Kwant 1.4.1 documentation

Release 1.4.1

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

Apr 11, 2019

CONTENTS

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 1.4

This article explains the user-visible changes in Kwant 1.4.0. Subsequently, the user-visible changes for each maintenance release of the 1.4.x series are listed (if there were any).

Summary: release highlights

- Adding magnetic field to systems, even in complicated cases, is now specially *supported*.
- The *KPM module can now calculate conductivities*.
- The Qsymm library for Hamiltonian symmetry analysis has been *integrated*.
- The handling of system parameters has been *improved* and optimized.
- Plotting has been improved, most notably through the addition of a *routine that plots densities with interpolation*.
- *Installing Kwant on Windows* is now much easier thanks to Conda packages.

Backwards-incompatible changes:

- *Restrictions on value functions when named parameters are given*

Automatic Peierls phase calculation

When defining systems with orbital magnetic fields it is often cumbersome to manually calculate the phases required by the Peierls substitution, and to ensure that the chosen gauge is consistent across the whole system (this is especially true for systems with leads that point in different directions). This release introduces *`kwant.physics.magnetic_gauge`*, which calculates the Peierls phases for you:

```
def hopping(a, b, t, peierls):
    return -t * peierls(a, b)

def B_syst(pos):
   return np.exp(-np.sum(pos * pos))

syst = make_system(hopping)
lead = make_lead(hopping).substituted(peierls='peierls_lead')
syst.attach_lead(lead)
syst = syst.finalized()

gauge = kwant.physics.magnetic_gauge(syst)

# B_syst in scattering region, 0 in lead.
peierls_syst, peierls_lead = gauge(B_syst, 0)

params = dict(t=1, peierls=peierls_syst, peierls_lead=peierls_lead)
kwant.hamiltonian_submatrix(syst, params=params)
```

Note that the API for this functionality is provisional, and may be revised in a future version of Kwant.

Conductivity calculations using `kwant.kpm.conductivity`

Kwant now has the ability to calculate conductivities using a combination of the Kernel Polynomial Method and the Kubo-Bastin relation. The following example calculates σ_{xy} for various chemical potentials at finite temperature:

```
syst = make_system().finalized()
sigma_xy = kwant.kpm.conductivity(syst, alpha='x', beta='y')
conductivities = [sigma_xy(mu=mu, temperature=0.1)
                  for mu in np.linspace(0, 4)]
```

Integration with Qsymm library

Kwant now contains an integration with the Qsymm library for analysing model symmetries. This functionality is available under `kwant.qsymm`. Here is an example for extracting the symmetry group of a graphene system:

```
import numpy as np
import kwant
import kwant.qsymm

s_0 = np.eye(2)

lat = kwant.lattice.honeycomb(norbs=[1, 1])
sym = kwant.TranslationalSymmetry(lat.vec((1, 0)), lat.vec((0, 1)))

graphene = kwant.Builder(sym)
graphene[[lat.a(0, 0), lat.b(0, 0)]] = 0
graphene[lat.neighbors()] = 1

symmetry_generators = kwant.qsymm.find_builder_symmetries(graphene)

# Let's find what the chiral symmetry looks like

def is_chiral(g):
  return g.antisymmetry and not g.conjugate and np.allclose(g.R, s_0)

print(next(g for g in symmetry_generators if is_chiral(g)))
```

`kwant.qsymm` also contains functionality for converting Qsymm models to Kwant Builders, and vice versa, and for working with continuum Hamiltonians (such as would be used with `kwant.continuum`). This integration requires separately installing Qsymm, which is available on the Python Package Index.

System parameter substitution

After the introduction of `Builder.fill` it has become possible to construct Kwant systems by first creating a "model" system with high symmetry and then filling a lower symmetry system with this model. Often, however, one wants to use different parameter values in different parts of a system. In previous versions of Kwant this was difficult to achieve.

Builders now have a method `substituted` that makes it easy to substitute different names for parameters. For example if a builder `model` has a parameter `V`, and one wishes to have different values for `V` in the scattering region and leads, one could do the following:

```
syst = kwant.Builder()
syst.fill(model.substituted(V='V_dot'), ...))

lead = kwant.Builder()
lead.fill(model.substituted(V='V_lead'), ...)

syst.attach_lead(lead)
syst = syst.finalized()

kwant.smatrix(syst, params=dict(V_dot=0, V_lead=1))
```

System parameters can now be inspected

In modern Kwant the preferred way to pass arguments to your models is by *name*, using a dictionary and `params`:

```
def onsite(site, magnetic_field, voltage):
    return magnetic_field * sigma_z + voltage * sigma_0

def lead_onsite(site, lead_voltage):
    return lead_voltage * sigma_0

syst = make_system(onsite)
syst.attach_lead(make_lead(lead_onsite))
```

(continues on next page)

(continued from previous page)

```
syst = syst.finalized()

# naming the arguments makes things clear!
kwant.smatrix(syst, params=dict(magnetic_field=0.5, voltage=1,
                                lead_voltage=0.2))
```

This is a much clearer and less error prone than passing arguments by *position* using `args`, as was required in older versions of Kwant. In this version of Kwant we introduce the `parameters` attribute of *finalized systems*, which allows inspection of the names of the parameters that the system (and its leads) expects:

```
>>> syst.parameters
frozenset({'magnetic_field', 'voltage'})
>>> syst.leads[0].parameters
frozenset({'V_lead'})
```

This is a provisional API that may be changed in a future version of Kwant.

Passing system arguments via `args` is deprecated in favor of `params`

It is now deprecated to pass arguments to systems by providing the `args` parameter (in `kwant.smatrix` and elsewhere). Passing arguments via `args` is error prone and requires that all value functions take the same formal parameters, even if they do not depend on all of them. The preferred way of passing parameters to Kwant systems is by passing a dictionary using `params`:

```
def onsite(site, magnetic_field, voltage):
  return magnetic_field * sigma_z + voltage * sigma_0

syst = make_system(onsite).finalized()

kwant.smatrix(syst, params=dict(magnetic_field=0.5, voltage=0.2))

# Compare this to the deprecated 'args'
kwant.smatrix(syst, args=(0.5, 0.2))
```

Providing `args` will be removed in a future Kwant version.

Interpolated density plots

A new function, *`kwant.plotter.density`*, has been added that can be used to visualize a density defined over the sites of a Kwant system. This convolves the "discrete" density (defined over the system sites) with a "bump" function in realspace. The output of *`density`* can be more informative that *`map`* when plotting systems with many sites, where it is not important to see the individual contribution from each site.

Configurable maximum velocity in stream plots

The function *`streamplot`* has got a new option `vmax`. Note that this option is not available in *`current`*. In order to use it, one has to call `streamplot` directly as shown in the docstring of `current`.

Improved heuristic for colorscale limits in `kwant.plotter.map`

Previously *`map`* would set the limits for the color scale to the extrema of the data being plotted when `vmin` and `vmax` were not provided. This is the behaviour of `matplotlib.imshow`. When the data to be plotted has very sharp and high peaks this would mean that most of the data would appear near the bottom of the color scale, and all of the features would be washed out by the presence of the peak. Now *`map`* employs a heuristic for setting the colorscale when there are outliers, and will emit a warning when this is detected.

Sites from different families are plotted in different colors by default

Previously *kwant.plotter.plot* would plot all sites in black. Now sites from different families are plotted in different colors, which improves the default plotting style. You can still customize the site coloring using the `site_color` parameter, as before.

`kwant.physics.Bands` can optionally return eigenvectors and velocities

kwant.physics.Bands now takes extra parameters that allow it to return the mode eigenvectors, and also the derivatives of the dispersion relation (up to second order) using the Hellman-Feynman relation:

```
syst = make_system().finalized()

bands = kwant.physics.Bands(syst)
(energies, velocities, vectors) = bands(k=0, derivative_order=1,
                                        return_eigenvectors=True)
```

Finalized Builders keep track of which sites were added when attaching leads

When attaching leads to an irregularly shaped scattering region, Kwant adds sites in order to make the interface with the leads "smooth". Previously, the information of which sites were added was not inspectable after finalization. Now the sites that were added from each lead are available in the `lead_paddings` attribute. See the documentation for *FiniteSystem* for details.

