Kwant 1.1.1 documentation

C. W. Groth, M. Wimmer, A. R. Akhmerov, X. Waintal, et al.

October 21, 2015

CHAPTER

ONE

PRELIMINARIES

1.1 About Kwant

Kwant is a free (open source) Python package for numerical calculations on tight-binding models with a strong focus on quantum transport. It is designed to be flexible and easy to use. Thanks to the use of innovative algorithms, Kwant is often faster than other available codes, even those entirely written in the low level FORTRAN and C/C++ languages.

Tight-binding models can describe a vast variety of systems and phenomena in quantum physics. Therefore, Kwant can be used to simulate

- metals,
- graphene,
- topological insulators,
- quantum Hall effect,
- superconductivity,
- spintronics,
- molecular electronics,
- any combination of the above, and many other things.

Kwant can calculate

- transport properties (conductance, noise, scattering matrix),
- dispersion relations,
- modes,
- wave functions,
- various Green's functions,
- out-of-equilibrium local quantities.

Other computations involving tight-binding Hamiltonians can be implemented easily.

1.2 What's new in Kwant

1.2.1 What's new in Kwant 1.1

This article explains the user-visible changes in Kwant 1.1, released on 21 October 2015. Please consult the full list of changes in Kwant for all the changes up to the most recent bugfix release.

Harmonize `Bands` with `modes`

Kwant's convention is that momenta are positive in the direction of `TranslationalSymmetry`. While the momenta returned by `modes` did respect this convention, the momenta read off the band structure as given by `Bands` had the wrong sign. This has been fixed now.

New option `add_cells` of `attach_lead`

Before actually attaching a lead to a builder, the method `attach_lead` of `Builder` prepares a "nice" interface by adding "missing" sites such that the first unit cell of the lead is completely connected with the system under construction. These sites and their hoppings are taken over from the lead.

By setting the new option `add_cells`, `attach_lead` can now be told to add *in addition* any number of complete unit cells of the lead to the system before attaching it. Among other things, this can be useful for

- controlling the hopping between the lead and the system (Leads are always attached with their inter-unit-cell hopping to the system, but absorbing one lead unit cell into the system allows to control this),
- creating a buffer for long range disorder present in the system to die away before the translation-invariant lead begins.

To support these applications, `attach_lead` now returns a list of all the sites that have been added to the system. Creating a buffer for disorder can be thus done as follows:

```
sys[sys.attach_lead(lead, add_cells=10)] = onsite
```

Note how we set the onsite Hamiltonians of the sites that have been added to the value used in the system.

New method `conductance_matrix`

`SMatrix` and `GreensFunction` have each gained a method `conductance_matrix` that returns the matrix G such that $I = GV$ where I and V are, respectively, the vectors of currents and voltages for all the leads. This matrix is useful for calculating non-local resistances. See Section 2.4 of the book by S. Datta.

Deduction of transmission probabilities

If `kwant.smatrix` or `kwant.greens_function` have been called with `check_hermicity=True` (on by default) and a restricted number of leads in the `out_leads` and `in_leads` parameters, calls to `transmission` and `conductance_matrix` will work whenever it is possible to deduce the result from current conservation.

This allows leaving out one lead (preferably the widest) from `out_leads` and `in_leads`, and still to calculate all transmission probabilities. Doing so has been measured to speed up computations by 20% in some cases.

Clearer error messages

The error messages (exceptions) that appear when the Kwant interface is used incorrectly have been improved in many cases. For example, if instead of

```
builder[lattice(0, 1)] = 1
```

one writes

```
builder[(0, 1)] = 1
```

the error message will be more helpful now.

Please continue reporting confusing error messages on the Kwant mailing list.

New option `pos_transform` of `map`

This option which already existed for `kwant.plotter.plot` is now also available for `kwant.plotter.map`.

1.2.2 What's new in Kwant 1.0

This article explains the new features in Kwant 1.0 compared to Kwant 0.2. Kwant 1.0 was released on 9 September 2013. Please consult the full list of changes in Kwant for all the changes up to the most recent bugfix release.

Lattice and shape improvements

Lattices now have a method `neighbors`, which calculates all the n-th shortest possible hoppings on this lattice. This replaces the `nearest` attribute that some lattices used to have.

`shape` uses an improved flood-fill algorithm, making it work better on narrow ribbons (which were sometimes buggy before with non-square lattices). Additionally, it was made symmetry-aware: If `shape` is used with a lead, the shape does not have to be limited along the lead direction anymore. In fact, if the shape function does not have the same symmetry as the lead, the result may be unexpected, so it is highly recommended to use shape functions that have the same symmetry as the lead.

`closest` now returns an exact, and not approximate closest point. A new method `n_closest` was added, which returns the n closest lattice points.

`possible_hoppings` replaced by `HoppingKind`

The `Builder` method `possible_hoppings` has been rendered obsolete. Where previously one would have had

```
for kind in lat.nearest:
    sys[sys.possible_hoppings(*kind)] = t
```

now it suffices to write

```
sys[lat.neighbors()] = t
```

This is possible because `Builder` now accepts *functions* as keys in addition to `Site` objects and tuples of them (hoppings). These functions are expected to yield either sites or hoppings, when given a builder instance as the sole argument. The use of such keys is to implement sets of sites or hoppings that depend on what is already present in the builder, such as `HoppingKind`. In the above example, `lat.neighbors()` is a list of `HoppingKind` objects.

Some renames

- site groups are now called site families. This affects all the names that used to contain "group" or "groups".
- lead slices are now referred to as lead cells: This affects all names that used to contain "slice" or "slices" in the context of leads.
- `self_energy` has been renamed to `selfenergy` in all cases, most notably in `kwant.physics.selfenergy`.
- `wave_func` has been renamed to `wave_function`,
- `MonatomicLattice` has been renamed to `Monatomic`,
- `PolyatomicLattice` has been renamed to `Polyatomic`.
- `solve` was split into two functions: `smatrix`, and `greens_function`. The former calculates the scattering matrix, the latter the retarded Green's function between the sites adjacent to the leads. It is temporarily not possible to mix self-energy and modes leads within the same system.

- The object that contained the results, *BlockResult* was also split into `SMatrix` and `GreensFunction`.

Band structure plots

A convenience function `bands` for quick plotting of band structure was implemented.

Immutable site families

In order to make naming more consistent, *kwant.make_lattice* was renamed and can be found now as `general`. Classes `Chain`, `Square`, and `Honeycomb` from `lattice` were made functions `chain`, `square`, and `honeycomb`.

In previous versions if one executed `a = kwant.lattice.square(); b = kwant.lattice.square()` then `a` and `b` were actually different lattices. This often led to confusions in more convoluted use cases, so this behavior was changed. Now two site families created with the same parameters are actually indistinguishable by Kwant. If it is desired to make two site families which have the same geometry, but mean different things, as for instance in *Superconductors: orbital vs. lattice degrees of freedom*, then the `name` argument has to be used when creating a lattice, e.g. `a = kwant.lattice.square(name='a'); b = kwant.lattice.square(name='b')`.

Parameters to Hamiltonian

Kwant now allows the Hamiltonian matrix elements to be described with functions that depend on an arbitrary number of parameters in addition to the sites on which they are defined.

Previously, functions defining the Hamiltonian matrix elements had to have the following prototypes:

```
def onsite(site):
    ...

def hopping(site1, site2):
    ...
```

If the Hamiltonian elements need to depend on some other external parameters (e.g. magnetic field) then those had to be provided by some other means than regular function parameters (e.g. global variables).

Now the value functions may accept arbitrary arguments after the *Site* arguments. These extra arguments can be specified when `smatrix` is called by setting the arguments:

args A tuple of values to be passed as the positional arguments to the Hamiltonian value functions (not including the *Site* arguments).

For example, if the hopping and onsite Hamiltonian value functions have the following prototype:

```
def onsite(site, t, B, pot):
    ...

def hopping(site1, site2, t, B, pot):
    ...
```

then the values of `t`, `B` and `pot` for which to solve the system can be passed to `smatrix` like this:

```
kwant.smatrix(sys, energy,
              args=(2., 3., 4.))
```

With many parameters it can be less error-prone to collect all of them into a single object and pass this object as the single argument. Such a parameter collection could be a dictionary, or a class instance, for example:

```
class SimpleNamespace(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
# With Python >= 3.3 we can have instead:
```

```
# from types import SimpleNamespace

def onsite(site, p):
    return p.mu * ...

def hopping(site1, site2, p):
    return p.t * exp(-1j * p.B * ...)

params = SimpleNamespace(t=1, mu=2)
for params.B in B_values:
    kwant.smatrix(sys, energy, args=[params])
```

Arguments can be passed in an equivalent way to `wave_function`, `hamiltonian_submatrix`, etc.

Calculation of modes separated from solving

The interface that solvers expect from leads attached to a `FiniteSystem` has been simplified and codified (see there). Similar to self-energy, calculation of modes is now the lead's own responsibility.

The new class `ModesLead` allows to attach leads that have a custom way of calculating their modes (e.g. ideal leads) directly to a `Builder`.

Modes or self-energies can now be precomputed before passing the system to a solver, using the method `precalculate`. This may save time, when the linear system has to be solved many times with the same lead parameters.

Change of the modes and lead_info format

The function `modes` now returns two objects: `PropagatingModes` and `StabilizedModes`. The first one contains the wave functions of all the propagating modes in real space, as well as their velocities and momenta. All these quantities were previously not directly available. The second object contains the propagating and evanescent modes in the compressed format expected by the sparse solver (previously this was the sole output of `modes`). Accordingly, the `lead_info` attribute of `SMatrix` contains the real space information about the modes in the leads (a list of `PropagatingModes` objects).

New module for random-access random numbers

The module `kwant.digest` provides functions that given some input compute a "random" output that depends on the input in a (cryptographically) intractable way. This functionality is useful for introducing disorder, e.g.:

```
def onsite(site):
    return 0.3 * kwant.digest.gauss(repr(site)) + 4
```

New module for random matrix theory Hamiltonians

The module `kwant.rmt` supports the creation of random matrix theory Hamiltonians.

Improved plotting functionality

The plotting functionality has been extended. By default, symbols and lines in plots are now relative to the system coordinates, i.e. will scale accordingly if different zoom-levels are used. Different styles for representing sites and hoppings are now possible. 3D plotting has been made more efficient.

1.2.3 What's new in Kwant 0.2

This article explains the user-visible changes in Kwant 0.2. Kwant 0.2 was released on 29 November 2012.

Improved performance

This has been the main focus of this release. Through optimization a level of performance has been reached that we consider satisfactory: runs of Kwant for mid-sized (100x100 say) systems now typically spend most time in highly optimized libraries and not anymore in Python-implemented code. For large, truly performance-critical systems almost all time is now spent in optimized libraries.

An important optimization has been replacing NumPy for most uses within Kwant by tinyarray. tinyarray provides a subset of NumPy's functionality in a way that is highly optimized for small arrays such as the tags of sites in Kwant.

New MUMPS-based solver

The code for sparse matrix solvers has been reorganized and a new solver has been added next to *kwant.solvers.sparse*: `kwant.solvers.mumps`. The new solver uses the MUMPS software package and is much (typically several times) faster than the UMFPACK-based old solver. In addition, MUMPS uses considerably less memory for a given system while at the same time it is able to take advantage of more than 2 GiB of RAM.

New tutorial dealing with superconductivity

Superconductors: orbital vs. lattice degrees of freedom

New `plotter` module

`plotter` has been rewritten using matplotlib, which allows plot post-processing, basic 3D plotting and many other features. Due to the possibility to easily modify a matplotlib plot after it has been generated, function `plot` has much fewer input parameters, and is less flexible than its previous implementation. Its interface is also much more similar to that of matplotlib. For the detailed interface and input description check `plot` documentation.

The behavior of `plot` with low level systems has changed. Arguments of plot which are functions are given site numbers in place of `Site` objects when plotting a low level system. This provides an easy way to make the appearance of lines and symbols depend on computation results.

A new function `map` was implemented. It allows to show a map of spatial dependence of a function of a system site (e.g. density of states) without showing the sites themselves.

`TranslationalSymmetry` is used differently

When constructing an instance of `TranslationalSymmetry` a sole parameter used to be expected: A sequence of sequences of 1d real space vectors. Now `TranslationalSymmetry` can take an arbitrary number of parameters, each of them a 1d real space vector. This reduced the number of parantheses necessary in the common case where there is just a single parameter

Example of old usage:

```
sym = kwant.TranslationalSymmetry([(-1, 0)])
```

New usage:

```
sym = kwant.TranslationalSymmetry((-1, 0))
```

Band structure functionality has been moved

The functionality that used to be provided by the method `energies` of `kwant.system.InfiniteSystem` has been moved to the `kwant.physics` package. See the documentation of `kwant.physics.Bands` and *Beyond transport: Band structure and closed systems*.

Calculation of the local density of states

The new function of sparse solvers `ldos` allows the calculation of the local density of states.

Calculation of wave functions in the scattering region

(Kwant 0.3 update: `wave_func` has been renamed to `wave_function`.)

The new function of sparse solvers `wave_func` allows the calculation of the wave function in the scattering region due to any mode of any lead.

Return value of sparse solver

The function *solve* of sparse solvers now always returns a single instance of `BlockResult`. The latter has been generalized to include more information for leads defined as infinite systems.

1.3 Installation instructions

Kwant can be installed either using prepared packages (Debian, Ubuntu, and Arch variants of GNU/Linux, Mac OS X, Microsoft Windows), or it can be built and installed from source.

In general, installation from packages is advisable, especially for novice users. Expert users may find it helpful to build Kwant from source, as this will also allow them to customize Kwant to use certain optimized versions of libraries.

1.3.1 Installing from packages

Debian and derivatives

The easiest way to install Kwant on a Debian system is using the pre-built packages we provide. Our packages are known to work with Debian "wheezy" and Debian "jessie", but they may also work on many other recent Debian-derived sytems as well. (For example, the following works with recent Ubuntu versions.)

The lines prefixed with `sudo` have to be run as root.

1. Add the following lines to `/etc/apt/sources.list`:

   ```
   deb http://downloads.kwant-project.org/debian/ stable main
   deb-src http://downloads.kwant-project.org/debian/ stable main
   ```

2. (Optional) Add the OpenPGP key used to sign the repositories by executing:

   ```
   sudo apt-key adv --keyserver pool.sks-keyservers.net --recv-key C3F147F5980F3535
   ```

   The fingerprint of the key is 5229 9057 FAD7 9965 3C4F 088A C3F1 47F5 980F 3535.

3. Update the package data, and install Kwant:

   ```
   sudo apt-get update
   sudo apt-get install python-kwant python-kwant-doc
   ```

   The `python-kwant-doc` package is optional and installs the HTML documentation of Kwant in the directory `/usr/share/doc/python-kwant-doc`.

Should the last command (`apt-get install`) fail due to unresolved dependencies, you can try to build and install your own packages, which is surprisingly easy:

```
cd /tmp

sudo apt-get build-dep tinyarray
apt-get source --compile tinyarray
sudo dpkg -i python-tinyarray_*.deb

sudo apt-get build-dep kwant
apt-get source --compile kwant
sudo dpkg -i python-kwant_*.deb python-kwant-doc_*.deb
```

This method should work for virtually all Debian-derived systems, even on exotic architectures.

Ubuntu and derivatives

Execute the following commands:

```
sudo apt-add-repository ppa:kwant-project/ppa
sudo apt-get update
sudo apt-get install python-kwant python-kwant-doc
```

This should provide Kwant for all versions of Ubuntu >= 12.04. The HTML documentation will be installed locally in the directory `/usr/share/doc/python-kwant-doc`.

Arch Linux

Arch install scripts for Kwant are kindly provided by Jörg Behrmann (formerly by Max Schlemmer). To install, follow the Arch User Repository installation instructions. Note that for checking the validity of the package you need to add the key used for signing to your user's keyring via:

```
gpg --keyserver pool.sks-keyservers.net --recv-key C3F147F5980F3535
```

The fingerprint of the key is 5229 9057 FAD7 9965 3C4F 088A C3F1 47F5 980F 3535.

Mac OS X

There is a number of different package managers for bringing software from the Unix/Linux world to Mac OS X. Since the community is quite split, we provide Kwant and its dependencies both via the homebrew and the MacPorts systems.

Mac OS X: homebrew

homebrew is a recent addition to the package managers on Mac OS X. It is lightweight, tries to be as minimalistic as possible and give the user freedom than Macports. We recommend this option if you have no preferences.

1. Open a terminal and install homebrew as described on the homebrew homepage (instructions are towards the end of the page)
2. Run

   ```
   brew doctor
   ```

   and follow its directions. It will ask for a few prerequisites to be installed, in particular

- the Xcode developer tools (compiler suite for Mac OS X) from http://developer.apple.com/downloads. You will need an Apple ID to download. Note that if you have one already from using the App store on the Mac/Ipad/Iphone/... you can use that one. Downloading the command line tools (not the full Xcode suite) is sufficient. If you have the full Xcode suite installed, you might need to download the command line tools manually if you have version 4 or higher. In this case go to *Xcode->Preferences*, click on *Download*, go to *Components*, select *Command Line Tools* and click on *Install*.

- although *brew doctor* might not complain about it right away, while we're at it, you should also install the X11 server from the XQuartz project if you have Mac OS X 10.8 or higher.

3. Add permanently `/usr/local/bin` before `/usr/bin/` in the `$PATH$` environment variable of your shell, for example by adding

   ```
   export PATH=/usr/local/bin:$PATH
   ```

   at the end of your `.bash_profile` or `.profile`. Then close the terminal and reopen it again.

4. Install a few prerequisites

   ```
   brew install gfortran python
   ```

5. Add additional repositories

   ```
   brew tap homebrew/science
   brew tap samueljohn/python
   brew tap michaelwimmer/kwant
   ```

6. Install Kwant and its prerequisites

   ```
   pip install nose
   brew install numpy scipy matplotlib
   brew install kwant
   ```

Notes:

- If something does not work as expected, use `brew doctor` for instructions (it will find conflicts and things like that).
- As mentioned, homebrew allows for quite some freedom. In particular, if you are an expert, you don't need necessarily to install numpy/scipy/matplotlib from homebrew, but can use your own installation. The only prerequisite is that they are importable from python. (the Kwant installation will in any case complain if they are not)
- In principle, you need not install the homebrew python, but could use Apple's already installed python. Homebrew's python is more up-to-date, though.

Mac OS X: MacPorts

MacPorts is a full-fledged package manager that recreates a whole Linux-like environment on your Mac.

In order to install Kwant using MacPorts, you have to

1. Install a recent version of MacPorts, as explained in the installation instructions of MacPorts. In particular, as explained there, you will have to install also a few prerequisites, namely

- the Xcode developer tools (compiler suite for Mac OS X) from http://developer.apple.com/downloads. You will need an Apple ID to download. Note that if you have one already from using the App store on the Mac/Ipad/Iphone/... you can use that one. You will also need the command line tools: Within Xcode 4, you have to download them by going to *Xcode->Preferences*, click on *Download*, go to *Components*, select *Command Line Tools* and click on *Install*. Alternatively, you can also directly download the command line tools from the Apple developer website.
- if you have Mac OS X 10.8 or higher, the X11 server from the XQuartz project.

2. After the installation, open a terminal and execute

   ```
   echo http://downloads.kwant-project.org/macports/ports.tar |\
   sudo tee -a /opt/local/etc/macports/sources.conf >/dev/null
   ```

   (this adds the Kwant MacPorts download link http://downloads.kwant-project.org/macports/ports.tar at the end of the `sources.conf` file.)

3. Execute

```
sudo port selfupdate
```

4. Now, install Kwant and its prerequisites

   ```
   sudo port install py27-kwant
   ```

5. Finally, we choose python 2.7 to be the default python

   ```
   sudo port select --set python python27
   ```

   After that, you will need to close and reopen the terminal to have all changes in effect.

Notes:

- If you have problems with macports because your institution's firewall blocks macports (more precisely, the *rsync* port), resulting in errors from `sudo port selfupdate`, follow these instructions.
- Of course, if you already have macports installed, you can skip step 1 and continue with step 2.

Microsoft Windows

There are multiple distributions of scientific Python software for Windows that provide the prerequisites for Kwant. We recommend to use the packages kindly provided by Christoph Gohlke. To install Kwant on Windows

1. Determine whether you have a 32-bit or 64-bit Windows installation by following these instructions.
2. Download and install Python 2.7 for the appropriate architecture (32-bit or 64-bit) from the official Python download site.
3. Open a command prompt, as described in "How do I get a command prompt" at the Microsoft Windows website.
4. In the command prompt window, execute:

   ```
   C:\Python27\python.exe C:\Python27\Tools\Scripts\win_add2path.py
   ```

   (Instead of typing this command, you can also just copy it from here and paste it into the command prompt window). If you did not use the default location to install Python in step 2, then replace `C:\Python27` by the actual location where Python is installed.
5. Reboot your computer.
6. Download the necessary packages (with the ending `.whl`) for your operating system (32 or 64 bit) and Python version (e.g. `cp27` for Python 2.7) from the website of Christoph Gohlke. For Kwant, we recommend to download at least NumPy, SciPy, Matplotlib, Nose, Tinyarray, and Kwant itself.
7. Now open a command prompt with administrator rights, as described in "How do I run a command with elevated permissions" at the Microsoft Windows website.

   In this new command prompt window, execute

   ```
   pip install <filename>
   ```

   for each of the downloaded files (replacing `<filename>` with it).

   Now you are done, you can `import kwant` from within Python scripts.

(Note that many other userful scientific packages are available in Gohlke's repository. For example, you might want to install IPython and its various dependencies so that you can use the IPython notebook.)

1.3.2 Building and installing from source

Prerequisites

Building Kwant requires

- Python 2.6 or 2.7 (Python 3 is not supported yet),
- SciPy 0.9 or newer,
- LAPACK and BLAS, (For best performance we recommend the free OpenBLAS or the nonfree MKL.)
- Tinyarray, a NumPy-like Python package optimized for very small arrays,
- An environment which allows to compile Python extensions written in C and C++.

The following software is highly recommended though not strictly required:

- matplotlib 1.1 or newer, for Kwant's plotting module and the tutorial,
- MUMPS, a sparse linear algebra library that will in many cases speed up Kwant several times and reduce the memory footprint. (Kwant uses only the sequential, single core version of MUMPS. The advantages due to MUMPS as used by Kwant are thus independent of the number of CPU cores of the machine on which Kwant runs.)
- The nose testing framework for running the tests included with Kwant.

In addition, to build a copy of Kwant that has been checked-out directly from its Git repository, you will also need Cython 0.22 or newer. You do not need Cython to build Kwant that has been unpacked from a source .tar.gz-file.

Generic instructions

Kwant can be built and installed following the usual Python conventions by running the following commands in the root directory of the Kwant distribution.

```
pip install .
```

Depending on your system, you might have to run the second command with administrator privileges (e.g. prefixing it with `sudo`). If you use Python older than 2.7.9, see pip installation instructions.

After installation, tests can be run with:

```
python -c 'import kwant; kwant.test()'
```

The tutorial examples can be found in the directory `tutorial` inside the root directory of the Kwant source distribution.

Unix-like systems (GNU/Linux)

Kwant should run on all recent Unix-like systems. The following instructions have been verified to work on Debian 7 (Wheezy) or newer, and on Ubuntu 12.04 or newer. For other distributions step 1 will likely have to be adapted. If Ubuntu-style `sudo` is not available, the respective command must be run as root.

1. Install the required packages. On Debian-based systems like Ubuntu this can be done by running the command

   ```
   sudo apt-get install python-dev python-scipy python-matplotlib python-nose g++ gfortran libop
   ```

2. Unpack Tinyarray, enter its directory. To build and install, run

   ```
   sudo pip install .
   ```

3. Inside the Kwant source distribution's root directory run

```
sudo pip install .
```

By default the package will be installed under `/usr/local`. Type `pip help install` for installation options and see pip documentation for a detailed description of `pip`.

Mac OS X: MacPorts

The required dependencies of Kwant are best installed with one of the packaging systems. Here we only consider the case of MacPorts in detail. Some remarks for homebrew are given below.

1. In order to set up MacPorts or homebrew, follow steps 1 - 3 of the respective instructions of MacPorts
2. Install the required dependencies:

   ```
   sudo port install gcc47 python27 py27-numpy py27-scipy py27-matplotlib mumps_seq
   sudo port select --set python python27
   ```

3. Unpack Tinyarray, enter its directory, build and install:

   ```
   python setup.py build
   sudo python setup.py install
   ```

5. Unpack Kwant, go to the Kwant directory, and edit `build.conf` to read:

   ```
   [lapack]
   extra_link_args = -Wl,-framework -Wl,Accelerate
   [mumps]
   include_dirs = /opt/local/include
   library_dirs = /opt/local/lib
   libraries = zmumps_seq mumps_common_seq pord_seq esmumps scotch scotcherr mpiseq gfortran
   ```

6. Then, build and install Kwant.

   ```
   CC=gcc-mp-4.7 LDSHARED='gcc-mp-4.7 -shared -undefined dynamic_lookup' python setup.py build
   sudo python setup.py install
   ```

You might note that installing Kwant on Mac OS X is somewhat more involved than installing on Linux. Part of the reason is that we need to mix Fortran and C code in Kwant: While C code is usually compiled using Apple compilers, Fortran code must be compiled with the Gnu Fortran compiler (there is no Apple Fortran compiler). For this reason we force the Gnu compiler suite with the environment variables `CC` and `LDSHARED` as shown above.

Mac OS X: homebrew

It is also possible to build Kwant using homebrew. The dependencies can be installed as

```
brew install gcc python
brew tap homebrew/science
brew tap homebrew/python
brew tap michaelwimmer/kwant
pip install nose six
brew install numpy scipy matplotlib
```

Note that during the installation you will be told which paths to add when you want to compile/link against scotch/metis/mumps; you need to add these to the build.conf file. Also, when linking against mumps, one needs also to link against metis (in addition to the libraries needed for MacPorts).

Windows

Our efforts to compile Kwant on Windows using only free software (MinGW) were only moderately successful. At the end of a very complicated process we obtained packages that worked, albeit unreliably. As the only

recommended way to compile Python extensions on Windows is using Visual C++, it may well be that there exists no easy solution.

It is possible to compile Kwant on Windows using non-free compilers, however we (the authors of Kwant) have no experience with this. The existing Windows binary installers of Kwant and Tinyarray were kindly prepared by Christoph Gohlke.

Build configuration

The setup script of Kwant has to know how to link against LAPACK & BLAS, and, optionally, MUMPS. By default it will assume that LAPACK and BLAS can be found under their usual names. MUMPS will be not linked against by default, except on Debian-based systems when the package `libmumps-scotch-dev` is installed.

All these settings can be configured by creating/editing the file `build.conf` in the root directory of the Kwant distribution. This configuration file consists of sections, one for each dependency, led by a [dependency-name] header and followed by name = value entries. Possible names are keyword arguments for `distutils.core.Extension` (For a complete list, see its documentation). The corresponding values are whitespace-separated lists of strings.

The two currently possible sections are [lapack] and [mumps]. The former configures the linking against LAPACK _AND_ BLAS, the latter against MUMPS (without LAPACK and BLAS).

Example `build.conf` for linking Kwant against a self-compiled MUMPS, SCOTCH and METIS:

```
[mumps]
libraries = zmumps mumps_common pord metis esmumps scotch scotcherr mpiseq
    gfortran
```

Example `build.conf` for linking Kwant with Intel MKL.:

```
[lapack]
libraries = mkl_intel_lp64 mkl_sequential mkl_core mkl_def
library_dirs = /opt/intel/mkl/lib/intel64
extra_link_args = -Wl,-rpath=/opt/intel/mkl/lib/intel64
```

The detailed syntax of `build.conf` is explained in the documentation of Python's configparser module.

Building the documentation

To build the documentation, the Sphinx documentation generator is required with `numpydoc` extension (version 0.5 or newer). If PDF documentation is to be built, the tools from the libRSVG (Debian/Ubuntu package `librsvg2-bin`) are needed to convert SVG drawings into the PDF format.

As a prerequisite for building the documentation, Kwant must have been built successfully using `./setup.py build` as described above (or Kwant must be already installed in Python's search path). HTML documentation is built by entering the `doc` subdirectory of the Kwant package and executing `make html`. PDF documentation is generated by executing `make latex` followed by `make all-pdf` in `doc/build/latex`.

Because of some quirks of how Sphinx works, it might be necessary to execute `make clean` between building HTML and PDF documentation. If this is not done, Sphinx may mistakenly use PNG files for PDF output or other problems may appear.

1.4 Authors of Kwant

The principal developers of Kwant are

- Christoph W. Groth (SPSMS-INAC-CEA Grenoble)
- Michael Wimmer (TU Delft)
- Anton R. Akhmerov (TU Delft)

- Xavier Waintal (SPSMS-INAC-CEA Grenoble)

The authors can be reached at authors@kwant-project.org.

Other people that have contributed to Kwant include

- Daniel Jaschke (SPSMS-INAC-CEA Grenoble)
- Joseph Weston (SPSMS-INAC-CEA Grenoble)

We thank Christoph Gohlke for the creation of Windows installers.

CEA is the French Commissariat à l'énergie atomique et aux énergies alternatives. The CEA is the copyright holder for the contributions of C. W. Groth, X. Waintal, and its other employees involved in Kwant.

To find out who wrote a certain part of Kwant, please use the "blame" feature of Git, the version control system.

1.4.1 Funding

During the development of Kwant 1.0, A. R. Akhmerov and M. Wimmer were supported by the Dutch Science Foundation NWO/FOM and by the ERC Advanced Investigator Grant of C. W. J. Beenakker who enthousiastically supported this project. A. R. Akhmerov was partially supported by a Lawrence Golub fellowship. C. W. Groth and X. Waintal were supported by the ERC Consolidator Grant MesoQMC. X. Waintal also acknowledges support from the STREP ConceptGraphene.

1.5 Citing Kwant

We provide Kwant as free software under a *BSD license* as a service to the physics community. If you have used Kwant for work that has lead to a scientific publication, please mention the fact that you used it explicitly in the text body. For example, you may add

> *the numerical calculations were performed using the Kwant code*

to the description of your numerical calculations. In addition, we ask you to cite the main paper that introduces Kwant:

> C. W. Groth, M. Wimmer, A. R. Akhmerov, X. Waintal, *Kwant: a software package for quantum transport*, New J. Phys. 16, 063065 (2014).

1.5.1 Other references we ask you to consider

If you have profited from the quantum transport functionality of Kwant, please also cite the upcoming paper that describes the relevant algorithms. The reference will also be added here once it is available.

Kwant owes much of its current performance to the use of the MUMPS library for solving systems of sparse linear equations. If you have done high-performance calculations, we suggest citing

> P. R. Amestoy, I. S. Duff, J. S. Koster, J. Y. L'Excellent, SIAM. J. Matrix Anal. & Appl. 23 (1), 15 (2001).

Finally, if you use the routine for generation of circular ensembles of random matrices, please cite

> F. Mezzadri, Notices Am. Math. Soc. 54, 592 (2007).

1.6 Contributing to Kwant and reporting problems

We see Kwant not just as a package with fixed functionality, but rather as a framework for implementing different physics-related algorithms using a common set of concepts and, if possible, a shared interface. We have designed it leaving room for growth, and plan to keep extending it.

External contributions to Kwant are highly welcome. You can help to advance the project not only by writing code, but also by reporting bugs, and fixing/improving the documentation. A mailing list is available for discussions.

If you have some code that works well with Kwant, or extends it in some useful way, please consider sharing it. Any external contribution will be clearly marked as such, and relevant papers will be added to the list of *suggested acknowledgements*. The complete development history is also made available through a web interface. If you plan to contribute, it is best to coordinate with us in advance either through the mailing list, or directly by email for matters that you prefer to not discuss publicly.

