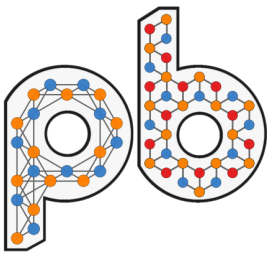
Pybinding User Guide

Release 0.9.4

Dean Moldovan

Jul 13, 2017

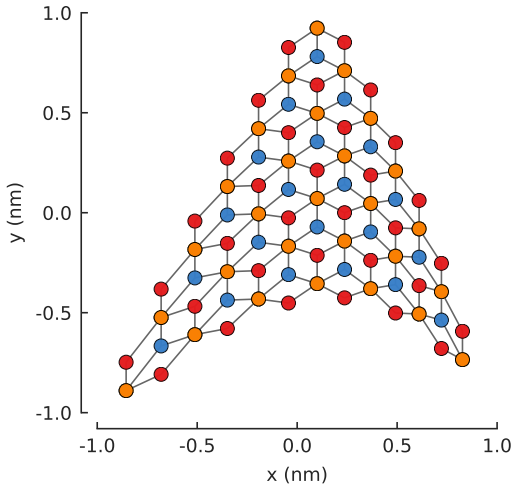
Contents



Pybinding is a scientific Python package for numerical tight-binding calculations in solid state physics. If you're just browsing, the *Tutorial* section is a good place to start. It gives a good overview of the most important features with lots of code examples.

As a very quick example, the following code creates a triangular quantum dot of bilayer graphene and then applies a custom asymmetric strain function:

```
import pybinding as pb
from pybinding.repository import graphene

def asymmetric_strain(c):
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = -c/2 * x**2 + c/3 * x + 0.1
        uy = -c*2 * x**2 + c/4 * x
        return x + ux, y + uy, z
    return displacement

model = pb.Model(
    graphene.bilayer(),
    pb.regular_polygon(num_sides=3, radius=1.1),
    asymmetric_strain(c=0.42)
)
model.plot()
```



Within the pybinding framework, tight-binding models are assembled from logical parts which can be mixed and matched in various ways. The package comes with a few predefined components: crystal lattices, shapes, symmetries, defects, fields and more (like the `graphene.bilayer()` lattice and the *`regular_polygon()`* shape shown above). Users can also define new components (just like the asymmetric strain above). This modular approach enables the construction of arbitrary tight-binding models with clear, easy-to-use code. Various solvers, computation routines and visualization tools are also part of the package. See the *Tutorial* for a walkthrough of the features.

The source code repository is located on Github where you can also post any questions, comments or issues that you might have.

Pybinding is a Python package for numerical tight-binding calculations in solid state physics. The main features include:

- **Declarative model construction** - The user just needs to describe *what* the model should be, but not *how* to build it. Pybinding will take care of the numerical details of building the Hamiltonian matrix so users can concentrate on the physics, i.e. the quantum properties of the model.
- **Fast compute** - Pybinding's implementation of the kernel polynomial method allows for very fast calculation of various physical properties of tight-binding systems. Exact diagonalization is also available through the use of scipy's eigenvalue solvers. The framework is very flexible and allows the addition of user-defined computation routines.
- **Result analysis and visualization** - The package contains utility functions for post-processing the raw result data. The included plotting functions are tailored for tight-binding problems to help visualize the model structure and to make sense of the results.

The main interface is written in Python with the aim to be as user-friendly and flexible as possible. Under the hood, C++11 is used to accelerate demanding tasks to deliver high performance with low memory usage.

CHAPTER 1

Background

The tight-binding model is an approximate approach of calculating the electronic band structure of solids using a basis of localized atomic orbitals. This model is applicable to a wide variety of systems and phenomena in quantum physics. The approach does not require computing from first principals, but instead simply uses parameterized matrix elements. In contrast to *ab initio* calculations, the tight-binding model can scale to large system sizes on the order of millions of atoms.

Python is a programming language which is easy to learn and a joy to use. It has deep roots in the scientific community as evidenced by the rich scientific Python library collection: SciPy. As such, Python is the ideal choice as the main interface for pybinding. In the core of the package, C++11 is used to accelerate model construction and the most demanding calculations. This is done silently in the background.

CHAPTER 2

Workflow

The general workflow starts with model definition. Three main parts are required to describe a tight-binding model:

- **The crystal lattice** - This step includes the specification of the primitive lattice vectors and the configuration of the unit cell (atoms, orbitals and spins). This can be user-defined, but the package also contains a repository of the pre-made specifications for several materials.
- **System geometry** - The model system can be infinite through the use of translational symmetry or it can be finite by specifying a shape. The two approaches can also be composed to create periodic systems with intricate structural patterns. The structure can be controlled up to fine details, e.g. to form specific edge types as well as various defects.
- **Fields** - Functions can be applied to the onsite and hopping energies of the model system to simulate external fields or various effects. These functions are be defined independently of any lattice or specific structure which makes them easily reusable and mutually composable.

Once the model description is complete, pybinding will build the tight-binding Hamiltonian matrix. The next step is to apply computations to the matrix to obtain the values of the desired quantum properties. To that end, there are the following possibilities:

- **Kernel polynomial method** - Pybinding implements a fast Chebyshev polynomial expansion routine which can be used to calculate various physical properties. For example, it's possible to quickly compute the local density of states or the transport characteristics of the system.
- **Exact diagonalization** - Eigensolvers may be used to calculate the eigenvalues and eigenvectors of the model system. Common dense and sparse matrix eigensolvers are available via SciPy.
- **User-defined compute** - Pybinding constructs the Hamiltonian in the standard sparse matrix CSR format which can be plugged into custom compute routines.

After the main computation is complete, various utility functions are available for post-processing the raw result data. The included plotting functions are tailored for tight-binding problems to help visualize the model structure and to make sense of the results.

CHAPTER 3

Citing

Pybinding is free to use under the simple conditions of the BSD open source license (included below). If you wish to use results produced with this package in a scientific publication, please just mention the package name in the text and cite the Zenodo DOI of this project:

You'll find a *"Cite as"* section in the bottom right of the Zenodo page. You can select a citation style from the dropdown menu or export the data in BibTeX and similar formats.

CHAPTER 4

BSD License

CHAPTER 5

Benchmarks

One of the main features of pybinding is an easy-to-use and fast model builder: it constructs the tight-binding Hamiltonian matrix. This can be a demanding task for large or complicated systems (with many parameters). Great care was taken to make this process fast.

We compare the performance of pybinding with the Kwant package. Both code packages are based on the numerical tight-binding method and can build identical Hamiltonian matrices. For calculations involving these matrices, the packages specialize in different ways: Kwant is intended for transport calculations with scattering systems while pybinding targets large finite-sized and periodic systems in 1 to 3 dimensions. Pybinding can also be used to construct scattering systems, however it does not have a builtin solver for transport problems. This is where the *Kwant compatibility* layer comes in: it's possible to build a system in pybinding and use Kwant's solvers for transport calculations. This combination takes advantage of the much faster model builder – see the comparison below.

System construction

The code used to obtain these results is available here: `Source code`. You can download it and try it on your own computer. Usage instructions are located at the top of the script file.

The benchmark constructs a circular graphene flake with a pn-junction and a constant magnetic field. The system build time is measured from the start of the definition to the point where the Hamiltonian matrix is fully constructed (a sparse matrix is used in both cases).

Pybinding builds the Hamiltonian much faster than Kwant: by two orders of magnitude. The main reason for this is in the way the system shape and fields are implemented. Both Kwant and pybinding take user-defined functions as parameters for model construction. Kwant calls these functions individually for each atom and hopping which is quite slow. Pybinding stores all atoms and hoppings in contiguous arrays and then calls the user-defined functions just once for the entire dataset. This takes advantage of vectorization and drastically improves performance. Similarly, the lower memory usage is achieved by using arrays and CSR matrices rather than linked lists and trees.

Please note that at the time of writing pybinding v0.8 does lack certain system construction features compared to Kwant. Specifically, it is currently not possible to build heterostructures in pybinding, but this will be resolved in the near future. New features will be added while maintaining good performance.

At first glance it may seem like system build time is not really relevant because it is only done once and then multiple calculations can be applied to the constructed system. However, every time a parameter is changed (like some field strength) the Hamiltonian matrix will need to be rebuilt. Even though Kwant does take this into account and only does a partial rebuild, pybinding is still much faster and this is very apparent in transport

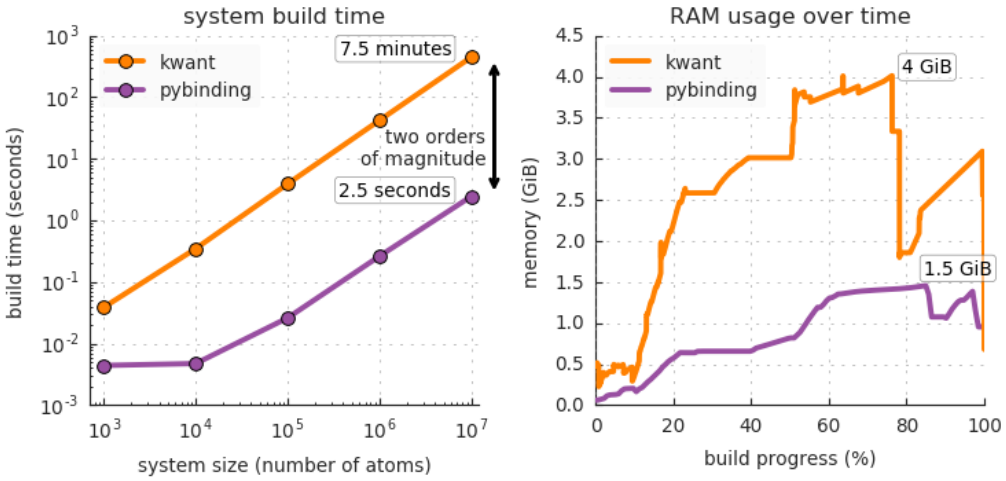


Fig. 5.1: The results were measured for pybinding v0.8.0 and kwant v1.2.2 using: Intel Core i7-4960HQ CPU, 16 GiB RAM, Python 3.5, macOS 10.11. The RAM usage was measured using memory_profiler v0.41.

calculations which sweep over some model parameter. For more information and a direct comparison, see the *Kwant compatibility* section.

CHAPTER 6

Changelog

v0.9.4 | 2017-07-13

- Fixed issues with multi-orbital models: matrix onsite terms were not set correctly if all the elements on the main diagonal were zero (#5), hopping terms were being applied asymmetrically for large multi-orbital systems (#6). Thanks to @oroszl (László Oroszlány) for reporting the issues.
- Fixed KPM Hamiltonian scaling for models with all zeros on the main diagonal but asymmetric spectrum bounds (non-zero KPM scaling factor `b`).
- Fixed compilation on certain Linux distributions (#4). Thanks to @nu11us (Will Eggleston) for reporting the issue.
- Fixed compilation with Visual Studio 2017.
- Improved support for plotting slices of multi-layer systems. See "Plotting Guide" > "Model structure" > "Slicing layers" in the documentation.

v0.9.3 | 2017-05-29

- Added support for Kwant v1.3.x and improved `Model.tokwant()` exporting of multi-orbital models.
- Fixed errors when compiling with GCC 6.

v0.9.2 | 2017-05-26

New KPM features and improvements

- Added a method for calculating spatial LDOS using KPM. See the "Kernel Polynomial Method" tutorial page and the `KPM.calc_spatial_ldos` API reference.
- Improved single-threaded performance of `KPM.calc_dos` by ~2x by switching to a more efficient vectorization method. (Multiple random starter vectors are now computed simultaneously and accelerated using SIMD intrinsics.)

- Various KPM methods now take advantage of multiple threads. This improves performance depending on the number of cores on the target machine. (However, for large systems performance is limited by RAM bandwidth, not necessarily core count.)
- LDOS calculations for multiple orbitals also take advantage of the same vectorization and multi-threading improvements. Single-orbital LDOS does not benefit from this but it has received its own modest performance tweaks.
- Long running KPM calculation now have a progress indicator and estimated completion time.

General improvements and bug fixes

- `StructureMap` can now be sliced using a shape. E.g. `s = pb.rectangle(5, 5); smap2 = smap[s]` which returns a smaller structure map cut down to the given shape.
- Plotting the structure of large or periodic systems is slightly faster now.
- Added 2D periodic supercells to the "Shape and symmetry" section of the tutorial.
- Added a few more examples to the "Plotting guide" (view rotation, separating sites and hoppings and composing multiple plots).
- Fixed broken documentation links when using the online search function.
- Fixed slow Hamiltonian build when hopping generators are used.

v0.9.1 | 2017-04-28

- Fixed an issue with multi-orbital models where onsite/hopping modifiers would return unexpected results if a new `energy` array was returned (rather than being modified in place).
- Fixed `Solver.calc_spatial_ldos` and `Solver.calc_probability` returning single-orbital results for multi-orbital models.
- Fixed slicing of `Structure` objects and made access to the `data` property of `SpatialMap` and `StructureMap` mutable again.

v0.9.0 | 2017-04-14

Updated requirements

- This version includes extensive internal improvements and raises the minimum requirements for installation. Starting with this release, only Python >= 3.5 is supported. Newer versions of the scientific Python packages are also required: numpy >= 1.12, scipy >= 0.19 and matplotlib >= 2.0.
- On Linux, the minimum compiler requirements have also been increased to get access to C++14 for the core of the library. To compile from source, you'll need GCC >= 5.0 or clang >= 3.5.

Multi-orbital models

- Improved support for models with multiple orbitals, spins and any additional degrees of freedom. These can now be specified simply by inputing a matrix as the onsite or hopping term (instead of a scalar value). For more details, see the "Multi-orbital models" section of the documentation.
- Lifted all limits on the number of sublattices and hoppings which can be defined in a `Lattice` object. The previous version was limited to a maximum of 128 onsite and hopping terms per unit cell (but those could be repeated an unlimited number of times to form a complete system). All restrictions are now removed so

that the unit cell size is only limited by available memory. In addition, the memory usage of the internal system format has been reduced.

- Added a 3-band model of group 6 transition metal dichalcogenides to the Material Repository. The available TMDs include: MoS2, WS2, MoSe2, WSe2, MoTe2, WTe2. These are all monolayers.

Composite shapes

- Complicated system geometries can now be created easily by composing multiple simple shapes. This is done using set operations, e.g. unions, intersections, etc. A complete guide for this functionality is available in the "Composite shapes" section of the documentation.

Kernel polynomial method

- The KPM implementation has been revised and significantly expanded. A guide and several examples are available in the "Kernel polynomial method" section of the documentation (part 9 of the Tutorial). For a complete overview of the available methods and kernels, see the `chebyshev` section of the API reference.
- New builtin computation methods include the stochastically-evaluated density of states (DOS) and electrical conductivity (using the Kubo-Bastin approach).
- The new low-level interface produces KPM expansion moments which allows users to create their own KPM-based computation routines.
- The performance of various KPM computations has been significantly improved for CPUs with AVX support (~1.5x speedup on average, but also up to 2x in some cases with complex numbers).

Miscellaneous

- Added the `pb.save()` and `pb.load()` convenience functions for getting result objects into/out of files. The data is saved in a compressed binary format (Python's builtin `pickle` format with protocol 4 and gzip). Loaded files can be immediately plotted: `result = pb.load("file.pbz")` and then `result.plot()` to see the data.
- The eigenvalue solvers now have a `calc_ldos` method for computing the local density of states as a function of energy (in addition to the existing `calc_spatial_ldos`).
- Improved plotting of `Lattice` objects. The view can now be rotated by passing the `axis="xz"` argument, or any other combination of x, y and z to define the plotting plane.

Deprecations and breaking changes

- Added `Lattice.add_aliases()` method. The old `Lattice.add_sublattice(..., alias=name)` way of creating aliases is deprecated.
- The `greens` module has been deprecated. This functionality is now covered by the KPM methods.
- The internal storage format of the `Lattice` and `System` classes has been revised. This shouldn't affect most users who don't need access to the low-level data.

v0.8.2 | 2017-01-26

- Added support for Python 3.6 (pybinding is available as a binary wheel for Windows and macOS).
- Fixed compatibility with matplotlib v2.0.
- Fixed a few minor bugs.

v0.8.1 | 2016-11-11

- Structure plotting functions have been improved with better automatic scaling of lattice site circle sizes and hopping line widths.
- Fixed Brillouin zone calculation for cases where the angle between lattice vectors is obtuse (#1). Thanks to @obgeneralao (Oliver B Generalao) for reporting the issue.
- Fixed a flaw in the example of a phosphorene lattice (there were extraneous t5 hoppings). Thanks to Long-long Li for pointing this out.
- Fixed missing CUDA source files in PyPI sdist package.
- Revised advanced installation instructions: compiling from source code and development.

v0.8.0 | 2016-07-01

New features

- Added support for scattering models. Semi-infinite leads can be attached to a finite-sized scattering region. Take a look at the documentation, specifically section 10 of the "Basic Tutorial", for details on how to construct such models.
- Added compatibility with Kwant for transport calculations. A model can be constructed in pybinding and then exported using the `Model.tokwant()` method. This makes it possible to use Kwant's excellent solver for transport problems. While Kwant does have its own model builder, pybinding is much faster in this regard: by two orders of magnitude, see the "Benchmarks" page in the documentation for a performance comparison.
- *Experimental:* Initial CUDA implementation of KPM Green's function (only for diagonal elements for now). See the "Experimental Features" section of the documentation.

Improvements

- The performance of the KPM Green's function implementation has been improved significantly: by a factor of 2.5x. The speedup was achieved with CPU code using portable SIMD intrinsics thanks to libsimdpp.
- The Green's function can now be computed for multiple indices simultaneously.
- The spatial origin of a lattice can be adjusted using the `Lattice.offset` attribute. See the "Advanced Topics" section.

Breaking changes

- The interface for structure plotting (as used in `System.plot()` and `StructureMap`) has been greatly improved. Some of the changes are not backwards compatible and may require some minor code changes after upgrading. See the "Plotting Guide" section of the documentation for details.
- The interfaces for the `Bands` and `StructureMap` result objects have been revised. Specifically, structure maps are now more consistent with ndarrays, so the old `smap.filter(smap.x > 0)` is replaced by `smap2 = smap[smap.x > 0]`. The "Plotting Guide" has a few examples and there is a full method listing in the "API Reference" section.

Documentation

- The API reference has been completely revised and now includes a summary on the main page.

- A few advanced topics are now covered, including some aspects of plotting. A few more random examples have also been added.
- Experimental features are now documented.

Bug fixes

- Fixed translational symmetry skipping directions for some 2D systems.
- Fixed computation of off-diagonal Green's function elements with `opt_level > 0`
- Fixed some issues with shapes which were not centered at `(x, y) = (0, 0)`.

v0.7.2 | 2016-03-14

- Lots of improvements to the documentation. The tutorial pages can now be downloaded and run interactively as Jupyter notebooks. The entire user guide is also available as a PDF file.
- The `sub_id` and `hop_id` modifier arguments can now be compared directly with their friendly string names. For example, this makes it possible to write `sub_id == 'A'` instead of the old `sub_id == lattice['A']` and `hop_id == 'gamma1'` instead of `hop_id == lattice('gamma1')`.
- The site state modifier can automatically remove dangling sites which have less than a certain number of neighbors (set using the `min_neighbors` decorator argument).
- Added optional `sites` argument for state, position, and onsite energy modifiers. It can be used instead of the `x, y, z, sub_id` arguments and contains a few helper methods. See the modifier API reference for more information.
- Fixed a bug where using a single KPM object for multiple calculations could return wrong results.
- *Experimental* `hopping_generator` which can be used to add a new hopping family connecting arbitrary sites independent of the main `Lattice` definition. This is useful for creating additional local hoppings, e.g. to model defects.

v0.7.1 | 2016-02-08

- Added support for double-precision floating point. Single precision is used by default, but it will be switched automatically to double if required by an onsite or hopping modifier.
- Added support for the 32-bit version of Python
- Tests are now included in the installed package. They can be run with:

```
import pybinding as pb
pb.tests()
```

- Available as a binary wheel for 32-bit and 64-bit Windows (Python 3.5 only) and OS X (Python 3.4 and 3.5)

v0.7.0 | 2016-02-01

Initial release

CHAPTER 7

Installation

Pybinding can be installed on Windows, Linux or Mac, with the following prerequisites:

- Python 3.5 or newer (Python 2.x is not supported)
- The SciPy stack of scientific packages, with required versions:
  - numpy >= v1.12
  - scipy >= v0.19
  - matplotlib >= v2.0
- If you're using Linux, you'll also need GCC >= v5.0 (or clang >= v3.5) and CMake >= v3.1.

You can install all of this in two ways:

Quick Install

The easiest way to install Python and SciPy is with Anaconda, a free scientific Python distribution for Windows, Linux and Mac. The following install guide will show you how to install the minimal version of Anaconda, Miniconda, and then install pybinding.

Note: If you run into any problems during the install process, check out the *Troubleshooting* section.

Windows

1. Download the Miniconda Python 3.x installer: Miniconda3-latest-Windows-x86_64.exe. Run it and accept the default options during the installation.
2. Open `Command Prompt` from the `Start` menu. Enter the following command to install the scientific Python packages with Miniconda:

   ```
   conda install numpy scipy matplotlib
   ```

3. The next command will download and install pybinding:

```
pip install pybinding
```

That's it, all done. Check out the *Tutorial* for some example scripts to get started. To run a script file, e.g. `example1.py`, enter the following command:

```
python example1.py
```

Linux

You will need gcc and g++ 5.0 or newer. To check, enter the following in terminal:

```
g++ --version
```

If your version is outdated, check with your Linux distribution on how to upgrade. If you have version 5.8 or newer, proceed with the installation.

1. Download the Miniconda Python 3.x installer: Miniconda3-latest-Linux-x86_64.sh. Run it in your terminal window:

   ```
   bash Miniconda3-latest-Linux-x86_64.sh
   ```

   Follow the installation steps. You can accept most of the default values, but make sure that you type `yes` to add Miniconda to `PATH`:

   ```
   Do you wish the installer to prepend the Miniconda3 install location
   to PATH in your /home/<user_name>/.bashrc ? [yes|no]
   [no] >>> yes
   ```

   Now, close your terminal window and open a new one for the changes to take effect.

2. Install CMake and the scientific Python packages:

   ```
   conda install cmake numpy scipy matplotlib
   ```

3. The next command will download and install pybinding:

   ```
   pip install pybinding
   ```

That's it, all done. Check out the *Tutorial* for some example scripts to get started. To run a script file, e.g. `example1.py`, enter the following command:

```
python example1.py
```

Mac OS X

1. Download the Miniconda Python 3.x installer: Miniconda3-latest-MacOSX-x86_64.sh. Run it in your terminal window:

   ```
   bash Miniconda3-latest-MacOSX-x86_64.sh
   ```

   Follow the installation steps. You can accept most of the default values, but make sure that you type `yes` to add Miniconda to `PATH`:

   ```
   Do you wish the installer to prepend the Miniconda3 install location
   to PATH in your /Users/<user_name>/.bash_profile ? [yes|no]
   [yes] >>> yes
   ```

   Now, close your terminal window and open a new one for the changes to take effect.

2. Install CMake and the scientific Python packages:

   ```
   conda install cmake numpy scipy matplotlib
   ```

3. The next command will download and install pybinding:

   ```
   pip install pybinding
   ```

That's it, all done. Check out the *Tutorial* for some example scripts to get started. To run a script file, e.g. `example1.py`, enter the following command:

```
python example1.py
```

Troubleshooting

If you already had Python installed, having multiple distributions may cause trouble in some cases. Check the `PATH` environment variable and make sure the Miniconda has priority.

Advanced Install

If you've completed the *Quick Install* guide, you can skip right to the *Tutorial*. This section is intended for users who wish to have more control over the install process or to compile from source code. If you're looking for a simple solution, see the *Quick Install* guide.

Without Anaconda

If you already have Python 3.x installed from python.org or anywhere else, you can use your existing distribution instead of Anaconda (or Miniconda). Note that this does require manually installing some dependencies.

Windows

1. Install the Visual C++ 2015 Runtime.
2. Install numpy, scipy and matplotlib binaries from Christoph Gohlke.
3. Pybinding is available as a binary wheel on PyPI. Install it with:

   ```
   pip3 install pybinding
   ```

Linux

Building pybinding from source is the only option on Linux.

1. Make sure you have gcc and g++ v5.0 or newer. To check, run `g++ --version` in your terminal. Refer to instruction from your Linux distribution in case you need to upgrade. Alternatively, you can use clang v3.5 or newer for compilation instead of gcc.
2. Install CMake >= v3.1 from their website or your package manager, e.g. `apt-get install cmake`.
3. Install numpy, scipy and matplotlib with the minimal versions as *stated previously*. The easiest way is to use your package manager, but note that the main repositories tend to keep outdated versions of SciPy packages. For instructions on how to compile the latest packages from source, see http://www.scipy.org/.
4. Install pybinding using pip:

```
pip3 install pybinding
```

macOS

All the required SciPy packages and pybinding are available as binary wheels on PyPI, so the installation is very simple:

```
pip3 install pybinding
```

Note that pip will resolve all the SciPy dependencies automatically.

Compiling from source

If you want to get the latest version (the master branch on GitHub), you will need to compile it from source code. Before you proceed, you'll need to have numpy, scipy and matplotlib. They can be installed either using Anaconda or following the procedure in the section just above this one. Once you have everything, follow the steps below to compile and install pybinding.

Windows

1. Install Visual Studio 2017 Community. The Visual C++ compiler is required, so make sure to select it during the customization step of the installation (C++ may not be installed by default).
2. Install CMake.
3. Build and install pybinding. The following command will instruct pip to download the latest source code from GitHub, compile everything and install the package:

   ```
   pip3 install git+https://github.com/dean0x7d/pybinding.git
   ```

Linux

You'll need gcc/g++ >= v5.0 (or clang >= v3.5) and CMake >= v3.1. See the previous section for details. If you have everything, pybinding can be installed from the latest source code using pip:

```
pip3 install git+https://github.com/dean0x7d/pybinding.git
```

macOS

1. Install Homebrew.
2. Install CMake: `brew install cmake`
3. Build and install pybinding. The following command will instruct pip to download the latest source code from GitHub, compile everything and install the package:

   ```
   pip3 install git+https://github.com/dean0x7d/pybinding.git
   ```

For development

If you would like to work on the pybinding source code itself, you can install it in an editable development environment. The procedure is similar to the "Compiling from source" section with the exception of the final step:

1. Clone the repository using git (you can change the url to your own GitHub fork):

   ```
   git clone --recursive https://github.com/dean0x7d/pybinding.git
   ```

2. Tell pip to install in development mode:

   ```
   cd pybinding
   pip3 install -e .
   ```

If you are new to Python/SciPy or if you're just not sure how to proceed, go with the *Quick Install* option. It will show you how to easily set up a new Python environment and install everything. That quick guide will be everything you need in most cases. However, If you would like a custom setup within your existing Python environment and you have experience compiling binary packages, you can check out the *Advanced Install* option.

CHAPTER 8

Tutorial

This section will present the essential features of pybinding with example code to get you started quickly. The tutorial assumes that you already have a basic understanding of tight-binding theory, the Python 3 programming language and at least part of the scientific Python stack (SciPy). But don't worry: while this tutorial will not specifically explain basic language and scientific Python concepts, they are presented in a straightforward way and will be easy to pick up on the fly.

Imports

This tutorial includes two kinds of example code: complete files and short code samples. Files are self-contained examples which can be downloaded and run. Code snippets are included directly within the tutorial text to illustrate features, thus they omit some common and repetitive code (like import statements) in order to save space and not distract from the main point. It is assumed that the following lines precede any other code:

```
import pybinding as pb
import numpy as np
import matplotlib.pyplot as plt

pb.pltutils.use_style()
```

The `pb` alias is always used for importing pybinding. This is similar to the common scientific package aliases: `np` and `plt`. These import conventions are used consistently in the tutorial.

The function *`pb.pltutils.use_style()`* applies pybinding's default style settings for matplotlib. This is completely optional and only affects the aesthetics of the generated figures.

Lattice

A *`Lattice`* object describes the unit cell of a crystal lattice. This includes the primitive vectors, positions of sublattice sites and hopping parameters which connect those sites. All of this structural information is used to build up a larger system by translation.

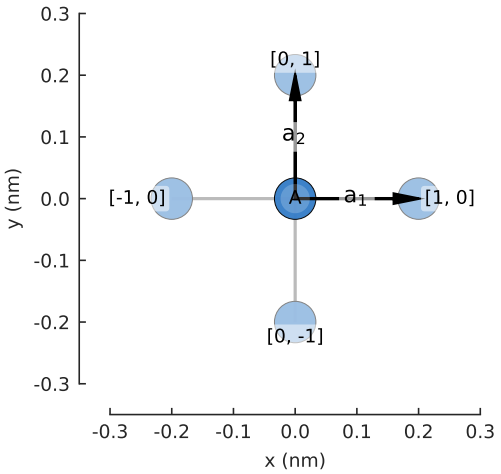
Square lattice

Starting from the basics, we'll create a simple square lattice.

```
import pybinding as pb

d = 0.2  # [nm] unit cell length
t = 1    # [eV] hopping energy

# create a simple 2D lattice with vectors a1 and a2
lattice = pb.Lattice(a1=[d, 0], a2=[0, d])
lattice.add_sublattices(
    ('A', [0, 0])  # add an atom called 'A' at position [0, 0]
)
lattice.add_hoppings(
    # (relative_index, from_sublattice, to_sublattice, energy)
    ([0, 1], 'A', 'A', t),
    ([1, 0], 'A', 'A', t)
)
```

It may not be immediately obvious what this code does. Fortunately, *Lattice* objects have a convenient *Lattice.plot()* method to easily visualize the constructed lattice.

```
lattice.plot()  # plot the lattice that was just constructed
plt.show()      # standard matplotlib show() function
```



In the figure we see lattice vectors a_1 and a_2 which were used to initialize *Lattice*. These vectors describe a Bravais lattice with an infinite set of positions,

$$\vec{R} = n_1\vec{a}_1 + n_2\vec{a}_2,$$

where n_1 and n_2 are integers. The blue circle labeled A represents the atom which was created with the *Lattice.add_sublattices()* method. The slightly faded out circles represent translations of the lattice in the primitive vector directions, i.e. using the integer index $[n_1, n_2]$.

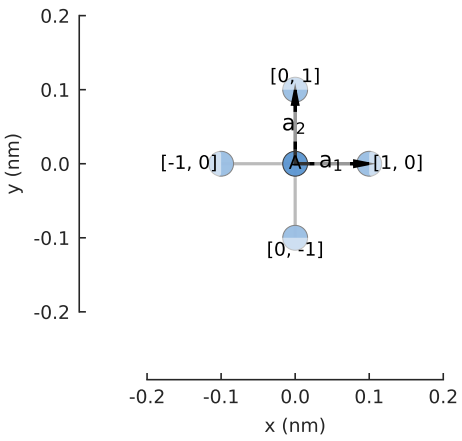
The hoppings are specified using the *Lattice.add_hoppings()* method and each one consists of `(relative_index, from_sublattice, to_sublattice, energy)`:

- The main cell always has the index $[n_1, n_2]$ = [0, 0]. The `relative_index` represents the number of integer steps needed to reach another cell starting from the main one. Each cell is labeled with its `relative_index`, as seen in the figure.
- A hopping is created between the main cell and a neighboring cell specified by `relative_index`. Two hoppings are added in the definition: [0, 1] and [1, 0]. The opposite hoppings [0, -1] and [-1, 0] are added automatically to maintain hermiticity.

- This lattice consists of only one sublattice so the `from` and `to` sublattice fields are trivial. Generally, `from_sublattice` indicates the sublattice in the [0, 0] cell and `to_sublattice` in the neighboring cell. This will be explained further in the next example.
- The last parameter is simply the value of the hopping energy.

It's good practice to build the lattice inside a function to make it easily reusable. Here we define the same lattice as before, but note that the unit cell length and hopping energy are function arguments, which makes the lattice easily configurable.

```
def square_lattice(d, t):
    lat = pb.Lattice(a1=[d, 0], a2=[0, d])
    lat.add_sublattices(('A', [0, 0]))
    lat.add_hoppings(([0, 1], 'A', 'A', t),
                     ([1, 0], 'A', 'A', t))
    return lat

# we can quickly set a shorter unit length `d`
lattice = square_lattice(d=0.1, t=1)
lattice.plot()
plt.show()
```

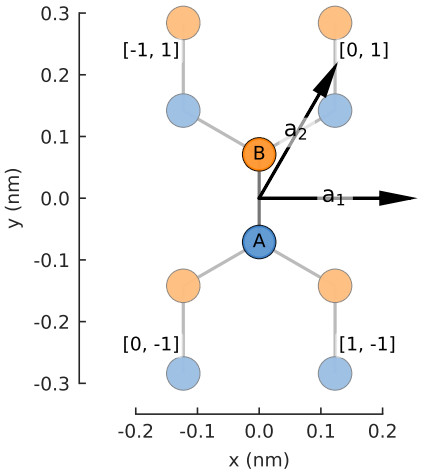


Graphene

The next example shows a slightly more complicated two-atom lattice of graphene.

```
from math import sqrt

def monolayer_graphene():
    a = 0.24595   # [nm] unit cell length
    a_cc = 0.142  # [nm] carbon-carbon distance
    t = -2.8      # [eV] nearest neighbour hopping

    lat = pb.Lattice(a1=[a, 0],
                     a2=[a/2, a/2 * sqrt(3)])
    lat.add_sublattices(('A', [0, -a_cc/2]),
                        ('B', [0,  a_cc/2]))
    lat.add_hoppings(
        # inside the main cell
        ([0,  0], 'A', 'B', t),
        # between neighboring cells
        ([1, -1], 'A', 'B', t),
        ([0, -1], 'A', 'B', t)
    )
```

```
    return lat

lattice = monolayer_graphene()
lattice.plot()
plt.show()
```



The *Lattice.add_sublattices()* method creates atoms A and B (blue and orange) at different offsets: $[0, -a_{cc}/2]$ and $[0, a_{cc}/2]$. Once again, the translated cells are given at positions $\vec{R} = n_1\vec{a}_1 + n_2\vec{a}_2$, however, this time the lattice vectors are not perpendicular which makes the integer indices $[n_1, n_2]$ slightly more complicate (see the labels in the figure).

The hoppings are defined as follows:

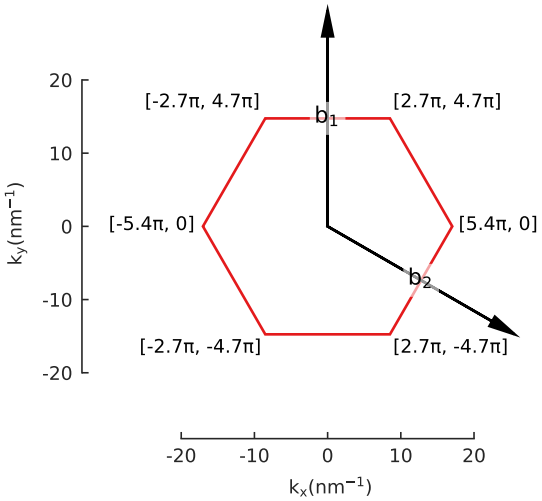
- `([0, 0], 'A', 'B', t)` specifies the hopping inside the main cell, from atom A to B. The main [0,0] cell is never labeled in the figure, but it is always the central cell where the lattice vectors originate.
- `([1, -1], 'A', 'B', t)` specifies the hopping between [0, 0] and [1, -1], from A to B. The opposite hopping is added automatically: [-1, 1], from B to A. In the tight-binding matrix representation, the opposite hopping is the Hermitian conjugate of the first one. The lattice specification always requires explicitly mentioning only one half of the hoppings while the other half is automatically added to guarantee hermiticity.
- `([0, -1], 'A', 'B', t)` is handled in the very same way.

The *Lattice.plot()* method will always faithfully draw any lattice that has been specified. It serves as a handy visual inspection tool.

Brillouin zone

The method *Lattice.plot_brillouin_zone()* is another handy tool that does just as its name implies.

```
lattice = monolayer_graphene()
lattice.plot_brillouin_zone()
```

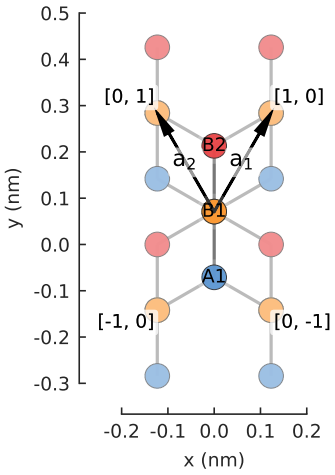


The reciprocal lattice vectors b_1 and b_2 are calculated automatically based on the real space vectors. There is no need to specify them manually. The first Brillouin zone is determined as the Wigner–Seitz cell in reciprocal space. By default, the plot method labels the vertices of the Brillouin zone.

Material repository

A few common lattices are included in pybinding's *Material Repository*. You can get started quickly by importing one of them. For example:

```
from pybinding.repository import graphene
lattice = graphene.bilayer()
lattice.plot()
```



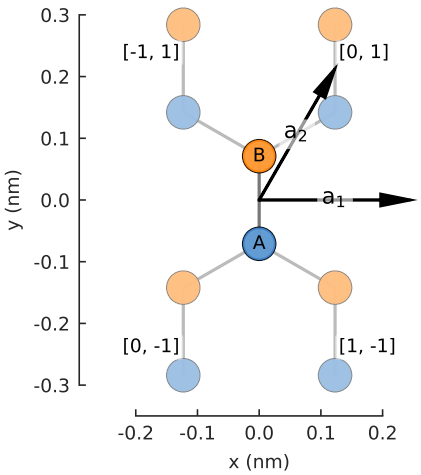
Further reading

Additional features of the `Lattice` class are explained in the *Advanced Topics* section. For more lattice specifications check out the *examples section*.

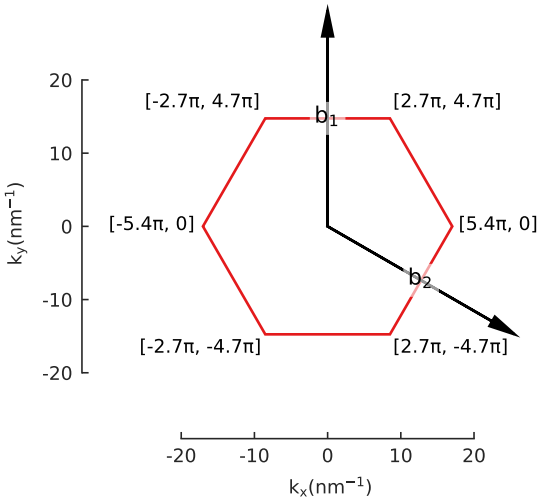
Example

```
"""Create and plot a monolayer graphene lattice and it's Brillouin zone"""
import pybinding as pb
```

```
import matplotlib.pyplot as plt
from math import sqrt

pb.pltutils.use_style()


def monolayer_graphene():
    """Return the lattice specification for monolayer graphene"""
    a = 0.24595   # [nm] unit cell length
    a_cc = 0.142  # [nm] carbon-carbon distance
    t = -2.8      # [eV] nearest neighbour hopping

    # create a lattice with 2 primitive vectors
    lat = pb.Lattice(
        a1=[a, 0],
        a2=[a/2, a/2 * sqrt(3)]
    )

    lat.add_sublattices(
        # name and position
        ('A', [0, -a_cc/2]),
        ('B', [0,  a_cc/2])
    )

    lat.add_hoppings(
        # inside the main cell
        ([0,  0], 'A', 'B', t),
        # between neighboring cells
        ([1, -1], 'A', 'B', t),
        ([0, -1], 'A', 'B', t)
    )

    return lat


lattice = monolayer_graphene()
lattice.plot()
plt.show()
```




```
lattice.plot_brillouin_zone()
plt.show()
```



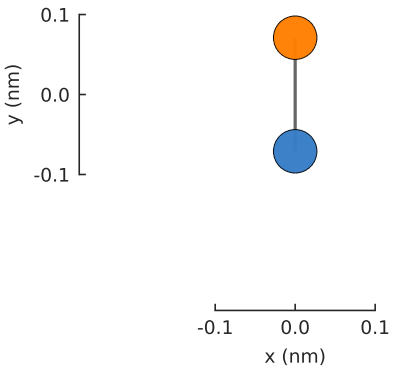
Band structure

In order to calculate the band structure of a crystal lattice, this section introduces the concepts of a *Model* and a *Solver*.

Model

A *Model* contains the full tight-binding description of the physical system that we wish to solve. We'll start by assigning a lattice to the model, and we'll use a pre-made one from the material repository.

```
from pybinding.repository import graphene

model = pb.Model(graphene.monolayer())
model.plot()
```



The result is not very exciting: just a single graphene unit cell with 2 atoms and a single hopping between them. The model does not assume translational symmetry or any other physical property. Given a lattice, it will just create a single unit cell. The model has a *System* attribute which keeps track of structural properties like the positions of lattice sites and the way they are connected, as seen in the figure above. The raw data can be accessed directly:

```
>>> model.system.x
[0, 0]
```

```
>>> model.system.y
[-0.071  0.071]
>>> model.system.sublattices
[0 1]
```

Each attribute is a 1D array where the number of elements is equal to the total number of lattice sites in the system. The model also has a *hamiltonian* attribute:

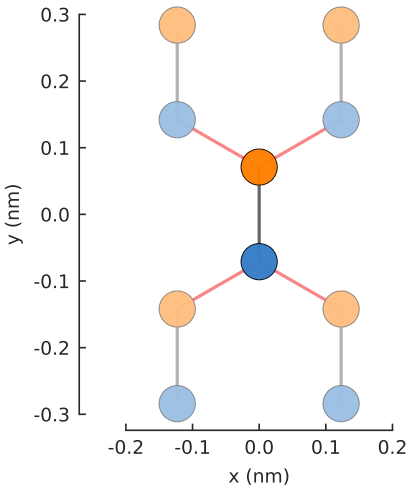
```
>>> model.hamiltonian
(0, 1)    -2.8
(1, 0)    -2.8
```

It's a sparse matrix (see scipy.sparse.csr_matrix) which corresponds to the tight-binding Hamiltonian of our model. The output above shows the default sparse representation of the data where each line corresponds to `(row, col) value`. Alternatively, we can see the dense matrix output:

```
>>> model.hamiltonian.todense()
[[ 0.0 -2.8]
 [-2.8  0.0]]
```

Next, we include *translational_symmetry()* to create an infinite graphene sheet.

```
model = pb.Model(
    graphene.monolayer(),
    pb.translational_symmetry()
)
model.plot()
```



The red lines indicate hoppings on periodic boundaries. The lighter colored circles represent the translations of the unit cell. The number of translations is infinite, but the plot only presents the first one in each lattice vector direction.

Solver

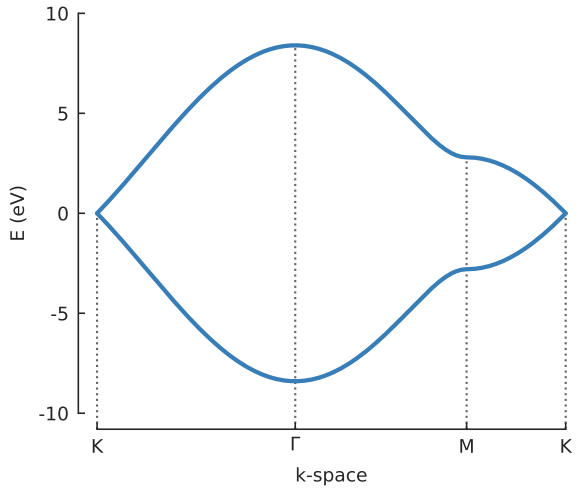
A *Solver* can exactly calculate the eigenvalues and eigenvectors of a Hamiltonian matrix. We'll take a look at various *Eigenvalue solvers* and their capabilities in a later section, but right now we'll just grab the *lapack()* solver which is the simplest and most appropriate for small systems.

```
>>> model = pb.Model(graphene.monolayer())
>>> solver = pb.solver.lapack(model)
>>> solver.eigenvalues
[-2.8  2.8]
>>> solver.eigenvectors
```

```
[[-0.707 -0.707]
 [-0.707  0.707]]
```

Beyond just the *eigenvalues* and *eigenvectors* properties, *Solver* has a convenient *calc_bands()* method which can be used to calculate the band structure of our model.

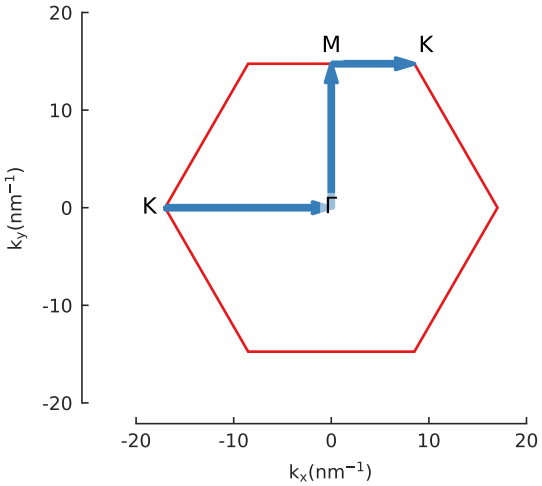
```
from math import sqrt, pi

model = pb.Model(graphene.monolayer(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

a_cc = graphene.a_cc
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



The points Γ, K and M are used to draw a path in the reciprocal space of graphene's Brillouin zone and *Solver.calc_bands()* calculates the band energy along that path. The return value of the method is a *Bands* result object.

All result objects have built-in plotting methods. Aside from the basic *plot()* seen above, *Bands* also has *plot_kpath()* which presents the path in reciprocal space. Plots can easily be composed, so to see the path in the context of the Brillouin zone, we can simply plot both:

```
model.lattice.plot_brillouin_zone(decorate=False)
bands.plot_kpath(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



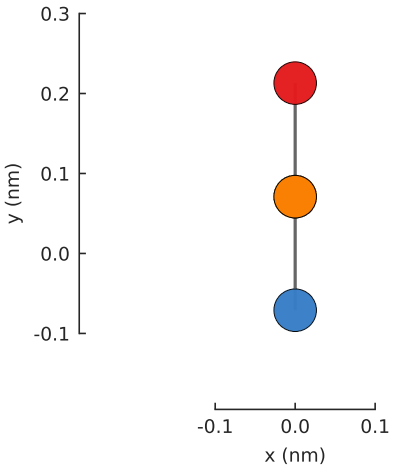
The extra argument for *Lattice.plot_brillouin_zone()* turns off the reciprocal lattice vectors and vertex coordinate labels (as seen in the previous section).

Note: The band structure along a path in k-space can also be calculated manually by saving an array of *Solver.eigenvalues* at different k-points. This process is shown on the *Eigensolver* page.

Switching lattices

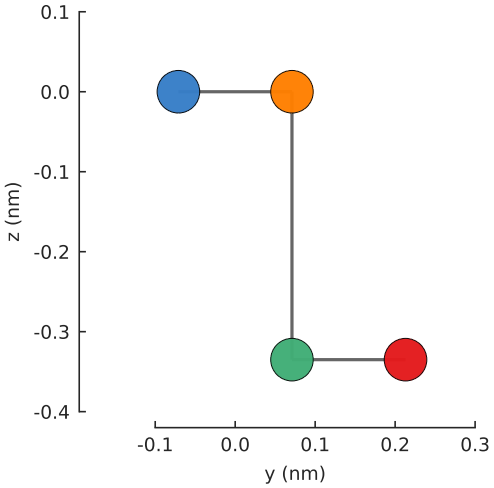
We can easily switch to a different material, just by passing a different lattice to the model. For this example, we'll use our pre-made `graphene.bilayer()` from the *Material Repository*. But you can create any lattice as described in the previous section: *Lattice*.

```
model = pb.Model(graphene.bilayer())
model.plot()
```



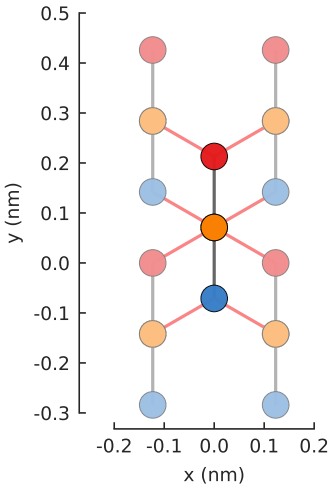
Without *translational_symmetry()*, the model is just a single unit cell with 4 atoms. Our bilayer lattice uses AB-stacking where a pair of atoms are positioned one on top of the another. By default, the *Model.plot()* method shows the xy-plane, so one of the bottom atoms isn't visible. We can pass an additional plot argument to see the yz-plane:

```
model = pb.Model(graphene.bilayer())
model.plot(axes='yz')
```



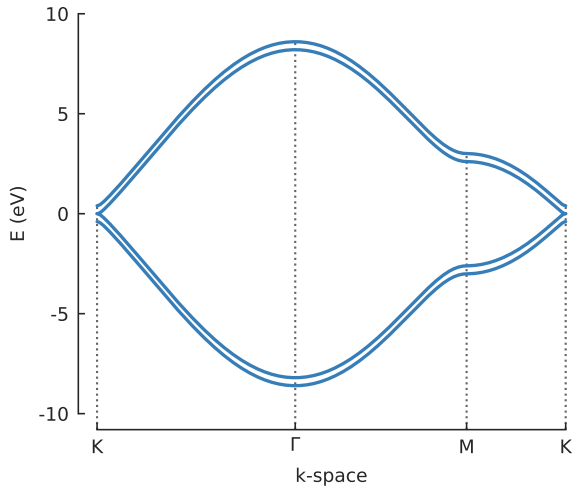
To compute the band structure, we'll need to include *translational_symmetry()*.

```
model = pb.Model(graphene.bilayer(), pb.translational_symmetry())
model.plot()
```



As before, the red hoppings indicate periodic boundaries and the lighter colored circles represent the first of an infinite number of translation units. We'll compute the band structure for the same Γ, K and M points as monolayer graphene:

```
solver = pb.solver.lapack(model)
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```

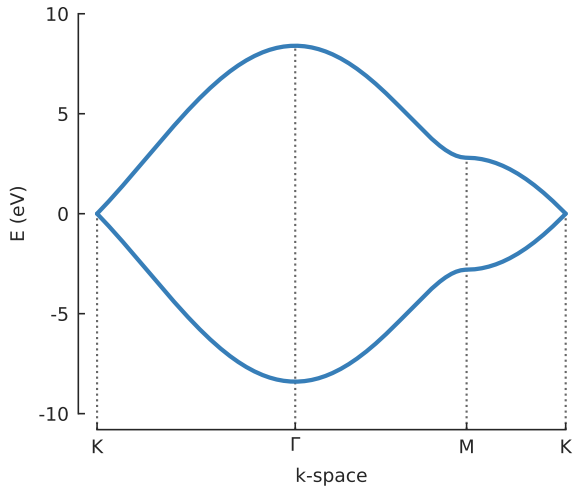


Further reading

Check out the *examples section* for more band structure calculations with various other lattices. *Eigenvalue solvers* will be covered in more detail at a later point in the tutorial, but this is enough information to get started. The next few sections are going to be dedicated to model building.

Example

```
"""Calculate and plot the band structure of monolayer graphene"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import sqrt, pi
from pybinding.repository import graphene

pb.pltutils.use_style()


model = pb.Model(
    graphene.monolayer(),  # predefined lattice from the material repository
    pb.translational_symmetry()    # creates an infinite sheet of graphene
)
solver = pb.solver.lapack(model)  # eigensolver from the LAPACK library

# significant points in graphene's Brillouin zone
a_cc = graphene.a_cc  # carbon-carbon distance
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

# plot the bands through the desired points
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
plt.show()
```



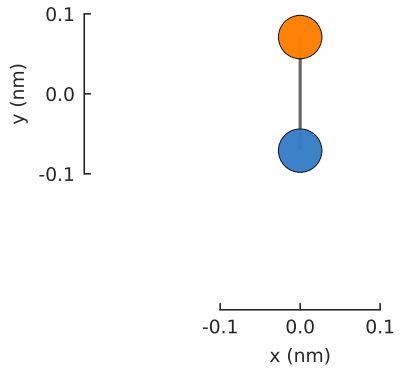
Finite size

This section introduces the concept of shapes with classes *Polygon* and *FreeformShape* which are used to model systems of finite size. The sparse eigensolver *arpack()* is also introduced as a good tool for exactly solving larger Hamiltonian matrices.

Primitive

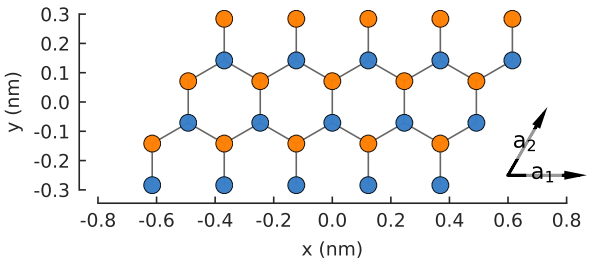
The simplest finite-sized system is just the unit cell of the crystal lattice.