`kwant.continuum.discretize` can be used with rectangular lattices

Previously the discretizer could only be used with lattices with the same lattice constant in all directions. Now it is possible to pass rectangular lattices to the discretizer:

```
kwant.continuum.discretize(
  'k_x**2 + k_y**2',
  grid=kwant.lattice.general([(1, 0), (0, 2)]),
)
```

This is useful when you need a finer discretization step in some spatial directions, and a coarser one in others.

Restrictions on value functions when named parameters are given

New restrictions apply to how value functions may accept arguments, when named parameters are given through `params`. (Nothing changes when the now deprcated `args` mechanism is used). The restrictions follow the principle that each value function must take a clearly specified set of named parameters. This allows to make the parameter handling less error-prone and faster.

In particular, when `params` is used, it is no longer possible for value functions to - take `*args` or `**kwargs`, - take keyword-only parameters, - have default parameters for arguments.

As an example, the following snippet no longer works because it uses default values:

```
syst = kwant.Builder()

# Parameter 't' has a default value of 1
def onsite(site, V, t=1):
  return V = 2 * t

def hopping(site_a, site_b, t=1):
  return -t

syst[...] = onsite
syst[...] = hopping

# Raises ValueError
syst = syst.finalized()
```

As a solution, simply remove the default values and always provide `t`. To deal with many parameters, the following idiom may be useful:

```
defaults = dict(a=0, b=1, c=2, d=3)
...
smatrix = kwant.smatrix(syst, E, params=dict(defaults, d=4, e=5))
```

Note that this allows to override defaults as well as to add additional parameters.

Installation on Microsoft Windows is available via Conda

Kwant is now packaged for the Conda package manager on Windows, and using Conda is the preferred method for installing Kwant on that platform. Please refer to the installation section of the Kwant website for details. Currently the MUMPS solver is not available for the Windows version of the Conda package; we hope to include MUMPS support in a later patch release.

Minimum required versions for some dependencies have increased

Kwant now requires at least the following versions:

- Python 3.5
- numpy 0.11.0
- scipy 0.17.0
- matplotlib 1.5.1

These versions (or newer) are available in the latest stable releases of Ubuntu and Debian GNU/Linux.

Changes in Kwant 1.4.1

- The list of user-visible changes was rearranged to emphasize backwards-incompatible changes by moving them to the top of the list.
- Restrictions on value functions no longer apply when the old `args` mechanism is used, this restores most of the backwards compatibility with previous Kwant versions: *Restrictions on value functions when named parameters are given*.
- The `args` parameter passing mechanism works again with *`wraparound`*-treated systems. Some restriction continue to appply, notably it is not possible to use `wraparound` with value functions that take `*args` or `*kwargs`.
- Kwant no longer requires the existence of a *parameters* attribute for low-level systems.
- A note about an *API change that affects low-level systems* that ocurred in Kwant 1.3 was added.

1.2.2 What's new in Kwant 1.3

This article explains the user-visible changes in Kwant 1.3.0, released on 19 May 2017. See also the full list of changes up to the most recent bugfix release of the 1.3 series.

Using high-symmetry builders as models

Builders now have a *`fill`* method that fills a builder instance with copies of a template builder. This can be used to "cut out" shapes from high-symmetry models, or to increase the symmetry period of a lead.

Thus Kwant gains the new concept of a "model". Models may be created manually, or with the new function *`kwant.continuum.discretize`* (see next paragraph). There is also support for finalizing models and e.g. calculating their band structure (see *Finalizing builders with multiple translational symmetries*).

Tools for continuum Hamiltonians

The new sub-package *continuum* is a collection of tools for working with continuum models and for discretizing them into tight-binding models. It aims at providing a handy interface to convert symbolic Hamiltonians into a builder with N-D translational symmetry that can be use to calculate tight-binding band structures or construct systems with different/lower symmetry. For example in just a few lines we can construct a two-band model that exhibits a quantum anomalous spin Hall phase:

```
def make_model(a):
    ham = ("alpha * (k_x * sigma_x - k_y * sigma_y)"
           "+ (m + beta * kk) * sigma_z"
           "+ (gamma * kk + U) * sigma_0")
    subs = {"kk": "k_x**2 + k_y**2"}
    return kwant.continuum.discretize(ham, locals=subs, grid=a)
```

From: `QAHE example script`

See the tutorial: *Discretizing continuous Hamiltonians*

See the reference documentation: *kwant.continuum – Tools for continuum systems*

Calculating charges and currents using the operator module

Often one may wish to calculate quantities that are defined over sites of the system (such as charge density, spin density along some axis etc), or over hoppings of the system (such as current or spin current). With the introduction of the `operator` module it has now become much easier to calculate such quantities. To obtain the regular density and current everywhere in a system due to a wavefunction `psi`, one only needs to do the following:

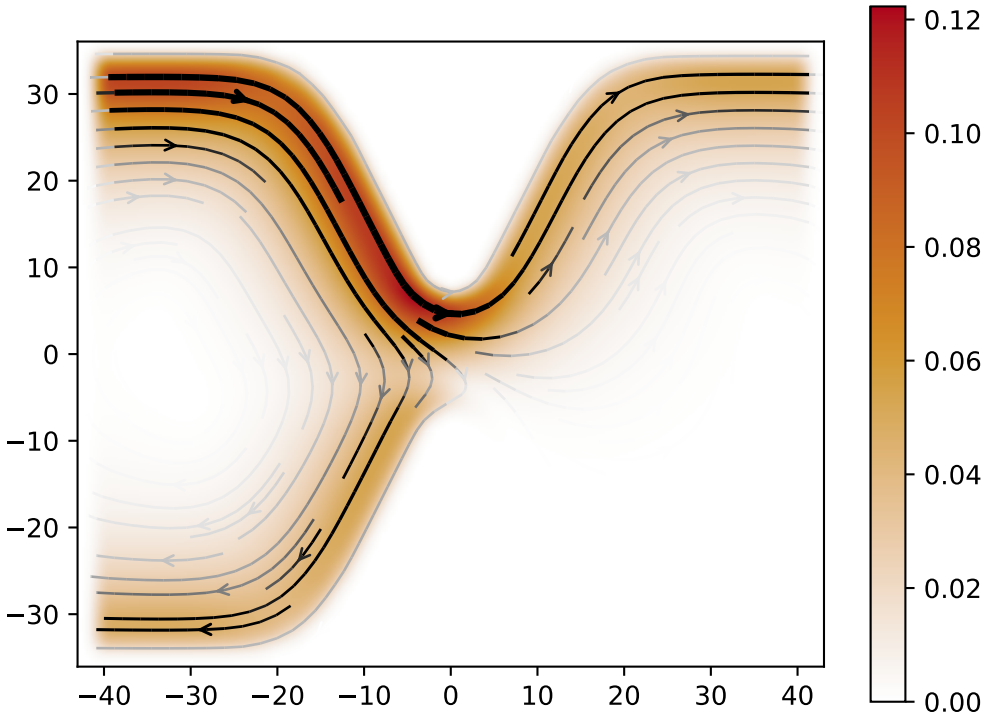
```
syst = make_system().finalized()
psi = kwant.wave_function(syst)(0)[0]

# create the operators
Q = kwant.operator.Density(syst)
J = kwant.operator.Current(syst)

# evaluate the expectation value with the wavefunction
q = Q(psi)
j = J(psi)
```

See the tutorial: *Computing local quantities: densities and currents*

Plotting of currents

Quantities defined on system hoppings (e.g. currents calculated using *Current*) can be directly plotted as a streamplot over the system using *kwant.plotter.current*. This is similar to how *kwant.plotter.map* can be used to plot quantities defined on sites. The example below shows edge states of a quantum anomalous Hall phase of the two-band model shown in the *above section*:

```
    J = kwant.operator.Current(syst).bind(params=params)
    current = sum(J(p) for p in psi)
    kwant.plotter.current(syst, current)
```



From: `QAHE example script`

Scattering states with discrete symmetries and conservation laws

Given a lead Hamiltonian that has a conservation law, it is now possible to construct lead modes that have definite values of the conservation law. This is done by declaring projectors that block diagonalize the Hamiltonian before the modes are computed. For a Hamiltonian that has one or more of the three fundamental discrete symmetries (time-reversal symmetry, particle-hole symmetry and chiral symmetry), it is now possible to declare the symmetries in Kwant. The symmetries are then used to construct scattering states that are properly related by symmetry. The discrete symmetries may be combined with conservation laws, such that if different blocks of the Hamiltonian are related by a discrete symmetry, the lead modes are computed to reflect this.