1.6.1 Reporting bugs

If you encounter a problem with Kwant, first try to reproduce it with as simple a system as possible. Double-check with the documentation that what you observe is actually a bug in Kwant. If you think it is, please check whether the problem is already known by searching the mailing list.

If the problem is not known yet, please email a bug report to the Kwant mailing list. A report should contain:

- The versions of software you are using (Kwant, Python, operating system, etc.)
- A description of the problem, i.e. what exactly goes wrong.
- Enough information to reproduce the bug, preferably in the form of a simple script.

1.6.2 How to contribute

We use the version control system Git to coordinate the development of Kwant. If you are new to Git, we invite you to learn its basics. (There's a plethora of information available on the Web.) Kwant's Git repository contains not only the source code, but also all of the reference documentation and the tutorial.

It is best to base your work on the latest version of Kwant:

```
git clone http://git.kwant-project.org/kwant
```

Then you can modify the code, and build Kwant and the documentation as described in the *installation instructions*.

Some things to keep in mind:

- Please keep the code consistent by adhering to the prevailing naming and formatting conventions. We generally follow the "Style Guide for Python Code" For docstrings, we follow NumPy's "Docstring Standard" and Python's "Docstring Conventions".
- Write tests for all the important functionality you add. Be sure not to break existing tests.

A useful trick for working on the source code is to build in-place so that there is no need to re-install after each change. This can be done with the following command

```
python setup.py build_ext -i --cython
```

The `kwant` subdirectory of the source distribution will be thus turned into a proper Python package that can be imported. To be able to import Kwant from within Python, one can either work in the root directory of the distribution (where the subdirectory `kwant` is located), or make a (symbolic) link from somewhere in the Python search path to the the package subdirectory.

The option `--cython` enables the translation of .pyx files into .c files. It is only needed if any .pyx files have been modified.

1.7 Kwant license

CHAPTER

TWO

TUTORIAL: LEARNING KWANT THROUGH EXAMPLES

2.1 Introduction

In this tutorial, the most important features of Kwant are explained using simple, but still physically meaningful examples. Each of the examples is commented extensively. In addition, you will find notes about more subtle, technical details at the end of each example. At first reading, these notes may be safely skipped.

A scientific article about Kwant is available as well, see Kwant website.

The article introduces Kwant with a somewhat different focus than the tutorial and it is the authors' intention that both texts complement each other. While the tutorial is more "hands-on", the article presents Kwant in a more conceptual way, as well as discussing questions of design and performance.

2.1.1 Quantum transport

This introduction to the software Kwant is written for people that already have some experience with the theory of quantum transport. Several introductions to the field are available, the most widely known is probably the book "Electronic transport in mesoscopic systems" by Supriyo Datta.

2.1.2 The Python programming language

Kwant is a library for Python. Care was taken to fit well with the spirit of the language and to take advantage of its expressive power. If you do not know Python yet, do not fear: Python is widely regarded as one of the most accessible programming languages. For an introduction we recommend the official Python Tutorial. The Beginner's Guide to Python contains a wealth of links to other tutorials, guides and books including some for absolute beginners.

2.1.3 Kwant

There are two steps in obtaining a numerical solution to a problem: The first is defining the problem in a computer-accessible way, the second solving it. The aim of a software package like Kwant is to make both steps easier.

In Kwant, the definition of the problem amounts to the creation of a tight binding system. The solution of the problem, i.e. the calculation of the values of physical observables, is achieved by passing the system to a *solver*.

The definition of a tight binding system can be seen as nothing else than the creation of a huge sparse matrix (the Hamiltonian). Equivalently, the sparse Hamiltonian matrix can be seen as an annotated *graph*: the nodes of the graph are the sites of the tight binding system, the edges are the hoppings. Sites are annotated with the corresponding on-site Hamiltonian matrix, hoppings are annotated with the corresponding hopping integral matrix.

One of the central goals of Kwant is to allow easy creation of such annotated graphs that represent tight binding system. Kwant can be made to know about the general structure of a particular system, the involved lattices and symmetries. For example, a system with a 1D translational symmetry may be used as a lead and attached to a another system. If both systems have sites which belong to the same lattices, the attaching can be done automatically, even if the shapes of the systems are irregular.

Once a tight binding system has been created, solvers provided by Kwant can be used to compute physical observables. Solvers expect the system to be in a different format than the one used for construction – the system has to be *finalized*. In a finalized system the tight binding graph is fixed but the matrix elements of the Hamiltonian may still change. The finalized format is both more efficient and simpler – the solvers don't have to deal with the various details which were facilitating the construction of the system.

The typical workflow with Kwant is as follows:

1. Create an "empty" tight binding system.
2. Set its matrix elements and hoppings.
3. Attach leads (tight binding systems with translational symmetry).
4. Pass the finalized system to a solver.

Please note that even though this tutorial mostly shows 2-d systems, Kwant is completely general with respect to the number of dimensions. Kwant does not care in the least whether systems live in one, two, three, or any other number of dimensions. The only exception is plotting, which out-of-the-box only works for up to three dimensions. (But custom projections can be specified!)

2.2 First steps: setting up a simple system and computing conductance

2.2.1 Discretization of a Schrödinger Hamiltonian

As first example, we compute the transmission probability through a two-dimensional quantum wire. The wire is described by the two-dimensional Schrödinger equation

$$H = \frac{-\hbar^2}{2m}(\partial_x^2 + \partial_y^2) + V(y)$$

with a hard-wall confinement $V(y)$ in y-direction.

To be able to implement the quantum wire with Kwant, the continuous Hamiltonian H has to be discretized thus turning it into a tight-binding model. For simplicity, we discretize H on the sites of a square lattice with lattice constant a. Each site with the integer lattice coordinates (i, j) has the real-space coordinates $(x, y) = (ai, aj)$.

Introducing the discretized positional states

$$|i, j\rangle \equiv |ai, aj\rangle = |x, y\rangle$$

the second-order differential operators can be expressed in the limit $a \to 0$ as

$$\partial_x^2 = \frac{1}{a^2} \sum_{i,j} \left(|i+1, j\rangle \langle i, j| + |i, j\rangle \langle i+1, j| - 2\,|i, j\rangle \langle i, j| \right),$$

and an equivalent expression for ∂_y^2. Subsitituting them in the Hamiltonian gives us

$$H = \sum_{i,j} \big[\left(V(ai, aj) + 4t\right) |i, j\rangle \langle i, j| - t\big(|i+1, j\rangle \langle i, j| + |i, j\rangle \langle i+1, j| + |i, j+1\rangle \langle i, j| + |i, j\rangle \langle i, j+1| \big)\big]$$

with

$$t = \frac{\hbar^2}{2ma^2}.$$

For finite a, this discretized Hamiltonian approximates the continuous one to any required accuracy. The approximation is good for all quantum states with a wave length considerably larger than a.

The remainder of this section demonstrates how to realize the discretized Hamiltonian in Kwant and how to perform transmission calculations. For simplicity, we choose to work in such units that $t = a = 1$.

2.2.2 Transport through a quantum wire

See also:

The complete source code of this example can be found in `tutorial/quantum_wire.py`

In order to use Kwant, we need to import it:

```
import kwant
```

Enabling Kwant is as easy as this [1] !

The first step is now the definition of the system with scattering region and leads. For this we make use of the `Builder` type that allows to define a system in a convenient way. We need to create an instance of it:

```
sys = kwant.Builder()
```

Observe that we just accessed `Builder` by the name `kwant.Builder`. We could have just as well written `kwant.builder.Builder` instead. Kwant consists of a number of sub-packages that are all covered in the *reference documentation*. For convenience, some of the most widely-used members of the sub-packages are also accessible directly through the top-level `kwant` package.

Apart from `Builder` we also need to specify what kind of sites we want to add to the system. Here we work with a square lattice. For simplicity, we set the lattice constant to unity:

```
a = 1
lat = kwant.lattice.square(a)
```

Since we work with a square lattice, we label the points with two integer coordinates *(i, j)*. `Builder` then allows us to add matrix elements corresponding to lattice points: `sys[lat(i, j)] = ...` sets the on-site energy for the point *(i, j)*, and `sys[lat(i1, j1), lat(i2, j2)] = ...` the hopping matrix element **from** point *(i2, j2)* **to** point *(i1, j1)*.

Note that we need to specify sites for `Builder` in the form `lat(i, j)`. The lattice object *lat* does the translation from integer coordinates to proper site format needed in Builder (more about that in the technical details below).

We now build a rectangular scattering region that is *W* lattice points wide and *L* lattice points long:

```
t = 1.0
W = 10
L = 30

# Define the scattering region

for i in xrange(L):
    for j in xrange(W):
        # On-site Hamiltonian
        sys[lat(i, j)] = 4 * t

        # Hopping in y-direction
        if j > 0:
            sys[lat(i, j), lat(i, j - 1)] = -t

        # Hopping in x-direction
        if i > 0:
            sys[lat(i, j), lat(i - 1, j)] = -t
```

Observe how the above code corresponds directly to the terms of the discretized Hamiltonian: “On-site Hamiltonian” implements

$$\sum_{i,j} \left(V(ai, aj) + 4t\right) |i, j\rangle \langle i, j|$$

[1] http://xkcd.com/353/

(with zero potential). "Hopping in x-direction" implements

$$\sum_{i,j} -t\big(\,|i+1,j\rangle\,\langle i,j| + |i,j\rangle\,\langle i+1,j|\,\big),$$

and "Hopping in y-direction" implements

$$\sum_{i,j} -t\big(\,|i,j+1\rangle\,\langle i,j| + |i,j\rangle\,\langle i,j+1|\,\big).$$

The hard-wall confinement is realized by not having hoppings (and sites) beyond a certain region of space.

Next, we define the leads. Leads are also constructed using `Builder`, but in this case, the system must have a translational symmetry:

```
sym_left_lead = kwant.TranslationalSymmetry((-a, 0))
left_lead = kwant.Builder(sym_left_lead)
```

Here, the `Builder` takes a `TranslationalSymmetry` as the optional parameter. Note that the (real-space) vector `(-a, 0)` defining the translational symmetry must point in a direction *away* from the scattering region, *into* the lead – hence, lead 0 [2] will be the left lead, extending to infinity to the left.

For the lead itself it is enough to add the points of one unit cell as well as the hoppings inside one unit cell and to the next unit cell of the lead. For a square lattice, and a lead in y-direction the unit cell is simply a vertical line of points:

```
for j in xrange(W):
    left_lead[lat(0, j)] = 4 * t
    if j > 0:
        left_lead[lat(0, j), lat(0, j - 1)] = -t
    left_lead[lat(1, j), lat(0, j)] = -t
```

Note that here it doesn't matter if you add the hoppings to the next or the previous unit cell – the translational symmetry takes care of that. The isolated, infinite is attached at the correct position using

```
sys.attach_lead(left_lead)
```

This call returns the lead number which will be used to refer to the lead when computing transmissions (further down in this tutorial). More details about attaching leads can be found in the tutorial *Nontrivial shapes*.

We also want to add a lead on the right side. The only difference to the left lead is that the vector of the translational symmetry must point to the right, the remaining code is the same:

```
sym_right_lead = kwant.TranslationalSymmetry((a, 0))
right_lead = kwant.Builder(sym_right_lead)

for j in xrange(W):
    right_lead[lat(0, j)] = 4 * t
    if j > 0:
        right_lead[lat(0, j), lat(0, j - 1)] = -t
    right_lead[lat(1, j), lat(0, j)] = -t

sys.attach_lead(right_lead)
```

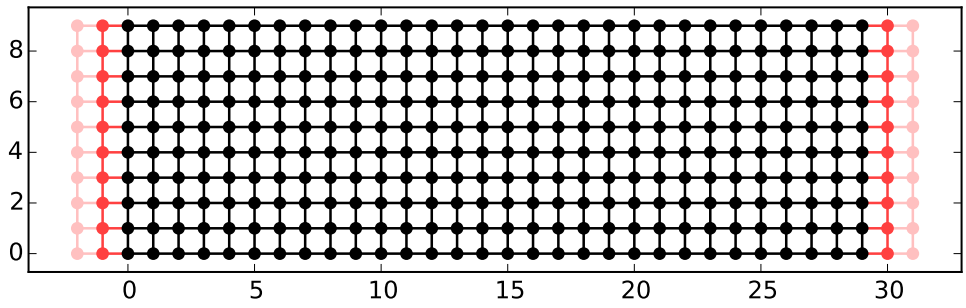
Note that here we added points with x-coordinate 0, just as for the left lead. You might object that the right lead should be placed L (or $L+1$?) points to the right with respect to the left lead. In fact, you do not need to worry about that.

Now we have finished building our system! We plot it, to make sure we didn't make any mistakes:

```
kwant.plot(sys)
```

This should bring up this picture:

[2] Leads are numbered in the python convention, starting from 0.



The system is represented in the usual way for tight-binding systems: dots represent the lattice points *(i, j)*, and for every nonzero hopping element between points there is a line connecting these points. From the leads, only a few (default 2) unit cells are shown, with fading color.

In order to use our system for a transport calculation, we need to finalize it

```
sys = sys.finalized()
```

Having successfully created a system, we now can immediately start to compute its conductance as a function of energy:

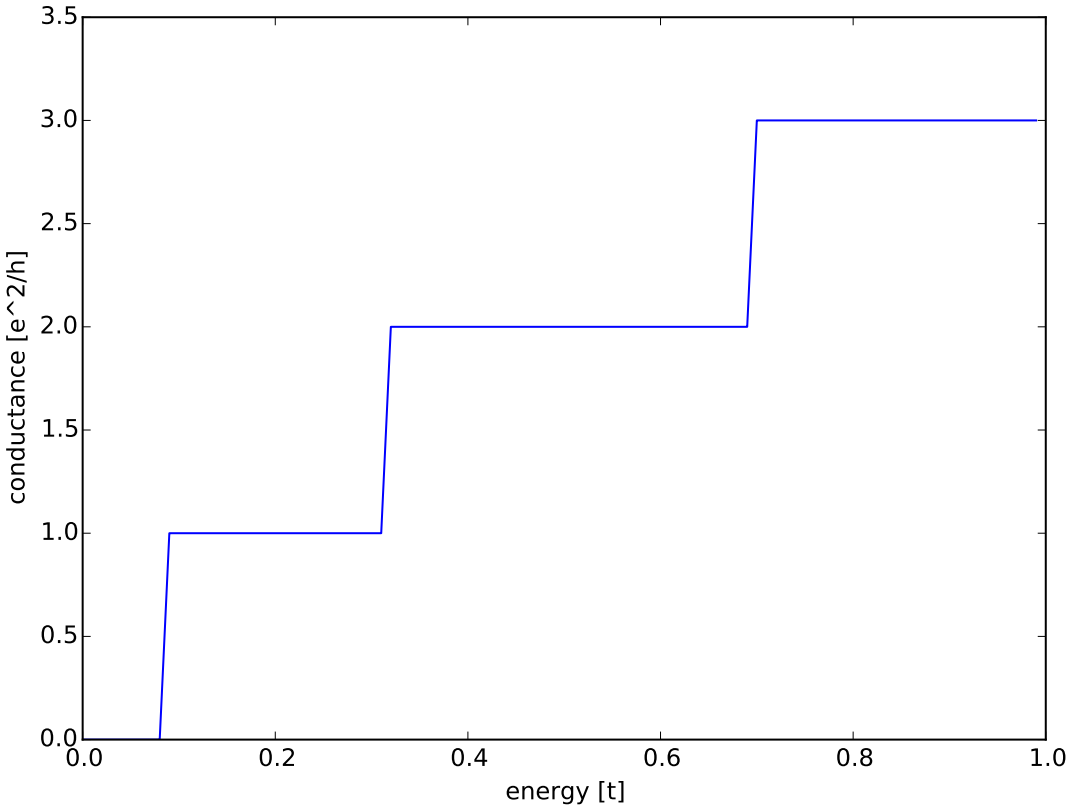
```
energies = []
data = []
for ie in xrange(100):
    energy = ie * 0.01

    # compute the scattering matrix at a given energy
    smatrix = kwant.smatrix(sys, energy)

    # compute the transmission probability from lead 0 to
    # lead 1
    energies.append(energy)
    data.append(smatrix.transmission(1, 0))
```

We use `kwant.smatrix` which is a short name for `kwant.solvers.default.smatrix` of the default solver module `kwant.solvers.default`. `kwant.smatrix` computes the scattering matrix `smatrix` solving a sparse linear system. `smatrix` itself allows to directly compute the total transmission probability from lead 0 to lead 1 as `smatrix.transmission(1, 0)`. The numbering used to refer to the leads here is the same as the numbering assigned by the call to `attach_lead` earlier in the tutorial.

Finally we can use `matplotlib` to make a plot of the computed data (although writing to file and using an external viewer such as gnuplot or xmgrace is just as viable)

```
pyplot.figure()
pyplot.plot(energies, data)
pyplot.xlabel("energy [t]")
pyplot.ylabel("conductance [e^2/h]")
pyplot.show()
```

This should yield the result



We see a conductance quantized in units of e^2/h, increasing in steps as the energy is increased. The value of the conductance is determined by the number of occupied subbands that increases with energy. **Technical details**

- In the example above, when building the system, only one direction of hopping is given, i.e. `sys[lat(i, j), lat(i, j-1)] = ...` and not also `sys[lat(i, j-1), lat(i, j)] = ...`. The reason is that `Builder` automatically adds the other direction of the hopping such that the resulting system is Hermitian.

  However, it does not hurt to define the opposite direction of hopping as well:

  ```
  sys[lat(1, 0), lat(0, 0)] = -t
  sys[lat(0, 0), lat(1, 0)] = -t.conj()
  ```

  (assuming that *t* is complex) is perfectly fine. However, be aware that also

  ```
  sys[lat(1, 0), lat(0, 0)] = -1
  sys[lat(0, 0), lat(1, 0)] = -2
  ```

  is valid code. In the latter case, the hopping `sys[lat(1, 0), lat(0, 0)]` is overwritten by the last line and also equals to -2.

- Some more details the relation between `Builder` and the square lattice *lat* in the example:

  Technically, `Builder` expects **sites** as indices. Sites themselves have a certain type, and belong to a **site family**. A site family is also used to convert something that represents a site (like a tuple) into a proper `Site` object that can be used with `Builder`.

  In the above example, *lat* is the site family. `lat(i, j)` then translates the description of a lattice site in terms of two integer indices (which is the natural way to do here) into a proper `Site` object.

  The concept of site families and sites allows `Builder` to mix arbitrary lattices and site families

- In the example, we wrote

```
sys = sys.finalized()
```

In doing so, we transform the `Builder` object (with which we built up the system step by step) into a `System` that has a fixed structure (which we cannot change any more).

Note that this means that we cannot access the `Builder` object any more. This is not necesarry any more, as the computational routines all expect finalized systems. It even has the advantage that python is now free to release the memory occupied by the `Builder` which, for large systems, can be considerable. Roughly speaking, the above code corresponds to

```
fsys = sys.finalized()
del sys
sys = fsys
```

- Even though the vector passed to the `TranslationalSymmetry` is specified in real space, it must be compatible with the lattice symmetries. A single lead can consists of sites belonging to more than one lattice, but of course the translational symmetry of the lead has to be shared by all of them.
- Instead of plotting to the screen (which is standard) `plot` can also write to a file specified by the argument `file`. For the plotting to the screen to work the module `matplotlib.pyplot` has to be imported. (An informative error message will remind you if you forget.) The reason for this is pretty technical: matplotlib's "backend" can only be chosen before `matplotlib.pyplot` has been imported. Would Kwant import that module by itself, it would deprive you of the possibility to choose a non-default backend later.

2.2.3 Building the same system with less code

See also:

The complete source code of this example can be found in `tutorial/quantum_wire_revisited.py`

Kwant allows for more than one way to build a system. The reason is that `Builder` is essentially just a container that can be filled in different ways. Here we present a more compact rewrite of the previous example (still with the same results).

Also, the previous example was written in the form of a Python script with little structure, and with everything governed by global variables. This is OK for such a simple example, but for larger projects it makes sense to partition the code into separate entities. In this example we therefore also aim at more structure.

We begin the program collecting all imports in the beginning of the file and put the build-up of the system into a separate function *make_system*:

```
import kwant

# For plotting
from matplotlib import pyplot


def make_system(a=1, t=1.0, W=10, L=30):
    # Start with an empty tight-binding system and a single square lattice.
    # `a` is the lattice constant (by default set to 1 for simplicity.
    lat = kwant.lattice.square(a)

    sys = kwant.Builder()
```

Previously, the scattering region was build using two `for`-loops. Instead, we now write:

```
    sys[(lat(x, y) for x in range(L) for y in range(W))] = 4 * t
```

Here, all lattice points are added at once in the first line. The construct `((i, j) for i in xrange(L) for j in xrange(W))` is a generator that iterates over all points in the rectangle as did the two `for`-loops in the previous example. In fact, a `Builder` can not only be indexed by a single lattice point – it also allows for lists of points, or, as in this example, a generator (as is also used in list comprehensions in python).

Having added all lattice points in one line, we now turn to the hoppings. In this case, an iterable like for the lattice points becomes a bit cumbersome, and we use instead another feature of Kwant:

```
    sys[lat.neighbors()] = -t
```

In regular lattices, hoppings form large groups such that hoppings within a group can be transformed into one another by lattice translations. In order to allow to easily manipulate such hoppings, an object `HoppingKind` is provided. When given a `Builder` as an argument, `HoppingKind` yields all the hoppings of a certain kind that can be added to this builder without adding new sites. When `HoppingKind` is given to `Builder` as a key, it means that something is done to all the possible hoppings of this kind. A list of `HoppingKind` objects corresponding to nearest neighbors in lattices in Kwant is obtained using `lat.neighbors()`. `sys[lat.neighbors()] = -t` then sets all of those hopping matrix elements at once. In order to set values for all the nth-nearest neighbors at once, one can similarly use `sys[lat.neighbors(n)] = -t`. More detailed example of using `HoppingKind` directly will be provided in *Matrix structure of on-site and hopping elements*.

The left lead is constructed in an analogous way:

```
    lead = kwant.Builder(kwant.TranslationalSymmetry((-a, 0)))
    lead[(lat(0, j) for j in xrange(W))] = 4 * t
    lead[lat.neighbors()] = -t
```

The previous example duplicated almost identical code for the left and the right lead. The only difference was the direction of the translational symmetry vector. Here, we only construct the left lead, and use the method `reversed` of `Builder` to obtain a copy of a lead pointing in the opposite direction. Both leads are attached as before and the finished system returned:

```
    sys.attach_lead(lead)
    sys.attach_lead(lead.reversed())

    return sys
```

The remainder of the script has been organized into two functions. One for the plotting of the conductance.

```
def plot_conductance(sys, energies):
    # Compute conductance
    data = []
    for energy in energies:
        smatrix = kwant.smatrix(sys, energy)
        data.append(smatrix.transmission(1, 0))

    pyplot.figure()
    pyplot.plot(energies, data)
    pyplot.xlabel("energy [t]")
    pyplot.ylabel("conductance [e^2/h]")
    pyplot.show()
```

And one `main` function.

```
def main():
    sys = make_system()

    # Check that the system looks as intended.
    kwant.plot(sys)

    # Finalize the system.
    sys = sys.finalized()

    # We should see conductance steps.
    plot_conductance(sys, energies=[0.01 * i for i in xrange(100)])
```

Finally, we use the following standard Python construct [3] to execute `main` if the program is used as a script (i.e. executed as `python quantum_wire_revisited.py`):

[3] http://docs.python.org/library/__main__.html

```
if __name__ == '__main__':
    main()
```

If the example, however, is imported inside Python using `import quantum_wire_revisted as qw`, `main` is not executed automatically. Instead, you can execute it manually using `qw.main()`. On the other hand, you also have access to the other functions, `make_system` and `plot_conductance`, and can thus play with the parameters.

The result of the example should be identical to the previous one. **Technical details**

- We have seen different ways to add lattice points to a `Builder`. It allows to
  - add single points, specified as sites
  - add several points at once using a generator (as in this example)
  - add several points at once using a list (typically less effective compared to a generator)

  For technical reasons it is not possible to add several points using a tuple of sites. Hence it is worth noting a subtle detail in

  ```
  sys[(lat(x, y) for x in range(L) for y in range(W))] = 4 * t
  ```

  Note that `(lat(x, y) for x in range(L) for y in range(W))` is not a tuple, but a generator.

  Let us elaborate a bit more on this using a simpler example:

  ```
  >>> a = (0, 1, 2, 3)
  >>> b = (i for i in xrange(4))
  ```

  Here, *a* is a tuple, whereas *b* is a generator. One difference is that one can subscript tuples, but not generators:

  ```
  >>> a[0]
  0
  >>> b[0]
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: 'generator' object is unsubscriptable
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: 'generator' object is unsubscriptable
  ```

  However, both can be used in `for`-loops, for example.

- In the example, we have added all the hoppings using `HoppingKind`. In fact, hoppings can be added in the same fashion as sites, namely specifying
  - a single hopping
  - several hoppings via a generator
  - several hoppings via a list

  A hopping is defined using two sites. If several hoppings are added at once, these two sites should be encapsulated in a tuple. In particular, one must write:

  ```
  sys[((lat(0,j+1), lat(0, j)) for j in xrange(W-1)] = ...
  ```

  or:

  ```
  sys[[(site1, site2), (site3, site4), ...]] = ...
  ```

  You might wonder, why it is then possible to write for a single hopping:

  ```
  sys[site1, site2] = ...
  ```

  instead of

```
sys[(site1, site2)] = ...
```

In fact, due to the way python handles subscripting, `sys[site1, site2]` is the same as `sys[(site1, site2)]`.

(This is the deeper reason why several sites cannot be added as a tuple – it would be impossible to distinguish whether one would like to add two separate sites, or one hopping.

2.3 More interesting systems: spin, potential, shape

Each of the following three examples highlights different ways to go beyond the very simple examples of the previous section.

2.3.1 Matrix structure of on-site and hopping elements

See also:

The complete source code of this example can be found in `tutorial/spin_orbit.py`

We begin by extending the simple 2DEG-Hamiltonian by a Rashba spin-orbit coupling and a Zeeman splitting due to an external magnetic field:

$$H = \frac{-\hbar^2}{2m}(\partial_x^2 + \partial_y^2) - i\alpha(\partial_x\sigma_y - \partial_y\sigma_x) + E_Z\sigma_z + V(y)$$

Here $\sigma_{x,y,z}$ denote the Pauli matrices.

It turns out that this well studied Rashba-Hamiltonian has some peculiar properties in (ballistic) nanowires: It was first predicted theoretically in Phys. Rev. Lett. 90, 256601 (2003) that such a system should exhibit non-monotonic conductance steps due to a spin-orbit gap. Only very recently, this non-monotonic behavior has been supposedly observed in experiment: Nature Physics 6, 336 (2010). Here we will show that a very simple extension of our previous examples will exactly show this behavior (Note though that no care was taken to choose realistic parameters).

The tight-binding model corresponding to the Rashba-Hamiltonian naturally exhibits a 2x2-matrix structure of onsite energies and hoppings. In order to use matrices in our program, we import the Tinyarray package. (NumPy would work as well, but Tinyarray is much faster for small arrays.)

```
import tinyarray
```

For convenience, we define the Pauli-matrices first (with σ_0 the unit matrix):

```
sigma_0 = tinyarray.array([[1, 0], [0, 1]])
sigma_x = tinyarray.array([[0, 1], [1, 0]])
sigma_y = tinyarray.array([[0, -1j], [1j, 0]])
sigma_z = tinyarray.array([[1, 0], [0, -1]])
```

Previously, we used numbers as the values of our matrix elements. However, `Builder` also accepts matrices as values, and we can simply write:

```
sys[(lat(x, y) for x in range(L) for y in range(W))] = \
    4 * t * sigma_0 + e_z * sigma_z
# hoppings in x-direction
sys[kwant.builder.HoppingKind((1, 0), lat, lat)] = \
    -t * sigma_0 - 1j * alpha * sigma_y
# hoppings in y-directions
sys[kwant.builder.HoppingKind((0, 1), lat, lat)] = \
    -t * sigma_0 + 1j * alpha * sigma_x
```

Note that the Zeeman energy adds to the onsite term, whereas the Rashba spin-orbit term adds to the hoppings (due to the derivative operator). Furthermore, the hoppings in x and y-direction have a different matrix structure.

We now cannot use `lat.neighbors()` to add all the hoppings at once, since we now have to distinguish x and y-direction. Because of that, we have to explicitly specify the hoppings in the form expected by `HoppingKind`:

- A tuple with relative lattice indices. For example, *(1, 0)* means hopping from *(i, j)* to *(i+1, j)*, whereas *(1, 1)* would mean hopping to *(i+1, j+1)*.
- The target lattice (where to hop to)
- The source lattice (where the hopping originates)

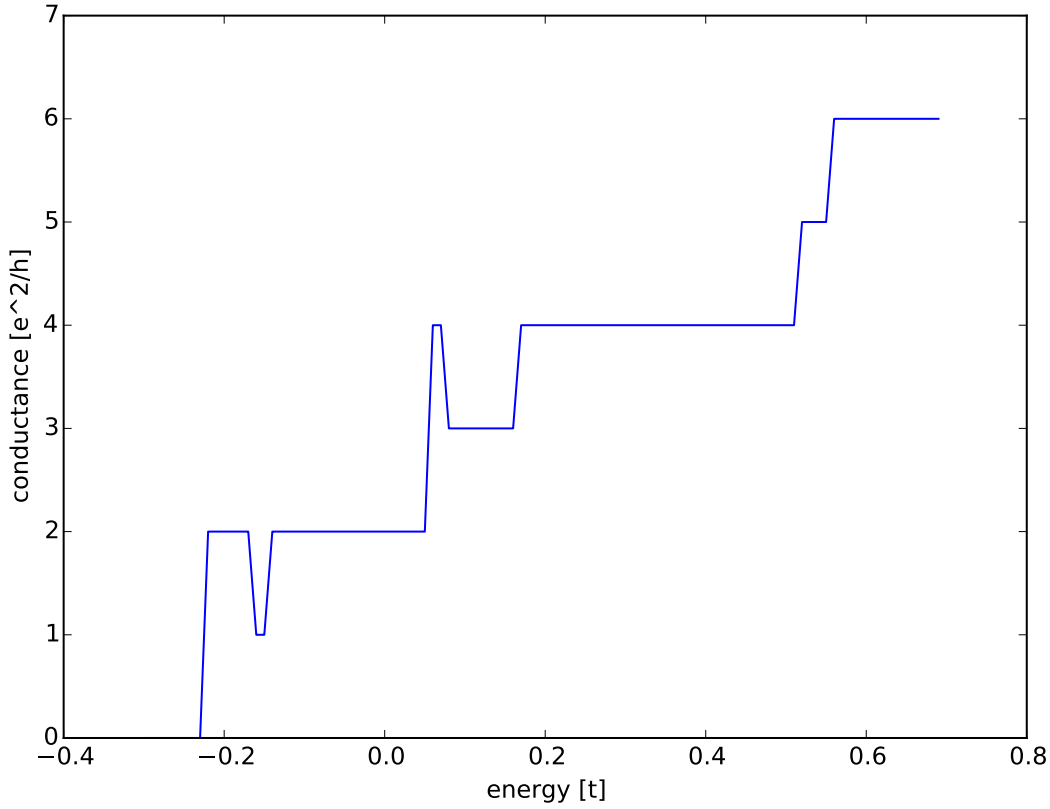
Since we are only dealing with a single lattice here, source and target lattice are identical, but still must be specified (for an example with hopping between different (sub)lattices, see *Beyond square lattices: graphene*).