```
from pybinding.repository import graphene

model = pb.Model(graphene.monolayer())
model.plot()
```



The unit cell can also be replicated a number of times to create a bigger system.

```
model = pb.Model(
    graphene.monolayer(),
    pb.primitive(a1=5, a2=3)
)
model.plot()
model.lattice.plot_vectors(position=[0.6, -0.25])
```

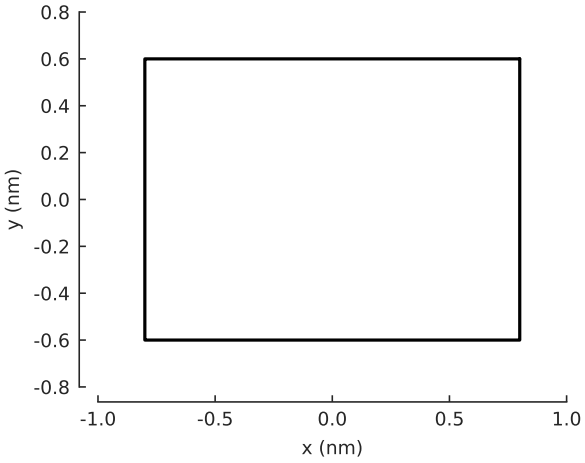


The *primitive()* parameter tells the model to replicate the unit cell 5 times in the a_1 vector direction and 3 times in the a_2 direction. However, to model realistic systems we need proper shapes.

Polygon

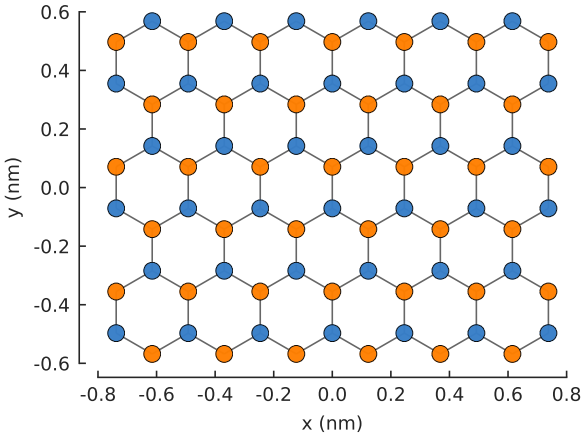
The easiest way to create a 2D shape is with the *Polygon* class. For example, a simple rectangle:

```
def rectangle(width, height):
    x0 = width / 2
    y0 = height / 2
    return pb.Polygon([[x0, y0], [x0, -y0], [-x0, -y0], [-x0, y0]])

shape = rectangle(1.6, 1.2)
shape.plot()
```



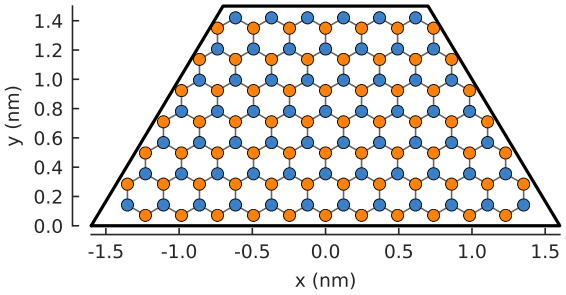
A *Polygon* is initialized with a list of vertices which should be given in clockwise or counterclockwise order. When added to a *Model* the lattice will expand to fill the shape.

```
model = pb.Model(
    graphene.monolayer(),
    rectangle(width=1.6, height=1.2)
)
model.plot()
```



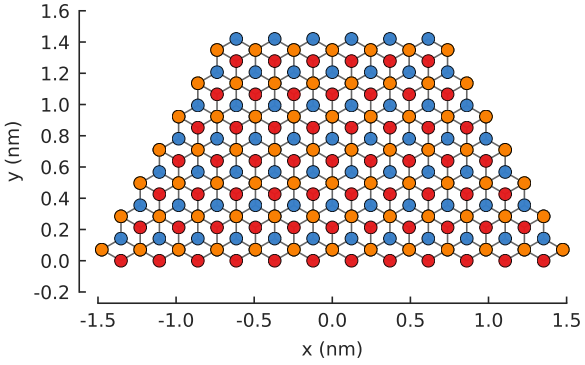
To help visualize the shape and the expanded lattice, the polygon outline can be plotted on top of the system by calling both plot methods one after another.

```
def trapezoid(a, b, h):
    return pb.Polygon([[-a/2, 0], [-b/2, h], [b/2, h], [a/2, 0]])

model = pb.Model(
    graphene.monolayer(),
    trapezoid(a=3.2, b=1.4, h=1.5)
)
model.plot()
model.shape.plot()
```



In general, a shape does not depend on a specific material, so it can be easily reused. Here, we shall switch to a `graphene.bilayer()` lattice, but we'll keep the same `trapezoid` shape as defined earlier:

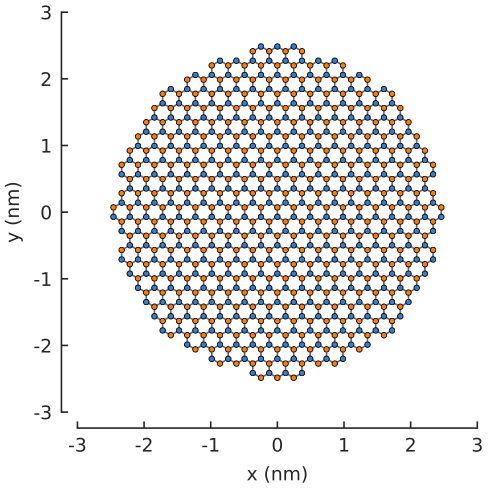
```
model = pb.Model(
    graphene.bilayer(),
    trapezoid(a=3.2, b=1.4, h=1.5)
)
model.plot()
```



Freeform shape

Unlike a *Polygon* which is defined by a list of vertices, a *FreeformShape* is defined by a `contains` function which determines if a lattice site is inside the desired shape.

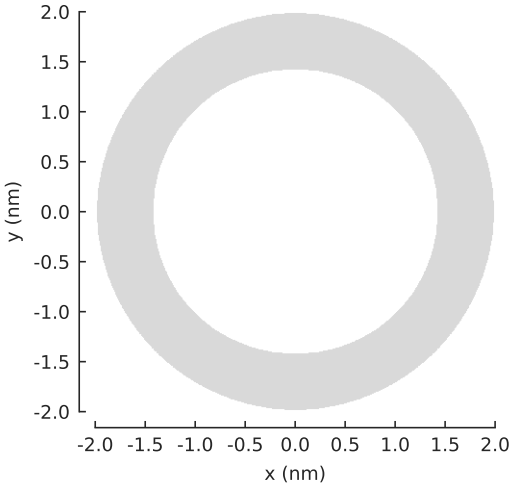
```
def circle(radius):
    def contains(x, y, z):
        return np.sqrt(x**2 + y**2) < radius
    return pb.FreeformShape(contains, width=[2*radius, 2*radius])

model = pb.Model(
    graphene.monolayer(),
    circle(radius=2.5)
)
model.plot()
```



The `width` parameter of *FreeformShape* specifies the bounding box width. Only sites inside the bounding box will be considered for the shape. It's like carving a sculpture from a block of stone. The bounding box can be thought of as the stone block, while the `contains` function is the carving tool that can give the fine detail of the shape.

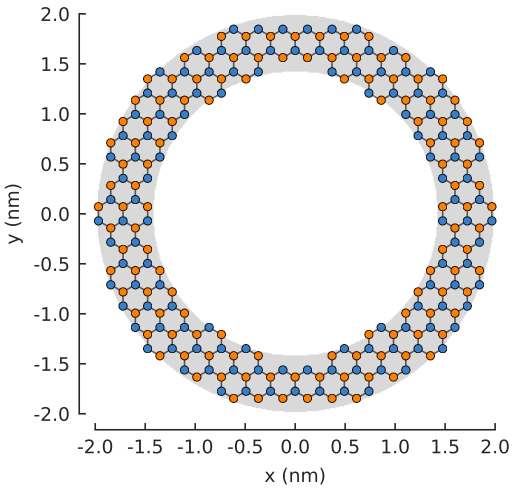
As with *Polygon*, we can visualize the shape with the *FreeformShape.plot()* method.

```
def ring(inner_radius, outer_radius):
    def contains(x, y, z):
        r = np.sqrt(x**2 + y**2)
        return np.logical_and(inner_radius < r, r < outer_radius)
    return pb.FreeformShape(contains, width=[2*outer_radius, 2*outer_radius])

shape = ring(inner_radius=1.4, outer_radius=2)
shape.plot()
```



The shaded area indicates the shape as determined by the `contains` function. Creating a model will cause the lattice to fill in the shape.

```
model = pb.Model(
    graphene.monolayer(),
    ring(inner_radius=1.4, outer_radius=2)
)
model.plot()
model.shape.plot()
```



Note that the `ring` example uses `np.logical_and` instead of the plain `and` keyword. This is because the `x, y, z` positions are not given as scalar numbers but as numpy arrays. Array comparisons return boolean arrays:

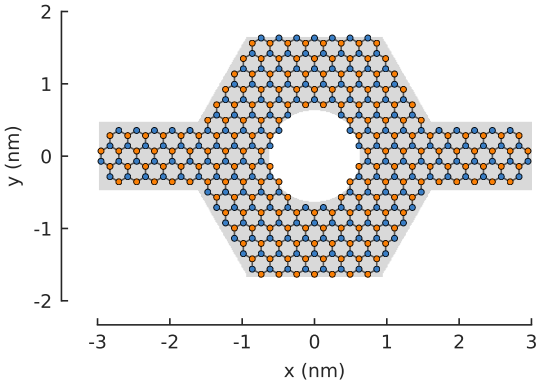
```
>>> x = np.array([7, 2, 3, 5, 1])
>>> x < 5
[False, True, True, False, True]
>>> 2 < x and x < 5
ValueError: ...
>>> np.logical_and(2 < x, x < 5)
[False, False, True, False, False]
```

The `and` keyword can only operate on scalar values, but `np.logical_and` can consider arrays. Likewise, `math.sqrt` does not work with arrays, but `np.sqrt` does.

Composite shape

Complicated system geometry can also be produced by composing multiple simple shapes. The following example gives a quick taste of how it works. For a full overview of this functionality, see the *Composite shapes* section.

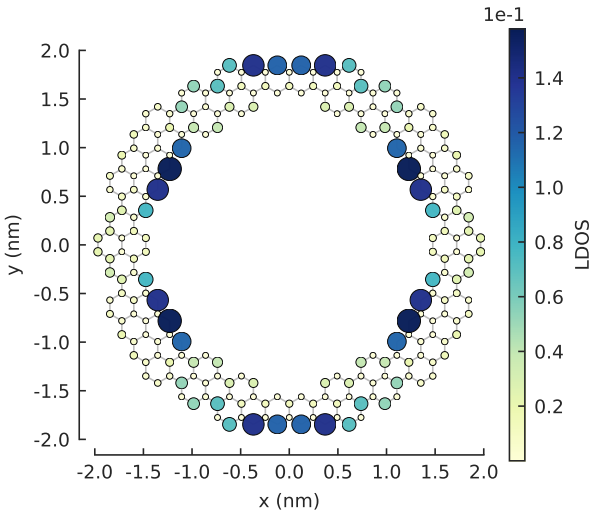
```
# Simple shapes
rectangle = pb.rectangle(x=6, y=1)
hexagon = pb.regular_polygon(num_sides=6, radius=1.92, angle=np.pi/6)
circle = pb.circle(radius=0.6)

# Compose them naturally
shape = rectangle + hexagon - circle

model = pb.Model(graphene.monolayer(), shape)
model.shape.plot()
model.plot()
```



Spatial LDOS

Now that we have a ring structure, we can exactly diagonalize its `model.hamiltonian` using a *`Solver`*. We previously used the *`lapack()`* solver to find all the eigenvalues and eigenvectors, but this is not efficient for larger systems. The sparse *`arpack()`* solver can calculate a targeted subset of the eigenvalues, which is usually desired and much faster. In this case, we are interested only in the 20 lowest energy states.

```
model = pb.Model(
    graphene.monolayer(),
    ring(inner_radius=1.4, outer_radius=2)
)
solver = pb.solver.arpack(model, k=20)  # only the 20 lowest eigenstates

ldos = solver.calc_spatial_ldos(energy=0, broadening=0.05)  # eV
ldos.plot(site_radius=(0.03, 0.12))
pb.pltutils.colorbar(label="LDOS")
```



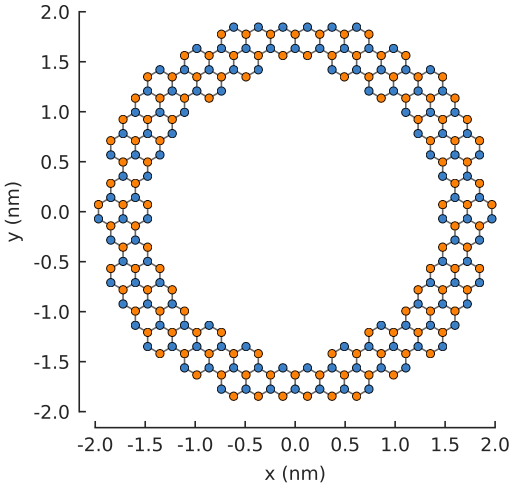
The convenient *Solver.calc_spatial_ldos()* method calculates the local density of states (LDOS) at every site for the given energy with a Gaussian broadening. The returned object is a *StructureMap* which holds the LDOS data. The *StructureMap.plot()* method will produce a figure similar to *Model.plot()*, but with a colormap indicating the LDOS value at each lattice site. In addition, the `site_radius` argument specifies a range of sizes which will cause the low intensity sites to appear as small circles while high intensity ones become large. The states with a high LDOS are clearly visible on the outer and inner edges of the graphene ring structure.

Further reading

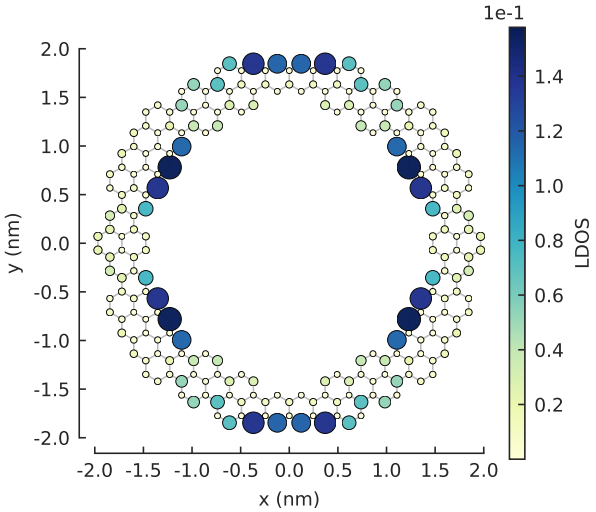
For more finite-sized systems check out the *examples section*.

Example

```
"""Model a graphene ring structure and calculate the local density of states"""
import pybinding as pb
import numpy as np
import matplotlib.pyplot as plt
from pybinding.repository import graphene

pb.pltutils.use_style()


def ring(inner_radius, outer_radius):
    """A simple ring shape"""
    def contains(x, y, z):
        r = np.sqrt(x**2 + y**2)
        return np.logical_and(inner_radius < r, r < outer_radius)

    return pb.FreeformShape(contains, width=[2 * outer_radius, 2 * outer_radius])


model = pb.Model(
    graphene.monolayer(),
    ring(inner_radius=1.4, outer_radius=2)  # length in nanometers
)

model.plot()
plt.show()
```




```
# only solve for the 20 lowest energy eigenvalues
solver = pb.solver.arpack(model, k=20)
ldos = solver.calc_spatial_ldos(energy=0, broadening=0.05)  # LDOS around 0 eV
ldos.plot(site_radius=(0.03, 0.12))
pb.pltutils.colorbar(label="LDOS")
plt.show()
```



Shape and symmetry

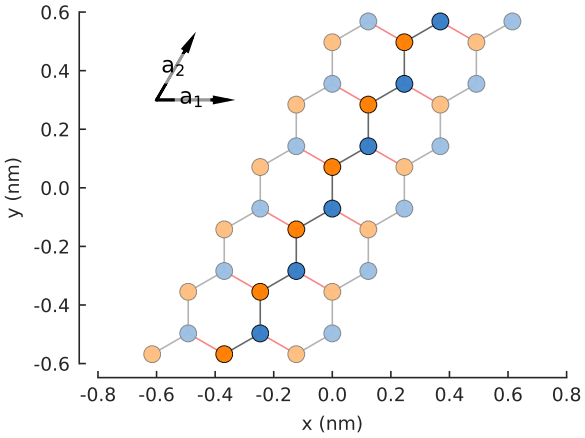
The last two sections showed how to model shape and symmetry individually, but we can be more creative and combine the two.

Nanoribbons

To create a graphene nanoribbon, we'll need a shape to give the finite width of the ribbon while the infinite length is achieved by imposing translational symmetry.

```
from pybinding.repository import graphene

model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),  # nm
```

```
    pb.translational_symmetry(a1=True, a2=False)
)
model.plot()
model.lattice.plot_vectors(position=[-0.6, 0.3])  # nm
```

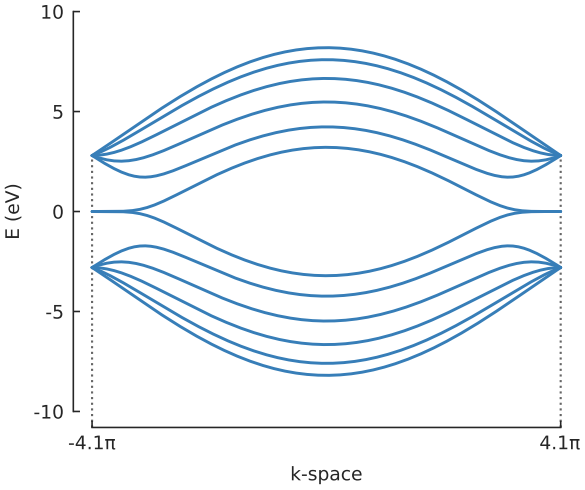


As before, the central darker circles represent the main cell of the nanoribbon, the lighter colored circles are the translations due to symmetry and the red lines are boundary hoppings. The two arrows in the upper left corner show the primitive lattice vectors of graphene.

The *translational_symmetry()* is applied only in the a_1 lattice vector direction which gives the ribbon its infinite length, but the symmetry is disabled in the a_2 direction so that the finite size of the shape is preserved. The builtin *rectangle()* shape gives the nanoribbon its 1.2 nm width.

The band structure calculations work just as before.

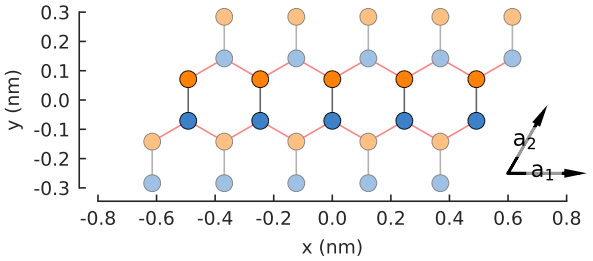
```
from math import pi, sqrt

solver = pb.solver.lapack(model)
a = graphene.a_cc * sqrt(3)  # ribbon unit cell length
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
```



This is the characteristic band structure for zigzag nanoribbons with zero-energy edge states. If we change the direction of the translational symmetry to a_2, the orientation will change, but we will still have a zigzag nanoribbon.

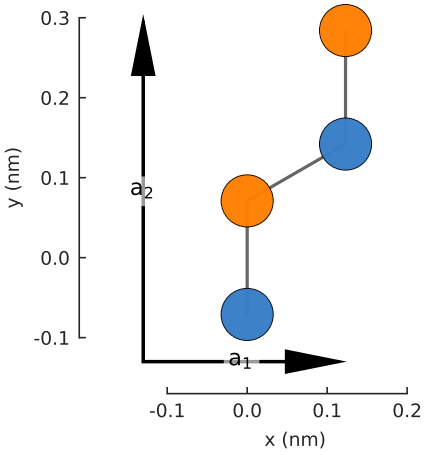
```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),  # nm
    pb.translational_symmetry(a1=False, a2=True)
```

```
)
model.plot()
model.lattice.plot_vectors(position=[0.6, -0.25])   # nm
```



Because of the nature of graphene's 2-atom unit cell and lattice vector, only zigzag edges can be created. In order to create armchair edges, we must introduce a different unit cell with 4 atoms.

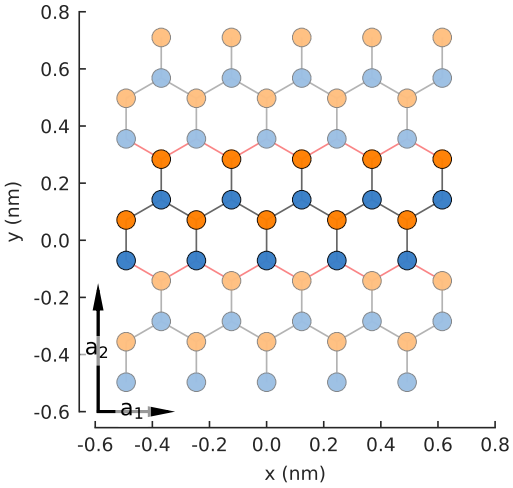
```
model = pb.Model(graphene.monolayer_4atom())
model.plot()
model.lattice.plot_vectors(position=[-0.13, -0.13])
```



Note: To learn how to create this 4-atom unit cell, see *Constructing a supercell*.

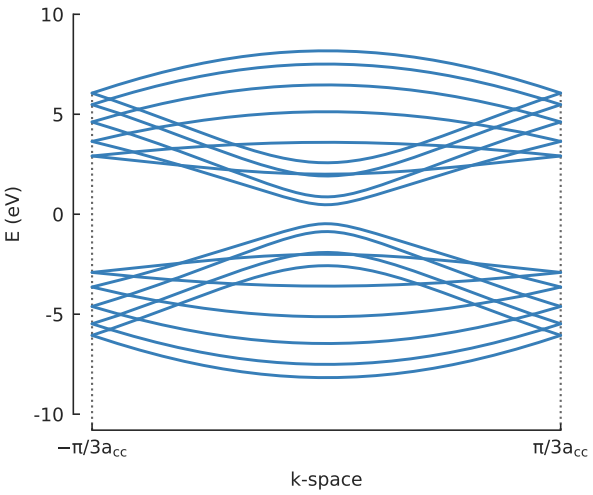
Notice that the lattice vectors a_1 and a_2 are at a right angle, unlike the sharp angle of the base 2-atom cell. The lattice properties are identical for the 2 and 4 atom cells, but the new geometry helps to create armchair edges.

```
model = pb.Model(
    graphene.monolayer_4atom(),
    pb.primitive(a1=5),
    pb.translational_symmetry(a1=False, a2=True)
)
model.plot()
model.lattice.plot_vectors(position=[-0.59, -0.6])
```



To calculate the band structure we must enter at least two points in k-space between which the energy will be calculated. Note that because the periodicity is in the direction of the second lattice vector a_2, the points in k-space are given as `[0, pi/d]` instead of just `pi/d` (which would be equivalent to `[pi/d, 0]`).

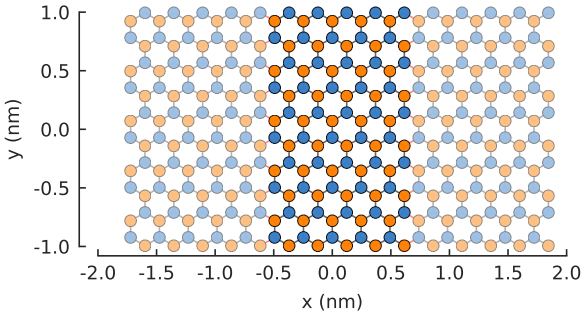
```
solver = pb.solver.lapack(model)
d = 3 * graphene.a_cc  # ribbon unit cell length
bands = solver.calc_bands([0, -pi/d], [0, pi/d])
bands.plot(point_labels=['$-\pi / 3 a_{cc}$', '$\pi / 3 a_{cc}$'])
```



1D periodic supercell

Up to now, we used *translational_symmetry()* with `True` or `False` parameters to enable or disable periodicity in certain directions. We can also pass a number to indicate the desired period length.

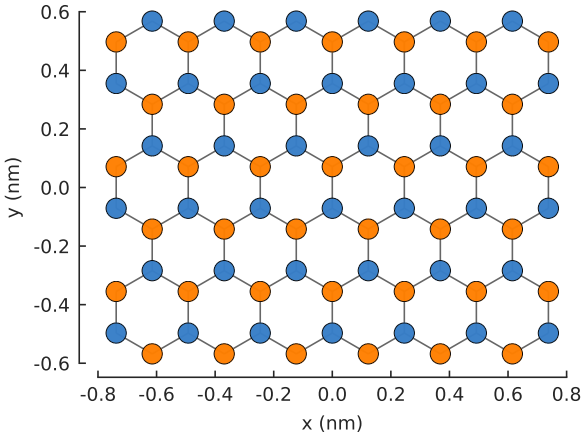
```
model = pb.Model(
    graphene.monolayer_4atom(),
    pb.rectangle(x=2, y=2),
    pb.translational_symmetry(a1=1.2, a2=False)
)
model.plot()
```



The period length is given in nanometers. Note that our base shape is a square with 2 nm sides. The base shape forms the supercell of the periodic structure, but because the period length (1.2 nm) is smaller than the shape (2 nm), the extra length is cut off by the periodic boundary.

If you specify a periodic length which is larger than the base shape, the periodic conditions will not be applied because the periodic boundary will not have anything to bind to.

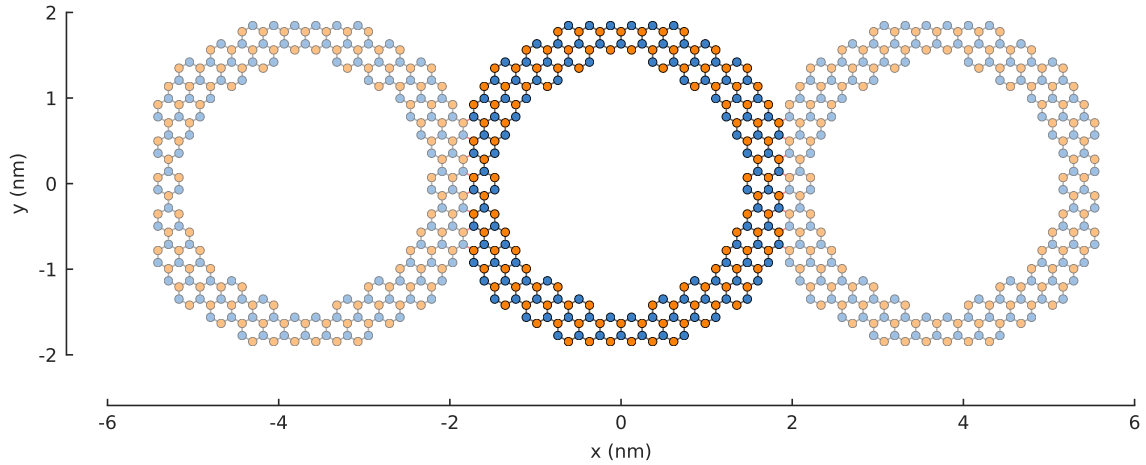
```
model = pb.Model(
    graphene.monolayer_4atom(),
    pb.rectangle(x=1.5, y=1.5),  # don't combine a small shape
    pb.translational_symmetry(a1=1.7, a2=False)  # with large period length
)
model.plot()
```



As you can see, making the period larger than the shape (1.7 nm vs. 1.5 nm), results in just the finite-sized part of the system. Don't do this.

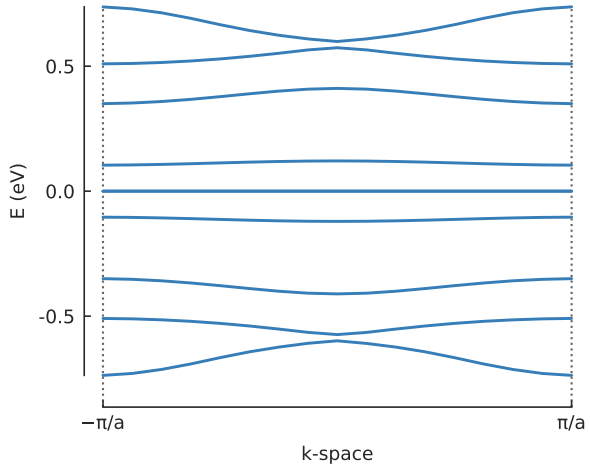
The combination of shape and symmetry can be more complex as shown here with a nanoribbon ring structure.

```
def ring(inner_radius, outer_radius):
    """Ring shape defined by an inner and outer radius"""
    def contains(x, y, z):
        r = np.sqrt(x**2 + y**2)
        return np.logical_and(inner_radius < r, r < outer_radius)
    return pb.FreeformShape(contains, width=[2*outer_radius, 2*outer_radius])

model = pb.Model(
    graphene.monolayer_4atom(),
    ring(inner_radius=1.4, outer_radius=2),
    pb.translational_symmetry(a1=3.8, a2=False)
)
plt.figure(figsize=[8, 3])
model.plot()
```



The period length of the translation in the a_1 direction is set to 3.8 nm. This ensures that the inner ring shape is preserved and the periodic boundaries are placed on the outer edges.

```
solver = pb.solver.arpack(model, k=10) # only the 10 lowest states
a = 3.8  # [nm] unit cell length
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot(point_labels=['$-\pi / a$', '$\pi / a$'])
```

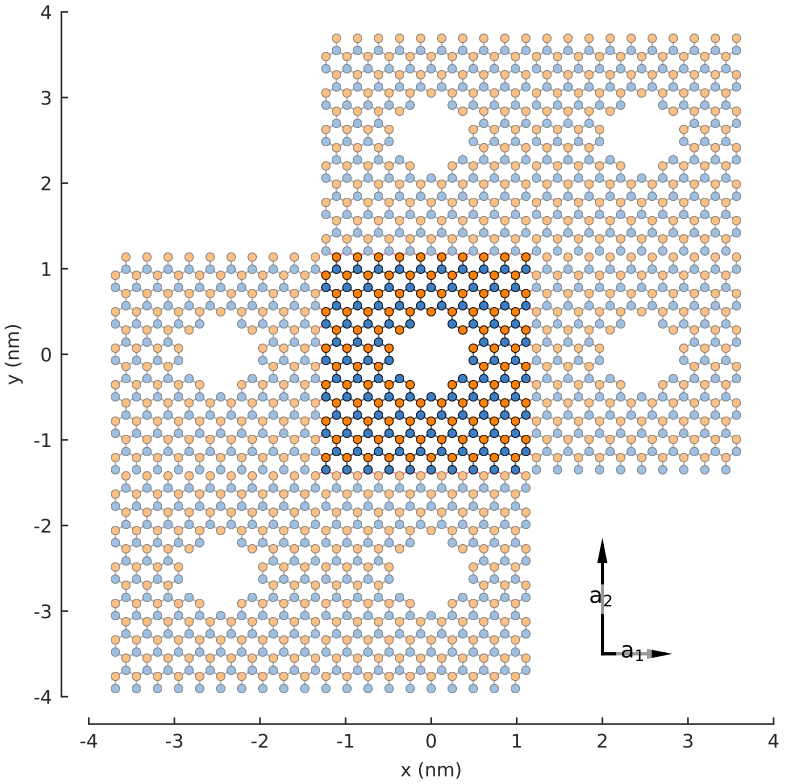


2D periodic supercell

A 2D periodic system made up of just a primitive cell was already covered in the *Band structure* section. Here, we'll create a system with a periodic unit cell which is larger than the primitive cell. Similar to the 1D case, this is accomplished by giving `translational_symmetry()` specific lengths for the translation directions. As an example, we'll take a look at a graphene antidot superlattice:

```
width = 2.5
rectangle = pb.rectangle(x=width * 1.2, y=width * 1.2)
dot = pb.circle(radius=0.4)

model = pb.Model(
    graphene.monolayer_4atom(),
    rectangle - dot,
    pb.translational_symmetry(a1=width, a2=width)
)
plt.figure(figsize=(5, 5))
```

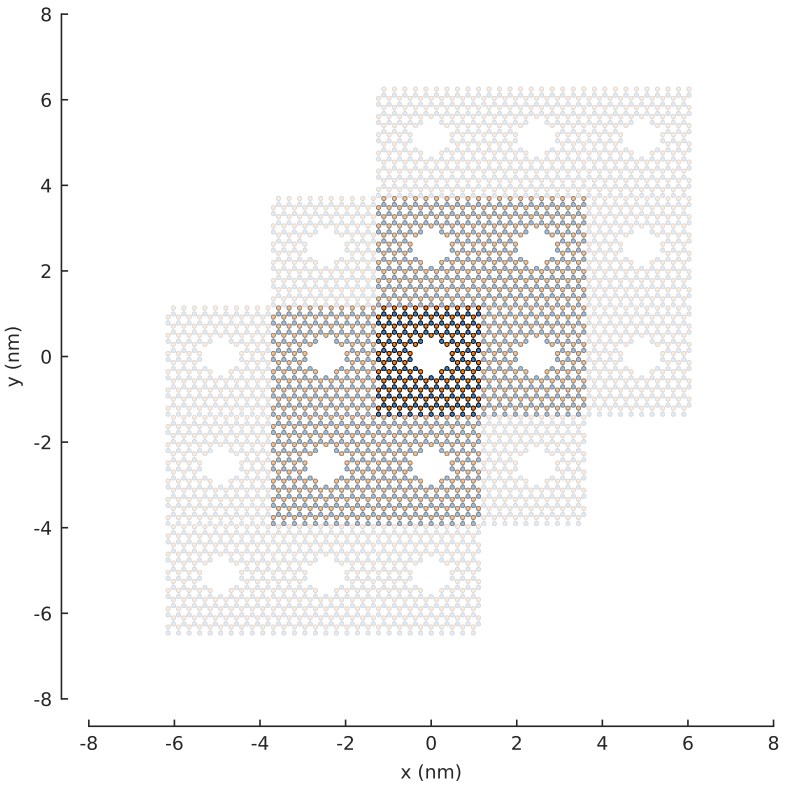
```
model.plot()
model.lattice.plot_vectors(position=[2, -3.5], scale=3)
```



The antidot unit cell is created using a *composite shape*. Note that the width of the rectangle is made to be slightly larger than the period length. Just like the 1D case, this is necessary in order to give *translational_symmetry()* some room to cut off the edges of the system and create periodic boundaries as needed. If the unit cell size is smaller then the period length, translational symmetry cannot be applied.

In the figure above, notice that 6 translations of the unit cell are presented and it appears as if 2 are missing. This is only in appearance. By default, *Model.plot()* shows just the first-nearest translations of the unit cell. It just so happens that the 2 which appear missing are second-nearest translations. To see this in the figure, we can set the num_periods argument to a higher value:

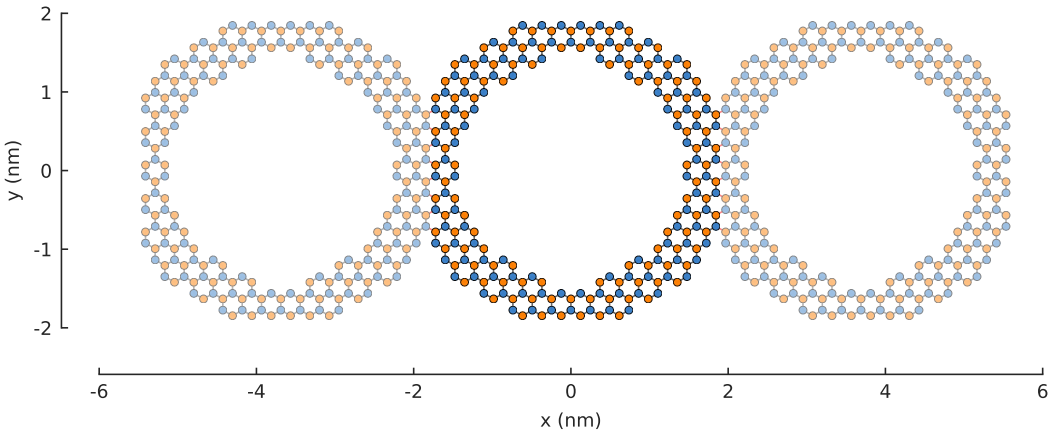
```
plt.figure(figsize=(5, 5))
model.plot(num_periods=2)
```



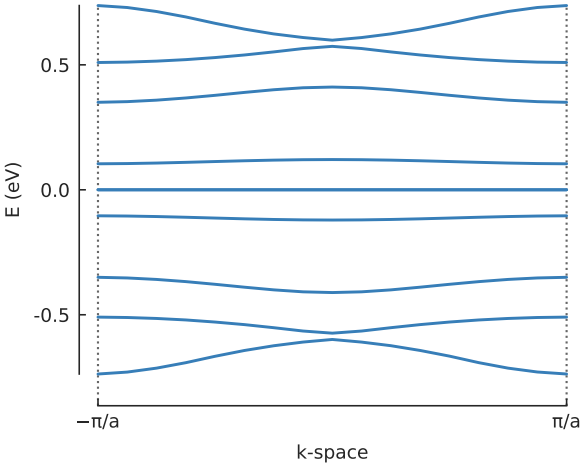
Example

Note the zero-energy mode in the band structure. For wave vector $k = 0$, states on the outer edge of the ring have the highest LDOS intensity, but for $k = \pi/a$ the inner edge states dominate.

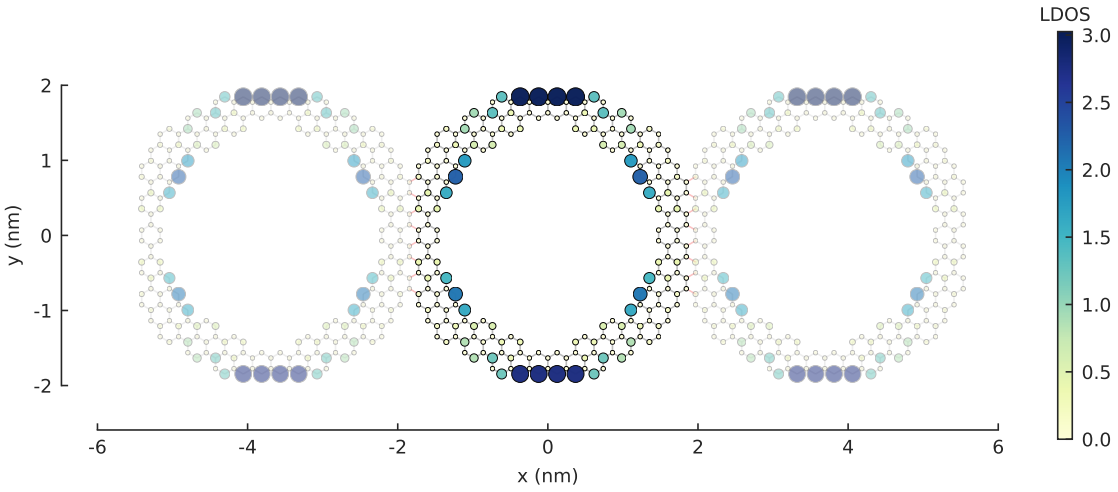
```
"""Model an infinite nanoribbon consisting of graphene rings"""
import pybinding as pb
import numpy as np
import matplotlib.pyplot as plt
from pybinding.repository import graphene
from math import pi

pb.pltutils.use_style()


def ring(inner_radius, outer_radius):
    """A simple ring shape"""
    def contains(x, y, z):
        r = np.sqrt(x**2 + y**2)
        return np.logical_and(inner_radius < r, r < outer_radius)

    return pb.FreeformShape(contains, width=[2 * outer_radius, 2 * outer_radius])


model = pb.Model(
    graphene.monolayer_4atom(),
    ring(inner_radius=1.4, outer_radius=2),  # length in nanometers
    pb.translational_symmetry(a1=3.8, a2=False)  # period in nanometers
)

plt.figure(figsize=pb.pltutils.cm2inch(20, 7))
model.plot()
plt.show()
```



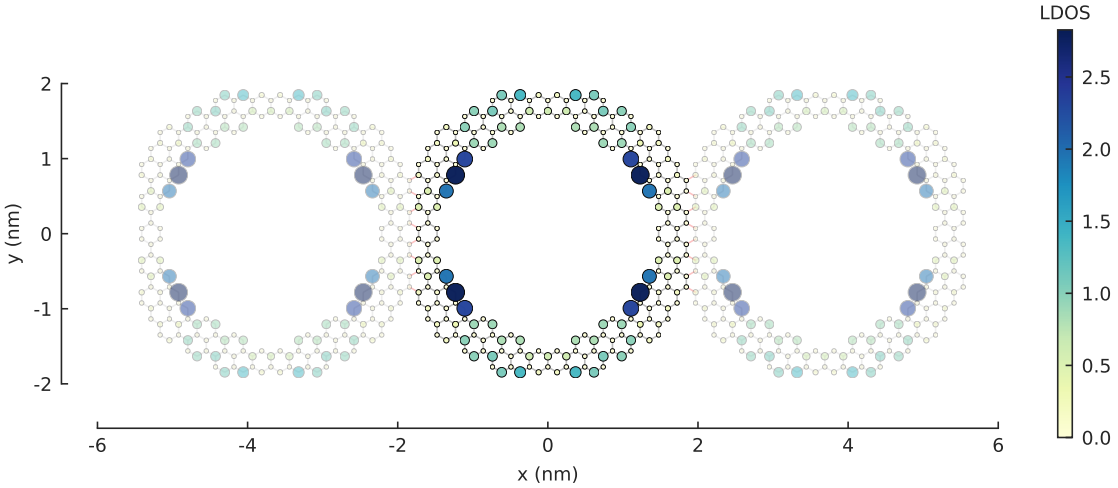

```
# only solve for the 10 lowest energy eigenvalues
solver = pb.solver.arpack(model, k=10)
a = 3.8  # [nm] unit cell length
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot(point_labels=[r'$-\pi / a$', r'$\pi / a$'])
plt.show()
```




```
solver.set_wave_vector(k=0)
ldos = solver.calc_spatial_ldos(energy=0, broadening=0.01)  # LDOS around 0 eV

plt.figure(figsize=pb.pltutils.cm2inch(20, 7))
ldos.plot(site_radius=(0.03, 0.12))
pb.pltutils.colorbar(label="LDOS")
plt.show()
```




```
solver.set_wave_vector(k=pi/a)
ldos = solver.calc_spatial_ldos(energy=0, broadening=0.01)  # LDOS around 0 eV

plt.figure(figsize=pb.pltutils.cm2inch(20, 7))
ldos.plot(site_radius=(0.03, 0.12))
pb.pltutils.colorbar(label="LDOS")
plt.show()
```



Fields and effects

This section will introduce `@onsite_energy_modifier` and `@hopping_energy_modifier` which can be used to add various fields to the model. These functions can apply user-defined modifications to the Hamiltonian matrix which is why we shall refer to them as *modifier* functions.

Electric potential

We can define a simple potential function like the following:

```
@pb.onsite_energy_modifier
def potential(x, y):
    return np.sin(x)**2 + np.cos(y)**2
```

Here `potential` is just a regular Python function, but we attached a pretty `@` decorator to it. The `@onsite_energy_modifier` decorator gives an ordinary function a few extra properties which we'll talk

about later. For now, just keep in mind that this is required to mark a function as a *modifier* for use with pybinding models. The `x` and `y` arguments are lattice site positions and the return value is the desired potential. Note the use of `np.sin` instead of `math.sin`. The `x` and `y` coordinates are `numpy` arrays, not individual numbers. This is true for all modifier arguments in pybinding. When you write modifier functions, make sure to always use `numpy` operations which work with arrays, unlike regular `math`.

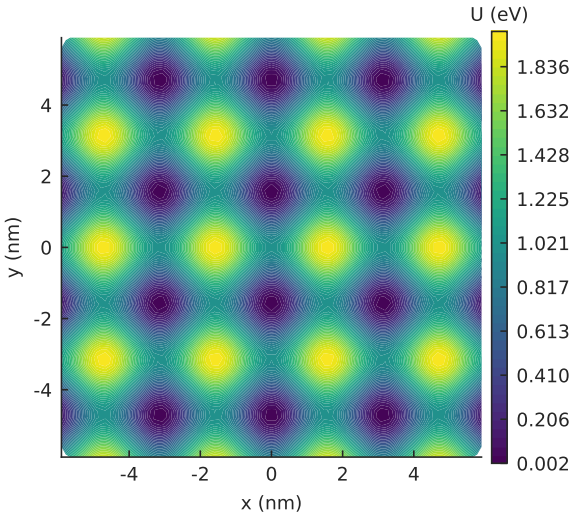
Note: Modifier arguments are passed as arrays for performance. Working with individual numbers would require calling the potential function individually for each lattice site which would be extremely slow. Arrays are much faster.

To use the potential function, just place it in a *Model* parameter list:

```
from pybinding.repository import graphene

model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(12),
    potential
)
```

To visualize the potential, there's the handy *Model.onsite_map* property which is a *StructureMap* of the onsite energy of the Hamiltonian matrix.

```
model.onsite_map.plot_contourf()
pb.pltutils.colorbar(label="U (eV)")
```

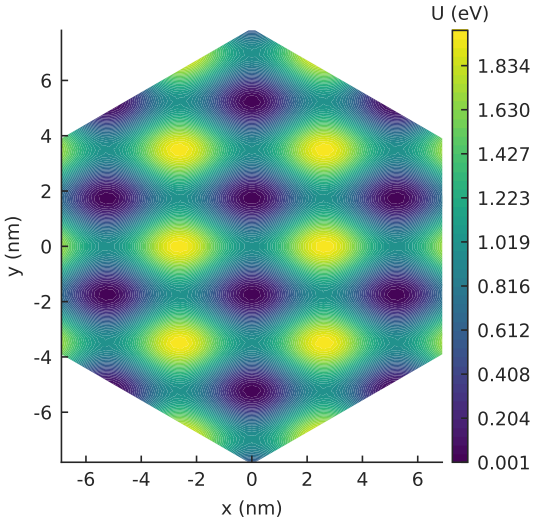


The figure shows a 2D colormap representation of our wavy potential in a square system. The *StructureMap.plot_contourf()* method we just called is implemented in terms of matplotlib's `contourf` function with some slight adjustments for convenience.

To make the potential more flexible, it's a good idea to enclose it in an outer function, just like this:

```
def wavy(a, b):
    @pb.onsite_energy_modifier
    def potential(x, y):
        return np.sin(a * x)**2 + np.cos(b * y)**2
    return potential

model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=6, radius=8),
    wavy(a=0.6, b=0.9)
)
```

```
model.onsite_map.plot_contourf()
pb.pltutils.colorbar(label="U (eV)")
```



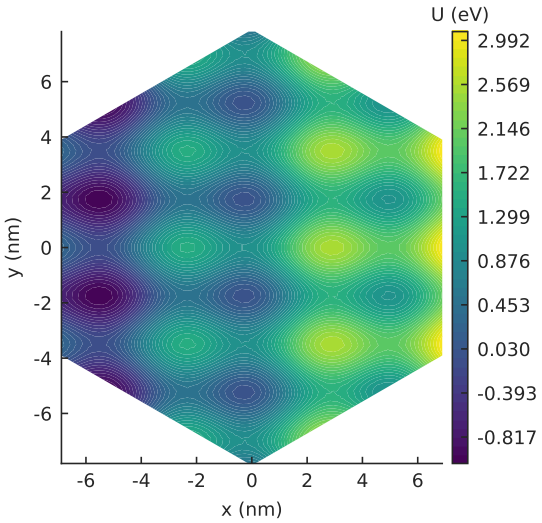
Note that we are using a system with hexagonal shape this time (via *regular_polygon()*). The potential is only plotted inside the area of the actual system.

We can make one more improvement to our `wavy` function. We'll add an `energy` argument:

```
def wavy2(a, b):
    @pb.onsite_energy_modifier
    def potential(energy, x, y):
        v = np.sin(a * x)**2 + np.cos(b * y)**2
        return energy + v
    return potential
```

The `energy` argument contains the existing onsite energy in the system before the new potential function is applied. By adding to the existing energy, instead of just setting it, we can compose multiple functions. For example, let's combine the improved `wavy2` with a linear potential.

```
def linear(k):
    @pb.onsite_energy_modifier
    def potential(energy, x):
        return energy + k*x
    return potential

model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=6, radius=8),
    wavy2(a=0.6, b=0.9),
    linear(k=0.2)
)
model.onsite_map.plot_contourf()
pb.pltutils.colorbar(label="U (eV)")
```



We see a similar wavy pattern as before, but the magnitude increases linearly along the x-axis because of the contribution of the `linear` potential.

About the decorator

Now that you have a general idea of how to add and compose electric potentials in a model, we should talk about the role of the *`@onsite_energy_modifier`*. The full signature of a potential function looks like this:

```
@pb.onsite_energy_modifier
def potential(energy, x, y, z, sub_id):
    return ... # some function of the arguments
```

This function uses all of the possible arguments of an onsite energy modifier: `energy`, `x`, `y`, `z` and `sub_id`. We have already explained the first three. The `z` argument is, obviously, the z-axis coordinate of the lattice sites. The `sub_id` argument tells us which sublattice a site belongs to. Its usage will be explained below.

As we have seen before, we don't actually need to define a function to take all the arguments. They are optional. The `@` decorator will recognize a function which takes any of these arguments and it will adapt it for use in a pybinding model. Previously, the `linear` function accepted only the `energy` and `x` arguments, but `wavy` also included the `y` argument. The order of arguments is not important, only their names are. Therefore, this is also a valid modifier:

```
@pb.onsite_energy_modifier
def potential(x, y, energy, sub_id):
    return ... # some function
```

But the argument names must be exact: a typo or an extra unknown argument will result in an error. The decorator checks this at definition time and decides if the given function is a valid modifier or not, so any errors will be caught early.

Opening a band gap

The last thing to explain about *`@onsite_energy_modifier`* is the use of the `sub_id` argument. It tells us which sublattice a site belongs to. If you remember from early on in the tutorial, *in the process of specifying a lattice*, we gave each sublattice a unique name. This name can be used to filter out sites of a specific sublattice. For example, let's add mass to electrons in graphene:

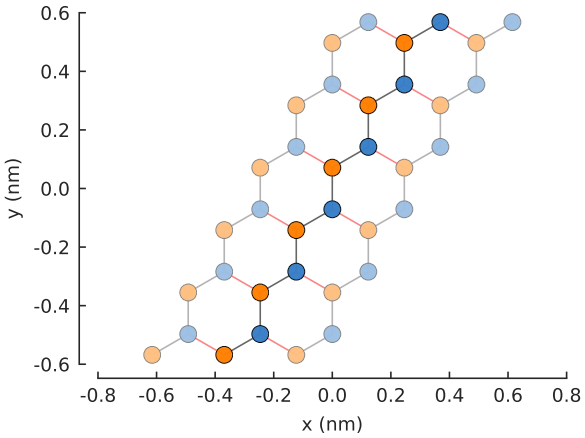
```
def mass_term(delta):
    """Break sublattice symmetry with opposite A and B onsite energy"""
    @pb.onsite_energy_modifier
    def potential(energy, sub_id):
```

```
        energy[sub_id == 'A'] += delta
        energy[sub_id == 'B'] -= delta
        return energy
    return potential
```

Note that we don't need `x`, `y` or `z` arguments because this will be applied everywhere evenly. The `mass_term` function will add an energy `delta` to all sites on sublattice `A` and subtract `delta` from all `B` sites. Note that we are indexing the `energy` array with a condition on the `sub_id` array of the same length. This is a standard `numpy` indexing technique which you should be familiar with.

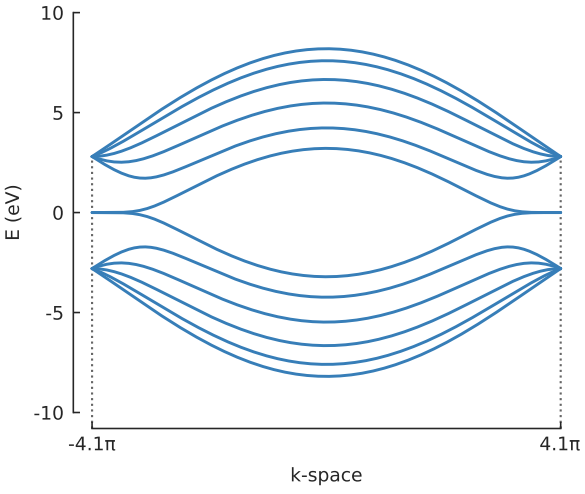
The simplest way to demonstrate our new `mass_term` is with a graphene nanoribbon. First, let's just remind ourselves what a pristine zigzag nanoribbon looks like:

```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),
    pb.translational_symmetry(a1=True, a2=False)
)
model.plot()
```



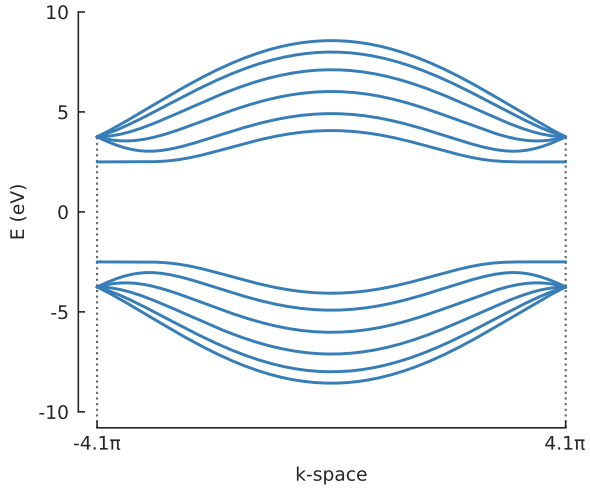
And let's see its band structure:

```
from math import pi, sqrt

solver = pb.solver.lapack(model)
a = graphene.a_cc * sqrt(3)
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
```



Note that the bands touch at zero energy: there is not band gap. Now, let's include the mass term and compute the band structure again.