See the updated tutorial: *Superconductors: orbital degrees of freedom, conservation laws and symmetries*

Named parameters for value functions

In Kwant < 1.3 whenever Hamiltonian values were provided as functions, they all had to take the same extra parameters (after the site(s)) regardless of whether or not they actually used them at all. For example, if we had some onsite potential and a magnetic field that we model using the Peierls substitution, we would have to define our value functions like so:

```
# formally depends on 'B', but 'B' is never used
def onsite(site, V, B):
    return V

# formally depends on 'V', but 'V' is never used
def hopping(site_a, site_b, V, B):
    return (site_b.pos[1] - site_a.pos[1]) * B
```

This was because previously extra arguments were provided to the system by passing them as a sequence via the `args` parameter to various Kwant functions (e.g. `kwant.smatrix` or `hamiltonian_submatrix`).

In Kwant 1.3 it is now possible for value functions to depend on different parameters, e.g.:

```
def onsite(site, V):
    return V

def hopping(site_a, site_b, B):
    return (site_b.pos[1] - site_a.pos[1]) * B
```

If you make use of this feature then you must in addition pass your arguments via the `params` parameter. The value provided to `params` must be a `dict` that maps parameter names to values, e.g.:

```
kwant.smatrix(syst, params=dict(B=0.1, V=2))
```

as opposed to the old way:

```
kwant.smatrix(syst, args=(2, 0.1))
```

Passing a dictionary of parameters via `params` is now the recommended way to provide parameters to the system.

Reference implementation of the kernel polynomial method

The kernel polynomial method is now implemented within Kwant to obtain the density of states or, more generally, the spectral density of a given operator acting on a system or Hamiltonian.

See the tutorial: *Calculating spectral density with the kernel polynomial method*

See the reference documentation: *kwant.kpm – Kernel Polynomial Method*

Finalizing builders with multiple translational symmetries

While it remains impossible to finalize a builder with more than a single direction of translational symmetry, the `wraparound` module has been added as a temporary work-around until the above limitation gets lifted.

The function *wraparound* transforms all (or all but one) translational symmetries of a given builder into named momentum parameters *k_x*, *k_y*, etc. This makes it easy to compute transport through systems with periodic boundary conditions or across infinite planes.

Plotting the 2-d band structure of graphene is now as straightforward as:

```
from matplotlib import pyplot
import kwant

lat = kwant.lattice.honeycomb()
sym = kwant.TranslationalSymmetry(lat.vec((1, 0)), lat.vec((0, 1)))

bulk = kwant.Builder(sym)
bulk[ [lat.a(0, 0), lat.b(0, 0)] ] = 0
bulk[lat.neighbors()] = 1
wrapped = kwant.wraparound.wraparound(bulk).finalized()
kwant.wraparound.plot_2d_bands(wrapped)
```

Consistent ordering of sites in finalized builders

In Python 3 the internal ordering of dictionaries is not deterministic. This meant that running a Kwant script twice would produce systems with different ordering of sites, which leads to non-reproducible calculations. Now, sites in finalized builders are always ordered first by their site family, then by their tag.

Coincidentally, this means that you can plot a wavefunction in a simple 1D system by just saying:

```
lattice_1D = chain()
syst = make_system(lattice_1D)
h = syst.hamiltonian_submatrix()
pyplot.plot(np.eigs(h)[1][0])
```

attach_lead() can now handle leads with greater than nearest-neighbor hoppings

When attaching a lead with greater than nearest-neighbor hoppings, the symmetry period of the finalized lead is suitably extended and the unit cell size is increased.

Pickling support

It is now possible to pickle and unpickle *Builder* and *System* instances.

Improved build configuration

The name of the build configuration file, `build.conf` by default, is now configurable with the `--configfile=PATH` option to `setup.py`. (This makes build configuration usable with the `pip` tool.) The build configuration as specified in this file is now more general, allowing to modify any build parameter for any of the compiled extensions contained in Kwant. See the *Installation instructions* for details.

Builder.neighbors() respects symmetries

Given a site, the method *neighbors* of *Builder* returns an iterator over sites that are connected by a hopping to the provided site. This is in contrast to previous versions of Kwant, where the neighbors were yielded not of the provided site, but of it's image in the fundamental domain.

This change is documented here for completeness. We expect that the vast majority of users of Kwant will not be affected by it.

API change that affects low-level systems

The *hamiltonian* method of low-level systems must now accept a *params* keyword parameter.

1.2.3 What's new in Kwant 1.2

This article explains the user-visible changes in Kwant 1.2.2, released on 9 December 2015. See also the full list of changes up to the most recent bugfix release of the 1.2 series.

Kwant 1.2 is identical to Kwant 1.1 except that it has been updated to run on Python 3.4 and above. Bugfix releases for the 1.1 and 1.2 series will mirror each other, i.e. 1.1.3 and 1.2.3 will fix the same bugs.

Starting with Kwant 1.2, all Kwant development will target Python 3. We plan, however, to maintain Python 2 support with the 1.1 series for several years.

How to upgrade Kwant scripts to Python 3

Even though the interface and functionality of Kwant remain unchanged between versions 1.1 and 1.2, scripts using Kwant need to be updated to Python 3. This can be done by running the automatic conversion tool on the command line:

```
2to3 -w example.py
```

(The above command will rename the original file to `example.py.bak`.) The necessary changes are typically only superficial, as described in What's New In Python 3.0.

1.2.4 What's new in Kwant 1.1

This article explains the user-visible changes in Kwant 1.1.0, released on 21 October 2015. See also the full list of changes up to the most recent bugfix release of the 1.1 series.

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:

```
syst[syst.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 *smatrix* or *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 `kwant.plotter.map`

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

1.2.5 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:
    syst[syst.possible_hoppings(*kind)] = t
```

now it suffices to write

```
syst[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 *kwant.lattice.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 degrees of freedom, conservation laws and symmetries*, 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(syst, 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
```

(continues on next page)

(continued from previous page)

```
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(syst, 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.

1.2.6 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 degrees of freedom, conservation laws and symmetries

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 of Kwant

Ready-to-use Kwant packages are available for many platforms (like GNU/Linux, Mac OS X, Microsoft Windows). See the installation page of the Kwant website for instructions on how to install Kwant on your platform. This is the recommended way for new users.

The remainder of this section documents how to build Kwant from source. This information is mostly of interest to contributors and packagers.

1.3.1 Generic instructions

Obtaining the source code

Source distributions of Kwant (and Tinyarray) are available at the downloads section of the Kwant website as well as PyPI. The sources may be also cloned directly from the official Kwant git repository.

Prerequisites

Building Kwant requires

- Python 3.5 or above (Kwant 1.1 is the last version to support Python 2),
- NumPy 1.11.0 or newer,
- SciPy 0.17.0 or newer,
- LAPACK and BLAS, (For best performance we recommend the free OpenBLAS or the nonfree MKL.)
- Tinyarray 1.2 or newer,

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 1.5.1 or newer, for the module *kwant.plotter* and the tutorial,
- SymPy 0.7.6 or newer, for the subpackage *kwant.continuum*.
- Qsymm 1.1.0 or newer, for the subpackage *kwant.qsymm*.
- 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 py.test testing framework 2.8 or newer for running the tests included with Kwant.

In addition, to build a copy of Kwant that has been checked-out directly from version control, 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.

Building and installing Kwant

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