Again, it is enough to specify one direction of the hopping (i.e. when specifying *(1, 0)* it is not necessary to specify *(-1, 0)*), `Builder` assures hermiticity.

The leads also allow for a matrix structure,

```
lead[(lat(0, j) for j in xrange(W))] = 4 * t * sigma_0 + e_z * sigma_z
# hoppings in x-direction
lead[kwant.builder.HoppingKind((1, 0), lat, lat)] = \
    -t * sigma_0 - 1j * alpha * sigma_y
# hoppings in y-directions
lead[kwant.builder.HoppingKind((0, 1), lat, lat)] = \
    -t * sigma_0 + 1j * alpha * sigma_x
```

The remainder of the code is unchanged, and as a result we should obtain the following, clearly non-monotonic conductance steps:



Technical details

- The Tinyarray package, one of the dependencies of Kwant, implements efficient small arrays. It is used internally in Kwant for storing small vectors and matrices. For performance, it is preferable to define small arrays that are going to be used with Kwant using Tinyarray. However, NumPy would work as well:

```
import numpy
sigma_0 = numpy.array([[1, 0], [0, 1]])
sigma_x = numpy.array([[0, 1], [1, 0]])
sigma_y = numpy.array([[0, -1j], [1j, 0]])
sigma_z = numpy.array([[1, 0], [0, -1]])
```

Tinyarray arrays behave for most purposes like NumPy arrays except that they are immutable: they cannot be changed once created. This is important for Kwant: it allows them to be used directly as dictionary keys.

- It should be emphasized that the relative hopping used for `HoppingKind` is given in terms of lattice indices, i.e. relative to the Bravais lattice vectors. For a square lattice, the Bravais lattice vectors are simply *(a,0)* and *(0,a)*, and hence the mapping from lattice indices *(i,j)* to real space and back is trivial. This becomes more involved in more complicated lattices, where the real-space directions corresponding to, for example, *(1,0)* and *(0,1)* need not be orthogonal any more (see *Beyond square lattices: graphene*).

2.3.2 Spatially dependent values through functions

See also:

The complete source code of this example can be found in `tutorial/quantum_well.py`

Up to now, all examples had position-independent matrix-elements (and thus translational invariance along the wire, which was the origin of the conductance steps). Now, we consider the case of a position-dependent potential:

$$H = \frac{\hbar^2}{2m}(\partial_x^2 + \partial_y^2) + V(x, y)$$

The position-dependent potential enters in the onsite energies. One possibility would be to again set the onsite matrix elements of each lattice point individually (as in *Transport through a quantum wire*). However, changing the potential then implies the need to build up the system again.

Instead, we use a python *function* to define the onsite energies. We define the potential profile of a quantum well as:

```
def make_system(a=1, t=1.0, W=10, L=30, L_well=10):
    # Start with an empty tight-binding system and a single square lattice.
    # `a` is the lattice constant (by default set to 1 for simplicity.
    lat = kwant.lattice.square(a)

    sys = kwant.Builder()

    #### Define the scattering region. ####
    # Potential profile
    def potential(site, pot):
        (x, y) = site.pos
        if (L - L_well) / 2 < x < (L + L_well) / 2:
            return pot
        else:
            return 0
```

This function takes two arguments: the first of type `Site`, from which you can get the real-space coordinates using `site.pos`, and the value of the potential as the second. Note that in *potential* we can access variables of the surrounding function: *L* and *L_well* are taken from the namespace of *make_system*.

Kwant now allows us to pass a function as a value to `Builder`:

```
    def onsite(site, pot=0):
        return 4 * t + potential(site, pot)

    sys[(lat(x, y) for x in range(L) for y in range(W))] = onsite
    sys[lat.neighbors()] = -t
```

For each lattice point, the corresponding site is then passed as the first argument to the function *onsite*. The values of any additional parameters, which can be used to alter the Hamiltonian matrix elements at a later stage, are

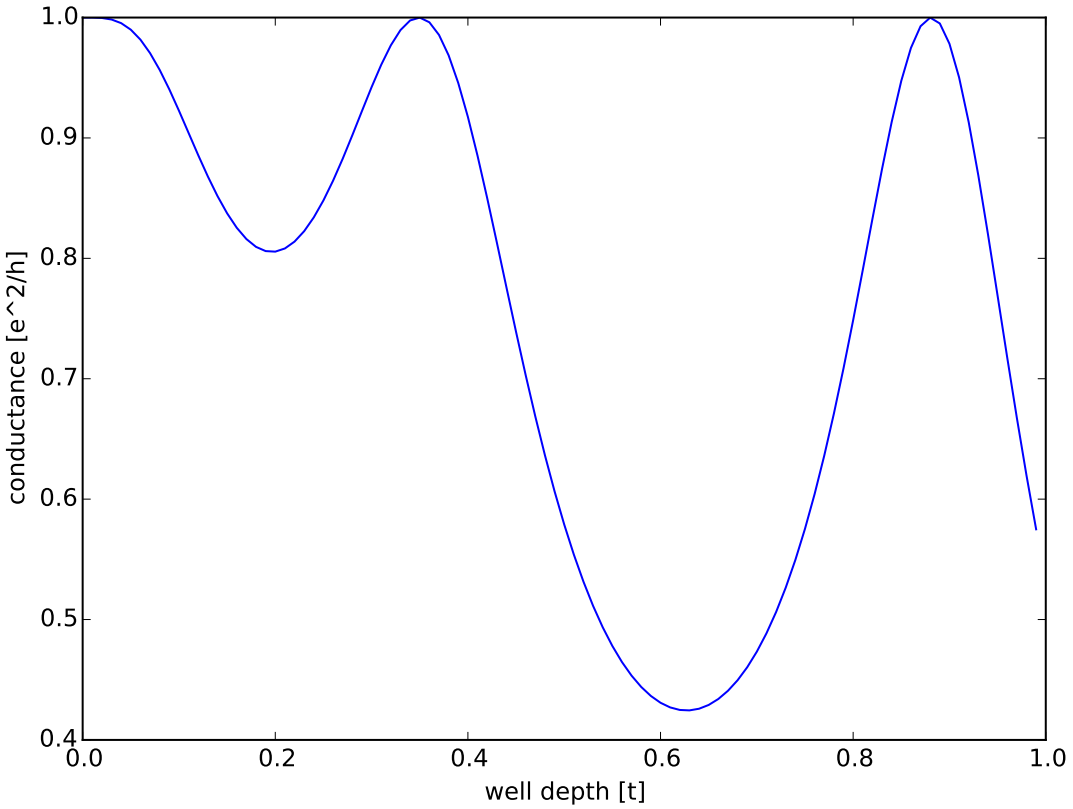
specified later during the call to *smatrix*. Note that we had to define *onsite*, as it is not possible to mix values and functions as in `sys[...] = 4 * t + potential`.

For the leads, we just use constant values as before. If we passed a function also for the leads (which is perfectly allowed), this function would need to be compatible with the translational symmetry of the lead – this should be kept in mind.

Finally, we compute the transmission probability:

```
# Compute conductance
data = []
for welldepth in welldepths:
    smatrix = kwant.smatrix(sys, energy, args=[-welldepth])
    data.append(smatrix.transmission(1, 0))

pyplot.figure()
pyplot.plot(welldepths, data)
pyplot.xlabel("well depth [t]")
pyplot.ylabel("conductance [e^2/h]")
pyplot.show()
```

`kwant.smatrix` allows us to specify a list, *args*, that will be passed as additional arguments to the functions that provide the Hamiltonian matrix elements. In this example we are able to solve the system for different depths of the potential well by passing the potential value. We obtain the result:



Starting from no potential (well depth = 0), we observe the typical oscillatory transmission behavior through resonances in the quantum well.

Warning: If functions are used to set values inside a lead, then they must satisfy the same symmetry as the lead does. There is (currently) no check and wrong results will be the consequence of a misbehaving function.

Technical details

- Functions can also be used for hoppings. In this case, they take two `Site`'s as arguments and then an

arbitrary number of additional arguments.

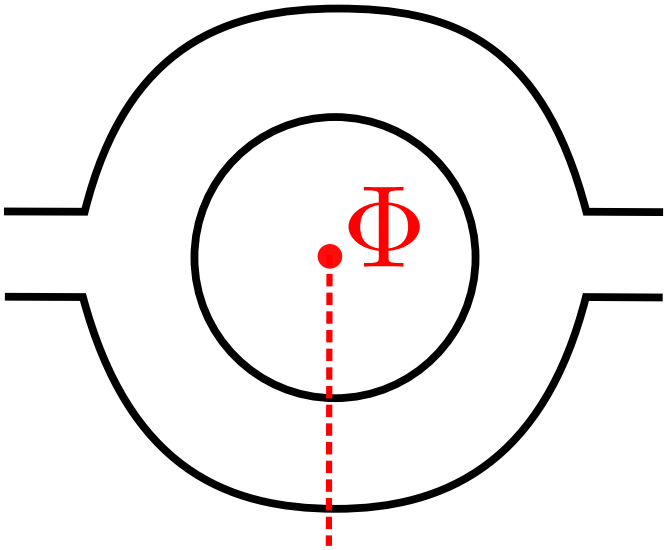
- Apart from the real-space position *pos*, `Site` has also an attribute *tag* containing the lattice indices of the site.

2.3.3 Nontrivial shapes

See also:

The complete source code of this example can be found in `tutorial/ab_ring.py`

Up to now, we only dealt with simple wire geometries. Now we turn to the case of a more complex geometry, namely transport through a quantum ring that is pierced by a magnetic flux Φ:



For a flux line, it is possible to choose a gauge such that a charged particle acquires a phase $e\Phi/h$ whenever it crosses the branch cut originating from the flux line (branch cut shown as red dashed line) [4]. There are more symmetric gauges, but this one is most convenient to implement numerically.

Defining such a complex structure adding individual lattice sites is possible, but cumbersome. Fortunately, there is a more convenient solution: First, define a boolean function defining the desired shape, i.e. a function that returns `True` whenever a point is inside the shape, and `False` otherwise:

```
def make_system(a=1, t=1.0, W=10, r1=10, r2=20):
    # Start with an empty tight-binding system and a single square lattice.
    # `a` is the lattice constant (by default set to 1 for simplicity).

    lat = kwant.lattice.square(a)

    sys = kwant.Builder()

    #### Define the scattering region. ####
    # Now, we aim for a more complex shape, namely a ring (or annulus)
    def ring(pos):
        (x, y) = pos
        rsq = x ** 2 + y ** 2
        return (r1 ** 2 < rsq < r2 ** 2)
```

Note that this function takes a real-space position as argument (not a `Site`).

We can now simply add all of the lattice points inside this shape at once, using the function *shape* provided by the lattice:

[4] The corresponding vector potential is $A_x(x, y) = \Phi\delta(x)\Theta(-y)$ which yields the correct magnetic field $B(x, y) = \Phi\delta(x)\delta(y)$.

```
sys[lat.shape(ring, (0, r1 + 1))] = 4 * t
sys[lat.neighbors()] = -t
```

Here, `lat.shape` takes as a second parameter a (real-space) point that is inside the desired shape. The hoppings can still be added using `lat.neighbors()` as before.

Up to now, the system contains constant hoppings and onsite energies, and we still need to include the phase shift due to the magnetic flux. This is done by **overwriting** the values of hoppings in x-direction along the branch cut in the lower arm of the ring. For this we select all hoppings in x-direction that are of the form *(lat(1, j), lat(0, j))* with `j<0`:

```
def fluxphase(site1, site2, phi):
    return exp(1j * phi)

def crosses_branchcut(hop):
    ix0, iy0 = hop[0].tag

    # builder.HoppingKind with the argument (1, 0) below
    # returns hoppings ordered as ((i+1, j), (i, j))
    return iy0 < 0 and ix0 == 1  # ix1 == 0 then implied

# Modify only those hopings in x-direction that cross the branch cut
def hops_across_cut(sys):
    for hop in kwant.builder.HoppingKind((1, 0), lat, lat)(sys):
        if crosses_branchcut(hop):
            yield hop
sys[hops_across_cut] = fluxphase
```

Here, *crosses_branchcut* is a boolean function that returns `True` for the desired hoppings. We then use again a generator (this time with an `if`-conditional) to set the value of all hoppings across the branch cut to *fluxphase*. The rationale behind using a function instead of a constant value for the hopping is again that we want to vary the flux through the ring, without constantly rebuilding the system – instead the flux is governed by the parameter *phi*.

For the leads, we can also use the `lat.shape`-functionality:

```
sym_lead = kwant.TranslationalSymmetry((-a, 0))
lead = kwant.Builder(sym_lead)

def lead_shape(pos):
    (x, y) = pos
    return (-W / 2 < y < W / 2)

lead[lat.shape(lead_shape, (0, 0))] = 4 * t
lead[lat.neighbors()] = -t
```

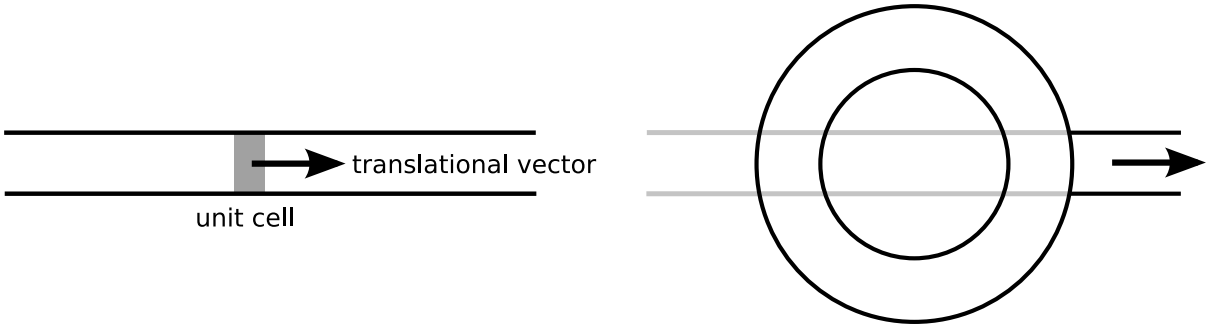
Here, the shape must be compatible with the translational symmetry of the lead `sym_lead`. In particular, this means that it should extend to infinity along the translational symmetry direction (note how there is no restriction on `x` in `lead_shape`) [5].

Attaching the leads is done as before:

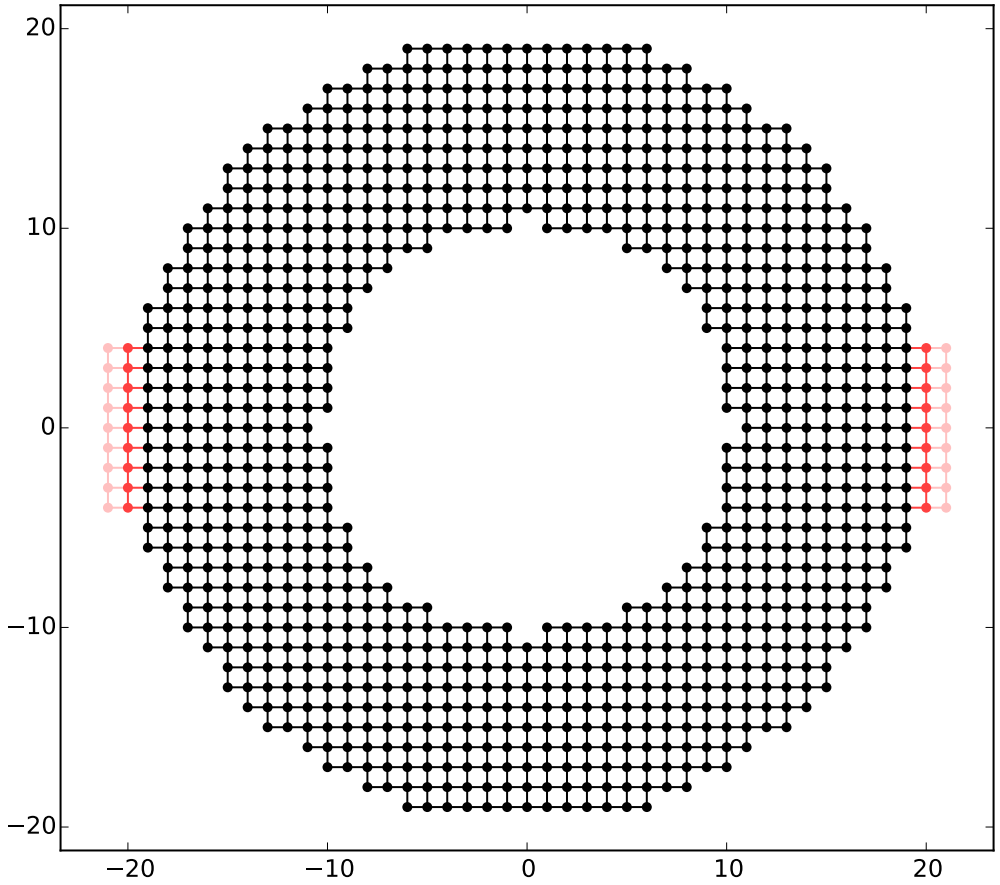
```
sys.attach_lead(lead)
sys.attach_lead(lead.reversed())
```

In fact, attaching leads seems not so simple any more for the current structure with a scattering region very much different from the lead shapes. However, the choice of unit cell together with the translational vector allows to place the lead unambiguously in real space – the unit cell is repeated infinitely many times in the direction and opposite to the direction of the translational vector. Kwant examines the lead starting from infinity and traces it back (going opposite to the direction of the translational vector) until it intersects the scattering region. At this intersection, the lead is attached:

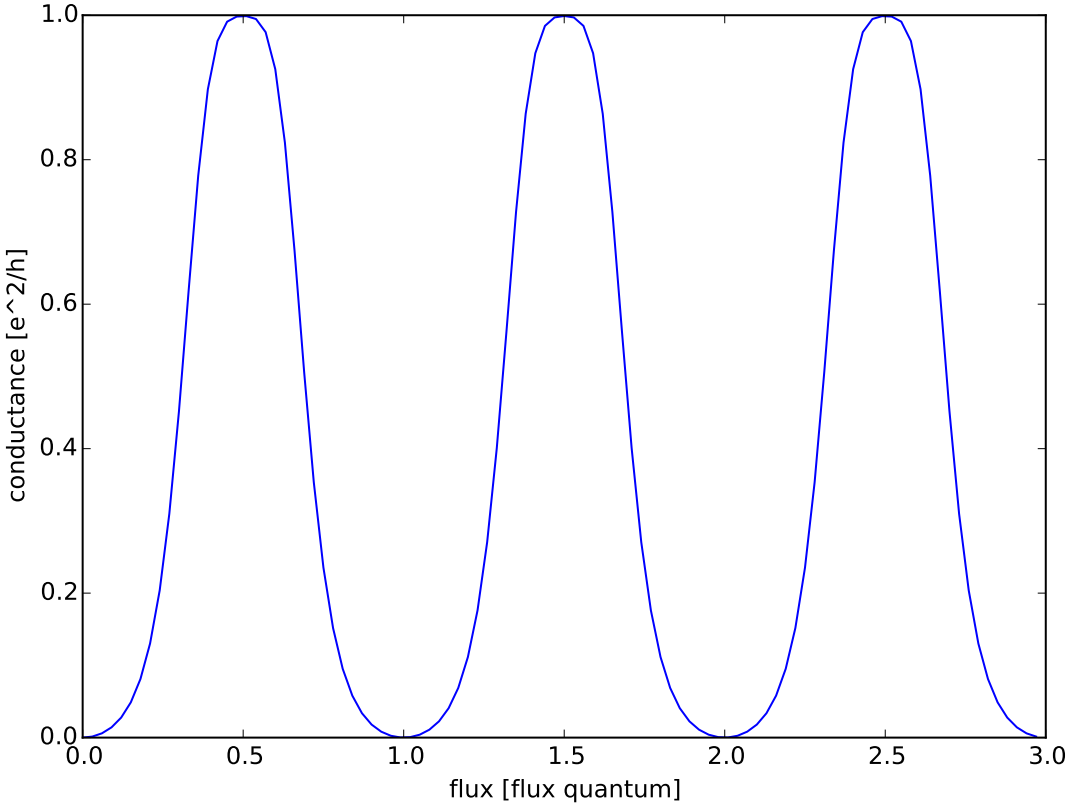
[5] Despite the "infinite" shape, the unit cell will still be finite; the `TranslationalSymmetry` takes care of that.



After the lead has been attached, the system should look like this:

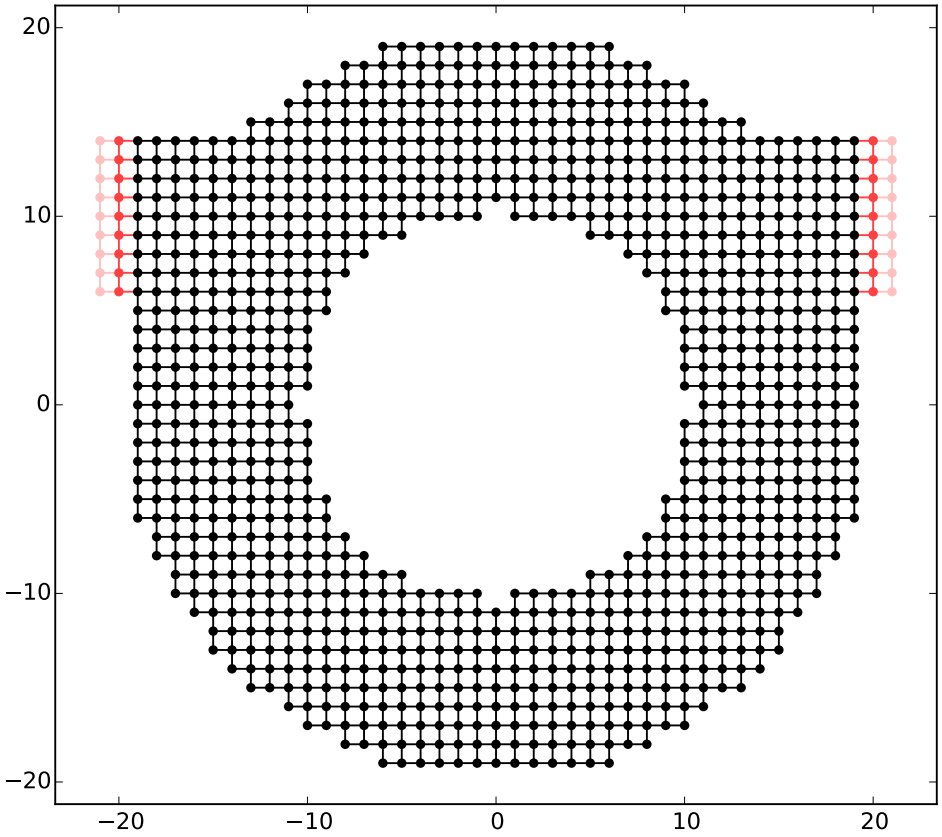


The computation of the conductance goes in the same fashion as before. Finally you should get the following result:



where one can observe the conductance oscillations with the period of one flux quantum. **Technical details**

- Leads have to have proper periodicity. Furthermore, the Kwant format requires the hopping from the leads to the scattering region to be identical to the hoppings between unit cells in the lead. `attach_lead` takes care of all these details for you! In fact, it even adds points to the scattering region, if proper attaching requires this. This becomes more apparent if we attach the leads a bit further away from the central axis o the ring, as was done in this example:

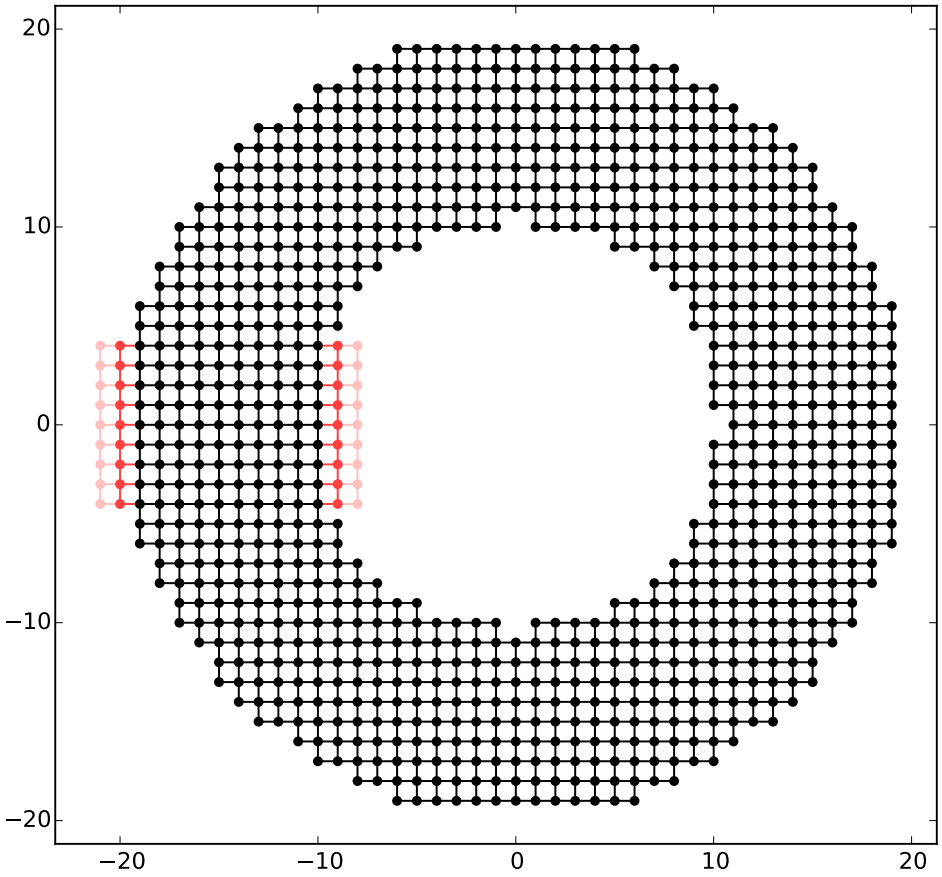


- Per default, `attach_lead` attaches the lead to the "outside" of the structure, by tracing the lead backwards, coming from infinity.

  One can also attach the lead to the inside of the structure, by providing an alternative starting point from where the lead is traced back:

  ```
  sys.attach_lead(lead1, lat(0, 0))
  ```

  starts the trace-back in the middle of the ring, resulting in the lead being attached to the inner circle:



Note that here the lead is treated as if it would pass over the other arm of the ring, without intersecting it.

2.4 Beyond transport: Band structure and closed systems

2.4.1 Band structure calculations

See also:

The complete source code of this example can be found in `tutorial/band_structure.py`

When doing transport simulations, one also often needs to know the band structure of the leads, i.e. the energies of the propagating plane waves in the leads as a function of momentum. This band structure contains information about the number of modes, their momenta and velocities.

In this example, we aim to compute the band structure of a simple tight-binding wire.

Computing band structures in Kwant is easy. Just define a lead in the usual way:

```
def make_lead(a=1, t=1.0, W=10):
    # Start with an empty lead with a single square lattice
    lat = kwant.lattice.square(a)
```

```
    sym_lead = kwant.TranslationalSymmetry((-a, 0))
    lead = kwant.Builder(sym_lead)

    # build up one unit cell of the lead, and add the hoppings
    # to the next unit cell
    for j in xrange(W):
        lead[lat(0, j)] = 4 * t

        if j > 0:
            lead[lat(0, j), lat(0, j - 1)] = -t

        lead[lat(1, j), lat(0, j)] = -t

    return lead
```

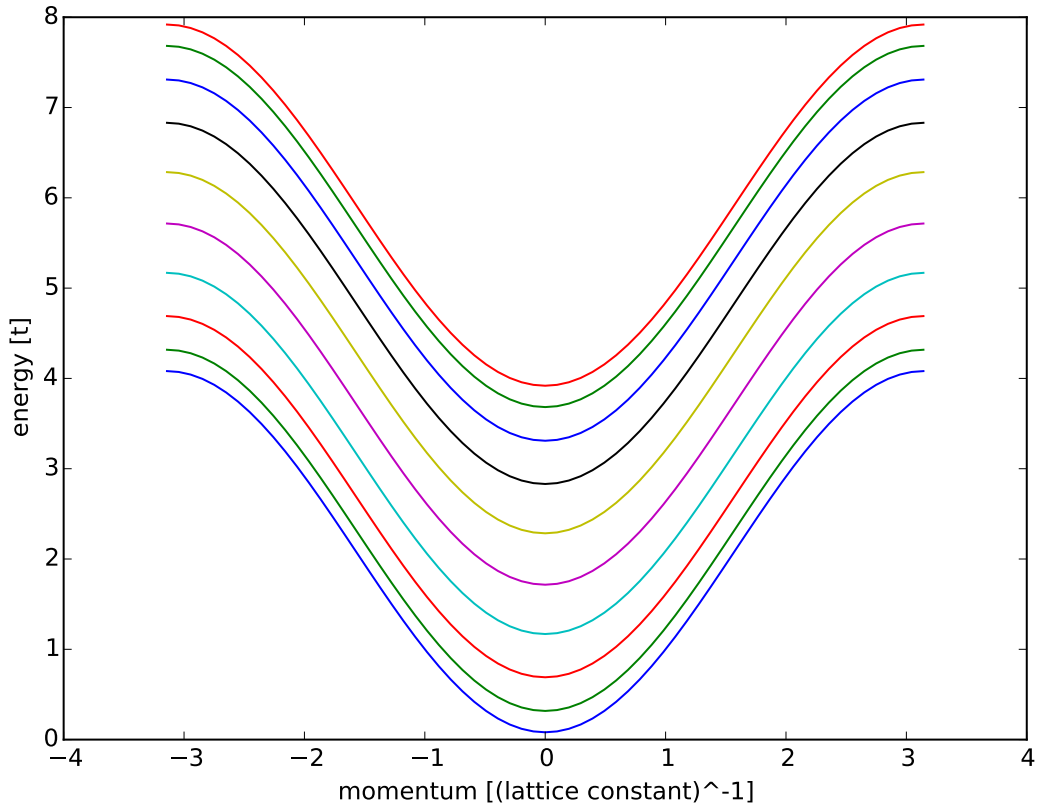
"Usual way" means defining a translational symmetry vector, as well as one unit cell of the lead, and the hoppings to neighboring unit cells. This information is enough to make the infinite, translationally invariant system needed for band structure calculations.

In the previous examples `Builder` instances like the one created above were attached as leads to the `Builder` instance of the scattering region and the latter was finalized. The thus created system contained implicitly finalized versions of the attached leads. However, now we are working with a single lead and there is no scattering region. Hence, we have to finalize the `Builder` of our sole lead explicitly.

That finalized lead is then passed to `bands`. This function calculates energies of various bands at a range of momenta and plots the calculated energies. It is really a convenience function, and if one needs to do something more profound with the dispersion relation these energies may be calculated directly using `Bands`. For now we just plot the bandstructure:

```
def main():
    lead = make_lead().finalized()
    kwant.plotter.bands(lead, show=False)
    pyplot.xlabel("momentum [(lattice constant)^-1]")
    pyplot.ylabel("energy [t]")
    pyplot.show()
```

This gives the result:



where we observe the cosine-like dispersion of the square lattice. Close to `k=0` this agrees well with the quadratic dispersion this tight-binding Hamiltonian is approximating.

2.4.2 Closed systems

See also:

The complete source code of this example can be found in `tutorial/closed_system.py`

Although Kwant is (currently) mainly aimed towards transport problems, it can also easily be used to compute properties of closed systems – after all, a closed system is nothing more than a scattering region without leads!

In this example, we compute the wave functions of a closed circular quantum dot and its spectrum as a function of magnetic field (Fock-Darwin spectrum).