```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),
    pb.translational_symmetry(a1=True, a2=False),
    mass_term(delta=2.5)  # eV
)
solver = pb.solver.lapack(model)
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
```



We set a very high `delta` value of 2.5 eV for illustration purposes. Indeed, a band gap of 5 eV (`delta * 2`) is quite clearly visible in the band structure.

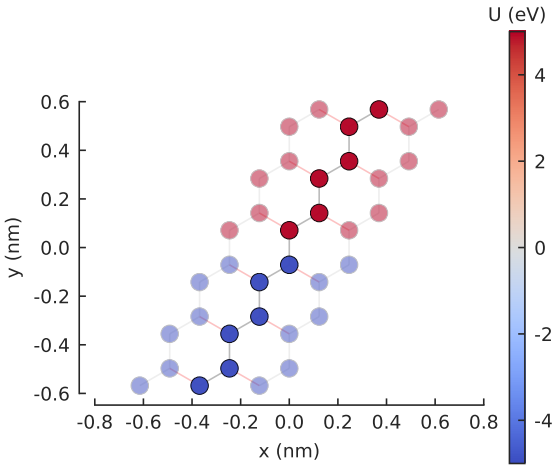
PN junction

While we're working with a nanoribbon, let's add a PN junction along its main axis.

```
def pn_junction(y0, v1, v2):
    @pb.onsite_energy_modifier
    def potential(energy, y):
        energy[y < y0] += v1
        energy[y >= y0] += v2
        return energy
    return potential
```

The `y0` argument is the position of the junction, while `v1` and `v2` are the values of the potential (in eV) before and after the junction. Let's add it to the nanoribbon:

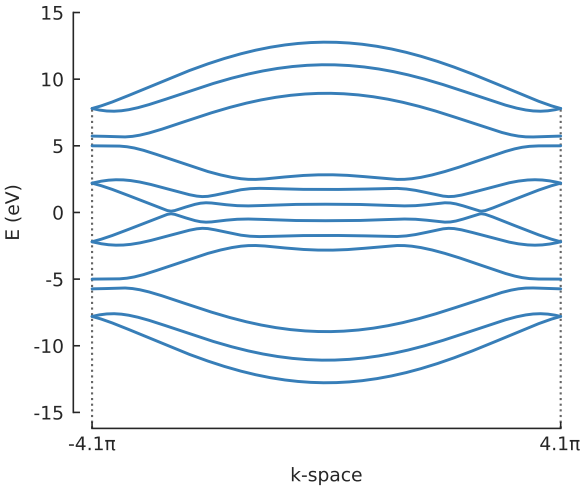
```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),
    pb.translational_symmetry(a1=True, a2=False),
    pn_junction(y0=0, v1=-5, v2=5)
)
model.onsite_map.plot(cmap="coolwarm", site_radius=0.04)
pb.pltutils.colorbar(label="U (eV)")
```



Remember that the *Model.onsite_map* property is a *StructureMap*, which has several plotting methods. A contour plot would not look at all good for such a small nanoribbon, but the method *StructureMap.plot()* is perfect. As before, the ribbon has infinite length along the x-axis and the transparent sites represent the periodic boundaries. The PN junction splits the ribbon in half along its main axis.

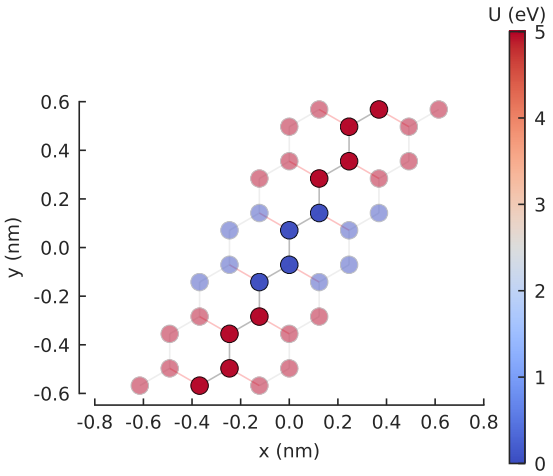
We can compute and plot the band structure:

```
solver = pb.solver.lapack(model)
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
```



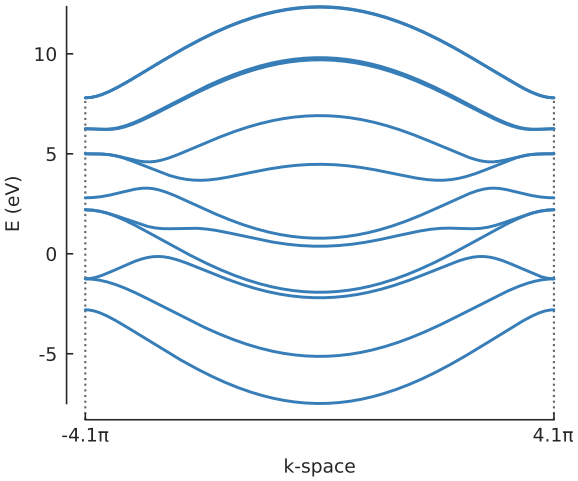
Next, let's create a square potential well. We could define a new modifier function, as before. But lets take a different approach and create the well by composing two PN junctions.

```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),
    pb.translational_symmetry(a1=True, a2=False),
    pn_junction(y0=-0.2, v1=5, v2=0),
    pn_junction(y0=0.2, v1=0, v2=5)
)
model.onsite_map.plot(cmap="coolwarm", site_radius=0.04)
pb.pltutils.colorbar(label="U (eV)")
```



It works as expected. This can sometimes be a nice and quick way to extend a model. The square well affects the band structure by breaking electron-hole symmetry:

```
solver = pb.solver.lapack(model)
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
```



Magnetic field

To model a magnetic field, we need to apply the Peierls substitution:

$$t_{nm} \to t_{nm} \mathrm{e}^{i \frac{2\pi}{\Phi_0} \int_n^m \vec{A}_{nm} \cdot d\vec{l}}$$

Here t_{nm} is the hopping energy between two sites, $\Phi_0 = h/e$ is the magnetic quantum, h is the Planck constant and $\vec{A}_{nm}$ is the magnetic vector potential along the path between sites n and m. We want the magnetic field to be perpendicular to the graphene plane, so we can take the gauge $\vec{A}(x, y, z) = (By, 0, 0)$.

This can all be expressed with a *@hopping_energy_modifier*:

```
from pybinding.constants import phi0

def constant_magnetic_field(B):
    @pb.hopping_energy_modifier
    def function(energy, x1, y1, x2, y2):
        # the midpoint between two sites
        y = 0.5 * (y1 + y2)
```

```
        # scale from nanometers to meters
        y *= 1e-9

        # vector potential along the x-axis
        A_x = B * y

        # integral of (A * dl) from position 1 to position 2
        peierls = A_x * (x1 - x2)
        # scale from nanometers to meters (because of x1 and x2)
        peierls *= 1e-9

        # the Peierls substitution
        return energy * np.exp(1j * 2*pi/phi0 * peierls)
    return function
```

The `energy` argument is the existing hopping energy between two sites at coordinates (`x1`, `y1`) and (`x2`, `y2`). The function computes and returns the Peierls substitution as given by the equation above.

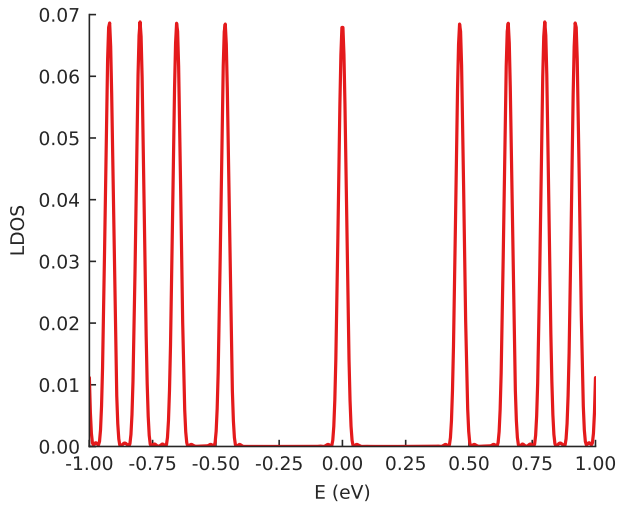
The full signature of a *`@hopping_energy_modifier`* is actually:

```
@pb.hopping_energy_modifier
def function(energy, x1, y1, z1, x2, y2, z2, hop_id):
    return ... # some function of the arguments
```

The `hop_id` argument tells us which type of hopping it is. Hopping types can be specifically named during the creation of a lattice. This can be used to apply functions only to specific hoppings. However, as with all the modifier arguments, it's optional, so we only take what we need.

To test out our `constant_magnetic_field`, we'll calculate the local density of states (LDOS), where we expect to see peaks corresponding to Landau levels. The computation method used here is explained in detail in the *Kernel polynomial method* section of the tutorial.

```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(30),
    constant_magnetic_field(B=200)  # Tesla
)
kpm = pb.kpm(model)

ldos = kpm.calc_ldos(energy=np.linspace(-1, 1, 500), broadening=0.015, position=[0,
↪ 0])
ldos.plot()
plt.show()
```



The values of the magnetic field is exaggerated here (200 Tesla), but that is done to keep the computation time low

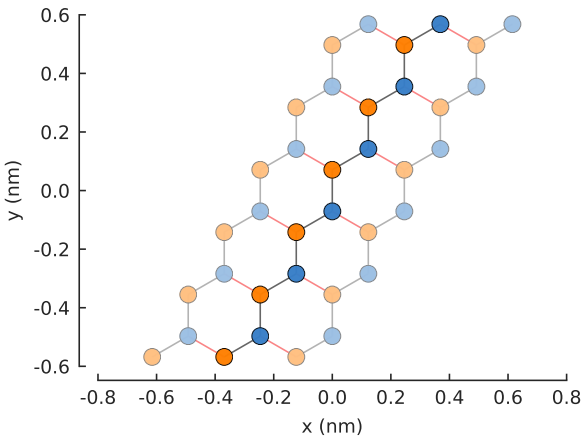
for the tutorial (less than 0.5 seconds for this LDOS calculation).

Further reading

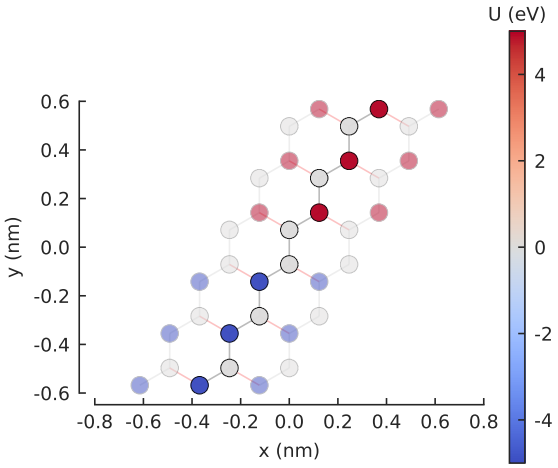
Take a look at the *Modifiers* API reference for more information.

Example

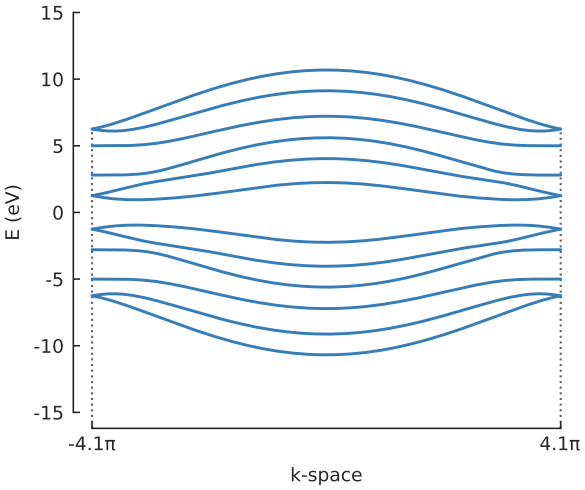
```
"""PN junction and broken sublattice symmetry in a graphene nanoribbon"""
import pybinding as pb
import matplotlib.pyplot as plt
from pybinding.repository import graphene
from math import pi, sqrt

pb.pltutils.use_style()


def mass_term(delta):
    """Break sublattice symmetry with opposite A and B onsite energy"""
    @pb.onsite_energy_modifier
    def potential(energy, sub_id):
        energy[sub_id == 'A'] += delta
        energy[sub_id == 'B'] -= delta
        return energy

    return potential


def pn_junction(y0, v1, v2):
    """PN junction potential

    The `y0` argument is the position of the junction, while `v1` and `v2`
    are the values of the potential (in eV) before and after the junction.
    """
    @pb.onsite_energy_modifier
    def potential(energy, y):
        energy[y < y0] += v1
        energy[y >= y0] += v2
        return energy

    return potential


model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(1.2),  # width in nanometers
    pb.translational_symmetry(a1=True, a2=False),
    mass_term(delta=2.5),  # eV
    pn_junction(y0=0, v1=-2.5, v2=2.5)  # y0 in [nm] and v1, v2 in [eV]
)
model.plot()
plt.show()
```




```
# plot the potential: note that pn_junction cancels out delta on some sites
model.onsite_map.plot(cmap="coolwarm", site_radius=0.04)
pb.pltutils.colorbar(label="U (eV)")
plt.show()
```




```
# compute the bands
solver = pb.solver.lapack(model)
a = graphene.a_cc * sqrt(3)   # nanoribbon unit cell length
bands = solver.calc_bands(-pi/a, pi/a)
bands.plot()
plt.show()
```



Defects and strain

This section will introduce *@site_state_modifier* and *@site_position_modifier* which can be used to add defects and strain to the model. These modifiers are applied to the structure of the system before the Hamiltonian matrix is created.

Vacancies

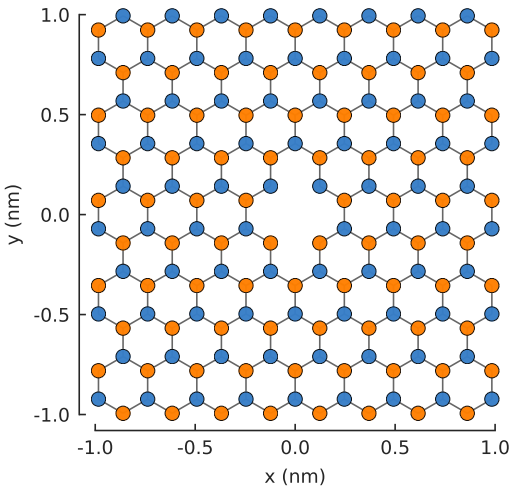
A *@site_state_modifier* can be used to create vacancies in a crystal lattice. The definition is very similar to the onsite and hopping modifiers explained in the previous section.

```
def vacancy(position, radius):
    @pb.site_state_modifier
    def modifier(state, x, y):
        x0, y0 = position
        state[(x-x0)**2 + (y-y0)**2 < radius**2] = False
        return state
    return modifier
```

The `state` argument indicates the current boolean state of a lattice site. Only valid sites (`True` state) will be included in the final Hamiltonian matrix. Therefore, setting the state of sites within a small radius to `False` will exclude them from the final system. The `x` and `y` arguments are lattice site positions. As with the other modifiers, the arguments are optional (`z` is not needed for this example) but the full signature of the site state modifier can be found on its *API reference page*.

This is actually very similar to the way a *FreeformShape* works. In fact, it is possible to create defects by defining them directly in the shape. However, such an approach would not be very flexible since we would need to create an entire new shape in order to change either the vacancy type or the shape itself. By defining the vacancy as a modifier, we can simply compose it with any existing shapes:

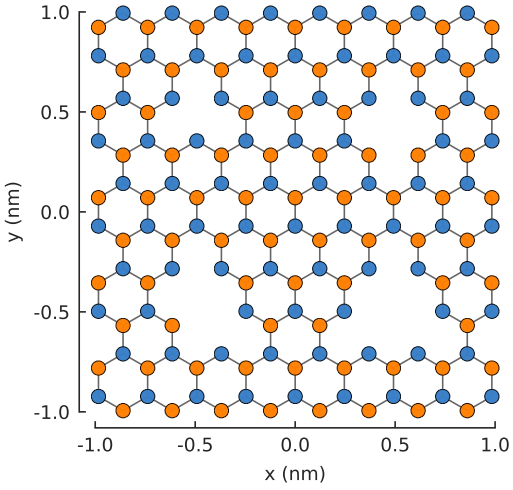
```
from pybinding.repository import graphene

model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(2),
    vacancy(position=[0, 0], radius=0.1)
)
model.plot()
```



The resulting 2-atom vacancy is visible in the center of the system. The two vacant sites are completely removed from the final Hamiltonian matrix. If we were to inspect the number of rows and columns by looking up `model.hamiltonian.shape`, we would see that the size of the matrix is reduced by 2.

Any number of modifiers can be included in the model and they will compose as expected. We can take advantage of this and create four different vacancies, with 1 to 4 missing atoms:

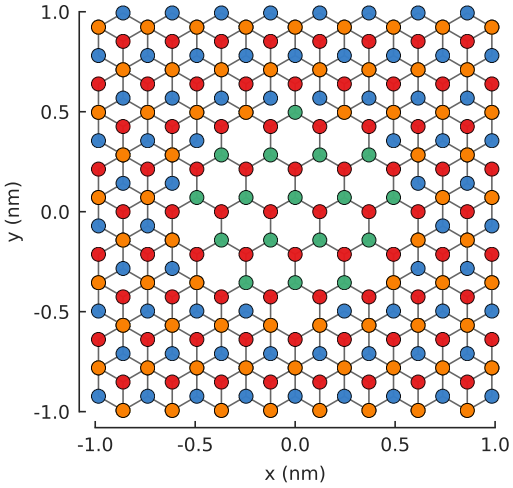
```
model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(2),
    vacancy(position=[-0.50,  0.50], radius=0.1),
    vacancy(position=[ 0.50,  0.45], radius=0.15),
    vacancy(position=[-0.45, -0.45], radius=0.15),
    vacancy(position=[ 0.50, -0.50], radius=0.2),
)
model.plot()
```



Layer defect

The site state modifier also has access to sublattice information. This can be used, for example, with bilayer graphene to remove a single layer in a specific area. We'll use the bilayer lattice that's included in the *Material Repository*. The `graphene.bilayer()` lattice is laid out so that sublattices A1 and B1 belong to the top layer, while A2 and B2 are on the bottom.

```
def scrape_top_layer(position, radius):
    """Remove the top layer of graphene in the area specified by position and␣
↪radius"""
    @pb.site_state_modifier
    def modifier(state, x, y, sub_id):
        x0, y0 = position
        is_within_radius = (x-x0)**2 + (y-y0)**2 < radius**2
        is_top_layer = np.logical_or(sub_id == 'A1', sub_id == 'B1')
        final_condition = np.logical_and(is_within_radius, is_top_layer)
        state[final_condition] = False
        return state
    return modifier

model = pb.Model(
    graphene.bilayer(),
    pb.rectangle(2),
    scrape_top_layer(position=[0, 0], radius=0.5)
)
model.plot()
```



The central monolayer area is nicely visible in the figure. We can actually create the same structure in a different way: by considering the `z` position of the lattice site to distinguish the layers. An alternative modifier definition is given below. It would generate the same figure. Which method is more convenient is up to the user.

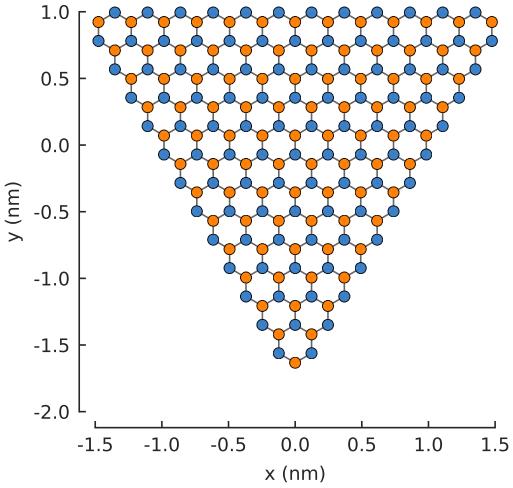
```
def scrape_top_layer_alt(position, radius):
    """Alternative definition of `scrape_top_layer`"""
    @pb.site_state_modifier
    def modifier(state, x, y, z):
        x0, y0 = position
        is_within_radius = (x-x0)**2 + (y-y0)**2 < radius**2
        is_top_layer = (z == 0)
        final_condition = np.logical_and(is_within_radius, is_top_layer)
        state[final_condition] = False
        return state
    return modifier
```

Note: As with the onsite and hopping modifiers, all the arguments are given as numpy arrays. Therefore, we must use the array-specific `np.logical_or()` / `np.logical_and()` functions instead of the plain `or`/`and` keywords.

Strain

A *`@site_position_modifier`* can be used to model the lattice site displacement caused by strain. Let's start with a simple triangular system:

```
from math import pi

model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=3, radius=2, angle=pi),
)
model.plot()
```

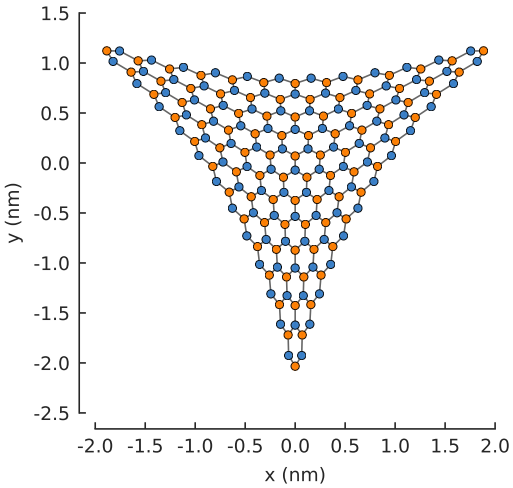


We're going to apply strain in three directions, as if we are pulling outward on the vertices of the triangle. The displacement function for this kind of strain is given below. The `c` parameter lets us control the intensity of the strain.

```
def triaxial_displacement(c):
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = 2*c * x*y
        uy = c * (x**2 - y**2)
        return x + ux, y + uy, z
    return displacement
```

The modifier function takes the `x`, `y`, `z` coordinates as arguments. The displacement `ux`, `uy` is computed and the modified coordinates are returned. The `z` argument is returned unchanged but we still need it here because the modifier is expected to always return all three.

```
model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=3, radius=2, angle=pi),
    triaxial_displacement(c=0.15)
)
model.plot()
```



As seen in the figure, the displacement has been applied to the lattice sites and the new position data is saved in the system. However, the hopping energies have not been modified yet. Every hopping element of the Hamiltonian matrix is equal to the hopping energy of pristine graphene:

```
>>> np.all(model.hamiltonian.data == -2.8)
True
```

We now need to use the new position data to modify the hopping energy according to the relation $t = t_0 e^{-\beta(\frac{d}{a_{cc}}-1)}$, where t_0 is the original unstrained hopping energy, β controls the strength of the strain-induced hopping modulation, d is the strained distance between two atoms and a_{cc} is the unstrained carbon-carbon distance. This can be implemented using a *@hopping_energy_modifier*:

```
@pb.hopping_energy_modifier
def strained_hopping(energy, x1, y1, z1, x2, y2, z2):
    d = np.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)
    beta = 3.37
    w = d / graphene.a_cc - 1
    return energy * np.exp(-beta*w)
```

The structural modifiers (site state and position) are always automatically applied to the model before energy modifiers (onsite and hopping). Thus, our `strain_hopping` modifier will get the new displaced coordinates as its arguments, from which it will calculate the strained hopping energy.

```
model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=3, radius=2, angle=pi),
    triaxial_displacement(c=0.15),
    strained_hopping
)
```

Including the hopping modifier along with the displacement will yield position dependent hopping energy, thus the elements of the Hamiltonian will no longer be all equal:

```
>>> np.all(model.hamiltonian.data == -2.8)
False
```

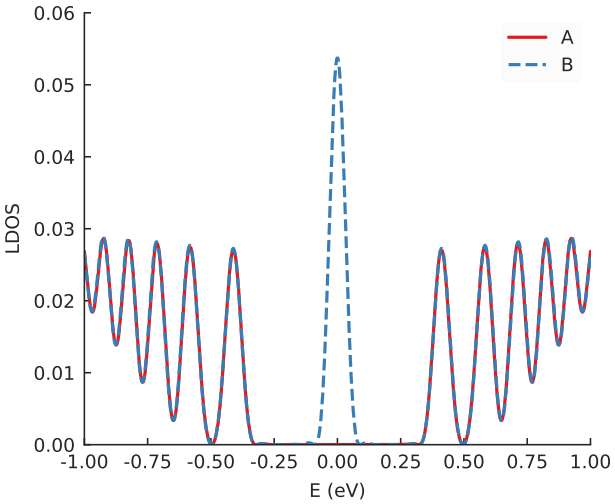
However, it isn't convenient to keep track of the displacement and strained hoppings separately. Instead, we can package them together in one function which is going to return both modifiers:

```
def triaxial_strain(c, beta=3.37):
    """Produce both the displacement and hopping energy modifier"""
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = 2*c * x*y
        uy = c * (x**2 - y**2)
        return x + ux, y + uy, z

    @pb.hopping_energy_modifier
    def strained_hopping(energy, x1, y1, z1, x2, y2, z2):
        l = np.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)
        w = l / graphene.a_cc - 1
        return energy * np.exp(-beta*w)

    return displacement, strained_hopping
```

The `triaxial_strain` function now has everything we need. We'll apply it to a slightly larger system so that we can clearly calculate the local density of states (LDOS). For more information about this computation method see the *Kernel polynomial method* section. Right now, it's enough to know that we will calculate the LDOS at the center of the strained system, separately for sublattices A and B.

```
model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=3, radius=40, angle=pi),
    triaxial_strain(c=0.0025)
)
```

```
kpm = pb.kpm(model)

for sub_name in ['A', 'B']:
    ldos = kpm.calc_ldos(energy=np.linspace(-1, 1, 500), broadening=0.03,
                         position=[0, 0], sublattice=sub_name)
    ldos.plot(label=sub_name, ls="--" if sub_name == "B" else "-")
pb.pltutils.legend()
```



Strain in graphene has an effect similar to a magnetic field. That's why we see Landau-level-like features in the LDOS. Note that the zero-energy peak has double intensity on one sublattice but zero on the other: this is a unique feature of the strain-induced pseudo-magnetic field.

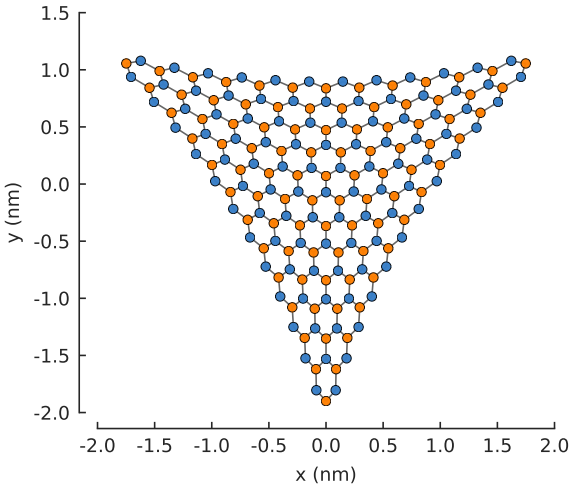
Further reading

Take a look at the *Modifiers* API reference for more information.

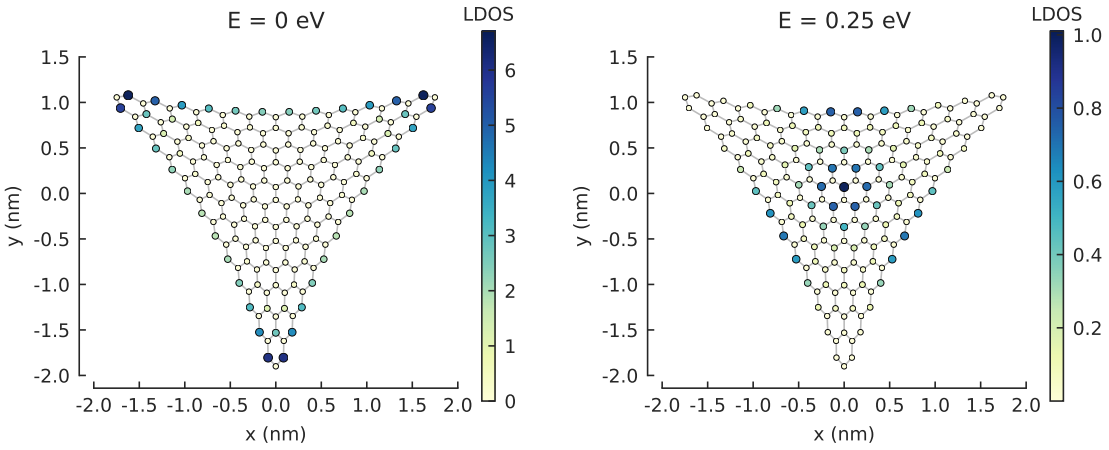
Example

```
"""Strain a triangular system by pulling on its vertices"""
import pybinding as pb
import numpy as np
import matplotlib.pyplot as plt
from pybinding.repository import graphene
from math import pi

pb.pltutils.use_style()


def triaxial_strain(c):
    """Strain-induced displacement and hopping energy modification"""
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = 2*c * x*y
        uy = c * (x**2 - y**2)
        return x + ux, y + uy, z

    @pb.hopping_energy_modifier
    def strained_hopping(energy, x1, y1, z1, x2, y2, z2):
        l = np.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)
        w = l / graphene.a_cc - 1
        return energy * np.exp(-3.37 * w)
```

```
    return displacement, strained_hopping


model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=3, radius=2, angle=pi),
    triaxial_strain(c=0.1)
)
model.plot()
plt.show()
```




```
plt.figure(figsize=(7, 2.5))
grid = plt.GridSpec(nrows=1, ncols=2)
for block, energy in zip(grid, [0, 0.25]):
    plt.subplot(block)
    plt.title("E = {} eV".format(energy))

    solver = pb.solver.arpack(model, k=30, sigma=energy)
    ldos_map = solver.calc_spatial_ldos(energy=energy, broadening=0.03)
    ldos_map.plot()
    pb.pltutils.colorbar(label="LDOS")

plt.show()
```



Eigenvalue solvers

Solvers were first introduced in the *Band structure* section and then used throughout the tutorial to present the results of the various models we constructed. This section will take a more detailed look at the concrete *lapack()* and *arpack()* eigenvalue solvers and their common *Solver* interface.

LAPACK

The *Solver* class establishes the interface of a solver within pybinding, but it does not contain a concrete diagonalization routine. For this reason we never instantiate the plain solver, only its implementations such as *solver.lapack()*.

The LAPACK implementation works on dense matrices which makes it well suited only for small systems. However, a great advantage of this solver is that it always solves for all eigenvalues and eigenvectors of a Hamiltonian matrix. This makes it perfect for calculating the entire band structure of the bulk or nanoribbons, as has been shown several times in this tutorial.

Internally, this solver uses the `scipy.linalg.eigh()` function for dense Hermitian matrices. See the *solver.lapack()* API reference for more details.

ARPACK

The *solver.arpack()* implementation works on sparse matrices which makes it suitable for large systems. However, only a small subset of the total eigenvalues and eigenvectors can be calculated. This tutorial already contains a few examples where the ARPACK solver was used, and one more is presented below.

Internally, the `scipy.sparse.linalg.eigsh()` function is used to solve large sparse Hermitian matrices. The first argument to *solver.arpack()* must be the pybinding *Model*, but the following arguments are the same as `eigsh()`, so the solver routine can be tweaked as desired. Rather than reproduce the full list of options here, we refer you to the scipy `eigsh()` reference documentation. Here, we will focus on the specific features of solvers within pybinding.

Solver interface

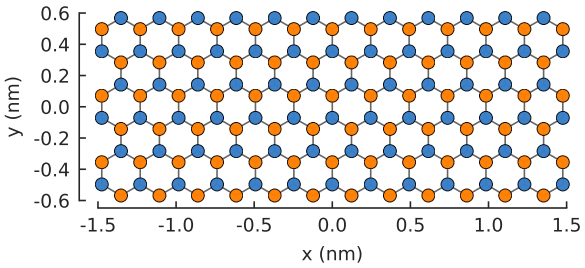
No matter which concrete solver is used, they all share a common *Solver* interface. The two primary properties are *eigenvalues* and *eigenvectors*. These are the raw results of the exact diagonalization of the Hamiltonian matrix.

```
>>> from pybinding.repository import graphene
>>> model = pb.Model(graphene.monolayer())
>>> model.hamiltonian.todense()
[[ 0.0 -2.8]
 [-2.8  0.0]]
>>> solver = pb.solver.lapack(model)
>>> solver.eigenvalues
[-2.8 2.8]
>>> solver.eigenvectors
[[-0.707 -0.707]
 [-0.707  0.707]]
```

The properties contain just the raw data. However, *Solver* also offers a few convenient calculation methods. We'll demonstrate these on a simple rectangular graphene system.

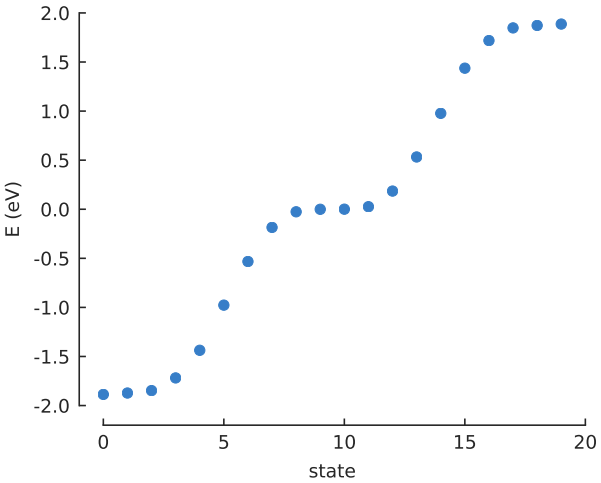
```
from pybinding.repository import graphene

model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(x=3, y=1.2)
```

```
)
model.plot()
```



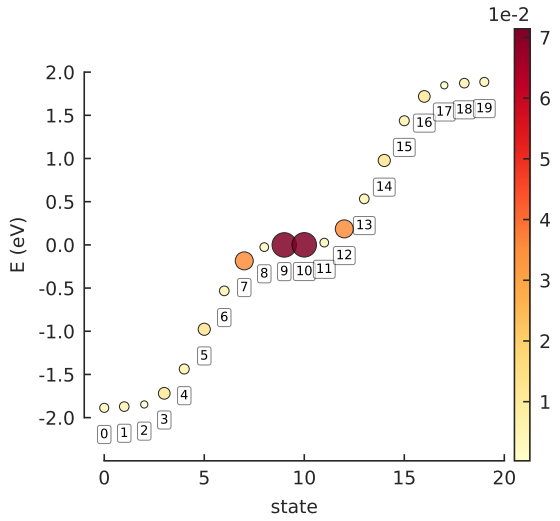
First, we'll take a look at the *calc_eigenvalues()* method. While its job is essentially the same as the *eigenvalues* property, there is one key difference: the property returns a raw array, while the method returns an *Eigenvalues* result object. These objects have convenient functions built in and they know how to plot their data:

```
solver = pb.solver.arpack(model, k=20)  # for the 20 lowest energy eigenvalues
eigenvalues = solver.calc_eigenvalues()
eigenvalues.plot()
```



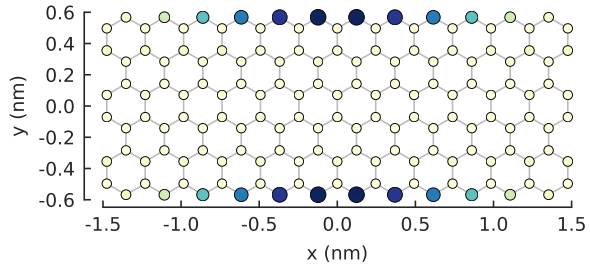
The basic plot just shows the state number and energy of each eigenstate, but we can also do something more interesting. If we pass a position argument to *calc_eigenvalues()* it will calculate the probability density $|\Psi(\vec{r})|^2$ at that position for each eigenstate and we can view the result using *Eigenvalues.plot_heatmap()*:

```
eigenvalues = solver.calc_eigenvalues(map_probability_at=[0.1, 0.6])  # position␣
↪in [nm]
eigenvalues.plot_heatmap(show_indices=True)
pb.pltutils.colorbar()
```



In this case we are interested in the probability density at `[x, y] = [0.1, 0.6]`, i.e. a lattice site at the top zigzag edge of our system. Note that the given position does not need to be precise: the probability will be computed for the site closest to the given coordinates. From the figure we can see that the probability at the edge is highest for the two zero-energy states: numbers 9 and 10. We can take a look at the spatial map of state 9 using the *`calc_probability()`* method:

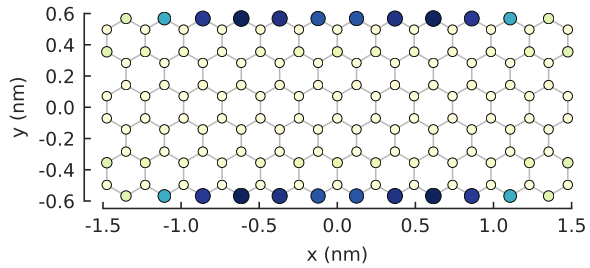
```
probability_map = solver.calc_probability(9)
probability_map.plot()
```



The result object in this case is a *`StructureMap`* with the probability density $|\Psi(\vec{r})|^2$ as its data attribute. As expected, the most prominent states are at the zigzag edges of the system.

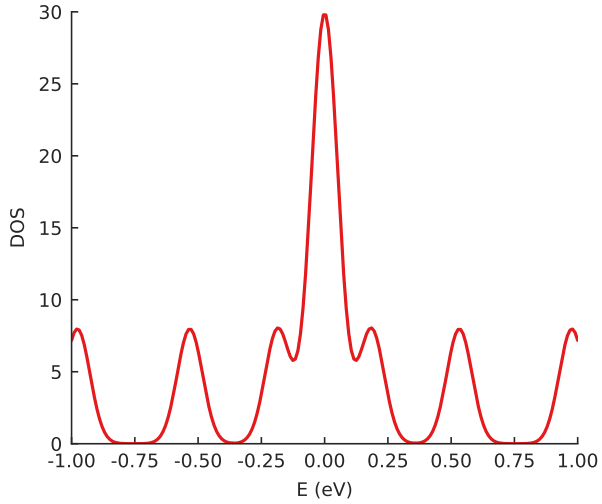
An alternative way to get a spatial map of the system is via the local density of states (LDOS). The *`calc_spatial_ldos()`* method makes this easy. The LDOS map is requested for a specific energy value instead of a state number and it considers multiple states within a Gaussian function with the specified broadening:

```
ldos_map = solver.calc_spatial_ldos(energy=0, broadening=0.05)  # [eV]
ldos_map.plot()
```



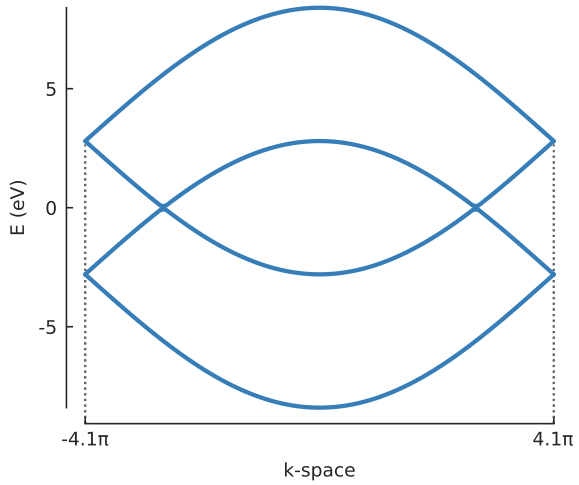
The total density of states can be calculated with *`calc_dos()`*:

```
dos = solver.calc_dos(energies=np.linspace(-1, 1, 200), broadening=0.05)  # [eV]
dos.plot()
```



Our example system is quite small so the DOS does not resemble bulk graphene. The zero-energy peak stands out as the signature of the zigzag edge states. For periodic systems, the wave vector can be controlled using *Solver.set_wave_vector()*. This allows us to compute the eigenvalues at various points in k-space. For example:

```
from math import pi

model = pb.Model(
    graphene.monolayer(),
    pb.translational_symmetry()
)
solver = pb.solver.lapack(model)

kx_lim = pi / graphene.a
kx_path = np.linspace(-kx_lim, kx_lim, 100)
ky_outer = 0
ky_inner = 2*pi / (3*graphene.a_cc)

outer_bands = []
for kx in kx_path:
    solver.set_wave_vector([kx, ky_outer])
    outer_bands.append(solver.eigenvalues)

inner_bands = []
for kx in kx_path:
    solver.set_wave_vector([kx, ky_inner])
    inner_bands.append(solver.eigenvalues)

for bands in [outer_bands, inner_bands]:
    result = pb.results.Bands(kx_path, bands)
    result.plot()
```



This example shows the basic principle of iterating over a path in k-space in order to calculate the band structure. However, this is made much easier with the *`Solver.calc_bands()`* method. This was already covered in the *Band structure* section and will not be repeated here. But keep in mind that this calculation does not need to be done manually, *`Solver.calc_bands()`* is the preferred way.

Further reading

Take a look at the *`solver`* and `results` reference pages for more detailed information. More solver examples are available throughout this tutorial.

Kernel polynomial method

The kernel polynomial method (KPM) can be used to quickly compute various physical properties of very large tight-binding systems. It makes use of Chebyshev polynomial expansion together with damping kernels. Pybinding includes a fast *`kpm()`* implementation with several easy-to-use computation methods as well as a low-level interface for computing KPM expansion moments.

About KPM

For a full review of the kernel polynomial method, see the reference paper Rev. Mod. Phys. 78, 275 (2006). Here, we shall only briefly describe the main characteristics of KPM and some specifics of its implementation in pybinding.

As we saw on the previous page, exactly solving a tight-binding problem implies the diagonalization of the Hamiltonian matrix. However, the computational resources required by eigenvalue solvers scale up rapidly with system size which makes it challenging to solve realistically large systems. A fundamentally different approach is to set aside the requirement for exact solutions (avoid diagonalization altogether) and instead use approximative methods to calculate the properties of interest. This is the main idea behind KPM which approximates functions as a series of Chebyshev polynomials.

The approximative nature of the method presents an opportunity for additional performance tuning. Results may be computed very quickly with low accuracy to get an initial estimate for the problem at hand. Once final results are required, the accuracy can be increased at the cost of longer computation time. Within pybinding, this KPM calculation quality is frequently expressed as an energy broadening parameter.

One of the great benefits of this method is that spatially dependent properties such as the local density of states (LDOS) or Green's function are calculated separately for each spatial position. This means that localized properties can be computed extremely quickly. For this application, KPM can be seen as orthogonal to traditional eigenvalue solvers. Sparse diagonalization produces results for a very small energy range (eigenvalues) but does so for all positions simultaneously (eigenvectors). With KPM, it's possible to separate and compute individual

positions but for the entire energy spectrum at once. In this way, the two approaches complement each other nicely.

Builtin methods

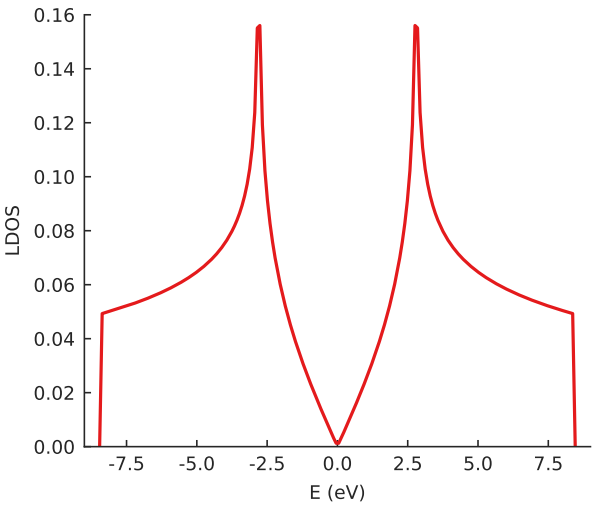
Before using any of the computation methods, the main *KPM* object needs to be created for a specific model:

```
model = pb.Model(...)
kpm = pb.kpm(model)
# ... use kpm
```

LDOS

The *KPM.calc_ldos()* method makes it very easy to calculate the local density of states (LDOS). In the next example we'll use a large square sheet of pristine graphene:

```
from pybinding.repository import graphene

model = pb.Model(graphene.monolayer(), pb.rectangle(60, 60))
kpm = pb.kpm(model)

ldos = kpm.calc_ldos(energy=np.linspace(-9, 9, 200), broadening=0.05, position=[0,␣
↪0])
ldos.plot()
```



The LDOS is calculated for energies between -9 and 9 eV with a Gaussian broadening of 50 meV. Since this is the *local* density of states, position is also a required argument. We target the center of our square system where we expect to see the well-known LDOS shape of pristine graphene.

Thanks to KPM, the calculation of this local property is very fast: about 0.1 seconds for the example above with a 60 x 60 nm sheet of graphene. The broadening parameter offers the possibility for performance tuning – calculation time is inversely proportional to broadening width. KPM performs the computation for the entire spectrum simultaneously, so the selected energy range and the number of sample points have almost no effect on performance. The broadening width (i.e. the precision of the results) is the main factor which determines the duration of the calculation.

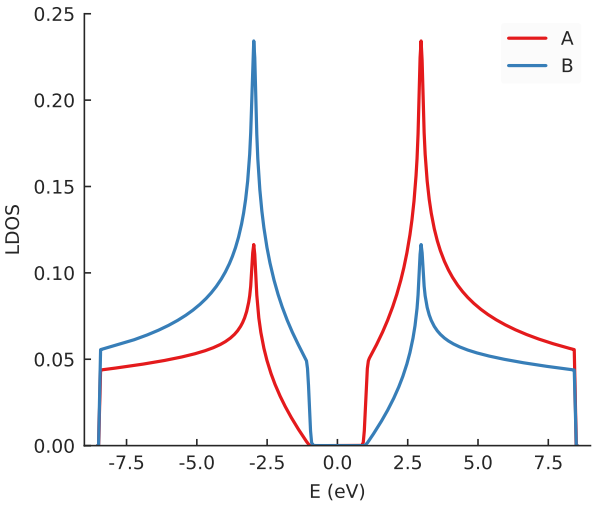
The result of the calculation is a *Series* object which contains the LDOS data, the energy array for which it was calculated, and the associated data labels. This allows the *Series.plot()* method to automatically plot a nicely labeled line plot, as seen above. Accessing the raw data represented on the y-axis is possible via the `Series.data` attribute, i.e. `ldos.data` in this specific case.

Tight-binding systems have lattice sites at discrete positions, which in principle means that we cannot freely choose just any position for LDOS calculations. However, as a convenience the *KPM.calc_ldos()* method will automatically find a valid site closest to the given target position. We can optionally also choose a specific sublattice:

```
ldos = kpm.calc_ldos(energy=np.linspace(-9, 9, 200), broadening=0.05,
                     position=[0, 0], sublattice="B")
```

In this case we would calculate the LDOS at a site of sublattice B closest to the center of the system. We can try that on a graphene system with a mass term:

```
model = pb.Model(
    graphene.monolayer(),
    graphene.mass_term(1),
    pb.rectangle(60)
)
kpm = pb.kpm(model)

for sub_name in ["A", "B"]:
    ldos = kpm.calc_ldos(energy=np.linspace(-9, 9, 500), broadening=0.05,
                         position=[0, 0], sublattice=sub_name)
    ldos.plot(label=sub_name)
pb.pltutils.legend()
```

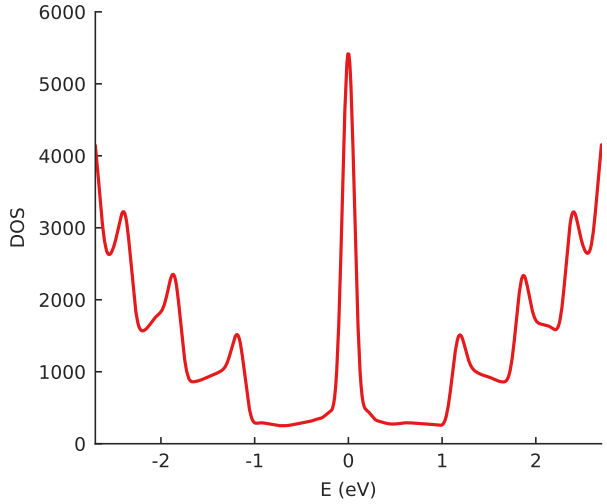


Multiple plots compose nicely here. A large band gap is visible at zero energy due to the inclusion of *graphene.mass_term()*. It places an onsite potential with the opposite sign in each sublattice. This is also why the LDOS lines for A and B sublattices are antisymmetric around zero energy with respect to one another.

DOS

The following example demonstates the usage of the *KPM.calc_dos()* method which computes the total density of states (DOS) in a system:

```
model = pb.Model(graphene.monolayer(), pb.rectangle(400, 2))
kpm = pb.kpm(model)

dos = kpm.calc_dos(energy=np.linspace(-2.7, 2.7, 500), broadening=0.06, num_
↪random=16)
dos.plot()
```



The example system here is a very long but narrow (400 x 2 nm) rectangle of graphene, i.e. a zigzag nanoribbon of finite length. The pronounced zero-energy peak is due to zigzag edge states and the additional higher-energy DOS peaks reflect the quantized band structure of the narrow nanoribbon.

A specific feature of the KPM-based DOS calculation is that it can be approximated very quickly using stochastic methods. Instead of computing the density of states at each sites individually and summing up the results, the DOS is calculated for all sites at the same time, but with a random contribution of each site. By repeating this procedure multiple times with different random staring states, the full DOS is recovered. This presents an additional knob for performance/quality tuning via the `num_random` parameter.

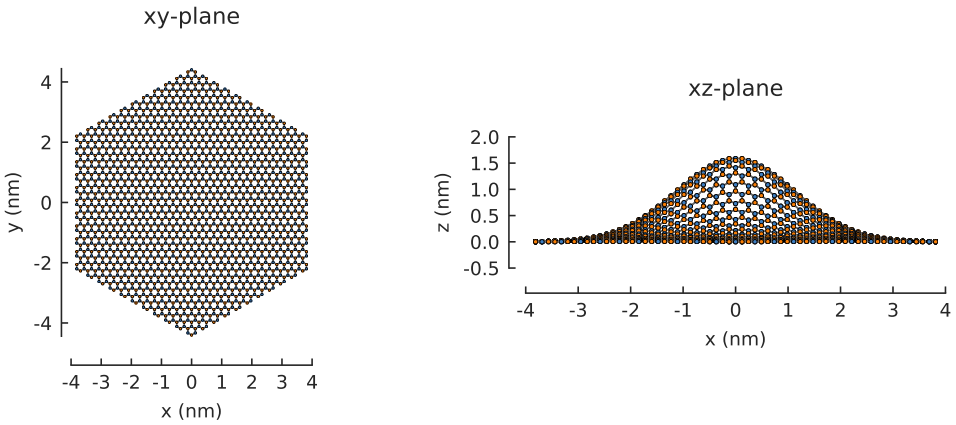
For this example, we keep `num_random` low to keep the calculation time under 1 second. Increasing this number would smooth out the DOS further. Luckily, the stochastic evaluation converges as a function of both the system size and number of random samples. Thus, the larger the model system, the smaller `num_random` needs to be for the same result quality.

Spatial LDOS

To see the spatial distribution of the density of states, we could call the *KPM.calc_ldos()* method for several positions and populate a `SpatialMap`. However, this would be tedious and slow, so instead we have *KPM.calc_spatial_ldos()* which makes this much simpler. Let's use a strained bit of graphene as an example:

```
def gaussian_bump_strain(height, sigma):
    """Out-of-plane deformation (bump)"""
    @pb.site_position_modifier
    def displacement(x, y, z):
        dz = height * np.exp(-(x**2 + y**2) / sigma**2)  # gaussian
        return x, y, z + dz  # only the height changes

    @pb.hopping_energy_modifier
    def strained_hoppings(energy, x1, y1, z1, x2, y2, z2):
        d = np.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)  # strained neighbor␣
↪distance
        return energy * np.exp(-3.37 * (d / graphene.a_cc - 1))  # see strain␣
↪section

    return displacement, strained_hoppings

model = pb.Model(graphene.monolayer().with_offset([-graphene.a / 2, 0]),
                 pb.regular_polygon(num_sides=6, radius=4.5),
                 gaussian_bump_strain(height=1.6, sigma=1.6))

plt.figure(figsize=(6.7, 2.2))
plt.subplot(121, title="xy-plane", ylim=[-5, 5])
model.plot()
```