```
python3 setup.py build
python3 setup.py install
```

Depending on your system, you might have to run the second command with administrator privileges (e.g. prefixing it with `sudo`).

After installation, tests can be run with:

```
python3 -c 'import kwant; kwant.test()'
```

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

(Cython will be run automatically when the source tree has been checked out of version control. Kwant tarballs include the Cython-generated files, and cythonization is disabled when building not from git. If ever necessary, this default can be overridden by giving the `--cython` option to setup.py.)

Build configuration

Kwant contains several extension modules. The compilation and linking of these modules can be configured by editing a build configuration file. By default, this file is `build.conf` in the root directory of the Kwant distribution. A different path may be provided using the `--configfile=PATH` option.

This configuration file consists of sections, one for each extension module that is contained in Kwant, led by a `[section name]` header and followed by `key = value` lines.

The sections bear the names of the extension modules, for example `[kwant.operator]`. There can be also a `[DEFAULT]` section that provides default values for all extensions, also those not explicitly present in the file.

Possible keys are the keyword arguments for `distutils.core.Extension` (For a complete list, see its documentation). The corresponding values are whitespace-separated lists of strings.

Example `build.conf` for compiling Kwant with C assertions and Cython's line trace feature:

```
[DEFAULT]
undef_macros = NDEBUG
define_macros = CYTHON_TRACE=1
```

Kwant can optionally be linked against MUMPS. The main application of build configuration is adopting the build process to the various deployments of MUMPS. MUMPS will be not linked against by default, except on Debian-based systems when the package `libmumps-scotch-dev` is installed.

The section `[kwant.linalg._mumps]` may be used to adapt the build process. (For simplicity and backwards compatibility, `[mumps]` is an aliases for the above.)

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
```

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 `python3 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.

When `make html` is run, modified tutorial example scripts are executed to update any figures that might have changed. The machinery behind this works as follows. The canonical source for a tutorial script, say `graphene.py` is the file `doc/source/images/graphene.py.diff`. This diff file contains the information to recreate two versions of `graphene.py`: a version that is presented in the documentation (`doc/source/tutorial/graphene.py`), and a version that is used to generate the figures for the documentation (`doc/source/images/graphene.py`). Both versions are related but differ e.g. in the details of the plotting. When `make html` is run, both versions are extracted form the diff file.

The diff file may be modified directly. Another possible way of working is to directly modify either the tutorial script or the figure generation script. Then `make html` will use the command line tool wiggle to propagate the modifications accordingly. This will often just work, but may sometimes result in conflicts, in which case a message will be printed. The conflicts then have to be resolved much like with a version control system.

1.3.2 Hints for specific platforms

Unix-like systems (GNU/Linux)

Kwant should run on all recent Unix-like systems. The following instructions have been verified to work on Debian 8 (Jessie) or newer, and on Ubuntu 14.04 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 python3-dev python3-setuptools python3-scipy python3-matplotlib␣
↪python3-pytest python3-sympy g++ gfortran libmumps-scotch-dev
```

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

```
python3 setup.py build
sudo python3 setup.py install
```

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

```
python3 setup.py build
sudo python3 setup.py install
```

By default the package will be installed under `/usr/local`. Run `python3 setup.py --help install` for installation options.

Mac OS X: MacPorts

The following instructions are valid for Kwant 1.1 with Python 2.7. They need to be updated for Kwant 1.2. (Help is welcome.)

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. Install a recent version of MacPorts, as explained in the installation 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
```

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

```
[mumps]
include_dirs = /opt/local/include
library_dirs = /opt/local/lib
libraries = zmumps_seq mumps_common_seq pord_seq esmumps scotch scotcherr mpiseq gfortran
```

5. 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

The following instructions are valid for Kwant 1.1 with Python 2.7. They need to be updated for Kwant 1.2. (Help is welcome.)

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 kwant-project/kwant
pip install pytest pytest-runner 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).

Microsoft 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.

1.4 Authors of Kwant

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

The principal developers of Kwant are

- Christoph Groth (CEA Grenoble)
- Michael Wimmer (TU Delft)
- Anton Akhmerov (TU Delft)
- Xavier Waintal (CEA Grenoble)
- Joseph Weston (TU Delft)

Contributors to Kwant include

- Jörg Behrmann (FU Berlin)
- Paul Clisson (CEA Grenoble)
- Mathieu Istas (CEA Grenoble)
- Daniel Jaschke (CEA Grenoble)
- Bas Nijholt (TU Delft)
- Michał Nowak (TU Delft)
- Viacheslav Ostroukh (Leiden University)
- Pablo Pérez Piskunow (TU Delft)
- Tómas Örn Rosdahl (TU Delft)
- Sebastian Rubbert (TU Delft)
- Rafał Skolasiński (TU Delft)
- Adrien Sorgniard (CEA Grenoble)
- Dániel Varjas (TU Delft)
- Thomas Kloss (CEA Grenoble)
- Pierre Carmier (CEA Grenoble)

We thank Christoph Gohlke for the creation of installers for Microsoft Windows.

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

Research related to Kwant was funded by

- the US Office of Naval Research,
- the European Research Council,
- the Netherlands Organisation for Scientific Research NWO (formerly NWO/FOM).
- the French National Agency for Research (ANR)
- the Future and Emerging Technologies (FET) & Information and Communication Technologies (ICT) actions
- Lawrence Golub fellowship

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. Contributions to Kwant are highly welcome. You can help the project not only by writing code, but also by reporting bugs, and fixing/improving the website and the documentation. Please see the "Contributing" page of the Kwant website for more information.

1.7 Kwant license

(CEA = Commissariat à l'énergie atomique et aux énergies alternatives)

IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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 *Citing Kwant*.

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

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

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 `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:

```
syst = 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: `syst[lat(i, j)] = ...` sets the on-site energy for the point *(i, j)*, and `syst[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 range(L):
    for j in range(W):
        # On-site Hamiltonian
        syst[lat(i, j)] = 4 * t

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

        # Hopping in x-direction
        if i > 0:
            syst[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

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

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

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

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

and "Hopping in y-direction" implements

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

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 range(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

```
syst.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 range(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

syst.attach_lead(right_lead)
```

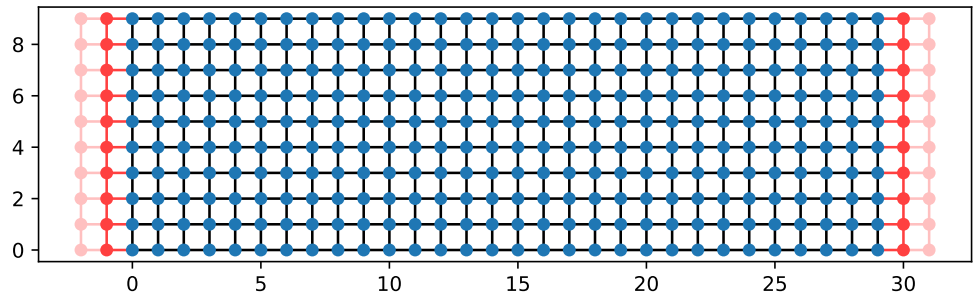
Note that here we added points with x-coordinate 0, just as for the left lead. You might object that the right lead should be placed L (or $L+1$?) points to the right with respect to the left lead. In fact, you do not need to worry about that.

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

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

```
kwant.plot(syst)
```

This should bring up this picture:



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

