity` method

```
G.set_periodicity(direction = 'x',region      = 'Periodic_x')

G.set_periodicity(direction = 'y',region      = 'Periodic_y')
```

At this point, we are ready to save the mesh on disk

```
G.save()
```

If everything went smoothly, you should see `mesh.geo` in your current directory. You can open them with GMSH to check that the geometry has been created correctly. To create a meshed geometrym we use the function `get_mesh()`

```
from openbte import get_mesh

mesh = get_mesh()
```

Before setting up the solvers, we need to specify boundary conditions and perturbation. In this case, we apply a difference of temperature of $\Delta T_{\text{ext}} = 1$ K along x

```
from openbte.objects import BoundaryConditions

boundary_conditions = BoundaryConditions(periodic={'Periodic_x': 1,'Periodic_y':0},
↪diffuse='Boundary')
```

Note that we also specifies diffuse boundary conditions along the region `Boundary`. In this example, we are interested in the effective thermal conductivity along x

```
from openbte.objects import EffectiveThermalConductivity

effective_kappa = EffectiveThermalConductivity(normalization=-1,contact='Periodic_x')
```

where `normalization` (α) is used in the calculation of the effective thermal conductivity $\kappa_{\text{eff}} = \alpha \int_{-L/2}^{L/2} \mathbf{J}(L/2, y) \cdot \hat{\mathbf{n}} dy$. For rectangular domain, $\alpha = -L_x/L_y/\Delta T_{\text{ext}}$.

To run BTE calculations, we first solve standard heat conduction

```
from openbte import Fourier

fourier     = Fourier(mesh,mat.thermal_conductivity,boundary_conditions,\
                      effective_thermal_conductivity=effective_kappa)
```

Finally, using `fourier` as first guess, we solve the BTE

```
from openbte import BTE_RTA

bte     = BTE_RTA(mesh,mat,boundary_conditions,fourier=fourier,\
          effective_thermal_conductivity=effective_kappa)
```

Before plotting the results, we group together Fourier and BTE results

```
from openbte.objects import OpenBTEResults

results =  OpenBTEResults(mesh=mesh,material = mat,solvers={'bte':bte,'fourier':fourier})
```

Lastly, the temperature and heat flux maps can be obtained with

```
results.show()
```

GMSH

CHAPTER

FOUR

EXAMPLE 2: INVERSE DESIGN (NEW!)

In this example, we will design a nanomaterial withi a prescribed effective thermal conductivity tensor (diagonal components). To this end, we first get a BTE solver, specifically developed for inverse design

```
from openbte.inverse import bte

grid = 20

f = bte.get_solver(Kn=1, grid=grid,directions=[[1,0],[0,1]])
```

The function `f` takes the material density and gives the effective thermal conductivity tensor as well as its gradient wrt the material density. It is a differentiable function, i.e. it can be composed with arbitrary JAX functions to obtain end-to-end differentiability. This approach allows us to write down generic cost functions. In this case, we want to minimize $||\kappa - \tilde{\kappa}||$, where $\tilde{\kappa}$ is the desired value

```
from jax import numpy as jnp

kd = jnp.array([0.3,0.2])
def objective(x):

    k,aux  = f(x)

    g  = jnp.linalg.norm(k-kd)

    return g,(k,aux)
```