```
plt.subplot(122, title="xz-plane")
model.plot(axes="xz")
```

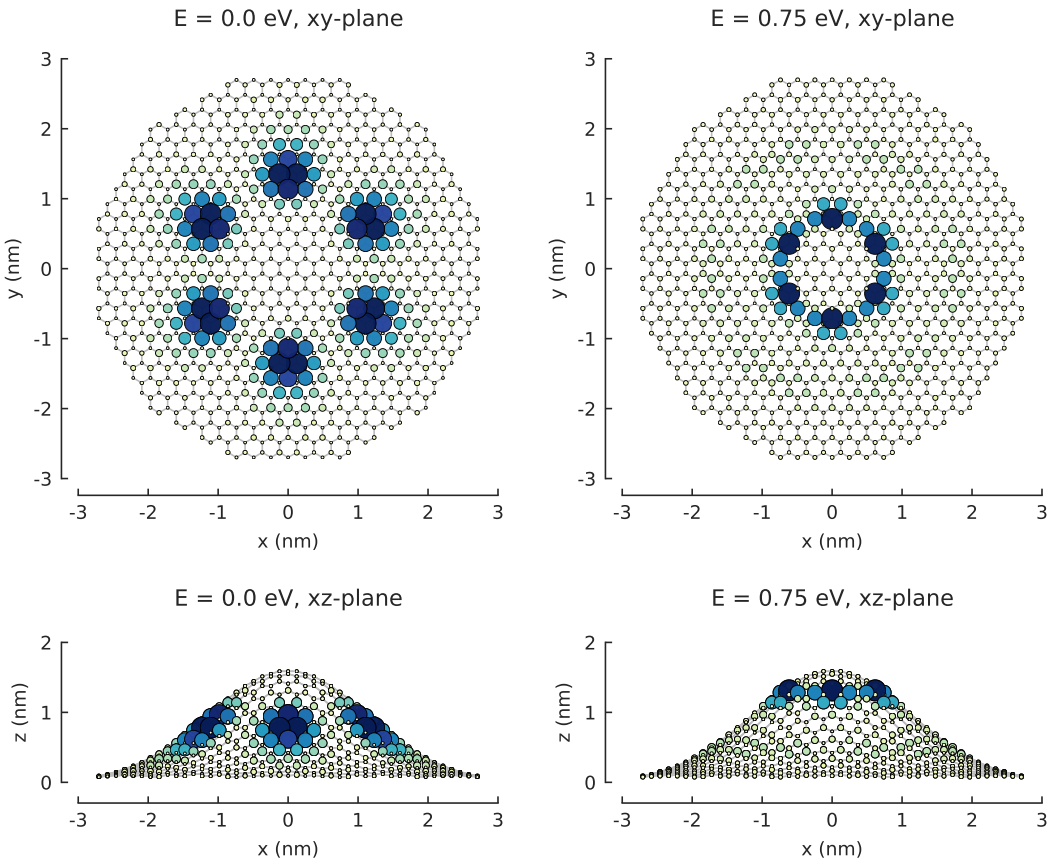


The bump produces purely out-of-plane strain so the xy-plane does not show any signs of the deformation. Switching to the xz-plane reveals the bump.

The *KPM.calc_spatial_ldos()* method takes the same `energy` and `broadening` arguments as we've seen before. KPM computes the entire spectrum simultaneously, so it's practically "free" to compute the spatial LDOS at multiple energy values in one calculation (this is in contrast to *Solver.calc_spatial_ldos()* which only targets a single energy).

The `shape` argument specifies the area where the LDOS is to be calculated, i.e. the sites which are contained within the given shape. We could just specify the same shape as the model, thus taking all sites into consideration, but the calculation is faster for smaller areas so we'll narrow our focus. Our model shape is hexagonal, but we're only interested in the LDOS at the bump so we can look at a smaller circular area:

```
kpm = pb.kpm(model)
spatial_ldos = kpm.calc_spatial_ldos(energy=np.linspace(-3, 3, 100), broadening=0.
↪2,  # eV
                                     shape=pb.circle(radius=2.8))  # only within␣
↪the shape
plt.figure(figsize=(6.7, 6))
gridspec = plt.GridSpec(2, 2, height_ratios=[1, 0.3], hspace=0)

energies = [0.0, 0.75, 0.0, 0.75]  # eV
planes = ["xy", "xy", "xz", "xz"]

for g, energy, axes in zip(gridspec, energies, planes):
    plt.subplot(g, title="E = {} eV, {}-plane".format(energy, axes))
    smap = spatial_ldos.structure_map(energy)
    smap.plot(site_radius=(0.02, 0.15), axes=axes)
```



The result of the calculation is a *SpatialLDOS* object which stores the spatial LDOS for several energy values. Calling *SpatialLDOS.structure_map()* selects a specific energy.

Green's function

The *KPM.calc_greens()* can then be used to calculate Green's function corresponding to Hamiltonian matrix element `i, j` for the desired energy range and broadening:

```
g_ij = kpm.calc_greens(i, j, energy=np.linspace(-9, 9, 100), broadening=0.1)
```

The result is raw Green's function data for the given matrix element.

Conductivity

The *KPM.calc_conductivity()* method computes the conductivity as a function of chemical potential. The implementation uses the Kubo-Bastin formula expanded in terms of Chebyshev polynomials, as described in https://doi.org/10.1103/PhysRevLett.114.116602. The following example calculates the conductivity tensor for the quantum Hall effect in graphene with a magnetic field:

```
width = 40  # nanometers
model = pb.Model(
    graphene.monolayer(), pb.rectangle(width, width),
    graphene.constant_magnetic_field(magnitude=1500)  # exaggerated field strength
)

# The conductivity calculation is based on Green's function
# for which the Lorentz kernel produces better results.
kpm = pb.chebyshev.kpm(model, kernel=pb.lorentz_kernel())

directions = {
```

```
    r"$\sigma_{xx}$": "xx",  # longitudinal conductivity
    r"$\sigma_{xy}$": "xy",  # off-diagonal (Hall) conductivity
}
for name, direction in directions.items():
    sigma = kpm.calc_conductivity(chemical_potential=np.linspace(-1.5, 1.5, 300),
                                  broadening=0.1, direction=direction,␣
↪temperature=0,
                                  volume=width**2, num_random=10)
    sigma.data *= 4  # to account for spin and valley degeneracy
    sigma.plot(label=name)
pb.pltutils.legend()
```

Note: The calculation above takes about a minute to complete. Please take note of that if you've downloaded this page as a Jupyter notebook and are executing the code on your own computer. If you're viewing this online, you'll notice that the result figure is not shown. This is because all of the figures in pybinding's documentation are generated automatically by readthedocs.org (RTD) from the example code (not when you load the webpage, but when a new documentation revision is uploaded). RTD has a documentation build limit of 15 minutes so all of the example code presented on these pages is kept short and fast, preferably under 1 second for each snippet. The long runtime of this conductivity calculation forces us to skip it in order to conserve documentation build time.

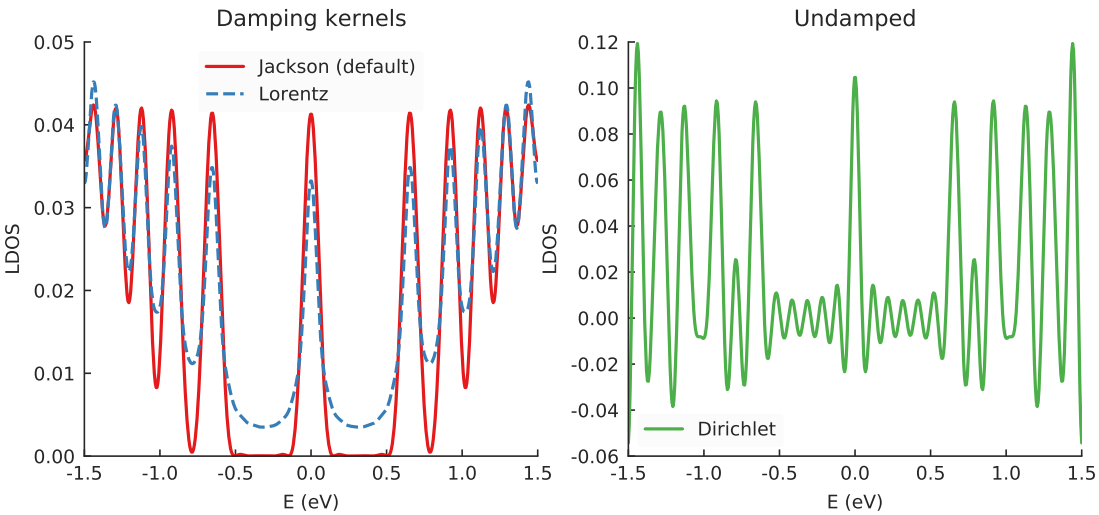
You can execute this code on your own computer to see the results. The parameters here have been tuned in order to take the minimal amount of time while still showing the desired effect. However, that is not the most aesthetically pleasing result. To improve the quality of the resulting figure, you can increase the size of the system, reduce the magnetic field strength, reduce the broadening and increase the number of random vectors. That could extend the computation time from a few minutes to several hours.

Damping kernels

KPM approximates a function as a series of Chebyshev polynomials. This series is infinite, but numerical calculations must end at some point, thus taking into account only a finite number of terms. This truncation results in a loss of precision and high frequency oscillations in the computed function. In order to damp these fluctuations, the function can be convolved with various damping kernels (the K in KPM).

Pybinding offers three option: *jackson_kernel()*, *lorentz_kernel()* and *dirichlet_kernel()*. The Jackson kernel is enabled by default and it is the best choice for most applications. The following example compares the three kernels:

```
plt.figure(figsize=(6.7, 2.8))
model = pb.Model(graphene.monolayer(), pb.circle(30),
                 graphene.constant_magnetic_field(400))

plt.subplot(121, title="Damping kernels")
kernels = {"Jackson (default)": pb.jackson_kernel(),
           "Lorentz": pb.lorentz_kernel()}
for name, kernel in kernels.items():
    kpm = pb.kpm(model, kernel=kernel)
    ldos = kpm.calc_ldos(np.linspace(-1.5, 1.5, 500), broadening=0.05, position=[0,
↪ 0])
    ldos.plot(label=name, ls="--" if name == "Lorentz" else "-")
pb.pltutils.legend()

plt.subplot(122, title="Undamped")
kpm = pb.kpm(model, kernel=pb.dirichlet_kernel())
ldos = kpm.calc_ldos(np.linspace(-1.5, 1.5, 500), broadening=0.05, position=[0, 0])
ldos.plot(label="Dirichlet", color="C2")
pb.pltutils.legend()
```



Computing the LDOS in graphene with a magnetic field reveals several peaks which correspond to Landau levels. The Jackson kernel produces the best results. The `broadening` argument of the calculation was set to 50 meV. With the Jackson kernel, the LDOS appears as if it was convolved with a Gaussian of that width. On the other hand, the Lorentz kernel applies an effective Lorentzian broadening of the same 50 meV but produces poorer results (not as sharp) simply due to the difference in slopes of the Gaussian and Lorentzian curves.

Lastly, there is the Dirichlet kernel. It essentially doesn't apply any damping and represent the raw result of the truncated Chebyshev series. Note that the Landau levels are still present, but there are also lots of extra oscillations (noise). The Dirichlet kernel is here mainly for demonstration purposes and is rarely useful.

Out of the two proper kernels, Jackson is the default and appropriate for most applications. The Lorentz kernels is mostly suited for Green's function (and thus also conductivity) or in cases where the extra smoothing of the Lorentzian may be preferable (sometimes purely aesthetically).

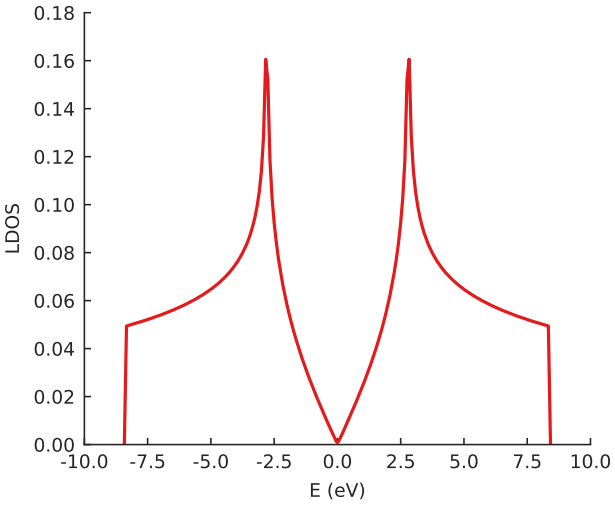
Low-level interface

The KPM-based calculation methods presented so far have been user-friendly and aimed at computing a single physical property of a model. Pybinding also offers a low-level KPM interface via the *`KPM.moments()`* method. It can be used to generally compute KPM expansion moments of the form $\mu_n =< \beta|op \cdot T_n(H)|\alpha >$. For more information on how to use these moments to reconstruct various functions, see Rev. Mod. Phys. 78, 275 (2006) which explains everything in great detail.

We'll just leave a quick example here. The following code calculates the LDOS in the center of a rectangular graphene flake. This is exactly like the first example in the LDOS section above, except that we are using the low-level interface. There is no special advantage to doing this calculation manually (in fact, the high-level method is faster). This is here simply for demonstration. The intended usage of the low-level interface is to create KPM-based computation methods which are not already covered by the builtins described above.

```
model = pb.Model(graphene.monolayer(), pb.rectangle(60, 60))
kpm = pb.kpm(model, kernel=pb.jackson_kernel())

# Construct a unit vector which is equal to 1 at the position
# where we want to calculate the local density of states
idx = model.system.find_nearest(position=[0, 0], sublattice="A")
alpha = np.zeros(model.hamiltonian.shape[0])
alpha[idx] = 1

# The broadening and the kernel determine the needed number of moments
a, b = kpm.scaling_factors
broadening = 0.05  # (eV)
num_moments = kpm.kernel.required_num_moments(broadening / a)

# Main calculation
```

```
moments = kpm.moments(num_moments, alpha)  # optionally also takes beta and an␣
↪operator

# Reconstruct the LDOS function
energy = np.linspace(-8.42, 8.42, 200)
scaled_energy = (energy - b) / a
ns = np.arange(num_moments)
k = 2 / (a * np.pi * np.sqrt(1 - scaled_energy**2))
chebyshev = np.cos(ns * np.arccos(scaled_energy[:, np.newaxis]))
ldos = k * np.sum(moments.real * chebyshev, axis=1)

plt.plot(energy, ldos)
plt.xlabel("E (eV)")
plt.ylabel("LDOS")
pb.pltutils.despine()
```



Further reading

For an additional examples see the *Magnetic field* subsection of *Fields and effects* as well as the *Strain modifier* subsection of *Defects and strain*. The reference page for the `chebyshev` submodule contains more information.

Scattering model

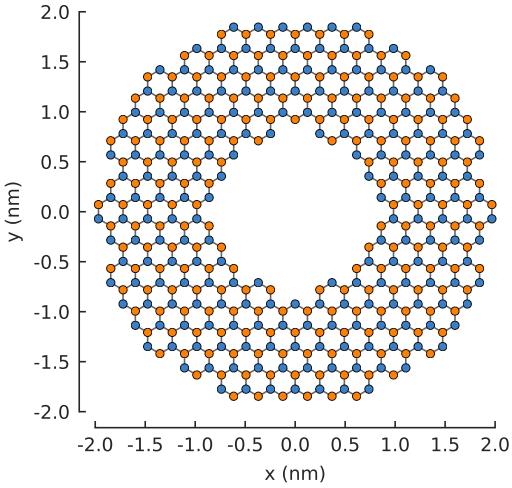
This section introduces the ability to attach semi-infinite leads to a finite-sized central region, thereby creating a scattering model.

Attaching leads

To start with, we need a finite-sized system to serve as the central scattering region. We'll just make a simple ring. Refer to the *Finite size* section for more details.

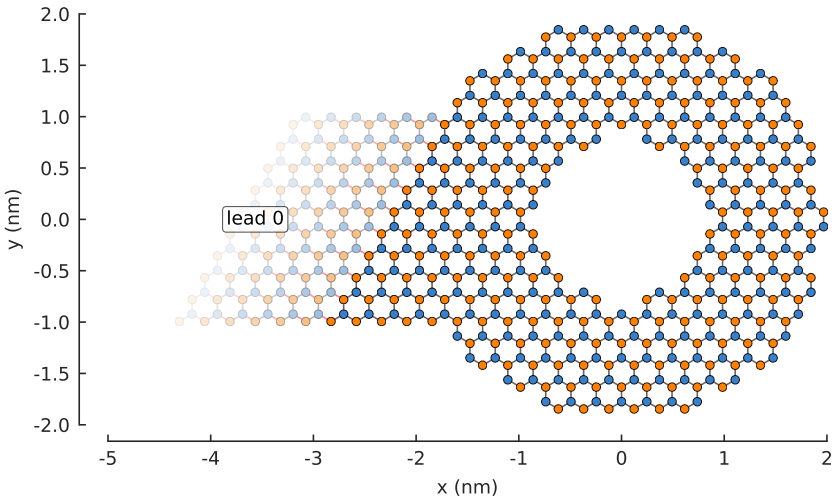
```
from pybinding.repository import graphene

def ring(inner_radius, outer_radius):
    """A simple ring shape"""
    def contains(x, y, z):
        r = np.sqrt(x**2 + y**2)
        return np.logical_and(inner_radius < r, r < outer_radius)
    return pb.FreeformShape(contains, width=[2*outer_radius, 2*outer_radius])
```

```
model = pb.Model(graphene.monolayer(), ring(0.8, 2))
model.plot()
```



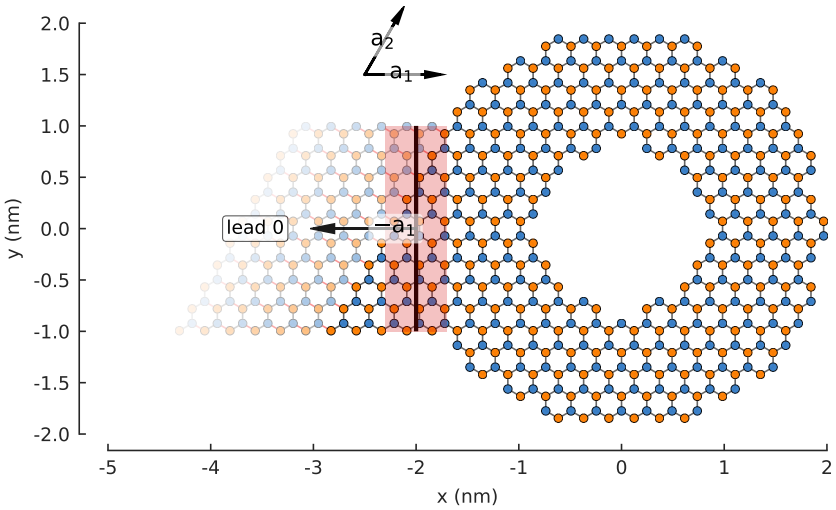
To attach a lead to this system, we call the *Model.attach_lead()* method:

```
model.attach_lead(direction=-1, contact=pb.line([-2, -1], [-2, 1]))

plt.figure(figsize=(6, 3))  # make the figure wider
model.plot()
```



The lead is semi-infinite, but to be practical for the figure, only a few repetitions of the lead's unit cell are drawn. They fade out gradually along the direction where the lead goes to infinity. The periodic hoppings between the unit cells are shown in red. The label indicates that this lead has the index 0. It's attributes can be accessed using this index and the *Model.leads* list. The lead was created using two parameters: `direction` and the `contact` shape. To illustrate the meaning of these parameters, we'll draw them using the *Lead.plot_contact()* method:

```
plt.figure(figsize=(6, 3))  # make the figure wider
model.plot()
model.leads[0].plot_contact()  # red shaded area and arrow
model.lattice.plot_vectors(position=[-2.5, 1.5], scale=3)
```

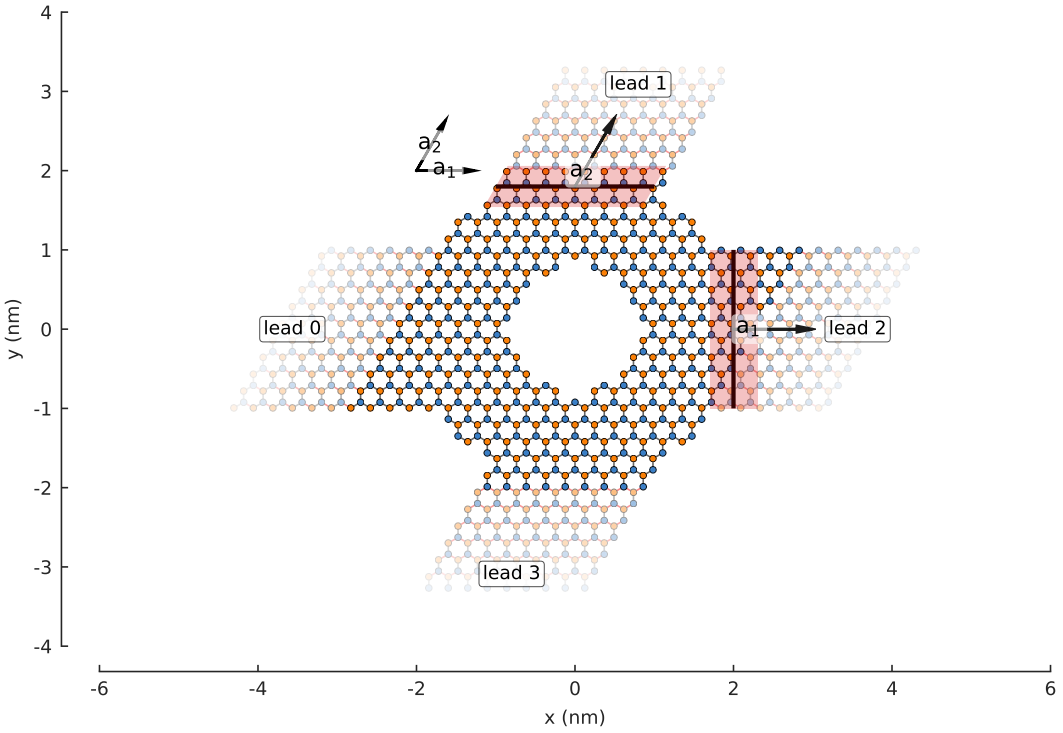


The direction of a lead is specified in terms of lattice vectors. In this case `direction=-1` indicates that it should be opposite the a_1 lattice vector, as shown in the figure with the arrow labeled $-a_1$. For 2D systems, the allowed directions are $\pm 1, \pm 2$. The position of the lead is chosen by specifying a `contact` shape. The intersection of a semi-infinite lead and a 2D system is a 1D line, which is why we specified `contact=pb.line([-2, -1], [-2, 1])`, where the two parameters given to *`line()`* are point positions. The line is drawn in the figure above in the middle of the red shaded area (the red area itself does not have any physical meaning, it's just there to draw attention to the line).

Note: For a 3D system, the lead contact area would be 2D shape, which could be specified by a *`Polygon`* or a *`FreeformShape`*.

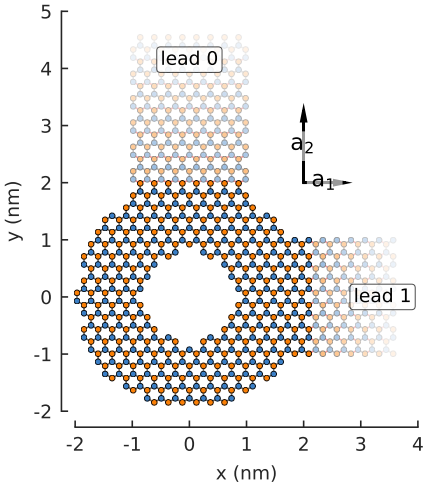
We can now proceed to attach a few more leads:

```
model.attach_lead(direction=+2, contact=pb.line([-1,  1.8], [1,  1.8]))
model.attach_lead(direction=+1, contact=pb.line([ 2, -1  ], [2,  1  ]))
model.attach_lead(direction=-2, contact=pb.line([-1, -1.8], [1, -1.8]))

plt.figure(figsize=(6.9, 6))
model.plot()
model.leads[1].plot_contact()
model.leads[2].plot_contact()
model.lattice.plot_vectors(position=[-2, 2], scale=3)
```



Notice that leads 1 and 3 are not perpendicular to leads 0 and 2. This is due to the angle of the primitive lattice vectors a_1 and a_2, as shown in the same figure. All of the leads also have zigzag edges because of this primitive vector arrangement. If we substitute the regular graphene lattice with `graphene.monolayer_4atom()`, the primitive vectors will be perpendicular and we'll get different leads in the ± 2 directions:

```
model = pb.Model(graphene.monolayer_4atom(), ring(0.8, 2))
model.attach_lead(direction=+2, contact=pb.line([-1,  1.8], [1, 1.8]))
model.attach_lead(direction=+1, contact=pb.line([ 2, -1  ], [2, 1  ]))
model.plot()
model.lattice.plot_vectors(position=[2, 2], scale=3)
```

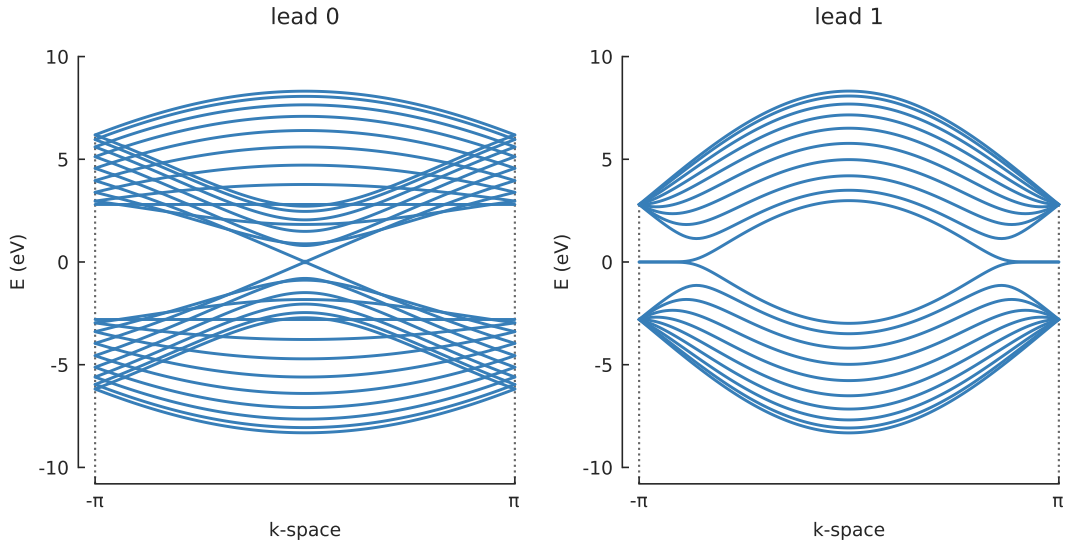


Lead attributes

The attached leads can be accessed using the *Model.leads* list. Each entry is a *Lead* object with a few useful attributes. The unit cell of a lead is described by the Hamiltonian *Lead.h0*. It's a sparse matrix, just like the *Model.hamiltonian* of finite-sized main system. The hoppings between unit cell of the lead are described by the *Lead.h1* matrix. See the *Lead* API page for more details.

Each lead also has a *Lead.plot_bands()* method which can be used to quickly view the band structure of an isolated lead. For the last model which was constructed and shown in the figure above, the band plots of the leads are:

```
plt.figure(figsize=(6.7, 3))
plt.subplot('121')
model.leads[0].plot_bands()
plt.subplot('122')
model.leads[1].plot_bands()
```

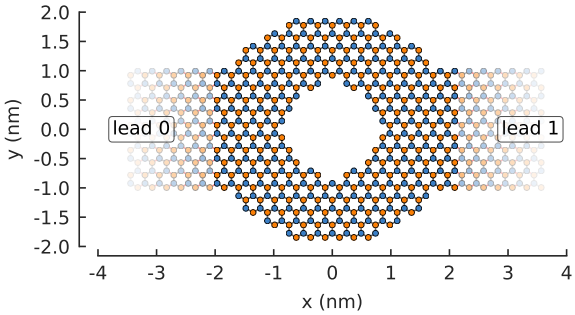


This is expected as lead 0 has armchair edges, while lead 1 has zigzag edges.

Fields in the leads

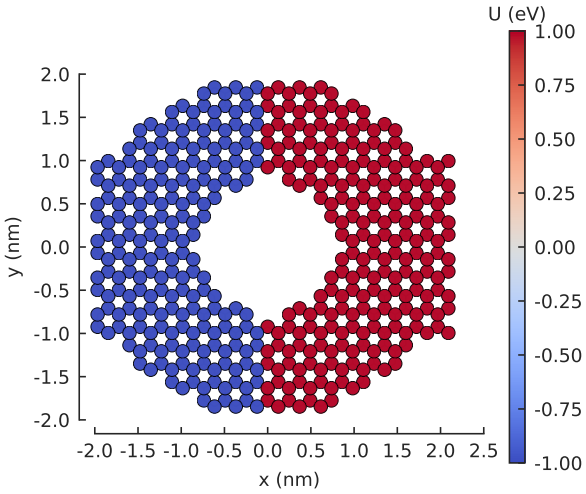
There is no need to specifically apply a field to a lead. Fields (and all modifier functions) are always applied globally to both the main system and all leads. For example, we can define a PN junction at $x_0 = 0$ and pass it to the model:

```
def pn_junction(x0, v1, v2):
    @pb.onsite_energy_modifier
    def potential(energy, x):
        energy[x < x0] += v1
        energy[x >= x0] += v2
        return energy
    return potential

model = pb.Model(
    graphene.monolayer_4atom(),
    ring(0.8, 2),
    pn_junction(x0=0, v1=-1, v2=1)
)
model.attach_lead(direction=-1, contact=pb.line([-2, -1], [-2,  1]))
model.attach_lead(direction=+1, contact=pb.line([ 2, -1], [ 2,  1]))
model.plot()
```



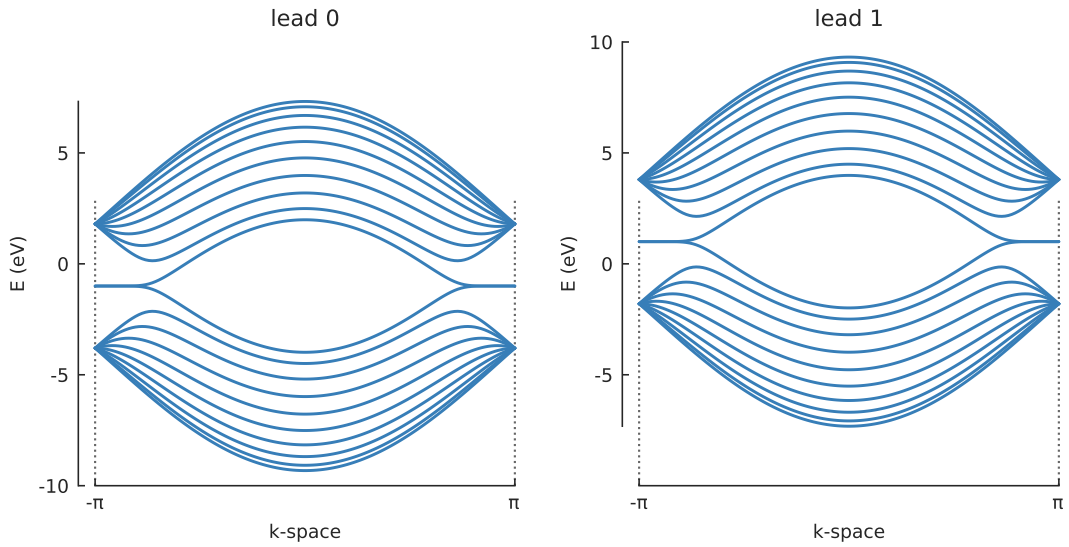
We can view the potential applied to the main system using the *Model.onsite_map* property.

```
model.onsite_map.plot(cmap="coolwarm", site_radius=0.06)
pb.pltutils.colorbar(label="U (eV)")
```



The appropriate potential is automatically applied to the leads depending on their position, left or right of the PN junction. We can quickly check this by plotting the band structure:

```
plt.figure(figsize=(6.7, 3))
plt.subplot('121')
model.leads[0].plot_bands()
plt.ylim(-10, 10)
plt.subplot('122')
model.leads[1].plot_bands()
plt.ylim(-10, 10)
```



The leads are identical, except for a ± 1 eV shift due to the PN junction, as expected.

Solving a scattering problem

At this time, pybinding doesn't have a builtin solver for scattering problems. However, they can be solved using Kwant. An arbitrary model can be constructed in pybinding and then exported using the `Model.tokwant()` method. See the *Kwant compatibility* page for details.

Alternatively, any user-defined solver and/or computation routine can be used. Pybinding generates the model information in a standard CSR matrix format. The required Hamiltonian matrices are `Model.hamiltonian` for the main scattering region and `Lead.h0` and `Lead.h1` for each of the leads found in `Model.leads`. For more information see the `Model` and `Lead` API reference pages.

CHAPTER 9

Additional Topics

This section will deal with a few of the more advanced features of pybinding. It is assumed that you are already familiar with the *Tutorial*.

Lattice specification

This section covers a few extra features of the `Lattice` class. It is assumed that you are already familiar with the *Tutorial*.

First, we set a few constants which are going to be needed in the following examples:

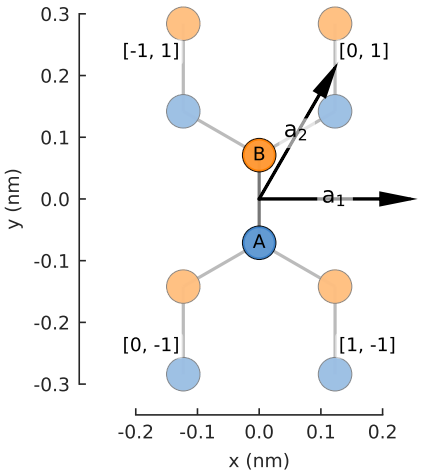
```
from math import sqrt, pi

a = 0.24595   # [nm] unit cell length
a_cc = 0.142  # [nm] carbon-carbon distance
t = -2.8      # [eV] nearest neighbour hopping

Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]
```

Intrinsic onsite energy

During the construction of a `Lattice` object, the full signature of a sublattice is (`name`, `offset`, `onsite_energy=0.0`), where the last argument is optional. The `name` and `offset` arguments were already explained in the basic tutorial. The `onsite_energy` is applied as an intrinsic part of the sublattice site. As an example, we'll add this term to monolayer graphene:

```
def monolayer_graphene(onsite_energy=[0, 0]):
    lat = pb.Lattice(a1=[a, 0], a2=[a/2, a/2 * sqrt(3)])
    lat.add_sublattices(('A', [0, -a_cc/2], onsite_energy[0]),
                        ('B', [0,  a_cc/2], onsite_energy[1]))
    lat.add_hoppings(([0,  0], 'A', 'B', t),
                     ([1, -1], 'A', 'B', t),
                     ([0, -1], 'A', 'B', t))
```

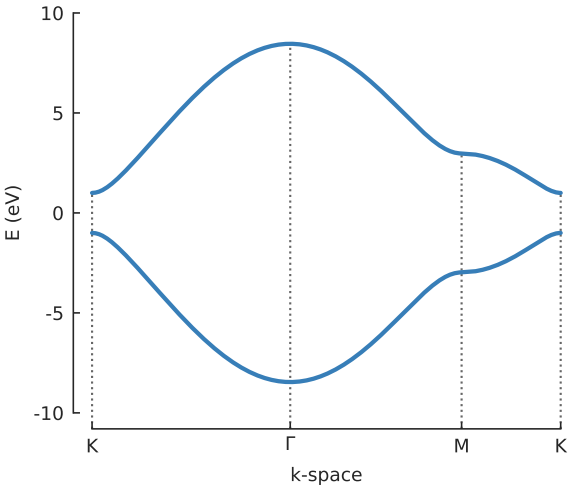
```
    return lat

lattice = monolayer_graphene()
lattice.plot()
```



Note: See *Lattice.add_one_sublattice()* and *Lattice.add_sublattices()*.

The effect of the onsite energy becomes apparent if we set opposite values for the A and B sublattices. This opens a band gap in graphene:

```
model = pb.Model(
    monolayer_graphene(onsite_energy=[-1, 1]),  # eV
    pb.translational_symmetry()
)
solver = pb.solver.lapack(model)
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```



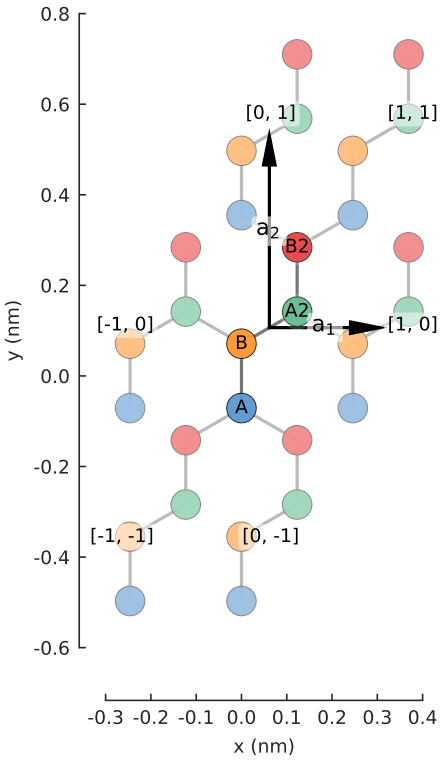
An alternative way of doing this was covered in the *Opening a band gap* section of the basic tutorial. There, an *@onsite_energy_modifier* was used to produce the same effect. The modifier is applied only after the system is constructed so it can depend on the final (x, y, z) coordinates. Conversely, when the onsite energy is specified directly in a *Lattice* object, it models an intrinsic part of the lattice and cannot depend on position. If both the intrinsic energy and the modifier are specified, the values are added up.

Constructing a supercell

A primitive cell is the smallest unit cell of a crystal. For graphene, this is the usual 2-atom cell. It's translated in space to construct a larger system. Sometimes it can be convenient to use a larger unit cell instead, i.e. a supercell consisting of multiple primitive cells. This allows us to slightly adjust the geometry of the lattice. For example, the 2-atom primitive cell of graphene has vectors at an acute angle with regard to each other. On the other hand, a 4-atom supercell is rectangular which makes certain model geometries easier to create. It also makes it possible to realize armchair edges, as shown in *Nanoribbons* section of the basic tutorial.

We can create a 4-atom cell by adding two more sublattice to the `Lattice` specification:

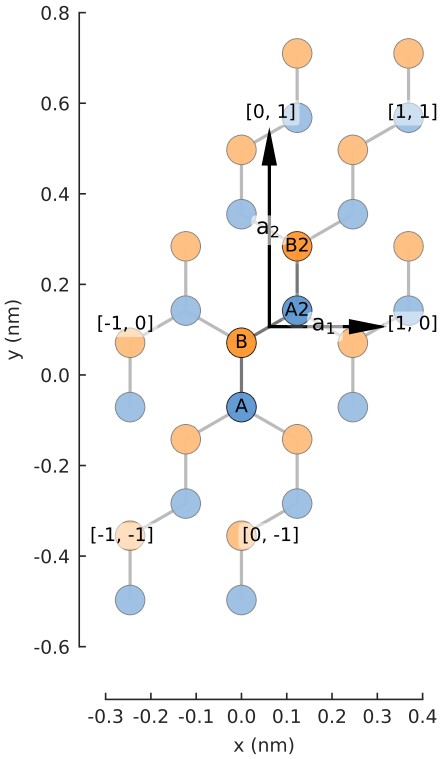
```
def monolayer_graphene_4atom():
    lat = pb.Lattice(a1=[a, 0], a2=[0, 3*a_cc])
    lat.add_sublattices(('A',  [  0, -a_cc/2]),
                        ('B',  [  0,  a_cc/2]),
                        ('A2', [a/2,    a_cc]),
                        ('B2', [a/2,  2*a_cc]))
    lat.add_hoppings(
        # inside the unit sell
        ([0, 0], 'A',  'B',  t),
        ([0, 0], 'B',  'A2', t),
        ([0, 0], 'A2', 'B2', t),
        # between neighbouring unit cells
        ([-1, -1], 'A', 'B2', t),
        ([ 0, -1], 'A', 'B2', t),
        ([-1,  0], 'B', 'A2', t),
    )
    return lat

lattice = monolayer_graphene_4atom()
plt.figure(figsize=(5, 5))
lattice.plot()
```



Note the additional sublattices A2 and B2, shown in green and red in the figure. As defined above, these are interpreted as new and distinct lattice sites. However, we would like to have sublattices A2 and B2 be equivalent

to A and B. *Lattice.add_aliases()* does exactly that:

```
def monolayer_graphene_4atom():
    lat = pb.Lattice(a1=[a, 0], a2=[0, 3*a_cc])
    lat.add_sublattices(('A',  [  0, -a_cc/2]),
                        ('B',  [  0,  a_cc/2]))
    lat.add_aliases(('A2', 'A', [a/2, a_cc]),
                    ('B2', 'B', [a/2, 2*a_cc]))
    lat.add_hoppings(
        # inside the unit sell
        ([0, 0], 'A',  'B',  t),
        ([0, 0], 'B',  'A2', t),
        ([0, 0], 'A2', 'B2', t),
        # between neighbouring unit cells
        ([-1, -1], 'A', 'B2', t),
        ([ 0, -1], 'A', 'B2', t),
        ([-1,  0], 'B', 'A2', t),
    )
    return lat

lattice = monolayer_graphene_4atom()
plt.figure(figsize=(5, 5))
lattice.plot()
```

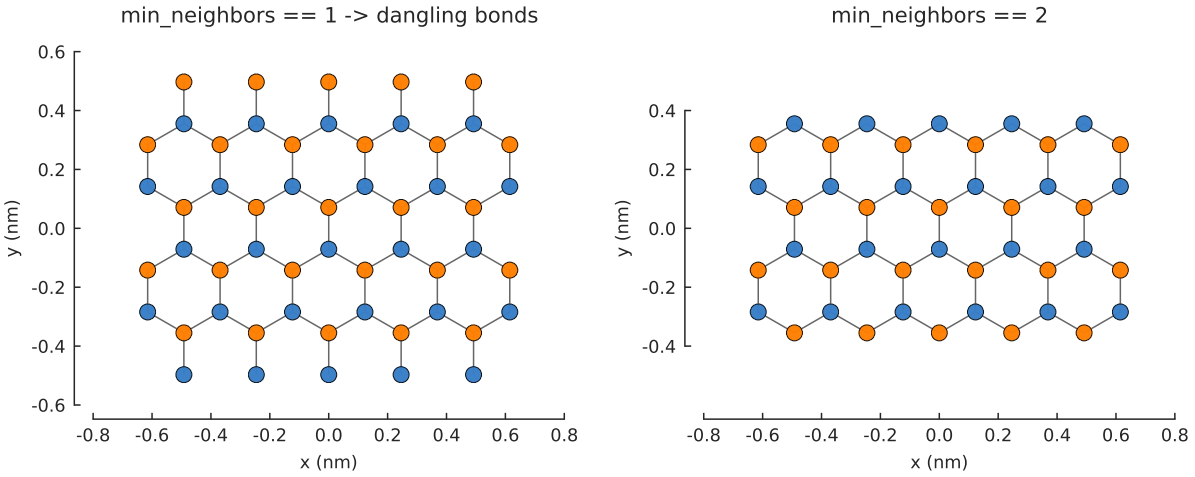


Now we have a supercell with only two unique sublattices: A and B. The 4-atom graphene unit cell is rectangular which makes it a more convenient building block than the oblique 2-atom cell.

Removing dangling bonds

When a finite-sized graphene system is constructed, it's possible that it will contain a few dangling bonds on the edge of the system. These are usually not desired and can be removed easily by setting the *Lattice.min_neighbors* attribute:

```
plt.figure(figsize=(8, 3))
lattice = monolayer_graphene()
shape = pb.rectangle(x=1.4, y=1.1)

plt.subplot(121, title="min_neighbors == 1 -> dangling bonds")
model = pb.Model(lattice, shape)
model.plot()

plt.subplot(122, title="min_neighbors == 2", ylim=[-0.6, 0.6])
model = pb.Model(lattice.with_min_neighbors(2), shape)
model.plot()
```

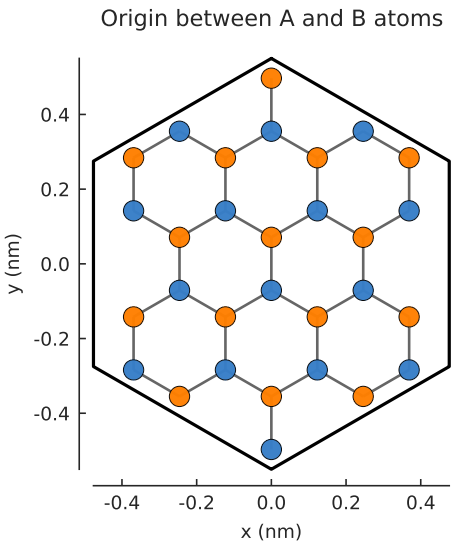


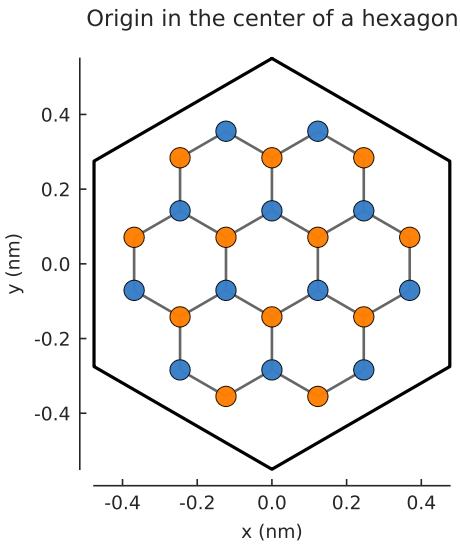
The dangling atoms on the edges have only one neighbor which makes them unique. When we use the *Lattice.with_min_neighbors()* method, the model is required to remove any atoms which have less than the specified minimum number of neighbors. Note that setting *min_neighbors* to 3 would produce an empty system since it is impossible for all atoms to have at least 3 neighbors.

Global lattice offset

When we defined `monolayer_graphene()` at the start of this section, we set the positions of the sublattices as $[x, y] = [0, \pm a_{cc}]$, i.e. the coordinate system origin is at the midpoint between A and B atoms. It can sometimes be convenient to choose a different origin position such as the center of a hexagon formed by the carbon atoms. Rather than define an entirely new lattice with different positions for A and B, we can simply offset the entire lattice by setting the *Lattice.offset* attribute:

```
plt.figure(figsize=(8, 3))
shape = pb.regular_polygon(num_sides=6, radius=0.55)

plt.subplot(121, title="Origin between A and B atoms")
model = pb.Model(monolayer_graphene(), shape)
model.plot()
model.shape.plot()

plt.subplot(122, title="Origin in the center of a hexagon")
model = pb.Model(monolayer_graphene().with_offset([a/2, 0]), shape)
model.plot()
model.shape.plot()
```





Note that the shape remains unchanged, only the lattice shifts position. We could have achieved the same result by only moving the shape, but then the center of the shape would not match the origin of the coordinate system. The *`Lattice.with_offset()`* makes it easy to position the lattice as needed. Note that the given offset must be within half the length of a primitive lattice vector (positive or negative). Beyond that length the lattice repeats periodically, so it doesn't make sense to shift it any father.

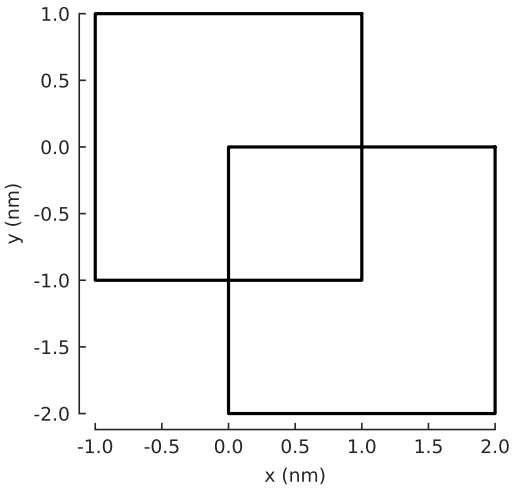
Composite shapes

The basic usage of shapes was explained in the *Finite size* section of the tutorial. An overview of all the classes and function is available in the *Shapes* API reference. This section show how multiple of those shapes can be composed to quickly create intricate systems.

Moving shapes

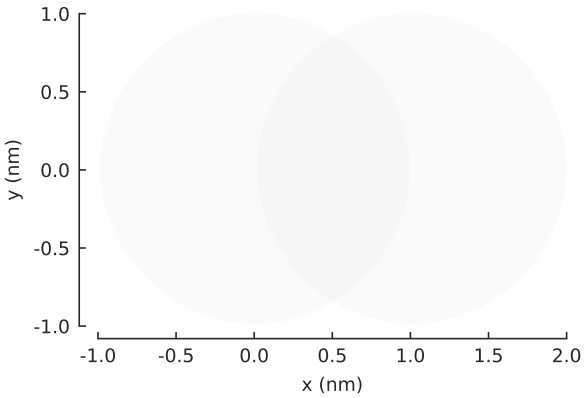
All shapes have a *`with_offset()`* method which simply translates the shape by a vector:

```
shape = pb.rectangle(2, 2)
translated_shape = shape.with_offset([1, -1])
shape.plot()
translated_shape.plot()
```



This applies to any kind of shape, including user-defined freeform shapes:

```
def circle(radius):
    def contains(x, y, z):
        return np.sqrt(x**2 + y**2) < radius
    return pb.FreeformShape(contains, width=[2*radius, 2*radius])

shape = circle(1)
translated_shape = shape.with_offset([1, 0])
shape.plot()
translated_shape.plot()
```

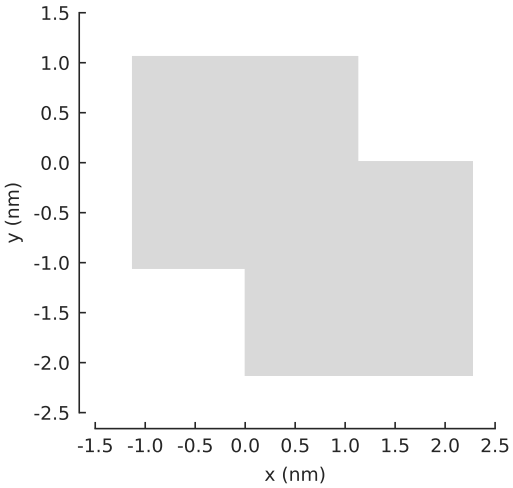


Note that *Polygon* and *FreeformShape* are presented differently in the plots. For polygons, a line which connects all vertices is plotted. Freeform shapes are shown as a lightly shaded silhouette which is filled in by calling the `contains` function and placing dark pixels at positions where it returned `True`.

Using set operations

In the examples above we placed 2 shapes so that they overlap, but those were only plots. In order to create a composite shape, we can use logical and arithmetic operator. For example, addition:

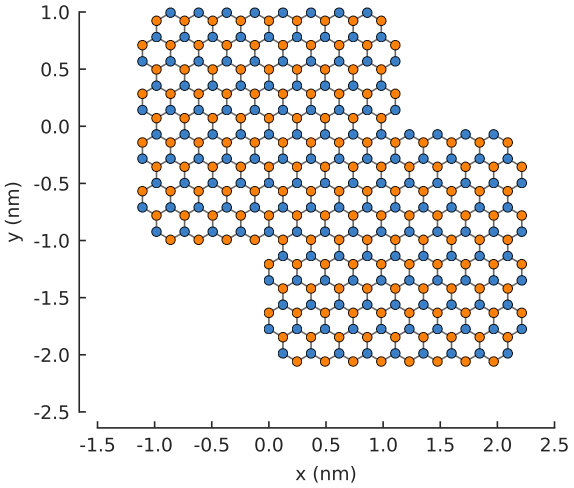
```
s1 = pb.rectangle(2.3, 2.15)
s2 = s1.with_offset([1.12, -1.05])

composite_shape = s1 + s2
composite_shape.plot()
```



Note that even though we have combined two polygons, the composite shape is plotted in the style of a freeform shape. This is intentional to allow making completely generic shapes.