To compute the eigenenergies and eigenstates, we will make use of the sparse linear algebra functionality of SciPy, which interfaces the ARPACK package:

```
import scipy.sparse.linalg as sla
```

We set up the system using the *shape*-function as in *Nontrivial shapes*, but do not add any leads:

```
lat = kwant.lattice.square(a)

sys = kwant.Builder()

# Define the quantum dot
def circle(pos):
    (x, y) = pos
    rsq = x ** 2 + y ** 2
    return rsq < r ** 2
```

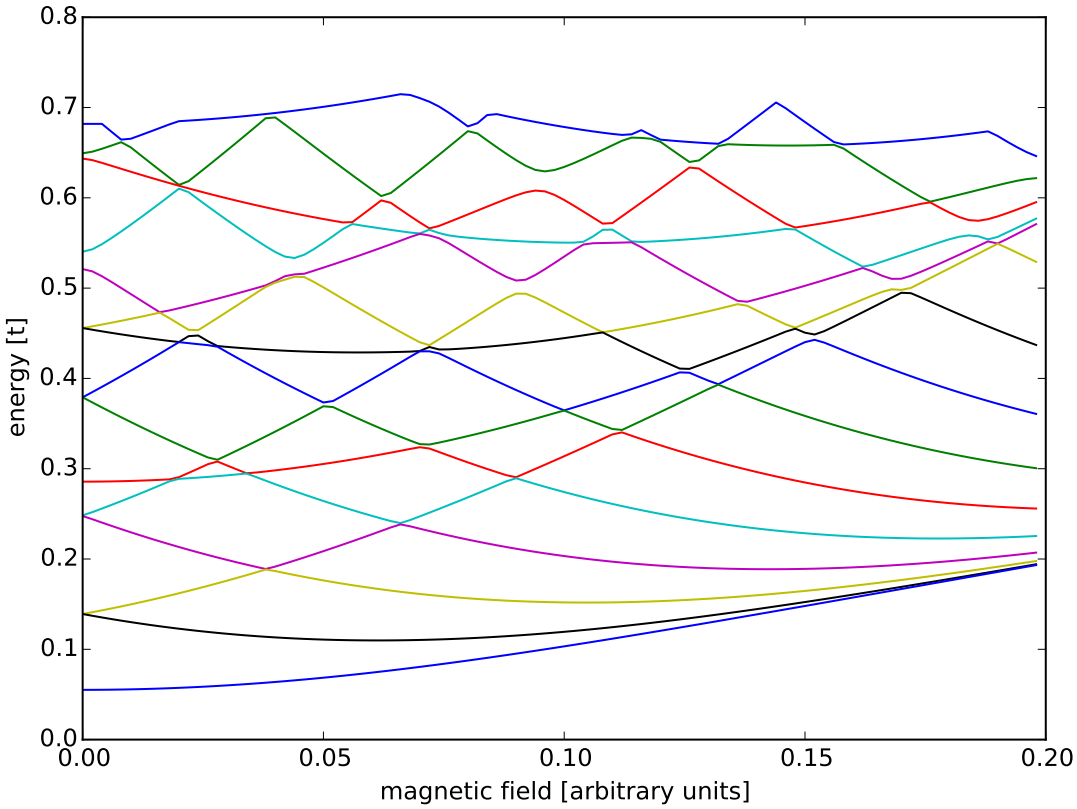
```
    def hopx(site1, site2, B=0):
        # The magnetic field is controlled by the parameter B
        y = site1.pos[1]
        return -t * exp(-1j * B * y)

    sys[lat.shape(circle, (0, 0))] = 4 * t
    # hoppings in x-direction
    sys[kwant.builder.HoppingKind((1, 0), lat, lat)] = hopx
    # hoppings in y-directions
    sys[kwant.builder.HoppingKind((0, 1), lat, lat)] = -t

    # It's a closed system for a change, so no leads
    return sys
```

We add the magnetic field using a function and a global variable as we did in the two previous tutorial. (Here, the gauge is chosen such that $A_x(y) = -By$ and $A_y = 0$.)

The spectrum can be obtained by diagonalizing the Hamiltonian of the system, which in turn can be obtained from the finalized system using `hamiltonian_submatrix`:

```
def plot_spectrum(sys, Bfields):

    # In the following, we compute the spectrum of the quantum dot
    # using dense matrix methods. This works in this toy example, as
    # the system is tiny. In a real example, one would want to use
    # sparse matrix methods

    energies = []
    for B in Bfields:
        # Obtain the Hamiltonian as a dense matrix
        ham_mat = sys.hamiltonian_submatrix(args=[B], sparse=True)

        # we only calculate the 15 lowest eigenvalues
        ev = sla.eigsh(ham_mat, k=15, which='SM', return_eigenvectors=False)

        energies.append(ev)

    pyplot.figure()
    pyplot.plot(Bfields, energies)
    pyplot.xlabel("magnetic field [arbitrary units]")
    pyplot.ylabel("energy [t]")
    pyplot.show()
```

Note that we use sparse linear algebra to efficiently calculate only a few lowest eigenvalues. Finally, we obtain the result:

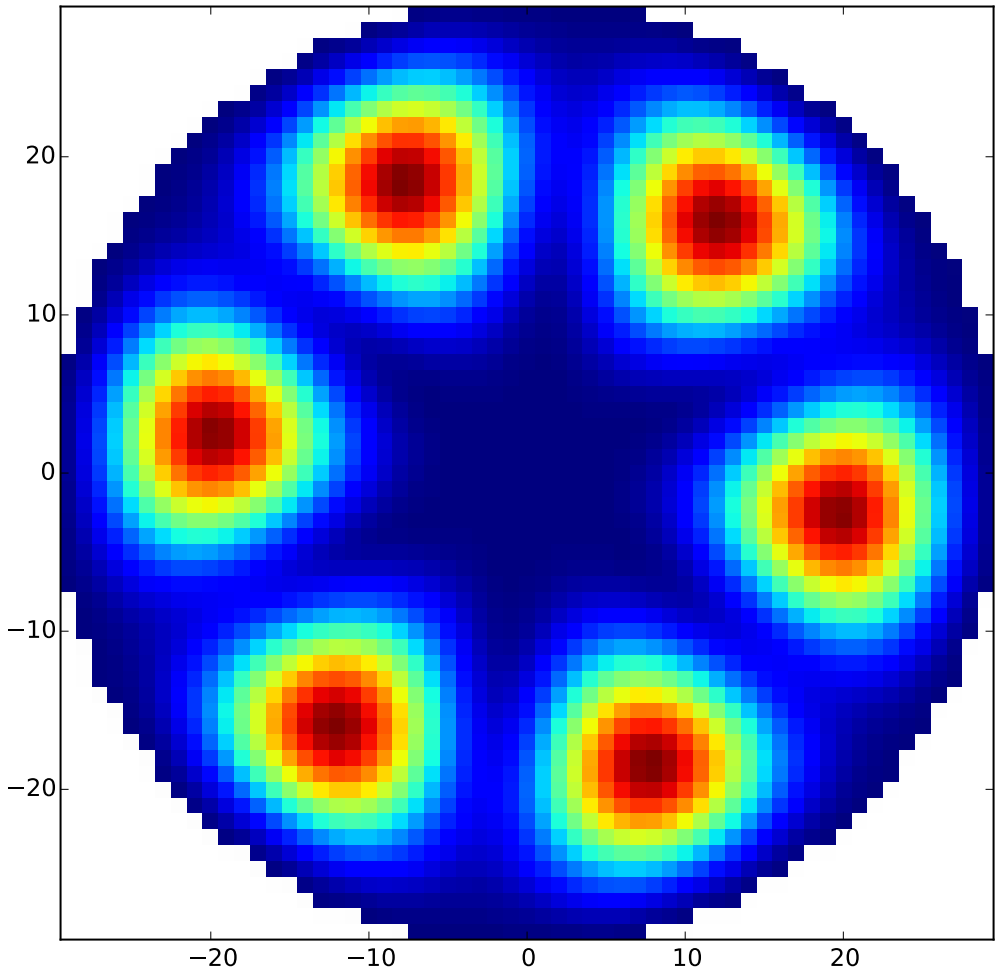


At zero magnetic field several energy levels are degenerate (since our quantum dot is rather symmetric). These degeneracies are split by the magnetic field, and the eigenenergies flow towards the Landau level energies at higher magnetic fields [6].

The eigenvectors are obtained very similarly, and can be plotted directly using `map`:

```
def plot_wave_function(sys):
    # Calculate the wave functions in the system.
    ham_mat = sys.hamiltonian_submatrix(sparse=True)
    evecs = sla.eigsh(ham_mat, k=20, which='SM')[1]

    # Plot the probability density of the 10th eigenmode.
    kwant.plotter.map(sys, np.abs(evecs[:, 9])**2,
                      colorbar=False, oversampling=1)
```

[6] Again, in this tutorial example no care was taken into choosing appropriate material parameters or units. For this reason, magnetic field is given only in "arbitrary units".



The last two arguments to `map` are optional. The first prevents a colorbar from appearing. The second, `oversampling=1`, makes the image look better for the special case of a square lattice. **Technical details**

- `hamiltonian_submatrix` can also return a sparse matrix, if the optional argument `sparse=True`. The sparse matrix is in SciPy's `scipy.sparse.coo_matrix` format, which can be easily be converted to various other sparse matrix formats (see SciPy's documentation).

2.5 Beyond square lattices: graphene

See also:

The complete source code of this example can be found in `tutorial/graphene.py`

In the following example, we are going to calculate the conductance through a graphene quantum dot with a p-n junction and two non-collinear leads. In the process, we will touch all of the topics that we have seen in the previous tutorials, but now for the honeycomb lattice. As you will see, everything carries over nicely.

We begin by defining the honeycomb lattice of graphene. This is in principle already done in `kwant.lattice.honeycomb`, but we do it explicitly here to show how to define a new lattice:

```
graphene = kwant.lattice.general([(1, 0), (sin_30, cos_30)],
                                 [(0, 0), (0, 1 / sqrt(3))])
a, b = graphene.sublattices
```

The first argument to the `general` function is the list of primitive vectors of the lattice; the second one is the coordinates of basis atoms. The honeycomb lattice has two basis atoms. Each type of basis atom by itself forms a regular lattice of the same type as well, and those *sublattices* are referenced as *a* and *b* above.

In the next step we define the shape of the scattering region (circle again) and add all lattice points using the `shape`-functionality:

```
def make_system(r=10, w=2.0, pot=0.1):

    #### Define the scattering region. ####
    # circular scattering region
    def circle(pos):
        x, y = pos
        return x ** 2 + y ** 2 < r ** 2

    sys = kwant.Builder()

    # w: width and pot: potential maximum of the p-n junction
    def potential(site):
        (x, y) = site.pos
        d = y * cos_30 + x * sin_30
        return pot * tanh(d / w)

    sys[graphene.shape(circle, (0, 0))] = potential
```

As you can see, this works exactly the same for any kind of lattice. We add the onsite energies using a function describing the p-n junction; in contrast to the previous tutorial, the potential value is this time taken from the scope of *make_system*, since we keep the potential fixed in this example.

As a next step we add the hoppings, making use of `HoppingKind`. For illustration purposes we define the hoppings ourselves instead of using `graphene.neighbors()`:

```
    hoppings = (((0, 0), a, b), ((0, 1), a, b), ((-1, 1), a, b))
```

The nearest-neighbor model for graphene contains only hoppings between different basis atoms. For this type of hoppings, it is not enough to specify relative lattice indices, but we also need to specify the proper target and source sublattices. Remember that the format of the hopping specification is `(i,j), target, source`. In the previous examples (i.e. *Matrix structure of on-site and hopping elements*) `target=source=lat`, whereas here we have to specify different sublattices. Furthermore, note that the directions given by the lattice indices *(1, 0)* and *(0, 1)* are not orthogonal anymore, since they are given with respect to the two primitive vectors `[(1, 0), (sin_30, cos_30)]`.

Adding the hoppings however still works the same way:

```
    sys[[kwant.builder.HoppingKind(*hopping) for hopping in hoppings]] = -1
```

Modifying the scattering region is also possible as before. Let's do something crazy, and remove an atom in sublattice A (which removes also the hoppings from/to this site) as well as add an additional link:

```
    del sys[a(0, 0)]
    sys[a(-2, 1), b(2, 2)] = -1
```

Note again that the conversion from a tuple *(i,j)* to site is done by the sublattices *a* and *b*.

The leads are defined almost as before:

```
    # left lead
    sym0 = kwant.TranslationalSymmetry(graphene.vec((-1, 0)))

    def lead0_shape(pos):
```

```
        x, y = pos
        return (-0.4 * r < y < 0.4 * r)

    lead0 = kwant.Builder(sym0)
    lead0[graphene.shape(lead0_shape, (0, 0))] = -pot
    lead0[[kwant.builder.HoppingKind(*hopping) for hopping in hoppings]] = -1

    # The second lead, going to the top right
    sym1 = kwant.TranslationalSymmetry(graphene.vec((0, 1)))

    def lead1_shape(pos):
        v = pos[1] * sin_30 - pos[0] * cos_30
        return (-0.4 * r < v < 0.4 * r)

    lead1 = kwant.Builder(sym1)
    lead1[graphene.shape(lead1_shape, (0, 0))] = pot
    lead1[[kwant.builder.HoppingKind(*hopping) for hopping in hoppings]] = -1
```

Note the method `vec` used in calculating the parameter for `TranslationalSymmetry`. The latter expects a real-space symmetry vector, but for many lattices symmetry vectors are more easily expressed in the natural coordinate system of the lattice. The *vec*-method is thus used to map a lattice vector to a real-space vector.

Observe also that the translational vectors `graphene.vec((-1, 0))` and `graphene.vec((0, 1))` are *not* orthogonal any more as they would have been in a square lattice – they follow the non-orthogonal primitive vectors defined in the beginning.

Later, we will compute some eigenvalues of the closed scattering region without leads. This is why we postpone attaching the leads to the system. Instead, we return the scattering region and the leads separately.

```
    return sys, [lead0, lead1]
```

The computation of some eigenvalues of the closed system is done in the following piece of code:

```
def compute_evs(sys):
    # Compute some eigenvalues of the closed system
    sparse_mat = sys.hamiltonian_submatrix(sparse=True)

    evs = sla.eigs(sparse_mat, 2)[0]
    print evs.real
```

Here we use in contrast to the previous example a sparse matrix and the sparse linear algebra functionality of SciPy.

The code for computing the band structure and the conductance is identical to the previous examples, and needs not be further explained here.

Finally, in the *main* function we make and plot the system:

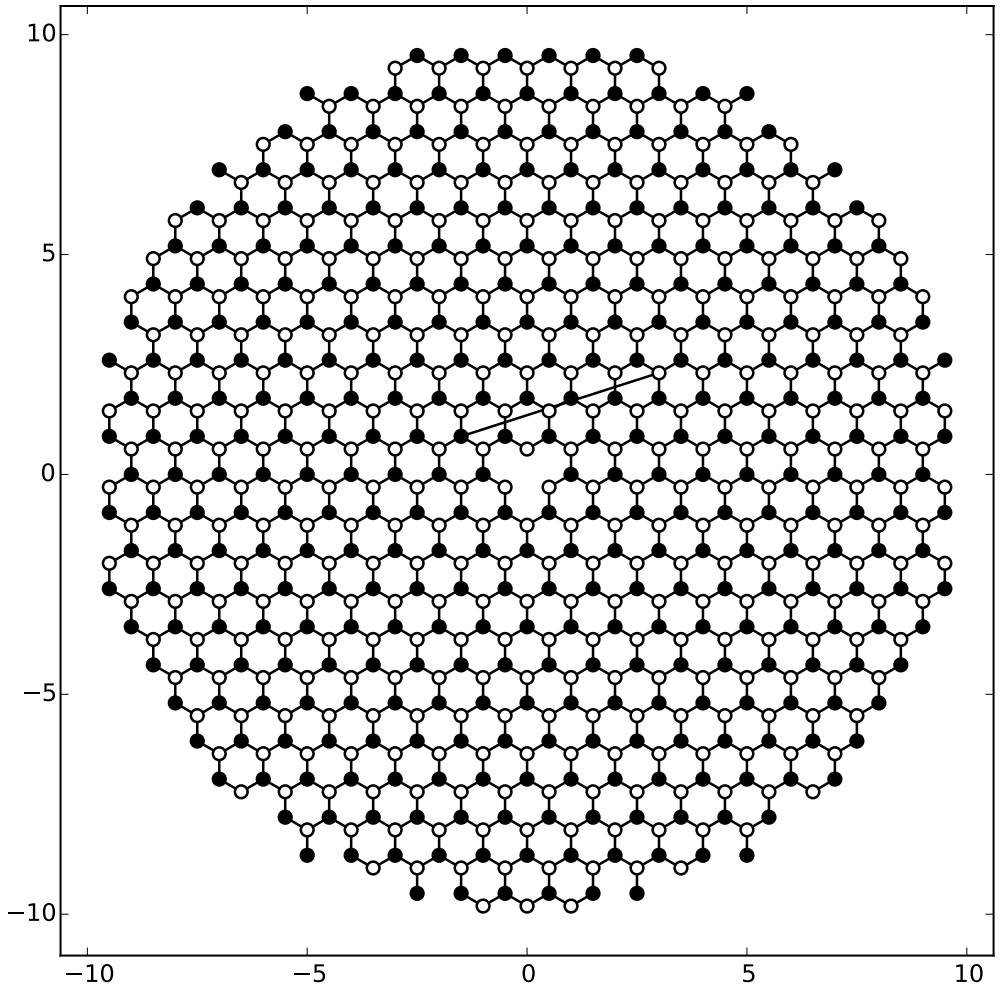
```
def main():
    pot = 0.1
    sys, leads = make_system(pot=pot)

    # To highlight the two sublattices of graphene, we plot one with
    # a filled, and the other one with an open circle:
    def family_colors(site):
        return 0 if site.family == a else 1

    # Plot the closed system without leads.
    kwant.plot(sys, site_color=family_colors, site_lw=0.1, colorbar=False)
```

We customize the plotting: we set the *site_colors* argument of `plot` to a function which returns 0 for sublattice *a* and 1 for sublattice *b*:

```
def family_colors(site):
    return 0 if site.family == a else 1
```

The function `plot` shows these values using a color scale (grayscale by default). The symbol *size* is specified in points, and is independent on the overall figure size.

Plotting the closed system gives this result:

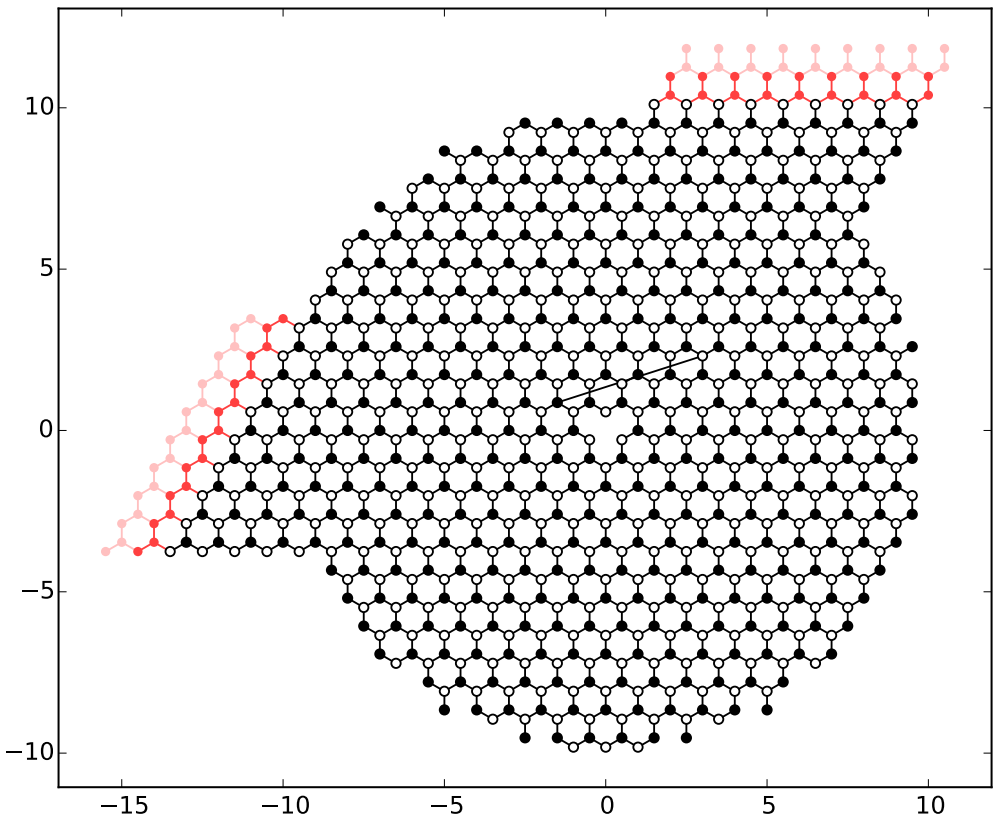


Computing the eigenvalues of largest magnitude,

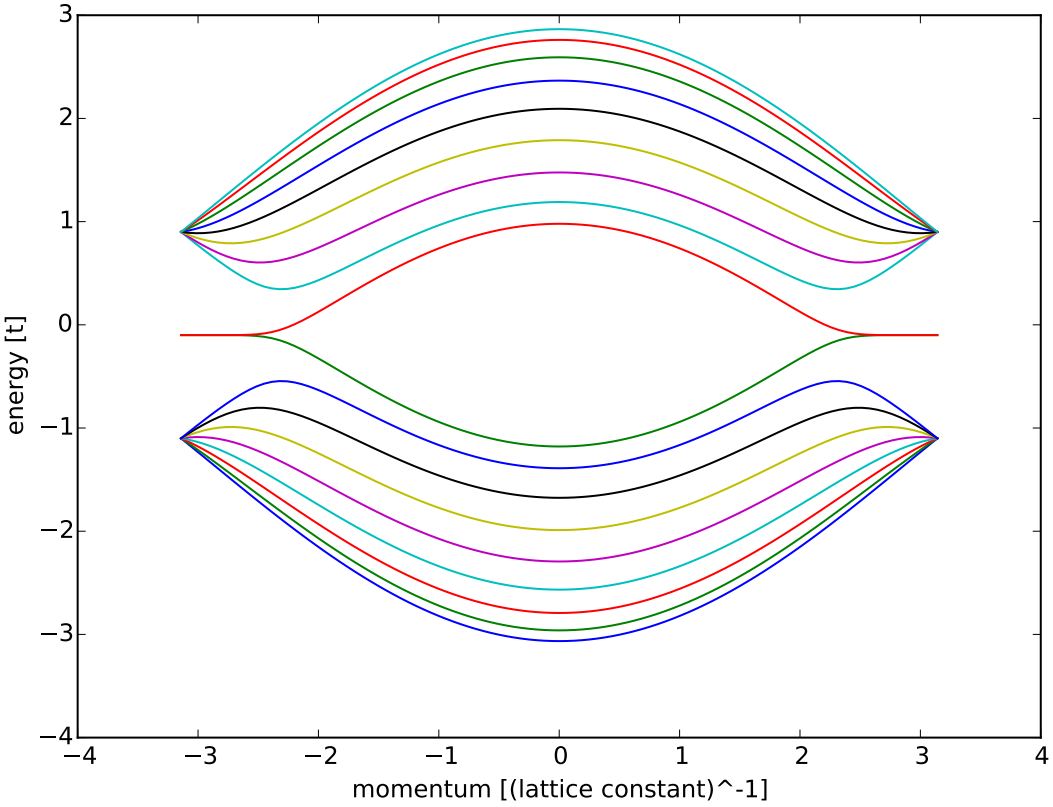
```
compute_evs(sys.finalized())
```

should yield two eigenvalues equal to `[ 3.07869311, -3.06233144]`.

The remaining code of *main* attaches the leads to the system and plots it again:

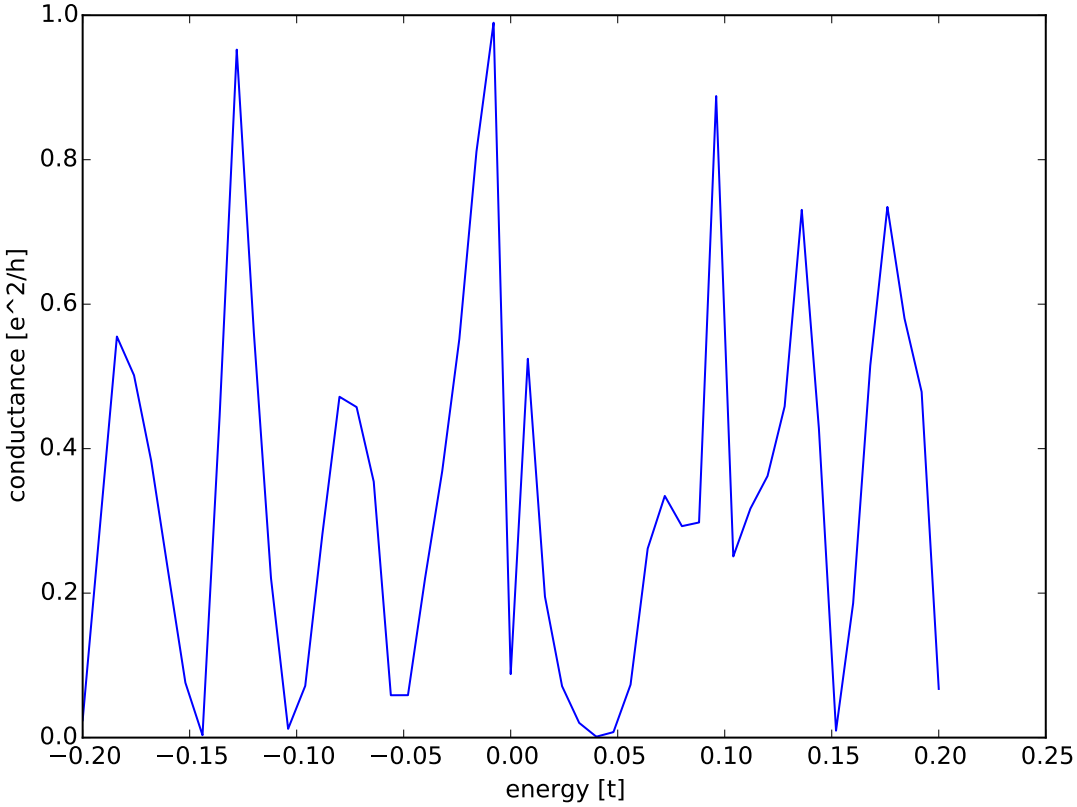


It computes the band structure of one of lead 0:



showing all the features of a zigzag lead, including the flat edge state bands (note that the band structure is not symmetric around zero energy, due to a potential in the leads).

Finally the transmission through the system is computed,



showing the typical resonance-like transmission probability through an open quantum dot **Technical details**

- In a lattice with more than one basis atom, you can always act either on all sublattices at the same time, or on a single sublattice only.

  For example, you can add lattice points for all sublattices in the current example using:

  ```
  sys[graphene.shape(...)] = ...
  ```

  or just for a single sublattice:

  ```
  sys[a.shape(...)] = ...
  ```

  and the same of course with b. Also, you can selectively remove points:

  ```
  del sys[graphene.shape(...)]
  del sys[a.shape(...)]
  ```

  where the first line removes points in both sublattices, whereas the second line removes only points in one sublattice.

2.6 Superconductors: orbital vs. lattice degrees of freedom

This example deals with superconductivity on the level of the Bogoliubov-de Gennes (BdG) equation. In this framework, the Hamiltonian is given as

$$H = \begin{pmatrix} H_0 - \mu & \Delta \\ \Delta^\dagger & \mu - \mathcal{T}H\mathcal{T}^{-1} \end{pmatrix}$$

where H_0 is the Hamiltonian of the system without superconductivity, μ the chemical potential, Δ the superconducting order parameter, and $\mathcal{T}$ the time-reversal operator. The BdG Hamiltonian introduces electron and hole

degrees of freedom (an artificial doubling - be aware of the fact that electron and hole excitations are related!), which we now implement in Kwant.

For this we restrict ourselves to a simple spinless system without magnetic field, so that Δ is just a number (which we choose real), and $\mathcal{T}H\mathcal{T}^{-1} = H_0^* = H_0$.

2.6.1 "Orbital description": using matrices

See also:

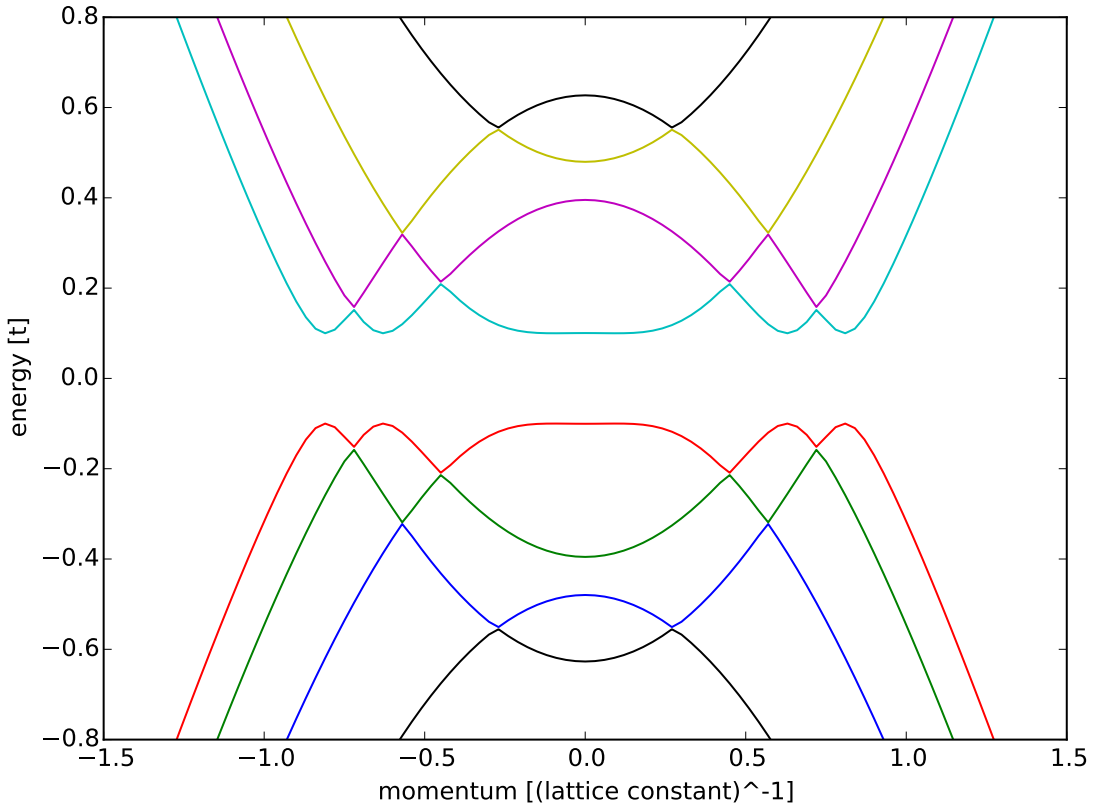
The complete source code of this example can be found in `tutorial/superconductor_band_structure.py`

We begin by computing the band structure of a superconducting wire. The most natural way to implement the BdG Hamiltonian is by using a 2x2 matrix structure for all Hamiltonian matrix elements:

```
tau_x = tinyarray.array([[0, 1], [1, 0]])
tau_z = tinyarray.array([[1, 0], [0, -1]])


def make_lead(a=1, t=1.0, mu=0.7, Delta=0.1, W=10):
    # Start with an empty lead with a single square lattice
    lat = kwant.lattice.square(a)

    sym_lead = kwant.TranslationalSymmetry((-a, 0))
    lead = kwant.Builder(sym_lead)

    # build up one unit cell of the lead, and add the hoppings
    # to the next unit cell
    for j in xrange(W):
        lead[lat(0, j)] = (4 * t - mu) * tau_z + Delta * tau_x

        if j > 0:
            lead[lat(0, j), lat(0, j - 1)] = -t * tau_z

        lead[lat(1, j), lat(0, j)] = -t * tau_z

    return lead
```

As you see, the example is syntactically equivalent to our *spin example*, the only difference is now that the Pauli matrices act in electron-hole space.