The gradient is managed automatically. Finally, the optimization is done with

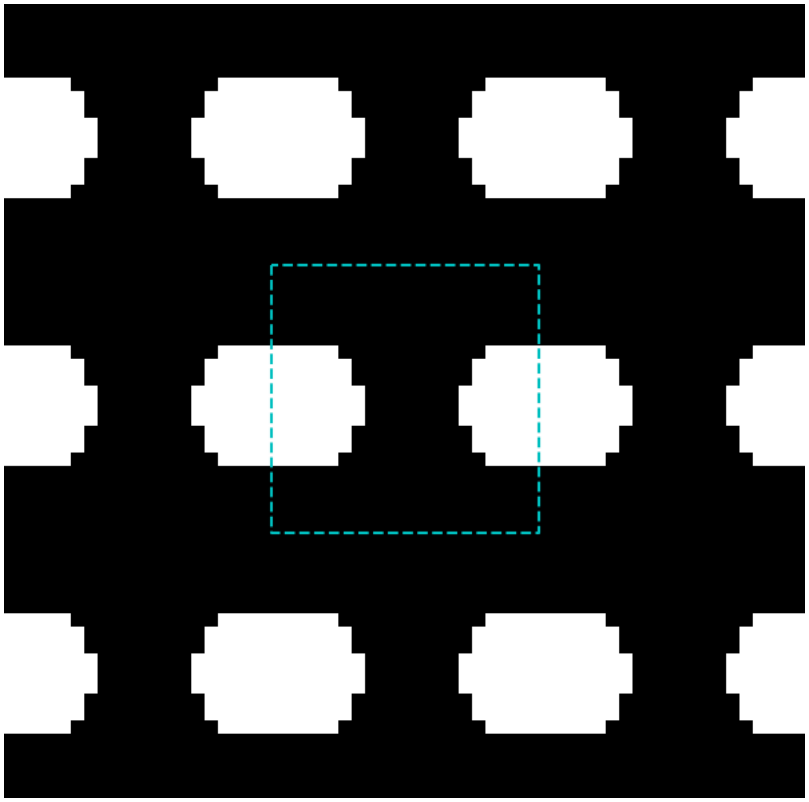
```
from openbte.inverse import matinverse as mi

L = 100 #nm
R = 30  #nm

x = mi.optimize(objective,grid = grid,L = L,R = R,min_porosity=0.05)
```

where `R` is the radius of the conic filter. Lastly, you can visualize the structure with

```
mi.plot(x)
```



CHAPTER

FIVE

REFERENCES

OpenBTE has been originally developed by Giuseppe Romano. If you use the code, we request to cite

G. Romano, OpenBTE: a Solver for ab-initio Phonon Transport in Multidimensional Structures, arXiv:2106.02764, (2021) Link.

Furthermore, please consider citing the feature specific references, listed below.

Inverse design:

G. Romano and S. G. Johnson, Inverse design in nanoscale heat transport via interpolating interfacial phonon transmission, Structural and Multidisciplinary Optimization, (2022) Link

Anisotropic MFP-BTE (`rta2DSim` material model):

G. Romano, Efficient calculations of the mode-resolved ab-initio thermal conductivity in nanostructures, arXiv:2105.08181 (2021) Link

Gray-BTE (`Gray2D` material model):

G. Romano, A Di Carlo, and J.C. Grossman, Mesoscale modeling of phononic thermal conductivity of porous Si: interplay between porosity, morphology and surface roughness. Journal of Computational Electronics 11 (1), 8-13 52 (2012) Link

CHAPTER

SIX

MODE-RESOLVED BULK DATA

The material underlying the structure is described by the following mode-resolved quantities:

Property [symbol]	Shape	Units
`scattering_time` [τ]	N	s
`heat_capacity` [C]	N	Jm $^{-3}$ K $^{-1}$
`group_velocity` [$\mathbf{v}$]	$N \times 3$	ms $^{-1}$

where $N = N_q \times N_b$, with N_q and N_b being the number of wave-vectors and polarization, respectively. The bulk thermal conductivity tensor is given by

$$\kappa^{\alpha\beta} = \sum_{\mu} C_{\mu} \tau_{\mu} v_{\mu}^{\alpha} v_{\mu}^{\beta}.$$

The (specific) heat capacity is defined by

$$C_{\mu} = \frac{k_B}{N_q \mathcal{V}} \left[\frac{\eta_{\mu}}{\sinh(\eta_{\mu})} \right]^2$$

with $\eta_{\mu} = \hbar\omega_{\mu}/(2k_bT)$. Note that the presence of N_q in the heat capacity is not standard but introduced here for convenience.

The material data is stored as a `sqlite3` database (with *.db* extension), which can be conveniently handled by sqlitedict. With your material file, say `foo.db`, is in the current directory, you can load it with

```
from openbte import load_rta

rta_data = load_rta('foo',source='local')
```

The above code is nothing than a wrapper to the `sqlite3`'s loader. Note that there is also a minimal database of precomputed materials. Currently, it includes silicon at room temperature, computed with AlmaBTE.

```
from openbte import load_rta

rta_data = load_rta('Si_rta',source='database')
```

To save your own material data, you can also use a wrapper

```
import openbte.utils

#C = ...
#v   = ...
#tau = ...
```

(continues on next page)