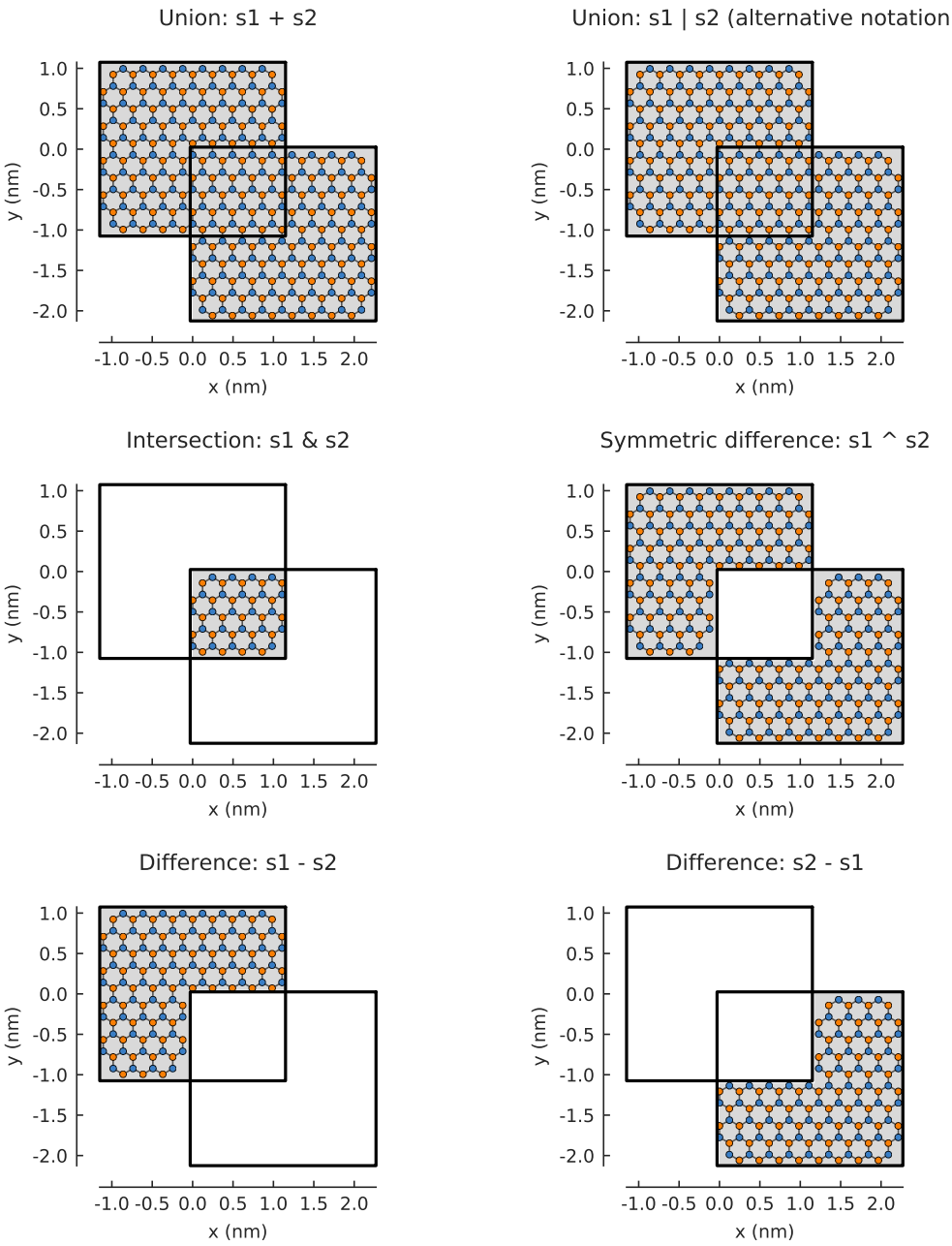
The + operator creates a union of the two shapes and the result can be used with a model:

```
from pybinding.repository import graphene

model = pb.Model(graphene.monolayer(), composite_shape)
model.plot()
```



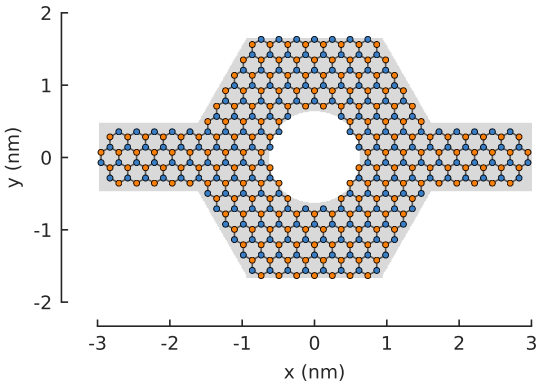
Shapes are composed in terms of set operations (e.g. unions, intersections) and the syntax mirrors that of Python's builtin `set`. The available operators and their results are shown in the code and figure below. Note that the `+` and `|` operators perform the same function (union). Both are available simply for convenience. Apart from `-`, all the operators are symmetric.

```
grid = plt.GridSpec(3, 2, hspace=0.4)
plt.figure(figsize=(6.7, 8))

titles_and_shapes = [
    ("Union: s1 + s2",                          s1 + s2),
    ("Union: s1 | s2 (alternative notation)",   s1 | s2),
    ("Intersection: s1 & s2",                   s1 & s2),
    ("Symmetric difference: s1 ^ s2",           s1 ^ s2),
    ("Difference: s1 - s2",                     s1 - s2),
    ("Difference: s2 - s1",                     s2 - s1)
]

for g, (title, shape) in zip(grid, titles_and_shapes):
    plt.subplot(g, title=title)
    s1.plot()
    s2.plot()
    model = pb.Model(graphene.monolayer(), shape)
    model.shape.plot()
    model.plot()
```



This isn't limited to just two operands. Any number of shapes can be freely combined:

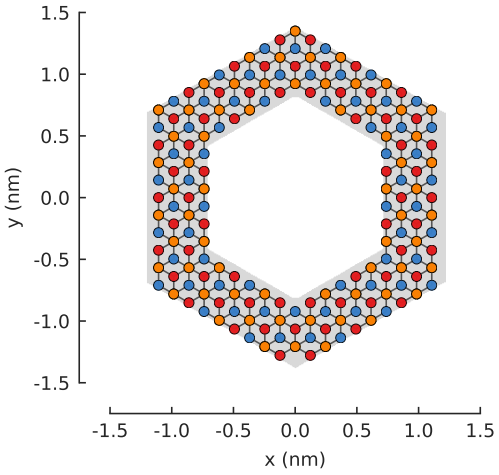
```
from math import pi

rectangle = pb.rectangle(x=6, y=1)
hexagon = pb.regular_polygon(num_sides=6, radius=1.92, angle=pi/6)
circle = pb.circle(radius=0.6)

model = pb.Model(
    graphene.monolayer(),
    (rectangle + hexagon) ^ circle
)
model.shape.plot()
model.plot()
```



Additional examples

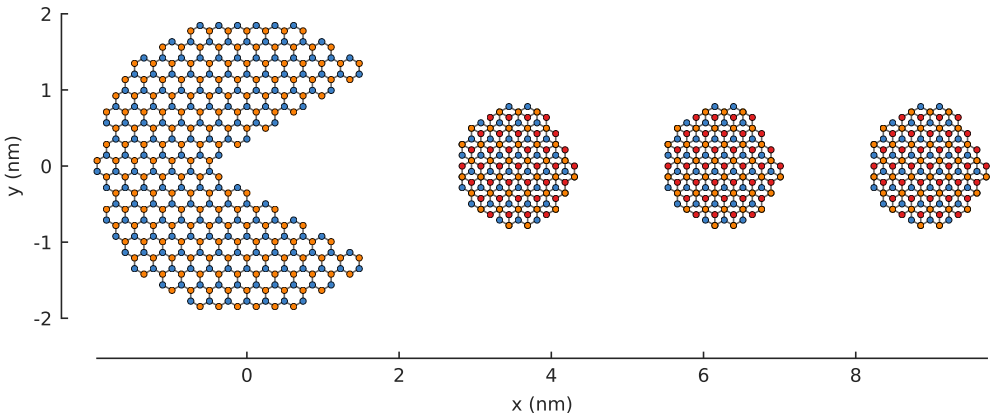
Circular rings are easy to create even with a *FreeformShape*, but composites make it trivial to create rings as the difference of any two shapes:

```
outer = pb.regular_polygon(num_sides=6, radius=1.4)
inner = pb.regular_polygon(num_sides=6, radius=0.8)
model = pb.Model(graphene.bilayer(), outer - inner)
model.shape.plot()
model.plot()
```



Of course, we can also go a bit wild:

```
plt.figure(figsize=(6.7, 2.6))

circle = pb.circle(radius=2)
triangle = pb.regular_polygon(num_sides=3, radius=2, angle=pi / 6).with_offset([1.
↪4, 0])
pm = pb.Model(graphene.monolayer(), circle - triangle)
pm.plot()

dot = pb.circle(radius=0.8)
for x in [3.55, 6.25, 8.95]:
    pd = pb.Model(graphene.bilayer(), dot.with_offset([x, 0]))
    pd.plot()
```



Multi-orbital models

In pybinding, if an onsite or hopping energy term is defined as a matrix (instead of a scalar), we refer to the resulting model as *multi-orbital*. The elements of the matrix term may correspond to different spins, electrons and holes, or any other degrees of freedom. These can have different physical meaning depending on the intend of the model. Because we're talking in generic terms here, we'll use *orbital* as a blanket term to refer to any degree of freedom, i.e. matrix element of an onsite or hopping term.

This section describes how these models can be defined and how the presence of multiple orbitals affects modifier functions and the results obtained from solvers. In general, it is as simple as replacing a scalar value with a matrix while all of the principals described in the *Tutorial* still apply.

Onsite and hopping matrices

Starting from the very beginning, the orbital count of a site is determined by the shape of the onsite energy matrix. Let's take a look at a few possibilities:

```
lat = pb.Lattice([1, 0], [0, 1])
lat.add_sublattices(
    ("A", [0.0, 0.0], 0.5),           # single-orbital: scalar
    ("B", [0.0, 0.2], [[1.5,  2j],    # two-orbital: 2x2 Hermitian matrix
                       [-2j, 1.5]]),
    ("C", [0.3, 0.1], np.zeros(2)),   # two-orbital: zero onsite term
    ("D", [0.1, 0.0], [[4, 0, 0],     # three-orbital: only diagonal
                       [0, 5, 0],
                       [0, 0, 6]]),
    ("E", [0.2, 0.2], [4, 5, 6])      # three-orbital: only diagonal, terse notation
)
```

The onsite term is required to be a square Hermitian matrix. If a 1D array is given instead of a matrix, it will be interpreted as the main diagonal of a square matrix (see sublattices D and E which have identical onsite term specified with different notations).

As seen above, sublattices don't need to all have the same orbital count. The only thing to keep in mind is that the hopping matrix which connect a pair of sublattice sites must have the appropriate shape: the number of rows must match the orbital count of the source sublattice and the number of columns must match the destination sublattice.

```
lat.add_hoppings(
    ([0, 1], "A", "A", 1.2),          # scalar
    ([0, 1], "B", "B", [[1, 2],       # 2x2
                        [3, 4]]),
    ([0, 0], "B", "C", [[2j, 0],      # 2x2
                        [1j, 0]]),
```

```
    ([0, 0], "A", "D", [[1, 2, 3]]),  # 1x3
    ([0, 1], "D", "A", [[7],          # 3x1
                        [8],
                        [9]]),
    ([0, 0], "B", "D", [[1j, 0,  0],  # 2x3
                        [2,  0, 3j]])
)
```

If a matrix of the wrong shape is given, an informative error is raised:

```
>>> lat.add_one_hopping([0, 0], "A", "B", 0.6)
RuntimeError: Hopping size mismatch: from 'A' (1) to 'B' (2) with matrix (1, 1)
>>> lat.add_one_hopping([0, 1], "D", "D", [[1, 2, 3],
...                                        [4, 5, 6]])
RuntimeError: Hopping size mismatch: from 'D' (3) to 'D' (3) with matrix (2, 3)
```

After the *Lattice* is complete, a *Model* can be built as usual:

```
>>> model = pb.Model(lat, pb.primitive(2, 2))
>>> model.system.num_sites
20  # <-- 5 sites per unit cell and 2x2 cells: 5*2*2 == 20
>>> model.hamiltonian.shape
(44, 44)  # <-- 11 (1+2+2+3+3) orbitals per unit cell and 2x2 cells: 11*2*2 = 44
```

Sites refer to physical locations so their total count corresponds to the number of sublattices (A to E) multiplied by the number of times the unit cell is repeated. The Hamiltonian matrix is larger than `num_sites` due to the extra orbitals.

Effect on modifier functions

The *@onsite_energy_modifier* and *@hopping_energy_modifier* functions work equally well for single- and multi-orbital models. In case of the latter, the `energy` argument of the modifiers will have a shape matching the onsite/hopping matrix term.

```
@pb.onsite_energy_modifier
def potential(energy, x):
    """Linear onsite potential as a function of x for a 2-orbital model"""
    return energy + np.eye(2) * x
```

Note the `np.eye(2)` in the code above. The number 2 matches the 2-orbital structure of a specific model. Without this, `energy + x` would also add the value to the off-diagonal elements of the onsite matrix which is not desirable in this case.

The modifier defined above will only work for 2-orbital models. In general, we might want to create modifiers which work with any n-orbital model or with a mixed number of orbitals. For this we can use the `sub_id` modifier argument and its `.eye` attribute which supplies the correct matrix shape for any sublattice:

```
@pb.onsite_energy_modifier
def potential(energy, x, sub_id):
    """Same as above, but works for any n-orbital model"""
    return energy + sub_id.eye * x
```

Even more generally, if we wish to apply completely different functions to the various sublattices, the `sub_id` argument can be used to create different branches in the modifier:

```
@pb.onsite_energy_modifier
def potential(energy, x, sub_id):
    """Applies different functions to different sublattices"""
    if sub_id == "A":
        return energy + x  # we know sublattice A is single-orbital
    elif sub_id == "D":
```

```
        energy[x > 0] += sub_id.eye * x  # the notation can be mixed with numpy␣
↪indexing
        return energy                    # apply only to sites where x > 0
    elif sub_id == "B":
        sigma_y = np.array([[0, -1j],
                            [1j,  0]])
        return energy + sigma_y * 1.3 - np.eye(2) * 0.6  # add multiple 2x2␣
↪matrices
    else:
        return energy  # leave the other sublattices unchanged
```

This branching behavior is only supported by the `sub_id` and `hop_id` arguments. Do not try to create branches like this using any of the other modifier arguments:

```
"""Creating a position-dependent potential"""
# This is an error with anything except sub_id or hop_id
if x > 0:
    return energy + 1
else:
    return energy - 1

# Use this notation instead
energy[x >  0] += 1
energy[x <= 0] -= 1
```

On the other hand, `sub_id` and `hop_id` can be used with either of these variants with just a single caveat:

```
"""Sublattice-dependent potential"""
# This always works with sub_id and hop_id
if sub_id == "A":
    return energy + 1
else:
    return energy - 1

# This only works when all sublattices have the same number of orbitals,
# but it will raise an error for mixed orbital counts.
energy[sub_id == "A"] += 1
energy[sub_id == "B"] -= 1
```

Local properties and plotting

When examining the local properties of a multi-orbital model, it is important to make the distinction between system indices which correspond to sites (unique positions) and Hamiltonian indices which correspond to the onsite or hopping terms in the Hamiltonian.

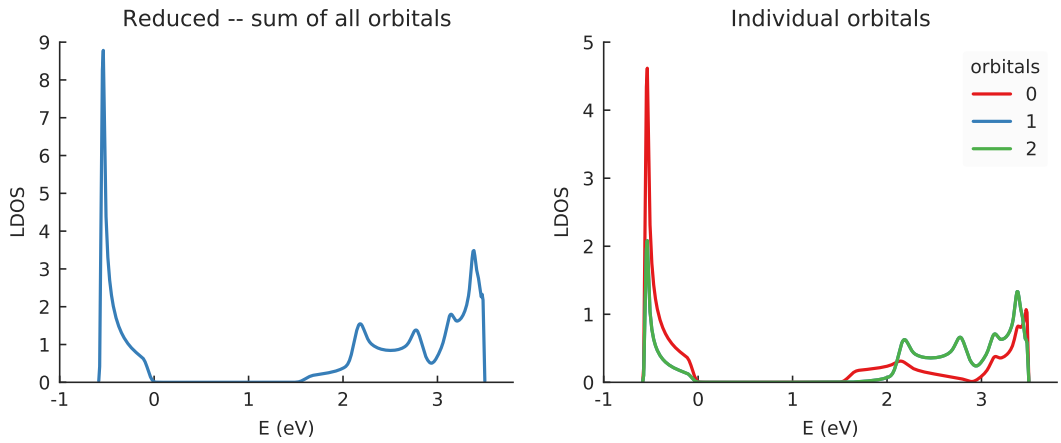
As shown in one of the previous examples, the number of sites in a system does not have to be equal to the size of the Hamiltonian matrix (`hamiltonian.shape[0] >= num_sites`). This affects how the system and Hamiltonian are indexed. System indices are always scalars and point to a single site position. For single-orbital models there is a 1:1 correspondence between system and Hamiltonian indices. However, for multi-orbital models the Hamiltonian indices are 1D arrays with a size corresponding to the number of orbitals on the target site.

```
>>> model = pb.Model(lat, pb.primitive(2, 2))
>>> sys_idx = model.system.find_nearest(position=[0, 0], sublattice="D")
>>> sys_idx  # <-- Points to a site on sublattice D which is closest to the target␣
↪position.
15           #     It's always a scalar.
>>> model.system.x[sys_idx]
0.1  # <-- Not exactly 0 as requested, but the closest site to it.
>>> model.system.y[sys_idx]
0.0
```

```
>>> ham_idx = model.system.to_hamiltonian_indices(sys_idx)
>>> ham_idx   # <-- Array of integers which can be used to index the Hamiltonian␣
↪matrix.
[29, 30, 31]  #     Size 3 because the selected site is on the 3-orbital␣
↪sublattice D.
>>> ham = model.hamiltonian.todense()
>>> ham[np.ix_(ham_idx, ham_idx)]  # Returns the onsite hopping term of sublattice␣
↪D.
[[4, 0, 0],
 [0, 5, 0],
 [0, 0, 6]]
```

Functions which compute various local properties take into account the presence of multiple orbitals on a single site. For example, when calculating the local density of states, one of the input parameters is the target site position. By default, the resulting LDOS is calculated as the sum of all orbitals but this is optional as shown in the following example:

```
"""Calculate the LDOS in the center of a MoS2 quantum dot"""
from pybinding.repository import group6_tmd

model = pb.Model(group6_tmd.monolayer_3band("MoS2"),
                 pb.regular_polygon(6, 20))

kpm = pb.kpm(model)
energy = np.linspace(-1, 3.8, 500)
broadening = 0.05
position = [0, 0]

plt.figure(figsize=(6.7, 2.3))

plt.subplot(121, title="Reduced -- sum of all orbitals")
ldos = kpm.calc_ldos(energy, broadening, position)
ldos.plot(color="C1")

plt.subplot(122, title="Individual orbitals")
ldos = kpm.calc_ldos(energy, broadening, position, reduce=False)
ldos.plot()
```



Kwant compatibility

Kwant is a Python package for numerical tight-binding similar to pybinding, but it's specialized for transport calculations. Since the two packages work with the same kind of Hamiltonian matrices, it's possible to build a model in pybinding and use Kwant to compute the transport properties. The advantage for pybinding users is access to Kwant's transport solvers in addition to pybinding's builtin *computational routines*. The advantage

for Kwant users is the much faster system build times made possible by pybinding's model builder – see the *Benchmarks*.

Exporting a model

The procedure for constructing and solving transport problems in Kwant can be summarized with the following lines of pseudo-code:

```
# 1. BUILD model system
builder = kwant.Builder()
...  # specify model parameters
system = builder.finalized()

# 2. COMPUTE scattering matrix
smatrix = kwant.smatrix(system)
...  # call smatrix methods
```

If we want to use pybinding to build the model, we can just replace the first part:

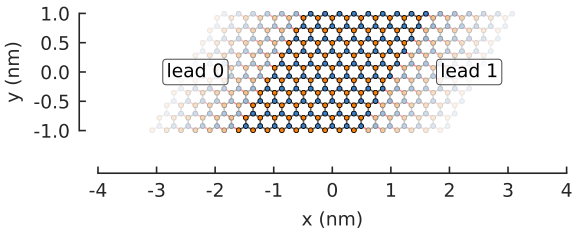
```
# 1. BUILD model system
model = pb.Model(...)  # specify model parameters
kwant_system = model.tokwant()  # export to kwant format

# 2. COMPUTE scattering matrix
smatrix = kwant.smatrix(kwant_system)
...  # call smatrix methods
```

A pybinding *Model* is defined as usual and then converted to the Kwant-compatible format by calling the *Model.tokwant()* method. The resulting `kwant_system` can be used as expected.

Complete example

A detailed overview of scattering model construction in pybinding is available in the *tutorial*. Here, we present a simple example of a graphene wire with a potential barrier:

```
from pybinding.repository import graphene

def potential_barrier(v0, x0):
    """Barrier height `v0` in eV with spatial position `-x0 <= x <= x0`"""
    @pb.onsite_energy_modifier(is_double=True)  # enable double-precision floating-
↪point
    def function(energy, x):
        energy[np.logical_and(-x0 <= x, x <= x0)] = v0
        return energy
    return function

def make_model(length, width, v0=0):
    model = pb.Model(
        graphene.monolayer(),
        pb.rectangle(length, width),
        potential_barrier(v0, length / 4)
    )
    model.attach_lead(-1, pb.line([-length/2, -width/2], [-length/2, width/2]))
    model.attach_lead(+1, pb.line([ length/2, -width/2], [ length/2, width/2]))
    return model

model = make_model(length=1, width=2)  # nm
model.plot()
```



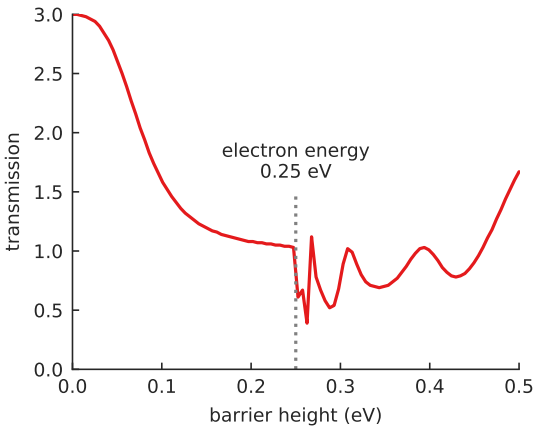
We can then vary the height of the potential barrier and calculate the transmission using Kwant:

```
import kwant

length, width = 15, 15  # nm
electron_energy = 0.25  # eV
barrier_heights = np.linspace(0, 0.5, 100)  # eV

transmission = []
for v in barrier_heights:
    model = make_model(length, width, v)  # pybinding model
    kwant_system = model.tokwant()  # export to kwant
    smatrix = kwant.smatrix(kwant_system, energy=electron_energy)
    transmission.append(smatrix.transmission(1, 0))
```

For more information about `kwant.smatrix` and other transport calculations, please refer to the Kwant website. That is outside the scope of this guide. The purpose of this section is to present the *`Model.tokwant()`* compatibility method. The exported system is then in the domain of Kwant.

From there, it's trivial to plot the results:

```
plt.plot(barrier_heights, transmission)
plt.ylabel("transmission")
plt.xlabel("barrier height (eV)")
plt.axvline(electron_energy, 0, 0.5, color="gray", linestyle=":")
plt.annotate("electron energy\n{} eV".format(electron_energy), (electron_energy, 0.
↪54),
             xycoords=("data", "axes fraction"), horizontalalignment="center")
pb.pltutils.despine()  # remove top and right axis lines
```



Note that the transmission was calculated for an energy value of 0.25 eV. As the height of the barrier is increased, two regimes are clearly distinguishable: transmission over and through the barrier.

Performance considerations

The Kwant documentation recommends separating model parameters into two parts: the structural data which remains constant and fields which can be varied. This yields better performance because only the field data needs

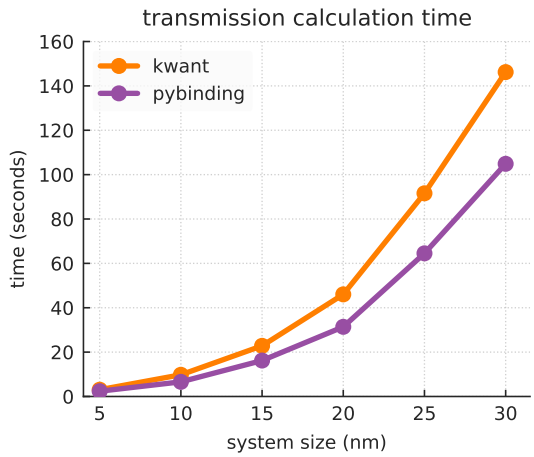
to be repopulated. This is demonstrated with the following pseudo-code which loops over some parameter `x`:

```
builder = kwant.Builder()
...  # specify structural parameters
system = builder.finalized()

for x in xs:
    smatrix = kwant.smatrix(system, args=[x])  # apply fields
    ...  # call smatrix methods
```

This separation is not required with pybinding. As pointed out in the *Benchmarks*, the fast builder makes it possible to fully reconstruct the model in every loop iteration at no extra performance cost. This simplifies the code since all the parameters can be applied in a single place:

```
def make_model(x):
    return pb.Model(..., x)  # all parameters in one place

for x in xs:
    smatrix = kwant.smatrix(make_model(x).tokwant())  # constructed all at once
    ...  # call smatrix methods
```

You can `download` a full example file which implements transport through a barrier like the one presented above. The script uses both builders so you can compare the implementation as well as the performance. Download the example file and try it on your system. Our results are presented below (measured using Intel Core i7-4960HQ CPU, 16 GiB RAM, Python 3.5, macOS 10.11). The size of the square scattering region is increased and we measure the total time required to calculate the transmission:



For each system size, the transmission is calculated as a function of barrier height for 100 values. Even though pybinding reconstructs the entire model every time the barrier is changed, the system build time is so fast that it doesn't affect the total calculation time. In fact, the extremely fast build actually enables pybinding to outperform Kwant in the overall calculation. Even though Kwant only repopulates field data at each loop iteration, this still takes more time than it does for pybinding to fully reconstruct the system.

Note that this example presents a relatively simple system with a square barrier. This is done to keep the run time to only a few minutes, for convenience. Here, pybinding speeds up the overall calculation by about 40%. For more realistic examples with larger scattering regions and complicated field functions with multiple parameters, a speedup of 3-4 times can be achieved by using pybinding's model builder.

Floating-point precision

Pybinding can generate the Hamiltonian matrix with one of four data types: real or complex numbers with single or double precision (32-bit or 64-bit floating point). The selection is dynamic. The starting case is always real with single precision and from there the data type is automatically promoted as needed by the model. For example, adding translationally symmetry or a magnetic field will cause the builder to switch to complex numbers – this is detected automatically. On the other hand, the switch to double precision needs to be requested

by the user. The onsite and hopping energy *modifiers* have an optional `is_double` parameter which can be set to `True`. The builder switches to double precision if requested by at least one modifier. Alternatively, *`force_double_precision()`* can be given to a *`Model`* as a direct parameter.

The reason for all of this is performance. Most solvers work faster with smaller data types: they consume less memory and bandwidth and SIMD vectorization becomes more efficient. This is assuming that single precision and/or real numbers are sufficient to describe the given model. In case of Kwant's solvers, it seems to require double precision in most cases. This is the reason for the `is_double=True` flag in the above example. Keep this in mind when exporting to Kwant.

CHAPTER 10

Plotting Guide

All of the plotting functions in pybinding are create using matplotlib. This means that you can customize the appearance of the figures using standard matplotlib commands. However, some plots (like lattice structure) are specialized to tight-binding models and have some additional options in contrast to ordinary plot templates (line plot, scatter, quiver, etc.). This guide will present the workflow for customizing figures in pybinding.

You can also create your own figures from scratch using just the raw data from pybinding. However, it is far more convenient to use pybinding's builtin plot methods as a base and use matplotlib's API to customize as needed. The builtin methods have already taken care of most of the work needed to represent arbitrary tight-binding models and their properties. This is done in the most general way possible in order to produce reasonable looking figures for most systems. However, because of the huge variety of tight-binding models, the preset style may not always be ideal. This is where this customization guide comes in.

Model structure

A structure plot presents the crystal structure of a model by drawing lattice sites as circles and hoppings as lines which connect the circles. At first glance, this seems like a combination of the standard scatter and line plots found in matplotlib, but the specific requirements of tight-binding complicate the implementation. This is why pybinding has its own specialized structure plotting functions. While these functions are based on matplotlib, they offer additional options which will be explained here.

Structure plot classes

A few different classes in pybinding use structure plots. These are *Lattice*, *Model*, *System*, *Lead* and *StructureMap*. They all represent some kind of spatial structure with sites and hoppings. Note that most of these classes are components of the main *Model*. Calling their plot methods will draw the structure which they represent. The following pseudo-code presents a few possibilities:

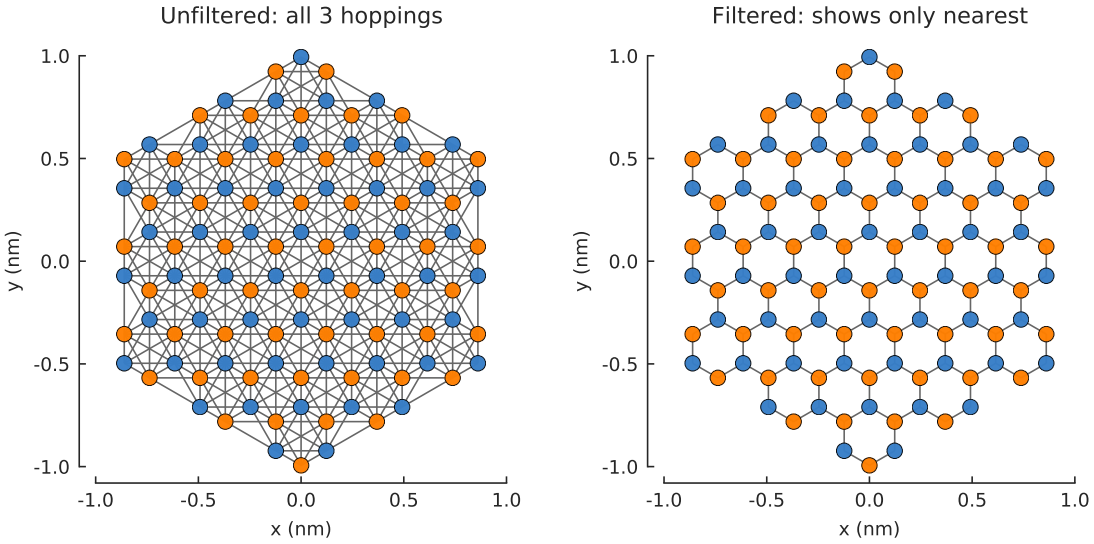
```
model = pb.Model(...)  # specify model
model.attach_lead(...)  # specify leads

model.lattice.plot()  # just the unit cell
model.plot()  # the main system and leads
model.system.plot()  # only the main system
model.leads[0].plot()  # only lead 0
```

In the following sections we'll present a few features of the structure plotting API. The examples will involve mainly `Model.plot()`, but all of these methods share the same common API.

Draw only certain hoppings

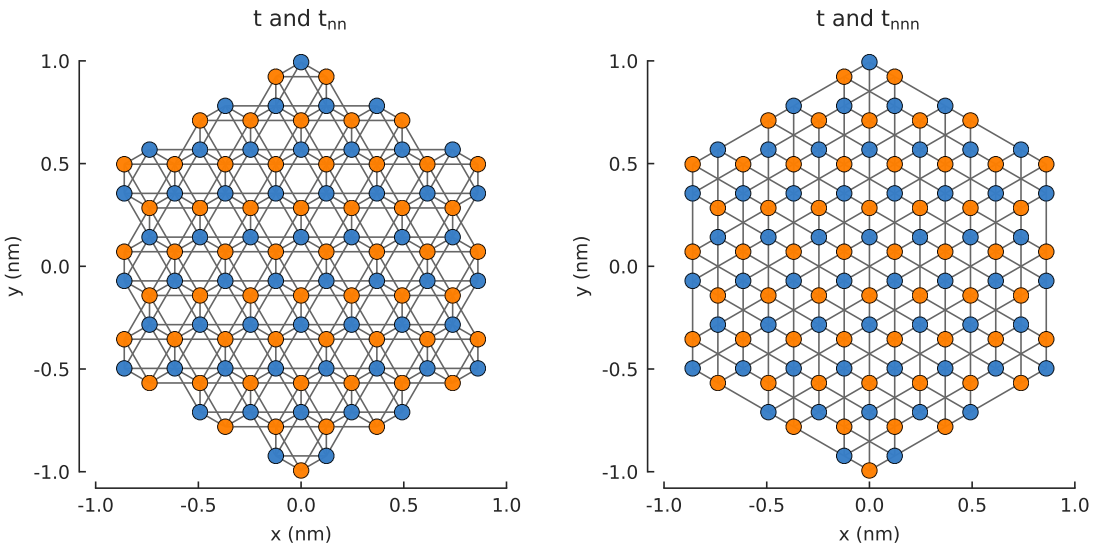
The structure plot usually draws lines for all hoppings. We can see an example here with the third-nearest-neighbor model of graphene. Note the huge number of hoppings in the figure below. The extra information may be useful for calculations, but it is not always desirable for figures because of the extra noise. To filter out some of the lines, we can pass the `draw_only` argument as a list of hopping names. For example, if we only want the first-nearest neighbors:

```
from pybinding.repository import graphene

plt.figure(figsize=(7, 3))
model = pb.Model(graphene.monolayer(nearest_neighbors=3), graphene.hexagon_ac(1))

plt.subplot(121, title="Unfiltered: all 3 hoppings")
model.plot()

plt.subplot(122, title="Filtered: shows only nearest")
model.plot(hopping={'draw_only': ['t']})
```



We can also select hoppings in any combination:

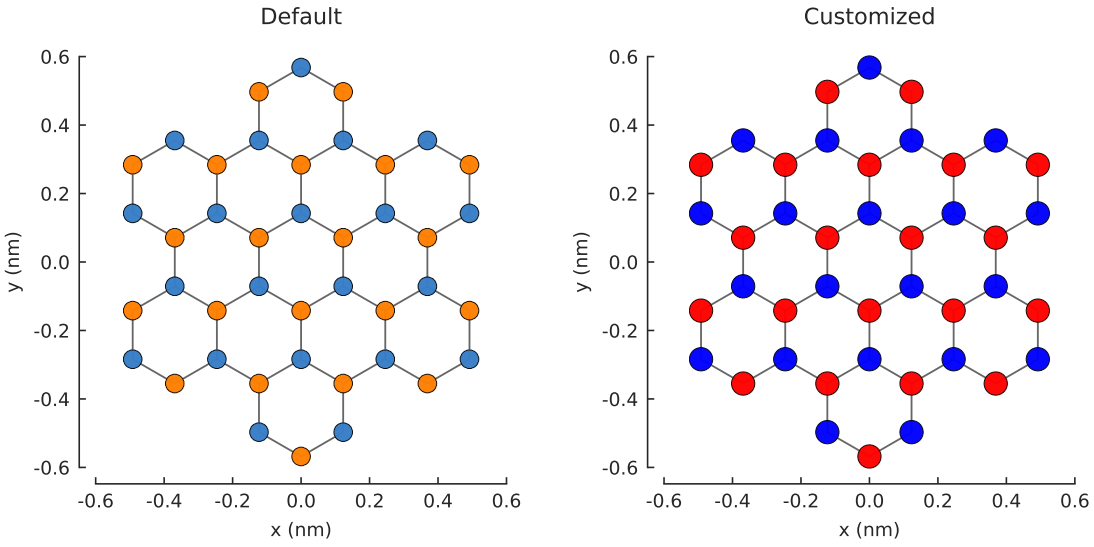
```
plt.figure(figsize=(7, 3))

plt.subplot(121, title="$t$ and $t_{nn}$")
model.plot(hopping={'draw_only': ['t', 't_nn']})

plt.subplot(122, title="$t$ and $t_{nnn}$")
model.plot(hopping={'draw_only': ['t', 't_nnn']})
```



Site radius and color

The site radius is given in data units (nanometers in this example). Colors are passed as a list of colors or a matplotlib colormap.

```
plt.figure(figsize=(7, 3))
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(0.5))

plt.subplot(121, title="Default")
model.plot()

plt.subplot(122, title="Customized")
model.plot(site={'radius': 0.04, 'cmap': ['blue', 'red']})
```

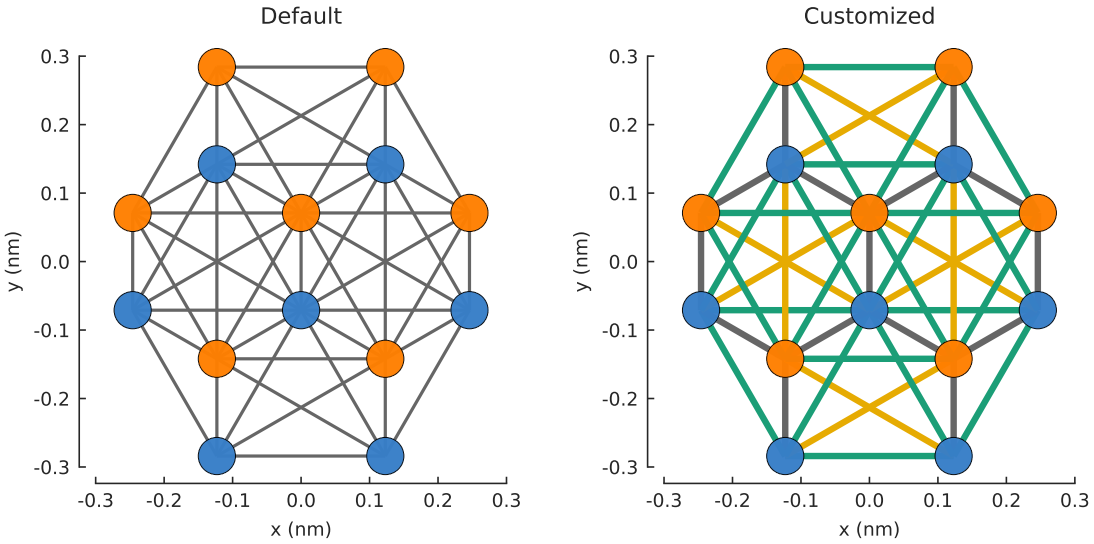


Hopping width and color

By default, all hopping kinds (nearest, next-nearest, etc.) are shown using the same line color, but they can be colorized using the `cmap` parameter.

```
plt.figure(figsize=(7, 3))
model = pb.Model(graphene.monolayer(nearest_neighbors=3), pb.rectangle(0.6))
```

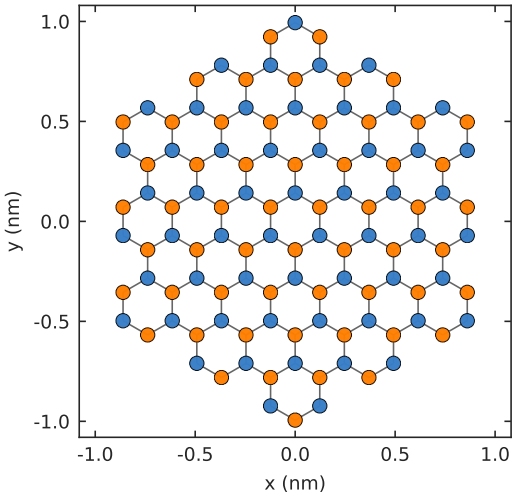
```
plt.subplot(121, title="Default")
model.plot()

plt.subplot(122, title="Customized")
model.plot(hopping={'width': 2, 'cmap': 'auto'})
```



Redraw all axes spines

By default, pybinding plots will remove the right and top axes spines. To recover those lines call the *pltutils.respine()* function.

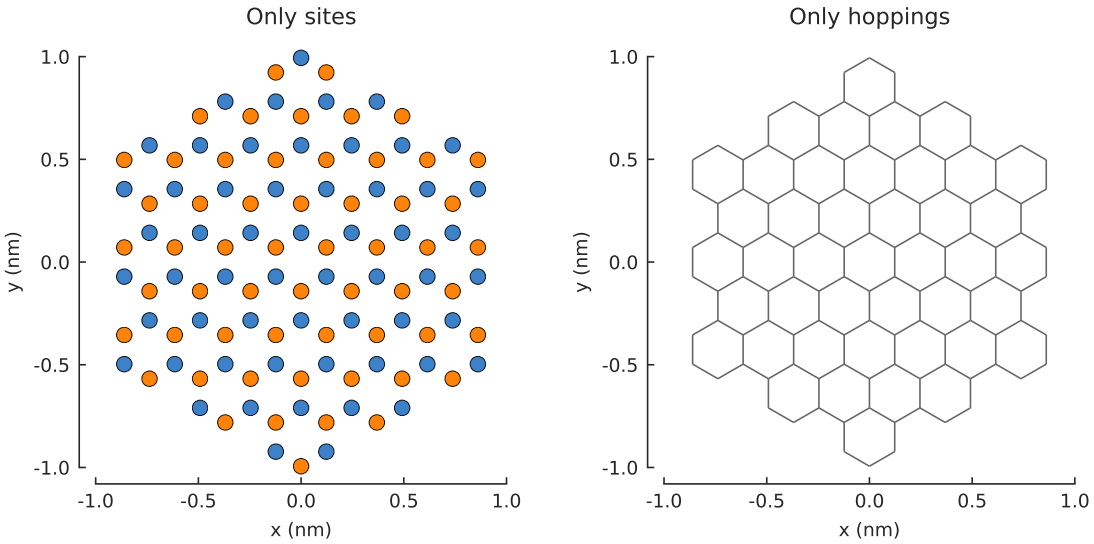
```
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(1))
model.plot()
pb.pltutils.respine()
```



Plot only sites or only hoppings

It can sometimes be useful to separate the plotting of sites and hoppings. Notably, for large systems drawing a huge number of hopping lines can become quite slow and they may even be too small to actually see in the figure. In such cases, removing the hoppings can speed up plotting considerably. Another use case is for the composition of multiple plots – see the next page for an example.

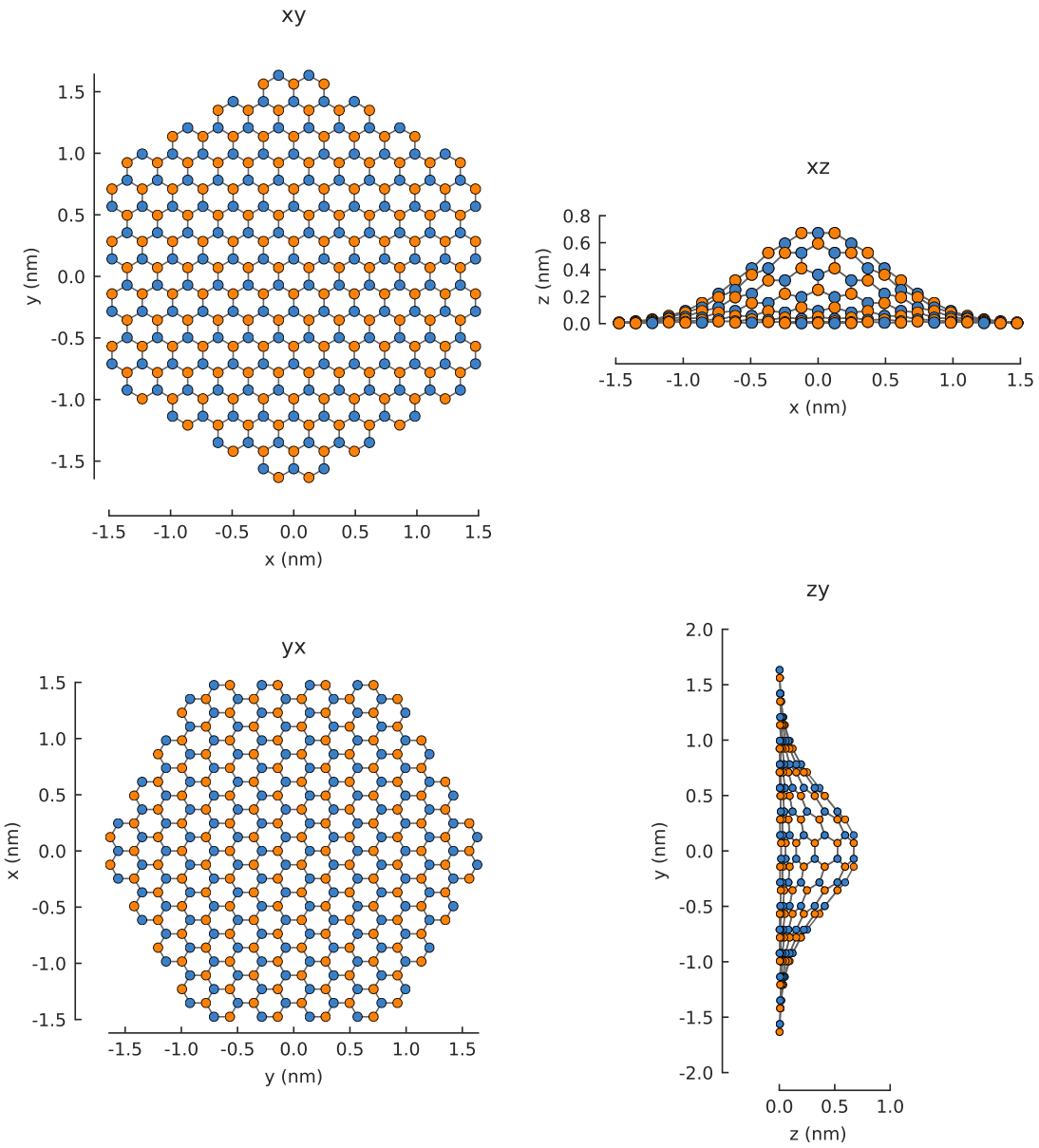
```
plt.figure(figsize=(7, 3))
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(1))

plt.subplot(121, title="Only sites")
model.plot(hopping={"width": 0})

plt.subplot(122, title="Only hoppings")
model.plot(site={"radius": 0})
```



Rotating the view

By default, all structure plots show the xy-plane. The view can be rotated by settings the `axes` argument to a string consisting of any combination of the letters “x”, “y” and “z”.

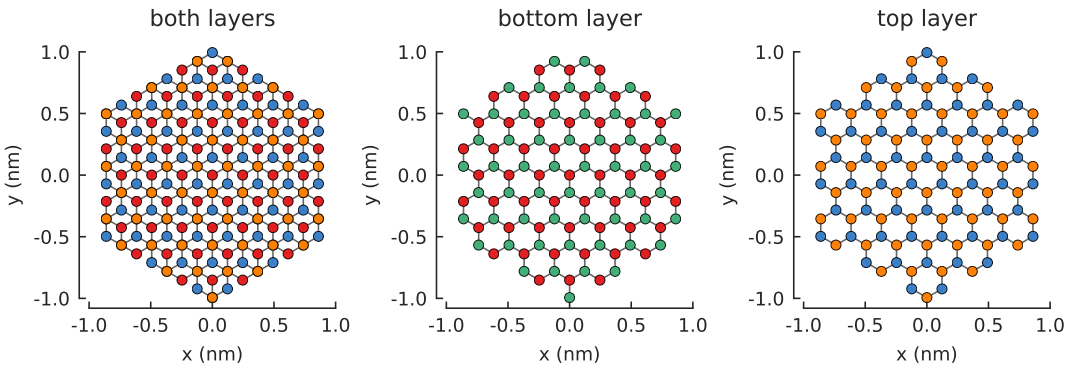
```
model = pb.Model(graphene.monolayer().with_offset([-graphene.a / 2, 0]),
                 pb.regular_polygon(num_sides=6, radius=1.8),
                 graphene.gaussian_bump(height=0.7, sigma=0.7))

plt.figure(figsize=(6.8, 7.5))
plt.subplot(221, title="xy", ylim=[-1.8, 1.8])
model.plot()
plt.subplot(222, title="xz")
model.plot(axes="xz")
plt.subplot(223, title="yx", xlim=[-1.8, 1.8])
model.plot(axes="yx")
plt.subplot(224, title="zy")
model.plot(axes="zy")
```



Slicing layers

For multilayer materials, it is sometimes useful to plot each layer individually.

```
model = pb.Model(graphene.bilayer().with_offset([graphene.a/2, 0]),
                 pb.regular_polygon(num_sides=6, radius=1))

plt.figure(figsize=(6.8, 1.8))
plt.subplot(131, title="both layers")
model.plot()

plt.subplot(132, title="bottom layer")
s = model.system
s[s.z < 0].plot()

plt.subplot(133, title="top layer")
s[s.z >= 0].plot()
```



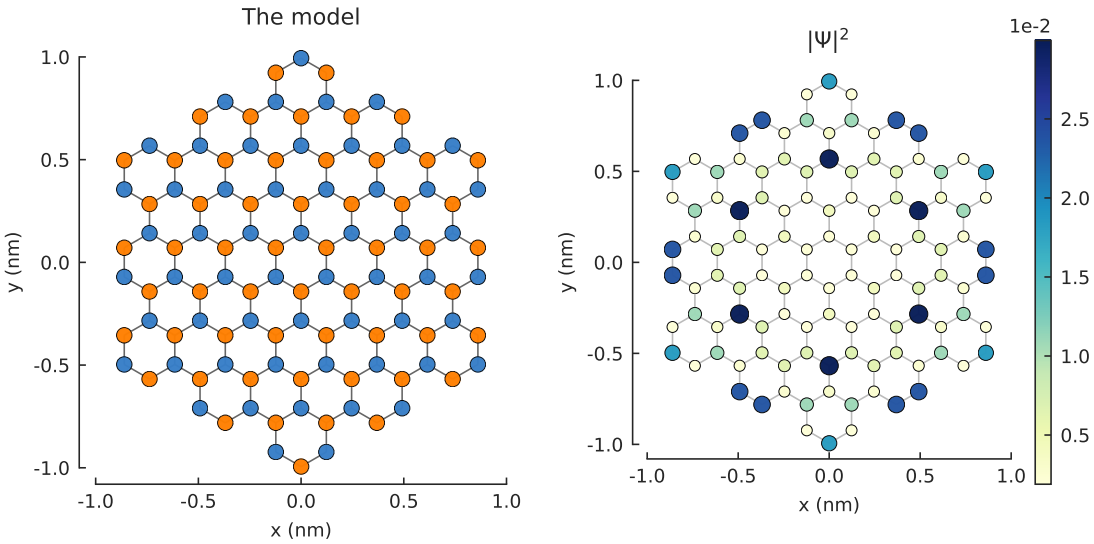
Structure-mapped data

As shown in the previous section, many classes in pybinding use structure plots in a similar way. One class stands out here: *StructureMap* can be used to map any arbitrary data onto the spatial structure of a model. *StructureMap* objects are produced in two cases: as the results of various computation functions (e.g. *Solver.calc_spatial_ldos()*) or returned from *Model.structure_map()* which can map custom user data.

Draw only certain hoppings

Just as before, we can draw only the desired hoppings. Note that `smap` is a *StructureMap* returned by *Solver.calc_probability()*.