Computing the band structure then yields the result



We clearly observe the superconducting gap in the spectrum. That was easy, wasn't it?

2.6.2 "Lattice description": using different lattices

See also:

The complete source code of this example can be found in `tutorial/superconductor_transport.py`

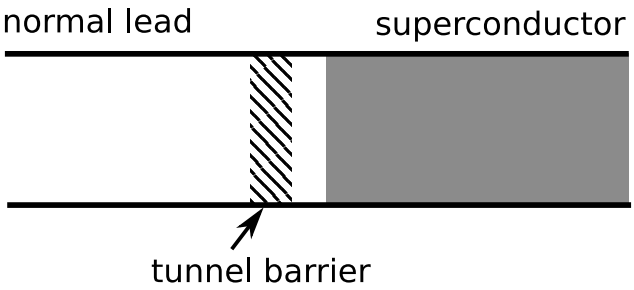
While it seems most natural to implement the BdG Hamiltonian using a 2x2 matrix structure for the matrix elements of the Hamiltonian, we run into a problem when we want to compute electronic transport in a system consisting of a normal and a superconducting lead: Since electrons and holes carry charge with opposite sign, we need to separate electron and hole degrees of freedom in the scattering matrix. In particular, the conductance of a N-S-junction is given as

$$G = \frac{e^2}{h}(N - R_{\text{ee}} + R_{\text{he}}),$$

where N is the number of channels in the normal lead, and R_{ee} the total probability of reflection from electrons to electrons in the normal lead, and R_{eh} the total probability of reflection from electrons to holes in the normal lead. However, the current version of Kwant does not allow for an easy and elegant partitioning of the scattering matrix in these two degrees of freedom [7].

In the following, we will circumvent this problem by introducing separate "leads" for electrons and holes, making use of different lattices. The system we consider consists of a normal lead on the left, a superconductor on the right, and a tunnel barrier in between:

[7] Well, there is a not so elegant way to do it still. See the technical details



As already mentioned above, we begin by introducing two different square lattices representing electron and hole degrees of freedom:

```
def make_system(a=1, W=10, L=10, barrier=1.5, barrierpos=(3, 4),
                mu=0.4, Delta=0.1, Deltapos=4, t=1.0):
    # Start with an empty tight-binding system and two square lattices,
    # corresponding to electron and hole degree of freedom
    lat_e = kwant.lattice.square(a, name='e')
    lat_h = kwant.lattice.square(a, name='h')
```

Note that since these two lattices have identical spatial parameters, the argument `name` to `square` has to be different. Any diagonal entry (kinetic energy, potentials, ...) in the BdG Hamiltonian corresponds to on-site energies or hoppings within the *same* lattice, whereas any off-diagonal entry (essentially, the superconducting order parameter Δ) corresponds to a hopping between *different* lattices:

```
    sys = kwant.Builder()

    #### Define the scattering region. ####
    sys[(lat_e(x, y) for x in range(L) for y in range(W))] = 4 * t - mu
    sys[(lat_h(x, y) for x in range(L) for y in range(W))] = mu - 4 * t

    # the tunnel barrier
    sys[(lat_e(x, y) for x in range(barrierpos[0], barrierpos[1])
         for y in range(W))] = 4 * t + barrier - mu
    sys[(lat_h(x, y) for x in range(barrierpos[0], barrierpos[1])
         for y in range(W))] = mu - 4 * t - barrier

    # hoppings for both electrons and holes
    sys[lat_e.neighbors()] = -t
    sys[lat_h.neighbors()] = t

    # Superconducting order parameter enters as hopping between
    # electrons and holes
    sys[((lat_e(x, y), lat_h(x, y)) for x in range(Deltapos, L)
         for y in range(W))] = Delta
```

Note that the tunnel barrier is added by overwriting previously set on-site matrix elements.

Note further, that in the code above, the superconducting order parameter is nonzero only in a part of the scattering region. Consequently, we have added hoppings between electron and hole lattices only in this region, they remain uncoupled in the normal part. We use this fact to attach purely electron and hole leads (comprised of only electron *or* hole lattices) to the system:

```
    # Symmetry for the left leads.
    sym_left = kwant.TranslationalSymmetry((-a, 0))

    # left electron lead
    lead0 = kwant.Builder(sym_left)
    lead0[(lat_e(0, j) for j in xrange(W))] = 4 * t - mu
    lead0[lat_e.neighbors()] = -t

    # left hole lead
    lead1 = kwant.Builder(sym_left)
```

```
    lead1[(lat_h(0, j) for j in xrange(W))] = mu - 4 * t
    lead1[lat_h.neighbors()] = t
```

This separation into two different leads allows us then later to compute the reflection probablities between electrons and holes explicitely.

On the superconducting side, we cannot do this separation, and can only define a single lead coupling electrons and holes (The += operator adds all the sites and hoppings present in one builder to another):

```
    sym_right = kwant.TranslationalSymmetry((a, 0))
    lead2 = kwant.Builder(sym_right)
    lead2 += lead0
    lead2 += lead1
    lead2[((lat_e(0, j), lat_h(0, j)) for j in xrange(W))] = Delta
```

We now have on the left side two leads that are sitting in the same spatial position, but in different lattice spaces. This ensures that we can still attach all leads as before:

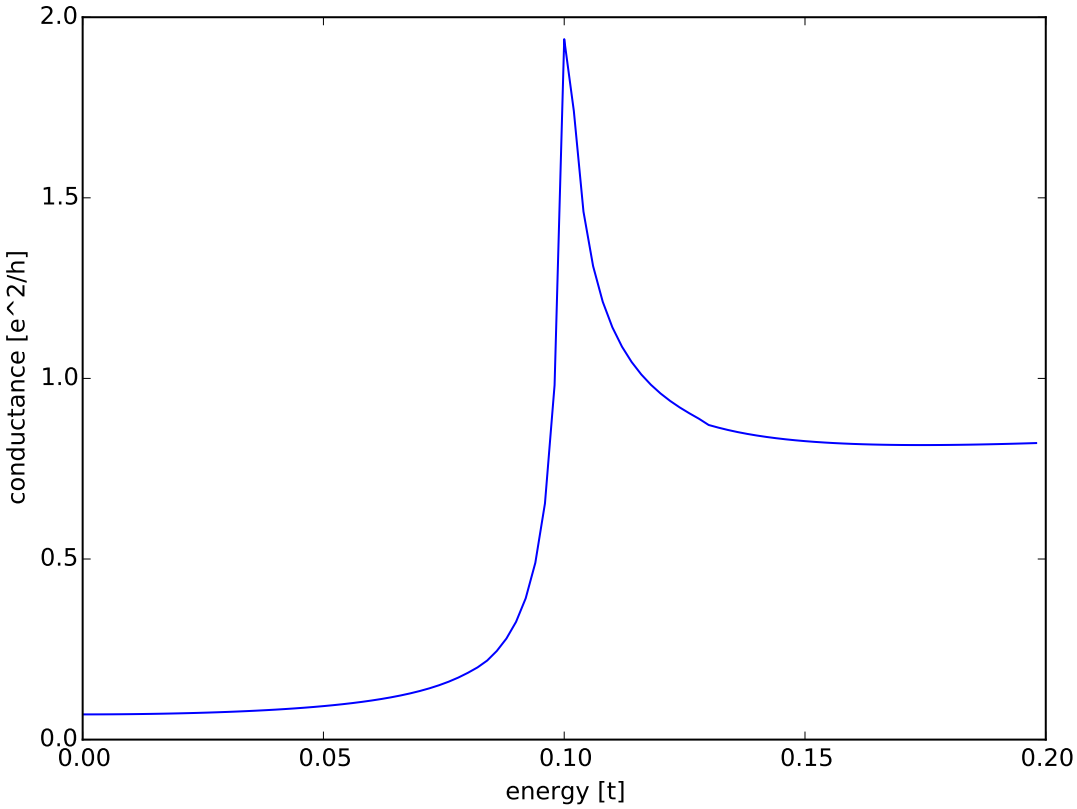
```
    sys.attach_lead(lead0)
    sys.attach_lead(lead1)
    sys.attach_lead(lead2)

    return sys
```

When computing the conductance, we can now extract reflection from electrons to electrons as `smatrix.transmission(0, 0)` (Don't get confused by the fact that it says `transmission` – transmission into the same lead is reflection), and reflection from electrons to holes as `smatrix.transmission(1, 0)`, by virtue of our electron and hole leads:

```
def plot_conductance(sys, energies):
    # Compute conductance
    data = []
    for energy in energies:
        smatrix = kwant.smatrix(sys, energy)
        # Conductance is N - R_ee + R_he
        data.append(smatrix.submatrix(0, 0).shape[0] -
                    smatrix.transmission(0, 0) +
                    smatrix.transmission(1, 0))
```

Note that `smatrix.submatrix(0,0)` returns the block concerning reflection within (electron) lead 0, and from its size we can extract the number of modes N.

Finally, for the default parameters, we obtain the following result:



We a see a conductance that is proportional to the square of the tunneling probability within the gap, and proportional to the tunneling probability above the gap. At the gap edge, we observe a resonant Andreev reflection.
Technical details

- If you are only interested in particle (thermal) currents you do not need to define separate electron and hole leads. In this case, you do not need to distinguish them. Still, separating the leads into electron and hole degrees of freedom makes the lead calculation in the solving phase more efficient.
- It is in fact possible to separate electron and hole degrees of freedom in the scattering matrix, even if one uses matrices for these degrees of freedom. In the solve step, `smatrix` returns an array containing the transverse wave functions of the lead modes (in *SMatrix.lead_info*. By inspecting the wave functions, electron and hole wave functions can be distinguished (they only have entries in either the electron part *or* the hole part. If you encounter modes with entries in both parts, you hit a very unlikely situation in which the standard procedure to compute the modes gave you a superposition of electron and hole modes. That is still OK for computing particle current, but not for electrical current).

2.7 Plotting Kwant systems and data in various styles

The plotting functionality of Kwant has been used extensively (through `plot` and `map`) in the previous tutorials. In addition to this basic use, `plot` offers many options to change the plotting style extensively. It is the goal of this tutorial to show how these options can be used to achieve various very different objectives.

2.7.1 2D example: graphene quantum dot

See also:

The complete source code of this example can be found in `tutorial/plot_graphene.py`

We begin by first considering a circular graphene quantum dot (similar to what has been used in parts of the tutorial *Beyond square lattices: graphene*.) In contrast to previous examples, we will also use hoppings beyond next-nearest neighbors:

```
lat = kwant.lattice.honeycomb()
a, b = lat.sublattices

def make_system(r=8, t=-1, tp=-0.1):

    def circle(pos):
        x, y = pos
        return x**2 + y**2 < r**2

    sys = kwant.Builder()
    sys[lat.shape(circle, (0, 0))] = 0
    sys[lat.neighbors()] = t
    sys.eradicate_dangling()
    if tp:
        sys[lat.neighbors(2)] = tp

    return sys
```

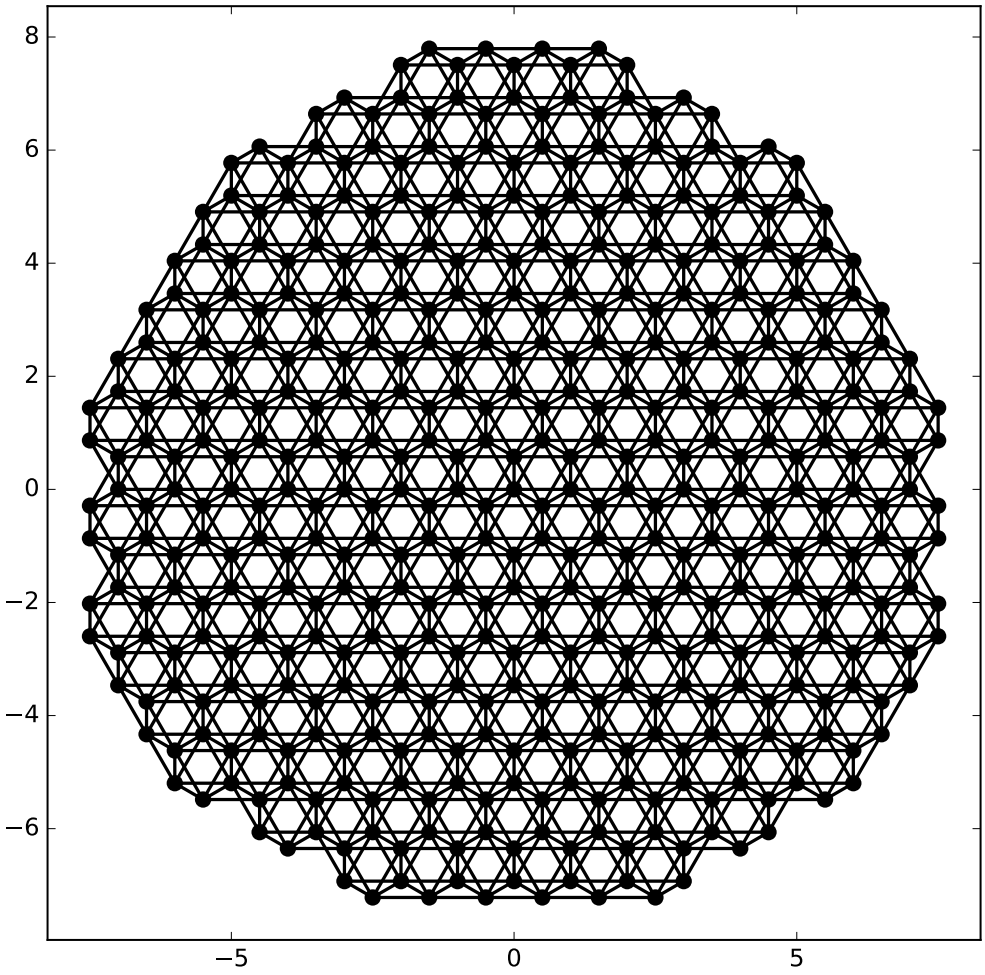
Note that adding hoppings hoppings to the n-th nearest neighbors can be simply done by passing n as an argument to `neighbors`. Also note that we use the method `eradicate_dangling` to get rid of single atoms sticking out of the shape. It is necessary to do so *before* adding the next-nearest-neighbor hopping [8].

Of course, the system can be plotted simply with default settings:

```
def plot_system(sys):
    kwant.plot(sys)
```

However, due to the richer structure of the lattice, this results in a rather busy plot:

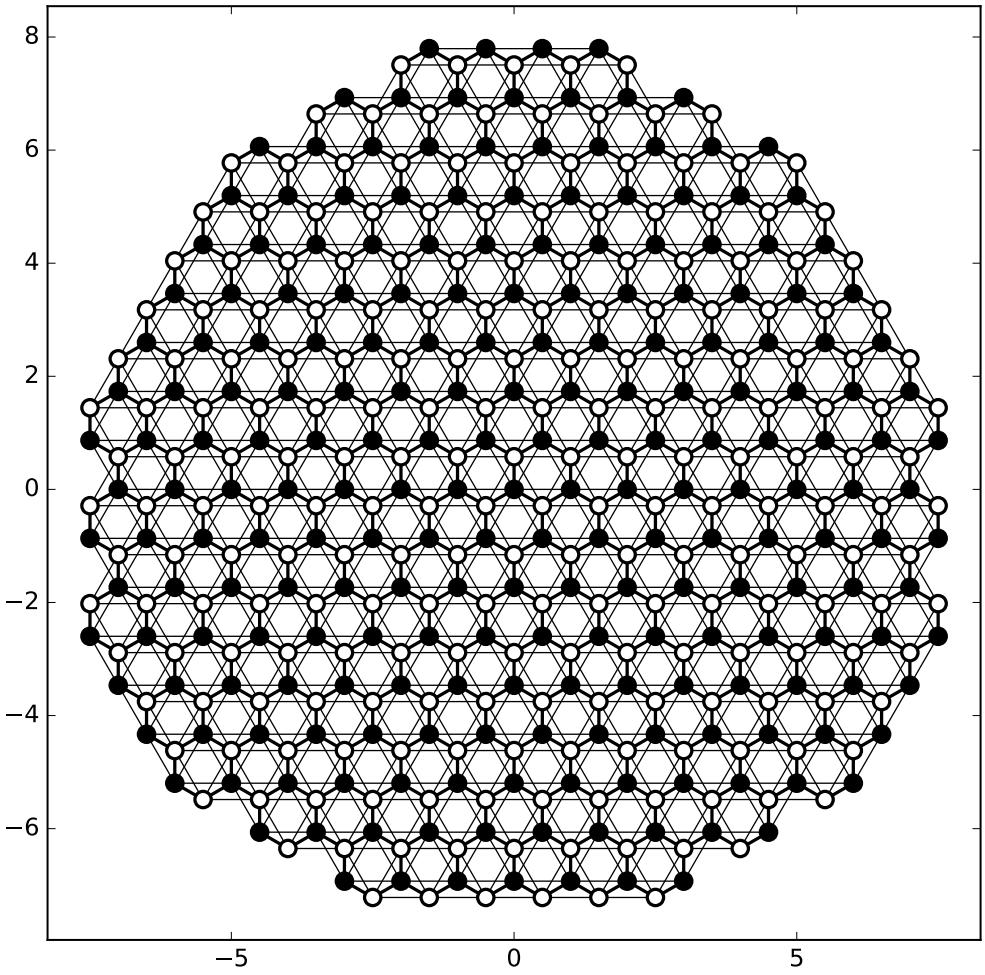
[8] A dangling site is defined as having only one hopping connecting it to the rest. With next-nearest-neighbor hopping also all sites that are dangling with only nearest-neighbor hopping have more than one hopping.



A much clearer plot can be obtained by using different colors for both sublattices, and by having different line widths for different hoppings. This can be achieved by passing a function to the arguments of `plot`, instead of a constant. For properties of sites, this must be a function taking one site as argument, for hoppings a function taking the start end end site of hopping as arguments:

```
def family_color(site):
    return 'black' if site.family == a else 'white'

def hopping_lw(site1, site2):
    return 0.04 if site1.family == site2.family else 0.1

kwant.plot(sys, site_lw=0.1, site_color=family_color, hop_lw=hopping_lw)
```

Note that since we are using an unfinalized Builder, a `site` is really an instance of `Site`. With these adjustments we arrive at a plot that carries the same information, but is much easier to interpret:



Apart from plotting the *system* itself, `plot` can also be used to plot *data* living on the system.

As an example, we now compute the eigenstates of the graphene quantum dot and intend to plot the wave function probability in the quantum dot. For aesthetic reasons (the wave functions look a bit nicer), we restrict ourselves to nearest-neighbor hopping. Computing the wave functions is done in the usual way (note that for a large-scale system, one would probably want to use sparse linear algebra):

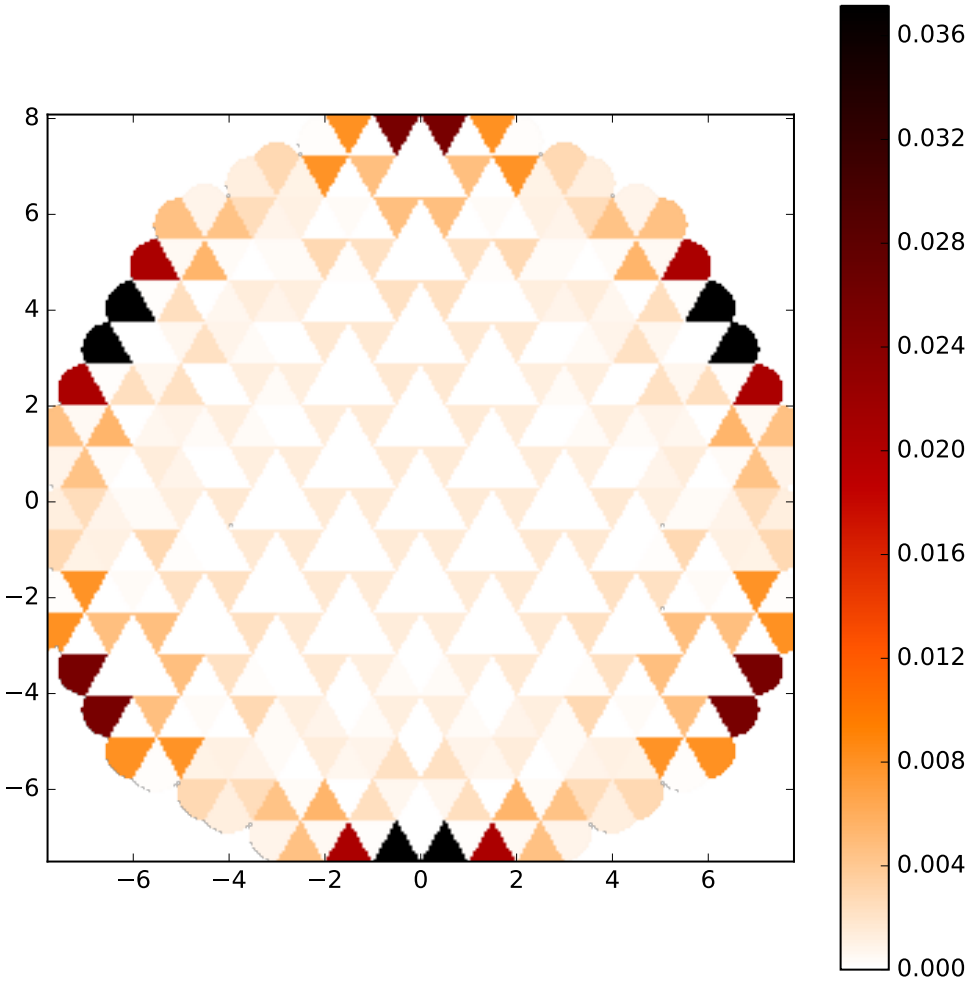
```
def plot_data(sys, n):
    import scipy.linalg as la

    sys = sys.finalized()
    ham = sys.hamiltonian_submatrix()
    evecs = la.eigh(ham)[1]

    wf = abs(evecs[:, n])**2
```

In most cases, to plot the wave function probability, one wouldn't use `plot`, but rather `map`. Here, we plot the *n*-th wave function using it:

```
    kwant.plotter.map(sys, wf, oversampling=10, cmap='gist_heat_r')
```

This results in a standard pseudocolor plot, showing in this case (`n=225`) a graphene edge state, i.e. a wave

function mostly localized at the zigzag edges of the quantum dot.



However although in general preferable, `map` has a few deficiencies for this small system: For example, there are a few distortions at the edge of the dot. (This cannot be avoided in the type of interpolation used in `map`). However, we can also use `plot` to achieve a similar, but smoother result.

For this note that `plot` can also take an array of floats (or function returning floats) as value for the *site_color* argument (the same holds for the hoppings). Via the colormap specified in *cmap* these are mapped to color, just as `map` does! In addition, we can also change the symbol shape depending on the sublattice. With a triangle pointing up and down on the respective sublattice, the symbols used by plot fill the space completely:

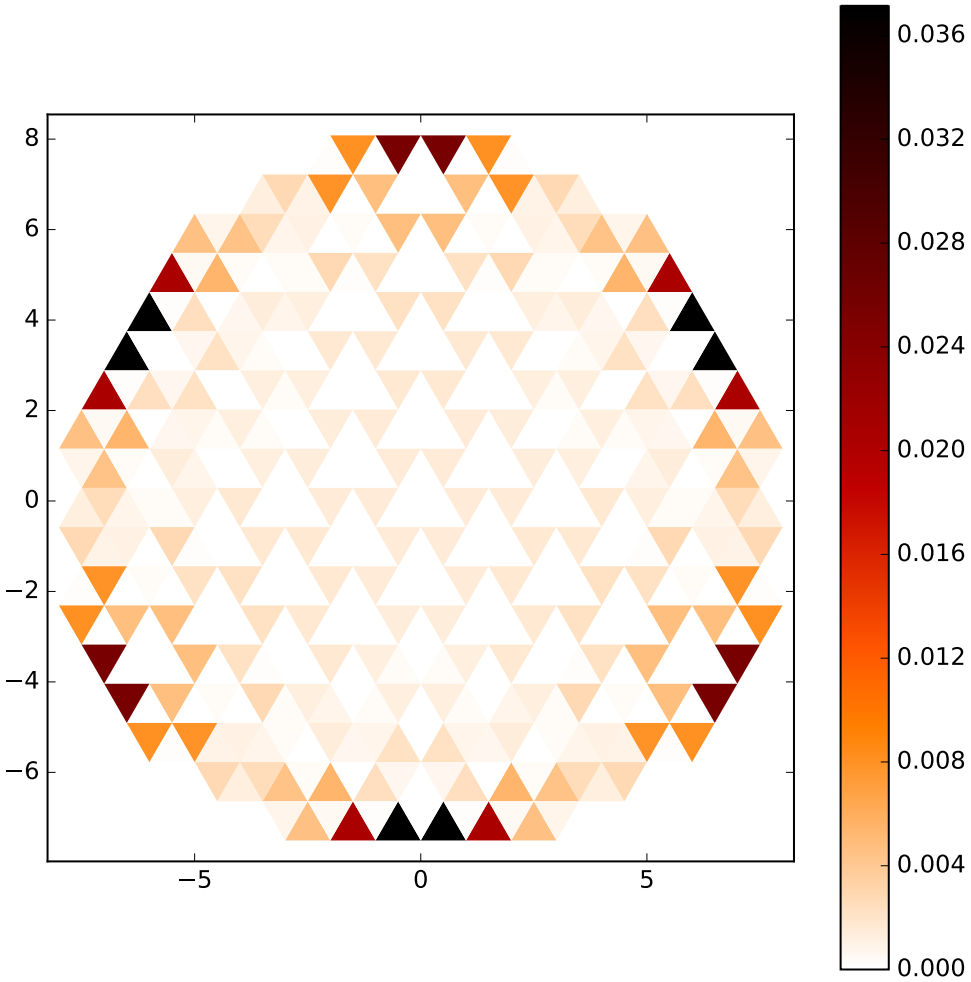
```
def family_shape(i):
    site = sys.sites[i]
    return ('p', 3, 180) if site.family == a else ('p', 3, 0)

def family_color(i):
    return 'black' if sys.site(i).family == a else 'white'

kwant.plot(sys, site_color=wf, site_symbol=family_shape,
           site_size=0.5, hop_lw=0, cmap='gist_heat_r')
```

Note that with `hop_lw=0` we deactivate plotting the hoppings (that would not serve any purpose here). More-

over, `site_size=0.5` guarantees that the two different types of triangles touch precisely: By default, `plot` takes all sizes in units of the nearest-neighbor spacing. `site_size=0.5` thus means half the distance between neighboring sites (and for the triangles this is interpreted as the radius of the inner circle).

Finally, note that since we are dealing with a finalized system now, a site *i* is represented by an integer. In order to obtain the original `Site`, `sys.site(i)` can be used.

With this we arrive at



with the same information as `map`, but with a cleaner look.

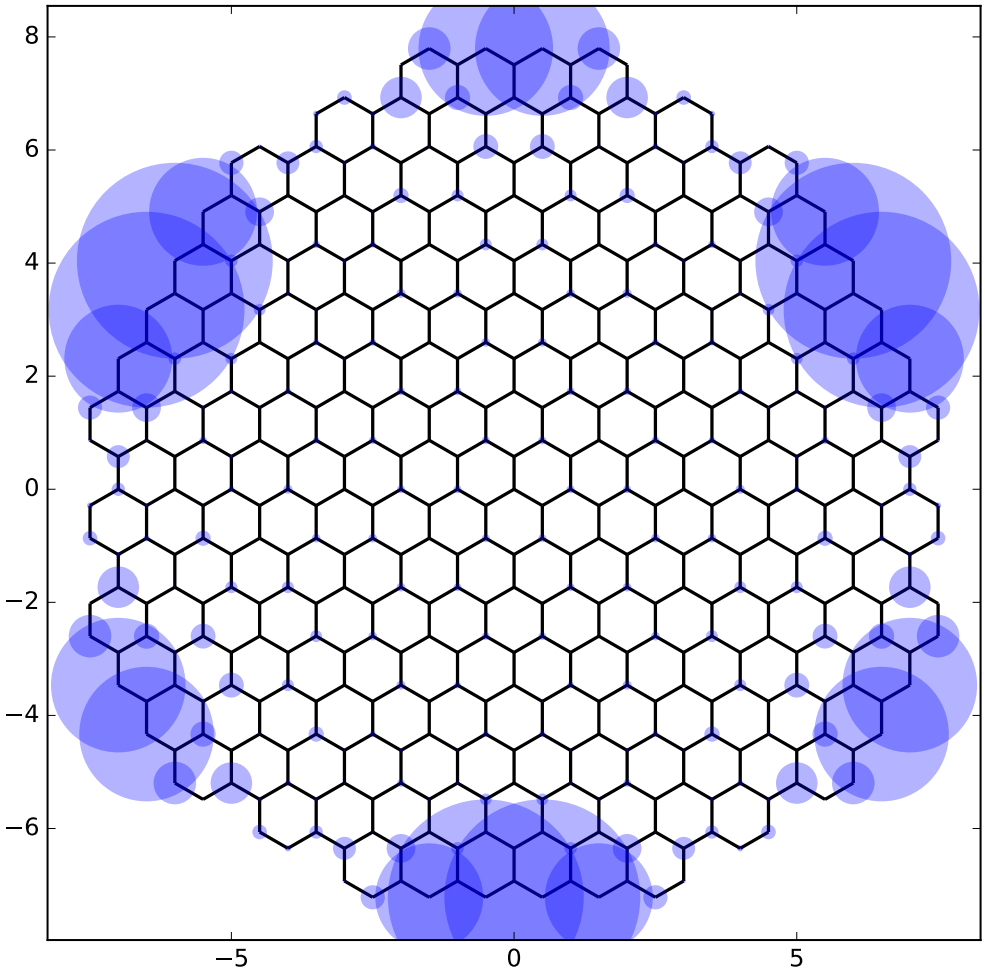
The way how data is presented of course influences what features of the data are best visible in a given plot. With `plot` one can easily go beyond pseudocolor-like plots. For example, we can represent the wave function probability using the symbols itself:

```
def site_size(i):
    return 3 * wf[i] / wf.max()

kwant.plot(sys, site_size=site_size, site_color=(0, 0, 1, 0.3),
           hop_lw=0.1)
```

Here, we choose the symbol size proportional to the wave function probability, while the site color is transparent to also allow for overlapping symbols to be visible. The hoppings are also plotted in order to show the underlying

lattice.

With this, we arrive at



which shows the edge state nature of the wave function most clearly.

2.7.2 3D example: zincblende structure

See also:

The complete source code of this example can be found in `tutorial/plot_zincblende.py`

Zincblende is a very common crystal structure of semiconductors. It is a face-centered cubic crystal with two inequivalent atoms in the unit cell (i.e. two different types of atoms, unlike diamond which has the same crystal structure, but two equivalent atoms per unit cell).