(continued from previous page)

```
utils.save('rta',{'scattering_time':tau,'heat_capacity':C,'group_velocity':v}
```

You can double-check the consistency of your data by comparing the resulting thermal conductivity with the expected one

```
import numpy as np
#C = ...
#v   = ...
#tau = ...
#kappa = ... #Expected thermal conductivity tensor

print(np.allclose(np.einsum('u,ui,uj,u->ij',C,v,v,tau)-kappa))
```

CHAPTER

SEVEN

ALMABTE INTERFACE

AlmaBTE is a package that computes the thermal conductivity of bulk materials, thin films and superlattices. OpenBTE is interfaced with AlmaBTE for RTA calculations via the script `almabte2openbte.py`.

Assuming you have `AlmaBTE` in your current `PATH`, this an example for `Si`.

- Download Silicon force constants from AlmaBTE's database

```
wget https://almabte.bitbucket.io/database/Si.tar.xz
tar -xf Si.tar.xz && rm -rf Si.tar.xz
```

- Compute bulk scattering time with AlmaBTE.

```
echo "<singlecrystal>
<compound name='Si'/>
<gridDensity A='8' B='8' C='8'/>
</singlecrystal>" > inputfile.xml

VCAbuilder inputfile.xml
phononinfo -f Si/Si_8_8_8.h5 300.0
```

- A file named `Si_8_8_8_300K.phononinfo` is in your current directory. Note that you can specify the temperature. Here we choose 300 K. The file `rta.db` can then be created with

```
AlmaBTE2OpenBTE Si_8_8_8_300K.phononinfo
```

Finally, you can load the data with

```
from openbte import load_rta

rta_data = load_rta('rta',source='local')
```

CHAPTER

EIGHT

VECTORIAL MFP INTERPOLATION

Next step is to perform vectorial MFP interpolation, as documented here. For systems with translational symmetry along the z axis, you can use

```
from openbte import RTA2DSym

rta_data = RTA2DSym(rta_data,**kwargs)
```

where options include `n_mfp` and `n_phi`, i.e. the number of MFP (default = 50) and polar angle bins (default = 48). If you are simulating a 3D material (not yet fully supported), you can use

```
from openbte import RTA3D

mat = RTA3D(rta_data,**kwargs)
```

where options now include `n_theta` as well, i.e. the azimuthal angular bins.

CHAPTER

NINE

GMSH

OpenBTE uses Gmsh as a backend for geometry building. Once the file `mesh.msh` is created, it can be imported with

```
from openbte import get_mesh

mesh = get_mesh()
```

The physical regions will then be referred to by boundary conditions. OpenBTE handles 2D and 3D geometries, although only 2D systems are currently supported. The name of physical regions associating to two periodic boundaries must end with `_a` and `_b`.

CHAPTER

TEN

GEOMETRY INTERFACE

For simple geometries, it is possible to write Gmsh code automatically using the `Geometry` module. As an example, let's start with the creation of a porous material, consisting of a unit-cell and a square pore located in the center. The fist step is to build the `Geometry` object

```
from openbte import Geometry

G = Geometry(0.1)
```

where `0.1` is the characteristic mesh size in nm. Then, the outer frame of the geometry is defined with the `add_shape` method

```
from openbte import rectangle

L         = 10 #nm

G.add_shape(rectangle(area = L*L,aspect_ratio = 1))
```

which in this case creates a square frame with side of 10 nm, whis is the unit-cell of the domain. The pore in the center added via

```
from openbte import rectangle

porosity = 0.2

area = porosity*L*L

G.add_hole(rectangle(area = area,x=0,y=0))
```

If no name of the hole is given, then it will be considered as a void region and not included in the meshing. If a name if given, then it is included in the simulation domain and referred to it during the BTE solution, e.g. for heat sources. Finally, we have to define the boundary regions. This task entails selecting the boundary, through `selector` and associate a name to it. The following selelectors are available: `outer`, `inner`, `all`, `top`, `bottom`, `left` and `right`. In this case, we assign the name `Boundary` to all internal region, i.e. the wall of the pore.

```
G.set_boundary_region(selector = 'inner',region = 'Boundary')
```

Periodic boundary conditions can be assigned with the `set_periodicity` method

```
G.set_periodicity(direction = 'x',region      = 'Periodic_x')

G.set_periodicity(direction = 'y',region      = 'Periodic_y')
```

The region names `Boundary`, `Periodic_x` and `Periodic_y` will be referred to in the boundary conditions. Note that **all** the boundaries of the simulation domain need to be associated to a region name. Lastly, the file `mesh.msh` is created with

```
G.save()
```

To inspect your geometry, you can call `gmsh` from your command line

```
gmsh mesh.geo
```

and check the physical regions in the `visibility` section of the `tool` drop-down menu.