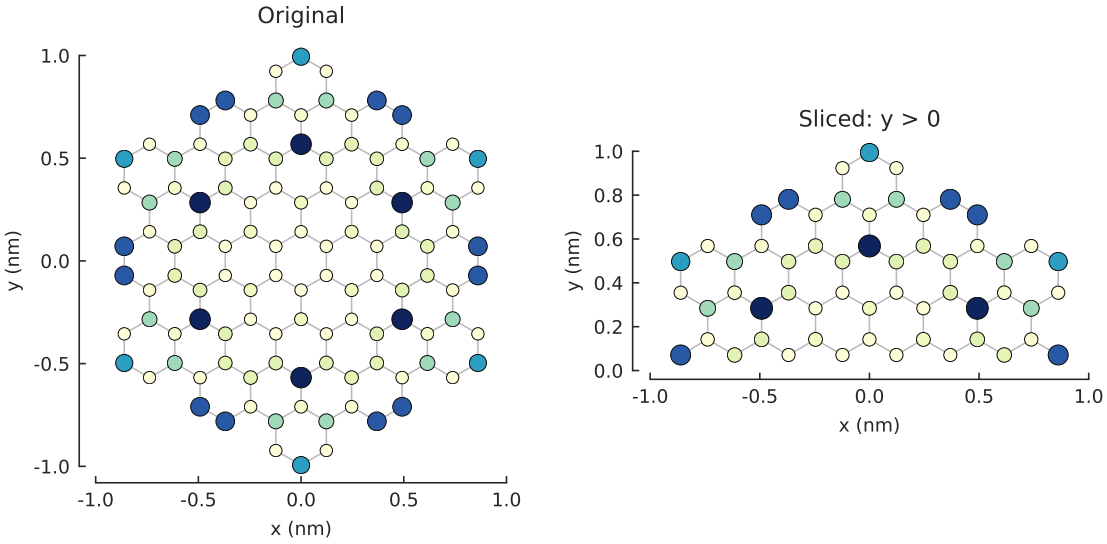
```
from pybinding.repository import graphene

plt.figure(figsize=(7, 3))

plt.subplot(121, title="The model")
model = pb.Model(graphene.monolayer(nearest_neighbors=3), graphene.hexagon_ac(1))
model.plot(hopping={'draw_only': ['t']})

plt.subplot(122, title="$|\Psi|^2$")
solver = pb.solver.arpack(model, k=10)
smap = solver.calc_probability(n=2)
smap.plot(hopping={'draw_only': ['t']})
pb.pltutils.colorbar()
```



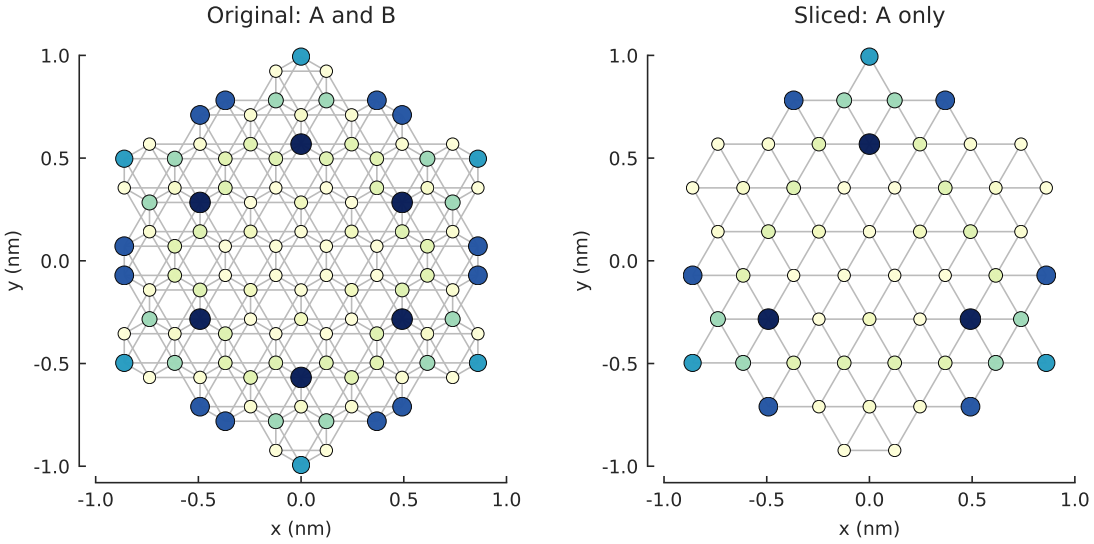
Slicing a structure

This follows a syntax similar to numpy fancy indexing where we can give a condition as the index.

```
plt.figure(figsize=(7, 3))

plt.subplot(121, title="Original")
smap.plot(hopping={'draw_only': ['t']})

plt.subplot(122, title="Sliced: y > 0")
upper = smap[smap.y > 0]
upper.plot(hopping={'draw_only': ['t']})
```



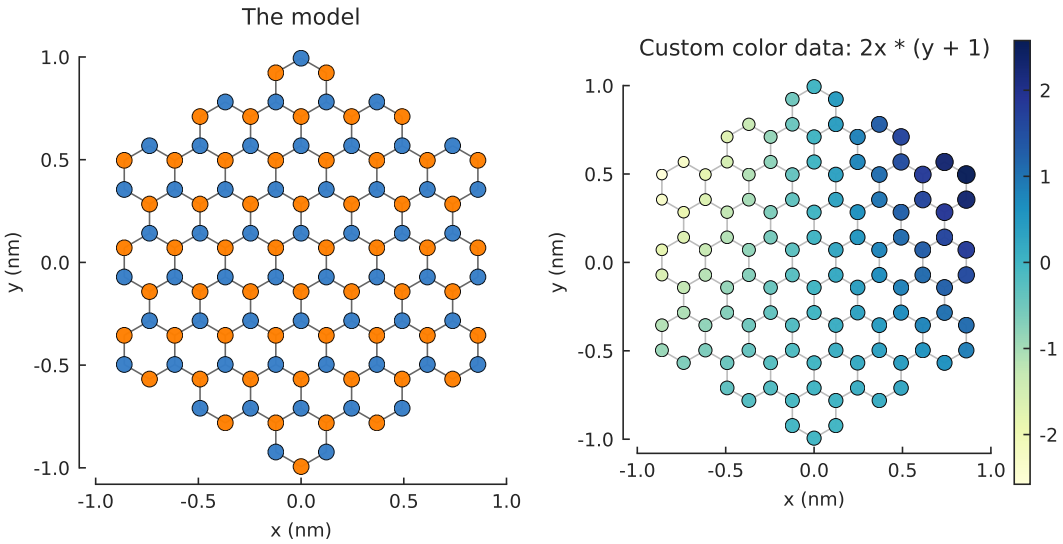

```
plt.figure(figsize=(7, 3))

plt.subplot(121, title="Original: A and B")
smap.plot(hopping={'draw_only': ['t', 't_nn']})

plt.subplot(122, title="Sliced: A only")
a_only = smap[smap.sublattices == 'A']
a_only.plot(hopping={'draw_only': ['t', 't_nn']})
```



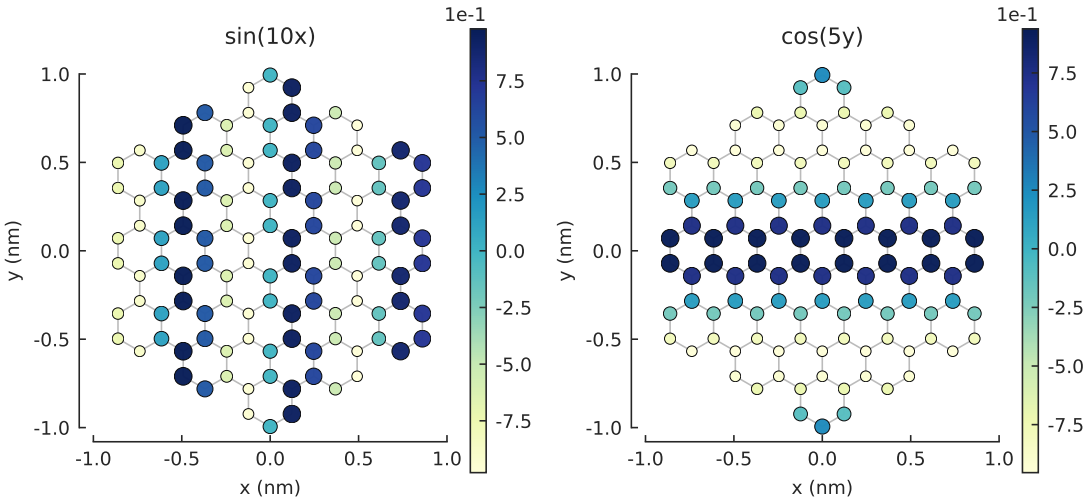
Mapping custom data

The method *Model.structure_map()* returns a *StructureMap* where any user-defined `data` can be mapped to the spatial positions of the lattice sites. The `data` just needs to be a 1D array with the same size as the total number of sites in the system.

```
plt.figure(figsize=(6.8, 3))

plt.subplot(121, title="The model")
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(1))
model.plot()

plt.subplot(122, title="Custom color data: 2x * (y + 1)")
custom_data = 2 * model.system.x * (model.system.y + 1)
smap = model.structure_map(custom_data)
smap.plot()
pb.pltutils.colorbar()
```



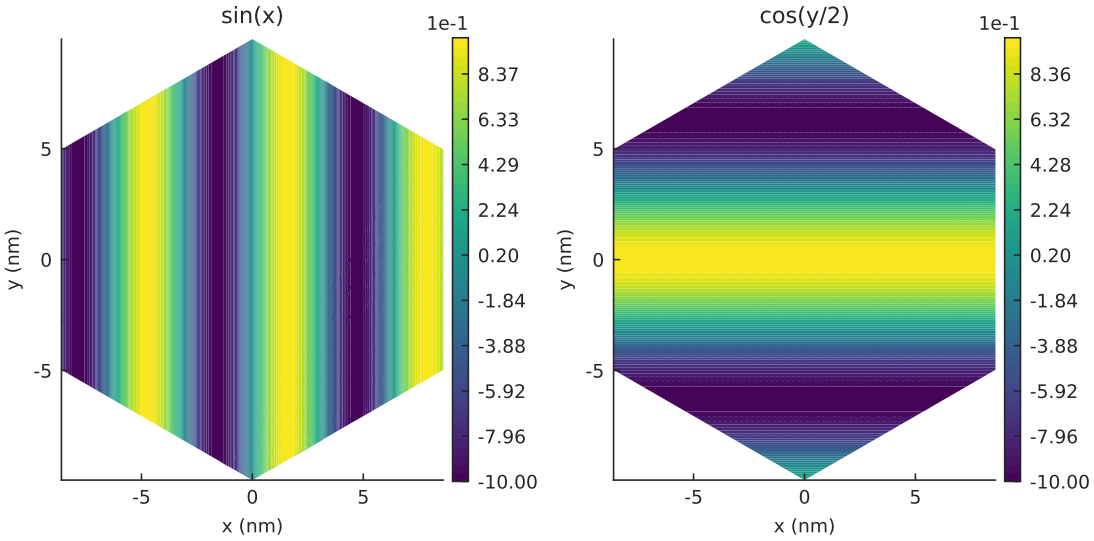

```
plt.figure(figsize=(6.8, 3))

plt.subplot(121, title="sin(10x)")
smap = model.structure_map(np.sin(10 * model.system.x))
smap.plot()
pb.pltutils.colorbar()

plt.subplot(122, title="cos(5y)")
smap = model.structure_map(np.cos(5 * model.system.y))
smap.plot()
pb.pltutils.colorbar()
```



Contour plots for large systems

For larger systems, structure plots don't make much sense because the details of the sites and hoppings would be too small to see. Contour plots look much better in this case.

```
plt.figure(figsize=(6.8, 3))
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(10))

plt.subplot(121, title="sin(x)")
smap = model.structure_map(np.sin(model.system.x))
smap.plot_contourf()
pb.pltutils.colorbar()

plt.subplot(122, title="cos(y/2)")
smap = model.structure_map(np.cos(0.5 * model.system.y))
smap.plot_contourf()
pb.pltutils.colorbar()
```

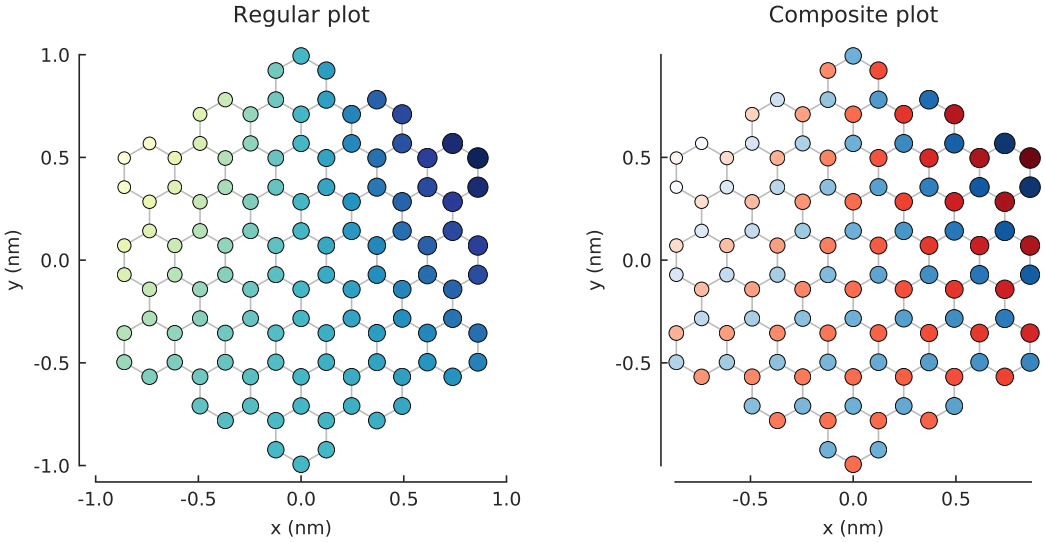


Composing multiple plots

Various plotting methods or even different invocations of the same method can be composed to create nice figures. For example, we may want to use different colormaps to distinguish between sublattices A and B when plotting

some data on top of the structure of graphene. Below, the first pass plots only the hopping lines, the second pass draws the sites of sublattice A and the third draws sublattice B. The darkness of the color indicates the intensity of the mapped data, while blue/red distinguishes the sublattices.

```
model = pb.Model(graphene.monolayer(), graphene.hexagon_ac(1))
custom_data = 2 * model.system.x * (model.system.y + 1)
smap = model.structure_map(custom_data)

plt.figure(figsize=(6.8, 3))
plt.subplot(121, title="Regular plot")
smap.plot()

plt.subplot(122, title="Composite plot")
smap.plot(site_radius=0)  # only draw hopping lines, no sites
a_only = smap[smap.sublattices == "A"]
a_only.plot(cmap="Blues", hopping={'width': 0})  # A sites, no hoppings
b_only = smap[smap.sublattices == "B"]
b_only.plot(cmap="Reds", hopping={'width': 0})  # B sites, no hoppings
```



CHAPTER 11

Random Examples

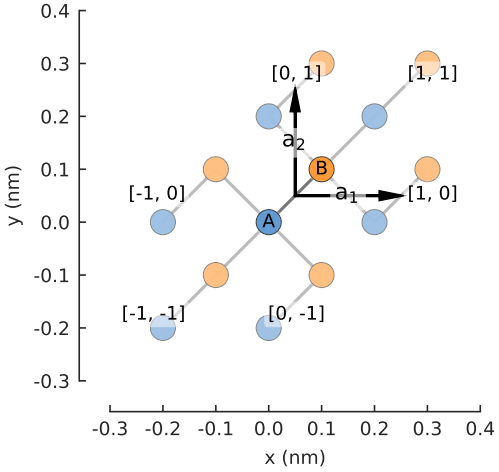
If you're just browsing, the *Tutorial* section is the best place to start. It gives a good overview of the most important features with lots of code examples. This section contains a few more examples which did not fit into *Tutorial* or *Additional Topics*.

Lattice specification and bands

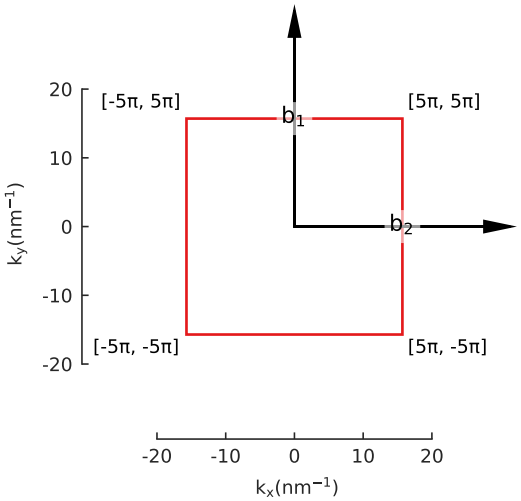
Checkerboard

Source code

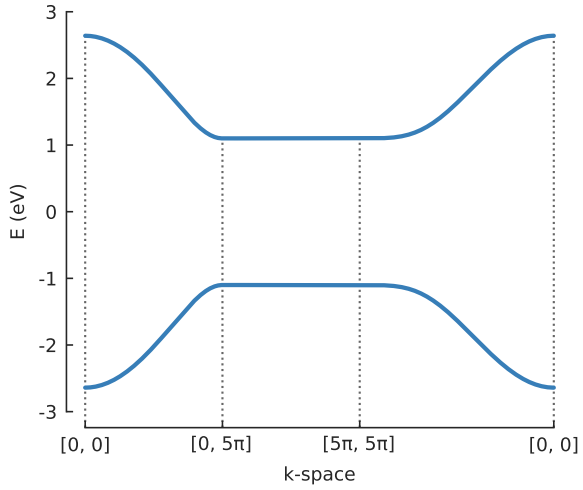
```
"""Two dimensional checkerboard lattice with real hoppings"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import pi

pb.pltutils.use_style()


def checkerboard(d=0.2, delta=1.1, t=0.6):
    lat = pb.Lattice(a1=[d, 0], a2=[0, d])
    lat.add_sublattices(
        ('A', [0, 0], -delta),
        ('B', [d/2, d/2], delta)
    )
    lat.add_hoppings(
        ([ 0,  0], 'A', 'B', t),
        ([ 0, -1], 'A', 'B', t),
        ([-1,  0], 'A', 'B', t),
        ([-1, -1], 'A', 'B', t)
    )
    return lat

lattice = checkerboard()
lattice.plot()
plt.show()
```




```
lattice.plot_brillouin_zone()
plt.show()
```



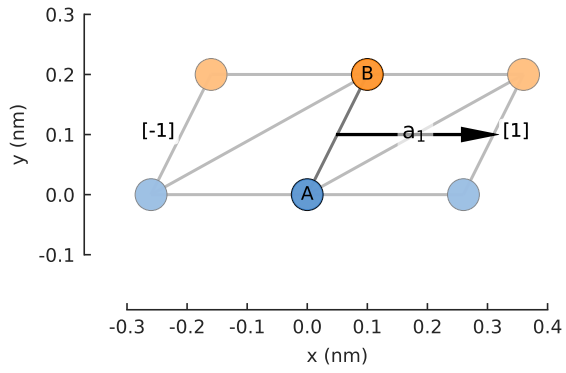

```
model = pb.Model(checkerboard(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

bands = solver.calc_bands([0, 0], [0, 5*pi], [5*pi, 5*pi], [0, 0])
bands.plot()
plt.show()
```



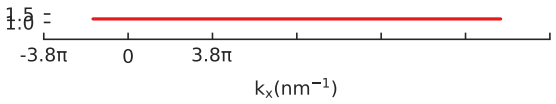
Trestle

Source code

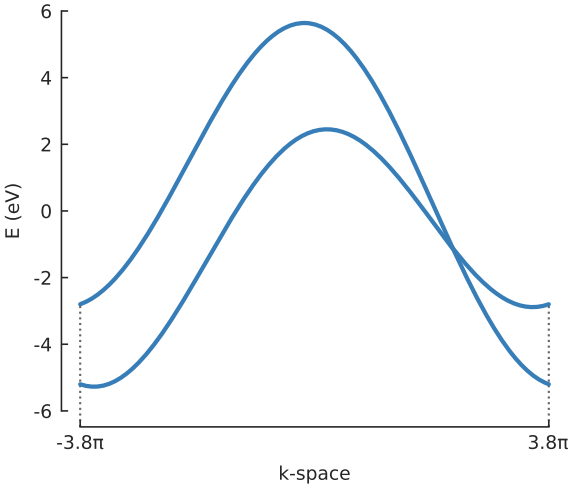
```
"""One dimensional lattice with complex hoppings"""
import pybinding as pb
import matplotlib.pyplot as plt

pb.pltutils.use_style()


def trestle(d=0.2, t1=0.8 + 0.6j, t2=2):
    lat = pb.Lattice(a1=1.3*d)
    lat.add_sublattices(
        ('A', [0,   0], 0),
        ('B', [d/2, d], 0)
    )
    lat.add_hoppings(
        (0, 'A', 'B', t1),
        (1, 'A', 'B', t1),
        (1, 'A', 'A', t2),
        (1, 'B', 'B', t2)
    )
    return lat

lattice = trestle()
lattice.plot()
plt.show()
```



```
lattice.plot_brillouin_zone()
plt.show()
```




```
model = pb.Model(trestle(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

start, end = lattice.brillouin_zone()
bands = solver.calc_bands(start, end)
bands.plot()
plt.show()
```

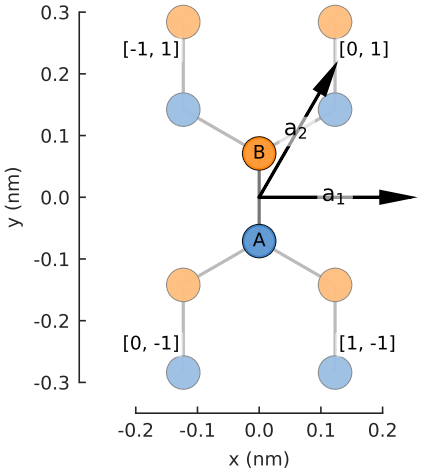


Monolayer graphene

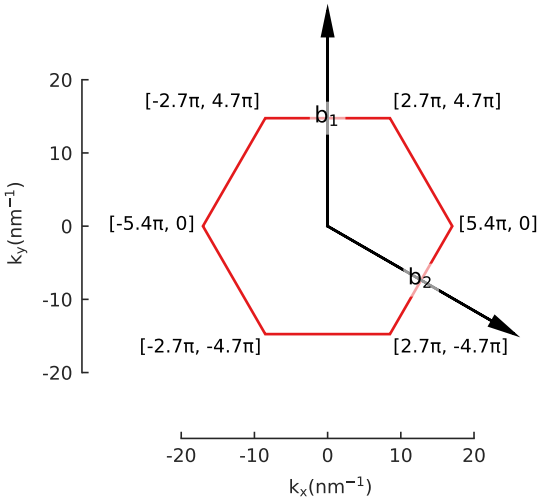
Source code

```
"""Create and plot a monolayer graphene lattice, its Brillouin zone and band␣
↪structure"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import sqrt, pi

pb.pltutils.use_style()


def monolayer_graphene():
    """Return the lattice specification for monolayer graphene"""
    a = 0.24595   # [nm] unit cell length
    a_cc = 0.142  # [nm] carbon-carbon distance
    t = -2.8      # [eV] nearest neighbour hopping

    # create a lattice with 2 primitive vectors
    lat = pb.Lattice(
        a1=[a, 0],
        a2=[a/2, a/2 * sqrt(3)]
    )

    lat.add_sublattices(
        # name and position
```

```
        ('A', [0, -a_cc/2]),
        ('B', [0,  a_cc/2])
    )

    lat.add_hoppings(
        # inside the main cell
        ([0,  0], 'A', 'B', t),
        # between neighboring cells
        ([1, -1], 'A', 'B', t),
        ([0, -1], 'A', 'B', t)
    )

    return lat


lattice = monolayer_graphene()
lattice.plot()
plt.show()
```

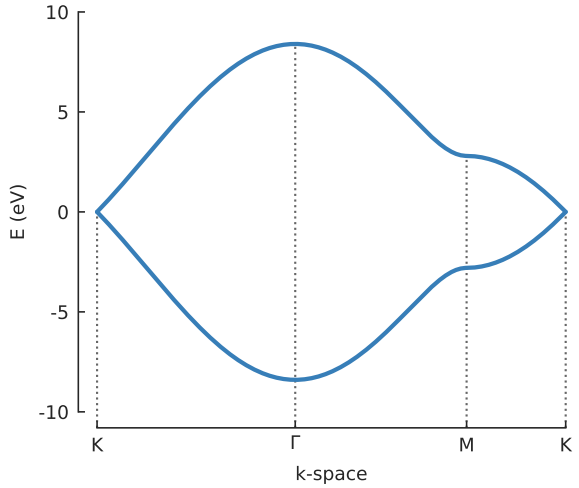



```
lattice.plot_brillouin_zone()
plt.show()
```

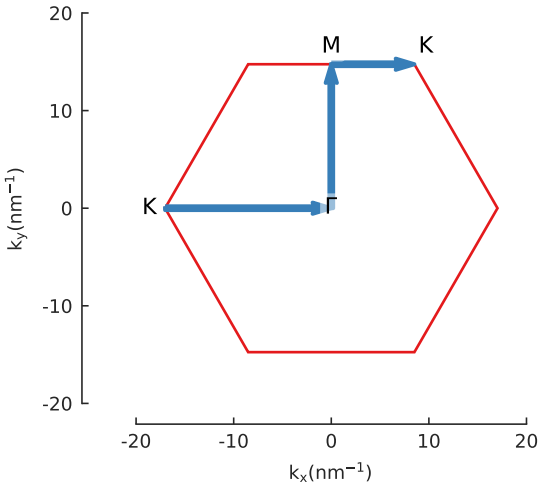



```
model = pb.Model(monolayer_graphene(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

a_cc = 0.142
```

```
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
plt.show()
```




```
model.lattice.plot_brillouin_zone(decorate=False)
bands.plot_kpath(point_labels=['K', r'$\Gamma$', 'M', 'K'])
```

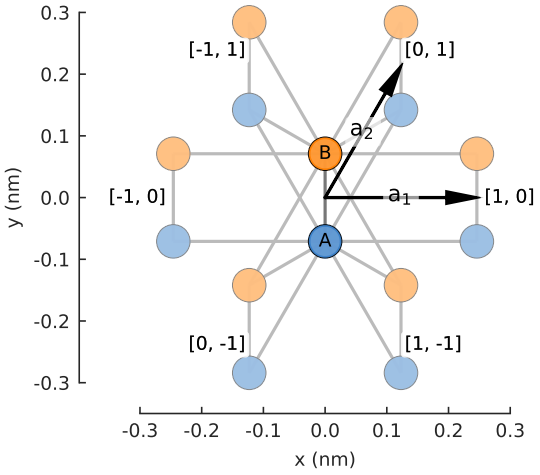


Monolayer graphene NN

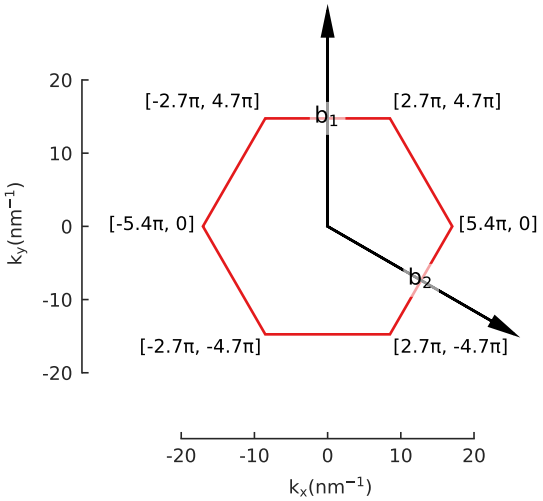
Source code

```
"""Monolayer graphene with next-nearest hoppings"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import sqrt, pi

pb.pltutils.use_style()


def monolayer_graphene_nn():
```

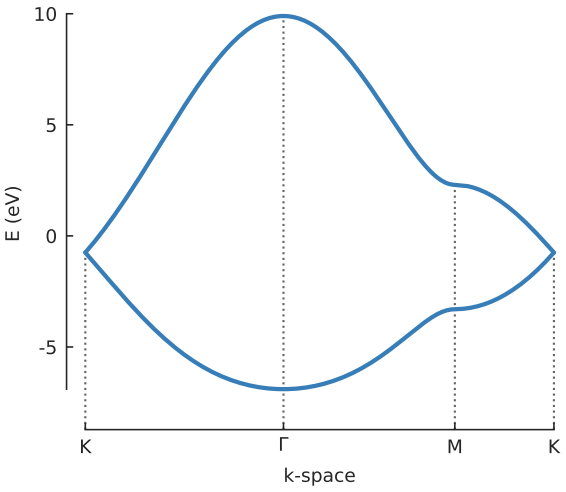
```
    a = 0.24595    # [nm] unit cell length
    a_cc = 0.142   # [nm] carbon-carbon distance
    t = -2.8       # [eV] nearest neighbour hopping
    t_nn = 0.25    # [eV] next-nearest neighbour hopping

    lat = pb.Lattice(
        a1=[a, 0],
        a2=[a/2, a/2 * sqrt(3)]
    )
    lat.add_sublattices(
        ('A', [0, -a_cc/2]),
        ('B', [0,  a_cc/2])
    )
    lat.add_hoppings(
        # between A and B inside the main cell
        ([0,  0], 'A', 'B', t),
        # between neighboring cells
        ([1, -1], 'A', 'B', t),
        ([0, -1], 'A', 'B', t),
        # next-nearest
        ([1,  0], 'A', 'A', t_nn),
        ([1,  0], 'B', 'B', t_nn),
        ([0,  1], 'A', 'A', t_nn),
        ([0,  1], 'B', 'B', t_nn),
        ([1, -1], 'A', 'A', t_nn),
        ([1, -1], 'B', 'B', t_nn)
    )
    return lat

lattice = monolayer_graphene_nn()
lattice.plot()
plt.show()
```




```
lattice.plot_brillouin_zone()
plt.show()
```




```
model = pb.Model(monolayer_graphene_nn(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

a_cc = 0.142
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

# Note the elector-hole asymmetry in the band structure (due to t_nn).
bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
plt.show()
```

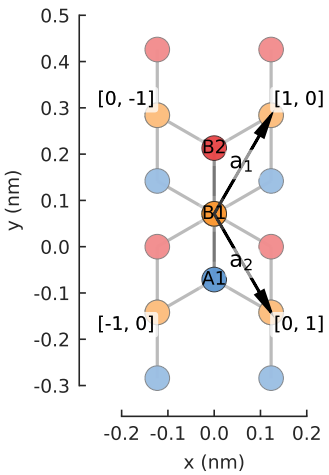


Bilayer graphene

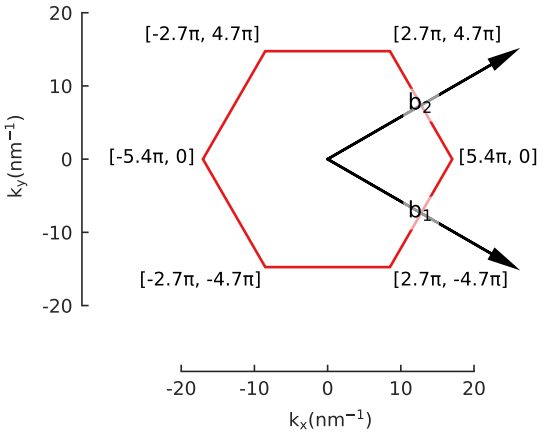
Source code

```
"""Build the simplest model of bilayer graphene and compute its band structure"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import sqrt, pi

pb.pltutils.use_style()
```

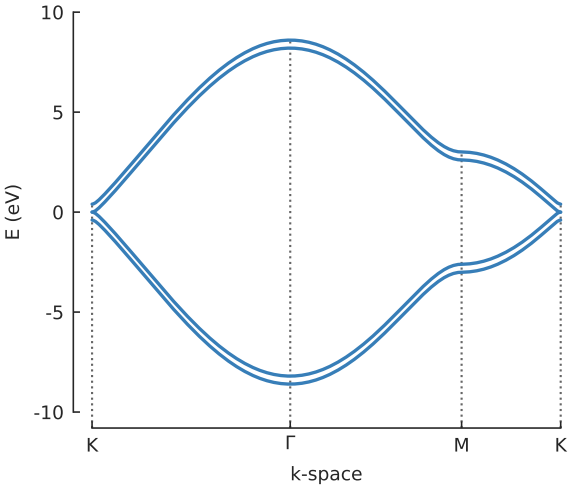
```
def bilayer_graphene():
    """Bilayer lattice in the AB-stacked form (Bernal-stacked)

    This is the simplest model with just a single intralayer and a single␣
↪interlayer hopping.
    """
    a = 0.24595    # [nm] unit cell length
    a_cc = 0.142   # [nm] carbon-carbon distance
    c0 = 0.335     # [nm] interlayer spacing

    lat = pb.Lattice(a1=[a/2, a/2 * sqrt(3)], a2=[a/2, -a/2 * sqrt(3)])

    lat.add_sublattices(
        ('A1', [0,  -a_cc/2,    0]),
        ('B1', [0,   a_cc/2,    0]),
        ('A2', [0,   a_cc/2, -c0]),
        ('B2', [0, 3*a_cc/2, -c0])
    )

    lat.register_hopping_energies({
        'gamma0': -2.8,   # [eV] intralayer
        'gamma1': -0.4,   # [eV] interlayer
    })

    lat.add_hoppings(
        # layer 1
        ([ 0,  0], 'A1', 'B1', 'gamma0'),
        ([ 0,  1], 'A1', 'B1', 'gamma0'),
        ([-1,  0], 'A1', 'B1', 'gamma0'),
        # layer 2
        ([ 0,  0], 'A2', 'B2', 'gamma0'),
        ([ 0,  1], 'A2', 'B2', 'gamma0'),
        ([-1,  0], 'A2', 'B2', 'gamma0'),
        # interlayer
        ([ 0,  0], 'B1', 'A2', 'gamma1')
    )

    return lat


lattice = bilayer_graphene()
lattice.plot()
plt.show()
```



```
lattice.plot_brillouin_zone()
plt.show()
```




```
model = pb.Model(bilayer_graphene(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

a_cc = 0.142
Gamma = [0, 0]
K1 = [-4*pi / (3*sqrt(3)*a_cc), 0]
M = [0, 2*pi / (3*a_cc)]
K2 = [2*pi / (3*sqrt(3)*a_cc), 2*pi / (3*a_cc)]

bands = solver.calc_bands(K1, Gamma, M, K2)
bands.plot(point_labels=['K', r'$\Gamma$', 'M', 'K'])
plt.show()
```



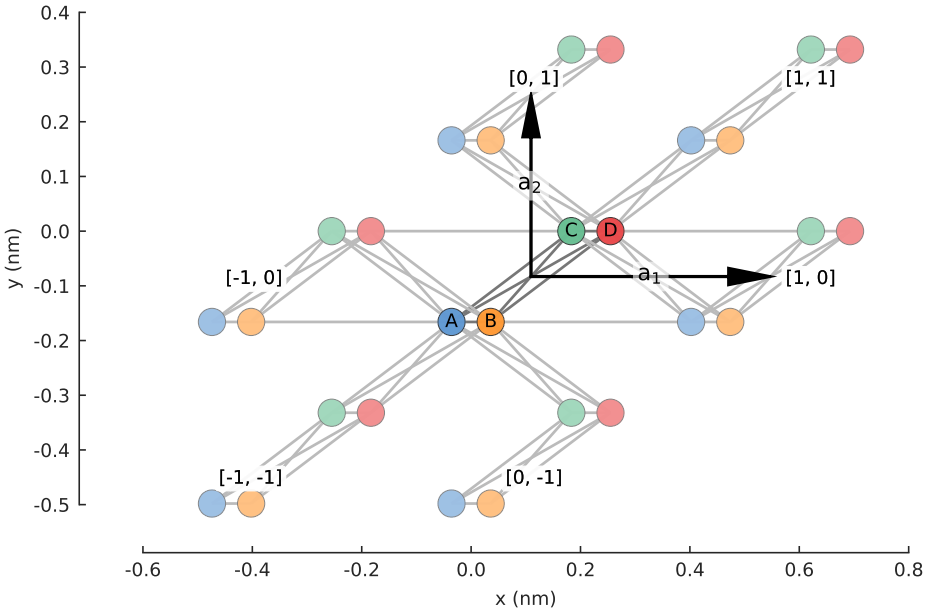
Phosphorene

Source code

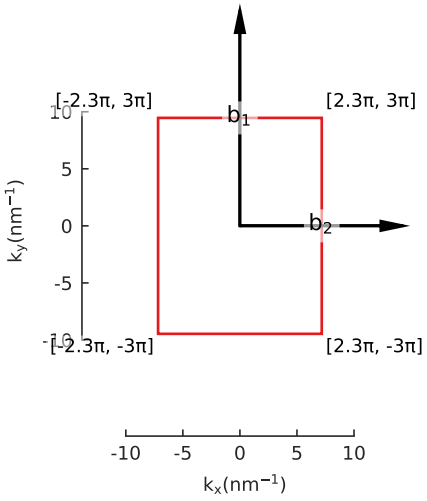
```
"""Create and plot a phosphorene lattice, its Brillouin zone and band structure"""
import pybinding as pb
import matplotlib.pyplot as plt
from math import pi, sin, cos
```

```
pb.pltutils.use_style()


def phosphorene_4band():
    """Monolayer phosphorene lattice using the four-band model"""
    a = 0.222
    ax = 0.438
    ay = 0.332
    theta = 96.79 * (pi / 180)
    phi = 103.69 * (pi / 180)

    lat = pb.Lattice(a1=[ax, 0], a2=[0, ay])

    h = a * sin(phi - pi / 2)
    s = 0.5 * ax - a * cos(theta / 2)
    lat.add_sublattices(
        ('A', [-s/2,         -ay/2, h], 0),
        ('B', [ s/2,         -ay/2, 0], 0),
        ('C', [-s/2 + ax/2,      0, 0], 0),
        ('D', [ s/2 + ax/2,      0, h], 0)
    )

    lat.register_hopping_energies({
        't1': -1.22,
        't2': 3.665,
        't3': -0.205,
        't4': -0.105,
        't5': -0.055
    })

    lat.add_hoppings(
        # t1
        ([-1,  0], 'A', 'D', 't1'),
        ([-1, -1], 'A', 'D', 't1'),
        ([ 0,  0], 'B', 'C', 't1'),
        ([ 0, -1], 'B', 'C', 't1'),
        # t2
        ([ 0,  0], 'A', 'B', 't2'),
        ([ 0,  0], 'C', 'D', 't2'),
        # t3
        ([ 0,  0], 'A', 'D', 't3'),
        ([ 0, -1], 'A', 'D', 't3'),
        ([ 1,  1], 'C', 'B', 't3'),
        ([ 1,  0], 'C', 'B', 't3'),
        # t4
        ([ 0,  0], 'A', 'C', 't4'),
        ([ 0, -1], 'A', 'C', 't4'),
        ([-1,  0], 'A', 'C', 't4'),
        ([-1, -1], 'A', 'C', 't4'),
        ([ 0,  0], 'B', 'D', 't4'),
        ([ 0, -1], 'B', 'D', 't4'),
        ([-1,  0], 'B', 'D', 't4'),
        ([-1, -1], 'B', 'D', 't4'),
        # t5
        ([-1,  0], 'A', 'B', 't5'),
        ([-1,  0], 'C', 'D', 't5')
    )

    return lat

plt.figure(figsize=(6, 6))
lattice = phosphorene_4band()
lattice.plot()
```

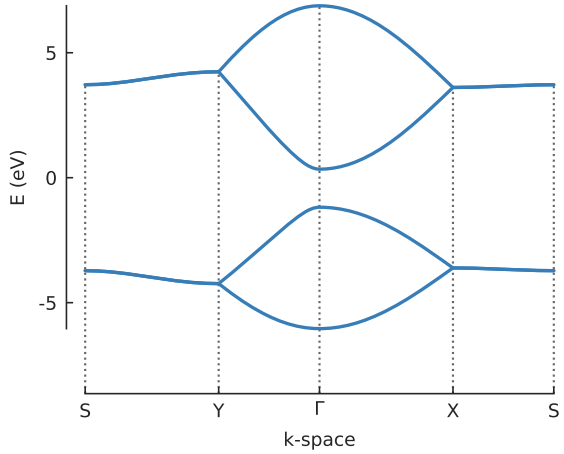
```
plt.show()
```



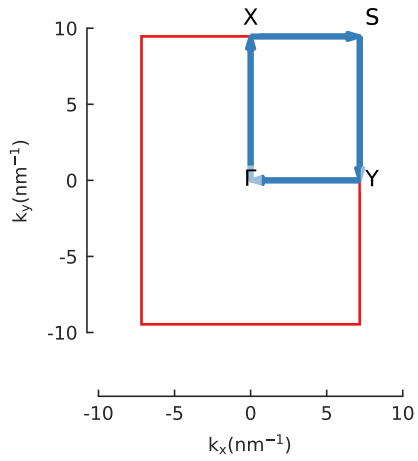

```
lattice.plot_brillouin_zone()
plt.show()
```




```
model = pb.Model(phosphorene_4band(), pb.translational_symmetry())
solver = pb.solver.lapack(model)

ax = 0.438
ay = 0.332
kx = pi / ax
ky = pi / ay
bands = solver.calc_bands([kx, ky], [kx, 0], [0, 0], [0, ky], [kx, ky])
bands.plot(point_labels=["S", "Y", r"$\Gamma$", "X", "S"])
plt.show()
```




```
model.lattice.plot_brillouin_zone(decorate=False)
bands.plot_kpath(point_labels=["S", "Y", r"$\Gamma$", "X", "S"])
plt.show()
```



Finite size

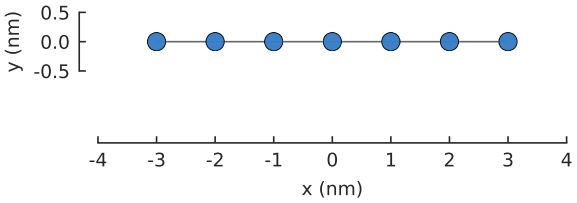
See the *tutorial page* for a detailed walkthrough of system construction. These are just a few quick examples.

1D lattices and line shape

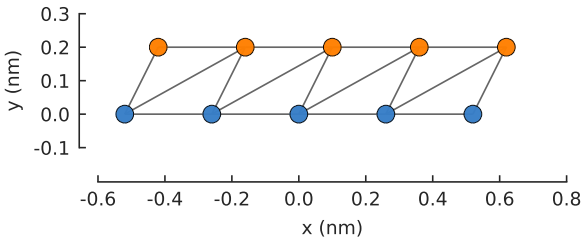
Source code

```
"""1D lattice chains - finite dimension are imposed using builtin `pb.line` shape"
↪"
import pybinding as pb
import matplotlib.pyplot as plt

pb.pltutils.use_style()


def simple_chain_lattice(a=1, t=-1):
    """Very simple 1D lattice"""
    lat = pb.Lattice(a)
```

```
    lat.add_one_sublattice('A', [0, 0])
    lat.add_one_hopping(1, 'A', 'A', t)
    return lat

model = pb.Model(
    simple_chain_lattice(),
    pb.line(-3.5, 3.5)  # line start/end in nanometers
)
model.plot()
plt.show()
```




```
def trestle(a=0.2, t1=0.8 + 0.6j, t2=2):
    """A more complicated 1D lattice with 2 sublattices"""
    lat = pb.Lattice(1.3 * a)
    lat.add_sublattices(
        ('A', [0,   0], 0),
        ('B', [a/2, a], 0)
    )
    lat.add_hoppings(
        (0, 'A', 'B', t1),
        (1, 'A', 'B', t1),
        (1, 'A', 'A', t2),
        (1, 'B', 'B', t2)
    )
    lat.min_neighbors = 2
    return lat

model = pb.Model(trestle(), pb.line(-0.7, 0.7))
model.plot()
plt.show()
```

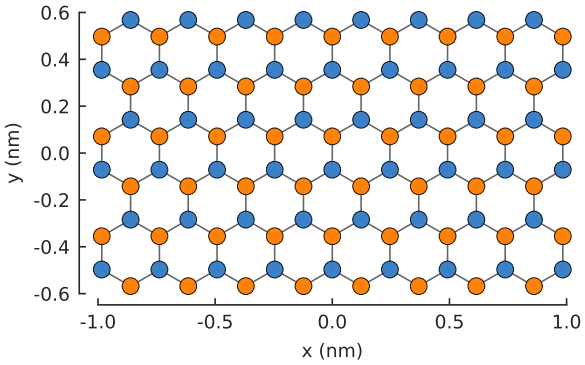


2D lattices and builtin shapes

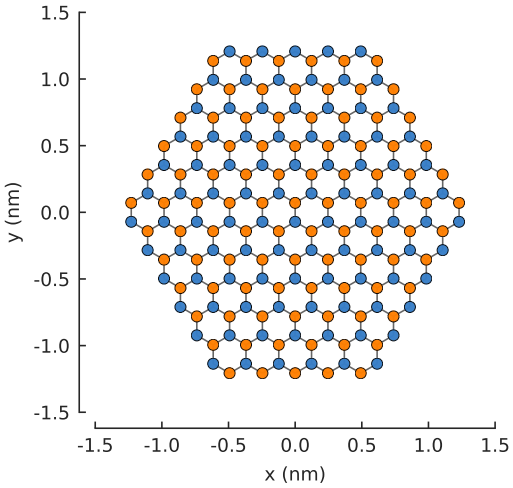
Source code

```
"""Several finite-sized systems created using builtin lattices and shapes"""
import pybinding as pb
from pybinding.repository import graphene
import matplotlib.pyplot as plt
from math import pi
```

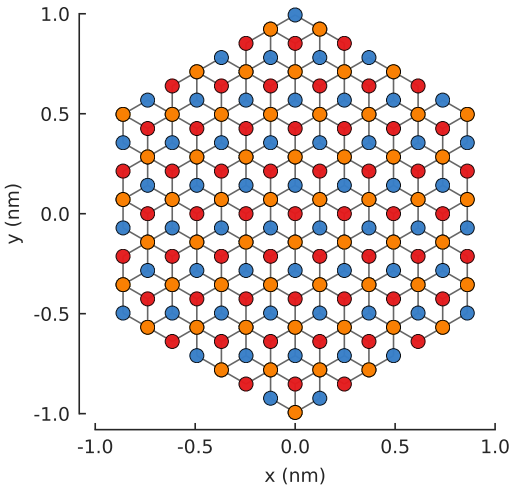
```
pb.pltutils.use_style()

model = pb.Model(
    graphene.monolayer(),
    pb.rectangle(x=2, y=1.2)
)
model.plot()
plt.show()
```




```
model = pb.Model(
    graphene.monolayer(),
    pb.regular_polygon(num_sides=6, radius=1.4, angle=pi/6)
)
model.plot()
plt.show()
```




```
# A graphene-specific shape which guaranties armchair edges on all sides
model = pb.Model(
    graphene.bilayer(),
    graphene.hexagon_ac(side_width=1)
)
model.plot()
plt.show()
```



Nanoribbons

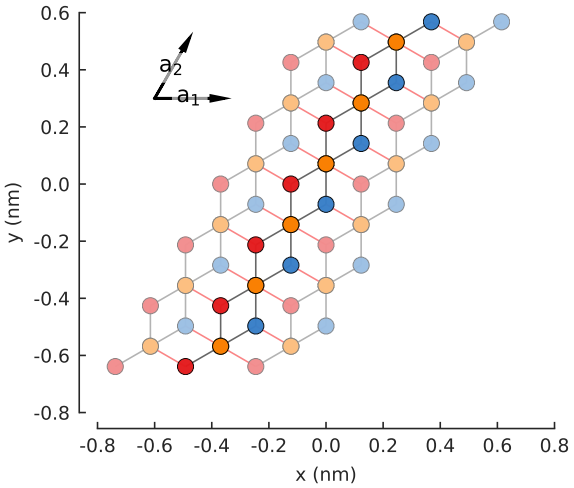
See the *Shape and symmetry* tutorial page for more details on nanoribbon construction. These are just a few quick examples.

Bilayer graphene

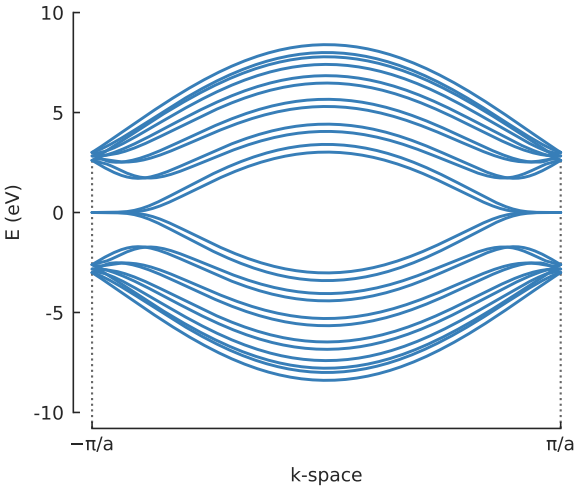
Source code

```
"""Bilayer graphene nanoribbon with zigzag edges"""
import pybinding as pb
import matplotlib.pyplot as plt
from pybinding.repository import graphene
from math import pi, sqrt

pb.pltutils.use_style()


def bilayer_graphene():
    """Bilayer lattice in the AB-stacked form (Bernal-stacked)"""
    lat = pb.Lattice(a1=[graphene.a, 0], a2=[0.5*graphene.a, 0.5*sqrt(3)*graphene.
↪a])

    c0 = 0.335  # [nm] interlayer spacing
    lat.add_sublattices(('A1', [0,  -graphene.a_cc/2,   0]),
                        ('B1', [0,   graphene.a_cc/2,   0]),
                        ('A2', [0,   graphene.a_cc/2, -c0]),
                        ('B2', [0, 3*graphene.a_cc/2, -c0]))
    lat.register_hopping_energies({'t': graphene.t, 't_layer': -0.4})
    lat.add_hoppings(
        # layer 1
        ([ 0,  0], 'A1', 'B1', 't'),
        ([ 1, -1], 'A1', 'B1', 't'),
        ([ 0, -1], 'A1', 'B1', 't'),
        # layer 2
        ([ 0,  0], 'A2', 'B2', 't'),
        ([ 1, -1], 'A2', 'B2', 't'),
        ([ 0, -1], 'A2', 'B2', 't'),
        # interlayer
        ([ 0,  0], 'B1', 'A2', 't_layer')
    )
```

```
    lat.min_neighbors = 2
    return lat

model = pb.Model(
    bilayer_graphene(),
    pb.rectangle(1.3),  # nm
    pb.translational_symmetry(a1=True, a2=False)
)
model.plot()
model.lattice.plot_vectors(position=[-0.6, 0.3])  # nm
plt.show()
```




```
solver = pb.solver.lapack(model)
bands = solver.calc_bands(-pi/graphene.a, pi/graphene.a)
bands.plot(point_labels=[r"$-\pi / a$", r"$\pi / a$"])
plt.show()
```



CHAPTER 12

Material Repository

The repository includes a few common lattices, shapes, fields and other kinds of helpful functions and constants. A material can be imported from `pybinding.repository`, for example:

```
from pybinding.repository import graphene

lattice = graphene.monolayer()
```

Or:

```
from pybinding.repository import phosphorene

lattice = phosphorene.monolayer_4band()
```

Graphene

Lattices

monolayer(*nearest_neighbors=1*, *onsite=(0, 0)*, ***kwargs*)

Monolayer graphene lattice up to `nearest_neighbors` hoppings

Parameters **nearest_neighbors** : int

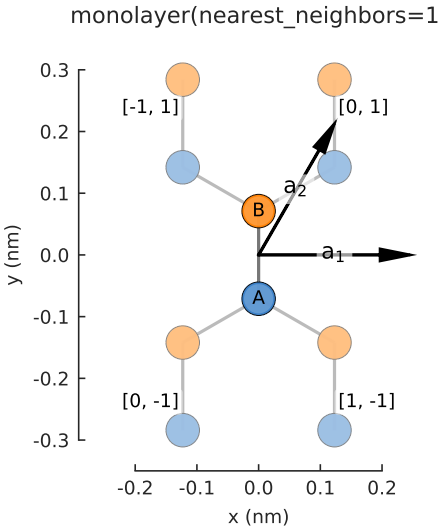
Number of nearest neighbors to consider.

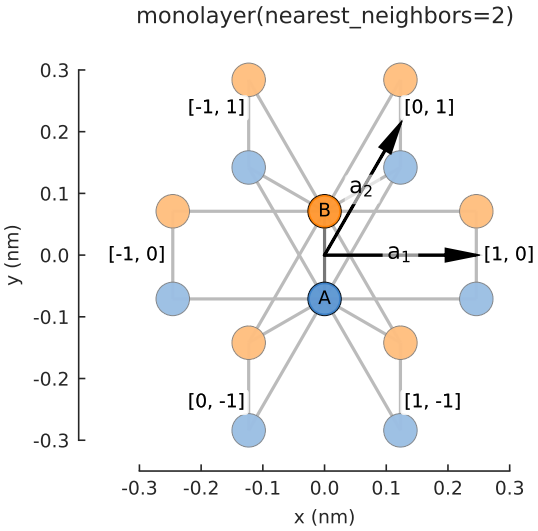
onsite : Tuple[float, float]

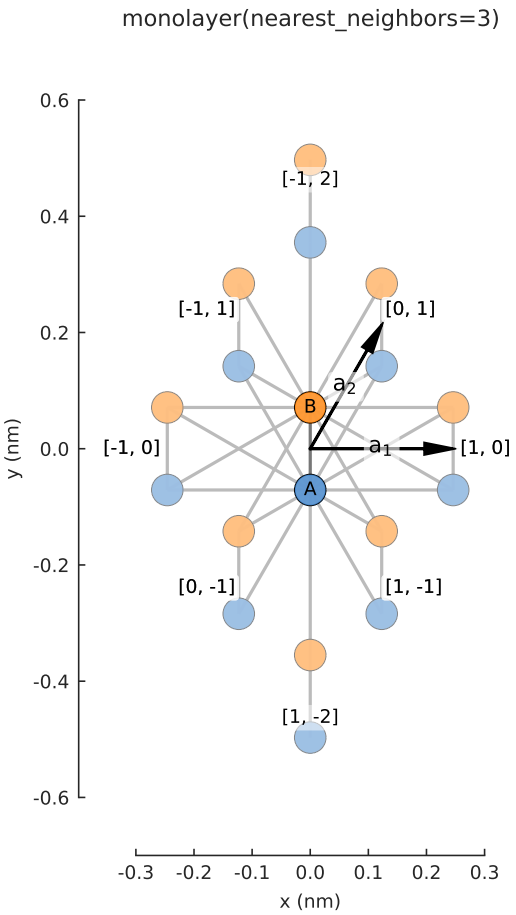
Onsite energy for sublattices A and B.