It is very easily generated in Kwant with `kwant.lattice.general`:

```
lat = kwant.lattice.general([(0, 0.5, 0.5), (0.5, 0, 0.5), (0.5, 0.5, 0)],
                            [(0, 0, 0), (0.25, 0.25, 0.25)])
a, b = lat.sublattices
```

Note how we keep references to the two different sublattices for later use.

A three-dimensional structure is created as easily as in two dimensions, by using the *shape*-functionality:

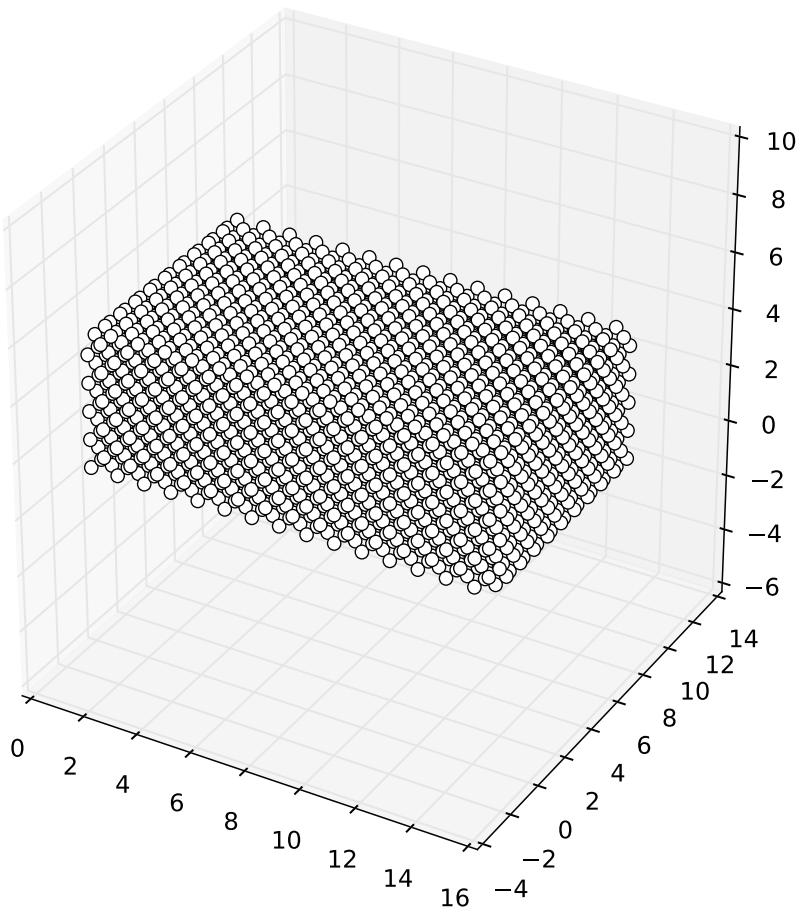
```
def make_cuboid(a=15, b=10, c=5):
    def cuboid_shape(pos):
        x, y, z = pos
        return 0 <= x < a and 0 <= y < b and 0 <= z < c

    sys = kwant.Builder()
    sys[lat.shape(cuboid_shape, (0, 0, 0))] = None
    sys[lat.neighbors()] = None

    return sys
```

We restrict ourselves here to a simple cuboid, and do not bother to add real values for onsite and hopping energies, but only the placeholder `None` (in a real calculation, several atomic orbitals would have to be considered).

`plot` can plot 3D systems just as easily as its two-dimensional counterparts:

```
    sys = make_cuboid()

    kwant.plot(sys)
```

resulting in



You might notice that the standard options for plotting are quite different in 3D than in 2D. For example, by default hoppings are not printed, but sites are instead represented by little "balls" touching each other (which is achieved by a default `site_size=0.5`). In fact, this style of plotting 3D shows quite decently the overall geometry of the system.

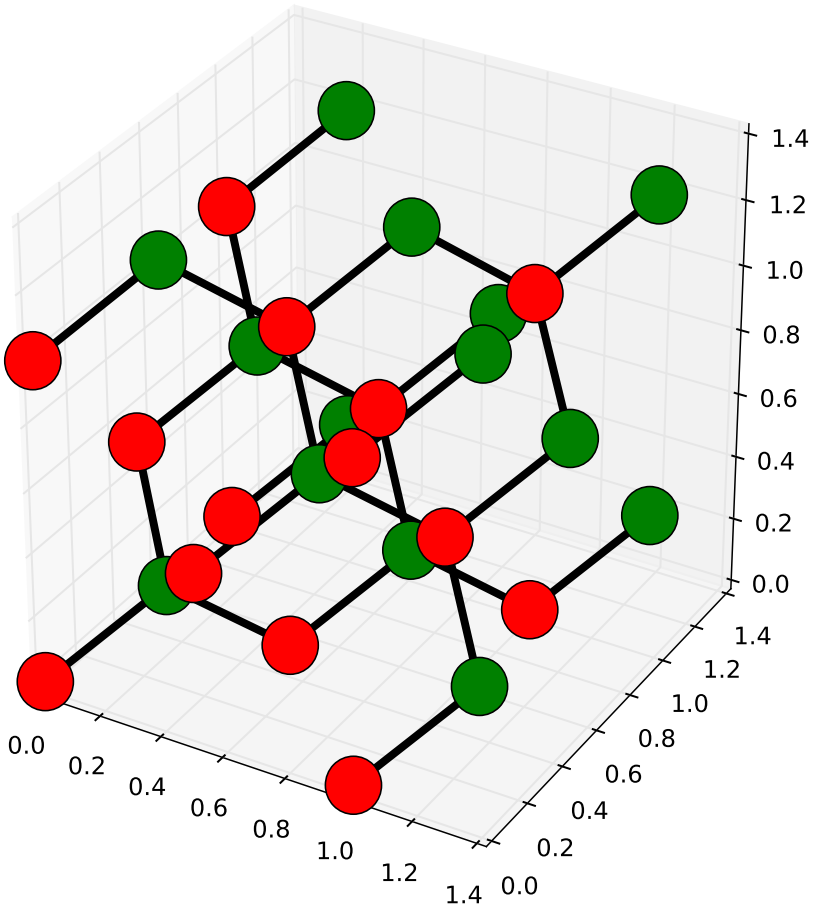
When plotting into a window, the 3D plots can also be rotated and scaled arbitrarily, allowing for a good inspection of the geometry from all sides.

Note: Interactive 3D plots usually do not have the proper aspect ratio, but are a bit squashed. This is due to bugs in matplotlib's 3D plotting module that does not properly honor the corresponding arguments. By resizing the plot window however one can manually adjust the aspect ratio.

Also for 3D it is possible to customize the plot. For example, we can explicitly plot the hoppings as lines, and color sites differently depending on the sublattice:

```
sys = make_cuboid(a=1.5, b=1.5, c=1.5)

def family_colors(site):
    return 'r' if site.family == a else 'g'

kwant.plot(sys, site_size=0.18, site_lw=0.01, hop_lw=0.05,
           site_color=family_colors)
```

which results in a 3D plot that allows to interactively (when plotted in a window) explore the crystal structure:



Hence, a few lines of code using Kwant allow to explore all the different crystal lattices out there!

Note:

- The 3D plots are in fact only *fake* 3D. For example, sites will always be plotted above hoppings (this is due to the limitations of matplotlib's 3d module)
- Plotting hoppings in 3D is inherently much slower than plotting sites. Hence, this is not done by default.

CHAPTER

THREE

CORE MODULES

The following modules of Kwant are used directly most frequently.

3.1 `kwant` – Top level package

For convenience, short names are provided for a few widely used objects from the sub-packages. Otherwise, this package has only very limited functionality of its own.

3.1.1 Generic functionality

The version of Kwant is available under the name `__version__`.

`KwantDeprecationWarning`	Class of warnings about a deprecated feature of Kwant.

kwant.KwantDeprecationWarning

exception `kwant.`**`KwantDeprecationWarning`**

Bases: `exceptions.Warning`

Class of warnings about a deprecated feature of Kwant.

DeprecationWarning has been made invisible by default in Python 2.7 in order to not confuse non-developer users with warnings that are not relevant to them. In the case of Kwant, by far most users are developers, so we feel that a KwantDeprecationWarning that is visible by default is useful.

3.1.2 From `kwant.builder`

`Builder`([symmetry])	A tight binding system defined on a graph.
`HoppingKind`(delta, family_a[, family_b])	A pattern for matching hoppings.

3.1.3 From `kwant.lattice`

`TranslationalSymmetry`(*periods)	A translational symmetry defined in real space.

3.1.4 From `kwant.plotter`

`plot`(sys[, num_lead_cells, unit, ...])	Plot a system in 2 or 3 dimensions.

3.1.5 From `kwant.solvers.default`

greens_function(sys[, energy, args, ...])	Compute the retarded Green's function of the system between its leads.
ldos(sys[, energy, args, check_hermiticity])	Calculate the local density of states of a system at a given energy.
smatrix(sys[, energy, args, out_leads, ...])	Compute the scattering matrix of a system.
wave_function(sys[, energy, args, ...])	Return a callable object for the computation of the wave function inside the scatte

3.2 `kwant.builder` – High-level construction of systems

3.2.1 Types

Builder([symmetry])	A tight binding system defined on a graph.
Site	A site, member of a SiteFamily.
HoppingKind(delta, family_a[, family_b])	A pattern for matching hoppings.
SimpleSiteFamily([name])	A site family used as an example and for testing.
BuilderLead(builder, interface)	A lead made from a Builder with a spatial symmetry.
SelfEnergyLead(selfenergy_func, interface)	A general lead defined by its self energy.
ModesLead(modes_func, interface)	A general lead defined by its modes wave functions.

kwant.builder.Builder

class kwant.builder.**Builder**(*symmetry=None*)

Bases: object

A tight binding system defined on a graph.

This is one of the central types in Kwant. It is used to construct tight binding systems in a flexible way.

The nodes of the graph are Site instances. The edges, i.e. the hoppings, are pairs (2-tuples) of sites. Each node and each edge has a value associated with it. The values associated with nodes are interpreted as on-site Hamiltonians, the ones associated with edges as hopping integrals.

To make the graph accessible in a way that is natural within the Python language it is exposed as a *mapping* (much like a built-in Python dictionary). Keys are sites or hoppings. Values are 2d arrays (e.g. NumPy or Tinyarray) or numbers (interpreted as 1 by 1 matrices).

Parameters **symmetry** : Symmetry or *None*

The symmetry of the system.

Notes

Values can be also functions that receive the site or the hopping (passed to the function as two sites) and possibly additional arguments and are expected to return a valid value. This allows to define systems quickly, to modify them without reconstructing, and to save memory for many-orbital models.

In addition to simple keys (single sites and hoppings) more powerful keys are possible as well that allow to manipulate multiple sites/hoppings in a single operation. Such keys are internally expanded into a sequence of simple keys by using the method Builder.expand. For example, `sys[general_key] = value` is equivalent to

```
for simple_key in sys.expand(general_key):
    sys[simple_key] = value
```

Builder instances automatically ensure that every hopping is Hermitian, so that if `builder[a, b]` has been set, there is no need to set `builder[b, a]`.

Builder instances can be made to automatically respect a `Symmetry` that is passed to them during creation. The behavior of builders with a symmetry is slightly more sophisticated. First of all, it is implicitly assumed throughout Kwant that **every** function assigned as a value to a builder with a symmetry possesses the same symmetry. Secondly, all keys are mapped to the fundamental domain of the symmetry before storing them. This may produce confusing results when neighbors of a site are queried.

The method `attach_lead` *works* only if the sites affected by them have tags which are sequences of integers. It *makes sense* only when these sites live on a regular lattice, like the ones provided by `kwant.lattice`.

Attaching a lead manually (without the use of `attach_lead`) amounts to creating a `Lead` object and appending it to this list.

`builder0 += builder1` adds all the sites, hoppings, and leads of `builder1` to `builder0`. Sites and hoppings present in both systems are overwritten by those in `builder1`. The leads of `builder1` are appended to the leads of the system being extended.

Warning: If functions are used to set values in a builder with a symmetry, then they must satisfy the same symmetry. There is (currently) no check and wrong results will be the consequence of a misbehaving function.

Examples

Define a site.

```
>>> builder[site] = value
```

Print the value of a site.

```
>>> print builder[site]
```

Define a hopping.

```
>>> builder[site1, site2] = value
```

Delete a site.

```
>>> del builder[site3]
```

Detach the last lead. (This does not remove the sites that were added to the scattering region by `attach_lead`.)

```
>>> del builder.leads[-1]
```

Attributes

leads	(list of `Lead` instances) The leads that are attached to the system.

Methods

`attach_lead`(*lead_builder*, *origin=None*, *add_cells=0*)

Attach a lead to the builder, possibly adding missing sites.

Returns the lead number (integer) of the attached lead.

Parameters **lead_builder** : `Builder` with 1D translational symmetry

Builder of the lead which has to be attached.

origin : `Site`

The site which should belong to a domain where the lead should begin. It is used to attach a lead inside the system, e.g. to an inner radius of a ring.

add_cells : int

Number of complete unit cells of the lead to be added to the system *after* the missing sites have been added.

Returns **added_sites** : list of `Site` objects that were added to the system.

Raises **ValueError**

If *lead_builder* does not have proper symmetry, has hoppings with range of more than one lead unit cell, or if it is not completely interrupted by the system.

Notes

This method is not fool-proof, i.e. if it returns an error, there is no guarantee that the system stayed unaltered.

The lead numbering starts from zero and increments from there, i.e. the leads are numbered in the order in which they are attached.

dangling()
Return an iterator over all dangling sites.

degree(*site*)
Return the number of neighbors of a site.

eradicate_dangling()
Keep deleting dangling sites until none are left.

expand(*key*)
Expand a general key into an iterator over simple keys.

Parameters **key** : builder key (see notes)

The key to be expanded

Notes

Keys are (recursively):

- Simple keys: sites or 2-tuples of sites (=hoppings).
- Any (non-tuple) iterable of keys, e.g. a list or a generator expression.
- Any function that returns a key when passed a builder as sole argument, e.g. a `HoppingKind` instance or the function returned by `shape`.

This method is internally used to expand the keys when getting or deleting items of a builder (i.e. `sys[key] = value` or `del sys[key]`).

finalized()
Return a finalized (=usable with solvers) copy of the system.

Returns **finalized_system** : `kwant.system.FiniteSystem`

If there is no symmetry.

finalized_system : `kwant.system.InfiniteSystem`

If a symmetry is present.

Notes

This method does not modify the Builder instance for which it is called.

Attached leads are also finalized and will be present in the finalized system to be returned.

Currently, only Builder instances without or with a 1D translational `Symmetry` can be finalized.

hopping_value_pairs()
Return an iterator over all (hopping, value) pairs.

hoppings()
Return an iterator over all Builder hoppings.

The hoppings that are returned belong to the fundamental domain of the `Builder` symmetry, and are not necessarily the ones that were set initially (but always the equivalent ones).

neighbors(*site*)
Return an iterator over all neighbors of a site.

reversed()
Return a shallow copy of the builder with the symmetry reversed.

This method can be used to attach the same infinite system as lead from two opposite sides. It requires a builder to which an infinite symmetry is associated.

site_value_pairs()
Return an iterator over all (site, value) pairs.

sites()
Return a read-only set over all sites.

The sites that are returned belong to the fundamental domain of the `Builder` symmetry, and are not necessarily the ones that were set initially (but always the equivalent ones).

kwant.builder.Site

class `kwant.builder.`**Site**
Bases: `tuple`

A site, member of a `SiteFamily`.

Sites are the vertices of the graph which describes the tight binding system in a `Builder`.

A site is uniquely identified by its family and its tag.

Parameters **family** : an instance of `SiteFamily`

The 'type' of the site.

tag : a hashable python object

The unique identifier of the site within the site family, typically a vector of integers.

Raises **ValueError**

If `tag` is not a proper tag for `family`.

Notes

For convenience, `family(*tag)` can be used instead of `Site(family, tag)` to create a site.

The parameters of the constructor (see above) are stored as instance variables under the same names. Given a site `site`, common things to query are thus `site.family`, `site.tag`, and `site.pos`.

Methods

count (*value*) → integer – return number of occurrences of value

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Attributes

family
The site family to which the site belongs.

pos
Real space position of the site.

tag
The tag of the site.

kwant.builder.HoppingKind

class kwant.builder.**HoppingKind**(*delta*, *family_a*, *family_b=None*)
Bases: object

A pattern for matching hoppings.

A hopping `(a, b)` matches precisely when the site family of `a` equals `family_a` and that of `b` equals `family_b` and `(a.tag - b.tag)` is equal to `delta`. In other words, the matching hoppings have the form: `(family_a(x + delta), family_b(x))`

Parameters **delta** : Sequence of integers

The sequence is interpreted as a vector with integer elements.

family_a : `SiteFamily`

family_b : `SiteFamily` or `None` (default)

The default value means: use the same family as `family_a`.

Notes

A `HoppingKind` is a callable object: When called with a `Builder` as sole argument, an instance of this class will return an iterator over all possible matching hoppings whose sites are already present in the system. The hoppings do *not* have to be already present in the system. For example:

```
kind = kwant.builder.HoppingKind((1, 0), lat)
sys[kind(sys)] = 1
```

Because a `Builder` can be indexed with functions or iterables of functions, `HoppingKind` instances (or any non-tuple iterables of them, e.g. a list) can be used directly as "wildcards" when setting or deleting hoppings:

```
kinds = [kwant.builder.HoppingKind(v, lat) for v in [(1, 0), (0, 1)]]
sys[kinds] = 1
```

Methods

Attributes

delta

family_a

family_b

kwant.builder.SimpleSiteFamily

class kwant.builder.**SimpleSiteFamily**(*name=None*)
Bases: kwant.builder.SiteFamily

A site family used as an example and for testing.

A family of sites tagged by any python objects where object satisfied condition `object == eval(repr(object))`.

It exists to provide a basic site family that can be used for testing the builder module without other dependencies. It can be also used to tag sites with non-numeric objects like strings should this every be useful.

Due to its low storage efficiency for numbers it is not recommended to use SimpleSiteFamily when kwant.lattice.Monatomic would also work.

Methods

normalize_tag(*tag*)

kwant.builder.BuilderLead

class kwant.builder.**BuilderLead**(*builder*, *interface*)
Bases: kwant.builder.Lead

A lead made from a Builder with a spatial symmetry.

Parameters **builder** : Builder

The tight-binding system of a lead. It has to possess appropriate symmetry, and it may not contain hoppings between further than neighboring images of the fundamental domain.

interface : sequence of Site instances

Sequence of sites in the scattering region to which the lead is attached.

Notes

The hopping from the scattering region to the lead is assumed to be equal to the hopping from a lead unit cell to the next one in the direction of the symmetry vector (i.e. the lead is 'leaving' the system and starts with a hopping).

The given order of interface sites is preserved throughout finalization.

Every system has an attribute *leads*, which stores a list of BuilderLead objects with all the information about the leads that are attached.

Methods

finalized()
Return a kwant.system.InfiniteSystem corresponding to the compressed lead.

The order of interface sites is kept during finalization.

kwant.builder.SelfEnergyLead

class kwant.builder.**SelfEnergyLead**(*selfenergy_func*, *interface*)
Bases: kwant.builder.Lead

A general lead defined by its self energy.

Parameters **selfenergy_func** : function

Function which returns the self energy matrix for the interface sites given the energy and optionally a sequence of extra arguments.

interface : sequence of Site instances

Methods

finalized()
Trivial finalization: the object is returned itself.

selfenergy(*energy*, *args=()*)

kwant.builder.ModesLead

class kwant.builder.**ModesLead**(*modes_func*, *interface*)
Bases: kwant.builder.Lead

A general lead defined by its modes wave functions.

Parameters **modes_func** : function

Function which returns the modes of the lead as a tuple of PropagatingModes and StabilizedModes given the energy and optionally a sequence of extra arguments.

interface :

sequence of Site instances

Methods

finalized()
Trivial finalization: the object is returned itself.

modes(*energy*, *args=()*)

selfenergy(*energy*, *args=()*)

3.2.2 Abstract base classes

SiteFamily(canonical_repr, name)	Abstract base class for site families.
Symmetry	Abstract base class for spatial symmetries.
Lead	Abstract base class for leads that can be attached to a Builder.

kwant.builder.SiteFamily

class kwant.builder.**SiteFamily**(*canonical_repr*, *name*)
Bases: object

Abstract base class for site families.

Site families are the 'type' of `Site` objects. Within a family, individual sites are uniquely identified by tags. Valid tags must be hashable Python objects, further details are up to the family.

Site families must be immutable and fully defined by their initial arguments. They must inherit from this abstract base class and call its __init__ function providing it with two arguments: a canonical representation and a name. The canonical representation will be returned as the objects representation and must uniquely identify the site family instance. The name is a string used to distinguish otherwise identical site families. It may be empty.

All site families must define the method `normalize_tag` which brings a tag to the standard format for this site family.

Site families that are intended for use with plotting should also provide a method *pos(tag)*, which returns a vector with real-space coordinates of the site belonging to this family with a given tag.

Methods

normalize_tag(*tag*)
Return a normalized version of the tag.

Raises TypeError or ValueError if the tag is not acceptable.

kwant.builder.Symmetry

class `kwant.builder.`**Symmetry**
Bases: `object`

Abstract base class for spatial symmetries.

Many physical systems possess a discrete spatial symmetry, which results in special properties of these systems. This class is the standard way to describe discrete spatial symmetries in Kwant. An instance of this class can be passed to a `Builder` instance at its creation. The most important kind of symmetry is translational symmetry, used to define scattering leads.

Each symmetry has a fundamental domain – a set of sites and hoppings, generating all the possible sites and hoppings upon action of symmetry group elements. A class derived from `Symmetry` has to implement mapping of any site or hopping into the fundamental domain, applying a symmetry group element to a site or a hopping, and a method `which` to determine the group element bringing some site from the fundamental domain to the requested one. Additionally, it has to have a property `num_directions` returning the number of independent symmetry group generators (number of elementary periods for translational symmetry).

A `ValueError` must be raised by the symmetry class whenever a symmetry is used together with sites whose site family is not compatible with it. A typical example of this is when the vector defining a translational symmetry is not a lattice vector.

Methods

act(*element*, *a*, *b=None*)
Act with a symmetry group element on a site or hopping.

in_fd(*site*)
Tell whether `site` lies within the fundamental domain.

to_fd(*a*, *b=None*)
Map a site or hopping to the fundamental domain.

If *b* is None, return a site equivalent to *a* within the fundamental domain. Otherwise, return a hopping equivalent to *(a, b)* but where the first element belongs to the fundamental domain.

This default implementation works but may be not efficient.

which(*site*)
Calculate the domain of the site.

Return the group element whose action on a certain site from the fundamental domain will result in the given `site`.

Attributes

num_directions
Number of elementary periods of the symmetry.

kwant.builder.Lead

class `kwant.builder.`**Lead**
Bases: `object`

Abstract base class for leads that can be attached to a `Builder`.

To attach a lead to a builder, append it to the builder's *leads* instance variable. See the documentation of `kwant.builder` for the concrete classes of leads derived from this one.

Attributes

interface	(sequence of sites)

Methods

finalized()
Return a finalized version of the lead.

Returns finalized_lead

Notes

The finalized lead must be an object that can be used as a lead in a `kwant.system.FiniteSystem`. It could be an instance of `kwant.system.InfiniteSystem` for example.

The order of sites for the finalized lead must be the one specified in *interface*.

3.3 `kwant.lattice` – Bravais lattices

3.3.1 General

`TranslationalSymmetry`(*periods)	A translational symmetry defined in real space.
`general`(prim_vecs[, basis, name])	Create a Bravais lattice of any dimensionality, with any number of sites.
`Monatomic`(prim_vecs[, offset, name])	A Bravais lattice with a single site in the basis.
`Polyatomic`(prim_vecs, basis[, name])	A Bravais lattice with an arbitrary number of sites in the basis.

kwant.lattice.TranslationalSymmetry

class `kwant.lattice.`**TranslationalSymmetry**(**periods*)
Bases: `kwant.builder.Symmetry`

A translational symmetry defined in real space.

Group elements of this symmetry are integer tuples of appropriate length.

Parameters **p0, p1, p2, ...** : sequences of real numbers

The symmetry periods in real space.

Notes

This symmetry automatically chooses the fundamental domain for each new *SiteFamily* it encounters. If this site family does not correspond to a Bravais lattice, or if it does not have a commensurate period, an error is produced. A certain flexibility in choice of the fundamental domain can be achieved by calling manually the `add_site_family` method and providing it the *other_vectors* parameter.

The fundamental domain for hoppings are all hoppings `(a, b)` with site *a* in fundamental domain of sites.

Methods

act(*element*, *a*, *b=None*)

add_site_family(*fam*, *other_vectors=None*)
Select a fundamental domain for site family and cache associated data.

Parameters **fam** : *SiteFamily*

the site family which has to be processed. Be sure to delete the previously processed site families from *site_family_data* if you want to modify the cache.

other_vectors : 2d array-like of integers

Bravais lattice vectors used to complement the periods in forming a basis. The fundamental domain consists of all the lattice sites for which the zero coefficients corresponding to the symmetry periods in the basis formed by the symmetry periods and *other_vectors*. If an insufficient number of *other_vectors* is provided to form a basis, the missing ones are selected automatically.

Raises **KeyError**

If *fam* is already stored in *site_family_data*.

ValueError

If lattice *fam* is incompatible with given periods.

in_fd(*site*)
Tell whether `site` lies within the fundamental domain.

reversed()
Return a reversed copy of the symmetry.

The resulting symmetry has all the period vectors opposite to the original and an identical fundamental domain.

to_fd(*a*, *b=None*)
Map a site or hopping to the fundamental domain.

If *b* is None, return a site equivalent to *a* within the fundamental domain. Otherwise, return a hopping equivalent to *(a, b)* but where the first element belongs to the fundamental domain.

This default implementation works but may be not efficient.

which(*site*)

Attributes

num_directions

kwant.lattice.general

kwant.lattice.**general**(*prim_vecs*, *basis=None*, *name=''*)

Create a Bravais lattice of any dimensionality, with any number of sites.

Parameters **prim_vecs** : 2d array-like of floats

The primitive vectors of the Bravais lattice

basis : 2d array-like of floats

The coordinates of the basis sites inside the unit cell

name : string or sequence of strings

Name of the lattice, or sequence of names of all of the sublattices. If the name of the lattice is given, the names of sublattices (if any) are obtained by appending their number to the name of the lattice.

Returns **lattice** : either Monatomic or Polyatomic

Resulting lattice.

Notes

This function is largely an alias to the constructors of corresponding lattices.

kwant.lattice.Monatomic

class kwant.lattice.**Monatomic**(*prim_vecs*, *offset=None*, *name=''*)

Bases: kwant.builder.SiteFamily, kwant.lattice.Polyatomic

A Bravais lattice with a single site in the basis.

Instances of this class provide the SiteFamily interface. Site tags (see SiteFamily) are sequences of integers and describe the lattice coordinates of a site.

Monatomic instances are used as site families on their own or as sublattices of Polyatomic lattices.

Parameters **prim_vecs** : 2d array-like of floats

Primitive vectors of the Bravais lattice.

offset : vector of floats

Displacement of the lattice origin from the real space coordinates origin.

Attributes

offset	(vector) Displacement of the lattice origin from the real space coordinates origin

Methods

closest(*pos*)

Find the lattice coordinates of the site closest to position pos.

n_closest (*pos*, *n=1*)
Find n sites closest to position pos.

Returns **sites** : numpy array

An array with sites coordinates.

neighbors (*n=1*, *eps=1e-08*)
Return n-th nearest neighbor hoppings.

Parameters **n** : integer

Order of the hoppings to return.

eps : float

Tolerance relative to the length of the shortest lattice vector for when to consider lengths to be approximately equal.

Returns **hoppings** : list of kwant.builder.HopplingKind objects

The n-th nearest neighbor hoppings.

Notes

The hoppings are ordered lexicographically according to sublattice from which they originate, sublattice on which they end, and their lattice coordinates. Out of the two equivalent hoppings (a hopping and its reverse) only the lexicographically larger one is returned.

normalize_tag (*tag*)

pos (*tag*)
Return the real-space position of the site with a given tag.

shape (*function*, *start*)
Return a key for all the lattice sites inside a given shape.

The object returned by this method is primarily meant to be used as a key for indexing `kwant.Builder` instances. See example below.

Parameters **function** : callable

A function of real space coordinates that returns a truth value: true for coordinates inside the shape, and false otherwise.

start : 1d array-like

The real-space origin for the flood-fill algorithm.

Returns **shape_sites** : function

Notes

When the function returned by this method is called, a flood-fill algorithm finds and yields all the lattice sites inside the specified shape starting from the specified position.

A `Symmetry` or `Builder` may be passed as sole argument when calling the function returned by this method. This will restrict the flood-fill to the fundamental domain of the symmetry (or the builder's symmetry). Note that unless the shape function has that symmetry itself, the result may be unexpected.

Examples

```
>>> def circle(pos):
...     x, y = pos
...     return x**2 + y**2 < 100
...
>>> lat = kwant.lattice.honeycomb()
>>> sys = kwant.Builder()
>>> sys[lat.shape(circle, (0, 0))] = 0
>>> sys[lat.neighbors()] = 1
```

vec(*int_vec*)
Return the coordinates of a Bravais lattice vector in real space.

Parameters **vec** : integer vector

Returns **output** : real vector

wire(*center*, *radius*)
Return a key for all the lattice sites inside an infinite cylinder.

This method makes it easy to define cylindrical (2d: rectangular) leads that point in any direction. The object returned by this method is primarily meant to be used as a key for indexing `kwant.Builder` instances. See example below.

Parameters **center** : 1d array-like of floats

A point belonging to the axis of the cylinder.

radius : float

The radius of the cylinder.

Notes

The function returned by this method is to be called with a *TranslationalSymmetry* instance (or a `Builder` instance whose symmetry is used then) as sole argument. All the lattice sites (in the fundamental domain of the symmetry) inside the specified infinite cylinder are yielded. The direction of the cylinder is determined by the symmetry.

Examples

```
>>> lat = kwant.lattice.honeycomb()
>>> sym = kwant.TranslationalSymmetry(lat.a.vec((-2, 1)))
>>> lead = kwant.Builder(sym)
>>> lead[lat.wire((0, -5), 5)] = 0
>>> lead[lat.neighbors()] = 1
```

Attributes

prim_vecs
(sequence of vectors) Primitive vectors

prim_vecs[i]‘ is the *i*-th primitive basis vector of the lattice displacement of the lattice origin from the real space coordinates origin.

kwant.lattice.Polyatomic

class `kwant.lattice.`**Polyatomic**(*prim_vecs*, *basis*, *name=’‘*)
Bases: `object`

A Bravais lattice with an arbitrary number of sites in the basis.

Contains `Monatomic` sublattices. Note that an instance of `Polyatomic` is not itself a `SiteFamily`, only its sublattices are.

Parameters **prim_vecs** : 2d array-like of floats

The primitive vectors of the Bravais lattice

basis : 2d array-like of floats

The coordinates of the basis sites inside the unit cell.

name : string or sequence of strings

The name of the lattice, or a sequence of the names of all the sublattices. If the name of the lattice is given, the names of sublattices are obtained by appending their number to the name of the lattice.

Raises **ValueError**

If dimensionalities do not match.

Methods

neighbors(*n=1*, *eps=1e-08*)

Return n-th nearest neighbor hoppings.

Parameters **n** : integer

Order of the hoppings to return.

eps : float

Tolerance relative to the length of the shortest lattice vector for when to consider lengths to be approximately equal.

Returns **hoppings** : list of kwant.builder.HopplingKind objects

The n-th nearest neighbor hoppings.