CHAPTER

ELEVEN

BOUNDARY CONDITIONS

The object `BoundaryConditions` connects the physical boundaries defined in `Geometry` with the actual physics. There are three boundary conditions: `diffuse`, `mixed` and `periodic`. In case of periodic boundary conditions, we can also specify a temperature jump (albeit is not strictly a temperature jump but a heat source/sink pair) applied along the associated direction. In our case, we apply a temperature jump of 1 K along x and associate the region `Boundary` to diffuse boundary conditions

```
from openbte.objects import BoundaryConditions

boundary_conditions = BoundaryConditions(periodic={'Periodic_x': 1,'Periodic_y':0},
↪diffuse='Boundary')
```

The mixed boundary conditions include a thermostatting boundaries and a boundary conductance [in $\mathrm{Wm}^{-2}\ \mathrm{K}^{-1}$]

```
from openbte.objects import BoundaryConditions

boundary_conditions = BoundaryConditions(mixed={'Isothermal': {'value':300,'boundary_
↪conductance':1e4}})
```

where we assumed that a region named `Isothermal` was previously defined.

CHAPTER

TWELVE

EFFECTIVE THERMAL CONDUCTIVITY

To compute the effective thermal conductivity, we average the flux over a given contact. For example, assuming a rectangular domain with size $L_x \times L_y$, the effective thermal conductivity for a perturbation applied along x (κ_{xx}) is given by

$$\kappa_{xx} = \alpha \int_{-L_y/2}^{L_y/2} \mathbf{J}(L/2, y) \cdot \hat{\mathbf{n}} dy$$

where $\alpha = -L_x/L_y/\Delta T_{\text{ext}}$. To this end, we define the boundary region name and the normalization factor $alpha$

```
from openbte.objects import EffectiveThermalConductivity

effective_kappa = EffectiveThermalConductivity(normalization=-1,contact='Periodic_x')
```

CHAPTER

THIRTEEN

SOLVER

Currently, OpenBTE supports the anisotropic-MFP-BTE

$$\mathbf{F}_\mu \cdot \nabla \tilde{T}_\mu + \tilde{T}_\mu = \left[\sum_{\mu''} \gamma_{\mu''}\right]^{-1} \sum_{\mu'} \gamma_{\mu'} \tilde{T}_{\mu'}$$

where $\mathbf{F} = \mathbf{v}_\tau \tau_\mu$ is the vectorial MFP and $\gamma_\mu = C_\mu / \tau_\mu$. The first step in solving this equation is to provide a first guess for the lattice temperature via solving the standard heat conduction equation. To this end, we use the `Fourier` solver

```
from openbte import Fourier

fourier     = Fourier(mesh,mat.thermal_conductivity,boundary_conditions,\
                      effective_thermal_conductivity=effective_kappa)
```

Finally, we can solve the BTE with

```
from openbte import BTE_RTA

bte     = BTE_RTA(mesh,mat,boundary_conditions,fourier=fourier,\
                  effective_thermal_conductivity=effective_kappa)
```

The result of the simulations can then be consolidated in a `OpenBTEResults` object

```
from openbte.objects import OpenBTEResults

results = OpenBTEResults(mesh=mesh,material = mat,solvers={'bte':bte,'fourier':fourier})
```

Lastly, we can save the results with

```
results.save()
```

where the file `state.db` is saved in you current directory. Optionally you can define a custom `filename`, without including the `.db` suffix. Once ready for postprocessing, results can be load with

```
from openbte.objects import OpenBTEResults

results = OpenBTEResults.load()
```

where an optional `filename` can be specified.

CHAPTER

FOURTEEN

PLOTTING THERMAL AND HEAT FLUX MAPS

Flux and temperatures maps can be visualized with

```
from openbte.objects import OpenBTEResults

results = OpenBTEResults.load()

results.show()
```

For advanced visualization, e.g. slicing etc…, you can save results in the `.vtu` format

```
results.vtu()
```

This will craete a file, `output.vtu`, compatible with the popular software Paraview .