****kwargs**

Specify the hopping parameters `t`, `t_nn` and `t_nnn`. If not given, the default values from *`graphene.constants`* will be used.


monolayer(nearest_neighbors=1)
[-1, 1]
[0, 1]
a2
B
a1
A
[0, -1]
[1, -1]
y (nm)
x (nm)
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
-0.2
-0.1
0.0
0.1
0.2


monolayer(nearest_neighbors=2)
[-1, 1]
[0, 1]
a2
B
[-1, 0]
a1
[1, 0]
A
[0, -1]
[1, -1]
y (nm)
x (nm)
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
-0.3
-0.2
-0.1
0.0
0.1
0.2
0.3

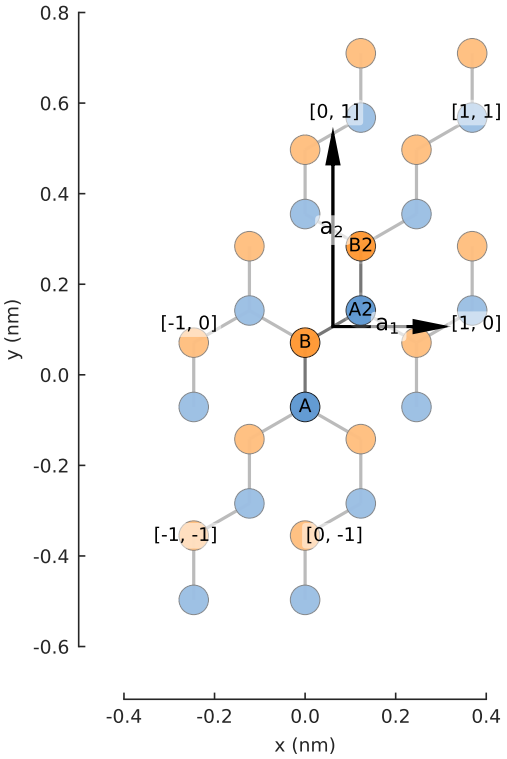


monolayer_4atom(*onsite=(0, 0)*)

Nearest-neighbor with 4 atoms per unit cell: square lattice instead of oblique

Parameters **onsite** : Tuple[float, float]

Onsite energy for sublattices A and B.



bilayer (*gamma3=False*, *gamma4=False*, *onsite=(0, 0, 0, 0)*)

Bilayer lattice in the AB-stacked form (Bernal-stacked)

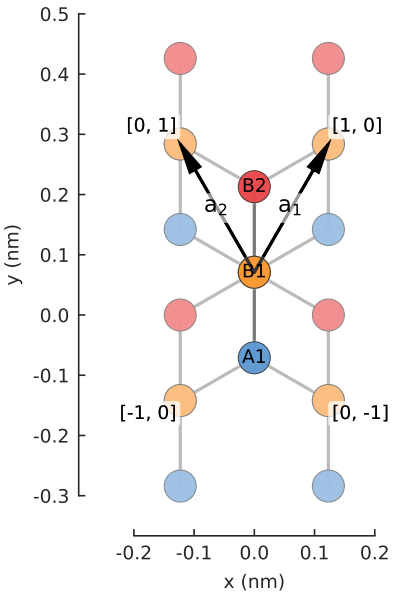
- γ_0 is the single-layer hopping within the top layer (A1/B1) and bottom layer (A2/B2)
- γ_1 is the inter-layer hopping between B1 and A2 (where atom B1 lies directly over A2)
- Hoppings γ_3 and γ_4 are optional (see parameters)

Parameters **gamma3, gamma4** : bool

Enable γ_3 and/or γ_4 hoppings. By default, only γ_0 and γ_1 are active.

onsite : Tuple[float, float, float, float]

Onsite energy for A1, B1, A2, B2



Constants

a = 0.24595
[nm] unit cell length

a_cc = 0.142
[nm] carbon-carbon distance

beta = 3.37
strain hopping modulation

t = -2.8
[eV] nearest neighbor hopping

t_nn = 0.1
[eV] next-nearest neighbor hopping

vf = 906091185689731.9
[nm/s] Fermi velocity

Shapes

hexagon_ac(*side_width*, *lattice_offset=(-0.122975, 0)*)
A graphene-specific shape which guaranties armchair edges on all sides

Parameters **side_width** : float

Hexagon side width. It may be adjusted slightly to ensure armchair edges.

lattice_offset : array_like

Offset the lattice so a carbon hexagon is at the center of the shape. The default value is specific for *monolayer()* and *bilayer()* lattices from this material repository.

Modifiers

mass_term(*delta*)
Break sublattice symmetry, make massive Dirac electrons, open a band gap

Only for monolayer graphene.

Parameters **delta** : float

Onsite energy +delta is added to sublattice 'A' and -delta to 'B'.

coulomb_potential(*beta*, *cutoff_radius=0.0*, *offset=(0, 0, 0)*)
A Coulomb potential created by an impurity in graphene

Parameters **beta** : float

Charge of the impurity [unitless].

cutoff_radius : float

Cut off the potential below this radius [nm].

offset: array_like

Position of the charge.

constant_magnetic_field(*magnitude*)
Constant magnetic field in the z-direction, perpendicular to the graphene plane

Parameters **magnitude** : float

In units of Tesla.

triaxial_strain(*magnetic_field*)
Triaxial strain corresponding to a homogeneous pseudo-magnetic field

Parameters **magnetic_field** : float

Intensity of the pseudo-magnetic field to induce.

gaussian_bump(*height*, *sigma*, *center=(0, 0)*)
Gaussian bump deformation

Parameters **height** : float

Height of the bump [nm].

sigma : float

Gaussian sigma parameter: controls the width of the bump [nm].

center : array_like

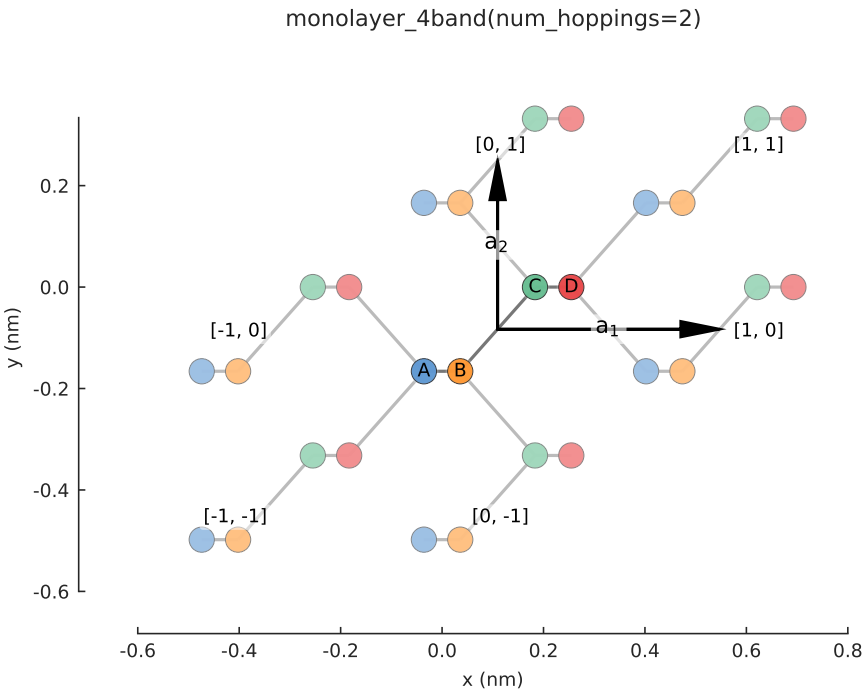
Position of the center of the bump.

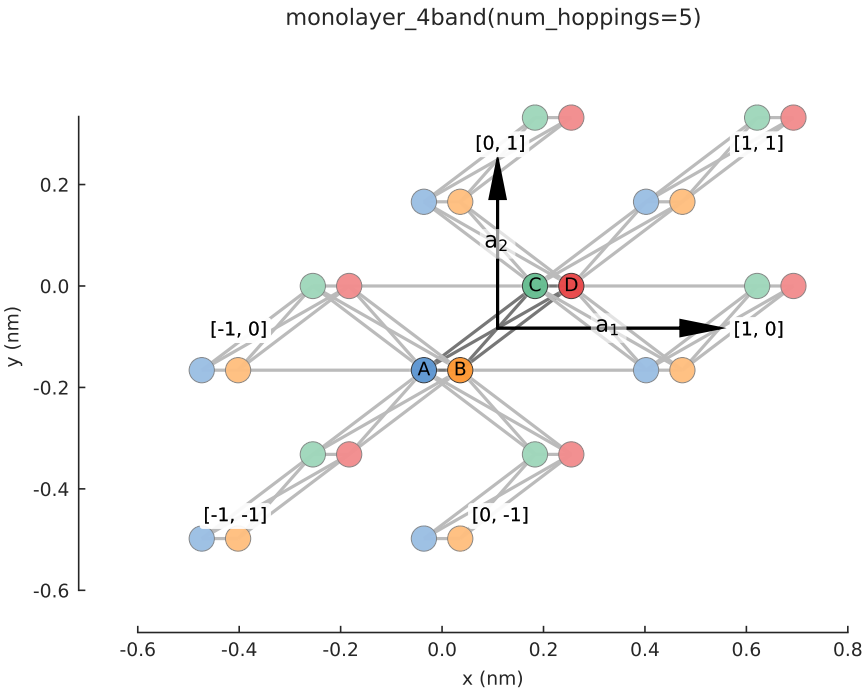
Phosphorene

monolayer_4band(*num_hoppings=5*)
Monolayer phosphorene lattice using the four-band model

Parameters **num_hoppings** : int

Number of hopping terms to consider: from t2 to t5.





Group 6 TMDs

Tight-binding models for group 6 transition metal dichalcogenides (TMD).

monolayer_3band(*name*, *override_params=None*)

Monolayer of a group 6 TMD using the nearest-neighbor 3-band model

Parameters **name** : str

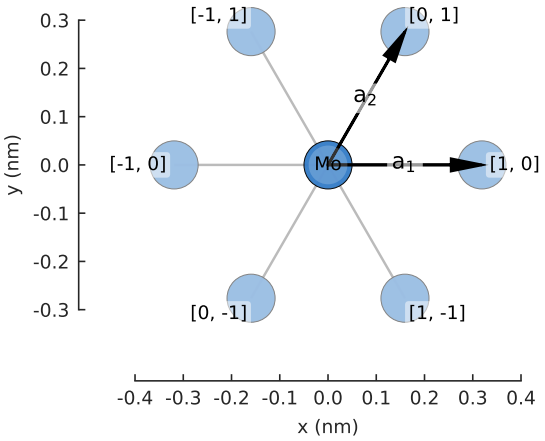
Name of the TMD to model. The available options are: MoS2, WS2, MoSe2, WSe2, MoTe2, WTe2. The relevant tight-binding parameters for these materials are given by https://doi.org/10.1103/PhysRevB.88.085433.

override_params : Optional[dict]

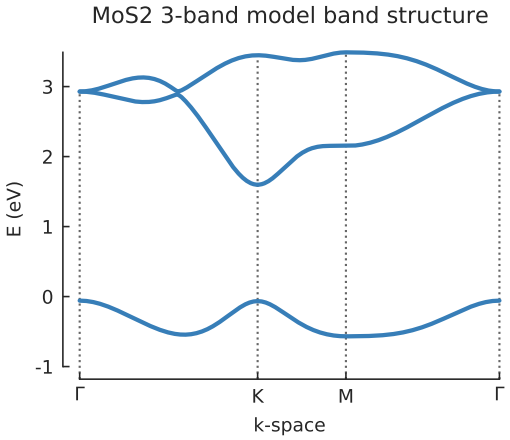
Replace or add new material parameters. The dictionary entries must be in the format `"name": [a, eps1, eps2, t0, t1, t2, t11, t12, t22]`.

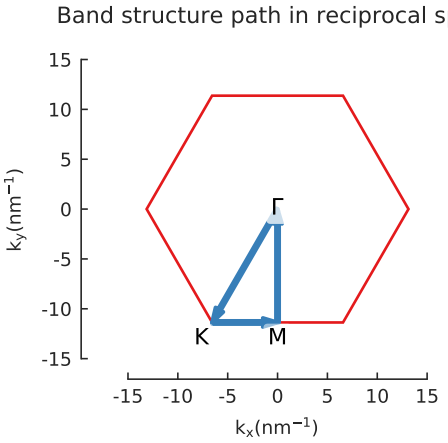
Examples

```
from pybinding.repository import group6_tmd

group6_tmd.monolayer_3band("MoS2").plot()
```

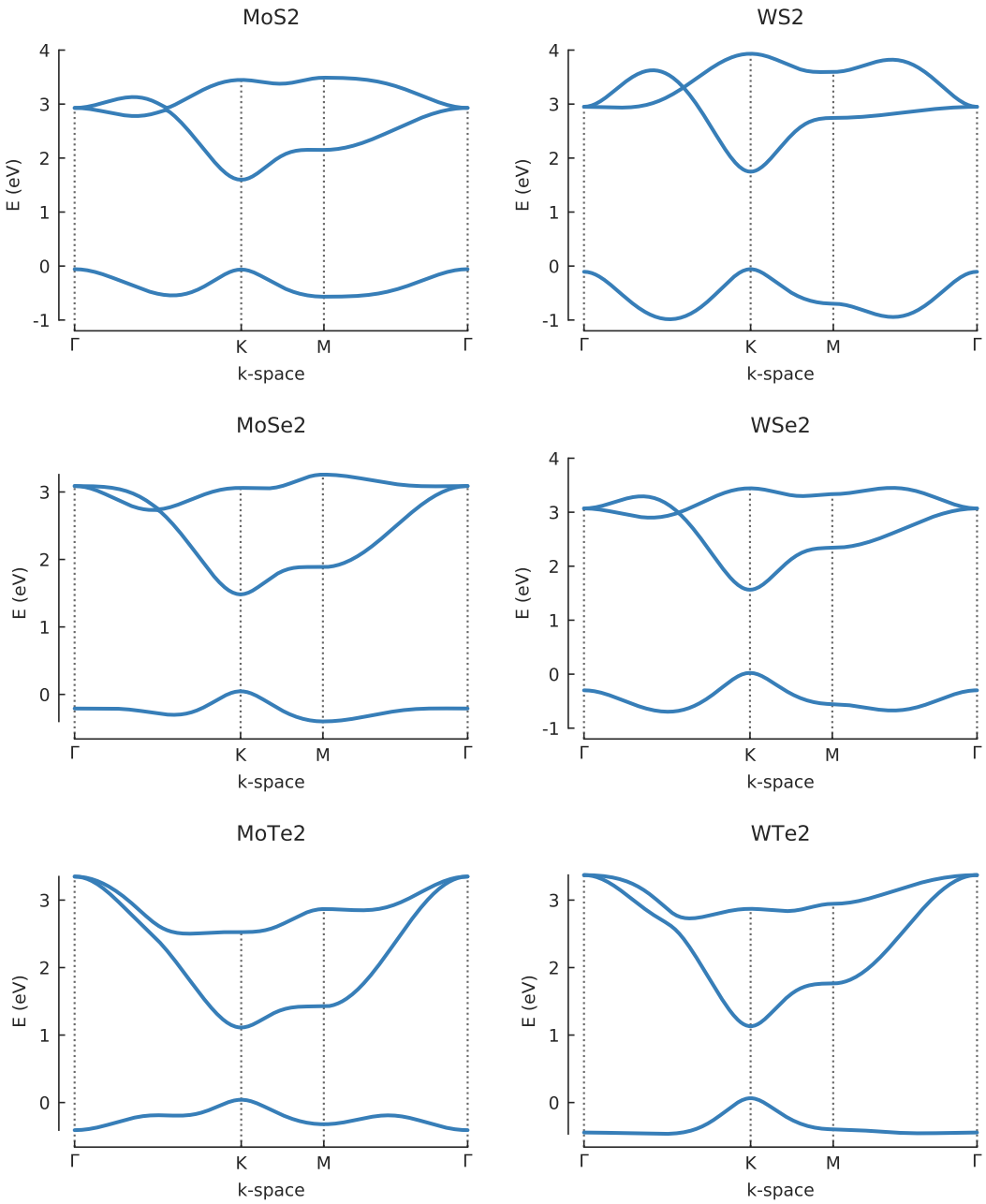



```
model = pb.Model(group6_tmd.monolayer_3band("MoS2"), pb.translational_
↪symmetry())
solver = pb.solver.lapack(model)

k_points = model.lattice.brillouin_zone()
gamma = [0, 0]
k = k_points[0]
m = (k_points[0] + k_points[1]) / 2

plt.figure(figsize=(6.7, 2.3))

plt.subplot(121, title="MoS2 3-band model band structure")
bands = solver.calc_bands(gamma, k, m, gamma)
bands.plot(point_labels=[r"$\Gamma$", "K", "M", r"$\Gamma$"])

plt.subplot(122, title="Band structure path in reciprocal space")
model.lattice.plot_brillouin_zone(decorate=False)
bands.plot_kpath(point_labels=[r"$\Gamma$", "K", "M", r"$\Gamma$"])
```






```
grid = plt.GridSpec(3, 2, hspace=0.4)
plt.figure(figsize=(6.7, 8))

for square, name in zip(grid, ["MoS2", "WS2", "MoSe2", "WSe2", "MoTe2", "WTe2
↪"]):
    model = pb.Model(group6_tmd.monolayer_3band(name), pb.translational_
↪symmetry())
    solver = pb.solver.lapack(model)

    k_points = model.lattice.brillouin_zone()
    gamma = [0, 0]
    k = k_points[0]
    m = (k_points[0] + k_points[1]) / 2

    plt.subplot(square, title=name)
    bands = solver.calc_bands(gamma, k, m, gamma)
    bands.plot(point_labels=[r"$\Gamma$", "K", "M", r"$\Gamma$"], lw=1.5)
```


MoS2
WS2
MoSe2
WSe2
MoTe2
WTe2
E (eV)
k-space
Γ
K
M

CHAPTER 13

API Reference

This detailed reference lists all the classes and functions contained in the package. If you are just looking to get started, read the *Tutorial* first.

The `Lattice` describes the unit cell of a crystal, while the `Model` is used to build up a larger system by translating the unit cell to fill a certain shape or symmetry. The model builds the Hamiltonian matrix by applying fields and other modifier parameters.

`Lattice`(a1[, a2, a3])	Unit cell of a Bravais lattice, the basic building block of a tight-binding model
`Model`(lattice, *args)	Builds a Hamiltonian from lattice, shape, symmetry and modifier parameters

Lattice

class **`Lattice`**(*a1*, *a2=None*, *a3=None*)

Unit cell of a Bravais lattice, the basic building block of a tight-binding model

This class describes the primitive vectors, positions of sublattice sites and hopping parameters which connect those sites. All of this structural information is used to build up a larger system by translation.

A few prebuilt lattices are available in the *Material Repository*.

Parameters **a1, a2, a3** : array_like

Primitive vectors of a Bravais lattice. A valid lattice must have at least one primitive vector (`a1`), thus forming a simple 1-dimensional lattice. If `a2` is also specified, a 2D lattice is created. Passing values for all three vectors will create a 3D lattice.

Attributes

`hoppings`	Dict of names and `HoppingFamily`
`min_neighbors`	Minimum number of neighbours required at each lattice site

Continued on next page

Table 13.2 – continued from previous page

ndim	The dimensionality of the lattice: number of primitive vectors
nhop	Number of hopping families
nsub	Number of sublattices
offset	Global lattice offset: sublattice offsets are defined relative to this
sublattices	Dict of names and `Sublattice`
vectors	Primitive lattice vectors

Methods

add_aliases(*aliases)	Add multiple new aliases
add_hoppings(*hoppings)	Add multiple new hoppings
add_one_alias(name, original, position)	Add a sublattice alias - useful for creating supercells
add_one_hopping(relative_index, from_sub, ...)	Add a new hopping
add_one_sublattice(name, position[, ...])	Add a new sublattice
add_sublattices(*sublattices)	Add multiple new sublattices
brillouin_zone()	Return a list of vertices which form the Brillouin zone (1D and 2D only)
plot([axes, vector_position])	Illustrate the lattice by plotting the primitive cell and its nearest neighbors
plot_brillouin_zone([decorate])	Plot the Brillouin zone and reciprocal lattice vectors
plot_vectors(position[, scale])	Plot lattice vectors in the xy plane
reciprocal_vectors()	Calculate the reciprocal space lattice vectors
register_hopping_energies(mapping)	Register a mapping of user-friendly names to hopping energies
site_radius_for_plot([max_fraction])	Return a good estimate for the lattice site radius for plotting
with_min_neighbors(number)	Return a copy of this lattice with a different minimum neighbor count
with_offset(position)	Return a copy of this lattice with a different offset

add_aliases(**aliases*)
Add multiple new aliases

Parameters ***aliases**

Each element should be a tuple containing the arguments for *add_one_alias()*. Works just like *add_sublattices()*.

add_hoppings(**hoppings*)
Add multiple new hoppings

Parameters ***hoppings**

Each element should be a tuple containing the arguments for a `add_one_hopping()` method call. See example.

Examples

These three calls:

```
lattice.add_one_hopping([0, 0], 'a', 'b', 0.8)
lattice.add_one_hopping([0, 1], 'a', 'a', 0.3)
lattice.add_one_hopping([1, 1], 'a', 'b', 0.8)
```

Can be replaced with a single call to:

```
lattice.add_hoppings(
    ([0, 0], 'a', 'b', 0.8),
    ([0, 1], 'a', 'a', 0.3),
    ([1, 1], 'a', 'b', 0.8),
)
```

add_one_alias(*name*, *original*, *position*)
Add a sublattice alias - useful for creating supercells

Create a new sublattice called `name` with the same properties as `original` (same onsite energy) but with at a different `position`. The new `name` is only used during lattice construction and the `original` will be used for the final system and Hamiltonian. This is useful when defining a supercell which contains multiple sites of one sublattice family at different positions.

Parameters **name** : str

User-friendly identifier of the alias.

original : str

Name of the original sublattice. It must already exist.

position : array_like

Cartesian position with respect to the origin. Usually different than the original.

add_one_hopping(*relative_index*, *from_sub*, *to_sub*, *hop_name_or_energy*)
Add a new hopping

For each new hopping, its Hermitian conjugate is added automatically. Doing so manually, i.e. adding a hopping which is the Hermitian conjugate of an existing one, will result in an exception being raised.

Parameters **relative_index** : array_like of int

Difference of the indices of the source and destination unit cells.

from_sub : str

Name of the sublattice in the source unit cell.

to_sub : str

Name of the sublattice in the destination unit cell.

hop_name_or_energy : float or str

The numeric value of the hopping energy or the name of a previously registered hopping.

add_one_sublattice(*name*, *position*, *onsite_energy=0.0*, *alias=''*)
Add a new sublattice

Parameters **name** : str

User-friendly identifier. The unique sublattice ID can later be accessed via this sublattice name as `lattice[sublattice_name]`.

position : array_like

Cartesian position with respect to the origin.

onsite_energy : float

Onsite energy to be applied only to sites of this sublattice.

alias : str

Deprecated: Use *add_one_alias()* instead.

add_sublattices(**sublattices*)
Add multiple new sublattices

Parameters ***sublattices**

Each element should be a tuple containing the arguments for a `add_one_sublattice()` method call. See example.

Examples

These three calls:

```
lattice.add_one_sublattice('a', [0, 0], 0.5)
lattice.add_one_sublattice('b', [0, 1], 0.0)
lattice.add_one_sublattice('c', [1, 0], 0.3)
```

Can be replaced with a single call to:

```
lattice.add_sublattices(
    ('a', [0, 0], 0.5),
    ('b', [0, 1], 0.0),
    ('c', [1, 0], 0.3)
)
```

brillouin_zone()
Return a list of vertices which form the Brillouin zone (1D and 2D only)

Returns List[array_like]

Examples

```
>>> lat_1d = Lattice(a1=1)
>>> np.allclose(lat_1d.brillouin_zone(), [-pi, pi])
True
>>> lat_2d = Lattice(a1=[0, 1], a2=[0.5, 0.5])
>>> np.allclose(lat_2d.brillouin_zone(), [[0, -2*pi], [2*pi, 0], [0, 2*pi],
↪ [-2*pi, 0]])
True
```

plot(*axes='xy'*, *vector_position='center'*, ***kwargs*)
Illustrate the lattice by plotting the primitive cell and its nearest neighbors

Parameters **axes** : str

The spatial axes to plot. E.g. 'xy', 'yz', etc.

vector_position : array_like or 'center'

Cartesian position to be used as the origin for the lattice vectors. By default the origin is placed in the center of the primitive cell.

****kwargs**

Forwarded to `System.plot()`.

plot_brillouin_zone(*decorate=True*, ***kwargs*)
Plot the Brillouin zone and reciprocal lattice vectors

Parameters **decorate** : bool

Label the vertices of the Brillouin zone and show the reciprocal vectors

****kwargs**

Forwarded to `plt.plot()`.

plot_vectors (*position*, *scale=1.0*)
Plot lattice vectors in the xy plane

Parameters **position** : array_like

Cartesian position to be used as the origin for the vectors.

scale : float

Multiply the length of the vectors by this number.

reciprocal_vectors ()
Calculate the reciprocal space lattice vectors

Returns list

Examples

```
>>> lat = Lattice(a1=[0, 1], a2=[0.5, 0.5])
>>> np.allclose(lat.reciprocal_vectors(), [[4*pi, 0, 0], [-2*pi, 2*pi, 0]])
True
```

register_hopping_energies (*mapping*)
Register a mapping of user-friendly names to hopping energies

Parameters **mapping** : dict

Keys are user-friendly hopping names and values are the numeric values of the hopping energy.

site_radius_for_plot (*max_fraction=0.33*)
Return a good estimate for the lattice site radius for plotting

Calculated heuristically base on the length (1D) or area (2D) of the unit cell. In order to prevent overlap between sites, if the computed radius is too large, it will be clamped to a fraction of the shortest inter-atomic spacing.

Parameters **max_fraction** : float

Set the upper limit of the calculated radius as this fraction of the shortest inter-atomic spacing in the lattice unit cell. Should be less than 0.5 to avoid overlap between neighboring lattice sites.

Returns float

with_min_neighbors (*number*)
Return a copy of this lattice with a different minimum neighbor count

Parameters **number** : int

The minimum number of neighbors.

Returns *Lattice*

with_offset (*position*)
Return a copy of this lattice with a different offset

It must be within half the length of a primitive lattice vector

Parameters **position** : array_like

Cartesian offset in the same length unit as the lattice vectors.

Returns *Lattice*

hoppings
Dict of names and `HoppingFamily`

min_neighbors
Minimum number of neighbours required at each lattice site

When constructing a finite-sized system, lattice sites with less neighbors than this minimum will be considered as "dangling" and they will be removed.

ndim
The dimensionality of the lattice: number of primitive vectors

nhop
Number of hopping families

nsub
Number of sublattices

offset
Global lattice offset: sublattice offsets are defined relative to this

It must be within half the length of a primitive lattice vector.

sublattices
Dict of names and `Sublattice`

vectors
Primitive lattice vectors

Model

class **Model**(*lattice*, **args*)
Builds a Hamiltonian from lattice, shape, symmetry and modifier parameters

The most important attributes are *system* and *hamiltonian* which are constructed based on the input parameters. The *System* contains structural data like site positions. The tight-binding Hamiltonian is a sparse matrix in the `scipy.sparse.csr_matrix` format.

Parameters **lattice** : *Lattice*

The lattice specification.

***args**

Can be any of: shape, symmetry or various modifiers. Note that:

- There can be at most one shape and at most one symmetry. Shape and symmetry can be composed as desired, but physically impossible scenarios will result in an empty system.
- Any number of modifiers can be added. Adding the same modifier more than once is allowed: this will usually multiply the modifier's effect.

Attributes

hamiltonian	Hamiltonian sparse matrix in the `scipy.sparse.csr_matrix` format
lattice	*Lattice* specification
leads	List of *Lead* objects
modifiers	List of all modifiers applied to this model
onsite_map	*StructureMap* of the onsite energy
shape	*Polygon* or *FreeformShape* object
system	Structural data like site positions and hoppings, see *System* for details

Methods

add(*args)	Add parameter(s) to the model
attach_lead(direction, contact)	Attach a lead to the main system
eval()	
plot([num_periods, lead_length, axes])	Plot the structure of the model: sites, hoppings, boundaries and leads
report()	Return a string with information about the last build
set_wave_vector(k: numpy.ndarray)	Set the wave vector for periodic models
structure_map(data)	Return a *StructureMap* of the model system mapped to the specified `data`
tokwant()	Convert this model into kwant format (finalized)

add(**args*)
Add parameter(s) to the model

Parameters ***args**

Any of: shape, symmetry, modifiers. Tuples and lists of parameters are expanded automatically, so `M.add(p0, [p1, p2])` is equivalent to `M.add(p0, p1, p2)`.

attach_lead(*direction*, *contact*)
Attach a lead to the main system

Not valid for 1D lattices.

Parameters **direction** : int

Lattice vector direction of the lead. Must be one of: 1, 2, 3, -1, -2, -3. For example, `direction=2` would create a lead which intersects the main system in the a_2 lattice vector direction. Setting `direction=-2` would create a lead on the opposite side of the system, but along the same lattice vector.

contact : `Shape`

The place where the lead should contact the main system. For a 2D lattice it's just a *line()* describing the intersection of the lead and the system. For a 3D lattice it's the area described by a 2D *FreeformShape*.

eval()

plot(*num_periods=1*, *lead_length=6*, *axes='xy'*, ***kwargs*)
Plot the structure of the model: sites, hoppings, boundaries and leads

Parameters **num_periods** : int

Number of times to repeat the periodic boundaries.

lead_length : int

Number of times to repeat the lead structure.

axes : str

The spatial axes to plot. E.g. 'xy', 'yz', etc.

****kwargs**

Additional plot arguments as specified in *structure_plot_properties()*.

report()
Return a string with information about the last build

set_wave_vector(*k: numpy.ndarray*)
Set the wave vector for periodic models

Parameters **k** : array_like

Wave vector in reciprocal space.

structure_map(*data*)
Return a *StructureMap* of the model system mapped to the specified `data`

Parameters **data** : Optional[array_like]

Data array to map to site positions.

Returns `StructureMap`

tokwant()
Convert this model into kwant format (finalized)

This is intended for compatibility with the kwant package: http://kwant-project.org/.

Returns kwant.system.System

Finalized system which can be used with kwant compute functions.

hamiltonian
Hamiltonian sparse matrix in the `scipy.sparse.csr_matrix` format

lattice
Lattice specification

leads
List of *Lead* objects

modifiers
List of all modifiers applied to this model

onsite_map
StructureMap of the onsite energy

shape
Polygon or *FreeformShape* object

system
Structural data like site positions and hoppings, see *System* for details

Shapes

The geometry of a finite-sized system can be defined using the *Polygon* class (2D only) or using *FreeformShape* (1 to 3 dimensions). A few common shapes are included in the package and listed below. These predefined shapes are just functions which configure and return a shape class object.

Building blocks

Polygon(vertices)	Shape defined by a list of vertices in a 2D plane
FreeformShape(contains, width[, center])	Shape in 1 to 3 dimensions, defined by a function and a bounding box
CompositeShape(shape1, shape2, op)	A composition of 2 shapes using some operator (and, or, xor...)

Polygon

class Polygon(*vertices*)
Shape defined by a list of vertices in a 2D plane

Attributes

vertices	(List[array_like]) Must be defined in clockwise or counterclockwise order.

Methods

contains(x: numpy.ndarray[float32[m, 1]], ...)	Return `True` if the given position is located within the shape
plot(**kwargs)	Line plot of the polygon
with_offset(vector)	Return a copy that's offset by the given vector

contains(*x: numpy.ndarray[float32[m, 1]], y: numpy.ndarray[float32[m, 1]], z: numpy.ndarray[float32[m, 1]]*)
Return `True` if the given position is located within the shape

Given arrays as input the return type is a boolean array.

Parameters **x, y, z** : array_like

Positions to test against the shape.

plot(***kwargs*)
Line plot of the polygon

Parameters ****kwargs**

Forwarded to `matplotlib.pyplot.plot()`.

with_offset(*vector*)
Return a copy that's offset by the given vector

FreeformShape

class FreeformShape(*contains*, *width*, *center=(0, 0, 0)*)
Shape in 1 to 3 dimensions, defined by a function and a bounding box

Note that this class can describe 3D shapes, but the *plot()* method can currently only draw in 2D. Nevertheless, a *Model* will accept 3D shapes without a problem.

Parameters **contains** : callable

The function which selects if a point is contained within the shape.

width : array_like

Width up to 3 dimensions which specifies the size of the bounding box.

center : array_like

The position of the center of the bounding box.

Methods

contains(x: numpy.ndarray[float32[m, 1]], ...)	Return `True` if the given position is located within the shape
plot([resolution])	Plot a lightly shaded silhouette of the freeform shape
with_offset(vector)	Return a copy that's offset by the given vector

contains(*x: numpy.ndarray[float32[m, 1]], y: numpy.ndarray[float32[m, 1]], z: numpy.ndarray[float32[m, 1]]*)
Return `True` if the given position is located within the shape

Given arrays as input the return type is a boolean array.

Parameters **x, y, z** : array_like

Positions to test against the shape.

plot(*resolution=(1000, 1000), **kwargs*)
Plot a lightly shaded silhouette of the freeform shape

This method only works for 2D shapes.

Parameters **resolution** : Tuple[int, int]

The (x, y) pixel resolution of the generated shape image.

****kwargs**

Forwarded to `matplotlib.pyplot.imshow()`.

with_offset(*vector*)
Return a copy that's offset by the given vector

CompositeShape

class CompositeShape(*shape1*, *shape2*, *op*)
A composition of 2 shapes using some operator (and, or, xor...)

This shape is usually not created directly but present the result of applying logical or arithmetic operators on other shapes.

Parameters **shape1, shape2** : _cpp.Shape

The shapes which shall be composed.

op : Callable

A logical operator (and, or, xor...) to use for the composition.

Methods

contains(x: numpy.ndarray[float32[m, 1]], ...)	Return `True` if the given position is located within the shape
plot([resolution])	Plot a lightly shaded silhouette of the composite shape
with_offset(vector)	Return a copy that's offset by the given vector

contains(*x: numpy.ndarray[float32[m, 1]], y: numpy.ndarray[float32[m, 1]], z: numpy.ndarray[float32[m, 1]]*)
Return `True` if the given position is located within the shape

Given arrays as input the return type is a boolean array.

Parameters **x, y, z** : array_like

Positions to test against the shape.

plot(*resolution=(1000, 1000), **kwargs*)
Plot a lightly shaded silhouette of the composite shape

Parameters **resolution** : Tuple[int, int]

The (x, y) pixel resolution of the generated image.

****kwargs**

Forwarded to `matplotlib.pyplot.imshow()`.

with_offset(*vector*)
Return a copy that's offset by the given vector

Predefined shapes

circle(radius[, center])	A circle in the xy plane
line(a, b)	A line shape intended for 1D lattices or to specify leads for 2D lattices
primitive([a1, a2, a3])	Follow the primitive lattice shape – just repeat the unit cell a number of times
rectangle(x[, y])	A rectangle in the xy plane
regular_polygon(num_sides, radius[, angle])	A polygon shape where all sides have equal length

circle

circle(*radius*, *center=(0, 0)*)
A circle in the xy plane

Parameters **radius** : float

center : array_like

Returns *FreeformShape*

line

line(*a*, *b*)
A line shape intended for 1D lattices or to specify leads for 2D lattices

Parameters **a, b** : Union[float, array_like]

Start and end points.

Returns `Line`

primitive

primitive(*a1=1*, *a2=1*, *a3=1*)
Follow the primitive lattice shape – just repeat the unit cell a number of times

Parameters **a1, a2, a3** : int or float

Number of times to repeat the unit cell in the respective lattice vector directions.

Returns `Primitive`

rectangle

rectangle(*x*, *y=None*)
A rectangle in the xy plane

Parameters **x** : float

Width of the rectangle.

y : float, optional

Height of the rectangle. If not given, assumed equal to `x`.

Returns *Polygon*

regular_polygon

regular_polygon(*num_sides*, *radius*, *angle=0*)
A polygon shape where all sides have equal length

Parameters **num_sides** : int

Number of sides.

radius : float

Radius of the circle which connects all the vertices of the polygon.

angle : float

Rotate the polygon.

Returns *Polygon*

Symmetry

translational_symmetry([a1, a2, a3])	Simple translational symmetry

translational_symmetry

translational_symmetry(*a1=True*, *a2=True*, *a3=True*)
Simple translational symmetry

Parameters **a1, a2, a3** : bool or float

Control translation in the 'a1, a2, a3' lattice vector directions. Possible values:

- False -> No translational symmetry in this direction.
- True -> Translation length is automatically set to the unit cell length.
- float value -> Manually set the translation length in nanometers.

Modifiers

The following decorators are used to create functions which express some feature of a tight-binding model, such as various fields, defects or geometric deformations.

Decorators

site_state_modifier([min_neighbors])	Modify the state (valid or invalid) of lattice sites, e.g. to create vacancies
site_position_modifier()	Modify the position of lattice sites, e.g. to apply geometric deformations
onsite_energy_modifier([is_double])	Modify the onsite energy, e.g. to apply an electric field
hopping_energy_modifier([is_double, is_complex])	Modify the hopping energy, e.g. to apply a magnetic field

site_state_modifier

site_state_modifier(*min_neighbors=0*)

Modify the state (valid or invalid) of lattice sites, e.g. to create vacancies

Parameters **min_neighbors** : int

After modification, remove dangling sites with less than this number of neighbors.

Notes

The function parameters must be a combination of any number of the following:

state [ndarray of bool] Indicates if a lattice site is valid. Invalid sites will be removed from the model after all modifiers have been applied.

x, y, z [ndarray] Lattice site position.

sub_id [ndarray] Sublattice identifier: Can be checked for equality with sublattice names specified in *Lattice*. For example, `state[sub_id == 'A'] = False` will invalidate only sites on sublattice A.

sites [*Sites*] Helper object. Can be used instead of `x, y, z, sub_id`. See *Sites*.

The function must return:

ndarray A modified `state` argument or an `ndarray` of the same dtype and shape.

Examples

```
def vacancy(position, radius):
    @pb.site_state_modifier
    def f(state, x, y):
        x0, y0 = position
        state[(x-x0)**2 + (y-y0)**2 < radius**2] = False
        return state
    return f

model = pb.Model(
    ... # lattice, shape, etc.
    vacancy(position=[0, 0], radius=0.1)
)
```

site_position_modifier

site_position_modifier()

Modify the position of lattice sites, e.g. to apply geometric deformations

Notes

The function parameters must be a combination of any number of the following:

x, y, z [ndarray] Lattice site position.

sub_id [ndarray of int] Sublattice identifier: can be checked for equality with sublattice names specified in *Lattice*. For example, `x[sub_id == 'A'] += 0.1` will only displace sites on sublattice A.

sites [*Sites*] Helper object. Can be used instead of `x, y, z, sub_id`. See *Sites*.

The function must return:

tuple of ndarray Modified 'x, y, z' arguments or 3 `ndarray` objects of the same dtype and shape.

Examples

```
def triaxial_displacement(c):
    @pb.site_position_modifier
    def displacement(x, y, z):
        ux = 2*c * x*y
        uy = c * (x**2 - y**2)
        return x + ux, y + uy, z
    return displacement

model = pb.Model(
    ... # lattice, shape, etc.
    triaxial_displacement(c=0.15)
)
```

onsite_energy_modifier

onsite_energy_modifier(*is_double=False*, ***kwargs*)

Modify the onsite energy, e.g. to apply an electric field

Parameters **is_double** : bool

Requires the model to use double precision floating point values. Defaults to single precision otherwise.

Notes

The function parameters must be a combination of any number of the following:

energy [ndarray] The onsite energy.

x, y, z [ndarray] Lattice site position.

sub_id [ndarray of int] Sublattice identifier: can be checked for equality with sublattice names specified in *Lattice*. For example, `energy[sub_id == 'A'] = 0` will set the onsite energy only for sublattice A sites.

sites [*Sites*] Helper object. Can be used instead of `x, y, z, sub_id`. See *Sites*.

The function must return:

ndarray A modified `potential` argument or an `ndarray` of the same dtype and shape.

Examples

```
def wavy(a, b):
    @pb.onsite_energy_modifier
    def f(x, y):
        return np.sin(a * x)**2 + np.cos(b * y)**2
    return f

model = pb.Model(
    ... # lattice, shape, etc.
    wavy(a=0.6, b=0.9)
)
```

hopping_energy_modifier

hopping_energy_modifier(*is_double=False*, *is_complex=False*, ***kwargs*)

Modify the hopping energy, e.g. to apply a magnetic field

Parameters **is_double** : bool

Requires the model to use double precision floating point values. Defaults to single precision otherwise.

is_complex : bool

Requires the model to use complex numbers. Even if this is set to `False`, the model will automatically switch to complex numbers if it finds that a modifier has returned complex numbers for real input. Manually setting this argument to `True` will speed up model build time slightly, but it's not necessary for correct operation.

Notes

The function parameters must be a combination of any number of the following:

energy [ndarray] The hopping energy between two sites.

x1, y1, z1, x2, y2, z2 [ndarray] Positions of the two lattice sites connected by the hopping parameter.

hop_id [ndarray of int] Hopping identifier: can be checked for equality with hopping names specified in *Lattice*. For example, `energy[hop_id == 't_nn'] *= 1.1` will only modify the energy of the hopping family named `t_nn`.

The function must return:

ndarray A modified `hopping` argument or an `ndarray` of the same dtype and shape.

Examples

```
def constant_magnetic_field(B):
    @pb.hopping_energy_modifier
    def f(energy, x1, y1, x2, y2):
        y = 0.5 * (y1 + y2) * 1e-9
        peierls = B * y * (x1 - x2) * 1e-9
        return energy * np.exp(1j * 2*pi/phi0 * peierls)
    return f

model = pb.Model(
    ... # lattice, shape, etc.
    constant_magnetic_field(B=10)
)
```

Predefined modifiers

constant_potential(magnitude)	Apply a constant onsite energy to every lattice site
force_double_precision()	Forces the model to use double precision even if that's not require by any modifier
force_complex_numbers()	Forces the model to use complex numbers even if that's not require by any modifier

constant_potential

constant_potential(*magnitude*)
Apply a constant onsite energy to every lattice site

Parameters **magnitude** : float

In units of eV.

force_double_precision

force_double_precision()
Forces the model to use double precision even if that's not require by any modifier

force_complex_numbers

force_complex_numbers()
Forces the model to use complex numbers even if that's not require by any modifier

Experimental

hopping_generator(name, energy)	Introduce a new hopping family (with a new `hop_id`) via a list of index pairs

hopping_generator

hopping_generator(*name*, *energy*)
Introduce a new hopping family (with a new `hop_id`) via a list of index pairs

This can be used to create new hoppings independent of the main *Lattice* definition. It's especially useful for creating additional local hoppings, e.g. to model defects.

Parameters **name** : string

Friendly identifier for the new hopping family.

energy : Union[float, complex]

Base hopping energy value.

Notes

The function parameters must be a combination of any number of the following:

x, y, z [np.ndarray] Lattice site position.

sub_id [np.ndarray] Sublattice identifier: can be checked for equality with sublattice names specified in *Lattice*.

The function must return:

Tuple[np.ndarray, np.ndarray] Arrays of index pairs which form the new hoppings.

Compute

After a *Model* is constructed, computational routines can be applied to determine various physical properties. The following submodules contain functions for exact diagonalization as well as some approximative compute

methods. Follow the links below for details.

solver	Eigensolvers with a few extra computation methods
chebyshev	Computations based on Chebyshev polynomial expansion

solver

Eigensolvers with a few extra computation methods

The *Solver* class is the main interface for dealing with eigenvalue problems. It is made to work specifically with pybinding's *Model* objects, but it may use any eigensolver algorithm under the hood.

A few different algorithms are provided out of the box: the *lapack()*, *arpack()* and *feast()* functions return concrete *Solver* implementation using the LAPACK, ARPACK and FEAST algorithms, respectively.

The *Solver* may easily be extended with new eigensolver algorithms. All that is required is a function which takes a Hamiltonian matrix and returns the computed eigenvalues and eigenvectors. See `_SolverPythonImpl` for example.

Classes

Solver(impl: _pybinding.Solver)	Computes the eigenvalues and eigenvectors of a Hamiltonian matrix

Functions

arpack(model, k[, sigma])	ARPACK *Solver* implementation for sparse matrices
feast(model, energy_range, initial_size_guess)	FEAST *Solver* implementation for sparse matrices
lapack(model, **kwargs)	LAPACK *Solver* implementation for dense matrices

class **Solver** (*impl: _pybinding.Solver*)

Computes the eigenvalues and eigenvectors of a Hamiltonian matrix

This the common interface for various eigensolver implementations. It should not be created directly, but via the specific functions: *lapack()*, *arpack()* and *feast()*. Those functions will set up their specific solver strategy and return a properly configured *Solver* object.

calc_bands (*k0*, *k1*, **ks*, *step=0.1*)

Calculate the band structure on a path in reciprocal space

Parameters **k0, k1, *ks** : array_like

Points in reciprocal space which form the path for the band calculation. At least two points are required.

step : float, optional

Calculation step length in reciprocal space units. Lower `step` values will return more detailed results.

Returns *Bands*

calc_dos (*energies*, *broadening*)

Calculate the density of states as a function of energy

$$\mathrm{DOS}(E) = \frac{1}{c\sqrt{2\pi}} \sum_n e^{-\frac{(E_n - E)^2}{2c^2}}$$

for each E in `energies`, where c is `broadening` and E_n is `eigenvalues[n]`.

Parameters **energies** : array_like

Values for which the DOS is calculated.

broadening : float

Controls the width of the Gaussian broadening applied to the DOS.

Returns *Series*

calc_eigenvalues(*map_probability_at=None*)

Return an *Eigenvalues* result object with an optional probability colormap

While the *eigenvalues* property returns the raw values array, this method returns a result object with more data. In addition to the energy states, this result may show a colormap of the probability density for each state at a single position.

Parameters **map_probability_at** : array_like, optional

Cartesian position where the probability density of each energy state should be calculated.

Returns *Eigenvalues*

calc_ldos(*energies*, *broadening*, *position*, *sublattice=''*, *reduce=True*)

Calculate the local density of states as a function of energy at the given position

$$\mathrm{LDOS}(E) = \frac{1}{c\sqrt{2\pi}} \sum_n |\Psi_n(r)|^2 e^{-\frac{(E_n - E)^2}{2c^2}}$$

for each E in `energies`, where c is `broadening`, E_n is `eigenvalues[n]` and r is a single site position determined by the arguments `position` and `sublattice`.

Parameters **energies** : array_like

Values for which the DOS is calculated.

broadening : float

Controls the width of the Gaussian broadening applied to the DOS.

position : array_like

Cartesian position of the lattice site for which the LDOS is calculated. Doesn't need to be exact: the method will find the actual site which is closest to the given position.

sublattice : str

Only look for sites of a specific sublattice, closest to `position`. The default value considers any sublattice.

reduce : bool

This option is only relevant for multi-orbital models. If true, the resulting LDOS will summed over all the orbitals at the target site and the result will be a 1D array. If false, the individual orbital results will be preserved and the result will be a 2D array with `shape == (energy.size, num_orbitals)`.

Returns *Series*

calc_probability(*n*, *reduce=1e-05*)

Calculate the spatial probability density

$$\mathrm{P}(r) = |\Psi_n(r)|^2$$

for each position r in `system.positions` where $\Psi_n(r)$ is `eigenvectors[:, n]`.

Parameters **n** : int or array_like

Index of the desired eigenstate. If an array of indices is given, the probability will be calculated at each one and a sum will be returned.

reduce : float, optional

Reduce degenerate states by summing their probabilities. Neighboring states are considered degenerate if their energy is difference is lower than the value of `reduce`. This is disabled by passing `reduce=0`.

Returns *StructureMap*

calc_spatial_ldos(*energy*, *broadening*)

Calculate the spatial local density of states at the given energy

$$\mathrm{LDOS}(r) = \frac{1}{c\sqrt{2\pi}} \sum_n |\Psi_n(r)|^2 e^{-\frac{(E_n - E)^2}{2c^2}}$$

for each position r in `system.positions`, where E is `energy`, c is `broadening`, E_n is `eigenvalues[n]` and $\Psi_n(r)$ is `eigenvectors[:, n]`.

Parameters **energy** : float

The energy value for which the spatial LDOS is calculated.

broadening : float

Controls the width of the Gaussian broadening applied to the DOS.

Returns *StructureMap*

clear()

Clear the computed results and start over

static **find_degenerate_states**(*energies*, *abs_tolerance=1e-05*)

Return groups of indices which belong to degenerate states

Parameters **energies** : array_like

abs_tolerance : float, optional

Examples

```
>>> energies = np.array([0.1, 0.1, 0.2, 0.5, 0.5, 0.5, 0.7, 0.8, 0.8])
>>> Solver.find_degenerate_states(energies)
[[0, 1], [3, 4, 5], [7, 8]]
```

```
>>> energies = np.array([0.1, 0.2, 0.5, 0.7])
>>> Solver.find_degenerate_states(energies)
[]
```

report(*shortform=False*) → str

Return a report of the last *solve()* computation

Parameters **shortform** : bool, optional

Return a short one line version of the report

set_wave_vector(*k*)

Set the wave vector for periodic models

Parameters **k** : array_like

Wave vector in reciprocal space.

solve()

Explicitly solve the eigenvalue problem right now

This method is usually not needed because the main result properties, *eigenvalues* and *eigenvectors*, will call this implicitly the first time they are accessed. However, since the *solve()* routine may be computationally expensive, it is useful to have the ability to call it ahead of time as needed.

eigenvalues
1D array of computed energy states

eigenvectors
2D array where each column represents a wave function

eigenvectors.shape == (system.num_sites, eigenvalues.size)

model
The tight-binding model attached to this solver

system
The tight-binding system attached to this solver (shortcut for Solver.model.system)

arpack(*model*, *k*, *sigma=0*, ***kwargs*)
ARPACK *Solver* implementation for sparse matrices

This solver is intended for large models with sparse Hamiltonian matrices. It only computes a small targeted subset of eigenvalues and eigenvectors. Internally this solver uses the `scipy.sparse.linalg.eigsh()` function for sparse Hermitian matrices.

Parameters **model** : Model

Model which will provide the Hamiltonian matrix.

k : int

The desired number of eigenvalues and eigenvectors. This number must be smaller than the size of the matrix, preferably much smaller for optimal performance. The computed eigenvalues are the ones closest to `sigma`.

sigma : float, optional

Look for eigenvalues near `sigma`.

****kwargs**

Advanced arguments: forwarded to `scipy.sparse.linalg.eigsh()`.

Returns *Solver*

feast(*model*, *energy_range*, *initial_size_guess*, *recycle_subspace=False*, *is_verbose=False*)
FEAST *Solver* implementation for sparse matrices

This solver is only available if the C++ extension module was compiled with FEAST.

Parameters **model** : Model

Model which will provide the Hamiltonian matrix.

energy_range : tuple of float

The lowest and highest eigenvalue between which to compute the solutions.

initial_size_guess : int

Initial user guess for number of eigenvalues which will be found in the given `energy_range`. This value may be completely wrong - the solver will autocorrect as needed. However, for optimal performance the estimate should be as close to 1.5 * actual_size as possible.

recycle_subspace : bool, optional

Reuse previously computed values as a starting point for the next computation. This improves performance when subsequent computations differ only slightly, as is the

case for the band structure of periodic systems where the results change gradually as a function of the wave vector. It may hurt performance otherwise.

is_verbose : bool, optional

Show the raw output from the FEAST routine.

Returns *Solver*

lapack(*model*, ***kwargs*)

LAPACK *Solver* implementation for dense matrices

This solver is intended for small models which are best represented by dense matrices. Always solves for all the eigenvalues and eigenvectors. Internally this solver uses the `scipy.linalg.eigh()` function for dense Hermitian matrices.

Parameters **model** : Model

Model which will provide the Hamiltonian matrix.

****kwargs**

Advanced arguments: forwarded to `scipy.linalg.eigh()`.

Returns *Solver*

chebyshev

Computations based on Chebyshev polynomial expansion

The kernel polynomial method (KPM) can be used to approximate various functions by expanding them in a series of Chebyshev polynomials.

Classes

KPM(impl)	The common interface for various KPM implementations
SpatialLDOS(data, energy, structure)	Holds the results of *KPM.calc_spatial_ldos()*

Functions

dirichlet_kernel()	The Dirichlet kernel – returns raw moments, least favorable choice
jackson_kernel()	The Jackson kernel – a good general-purpose kernel, appropriate for most applications
kpm(model[, energy_range, kernel, ...])	The default CPU implementation of the Kernel Polynomial Method
kpm_cuda(model[, energy_range, kernel])	Same as *kpm()* except that it's executed on the GPU using CUDA (if supported)
lorentz_kernel([lambda_value])	The Lorentz kernel – best for Green's function

class KPM(*impl*)

The common interface for various KPM implementations

It should not be created directly but via specific functions like *kpm()* or *kpm_cuda()*.

All implementations are based on: https://doi.org/10.1103/RevModPhys.78.275

calc_conductivity(*chemical_potential*, *broadening*, *temperature*, *direction='xx'*, *volume=1.0*, *num_random=1*, *num_points=1000*)

Calculate Kubo-Bastin electrical conductivity as a function of chemical potential

The return value is in units of the conductance quantum (e^2 / hbar) not taking into account spin or any other degeneracy.

The calculation is based on: https://doi.org/10.1103/PhysRevLett.114.116602.