```
syst = syst.finalized()
```

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

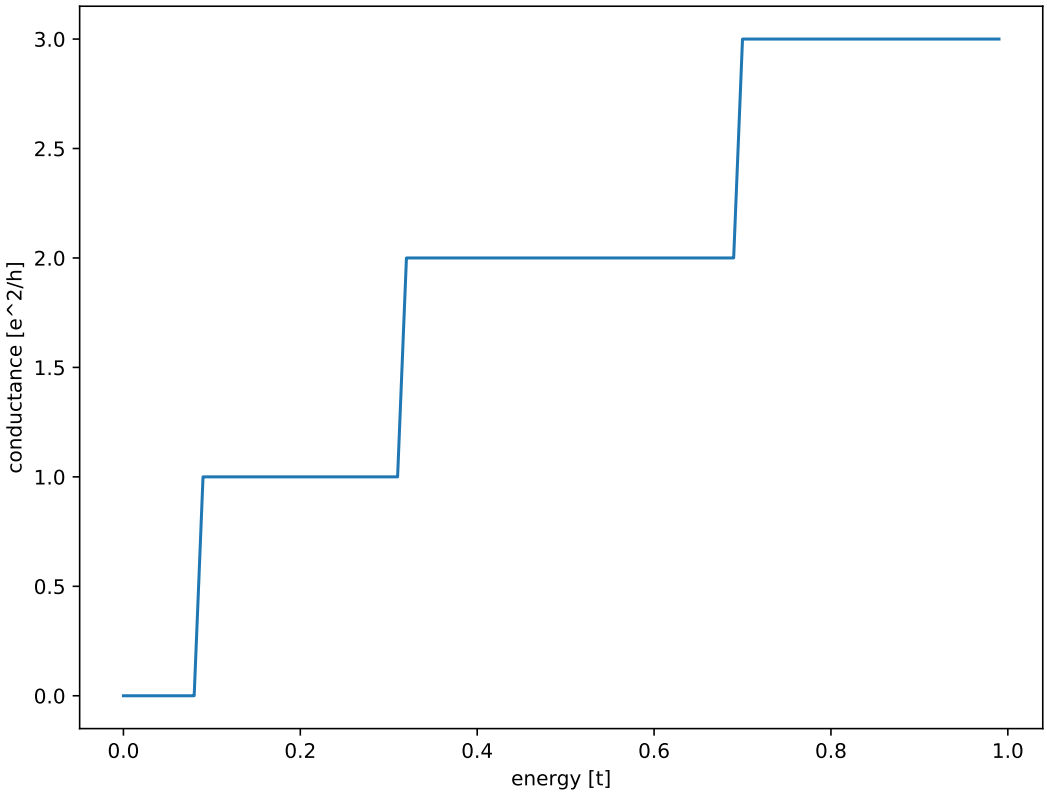
```
energies = []
data = []
for ie in range(100):
    energy = ie * 0.01

    # compute the scattering matrix at a given energy
    smatrix = kwant.smatrix(syst, 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. `syst[lat(i, j), lat(i, j-1)] = ...` and not also `syst[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:

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

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

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

  is valid code. In the latter case, the hopping `syst[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

```
syst = syst.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

```
fsyst = syst.finalized()
del syst
syst = fsyst
```

- 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`.
- Due to matplotlib's limitations, Kwant's plotting routines have the side effect of fixing matplotlib's "backend". If you would like to choose a different backend than the standard one, you must do so before asking Kwant to plot anything.

2.2.3 Building the same system with less code

See also:

The complete source code of this example can be found in `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)

    syst = kwant.Builder()
```

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

```
    syst[(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 range(L) for j in range(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:

```
    syst[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()`. `syst[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 `syst[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 range(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:

```
    syst.attach_lead(lead)
    syst.attach_lead(lead.reversed())

    return syst
```

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

```
def plot_conductance(syst, energies):
    # Compute conductance
    data = []
    for energy in energies:
        smatrix = kwant.smatrix(syst, 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():
    syst = make_system()

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

(continues on next page)

(continued from previous page)

```
    # Finalize the system.
    syst = syst.finalized()

    # We should see conductance steps.
    plot_conductance(syst, energies=[0.01 * i for i in range(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`):

```
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

  ```
      syst[(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 range(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
  ```

  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:

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

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

or:

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

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

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

instead of

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

In fact, due to the way python handles subscripting, `syst[site1, site2]` is the same as `syst[(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 `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:

```
    syst[(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
    syst[kwant.builder.HoppingKind((1, 0), lat, lat)] = \
        -t * sigma_0 + 1j * alpha * sigma_y / 2
    # hoppings in y-directions
    syst[kwant.builder.HoppingKind((0, 1), lat, lat)] = \
        -t * sigma_0 - 1j * alpha * sigma_x / 2
```

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)

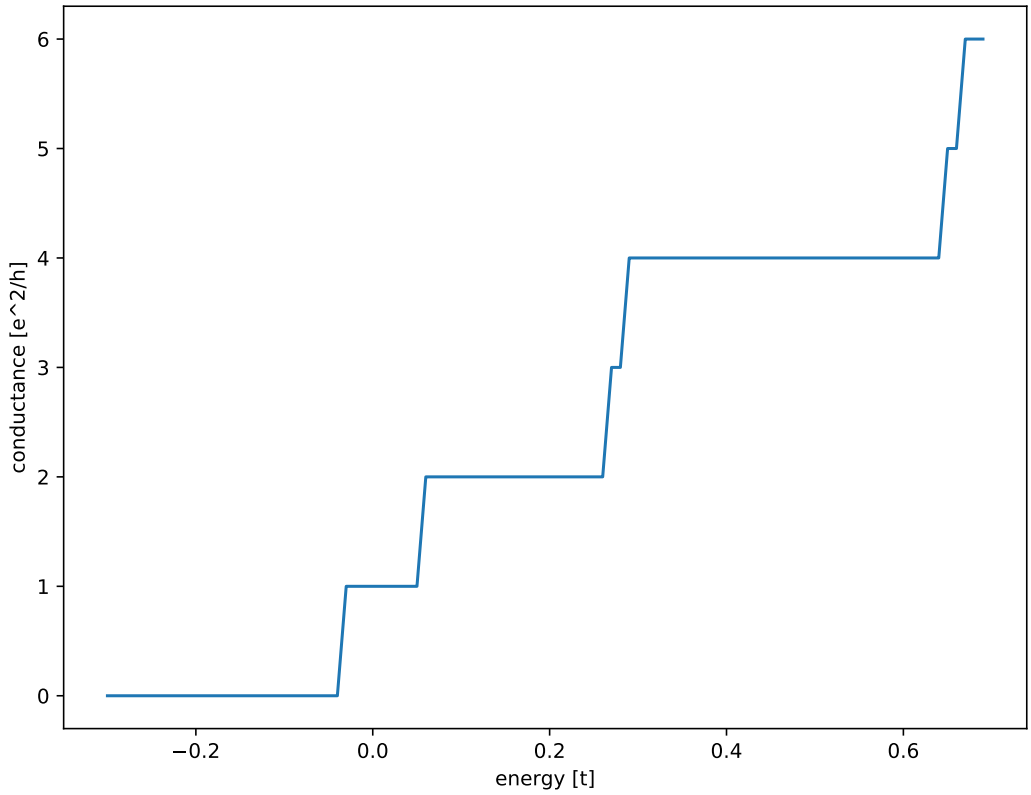
Since we are only dealing with a single lattice here, source and target lattice are identical, but still must be specified (for an example with hopping between different (sub)lattices, see *Beyond square lattices: graphene*).

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 range(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 / 2
    # hoppings in y-directions
    lead[kwant.builder.HoppingKind((0, 1), lat, lat)] = \
        -t * sigma_0 - 1j * alpha * sigma_x / 2
```

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 `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)

    syst = 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):
        return 4 * t + potential(site, pot)