CHAPTER

FIFTEEN

LINE DATA

Line data can be plotted with

```
N = 100

path = np.stack((np.zeros(N),np.linspace(-0.5,0.5,N))).T

x,data = results.plot_over_line(variables=['Temperature_Fourier','Temperature_BTE','Flux_
↪fourier','Flux_BTE'],x=path)
```

The variable `x` is the distance on the path and `data` is a dictionary containing the data interpolated on the path. For example, the temperature computed with BTE can be accessed with `data['Temperature_BTE']`.

CHAPTER

SIXTEEN

INVERSE DESIGN

Inverse design seeks to find a material structure that leads to the desired material property. In this space, `OpenBTE` provides routines for the standard heat conduction equation and Boltzmann transport equation in the single-MFP approximation, both solved over structured grids. We adopt the filtering and projecting method [O. Sigmund and K. Maute (2013)] and the newly introduce Transmisison Interpolation Method (TIM) [G. Romano and S. G Johnson (2022)]. The automatic differentiation framework is based on JAX. The optimization algorithm of choice is the method of moving asymptotes (MMA) [Svanberg (2002)], implemented in NLOpt. The first step is to import a solver from the submodule `openbte.inverse`

```
from openbte.inverse import bte
from openbte.inverse import fourier
```

Currently, only periodic structures can be simulated, with the perturbation being aligned along specified directions. Multiple directions can be considered in the same simulations, e.g.

```
grid = 20

b = bte.get_solver(Kn=1, grid=grid,directions=[[1,0],[0,1]])
f = fourier.get_solver(grid=grid,directions=[[1,0],[0,1]])
```

Note that both solvers also take the grid in input, with the BTE solver also requiring the Knudsen number. The variables `f` and `b` are fully differentiable solvers and can be integrated in an optimization pipeline. Here, we define the following cost function

```
from jax import numpy as jnp

kd = jnp.array([0.3,0.2])
def objective(x):

   k,aux  = f(x)

   g  = jnp.linalg.norm(k-kd)

   return g,(k,aux)
```

which seeks to engineer a material with a thermal conductivity tensor having the components $\kappa_{xx} = 0.3$ and $\kappa_{xx} = 0.2$. Note that in this case, we chose to use the Fourier solver. Lastly, we start the optimization with

```
from openbte.inverse import matinverse as mi

L = 100 #nm
R = 30  #nm
```

(continues on next page)

(continued from previous page)

```
x = mi.optimize(objective,grid = grid,L = L,R = R,min_porosity=0.05)
```

where we specify the size of the simulation domain, the radius of the conic filter

$$w_c = \begin{cases} \frac{1}{a}\left(1 - \frac{|\mathbf{r}_c|}{R}\right), & |\mathbf{r}| < R \\ 0, & \text{otherwise.} \end{cases}$$

Note that we also added the constraint of minimum porosity. Supported constraints include `min_porosity` and `max_porosity`. Other options include:

- `n_betas`: The number of beta doubling for projection (default = 10). It starts with $\beta = 2$, and always end with a pass with a very large β, not accounted for in this option
- `max_iter`: The number of iteration for each beta (default=25)
- `tol`: tolerance on the cost function
- `inequality_constraints`: a list of function to be used as inequality constraints. They follow the same syntax as the objective function
- `monitor`: whether to have intermediate structures plotted during optimization
- `output_file`: whether to save the convergence history (default==``output``).

Although this example pertains to thermal transport, the inverse design framework is quite general and can be used in combination with other differentiable solvers. If you wish to plug your own solver, it must follow this syntax

```
def solver(x)

    ...
    ...

    return (output,aux),jacobian
```

where `x` is the material density (N), `output` is the cost function (M), and `jacobian` is the sensitivity of the cost function with respect to the material density ($M \times N$). Then, it can be interfaced with

```
from openbte.inverse import matinverse as mi

s = mi.compose(solver)
```

From now on, `s` can be used in the optimizion pipeline, which includes filtering and projection. Note that `aux` includes variables that are not directly related to optimization but that are still worth retaining for later use, e.g. the temperature map.