Parameters **chemical_potential** : array_like

Values (in eV) for which the conductivity is calculated.

broadening : float

Width (in eV) of the smallest detail which can be resolved in the chemical potential. Lower values result in longer calculation time.

temperature : float

Value of temperature for the Fermi-Dirac distribution.

direction : Optional[str]

Direction in which the conductivity is calculated. E.g., "xx", "xy", "zz", etc.

volume : Optional[float]

The volume of the system.

num_random : int

The number of random vectors to use for the stochastic calculation of KPM moments. Larger numbers improve the quality of the result but also increase calculation time linearly. Fortunately, result quality also improves with system size, so the DOS of very large systems can be calculated accurately with only a small number of random vectors.

num_points : Optional[int]

Number of points for integration.

Returns *Series*

calc_dos(*energy*, *broadening*, *num_random=1*)

Calculate the density of states as a function of energy

Parameters **energy** : ndarray

Values for which the DOS is calculated.

broadening : float

Width, in energy, of the smallest detail which can be resolved. Lower values result in longer calculation time.

num_random : int

The number of random vectors to use for the stochastic calculation of KPM moments. Larger numbers improve the quality of the result but also increase calculation time linearly. Fortunately, result quality also improves with system size, so the DOS of very large systems can be calculated accurately with only a small number of random vectors.

Returns *Series*

calc_greens(*i*, *j*, *energy*, *broadening*)

Calculate Green's function of a single Hamiltonian element

Parameters **i, j** : int

Hamiltonian indices.

energy : ndarray

Energy value array.

broadening : float

Width, in energy, of the smallest detail which can be resolved. Lower values result in longer calculation time.

Returns ndarray

Array of the same size as the input `energy`.

calc_ldos(*energy*, *broadening*, *position*, *sublattice=''*, *reduce=True*)
Calculate the local density of states as a function of energy

Parameters **energy** : ndarray

Values for which the LDOS is calculated.

broadening : float

Width, in energy, of the smallest detail which can be resolved. Lower values result in longer calculation time.

position : array_like

Cartesian position of the lattice site for which the LDOS is calculated. Doesn't need to be exact: the method will find the actual site which is closest to the given position.

sublattice : str

Only look for sites of a specific sublattice, closest to `position`. The default value considers any sublattice.

reduce : bool

This option is only relevant for multi-orbital models. If true, the resulting LDOS will summed over all the orbitals at the target site and the result will be a 1D array. If false, the individual orbital results will be preserved and the result will be a 2D array with `shape == (energy.size, num_orbitals)`.

Returns *Series*

calc_spatial_ldos(*energy*, *broadening*, *shape*, *sublattice=''*)
Calculate the LDOS as a function of energy and space (in the area of the given shape)

Parameters **energy** : ndarray

Values for which the LDOS is calculated.

broadening : float

Width, in energy, of the smallest detail which can be resolved. Lower values result in longer calculation time.

shape : Shape

Determines the site positions at which to do the calculation.

sublattice : str

Only look for sites of a specific sublattice, within the `shape`. The default value considers any sublattice.

Returns *SpatialLDOS*

deferred_ldos(*energy*, *broadening*, *position*, *sublattice=''*)
Same as *calc_ldos()* but for parallel computation: see the *parallel* module

Parameters **energy** : ndarray

Values for which the LDOS is calculated.

broadening : float

Width, in energy, of the smallest detail which can be resolved. Lower values result in longer calculation time.

position : array_like

Cartesian position of the lattice site for which the LDOS is calculated. Doesn't need to be exact: the method will find the actual site which is closest to the given position.

sublattice : str

Only look for sites of a specific sublattice, closest to `position`. The default value considers any sublattice.

Returns Deferred

moments(*num_moments*, *alpha*, *beta=None*, *op=None*)

Calculate KPM moments in the form of expectation values

The result is an array of moments where each value is equal to:

$$\mu_n =< \beta|op \cdot T_n(H)|\alpha >$$

Parameters **num_moments** : int

The number of moments to calculate.

alpha : array_like

The starting state vector of the KPM iteration.

beta : Optional[array_like]

If not given, defaults to $\beta = \alpha$.

op : Optional[csr_matrix]

Operator in the form of a sparse matrix. If omitted, an identity matrix is assumed: $\mu_n =< \beta|T_n(H)|\alpha >$.

Returns ndarray

report(*shortform=False*)

Return a report of the last computation

Parameters **shortform** : bool, optional

Return a short one line version of the report

kernel

The damping kernel

model

The tight-binding model holding the Hamiltonian

scaling_factors

A tuple of KPM scaling factors `a` and `b`

system

The tight-binding system (shortcut for `KPM.model.system`)

class **SpatialLDOS**(*data*, *energy*, *structure*)

Holds the results of *KPM.calc_spatial_ldos()*

It behaves like a product of a *Series* and a *StructureMap*.

ldos(*position*, *sublattice=''*)

Return the LDOS as a function of energy at a specific position

Parameters **position** : array_like

sublattice : Optional[str]

Returns *Series*

structure_map(*energy*)
Return a *StructureMap* of the spatial LDOS at the given energy

Parameters **energy** : float

Produce a structure map for LDOS data closest to this energy value.

Returns *StructureMap*

kpm(*model*, *energy_range=None*, *kernel='default'*, *num_threads='auto'*, *silent=False*, ***kwargs*)
The default CPU implementation of the Kernel Polynomial Method

This implementation works on any system and is well optimized.

Parameters **model** : Model

Model which will provide the Hamiltonian matrix.

energy_range : Optional[Tuple[float, float]]

KPM needs to know the lowest and highest eigenvalue of the Hamiltonian, before computing the expansion moments. By default, this is determined automatically using a quick Lanczos procedure. To override the automatic boundaries pass a `(min_value, max_value)` tuple here. The values can be overestimated, but note that performance drops as the energy range becomes wider. On the other hand, underestimating the range will produce `NaN` values in the results.

kernel : Kernel

The kernel in the *Kernel* Polynomial Method. Used to improve the quality of the function reconstructed from the Chebyshev series. Possible values are *jackson_kernel()* or *lorentz_kernel()*. The Jackson kernel is used by default.

num_threads : int

The number of CPU threads to use for calculations. This is automatically set to the number of logical cores available on the current machine.

silent : bool

Don't show any progress messages.

Returns *KPM*

kpm_cuda(*model*, *energy_range=None*, *kernel='default'*, ***kwargs*)
Same as *kpm()* except that it's executed on the GPU using CUDA (if supported)

See *kpm()* for detailed parameter documentation. This method is only available if the C++ extension module was compiled with CUDA.

Parameters **model** : Model

energy_range : Optional[Tuple[float, float]]

kernel : Kernel

Returns *KPM*

jackson_kernel()
The Jackson kernel – a good general-purpose kernel, appropriate for most applications

Imposes Gaussian broadening `sigma = pi / N` where `N` is the number of moments. The broadening value is user-defined for each function calculation (LDOS, Green's, etc.). The number of moments is then determined based on the broadening – it's not directly set by the user.

lorentz_kernel(*lambda_value=4.0*)
The Lorentz kernel – best for Green's function

This kernel is most appropriate for the expansion of the Green's function because it most closely mimics the divergences near the true eigenvalues of the Hamiltonian. The Lorentzian broadening is given by `epsilon = lambda / N` where `N` is the number of moments.

Parameters **lambda_value** : float

May be used to fine-tune the smoothness of the convergence. Usual values are between 3 and 5. Lower values will speed up the calculation at the cost of accuracy. If in doubt, leave it at the default value of 4.

dirichlet_kernel()

The Dirichlet kernel – returns raw moments, least favorable choice

This kernel doesn't modify the moments at all. The resulting moments represent just a truncated series which results in lots of oscillation in the reconstructed function. Therefore, this kernel should almost never be used. It's only here in case the raw moment values are needed for some other purpose. Note that `required_num_moments()` returns `N = pi / sigma` for compatibility with the Jackson kernel, but there is no actual broadening associated with the Dirichlet kernel.

Experimental

parallel	Multi-threaded functions for parameter sweeps

parallel

Multi-threaded functions for parameter sweeps

Functions

ndsweep(factory[, plot, labels, tags, silent])	Do a multi-threaded n-dimensional parameter sweep
parallel_for(factory[, make_result])	Multi-threaded loop feed by the `factory` function
parallelize([num_threads, queue_size])	A decorator which creates factory functions for *parallel_for()*
sweep(factory[, plot, labels, tags, silent])	Do a multi-threaded parameter sweep

parallel_for(*factory*, *make_result=None*)

Multi-threaded loop feed by the `factory` function

Parameters **factory** : *Factory*

Factory function created with the *parallelize()* decorator.

make_result : callable, optional

Creates the final result from raw data. This result is also the final return value of *parallel_for()*.

Returns array_like

A result for each loop iteration.

Examples

```
@parallelize(x=np.linspace(0, 1, 10))
def factory(x):
    pb.Model(...)  # depends on `x`
    greens = pb.greens.kpm(model)
```

```
    return greens.deferred_ldos(...)  # may also depend on `x`

results = parallel_for(factory)
```

parallelize(*num_threads=num_cores*, *queue_size=num_cores*, ***kwargs*)

A decorator which creates factory functions for *parallel_for()*

The decorated function must return a `Deferred` compute kernel.

Parameters **num_threads** : int

Number of threads that will run in parallel. Defaults to the number of cores in the current machine.

queue_size : int

Number of `Deferred` jobs to be queued up for consumption by the worker threads. The maximum number of jobs that will be kept in memory at any one time will be `queue_size` + `num_threads`.

****kwargs**

Variables which will be iterated over in *parallel_for()* and passed to the decorated function. See example.

Examples

```
@parallelize(a=np.linspace(0, 1, 10), b=np.linspace(-2, 2, 10))
def factory(a, b):
    pb.Model(...)  # depends on `a` and `b`
    greens = pb.greens.kpm(model)
    return greens.deferred_ldos(...)  # may also depend on `a` and `b`

results = parallel_for(factory)
```

sweep(*factory*, *plot=<function <lambda>>*, *labels=None*, *tags=None*, *silent=False*)

Do a multi-threaded parameter sweep

Parameters **factory** : *Factory*

Factory function created with the *parallelize()* decorator.

plot : callable

Plotting functions which takes a *Sweep* result as its only argument.

labels, tags : dict

Forwarded to *Sweep* object.

silent : bool

Don't print status messages.

Returns *Sweep*

ndsweep(*factory*, *plot=None*, *labels=None*, *tags=None*, *silent=False*)

Do a multi-threaded n-dimensional parameter sweep

Parameters **factory** : *Factory*

Factory function created with the *parallelize()* decorator.

plot : callable

Plotting functions which takes a *NDSweep* result as its only argument.

labels, tags : dict

Forwarded to *NDSweep* object.

silent : bool

Don't print status messages.

Returns *NDSweep*

Results

Result objects are usually produced by compute functions, but they are also used to express certain model properties. They hold data and offer postprocessing and plotting methods specifically adapted to the nature of the physical properties (i.e. the stored data).

The utility functions *pb.save()* and *pb.load()* can be used to efficiently store entire result objects into files. The information about the kind of physical property is saved along with the raw data, i.e. executing `result = pb.load("data_file.pbz")` followed by `result.plot()` will work and present the appropriate figure.

save(obj, file)	Save an object to a compressed file
load(file)	Load an object from a compressed file
make_path(k0, k1, *ks[, step])	Create a path which connects the given k points
Bands(k_path, energy)	Band structure along a path in k-space
Eigenvalues(eigenvalues[, probability])	Hamiltonian eigenvalues with optional probability map
Series(variable, data[, labels])	A series of data points determined by a common relation, i.e.
SpatialMap(data, positions[, sublattices])	Represents some spatially dependent property: data mapped to site positions
StructureMap(data, sites, hoppings[, boundaries])	A subclass of *SpatialMap* that also includes hoppings between sites
Sweep(x, y, data[, labels, tags])	2D parameter sweep with `x` and `y` 1D array parameters and `data` 2D array result
NDSweep(variables, data[, labels, tags])	ND parameter sweep

save

save(*obj*, *file*)

Save an object to a compressed file

Essentially, this is just a wrapper for `pickle.dump()` with a few conveniences, like default pickle protocol 4 and gzip compression. The '.pbz' extension will be added if file has none.

Parameters **obj** : Any

Object to be saved.

file : Union[str, pathlib.Path]

May be a `str`, a `pathlib` object or a file object created with `open()`.

load

load(*file*)

Load an object from a compressed file

Wraps `pickle.load()` with the same conveniences as *pb.save()*.

Parameters **file** : Union[str, pathlib.Path]

May be a `str`, a `pathlib` object or a file object created with `open()`.

make_path

make_path (*k0*, *k1*, **ks*, *step=0.1*)

Create a path which connects the given k points

Parameters **k0, k1, *ks**

Points in k-space to connect.

step : float

Length in k-space between two samples. Smaller step -> finer detail.

Examples

```
>>> np.allclose(make_path(0, 3, -1, step=1).T, [0, 1, 2, 3, 2, 1, 0, -1])
True
>>> np.allclose(make_path([0, 0], [2, 3], [-1, 4], step=1.4),
...             [[0, 0], [1, 1.5], [2, 3], [0.5, 3.5], [-1, 4]])
True
```

Bands

class Bands (*k_path*, *energy*)

Band structure along a path in k-space

Attributes

k_path	(`Path`) The path in reciprocal space along which the bands were calculated. E.g. constructed using *`make_path()`*.
energy	(array_like) Energy values for the bands along the path in k-space.

Methods

`plot`([point_labels])	Line plot of the band structure
`plot_kpath`([point_labels])	Quiver plot of the k-path along which the bands were computed

plot (*point_labels=None*, ***kwargs*)

Line plot of the band structure

Parameters **point_labels** : List[str]

Labels for the `k_points`.

****kwargs**

Forwarded to `plt.plot()`.

plot_kpath (*point_labels=None*, ***kwargs*)

Quiver plot of the k-path along which the bands were computed

Combine with *`Lattice.plot_brillouin_zone()`* to see the path in context.

Parameters **point_labels** : List[str]

Labels for the k-points.

****kwargs**

Forwarded to `quiver()`.

Eigenvalues

class `Eigenvalues`(*eigenvalues*, *probability=None*)

Hamiltonian eigenvalues with optional probability map

Attributes

values	(np.ndarray)
probability	(np.ndarray)

Methods

plot([mark_degenerate, show_indices])	Standard eigenvalues scatter plot
plot_heatmap([size, mark_degenerate, ...])	Eigenvalues scatter plot with a heatmap indicating probability density

plot(*mark_degenerate=True*, *show_indices=False*, ***kwargs*)

Standard eigenvalues scatter plot

Parameters **mark_degenerate** : bool

Plot a line which connects degenerate states.

show_indices : bool

Plot index number next to all states.

****kwargs**

Forwarded to plt.scatter().

plot_heatmap(*size=(7, 77)*, *mark_degenerate=True*, *show_indices=False*, ***kwargs*)

Eigenvalues scatter plot with a heatmap indicating probability density

Parameters **size** : Tuple[int, int]

Min and max scatter dot size.

mark_degenerate : bool

Plot a line which connects degenerate states.

show_indices : bool

Plot index number next to all states.

****kwargs**

Forwarded to plt.scatter().

Series

class `Series`(*variable*, *data*, *labels=None*)

A series of data points determined by a common relation, i.e. $y = f(x)$

Attributes

variable	(array_like) Independent variable for which the data was computed.
data	(array_like) An array of values which were computed as a function of `variable`. It can be 1D or 2D. In the latter case each column represents the result of a different function applied to the same `variable` input.
labels	(dict) Plot labels: 'variable', 'data', 'title' and 'columns'.

Methods

plot(**kwargs)	Labeled line plot
reduced()	Return a copy where the data is summed over the columns
with_data(data)	Return a copy of this result object with different data

plot (***kwargs*)
Labeled line plot

Parameters ****kwargs**

Forwarded to `plt.plot()`.

reduced ()
Return a copy where the data is summed over the columns

Only applies to results which may have multiple columns of data, e.g. results for multiple orbitals for LDOS calculation.

with_data (*data*)
Return a copy of this result object with different data

SpatialMap

class SpatialMap (*data*, *positions*, *sublattices=None*)
Represents some spatially dependent property: data mapped to site positions

Attributes

data	1D array of values for each site, i.e. maps directly to x, y, z site coordinates
num_sites	Total number of lattice sites
positions	Lattice site positions.
sub	1D array of sublattices IDs, short for *.sublattices*
sublattices	1D array of sublattices IDs
x	1D array of coordinates, short for `.positions.x`
xyz	Return a new array with shape=(N, 3).
y	1D array of coordinates, short for `.positions.y`
z	1D array of coordinates, short for `.positions.z`

Methods

__getitem__(idx)	Same rules as numpy indexing
clipped(v_min, v_max)	Clip (limit) the values in the `data` array, see `clip()`
cropped(**limits)	Return a copy which retains only the sites within the given limits
plot_contour(**kwargs)	Contour plot of the xy plane
plot_contourf([num_levels])	Filled contour plot of the xy plane
plot_pcolor(**kwargs)	Color plot of the xy plane
with_data((data) -> pybinding.results.SpatialMap)	Return a copy of this object with different data mapped to the sites

__getitem__(*idx*)
Same rules as numpy indexing

clipped(*v_min*, *v_max*)
Clip (limit) the values in the `data` array, see `clip()`

cropped(***limits*)
Return a copy which retains only the sites within the given limits

Parameters ****limits**

Attribute names and corresponding limits. See example.

Examples

Leave only the data where -10 <= x < 10 and 2 <= y < 4:

```
new = original.cropped(x=[-10, 10], y=[2, 4])
```

plot_contour(***kwargs*)
Contour plot of the xy plane

Parameters ****kwargs**

Forwarded to `tricontour()`.

plot_contourf(*num_levels=50*, ***kwargs*)
Filled contour plot of the xy plane

Parameters **num_levels** : int

Number of contour levels.

****kwargs**

Forwarded to `tricontourf()`.

plot_pcolor(***kwargs*)
Color plot of the xy plane

Parameters ****kwargs**

Forwarded to `tripcolor()`.

with_data(*data*) → pybinding.results.SpatialMap
Return a copy of this object with different data mapped to the sites

data
1D array of values for each site, i.e. maps directly to x, y, z site coordinates

num_sites
Total number of lattice sites

positions
Lattice site positions. Named tuple with x, y, z fields, each a 1D array.

sub
1D array of sublattices IDs, short for *.sublattices*

sublattices
1D array of sublattices IDs

x
1D array of coordinates, short for `.positions.x`

xyz
Return a new array with shape=(N, 3). Convenient, but slow for big systems.

y
1D array of coordinates, short for `.positions.y`

z
1D array of coordinates, short for `.positions.z`

StructureMap

class **StructureMap**(*data*, *sites*, *hoppings*, *boundaries=()*)
A subclass of *SpatialMap* that also includes hoppings between sites

Attributes

boundaries	Boundary hoppings between different translation units (only for infinite systems)
data	1D array of values for each site, i.e. maps directly to x, y, z site coordinates
hoppings	Sparse matrix of hopping IDs
num_sites	Total number of lattice sites
positions	Lattice site positions.
spatial_map	Just the *SpatialMap* subset without hoppings
sub	1D array of sublattices IDs, short for *.sublattices*
sublattices	1D array of sublattices IDs
x	1D array of coordinates, short for `.positions.x`
xyz	Return a new array with shape=(N, 3).
y	1D array of coordinates, short for `.positions.y`
z	1D array of coordinates, short for `.positions.z`

Methods

__getitem__(idx)	Same rules as numpy indexing
clipped(v_min, v_max)	Clip (limit) the values in the `data` array, see `clip()`
cropped(**limits)	Return a copy which retains only the sites within the given limits
plot([cmap, site_radius, num_periods])	Plot the spatial structure with a colormap of *data* at the lattice sites
plot_contour(**kwargs)	Contour plot of the xy plane
plot_contourf([num_levels])	Filled contour plot of the xy plane
plot_pcolor(**kwargs)	Color plot of the xy plane
with_data(...)	Return a copy of this object with different data mapped to the sites

__getitem__(*idx*)
Same rules as numpy indexing

clipped(*v_min*, *v_max*)
Clip (limit) the values in the `data` array, see `clip()`

cropped(***limits*)
Return a copy which retains only the sites within the given limits

Parameters ****limits**

Attribute names and corresponding limits. See example.

Examples

Leave only the data where -10 <= x < 10 and 2 <= y < 4:

```
new = original.cropped(x=[-10, 10], y=[2, 4])
```

plot(*cmap='YlGnBu'*, *site_radius=(0.03, 0.05)*, *num_periods=1*, ***kwargs*)
Plot the spatial structure with a colormap of *data* at the lattice sites

Both the site size and color are used to display the data.

Parameters **cmap** : str

Matplotlib colormap to be used for the data.

site_radius : Tuple[float, float]

Min and max radius of lattice sites. This range will be used to visually represent the magnitude of the data.

num_periods : int

Number of times to repeat periodic boundaries.

****kwargs**

Additional plot arguments as specified in *structure_plot_properties()*.

plot_contour(***kwargs*)
Contour plot of the xy plane

Parameters ****kwargs**

Forwarded to `tricontour()`.

plot_contourf(*num_levels=50*, ***kwargs*)
Filled contour plot of the xy plane

Parameters **num_levels** : int

Number of contour levels.

****kwargs**

Forwarded to `tricontourf()`.

plot_pcolor(***kwargs*)
Color plot of the xy plane

Parameters ****kwargs**

Forwarded to `tripcolor()`.

with_data(*data*) → pybinding.results.StructureMap
Return a copy of this object with different data mapped to the sites

boundaries
Boundary hoppings between different translation units (only for infinite systems)

data
1D array of values for each site, i.e. maps directly to x, y, z site coordinates

hoppings
Sparse matrix of hopping IDs

num_sites
Total number of lattice sites

positions
Lattice site positions. Named tuple with x, y, z fields, each a 1D array.

spatial_map
Just the *SpatialMap* subset without hoppings

sub
1D array of sublattices IDs, short for *.sublattices*

sublattices
1D array of sublattices IDs

x
1D array of coordinates, short for `.positions.x`

xyz
Return a new array with shape=(N, 3). Convenient, but slow for big systems.

y
1D array of coordinates, short for `.positions.y`

z
1D array of coordinates, short for `.positions.z`

Sweep

class Sweep(*x*, *y*, *data*, *labels=None*, *tags=None*)
2D parameter sweep with `x` and `y` 1D array parameters and `data` 2D array result

Attributes

x	(array_like) 1D array with x-axis values – usually the primary parameter being swept.
y	(array_like) 1D array with y-axis values – usually the secondary parameter.
data	(array_like) 2D array with `shape == (x.size, y.size)` containing the main result data.
labels	(dict) Plot labels: 'title', 'x', 'y' and 'data'.
tags	(dict) Any additional user defined variables.

Methods

__getitem__(item)	Same rules as numpy indexing
colorbar(**kwargs)	Draw a colorbar with the label of `Sweep.data`
cropped([x, y])	Return a copy with data cropped to the limits in the x and/or y axes
interpolated([mul, size, kind])	Return a copy with interpolate data using `scipy.interpolate.interp1d`

Continued on next page

Table 13.30 – continued from previous page

mirrored([axis])	Return a copy with data mirrored in around specified axis
plot(**kwargs)	Plot a 2D colormap of `Sweep.data`
save_txt(filename)	Save text file with 3 columns: x, y, data.

__getitem__(*item*)
Same rules as numpy indexing

colorbar(***kwargs*)
Draw a colorbar with the label of `Sweep.data`

cropped(*x=None*, *y=None*)
Return a copy with data cropped to the limits in the x and/or y axes

A call with x=[-1, 2] will leave data only where -1 <= x <= 2.

Parameters **x, y** : Tuple[float, float]

Min and max data limit.

Returns *Sweep*

interpolated(*mul=None*, *size=None*, *kind='linear'*)
Return a copy with interpolate data using `scipy.interpolate.interp1d`

Call with `mul=2` to double the size of the x-axis and interpolate data to match. To interpolate in both axes pass a tuple, e.g. `mul=(4, 2)`.

Parameters **mul** : Union[int, Tuple[int, int]]

Number of times the size of the axes should be multiplied.

size : Union[int, Tuple[int, int]]

New size of the axes. Zero will leave size unchanged.

kind

Forwarded to `scipy.interpolate.interp1d`.

Returns *Sweep*

mirrored(*axis='x'*)
Return a copy with data mirrored in around specified axis

Only makes sense if the axis starts at 0.

Parameters **axis** : 'x' or 'y'

Returns *Sweep*

plot(***kwargs*)
Plot a 2D colormap of `Sweep.data`

Parameters ****kwargs**

Forwarded to `matplotlib.pyplot.pcolormesh()`.

save_txt(*filename*)
Save text file with 3 columns: x, y, data.

Parameters **filename** : str

NDSweep

class NDSweep(*variables*, *data*, *labels=None*, *tags=None*)
ND parameter sweep

Attributes

variables	(tuple of array_like) The parameters being swept.
data	(np.ndarray) Main result array with `shape == [len(v) for v in variables]`.
labels	(dict) Plot labels: 'title', 'x', 'y' and 'data'.
tags	(dict) Any additional user defined variables.

Components

The following submodules contain classes and functions which are not meant to created manually, but they are components of other classes (e.g. *Model*) so they are used regularly (even if indirectly).

system	Structural information and utilities
leads	Lead interface for scattering models

system

Structural information and utilities

Classes

Sites(positions[, ids])	Reference implementation of `AbstractSites`
System(impl: _pybinding.System)	Structural data of a tight-binding model

Functions

plot_hoppings(positions, hoppings[, width, ...])	Plot lines between lattice sites at `positions` based on the `hoppings` matrix
plot_periodic_boundaries(positions, ...[, ...])	Plot the periodic boundaries of a system
plot_sites(positions, data[, radius, ...])	Plot circles at lattice site `positions` with colors based on `data`
structure_plot_properties([axes, site, ...])	Process structure plot properties

class **Sites** (*positions*, *ids=None*)

Reference implementation of `AbstractSites`

argsort_nearest (*target_position*, *target_site_family=None*)

Return an ndarray of site indices, sorted by distance from the target

Parameters **target_position** : array_like

target_site_family : int

Look for a specific sublattice site. By default any will do.

Returns np.ndarray

Examples

```
>>> sites = Sites(([0, 1, 1.1], [0, 0, 0], [0, 0, 0]), [0, 1, 0])
>>> np.all(sites.argsort_nearest([1, 0, 0]) == [1, 2, 0])
True
```

```
>>> np.all(sites.argsort_nearest([1, 0, 0], target_site_family=0) == [2, 0,
↪ 1])
True
```

distances(*target_position*)
Return the distances of all sites from the target position

Parameters **target_position** : array_like

Examples

```
>>> sites = Sites(([0, 1, 1.1], [0, 0, 0], [0, 0, 0]), [0, 1, 0])
>>> np.allclose(sites.distances([1, 0, 0]), [1, 0, 0.1])
True
```

find_nearest(*target_position*, *target_site_family=''*)
Return the index of the position nearest the target

Parameters **target_position** : array_like

target_site_family : Optional[str]

Look for a specific sublattice site. By default any will do.

Returns int

Examples

```
>>> sites = Sites(([0, 1, 1.1], [0, 0, 0], [0, 0, 0]), [0, 1, 0])
>>> sites.find_nearest([1, 0, 0])
1
>>> sites.find_nearest([1, 0, 0], target_site_family=0)
2
```

positions
Named tuple of x, y, z positions

size
Total number of sites

xyz
Return a new array with shape=(N, 3). Convenient, but slow for big systems.

class **System**(*impl: _pybinding.System*)
Structural data of a tight-binding model

Stores positions, sublattice and hopping IDs for all lattice sites.

__getitem__(*idx*)
Same rules as numpy indexing

cropped(***limits*)
Return a copy which retains only the sites within the given limits

Parameters ****limits**

Attribute names and corresponding limits. See example.

Examples

Leave only the data where -10 <= x < 10 and 2 <= y < 4:

```
new = original.cropped(x=[-10, 10], y=[2, 4])
```

find_nearest (*position*, *sublattice=''*)
Find the index of the atom closest to the given position

Parameters **position** : array_like

Where to look.

sublattice : Optional[str]

Look for a specific sublattice site. By default any will do.

Returns int

plot (*num_periods=1*, ***kwargs*)
Plot the structure: sites, hoppings and periodic boundaries (if any)

Parameters **num_periods** : int

Number of times to repeat the periodic boundaries.

****kwargs**

Additional plot arguments as specified in *structure_plot_properties()*.

reduce_orbitals (*data*)
Sum up the contributions of individual orbitals in the given data

Takes a 1D array of `hamiltonian_size` and returns a 1D array of `num_sites` size where the multiple orbital data has been reduced per site.

Parameters **data** : array_like

Must be 1D and the equal to the size of the Hamiltonian matrix

Returns array_like

to_hamiltonian_indices (*system_idx*)
Translate the given system index into its corresponding Hamiltonian indices

System indices are always scalars and index a single (x, y, z) site position. For single-orbital models there is a 1:1 correspondence between system and Hamiltonian indices. However, for multi-orbital models the Hamiltonian indices are 1D arrays with a size corresponding to the number of orbitals on the target site.

Parameters **system_idx** : int

Returns array_like

with_data (*data*) → pybinding.results.StructureMap
Map some data to this system

boundaries
Boundary hoppings between different translation units (only for infinite systems)

expanded_positions
`positions` expanded to `hamiltonian_size` by replicating for each orbital

hamiltonian_size
The size of the Hamiltonian matrix constructed from this system

Takes into account the number of orbitals/spins at each lattice site which makes `hamiltonian_size` >= `num_sites`.

hoppings
Sparse matrix of hopping IDs

lattice
Lattice specification

num_sites
Total number of lattice sites

positions
Lattice site positions. Named tuple with x, y, z fields, each a 1D array.

sub
1D array of sublattices IDs, short for *.sublattices*

sublattices
1D array of sublattices IDs

x
1D array of coordinates, short for `.positions.x`

xyz
Return a new array with shape=(N, 3). Convenient, but slow for big systems.

y
1D array of coordinates, short for `.positions.y`

z
1D array of coordinates, short for `.positions.z`

plot_hoppings(*positions*, *hoppings*, *width=1.0*, *offset=(0, 0, 0)*, *blend=1.0*, *color='#666666'*, *axes='xyz'*, *boundary=()*, *draw_only=()*, ***kwargs*)
Plot lines between lattice sites at `positions` based on the `hoppings` matrix

Parameters **positions** : Tuple[array_like, array_like, array_like]

Site coordinates in the form of an (x, y, z) tuple of 1D arrays.

hoppings : `coo_matrix`

Sparse matrix with the hopping data, usually *System.hoppings*. The `row` and `col` indices of the sparse matrix are used to draw lines between lattice sites, while `data` determines the color.

width : float

Width of the hopping plot lines.

offset : Tuple[float, float, float]

Offset all positions by a constant value.

blend : float

Blend all colors to white (fake alpha blending): expected values between 0 and 1.

axes : str

The spatial axes to plot. E.g. 'xy', 'yz', etc.

color : str

Set the same color for all hopping lines. To assign a different color for each hopping ID, use the `cmap` parameter.

boundary : Tuple[int, array_like]

If given, apply the boundary (sign, shift).

draw_only : Iterable[str]

Only draw lines for the hoppings named in this list.

****kwargs**

Forwarded to `matplotlib.collections.LineCollection`.

Returns `matplotlib.collections.LineCollection`

plot_periodic_boundaries(*positions*, *hoppings*, *boundaries*, *data*, *num_periods=1*, ***kwargs*)

Plot the periodic boundaries of a system

Parameters **positions** : Tuple[array_like, array_like, array_like]

Site coordinates in the form of an (x, y, z) tuple of 1D arrays.

hoppings : `coo_matrix`

Sparse matrix with the hopping data, usually *`System.hoppings()`*. The `row` and `col` indices of the sparse matrix are used to draw lines between lattice sites, while `data` determines the color.

boundaries : List[Boundary]

Periodic boundaries of a *`System`*.

data : array_like

Color data at each site. Should be a 1D array of the same size as `positions`.

num_periods : int

Number of times to repeat the periodic boundaries.

****kwargs**

Additional plot arguments as specified in *`structure_plot_properties()`*.

plot_sites(*positions*, *data*, *radius=0.025*, *offset=(0, 0, 0)*, *blend=1.0*, *cmap='auto'*, *axes='xyz'*, ***kwargs*)

Plot circles at lattice site `positions` with colors based on `data`

Parameters **positions** : Tuple[array_like, array_like, array_like]

Site coordinates in the form of an (x, y, z) tuple of 1D arrays.

data : array_like

Color data at each site. Should be a 1D array of the same size as `positions`. If the data is discrete with few unique values, the discrete `colors` parameter should be used. For continuous data, setting a `cmap` (colormap) is preferred.

radius : Union[float, array_like]

Radius (in data units) of the plotted circles representing lattice sites. Should be a scalar value or an array with the same size as `positions`.

offset : Tuple[float, float, float]

Offset all positions by a constant value.

blend : float

Blend all colors to white (fake alpha blending): expected values between 0 and 1.

cmap : Union[str, List[str]]

Either a regular matplotlib colormap or a list of discrete colors to apply to the drawn circles. In the latter case, it is assumed that `data` is discrete with only a few unique values. For example, sublattice data for graphene will only contain two unique values for the A and B sublattices which will be assigned the first two colors from the `cmap` list. For continuous data, a regular matplotlib colormap should be used instead.

axes : str

The spatial axes to plot. E.g. 'xy', 'yz', etc.

****kwargs**

Forwarded to `matplotlib.collections.CircleCollection`.

Returns matplotlib.collections.CircleCollection

structure_plot_properties(*axes='xyz'*, *site=None*, *hopping=None*, *boundary=None*, ***kwargs*)

Process structure plot properties

Parameters **axes** : str

The spatial axes to plot. E.g. 'xy' for the default view, or 'yz', 'xz' and similar to plot a rotated view.

site : dict

Arguments forwarded to *plot_sites()*.

hopping : dict

Arguments forwarded to *plot_hoppings()*.

boundary : dict

Arguments forwarded to *plot_periodic_boundaries()*.

****kwargs**

Additional args are reserved for internal implementation.

Returns dict

leads

Lead interface for scattering models

The only way to create leads is using the *Model.attach_lead()* method. The classes represented here are the final product of that process, listed in *Model.leads*.

class Lead(*impl: _pybinding.Lead*, *index*)

Describes a single lead connected to a *Model*

Leads can only be created using *Model.attach_lead()* and accessed using *Model.leads*.

calc_bands(*start=-3.141592653589793*, *end=3.141592653589793*, *step=0.05*)

Calculate the band structure of an infinite lead

Parameters **start, end** : float

Points in reciprocal space which form the path for the band calculation.

step : float

Calculation step length in reciprocal space units. Lower `step` values will return more detailed results.

Returns Bands

plot(*lead_length=6*, ***kwargs*)

Plot the sites, hoppings and periodic boundaries of the lead

Parameters **lead_length** : int

Number of times to repeat the lead's periodic boundaries.

****kwargs**

Additional plot arguments as specified in *structure_plot_properties()*.

plot_bands(*start=-3.141592653589793*, *end=3.141592653589793*, *step=0.05*, ***kwargs*)

Plot the band structure of an infinite lead

Parameters **start, end** : float

Points in reciprocal space which form the path for the band calculation.

step : float

Calculation step length in reciprocal space units. Lower `step` values will return more detailed results.

****kwargs**

Forwarded to *`Bands.plot()`*.

`plot_contact`(*line_width=1.6*, *arrow_length=0.5*, *shade_width=0.3*, *shade_color='#d40a0c'*)

Plot the shape and direction of the lead contact region

Parameters **line_width** : float

Width of the line representing the lead contact.

arrow_length : float

Size of the direction arrow as a fraction of the contact line length.

shade_width : float

Width of the shaded area as a fraction of the arrow length.

shade_color : str

Color of the shaded area.

`h0`

Unit cell Hamiltonian as `csr_matrix`

`h1`

Hamiltonian which connects who unit cells, `csr_matrix`

`indices`

Main system indices (1d array) to which this lead is connected

`system`

Structural information, see *`System`*

Miscellaneous

`constants`	A few useful physical constants
`pltutils`	Collection of utility functions for matplotlib

constants

A few useful physical constants

Note that energy is expressed in units of eV.

`c` = 299792458

[m/s] speed of light

`e` = 1.602e-19

[C] electron charge

`epsilon0` = 8.854e-12

[F/m] vacuum permittivity

`hbar` = 6.582118989999999e-16

[eV*s] reduced Plank constant

`pauli` = x: [[0, 1], [1, 0]], y: [[0, -1j], [1j, 0]], z: [[1, 0], [0, -1]]

Pauli matrices – use the `.x`, `.y` and `.z` attributes

phi0 = 4.1356673328075734e-15
[V*s] magnetic quantum

pltutils

Collection of utility functions for matplotlib

Functions

cm2inch(*values)	Convert from centimeter to inch
colorbar([mappable, cax, ax, label, powerlimits])	Custom colorbar with modified style and optional label
despine([trim])	Remove the top and right spines
despine_all()	Remove all spines, axes labels and ticks
get_palette([name, num_colors, start])	Get a color palette from matplotlib's colormap database
legend(*args[, reverse, facecolor, lw])	Custom legend with modified style and option to reverse label order
respine()	Redraw all spines, opposite of *despine()*
set_palette([name, num_colors, start])	Set the active color palette
use_style([style])	Shortcut for `matplotlib.style.use()` with pybinding style applied by default

cm2inch(**values*)
Convert from centimeter to inch

Parameters ***values**

Returns tuple

Examples

```
>>> cm2inch(2.54, 5.08)
(1.0, 2.0)
```

colorbar(*mappable=None*, *cax=None*, *ax=None*, *label=''*, *powerlimits=(0, 0)*, ***kwargs*)
Custom colorbar with modified style and optional label

Changes default `pad` and `aspect` argument values and turns on rasterization for a nicer looking colorbar with smaller size in vector formats (pdf, svg).

Parameters **label** : str

Color data label.

powerlimits : Tuple[int, int]

Sets size thresholds for scientific notation.

mappable, cax, ax, **kwargs

Forwarded to `matplotlib.pyplot.colorbar()`.

despine(*trim=False*)
Remove the top and right spines

Parameters **trim** : bool

Trim spines so that they don't extend beyond the last major ticks.

despine_all()
Remove all spines, axes labels and ticks

get_palette(*name=None*, *num_colors=8*, *start=0*)
Get a color palette from matplotlib's colormap database

Parameters **name** : str, optional

Name of the palette to get. If `None`, get the active palette.

num_colors : int

Number of colors to retrieve.

start : int

Staring from this color number.

Returns List[color]

legend(**args*, *reverse=False*, *facecolor='0.98'*, *lw=0*, ***kwargs*)
Custom legend with modified style and option to reverse label order

Parameters **reverse** : bool

Reverse the label order.

facecolor : color

Legend background color.

lw : float

Frame width.

***args, **kwargs**

Forwarded to `matplotlib.pyplot.legend()`.

respine()
Redraw all spines, opposite of *despine()*

set_palette(*name=None*, *num_colors=8*, *start=0*)
Set the active color palette

Parameters **name** : str, optional

Name of the palette. If `None`, modify the active palette.

num_colors : int

Number of colors to retrieve.

start : int

Staring from this color number.

use_style(*style=pb_style*)
Shortcut for `matplotlib.style.use()` with pybinding style applied by default

Parameters **style** : dict

A matplotlib style specification.

CHAPTER 14

Experimental

This section documents some of the experimental features of pybinding. They may be incomplete or require additional work like compiling the code manually. Proceed with caution.

CUDA-based KPM

CUDA enables the execution of general purpose code on Nvidia GPUs. It can be used to accelerate computational algorithms which feature natural parallelism. Pybinding features experimental support for CUDA. It's used for kernel polynomial method (KPM) calculations – see *tutorial page* and *API reference*.

The CUDA-base KPM implementation is available via the *kpm_cuda()* function. It mirrors the API of the regular CPU-based *kpm()*. The only difference between them is where the calculation will take place. Note that the CUDA implementation is still experimental and that only diagonal Green's function elements will be computed on the GPU, while off-diagonal falls back to regular CPU code. This will be addressed in a future version.

By default, CUDA support is disabled. You will need to turn it on manually by recompiling the package. First, ensure that you have CUDA Toolkit 7.5 or newer installed. Next, remove any existing pybinding installation by executing the following command in terminal:

```
pip3 uninstall pybinding
```

Finally, reinstall it with CUDA turned on:

```
PB_CUDA=ON pip3 install pybinding --no-binary pybinding
```

Note that `pybinding` is written twice. This is not a mistake. The `--no-binary pybinding` flag tells pip to compile from source. Since this is all experimental: expect errors and no support.

FEAST eigensolver

The FEAST eigensolver significantly differs from traditional solvers like the ones found in LAPACK and ARPACK. It takes its inspiration from the density-matrix representation and contour integration in quantum mechanics. When solving a series of eigenvalue problems which are close to one another, as is the case for band structure calculations, the results of the previous calculation can be used as the starting point for the next. The algorithm also features natural parallelism where different eigenvalues can be computed separately without overlap.

Pybinding has experimental support for this solver. It can be accessed via *solver.feast()*. However, it is disabled by default and you will need to recompile the package in order to install it. Since FEAST requires Intel PARDISO, you will need to have Intel MKL installed before you continue. Next, remove any existing pybinding installation by executing the following command in terminal:

```
pip3 uninstall pybinding
```

Finally, reinstall it with MKL turned on:

```
PB_MKL=ON pip3 install pybinding --no-binary pybinding
```

Note that `pybinding` is written twice. This is not a mistake. The `--no-binary pybinding` flag tells pip to compile from source. Since this is all experimental: expect errors and no support.

Hopping generator

The *@hopping_generator* can be used to create new hoppings independent of the main lattice definition. It's especially useful for creating additional local hoppings, e.g. to model defects. Here, we present a way create twisted bilayer graphene with an arbitrary rotation angle θ.

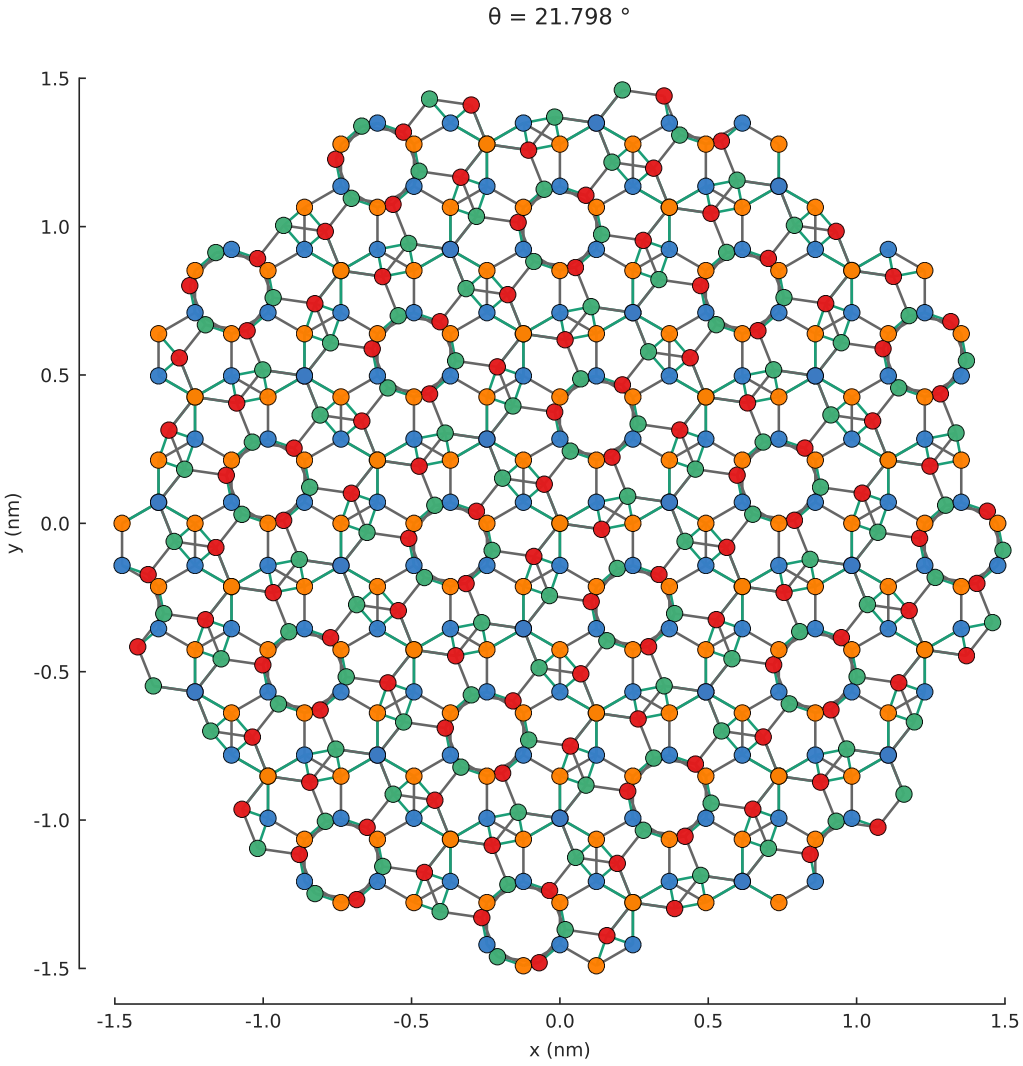
We start with two unconnected layers of graphene. A *@site_position_modifier* is applied to rotate just one layer. Then, a *@hopping_generator* finds and connects the layers via site pairs which satisfy the given criteria. The newly created hoppings all have identical energy at first. Finally, a *@hopping_energy_modifier* to applied to set the new interlayer hopping energy to the desired distance-dependent value.

This is an experimental feature, presented as is, without any additional support.

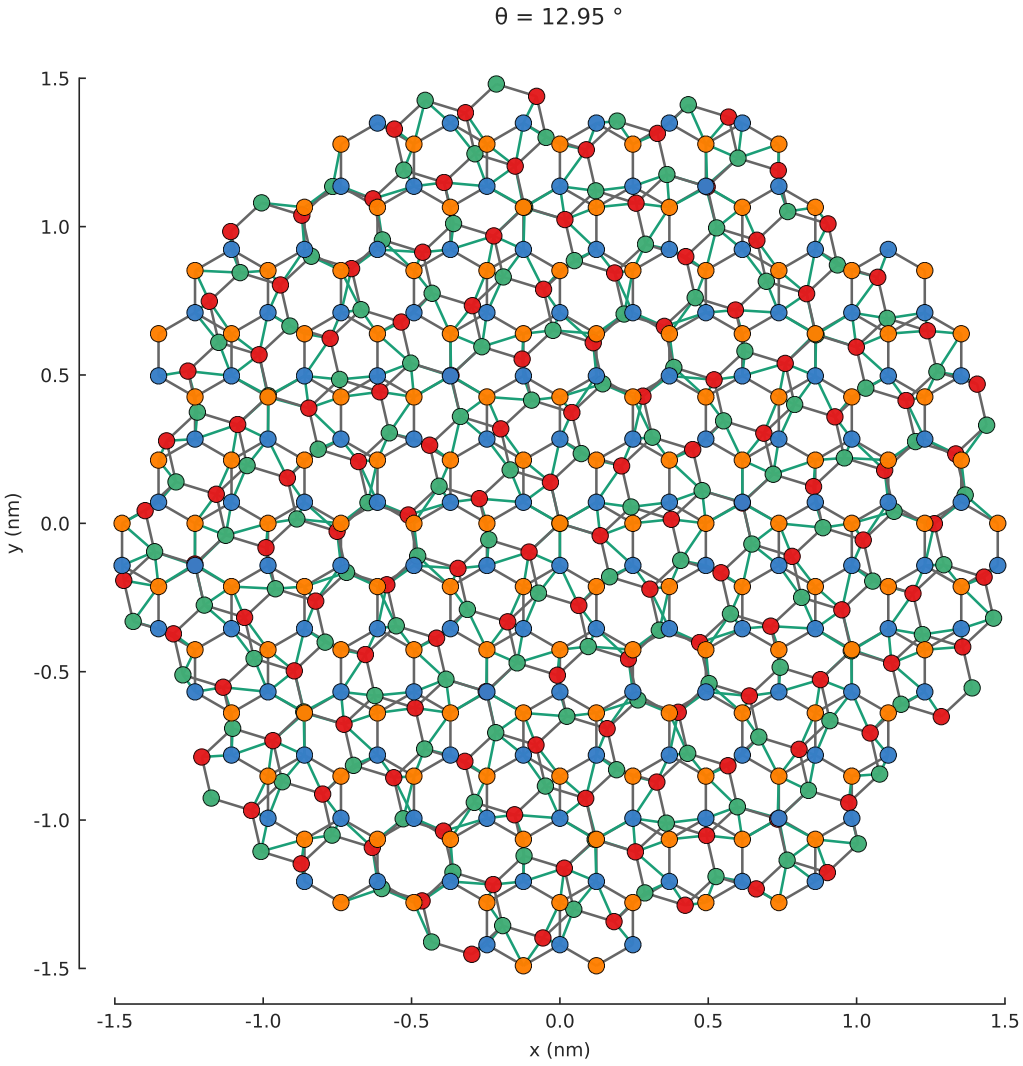
Source code

```
"""Construct a circular flake of twisted bilayer graphene (arbitrary angle)"""
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import cKDTree

import pybinding as pb

c0 = 0.335  # [nm] graphene interlayer spacing


def two_graphene_monolayers():
    """Two individual layers of monolayer graphene without any interlayer hopping"
↪"
    from pybinding.repository.graphene.constants import a_cc, a, t

    lat = pb.Lattice(a1=[a/2, a/2 * math.sqrt(3)], a2=[-a/2, a/2 * math.sqrt(3)])
    lat.add_sublattices(('A1', [0,  -a_cc,   0]),
                        ('B1', [0,      0,   0]),
                        ('A2', [0,      0, -c0]),
                        ('B2', [0,   a_cc, -c0]))
    lat.register_hopping_energies({'gamma0': t})
    lat.add_hoppings(
        # layer 1
        ([ 0,  0], 'A1', 'B1', 'gamma0'),
        ([ 0, -1], 'A1', 'B1', 'gamma0'),
        ([-1,  0], 'A1', 'B1', 'gamma0'),
        # layer 2
        ([ 0,  0], 'A2', 'B2', 'gamma0'),
        ([ 0, -1], 'A2', 'B2', 'gamma0'),
        ([-1,  0], 'A2', 'B2', 'gamma0'),
```

```
        # not interlayer hopping
    )
    lat.min_neighbors = 2
    return lat


def twist_layers(theta):
    """Rotate one layer and then a generate hopping between the rotated layers"""
    theta = theta / 180 * math.pi  # from degrees to radians

    @pb.site_position_modifier
    def rotate(x, y, z):
        """Rotate layer 2 by the given angle `theta`"""
        layer2 = (z < 0)
        x0 = x[layer2]
        y0 = y[layer2]
        x[layer2] = x0 * math.cos(theta) - y0 * math.sin(theta)
        y[layer2] = y0 * math.cos(theta) + x0 * math.sin(theta)
        return x, y, z

    @pb.hopping_generator('interlayer', energy=0.1)  # eV
    def interlayer_generator(x, y, z):
        """Generate hoppings for site pairs which have distance `d_min < d < d_max`
↪"""
        positions = np.stack([x, y, z], axis=1)
        layer1 = (z == 0)
        layer2 = (z != 0)

        d_min = c0 * 0.98
        d_max = c0 * 1.1
        kdtree1 = cKDTree(positions[layer1])
        kdtree2 = cKDTree(positions[layer2])
        coo = kdtree1.sparse_distance_matrix(kdtree2, d_max, output_type='coo_
↪matrix')

        idx = coo.data > d_min
        abs_idx1 = np.flatnonzero(layer1)
        abs_idx2 = np.flatnonzero(layer2)
        row, col = abs_idx1[coo.row[idx]], abs_idx2[coo.col[idx]]
        return row, col  # lists of site indices to connect

    @pb.hopping_energy_modifier
    def interlayer_hopping_value(energy, x1, y1, z1, x2, y2, z2, hop_id):
        """Set the value of the newly generated hoppings as a function of distance"
↪"
        d = np.sqrt((x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2)
        interlayer = (hop_id == 'interlayer')
        energy[interlayer] = 0.4 * c0 / d[interlayer]
        return energy

    return rotate, interlayer_generator, interlayer_hopping_value


model = pb.Model(
    two_graphene_monolayers(),
    pb.circle(radius=1.5),
    twist_layers(theta=21.798)
)
plt.figure(figsize=(6.5, 6.5))
model.plot(hopping=dict(width=1.6, cmap='auto'))
plt.title(r"$\theta$ = 21.798 $\degree$")
plt.show()
```




```
model = pb.Model(
    two_graphene_monolayers(),
    pb.circle(radius=1.5),
    twist_layers(theta=12.95)
)
plt.figure(figsize=(6.5, 6.5))
model.plot(hopping=dict(width=1.6, cmap='auto'))
plt.title(r"$\theta$ = 12.95 $\degree$")
plt.show()
```



- genindex

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z