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

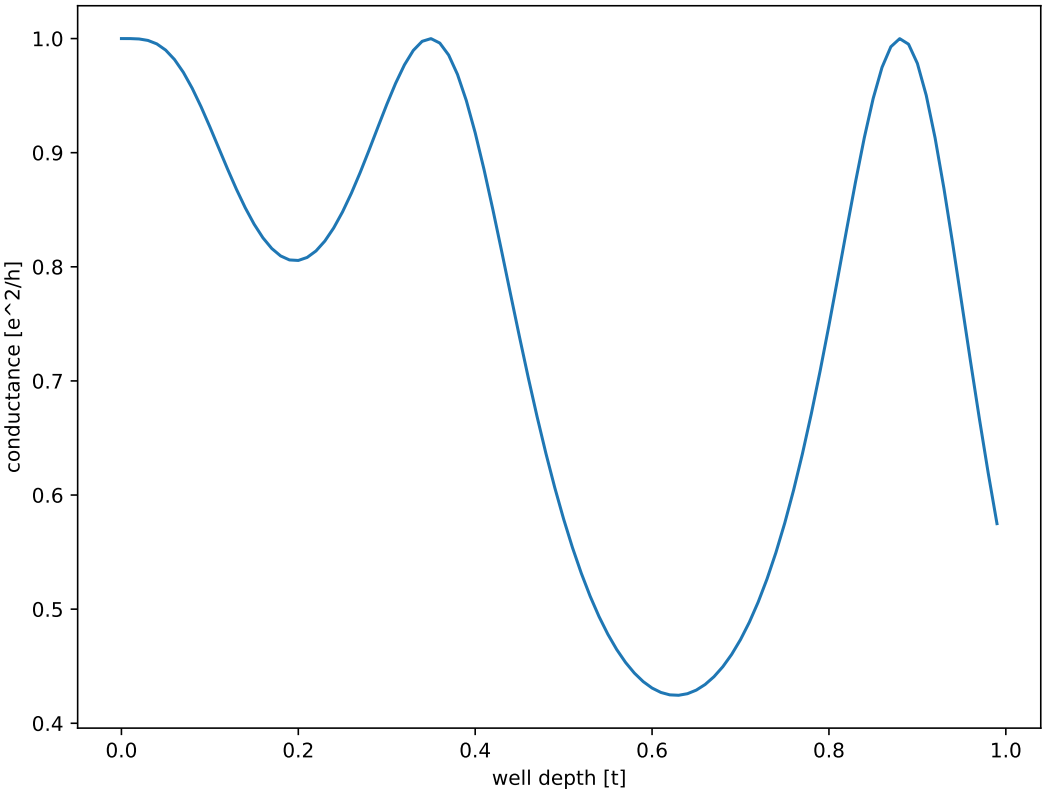
For each lattice point, the corresponding site is then passed as the first argument to the function *onsite*. The values of any additional parameters, which can be used to alter the Hamiltonian matrix elements at a later stage, are specified later during the call to *smatrix*. Note that we had to define *onsite*, as it is not possible to mix values and functions as in `syst[...] = 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(syst, energy, params=dict(pot=-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 dictionary, *params*, that contains the additional arguments required by 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 (remember above we defined our *onsite* function that takes a parameter named *pot*). 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

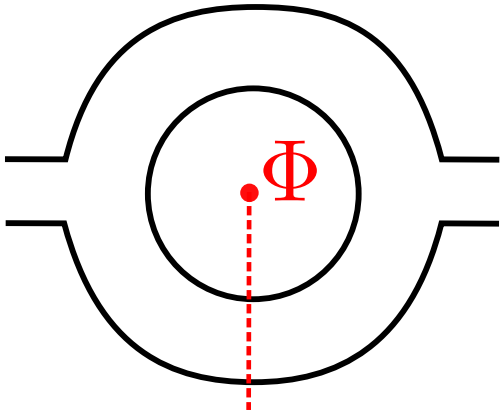
- Functions can also be used for hoppings. In this case, they take two *Site*'s as arguments and then an arbitrary number of additional arguments.
- Apart from the real-space position *pos*, *Site* has also an attribute *tag* containing the lattice indices of the site.

2.3.3 Nontrivial shapes

See also:

The complete source code of this example can be found in `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)[1]. 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)

    syst = 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:

```
    syst[lat.shape(ring, (0, r1 + 1))] = 4 * t
    syst[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 hopping_phase(site1, site2, phi):
        return -t * exp(1j * phi)

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

(continues on next page)

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

(continued from previous page)

```
        # 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(syst):
        for hop in kwant.builder.HoppingKind((1, 0), lat, lat)(syst):
            if crosses_branchcut(hop):
                yield hop
    syst[hops_across_cut] = hopping_phase
```

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:

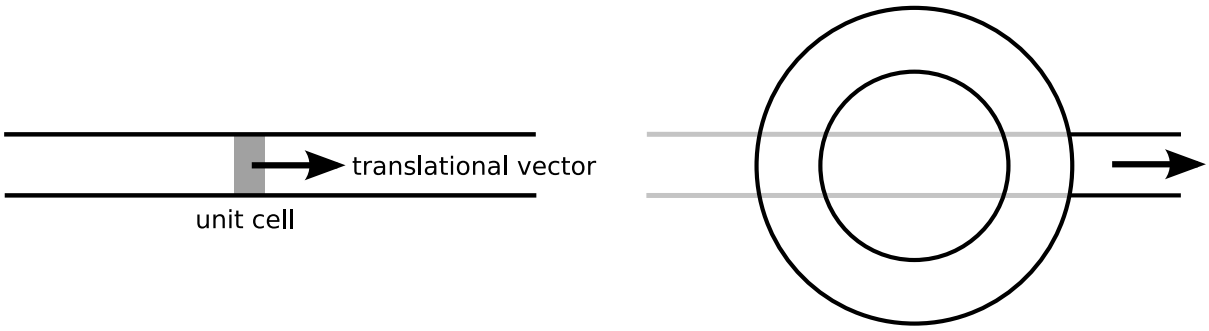
```
sym_lead = kwant.TranslationalSymmetry((-a, 0))
lead = kwant.Builder(sym_lead)

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`)[2].

Attaching the leads is done as before:

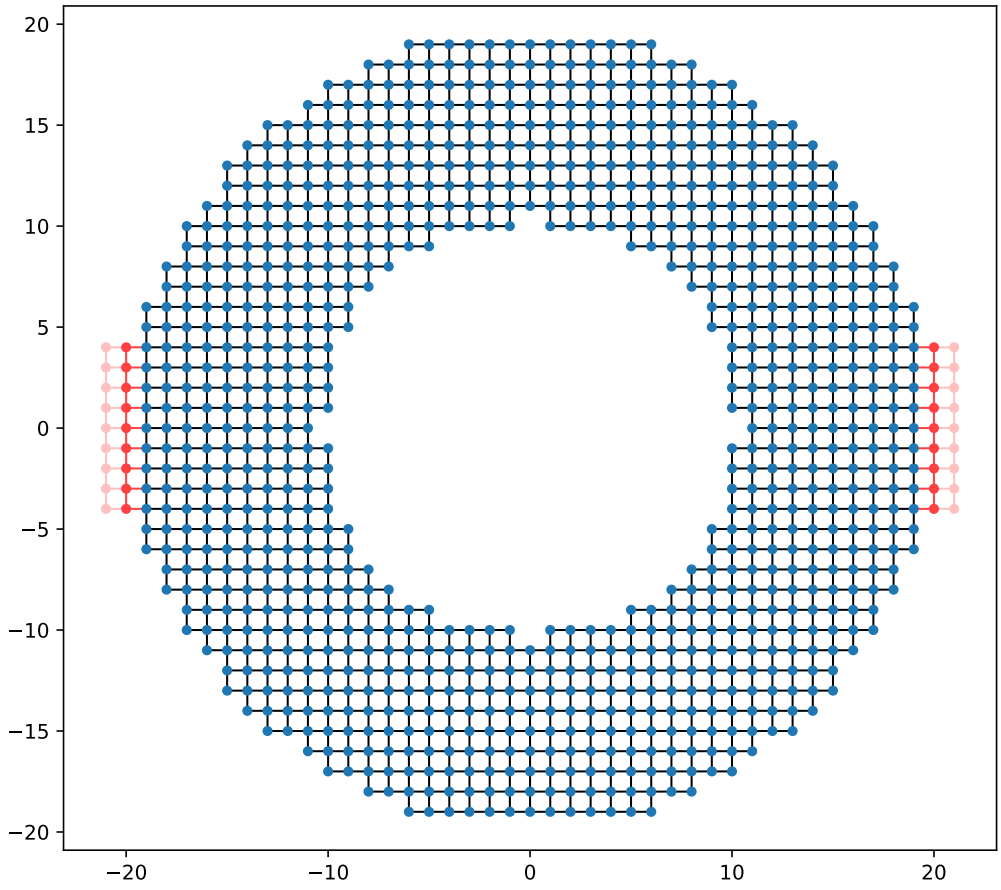
```
syst.attach_lead(lead)
syst.attach_lead(lead.reversed())
```