Notes

The hoppings are ordered lexicographically according to sublattice from which they originate, sublattice on which they end, and their lattice coordinates. Out of the two equivalent hoppings (a hopping and its reverse) only the lexicographically larger one is returned.

shape(*function*, *start*)

Return a key for all the lattice sites inside a given shape.

The object returned by this method is primarily meant to be used as a key for indexing `kwant.Builder` instances. See example below.

Parameters **function** : callable

A function of real space coordinates that returns a truth value: true for coordinates inside the shape, and false otherwise.

start : 1d array-like

The real-space origin for the flood-fill algorithm.

Returns **shape_sites** : function

Notes

When the function returned by this method is called, a flood-fill algorithm finds and yields all the lattice sites inside the specified shape starting from the specified position.

A `Symmetry` or `Builder` may be passed as sole argument when calling the function returned by this method. This will restrict the flood-fill to the fundamental domain of the symmetry (or the builder's symmetry). Note that unless the shape function has that symmetry itself, the result may be unexpected.

Examples

```
>>> def circle(pos):
...     x, y = pos
...     return x**2 + y**2 < 100
...
>>> lat = kwant.lattice.honeycomb()
>>> sys = kwant.Builder()
>>> sys[lat.shape(circle, (0, 0))] = 0
>>> sys[lat.neighbors()] = 1
```

vec(*int_vec*)
Return the coordinates of a Bravais lattice vector in real space.

Parameters **vec** : integer vector

Returns **output** : real vector

wire(*center*, *radius*)
Return a key for all the lattice sites inside an infinite cylinder.

This method makes it easy to define cylindrical (2d: rectangular) leads that point in any direction. The object returned by this method is primarily meant to be used as a key for indexing `kwant.Builder` instances. See example below.

Parameters **center** : 1d array-like of floats

A point belonging to the axis of the cylinder.

radius : float

The radius of the cylinder.

Notes

The function returned by this method is to be called with a *TranslationalSymmetry* instance (or a `Builder` instance whose symmetry is used then) as sole argument. All the lattice sites (in the fundamental domain of the symmetry) inside the specified infinite cylinder are yielded. The direction of the cylinder is determined by the symmetry.

Examples

```
>>> lat = kwant.lattice.honeycomb()
>>> sym = kwant.TranslationalSymmetry(lat.a.vec((-2, 1)))
>>> lead = kwant.Builder(sym)
>>> lead[lat.wire((0, -5), 5)] = 0
>>> lead[lat.neighbors()] = 1
```

Attributes

prim_vecs
(sequence of vectors) Primitive vectors

prim_vecs[i] is the *i*-th primitive basis vector of the lattice displacement of the lattice origin from the real space coordinates origin.

3.3.2 Library of lattices

chain([a, name])	Make a one-dimensional lattice.
square([a, name])	Make a square lattice.
triangular([a, name])	Make a triangular lattice.
honeycomb([a, name])	Make a honeycomb lattice.
kagome([a, name])	Make a kagome lattice.

kwant.lattice.chain

kwant.lattice.**chain**(*a=1*, *name=''*)
Make a one-dimensional lattice.

kwant.lattice.square

kwant.lattice.**square**(*a=1*, *name=''*)
Make a square lattice.

kwant.lattice.triangular

kwant.lattice.**triangular**(*a=1*, *name=''*)
Make a triangular lattice.

kwant.lattice.honeycomb

kwant.lattice.**honeycomb**(*a=1*, *name=''*)
Make a honeycomb lattice.

kwant.lattice.kagome

kwant.lattice.**kagome**(*a=1*, *name=''*)
Make a kagome lattice.

3.4 kwant.plotter – Plotting of systems

3.4.1 Plotting routine

plot(sys[, num_lead_cells, unit, ...])	Plot a system in 2 or 3 dimensions.
map(sys, value[, colorbar, cmap, vmin, ...])	Show interpolated map of a function defined for the sites of a system.
bands(sys[, args, momenta, file, show, dpi, ...])	Plot band structure of a translationally invariant 1D system.

kwant.plotter.plot

kwant.plotter.**plot**(*sys*, *num_lead_cells=2*, *unit='nn'*, *site_symbol=None*, *site_size=None*, *site_color=None*, *site_edgecolor=None*, *site_lw=None*, *hop_color=None*, *hop_lw=None*, *lead_site_symbol=None*, *lead_site_size=None*, *lead_color=None*, *lead_site_edgecolor=None*, *lead_site_lw=None*, *lead_hop_lw=None*, *pos_transform=None*, *cmap='gray'*, *colorbar=True*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*)

Plot a system in 2 or 3 dimensions.

Parameters **sys** : kwant.builder.Builder or kwant.system.FiniteSystem

A system to be plotted.

num_lead_cells : int

Number of lead copies to be shown with the system.

unit : 'nn', 'pt', or float

The unit used to specify symbol sizes and linewidths. Possible choices are:

- 'nn': unit is the shortest hopping or a typical nearst neighbor distance in the system if there are no hoppings. This means that symbol sizes/linewidths will scale as the zoom level of the figure is changed. Very short distances are discarded before searching for the shortest. This choice means that the symbols will scale if the figure is zoomed.
- 'pt': unit is points (point = 1/72 inch) in figure space. This means that symbols and linewidths will always be drawn with the same size independent of zoom level of the plot.
- float: sizes are given in units of this value in real (system) space, and will accordingly scale as the plot is zoomed.

The default value is 'nn', which allows to ensure that the images neighboring sites do not overlap.

site_symbol : symbol specification, function, array, or *None*

Symbol used for representing a site in the plot. Can be specified as

- 'o': circle with radius of 1 unit.
- 's': square with inner circle radius of 1 unit.
- *('p', nvert, angle)*: regular polygon with *nvert* vertices, rotated by *angle*. *angle* is given in degrees, and `angle=0` corresponds to one edge of the polygon pointing upward. The radius of the inner circle is 1 unit.
- 'no symbol': no symbol is plotted.
- 'S', *('P', nvert, angle)*: as the lower-case variants described above, but with an area equal to a circle of radius 1. (Makes the visual size of the symbol equal to the size of a circle with radius 1).
- matplotlib.path.Path instance.

Instead of a single symbol, different symbols can be specified for different sites by passing a function that returns a valid symbol specification for each site, or by passing an array of symbols specifications (only for kwant.system.FiniteSystem).

site_size : number, function, array, or *None*

Relative (linear) size of the site symbol.

site_color : `matplotlib` color description, function, array, or *None*

A color used for plotting a site in the system. If a colormap is used, it should be a function returning single floats or a one-dimensional array of floats.

site_edgecolor : `matplotlib` color description, function, array, or *None*

Color used for plotting the edges of the site symbols. Only valid matplotlib color descriptions are allowed (and no combination of floats and colormap as for site_color).

site_lw : number, function, array, or *None*

Linewidth of the site symbol edges.

hop_color : `matplotlib` color description or a function

Same as *site_color*, but for hoppings. A function is passed two sites in this case. (arrays are not allowed in this case).

hop_lw : number, function, or *None*

Linewidth of the hoppings.

lead_site_symbol : symbol specification or *None*

Symbol to be used for the leads. See *site_symbol* for allowed specifications. Note that for leads, only constants (i.e. no functions or arrays) are allowed. If None, then *site_symbol* is used if it is constant (i.e. no function or array), the default otherwise. The same holds for the other lead properties below.

lead_site_size : number or *None*

Relative (linear) size of the lead symbol

lead_color : `matplotlib` color description or *None*

For the leads, *num_lead_cells* copies of the lead unit cell are plotted. They are plotted in color fading from *lead_color* to white (alpha values in *lead_color* are supported) when moving from the system into the lead. Is also applied to the hoppings.

lead_site_edgecolor : `matplotlib` color description or *None*

Color of the symbol edges (no fading done).

lead_site_lw : number or *None*

Linewidth of the lead symbols.

lead_hop_lw : number or *None*

Linewidth of the lead hoppings.

cmap : `matplotlib` color map or a sequence of two color maps or *None*

The color map used for sites and optionally hoppings.

pos_transform : function or *None*

Transformation to be applied to the site position.

colorbar : bool

Whether to show a colorbar if colormap is used. Ignored if *ax* is provided.

file : string or file object or *None*

The output file. If *None*, output will be shown instead.

show : bool

Whether *matplotlib.pyplot.show()* is to be called, and the output is to be shown immediately. Defaults to *True*.

dpi : float or *None*

Number of pixels per inch. If not set the `matplotlib` default is used.

fig_size : tuple or *None*

Figure size *(width, height)* in inches. If not set, the default `matplotlib` value is used.

ax : `matplotlib.axes.Axes` instance or *None*

If *ax* is not *None*, no new figure is created, but the plot is done within the existing Axes *ax*. in this case, *file*, *show*, *dpi* and *fig_size* are ignored.

Returns **fig** : matplotlib figure

A figure with the output if *ax* is not set, else None.

Notes

- If *None* is passed for a plot property, a default value depending on the dimension is chosen. Typically, the default values result in acceptable plots.
- The meaning of "site" depends on whether the system to be plotted is a builder or a low level system. For builders, a site is a kwant.builder.Site object. For low level systems, a site is an integer – the site number.
- color and symbol definitions may be tuples, but not lists or arrays. Arrays of values (linewidths, colors, sizes) may not be tuples.
- The dimensionality of the plot (2D vs 3D) is inferred from the coordinate array. If there are more than three coordinates, only the first three are used. If there is just one coordinate, the second one is padded with zeros.
- The system is scaled to fit the smaller dimension of the figure, given its aspect ratio.

kwant.plotter.map

`kwant.plotter.`**`map`**(*sys*, *value*, *colorbar=True*, *cmap=None*, *vmin=None*, *vmax=None*, *a=None*, *method='nearest'*, *oversampling=3*, *num_lead_cells=0*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*, *pos_transform=None*)

Show interpolated map of a function defined for the sites of a system.

Create a pixmap representation of a function of the sites of a system by calling `mask_interpolate` and show this pixmap using matplotlib.

Parameters **sys** : kwant.system.FiniteSystem or kwant.builder.Builder

The system for whose sites *value* is to be plotted.

value : function or list

Function which takes a site and returns a value if the system is a builder, or a list of function values for each system site of the finalized system.

colorbar : bool, optional

Whether to show a color bar if numerical data has to be plotted. Defaults to *True*. If *ax* is provided, the colorbar is never plotted.

cmap : `matplotlib` color map or *None*

The color map used for sites and optionally hoppings, if *None*, `matplotlib` default is used.

vmin : float, optional

The lower saturation limit for the colormap; values returned by *value* which are smaller than this will saturate

vmax : float, optional

The upper saturation limit for the colormap; valued returned by *value* which are larger than this will saturate

a : float, optional

Reference length. If not given, it is determined as a typical nearest neighbor distance.

method : string, optional

Passed to `scipy.interpolate.griddata`: "nearest" (default), "linear", or "cubic"

oversampling : integer, optional

Number of pixels per reference length. Defaults to 3.

num_lead_cells : integer, optional

number of lead unit cells that should be plotted to indicate the position of leads. Defaults to 0.

file : string or file object or *None*

The output file. If *None*, output will be shown instead.

show : bool

Whether *matplotlib.pyplot.show()* is to be called, and the output is to be shown immediately. Defaults to *True*.

ax : `matplotlib.axes.Axes` instance or *None*

If *ax* is not *None*, no new figure is created, but the plot is done within the existing Axes *ax*. in this case, *file*, *show*, *dpi* and *fig_size* are ignored.

pos_transform : function or *None*

Transformation to be applied to the site position.

Returns **fig** : matplotlib figure

A figure with the output if *ax* is not set, else None.

Notes

- When plotting a system on a square lattice and *method* is "nearest", it makes sense to set *oversampling* to `1`. Then, each site will correspond to exactly one pixel.

kwant.plotter.bands

`kwant.plotter.`**bands**(*sys*, *args=()*, *momenta=65*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*)

Plot band structure of a translationally invariant 1D system.

Parameters **sys** : kwant.system.InfiniteSystem

A system bands of which are to be plotted.

args : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

momenta : int or 1D array-like

Either a number of sampling points on the interval [-pi, pi], or an array of points at which the band structure has to be evaluated.

file : string or file object or *None*

The output file. If *None*, output will be shown instead.

show : bool

Whether *matplotlib.pyplot.show()* is to be called, and the output is to be shown immediately. Defaults to *True*.

dpi : float

Number of pixels per inch. If not set the `matplotlib` default is used.

fig_size : tuple

Figure size *(width, height)* in inches. If not set, the default `matplotlib` value is used.

ax : `matplotlib.axes.Axes` instance or *None*

If *ax* is not *None*, no new figure is created, but the plot is done within the existing Axes *ax*. in this case, *file*, *show*, *dpi* and *fig_size* are ignored.

Returns **fig** : matplotlib figure

A figure with the output if *ax* is not set, else None.

Notes

See `physics.Bands` for the calculation of dispersion without plotting.

3.4.2 Data-generating functions

`sys_leads_sites`(sys[, num_lead_cells])	Return all the sites of the system and of the leads as a list.
`sys_leads_hoppings`(sys[, num_lead_cells])	Return all the hoppings of the system and of the leads as an iterator.
`sys_leads_pos`(sys, site_lead_nr)	Return an array of positions of sites in a system.
`sys_leads_hopping_pos`(sys, hop_lead_nr)	Return arrays of coordinates of all hoppings in a system.
`mask_interpolate`(coords, values[, a, ...])	Interpolate a scalar function in vicinity of given points.

kwant.plotter.sys_leads_sites

`kwant.plotter.`**`sys_leads_sites`**(*sys*, *num_lead_cells=2*)

Return all the sites of the system and of the leads as a list.

Parameters **sys** : kwant.builder.Builder or kwant.system.System instance

The system, sites of which should be returned.

num_lead_cells : integer

The number of times lead sites from each lead should be returned. This is useful for showing several unit cells of the lead next to the system.

Returns **sites** : list of (site, lead_number, copy_number) tuples

A site is a `builder.Site` instance if the system is not finalized, and an integer otherwise. For system sites *lead_number* is *None* and *copy_number* is *0*, for leads both are integers.

lead_cells : list of slices

lead_cells[i] gives the position of all the coordinates of lead *i* within *sites*.

Notes

Leads are only supported if they are of the same type as the original system, i.e. sites of `builder.BuilderLead` leads are returned with an unfinalized system, and sites of `system.InfiniteSystem` leads are returned with a finalized system.

kwant.plotter.sys_leads_hoppings

`kwant.plotter.`**`sys_leads_hoppings`**(*sys*, *num_lead_cells=2*)

Return all the hoppings of the system and of the leads as an iterator.

Parameters **sys** : kwant.builder.Builder or kwant.system.System instance

The system, sites of which should be returned.

num_lead_cells : integer

The number of times lead sites from each lead should be returned. This is useful for showing several unit cells of the lead next to the system.

Returns **hoppings** : list of (hopping, lead_number, copy_number) tuples

A site is a `builder.Site` instance if the system is not finalized, and an integer otherwise. For system sites *lead_number* is *None* and *copy_number* is *0*, for leads both are integers.

lead_cells : list of slices

lead_cells[i] gives the position of all the coordinates of lead *i* within *hoppings*.

Notes

Leads are only supported if they are of the same type as the original system, i.e. hoppings of `builder.BuilderLead` leads are returned with an unfinalized system, and hoppings of `system.InfiniteSystem` leads are returned with a finalized system.

kwant.plotter.sys_leads_pos

`kwant.plotter.`**`sys_leads_pos`**(*sys*, *site_lead_nr*)

Return an array of positions of sites in a system.

Parameters **sys** : `kwant.builder.Builder` or `kwant.system.System` instance

The system, coordinates of sites of which should be returned.

site_lead_nr : list of *(site, leadnr, copynr)* tuples

Output of `sys_leads_sites` applied to the system.

Returns **coords** : numpy.ndarray of floats

Array of coordinates of the sites.

Notes

This function uses *site.pos* property to get the position of a builder site and *sys.pos(sitenr)* for finalized systems. This function requires that all the positions of all the sites have the same dimensionality.

kwant.plotter.sys_leads_hopping_pos

kwant.plotter.**sys_leads_hopping_pos**(*sys*, *hop_lead_nr*)

Return arrays of coordinates of all hoppings in a system.

Parameters **sys** : kwant.builder.Builder or kwant.system.System instance

The system, coordinates of sites of which should be returned.

hoppings : list of *(hopping, leadnr, copynr)* tuples

Output of sys_leads_hoppings applied to the system.

Returns **coords** : (end_site, start_site): tuple of NumPy arrays of floats

Array of coordinates of the hoppings. The first half of coordinates in each array entry are those of the first site in the hopping, the last half are those of the second site.

Notes

This function uses *site.pos* property to get the position of a builder site and *sys.pos(sitenr)* for finalized systems. This function requires that all the positions of all the sites have the same dimensionality.

kwant.plotter.mask_interpolate

kwant.plotter.**mask_interpolate**(*coords*, *values*, *a=None*, *method='nearest'*, *oversampling=3*)

Interpolate a scalar function in vicinity of given points.

Create a masked array corresponding to interpolated values of the function at points lying not further than a certain distance from the original data points provided.

Parameters **coords** : np.ndarray

An array with site coordinates.

values : np.ndarray

An array with the values from which the interpolation should be built.

a : float, optional

Reference length. If not given, it is determined as a typical nearest neighbor distance.

method : string, optional

Passed to scipy.interpolate.griddata: "nearest" (default), "linear", or "cubic"

oversampling : integer, optional

Number of pixels per reference length. Defaults to 3.

Returns **array** : 2d NumPy array

The interpolated values.

min, max : vectors

The real-space coordinates of the two extreme ([0, 0] and [-1, -1]) points of array.

Notes

- *min* and *max* are chosen such that when plotting a system on a square lattice and *oversampling* is set to an odd integer, each site will lie exactly at the center of a pixel of the output array.
- When plotting a system on a square lattice and *method* is "nearest", it makes sense to set *oversampling* to `1`. Then, each site will correspond to exactly one pixel in the resulting array.

3.5 `kwant.solvers` – Library of solvers

3.5.1 Overview

Kwant offers several modules for computing the solutions to quantum transport problems, the so-called solvers. Each of these solvers may use different internal algorithms and/or depend on different external libraries. If the libraries needed by one solver are not installed, trying to import it will raise an `ImportError` exception. The *Installation instructions* list all the libraries that are required or can be used by Kwant and its solvers.

3.5.2 `kwant.solvers.default` – The default solver

There is one solver, `kwant.solvers.default` that is always available. For each Kwant installation it combines the best functionality of the *available* solvers into a single module. We recommend to use it unless there are specific reasons to use another. The following functions are provided.

`smatrix`(sys[, energy, args, out_leads, ...])	Compute the scattering matrix of a system.
`greens_function`(sys[, energy, args, ...])	Compute the retarded Green's function of the system between its leads.
`wave_function`(sys[, energy, args, ...])	Return a callable object for the computation of the wave function inside the scatte
`ldos`(sys[, energy, args, check_hermiticity])	Calculate the local density of states of a system at a given energy.

kwant.solvers.default.smatrix

`kwant.solvers.default.`**`smatrix`**(*sys*, *energy=0*, *args=()*, *out_leads=None*, *in_leads=None*, *check_hermiticity=True*)

Compute the scattering matrix of a system.

Parameters **sys** : `kwant.system.FiniteSystem`

Low level system, containing the leads and the Hamiltonian of a scattering region.

energy : number

Excitation energy at which to solve the scattering problem.

args : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

out_leads : sequence of integers or `None`

Numbers of leads where current or wave function is extracted. None is interpreted as all leads. Default is `None` and means "all leads".

in_leads : sequence of integers or `None`

Numbers of leads in which current or wave function is injected. None is interpreted as all leads. Default is `None` and means "all leads".

check_hermiticity : `bool`

Check if the Hamiltonian matrices are Hermitian. Enables deduction of missing transmission coefficients.

Returns **output** : `SMatrix`

See the notes below and `SMatrix` documentation.

Notes

This function can be used to calculate the conductance and other transport properties of a system. See the documentation for its output type, `SMatrix`.

The returned object contains the scattering matrix elements from the *in_leads* to the *out_leads* as well as information about the lead modes.

Both *in_leads* and *out_leads* must be sorted and may only contain unique entries.

kwant.solvers.default.greens_function

`kwant.solvers.default.`**`greens_function`**(*sys*, *energy=0*, *args=()*, *out_leads=None*, *in_leads=None*, *check_hermiticity=True*)

Compute the retarded Green's function of the system between its leads.

Parameters **sys** : `kwant.system.FiniteSystem`

Low level system, containing the leads and the Hamiltonian of a scattering region.

energy : number

Excitation energy at which to solve the scattering problem.

args : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

out_leads : sequence of integers or `None`

Numbers of leads where current or wave function is extracted. None is interpreted as all leads. Default is `None` and means "all leads".

in_leads : sequence of integers or `None`

Numbers of leads in which current or wave function is injected. None is interpreted as all leads. Default is `None` and means "all leads".

check_hermiticity : `bool`

Check if the Hamiltonian matrices are Hermitian. Enables deduction of missing transmission coefficients.

Returns **output** : `GreensFunction`

See the notes below and `GreensFunction` documentation.

Notes

This function can be used to calculate the conductance and other transport properties of a system. It is often slower and less stable than the scattering matrix-based calculation executed by `smatrix`, and is currently provided mostly for testing purposes and compatibility with RGF code.

It returns an object encapsulating the Green's function elements between the system sites interfacing the leads in *in_leads* and those interfacing the leads in *out_leads*. The returned object also contains a list with self-energies of the leads.

Both *in_leads* and *out_leads* must be sorted and may only contain unique entries.

kwant.solvers.default.wave_function

kwant.solvers.default.**wave_function**(*sys*, *energy=0*, *args=()*, *check_hermiticity=True*)
Return a callable object for the computation of the wave function inside the scattering region.

Parameters **sys** : kwant.system.FiniteSystem

The low level system for which the wave functions are to be calculated.

args : tuple of arguments, or empty tuple

Positional arguments to pass to the function(s) which evaluate the hamiltonian matrix elements

check_hermiticity : bool

Check if the Hamiltonian matrices are Hermitian.

Notes

The returned object can be itself called like a function. Given a lead number, it returns a 2d NumPy array that contains the wave function within the scattering region due to each incoming mode of the given lead. Index 0 is the mode number, index 1 is the orbital number. The modes appear in the same order as incoming modes in kwant.physics.modes.

Examples

```
>>> wf = kwant.solvers.default.wave_function(some_sys, some_energy)
>>> wfs_of_lead_2 = wf(2)
```

kwant.solvers.default.ldos

kwant.solvers.default.**ldos**(*sys*, *energy=0*, *args=()*, *check_hermiticity=True*)
Calculate the local density of states of a system at a given energy.

Parameters **sys** : kwant.system.FiniteSystem

Low level system, containing the leads and the Hamiltonian of the scattering region.

energy : number

Excitation energy at which to solve the scattering problem.

args : tuple of arguments, or empty tuple

Positional arguments to pass to the function(s) which evaluate the hamiltonian matrix elements

check_hermiticity : bool

Check if the Hamiltonian matrices are Hermitian.

Returns **ldos** : a NumPy array

Local density of states at each orbital of the system.

smatrix returns an object of the following type:

kwant.solvers.common.SMatrix(data, ...[, ...])	A scattering matrix.

kwant.solvers.common.SMatrix

class kwant.solvers.common.**SMatrix**(*data*, *lead_info*, *out_leads*, *in_leads*, *current_conserving=False*)

Bases: kwant.solvers.common.BlockResult

A scattering matrix.

Transport properties can be easily accessed using the transmission method (don't be fooled by the name, it can also compute reflection, which is just transmission from one lead back into the same lead.)

SMatrix however also allows for a more direct access to the result: The data stored in SMatrix is a scattering matrix with respect to lead modes and these modes themselves. The details of this data can be directly accessed through the instance variables *data* and *lead_info*. Subblocks of data corresponding to particular leads are conveniently obtained by submatrix.

Attributes

data	(NumPy array) a matrix containing all the requested matrix elements of the scattering matrix.
lead_info	(list of data) a list containing kwant.physics.PropagatingModes for each lead.
out_leads, in_leads	(sequence of integers) indices of the leads where current is extracted (out) or injected (in). Only those are listed for which SMatrix contains the calculated result.

Methods

block_coords(*lead_out*, *lead_in*)
Return slices corresponding to the block from lead_in to lead_out.

conductance_matrix()
Return the conductance matrix.

This is the matrix C such that $I = CV$ where I and V are, respectively, the vectors of currents and voltages for each lead.

This matrix is useful for calculating non-local resistances. See Section 2.4 of the book by S. Datta.

in_block_coords(*lead_in*)
Return a slice with the columns in the block corresponding to lead_in.

num_propagating(*lead*)
Return the number of propagating modes in the lead.

out_block_coords(*lead_out*)
Return a slice with the rows in the block corresponding to lead_out.

submatrix(*lead_out*, *lead_in*)
Return the matrix elements from lead_in to lead_out.

transmission(*lead_out*, *lead_in*)
Return transmission from lead_in to lead_out.

If the option current_conserving has been enabled for this object, this method will deduce missing transmission values whenever possible.

Current conservation is enabled by default for objects returned by smatrix and greens_function whenever the Hamiltonian has been verified to be Hermitian (option check_hermiticity, enabled by default).

The analog of smatrix, greens_function accordingly returns:

kwant.solvers.common.GreensFunction(data, ...)	Retarded Green's function.

kwant.solvers.common.GreensFunction

class kwant.solvers.common.**GreensFunction**(*data*, *lead_info*, *out_leads*, *in_leads*, *current_conserving=False*)

Bases: kwant.solvers.common.BlockResult

Retarded Green's function.

Transport properties can be easily accessed using the transmission method (don't be fooled by the name, it can also compute reflection, which is just transmission from one lead back into the same lead).

GreensFunction however also allows for a more direct access to the result: The data stored in GreensFunction is the real-space Green's function. The details of this data can be directly accessed through the instance variables *data* and *lead_info*. Subblocks of data corresponding to particular leads are conveniently obtained by submatrix.

Attributes

data	(NumPy array) a matrix containing all the requested matrix elements of Green's function.
lead_info	(list of matrices) a list with self-energies of each lead.
out_leads, in_leads	(sequence of integers) indices of the leads where current is extracted (out) or injected (in). Only those are listed for which SMatrix contains the calculated result.

Methods

block_coords(*lead_out*, *lead_in*)
Return slices corresponding to the block from lead_in to lead_out.

conductance_matrix()
Return the conductance matrix.

This is the matrix C such that $I = CV$ where I and V are, respectively, the vectors of currents and voltages for each lead.

This matrix is useful for calculating non-local resistances. See Section 2.4 of the book by S. Datta.

in_block_coords(*lead_in*)
Return a slice with the columns in the block corresponding to lead_in.

num_propagating(*lead*)
Return the number of propagating modes in the lead.

out_block_coords(*lead_out*)
Return a slice with the rows in the block corresponding to lead_out.

submatrix(*lead_out*, *lead_in*)
Return the matrix elements from lead_in to lead_out.

transmission(*lead_out*, *lead_in*)
Return transmission from lead_in to lead_out.

If the option current_conserving has been enabled for this object, this method will deduce missing transmission values whenever possible.

Current conservation is enabled by default for objects returned by smatrix and greens_function whenever the Hamiltonian has been verified to be Hermitian (option check_hermiticity, enabled by default).

Being just a thin wrapper around other solvers, the default solver selectively imports their functionality. To find out the origin of any function in this module, use Python's help. For example

```
>>> help(kwant.solvers.default.ldos)
```

3.5.3 Other solver modules

Unlike the default one, other solvers have to be imported manually. They provide, whenever possible, exactly the same interface as the default. Some allow for specific tuning that can improve performance. The differences to the default solver are listed in the documentation of each module.

`kwant.solvers.sparse` – Basic sparse matrix solver

This solver uses SciPy's scipy.sparse.linalg. The interface is identical to that of the `default solver`.

`scipy.sparse.linalg` currently uses internally either the direct sparse solver UMFPACK or if that is not installed, SuperLU. Often, SciPy's SuperLU will give quite poor performance and you will be warned if only SuperLU is found. The module variable `uses_umfpack` can be checked to determine if UMFPACK is being used.

`kwant.solvers.mumps` – High performance sparse solver based on MUMPS

This solver uses MUMPS. (Only the sequential, single core version of MUMPS is used.) MUMPS is a very efficient direct sparse solver that can take advantage of memory beyond 3GiB for the solution of large problems. Furthermore, it offers a choice of several orderings of the input matrix some of which can speed up a calculation significantly.

Compared with the `default solver`, this module adds several options that may be used to fine-tune performance. Otherwise the interface is identical. These options can be set and queried with the following functions.

`kwant.solvers.mumps.`**`options`**(*self*, *nrhs=None*, *ordering=None*, *sparse_rhs=None*)

Modify some options. Return the old options.

Parameters **nrhs** : number

number of right hand sides that should be solved simultaneously. A value around 5-10 gives optimal performance on many machines. If memory is an issue, it can be set to 1, to minimize memory usage (at the cost of slower performance). Default value is 6.

ordering : string

one of the ordering methods supported by the MUMPS solver (see `mumps`. The availability of certain orderings depends on the MUMPS installation.), or 'kwant_decides'. If `ordering=='kwant_decides'`, the ordering that typically gives the best performance is chosen from the available ones. One can also defer the choice of ordering to MUMPS by specifying 'auto', in some cases MUMPS however chooses poorly.

The choice of ordering can significantly influence the performance and memory impact of the solve phase. Typically the nested dissection orderings 'metis' and 'scotch' are most suited for physical systems. Default is 'kwant_decides'

sparse_rhs : True or False

whether to use a sparse right hand side in the solve phase of MUMPS. Preliminary tests have not shown a significant performance increase when this feature is used, but this needs more looking into. Default value is False.

Returns old_options: dict

dictionary containing the previous options.

Notes

Thanks to this method returning the old options as a dictionary it is easy to change some options temporarily:

```
>>> saved_options = kwant.solvers.mumps.options(nrhs=12)
>>> some_code()
>>> kwant.solvers.mumps.options(**saved_options)
```

kwant.solvers.mumps.**reset_options**(*self*)
Set the options to default values. Return the old options.

3.5.4 For Kwant experts: detail of the internal structure of a solver

Each solver module (except the default one) contains a class `Solver` (e.g. `kwant.solvers.sparse.Solver`), that actually implements that solver's functionality. For each module-level function provided by the solver, there is a correspondent method in the `Solver` class. The module-level functions are simply the methods of a hidden `Solver` instance that is present in each solver module.

The encapsulation in a class allows different solvers to easily share common code. It also makes it possible to use solvers with different options concurrently. Typically, one does not need this flexibility, and will not want to bother with the `Solver` class itself. Instead, one will use the module-level functions as explained in the previous sections.

3.6 `kwant.physics` – Physics-related algorithms

3.6.1 Leads

Bands(sys[, args])	Class of callable objects for the computation of energy bands.
modes(h_cell, h_hop[, tol, stabilization])	Compute the eigendecomposition of a translation operator of a lead.
selfenergy(h_cell, h_hop[, tol])	Compute the self-energy generated by the lead.
two_terminal_shotnoise(smatrix)	Compute the shot-noise in a two-terminal setup.
PropagatingModes(wave_functions, velocities, ...)	The calculated propagating modes of a lead.
StabilizedModes(vecs, vecslmbdainv, nmodes)	Stabilized eigendecomposition of the translation operator.

kwant.physics.Bands

class kwant.physics.**Bands**(*sys*, *args=()*)
Bases: object

Class of callable objects for the computation of energy bands.

Parameters **sys** : kwant.system.InfiniteSystem

The low level infinite system for which the energies are to be calculated.

args : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

Notes

An instance of this class can be called like a function. Given a momentum (currently this must be a scalar as all infinite systems are quasi-1-d), it returns a NumPy array containing the eigenenergies of all modes at this momentum

Examples

```
>>> bands = kwant.physics.Bands(some_sys)
>>> momenta = numpy.linspace(-numpy.pi, numpy.pi, 101)
>>> energies = [bands(k) for k in momenta]
>>> pyplot.plot(momenta, energies)
>>> pyplot.show()
```

Methods

kwant.physics.modes

kwant.physics.**modes**(*h_cell*, *h_hop*, *tol=1000000.0*, *stabilization=None*)

Compute the eigendecomposition of a translation operator of a lead.

Parameters **h_cell** : numpy array, real or complex, shape (N,N) The unit cell

Hamiltonian of the lead unit cell.

h_hop : numpy array, real or complex, shape (N,M)

The hopping matrix from a lead cell to the one on which self-energy has to be calculated (and any other hopping in the same direction).

tol : float

Numbers and differences are considered zero when they are smaller than *tol* times the machine precision.

stabilization : sequence of 2 booleans or None

Which steps of the eigenvalue prolem stabilization to perform. If the value is *None*, then Kwant chooses the fastest (and least stable) algorithm that is expected to be sufficient. For any other value, Kwant forms the eigenvalue problem in the basis of the hopping singular values. The first element set to *True* forces Kwant to add an anti-Hermitian term to the cell Hamiltonian before inverting. If it is set to *False*, the extra term will only be added if the cell Hamiltonian isn't invertible. The second element set to *True* forces Kwant to solve a generalized eigenvalue problem, and not to reduce it to the regular one. If it is *False*, reduction to a regular problem is performed if possible. Selecting the stabilization manually is mostly necessary for testing purposes.

Returns **propagating** : PropagatingModes

Contains the array of the wave functions of propagating modes, their momenta, and their velocities. It can be used to identify the gauge in which the scattering problem is solved.

stabilized : StabilizedModes

A basis of propagating and evanescent modes used by the solvers.

Notes

The propagating modes are sorted according to the longitudinal component of their k-vector, with incoming modes having k sorted in descending order, and outgoing modes having k sorted in ascending order. In simple cases where bands do not cross, this ordering corresponds to "lowest modes first". In general, however, it is necessary to examine the band structure – something this function is not doing by design.

Propagating modes with the same momentum are orthogonalized. All the propagating modes are normalized by current.

This function uses the most stable and efficient algorithm for calculating the mode decomposition that the Kwant authors are aware about. Its details are to be published.

kwant.physics.selfenergy

kwant.physics.**selfenergy**(*h_cell*, *h_hop*, *tol=1000000.0*)
Compute the self-energy generated by the lead.

Parameters **h_cell** : numpy array, real or complex, shape (N,N) The unit cell Hamiltonian of the lead unit cell.

h_hop : numpy array, real or complex, shape (N,M)

The hopping matrix from a lead cell to the one on which self-energy has to be calculated (and any other hopping in the same direction).

tol : float

Numbers are considered zero when they are smaller than *tol* times the machine precision.

Returns **Sigma** : numpy array, real or complex, shape (M,M)

The computed self-energy. Note that even if *h_cell* and *h_hop* are both real, *Sigma* will typically be complex. (More precisely, if there is a propagating mode, *Sigma* will definitely be complex.)

Notes

For simplicity this function internally calculates the modes first. This may cause a small slowdown, and can be improved if necessary.

kwant.physics.two_terminal_shotnoise

kwant.physics.**two_terminal_shotnoise**(*smatrix*)
Compute the shot-noise in a two-terminal setup.

In a two terminal system the shot noise is given by *tr((1 - t*t^dagger) * t*t^dagger)*.

Parameters **smatrix** : SMatrix instance

A two terminal scattering matrix.

Returns **noise** : float

Shot noise measured in noise quanta *2 e^3 |V| / pi hbar*.

kwant.physics.PropagatingModes

class kwant.physics.**PropagatingModes**(*wave_functions*, *velocities*, *momenta*)
Bases: object

The calculated propagating modes of a lead.

Notes

The sort order of all the three arrays is identical. The first half of the modes have negative velocity, the second half have positive velocity. The modes with negative velocity are ordered from larger to lower momenta, the modes with positive velocity vice versa.

The first dimension of *wave_functions* corresponds to the orbitals of all the sites in a unit cell, the second one to the number of the mode. Each mode is normalized to carry unit current. If several modes have the same momentum and velocity, an arbitrary orthonormal basis in the subspace of these modes is chosen.

Attributes

wave_functions	(numpy array) The wave functions of the propagating modes.
momenta	(numpy array) Momenta of the modes.
velocities	(numpy array) Velocities of the modes.

Methods

kwant.physics.StabilizedModes

class `kwant.physics.`**`StabilizedModes`**(*vecs*, *vecslmbdainv*, *nmodes*, *sqrt_hop=None*)
Bases: `object`

Stabilized eigendecomposition of the translation operator.

Due to the lack of Hermiticity of the translation operator, its eigendecomposition is frequently poorly conditioned. Solvers in Kwant use this stabilized decomposition of the propagating and evanescent modes in the leads. If the hopping between the unit cells of an infinite system is invertible, the translation eigenproblem is written in the basis *psi_n, h_hop^+ * psi_(n+1)*, with `h_hop` the hopping between unit cells. If *h_hop* is not invertible, and has the singular value decomposition *u s v^+*, then the eigenproblem is written in the basis *sqrt(s) v^+ psi_n, sqrt(s) u^+ psi_(n+1)*. In this basis we calculate the eigenvectors of the propagating modes, and the Schur vectors (an orthogonal basis) in the space of evanescent modes.

vecs and *vecslmbdainv* are the first and the second halves of the wave functions. The first *nmodes* are eigenmodes moving in the negative direction (hence they are incoming into the system in Kwant convention), the second *nmodes* are eigenmodes moving in the positive direction. The remaining modes are the Schur vectors of the modes evanescent in the positive direction. Propagating modes with the same eigenvalue are orthogonalized, and all the propagating modes are normalized to carry unit current. Finally the *sqrt_hop* attribute is *v sqrt(s)*.

Attributes

vecs	(numpy array) Translation eigenvectors.
vecslmbdainv	(numpy array) Translation eigenvectors divided by the corresponding eigenvalues.
nmodes	(int) Number of left-moving (or right-moving) modes.
sqrt_hop	(numpy array or None) Part of the SVD of *h_hop*, or None if the latter is invertible.

Methods

`selfenergy`()
Compute the self-energy generated by lead modes.

Returns **Sigma** : numpy array, real or complex, shape (M,M)

The computed self-energy. Note that even if *h_cell* and *h_hop* are both real, *Sigma* will typically be complex. (More precisely, if there is a propagating mode, *Sigma* will definitely be complex.)

CHAPTER

FOUR

MODULES MAINLY FOR INTERNAL USE

The following modules contain functionality that is most often used only internally by Kwant itself or by advanced users.

4.1 `kwant.system` – Low-level interface of systems

This module is the binding link between constructing tight-binding systems and doing calculations with these systems. It defines the interface which any problem-solving algorithm should be able to access, independently on how the system was constructed. This is achieved by using python abstract base classes (ABC) – classes, which help to ensure that any derived classes implement the necessary interface.

Any system which is provided to a solver should be derived from the appropriate class in this module, and every solver can assume that its input corresponds to the interface defined here.

`System`	Abstract general low-level system.
`InfiniteSystem`	Abstract infinite low-level system.
`FiniteSystem`	Abstract finite low-level system, possibly with leads.
`PrecalculatedLead`([modes, selfenergy])	A general lead defined by its self energy.

4.1.1 kwant.system.System

class `kwant.system.`**`System`**

Bases: `object`

Abstract general low-level system.

Notes

The sites of the system are indexed by integers ranging from 0 to `self.graph.num_nodes - 1`.

Optionally, a class derived from `System` can provide a method *pos* which is assumed to return the real-space position of a site given its index.

Attributes

graph	(kwant.graph.CGraph) The system graph.

Methods

`hamiltonian`(*i*, *j*, **args*)

Return the hamiltonian matrix element for sites *i* and *j*.

If `i == j`, return the on-site Hamiltonian of site *i*.

if `i != j`, return the hopping between site *i* and *j*.

Hamiltonians may depend (optionally) on positional and keyword arguments

hamiltonian_submatrix(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*)

Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

to_sites : sequence of sites or None (default)

from_sites : sequence of sites or None (default)

sparse : bool

Whether to return a sparse or a dense matrix. Defaults to *False*.

return_norb : bool

Whether to return arrays of numbers of orbitals. Defaults to *False*.

Returns **hamiltonian_part** : numpy.ndarray or scipy.sparse.coo_matrix

Submatrix of Hamiltonian of the system.

to_norb : array of integers

Numbers of orbitals on each site in to_sites. Only returned when *return_norb* is true.

from_norb : array of integers

Numbers of orbitals on each site in from_sites. Only returned when *return_norb* is true.

Notes

The returned submatrix contains all the Hamiltonian matrix elements from *from_sites* to *to_sites*. The default for *from_sites* and *to_sites* is *None* which means to use all sites of the system in the order in which they appear.

4.1.2 kwant.system.InfiniteSystem

class `kwant.system.`**InfiniteSystem**

Bases: `kwant.system.System`

Abstract infinite low-level system.

An infinite system consists of an infinite series of identical cells. Adjacent cells are connected by identical inter-cell hoppings.

Notes

The system graph of an infinite systems contains a single cell, as well as the part of the previous cell which is connected to it. The first *cell_size* sites form one complete single cell. The remaining *N* sites of the graph (*N* equals `graph.num_nodes - cell_size`) belong to the previous cell. They are included so that hoppings between cells can be represented. The N sites of the previous cell correspond to the first *N* sites of the fully included cell. When an InfiniteSystem is used as a lead, *N* acts also as the number of interface sites to which it must be connected.

The drawing shows three cells of an infinite system. Each cell consists of three sites. Numbers denote sites which are included into the system graph. Stars denote sites which are not included. Hoppings are included in the graph if and only if they occur between two sites which are part of the graph:

```
    * 2 *
... | | | ...
    * 0 3
    |/|/|
    *-1-4

<-- order of cells
```

The numbering of sites in the drawing is one of the two valid ones for that infinite system. The other scheme has the numbers of site 0 and 1 exchanged, as well as of site 3 and 4.

Attributes

cell_size	(integer) The number of sites in a single cell of the system.

Methods

cell_hamiltonian(*args=()*, *sparse=False*)
Hamiltonian of a single cell of the infinite system.

hamiltonian(*i*, *j*, **args*)
Return the hamiltonian matrix element for sites *i* and *j*.

If `i == j`, return the on-site Hamiltonian of site *i*.

if `i != j`, return the hopping between site *i* and *j*.

Hamiltonians may depend (optionally) on positional and keyword arguments

hamiltonian_submatrix(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*)
Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

to_sites : sequence of sites or None (default)

from_sites : sequence of sites or None (default)

sparse : bool

Whether to return a sparse or a dense matrix. Defaults to *False*.

return_norb : bool

Whether to return arrays of numbers of orbitals. Defaults to *False*.

Returns **hamiltonian_part** : numpy.ndarray or scipy.sparse.coo_matrix

Submatrix of Hamiltonian of the system.

to_norb : array of integers

Numbers of orbitals on each site in to_sites. Only returned when *return_norb* is true.

from_norb : array of integers

Numbers of orbitals on each site in from_sites. Only returned when *return_norb* is true.

Notes

The returned submatrix contains all the Hamiltonian matrix elements from *from_sites* to *to_sites*. The default for *from_sites* and *to_sites* is *None* which means to use all sites of the system in the order in which they appear.

inter_cell_hopping(*args=()*, *sparse=False*)
Hopping Hamiltonian between two cells of the infinite system.

modes(*energy=0*, *args=()*)
Return mode decomposition of the lead

See documentation of `PropagatingModes` and `StabilizedModes` for the return format details.

selfenergy(*energy=0*, *args=()*)
Return self-energy of a lead.

The returned matrix has the shape (s, s), where s is `sum(len(self.hamiltonian(i, i)) for i in range(self.graph.num_nodes - self.cell_size))`.

4.1.3 kwant.system.FiniteSystem

class `kwant.system.`**FiniteSystem**
Bases: `kwant.system.System`

Abstract finite low-level system, possibly with leads.

Notes

The length of *leads* must be equal to the length of *lead_interfaces*.

For lead `n`, the method leads[n].selfenergy must return a square matrix whose size is `sum(len(self.hamiltonian(site, site)) for site in self.lead_interfaces[n])`. The output of `leads[n].modes` has to be a tuple of *StabilizedModes*.

Often, the elements of *leads* will be instances of `InfiniteSystem`. If this is the case for lead `n`, the sites `lead_interfaces[n]` match the first `len(lead_interfaces[n])` sites of the InfiniteSystem.

Attributes

leads	(sequence of leads) Each lead has to provide a method `selfenergy(energy, args)`. It may provide `modes(energy, args)` as well.
lead_interfaces	(sequence of sequences of integers) Each sub-sequence contains the indices of the system sites to which the lead is connected.

Methods

hamiltonian(*i*, *j*, **args*)
Return the hamiltonian matrix element for sites *i* and *j*.

If `i == j`, return the on-site Hamiltonian of site *i*.

if `i != j`, return the hopping between site *i* and *j*.

Hamiltonians may depend (optionally) on positional and keyword arguments

hamiltonian_submatrix(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*)
Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method.

to_sites : sequence of sites or None (default)

from_sites : sequence of sites or None (default)

sparse : bool

Whether to return a sparse or a dense matrix. Defaults to *False*.

return_norb : bool

Whether to return arrays of numbers of orbitals. Defaults to *False*.

Returns **hamiltonian_part** : numpy.ndarray or scipy.sparse.coo_matrix

Submatrix of Hamiltonian of the system.

to_norb : array of integers

Numbers of orbitals on each site in to_sites. Only returned when *return_norb* is true.

from_norb : array of integers

Numbers of orbitals on each site in from_sites. Only returned when *return_norb* is true.

Notes

The returned submatrix contains all the Hamiltonian matrix elements from *from_sites* to *to_sites*. The default for *from_sites* and *to_sites* is *None* which means to use all sites of the system in the order in which they appear.

precalculate(*energy=0*, *args=()*, *leads=None*, *what='modes'*)

Precalculate modes or self-energies in the leads.

Construct a copy of the system, with the lead modes precalculated, which may significantly speed up calculations where only the system is changing.

Parameters **energy** : float

Energy at which the modes or self-energies have to be evaluated.

args : sequence

Additional parameters required for calculating the Hamiltonians

leads : sequence of integers or None

Numbers of the leads to be precalculated. If *None*, all are precalculated.

what : 'modes', 'selfenergy', 'all'

The quantitity to precompute. 'all' will compute both modes and self-energies. Defaults to 'modes'.

Returns **sys** : FiniteSystem

A copy of the original system with some leads precalculated.

Notes

If the leads are precalculated at certain *energy* or *args* values, they might give wrong results if used to solve the system with different parameter values. Use this function with caution.

4.1.4 kwant.system.PrecalculatedLead

class kwant.system.**PrecalculatedLead**(*modes=None*, *selfenergy=None*)
Bases: object

A general lead defined by its self energy.

Parameters **modes** : (kwant.physics.PropagatingModes, kwant.physics.StabilizedModes)

Modes of the lead.

selfenergy : numpy array

Lead self-energy.

Notes

At least one of modes and selfenergy must be provided.

Methods

modes(*energy=0*, *args=()*)

selfenergy(*energy=0*, *args=()*)

4.2 kwant.graph – Low-level, efficient directed graphs

Graphs, as handled by this module, consist of nodes (numbered by integers, usually ≥ 0). Pairs of nodes can be connected by edges (numbered by integers ≥ 0). An edge is described by a pair (tail, head) of node numbers and is always directed.

The basic workflow is to

1. create an object of type Graph,
2. add edges to it using the methods add_edge and add_edges,
3. create a compressed copy of the graph using the method compressed,
4. and use the thus created object for efficient queries.

Example:

```
>>> import kwant
>>> g = kwant.graph.Graph()
>>> g.add_edge(0, 1)
0
>>> g.add_edge(0, 2)
1
>>> g = g.compressed()
>>> list(g.out_neighbors(0))
[1, 2]
```

Node numbers can be assigned freely, but if they are not consecutive integers starting with zero, storage space is wasted in the compressed graph. Negative node numbers are special and can be allowed optionally (see further).

Whenever a method returns multiple edges or nodes (via an iterator), they appear in the order in which the edges associated with them were added to the graph during construction.

Edge IDs are non-negative integers which identify edges unambiguously. They are assigned automatically when the graph is compressed. The edge IDs of edges with the same tail will occupy a dense interval of integers. The IDs of edges sharing the same tail will be assigned from lowest to highest in the order in which these edges had been added.

The method `Graph.compressed` takes a parameter which determines whether the graph will be one-way (the default) or two-way. One-way graphs can be queried for the existence of an edge and provide the nodes to which a node points (=outgoing neighbors). In addition, two-way graphs can be queried for the nodes which point to a node (=incoming neighbors).

Another parameter of `Graph.compressed`, *edge_nr_translation*, determines whether it will be possible to use the method *edge_id* of the compressed graph. This method returns the edge ID of an edge given the edge number that was returned when an edge was added.

Negative node numbers can be allowed for a `Graph` (parameter *allow_negative_nodes* of the constructor). Edges with negative nodes are considered to be dangling: negative nodes can be neighbors of other nodes, but cannot be queried directly for neighbors. Consequently, "doubly-dangling" edges which connect two negative nodes do not make sense and are never allowed. The range of values used for the negative node numbers does not influence the required storage space in any way.

Compressed graphs have the read-only attributes *num_nodes* and *num_edges*.

4.2.1 Graph types

`Graph`	An uncompressed graph.
`CGraph`	A compressed graph which can be efficiently queried for the existence of edges and outgoing neighbors.

kwant.graph.Graph

class `kwant.graph.`**`Graph`**
Bases: `object`

An uncompressed graph. Used to make compressed graphs. (See `CGraph`.)

Methods

`add_edge``()`
Add the directed edge (*tail*, *head*) to the graph.

Parameters **tail** : integer

head : integer

Returns **edge_nr** : integer

The sequential number of the edge. This number can be used to query for the edge ID of an edge in the compressed graph.

Raises **ValueError**

If a negative node is added when this has not been allowed explicitly or if an edge is doubly-dangling.

`add_edges``()`
Add multiple edges in one pass.

Parameters **edges** : iterable of 2-sequences of integers

The parameter *edges* must be an iterable of elements which describe the edges to be added. For each edge-element, edge[0] and edge[1] must give, respectively, the tail and the head. Valid edges are, for example, a list of 2-integer-tuples, or an numpy.ndarray of integers with a shape (n, 2). The latter case is optimized.

Returns **first_edge_nr** : integer

The sequential number of the first of the added edges. The numbers of the other added edges are consecutive integers following the number of the first. Edge numbers can be used to query for the edge ID of an edge in the compressed graph.

compressed()
Build a CGraph from this graph.

Parameters **twoway** : boolean (default: False)

If set, it will be possible to query the compressed graph for incoming neighbors.

edge_nr_translation : boolean (default: False)

If set, it will be possible to call the method *edge_id*.

allow_lost_edges : boolean (default: False)

If set, negative tails are accepted even with one-way compression.

Raises **ValueError**

When negative tails occur while *twoway* and *allow_lost_edges* are both false.

Notes

In a one-way compressed graph, an edge with a negative tail is present only minimally: it is only possible to query the head of such an edge, given the edge ID. This is why one-way compression of a graph with a negative tail leads to a ValueError being raised, unless *allow_lost_edges* is true.

reserve()
Reserve space for edges.

Parameters **capacity** : integer

Number of edges for which to reserve space.

Notes

It is not necessary to call this method, but using it can speed up the creation of graphs.

write_dot()
Write a representation of the graph in dot format to *file*.

That resulting file can be visualized with dot(1) or neato(1) form the graphviz package.

Attributes

num_nodes

kwant.graph.CGraph

class kwant.graph.**CGraph**
Bases: object

A compressed graph which can be efficiently queried for the existence of edges and outgoing neighbors.

Objects of this class do not initialize the members themselves, but expect that they hold usable values. A good way to create them is by compressing a Graph.

Iterating over a graph yields a sequence of (tail, head) pairs of all edges. The number of an edge in this sequence equals its edge ID. The built-in function *enumerate* can thus be used to easily iterate over all edges along with their edge IDs.

Methods

all_edge_ids()
Return an iterator over all edge IDs of edges with a given tail and head.

Parameters **tail** : integer

head : integer

Returns **edge_id** : integer

Raises **NodeDoesNotExist**

EdgeDoesNotExistError

DisabledFeatureError

If `tail` is negative and the graph is not two-way compressed.

edge_id()
Return the edge ID of an edge given its sequential number.

Parameters **edge_nr** : integer

Returns **edge_id** : integer

Raises **DisabledFeatureError**

If `edge_nr_translation` was not enabled during graph compression.

EdgeDoesNotExistError

first_edge_id()
Return the edge ID of the first edge (tail, head).

Parameters **tail** : integer

head : integer

Returns **edge_id** : integer

Raises **NodeDoesNotExist**

EdgeDoesNotExistError

DisabledFeatureError

If `tail` is negative and the graph is not two-way compressed.

Notes

This method is useful for graphs where each edge occurs only once.

has_dangling_edges()

has_edge()
Does the graph contain the edge (tail, head)?

Parameters **tail** : integer

head : integer

Returns **had_edge** : boolean

Raises **NodeDoesNotExistError**

EdgeDoesNotExistError

DisabledFeatureError

If `tail` is negative and the graph is not two-way compressed.

head()
Return the head of an edge, given its edge ID.

Parameters **edge_id** : integer

Raises **EdgeDoesNotExistError**

Notes

This method executes in constant time. It works for all edge IDs, returning both positive and negative heads.

in_edge_ids()
Return the IDs of incoming edges of a node.

Parameters **node** : integer

Returns **edge_ids** : sequence of integers

Raises **NodeDoesNotExistError**

DisabledFeatureError

If the graph is not two-way compressed.

in_neighbors()
Return the nodes which point to a node.

Parameters **node** : integer

Returns **nodes** : sequence of integers

Raises **NodeDoesNotExistError**

DisabledFeatureError

If the graph is not two-way compressed.

out_edge_ids()
Return the IDs of outgoing edges of node.

Parameters **node** : integer

Returns **edge_ids** : sequence of integers

Raises **NodeDoesNotExistError**

out_neighbors()
Return the nodes a node points to.

Parameters **node** : integer

Returns **nodes** : sequence of integers

Raises **NodeDoesNotExistError**

tail()
Return the tail of an edge, given its edge ID.

Parameters **edge_id** : integer

Returns **tail** : integer

If the edge exists and is positive.

None

If the tail is negative.

Raises **EdgeDoesNotExistError**

Notes

The average performance of this method is O(log num_nodes) for non-negative tails and O(1) for negative ones.

write_dot()
Write a representation of the graph in dot format to *file*.

Parameters **file** : file-like object

Notes

That resulting file can be visualized with dot(1) or neato(1) form the graphviz package.

Attributes

edge_nr_translation

num_edges

num_nodes

num_px_edges

num_xp_edges

twoway

4.2.2 Graph algorithms

slice	TODO: write me.
make_undirected	undirected_graph(gr) expects a CGraph gr as input, which is interpreted
remove_duplicates	Remove duplicate edges in the CGraph gr (this applies to the case where there are multiple edges (i,j)
induced_subgraph	Return a subgraph of the CGraph gr by picking all nodes [0:gr.num_nodes] for which select is True.
print_graph	

kwant.graph.slice

kwant.graph.**slice**()
TODO: write me.

kwant.graph.make_undirected

kwant.graph.**make_undirected**()
undirected_graph(gr) expects a CGraph gr as input, which is interpreted as a directed graph, and returns a CGraph that is explicitely undirected, i.e. for every edge (i,j) there is also the edge (j,i). In the process, the function also removes all 'dangling' links, i.e. edges to or from negative node numbers.

If remove_dups == True (default value is True), any duplicates of edges will be removed (this applies to the case where there are multiple edges (i,j), not to having (i,j) and (j,i)).

The effect of the duplicate edges can be retained if calc_weights == True (default value is False), in which case a weight array is returned containing the multiplicity of the edges after the graph has been made undirected.

As a (somewhat drastic but illustrative) example, if make_undirected is applied to a undirected graph, it will return the same graph again (possibly with the order of edges changed) and a weight array with 2 everywhere. (Of course, in this case one does not need to call make_undirected ...)

make_undirected() will always return a one-way graph, regardless of whether the input was a two-way graph or not (NOTE: This restriction could be lifted, if necessary). In addition, the original edge_ids are lost – the resulting graph will have edge_ids that are not related to the original ones. (NOTE: there certainly is a relation, but as long as no-one needs it it remains unspecified)

kwant.graph.remove_duplicates

kwant.graph.**remove_duplicates**()

Remove duplicate edges in the CGraph gr (this applies to the case where there are multiple edges (i,j), not to having (i,j) and (j,i)). This function modifes the graph in place.

If edge_weights is provided, edge_weights is modified such that the new edge weights are the sum of the old edge weights if there are duplicate edges.

This function only works on simple graphs (not two-way graphs), and it does not work on graphs which have a relation between the edge number (given by the order the edges are added) and the edge_id (given by the order the edges appear in the graph), see the documentation of CGraph. (Both restrictions could be lifted if necessary.) Furthermore, the function does not support negative node numbers, i.e. dangling links (the concept of being duplicate is more complicated there.)

kwant.graph.induced_subgraph

kwant.graph.**induced_subgraph**()

Return a subgraph of the CGraph gr by picking all nodes [0:gr.num_nodes] for which select is True. select can be either a NumPy array, or a function that takes the node number as input. This function returns a CGraph as well.

The nodes in the new graph are again numbered sequentially from 0 to num_nodes-1, where num_nodes is the number of nodes in the subgraph. The numbering is done such that the ordering of the node numbers in the original and the subgraph are preserved (i.e. if nodes n1 and n2 are both in the subgraph, and original node number of n1 < original node number of n2, then also subgraph node number n1 < subgraph node number n2).

If edge_weights is provided, the function also returns the edge weights for the subgraph which are simply a subset of the original weights.

This function returns a simple graph, regardless of whether the input was a two-way graph or not (NOTE: This restriction could be lifted, if necessary). Also, the resulting edge_ids are not related to the original ones in any way (NOTE: There certainly is a relation, but as long no-one needs it, we do not specify it). Also, negative nodes are discarded (NOTE: this restriction can also be lifted).

kwant.graph.print_graph

kwant.graph.**print_graph**()

4.2.3 Other

gint_dtype	Data type used for graph nodes and edges

4.3 kwant.linalg – Linear algebra routines

This package wraps some selected LAPACK functionality not available via NumPy and also contains a Python-wrapper for MUMPS. It also has several algorithms for finding approximately orthonormal lattice bases. It is meant for internal use by Kwant itself, but of course nothing prevents you from using it directly.

The documentation of this package is not included here on purpose in order not to add too many things to this reference. Please consult the source code directly.

CHAPTER

FIVE

MISCELLANEOUS MODULES

The following modules provide functionality for special applications.

5.1 kwant.digest – Random-access random numbers

Random-access random numbers

This module provides routines that given some input compute a "random" output that depends on the input in a (cryptographically) intractable way.

This turns out to be very useful when one needs to map some irregular objects to random numbers in a deterministic and reproducible way.

Internally, the md5 hash algorithm is used. The randomness thus generated is good enough to pass the "dieharder" battery of tests: see the function `test` of this module.

`kwant.digest.`**`uniform`**(*input*, *salt=''*)
md5-hash *input* and *salt* and map the result to the [0,1) interval.

input must be some object that supports the buffer protocol (i.e. a string or a numpy/tinyarray array). *salt* must be a string or a bytes object.

`kwant.digest.`**`gauss`**(*input*, *salt=''*)
md5-hash *input* and *salt* and return the result as a standard normal distributed variable.

input must be some object that supports the buffer protocol (i.e. a string or a numpy/tinyarray array). *salt* must be a string or a bytes object.

`kwant.digest.`**`test`**(*n=20000*)
Test the generator with the dieharder suite generating n**2 samples.

Executing this function may take a very long time.

5.2 kwant.rmt – Random matrix theory Hamiltonians

`kwant.rmt.`**`gaussian`**(*n*, *sym='A'*, *v=1.0*)
Make a n * n random Gaussian Hamiltonian.

Parameters **n** : int

Size of the Hamiltonian. It should be even for all the classes except A, D, and AI, and in class CII it should be a multiple of 4.

sym : one of 'A', 'AI', 'AII', 'AIII', 'BDI', 'CII', 'D', 'DIII', 'C', 'CI'

Altland-Zirnbauer symmetry class of the Hamiltonian.

v : float

Variance every degree of freedom of the Hamiltonian. The probaility distribution of the Hamiltonian is *P(H) = exp(-Tr(H^2) / 2 v^2)*.

Returns **h** : numpy.ndarray

A numpy array drawn from a corresponding Gaussian ensemble.

Notes

The representations of symmetry operators are chosen according to Phys. Rev. B 85, 165409.

Matrix indices are grouped first according to orbital number, then sigma-index, then tau-index.

Chiral (sublattice) symmetry C always reads: H = -tau_z H tau_z.

Time reversal symmetry T reads: AI: H = H^*. BDI: H = tau_z H^* tau_z. CI: H = tau_x H^* tau_x. AII, CII: H = sigma_y H^* sigma_y. DIII: H = tau_y H^* tau_y.

Particle-hole symmetry reads: C, CI: H = -tau_y H^* tau_y. CII: H = -tau_z sigma_y H^* tau_z sigma_y. D, BDI: H = -H^*. DIII: H = -tau_x H^* tau_x.

This implementation should be sufficiently efficient for large matrices, since it avoids any matrix multiplication.

`kwant.rmt.`**`circular`**(*n*, *sym='A'*, *charge=None*)

Make a n * n matrix belonging to a symmetric circular ensemble.

Parameters **n** : int

Size of the matrix. It should be even for the classes C, CI, CII, AII, DIII (either T^2 = -1 or P^2 = -1).

sym : one of 'A', 'AI', 'AII', 'AIII', 'BDI', 'CII', 'D', 'DIII', 'C', 'CI'

Altland-Zirnbauer symmetry class of the matrix.

charge : int or None

Topological invariant of the matrix. Should be one of 1, -1 in symmetry classes D and DIII, should be from 0 to n in classes AIII and BDI, and should be from 0 to n / 2 in class CII. If charge is None, it is drawn from a binomial distribution with p = 1 / 2.

Returns **s** : numpy.ndarray

A numpy array drawn from a corresponding circular ensemble.

Notes

The representations of symmetry operators are chosen according to Phys. Rev. B 85, 165409, except class D.

Matrix indices are grouped first according to channel number, then sigma-index.

Chiral (sublattice) symmetry C always reads: s = s^+.

Time reversal symmetry T reads: AI, BDI: r = r^T. CI: r = -sigma_y r^T sigma_y. AII, DIII: r = -r^T. CII: r = sigma_y r^T sigma_y.

Particle-hole symmetry reads: CI: r = -sigma_y r^* sigma_y C, CII: r = sigma_y r^* sigma_y D, BDI: r = r^*. DIII: -r = r^*.

This function uses QR decomposition to probe symmetric compact groups, as detailed in arXiv:math-ph/0609050. For a reason as yet unknown, scipy implementation of QR decomposition also works for symplectic matrices.

- *genindex*

K

L

M

N

O

P

R

S

T

U

V

W