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:

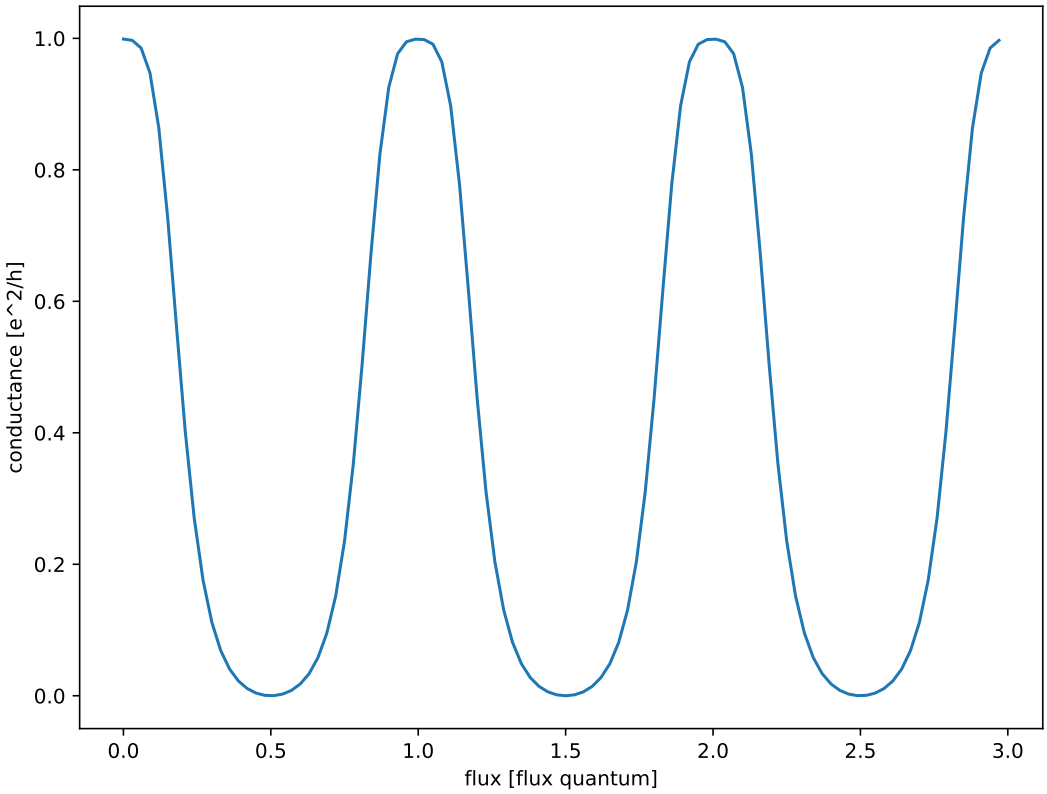


After the lead has been attached, the system should look like this:

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

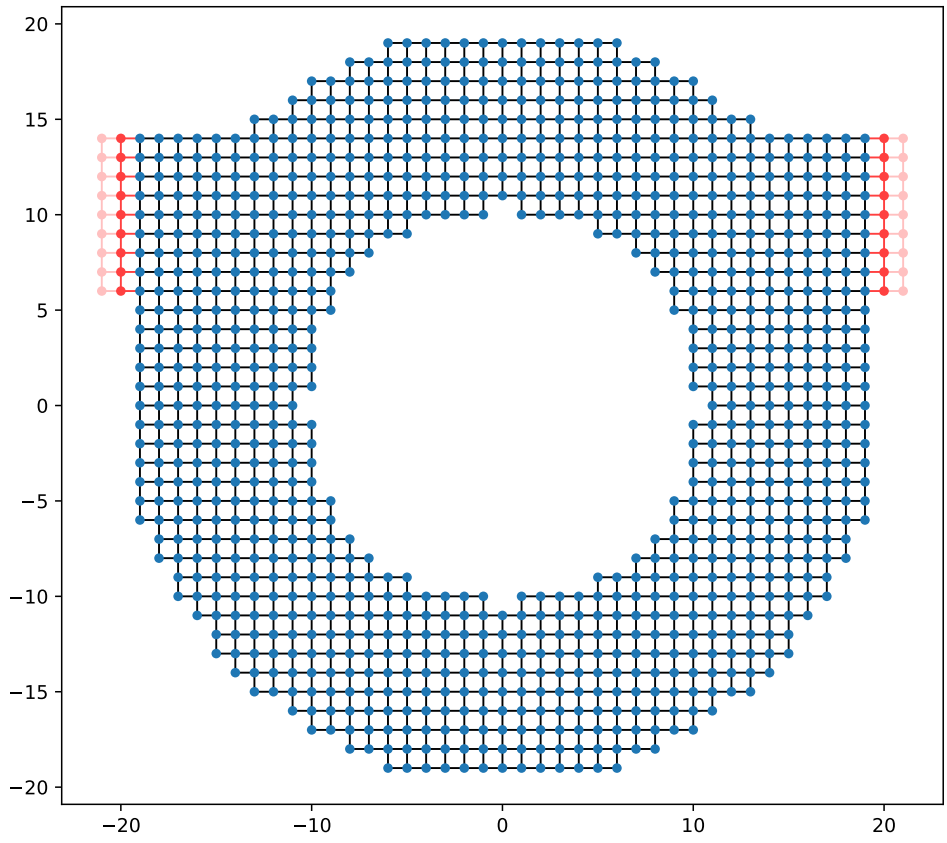


The computation of the conductance goes in the same fashion as before. Finally you should get the following result:



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:

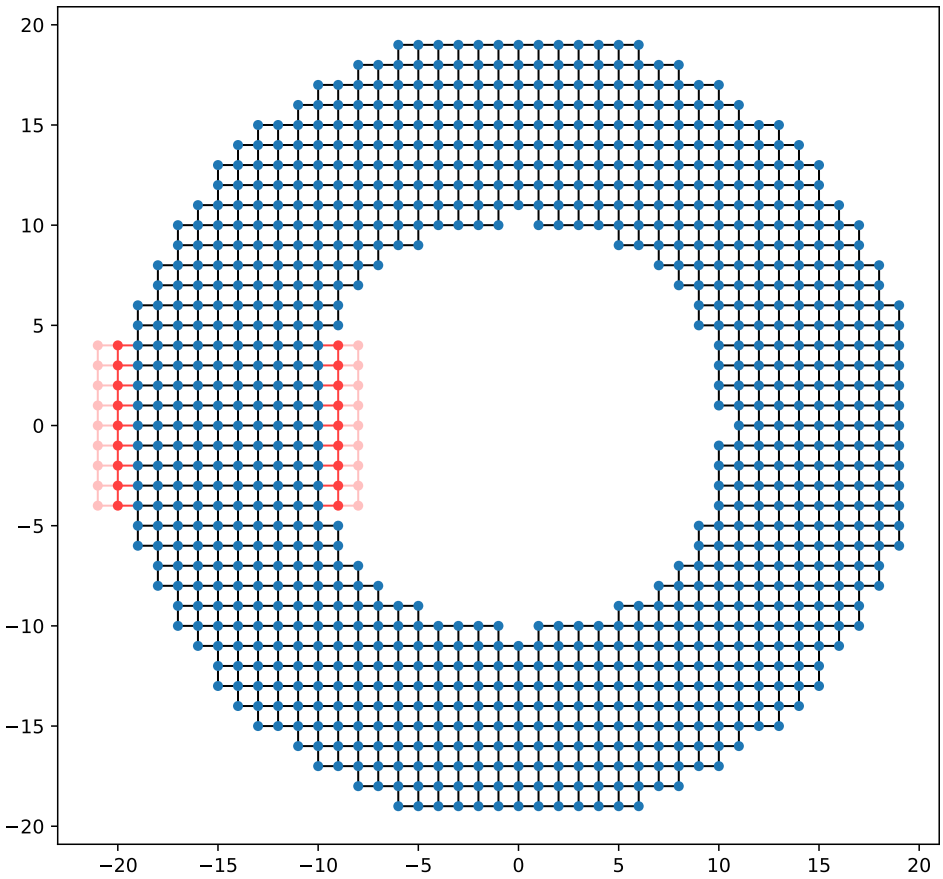


- Per default, *attach_lead* attaches the lead to the "outside" of the structure, by tracing the lead backwards, coming from infinity.

  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:

  ```
  syst.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 `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)
```

(continues on next page)

(continued from previous page)

```
    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 range(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
```

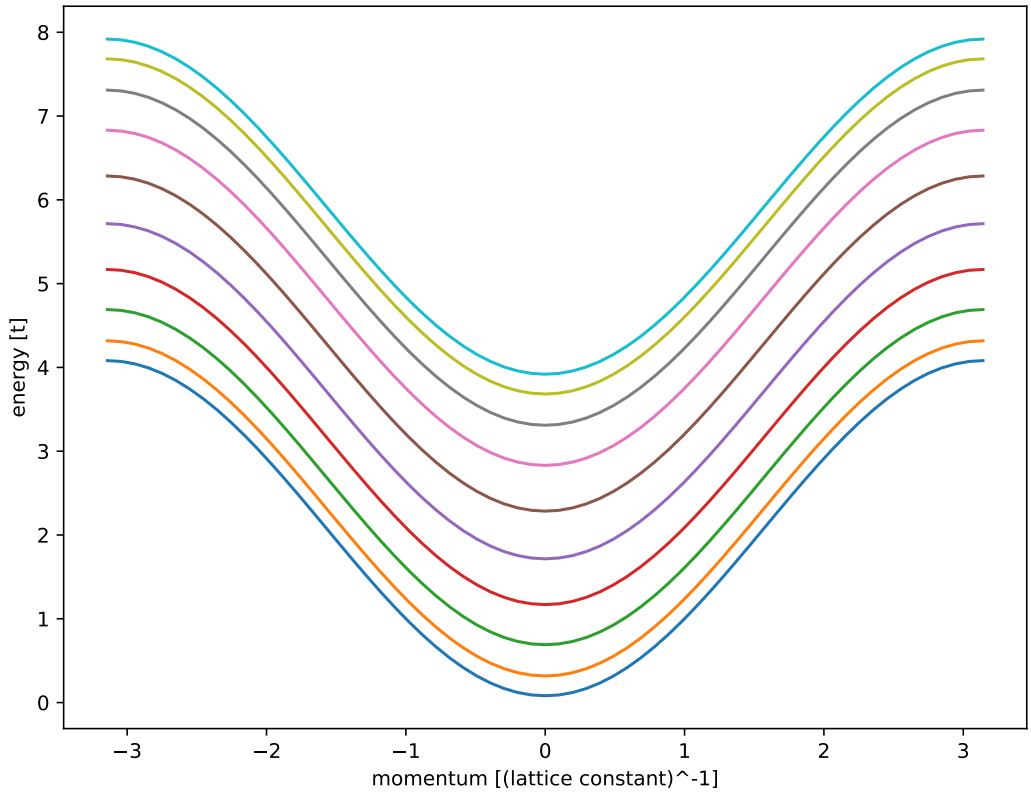
"Usual way" means defining a translational symmetry vector, as well as one unit cell of the lead, and the hoppings to neighboring unit cells. This information is enough to make the infinite, translationally invariant system needed for band structure calculations.

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 `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, norbs=1)

    syst = kwant.Builder()

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

(continues on next page)

(continued from previous page)

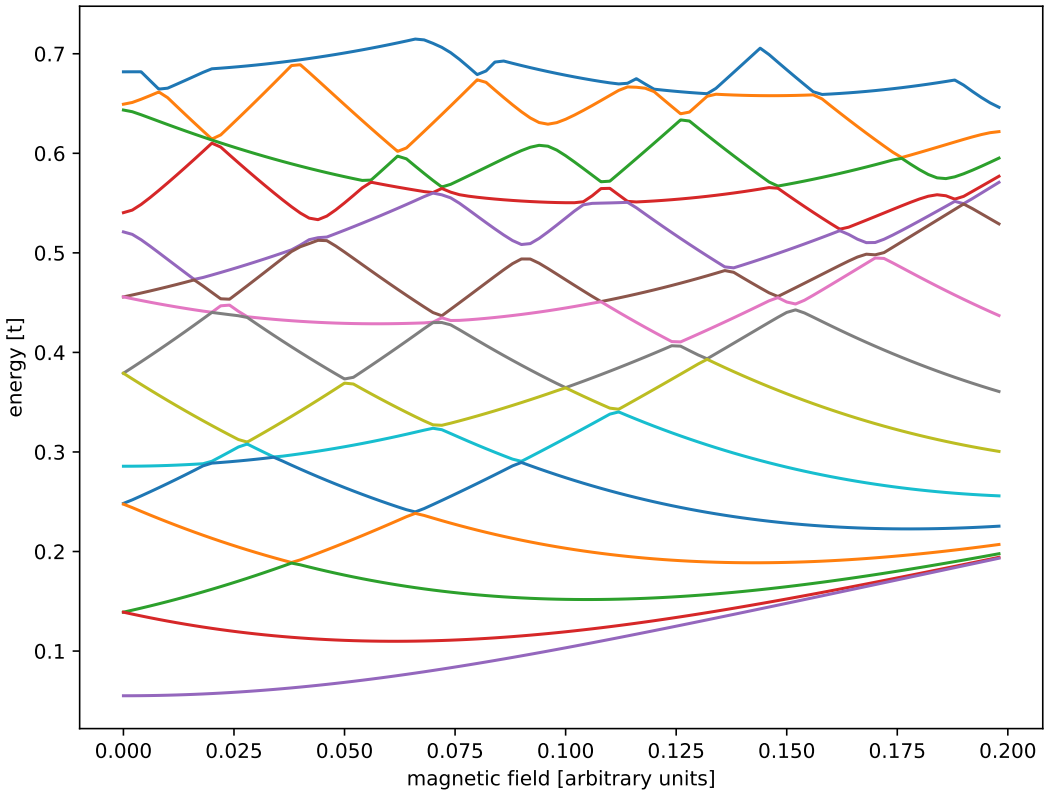
```
    def hopx(site1, site2, B):
        # The magnetic field is controlled by the parameter B
        y = site1.pos[1]
        return -t * exp(-1j * B * y)

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

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

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(syst, Bfields):

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

        # we only calculate the 15 lowest eigenvalues
        ev = sla.eigsh(ham_mat.tocsc(), k=15, sigma=0,
                       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:

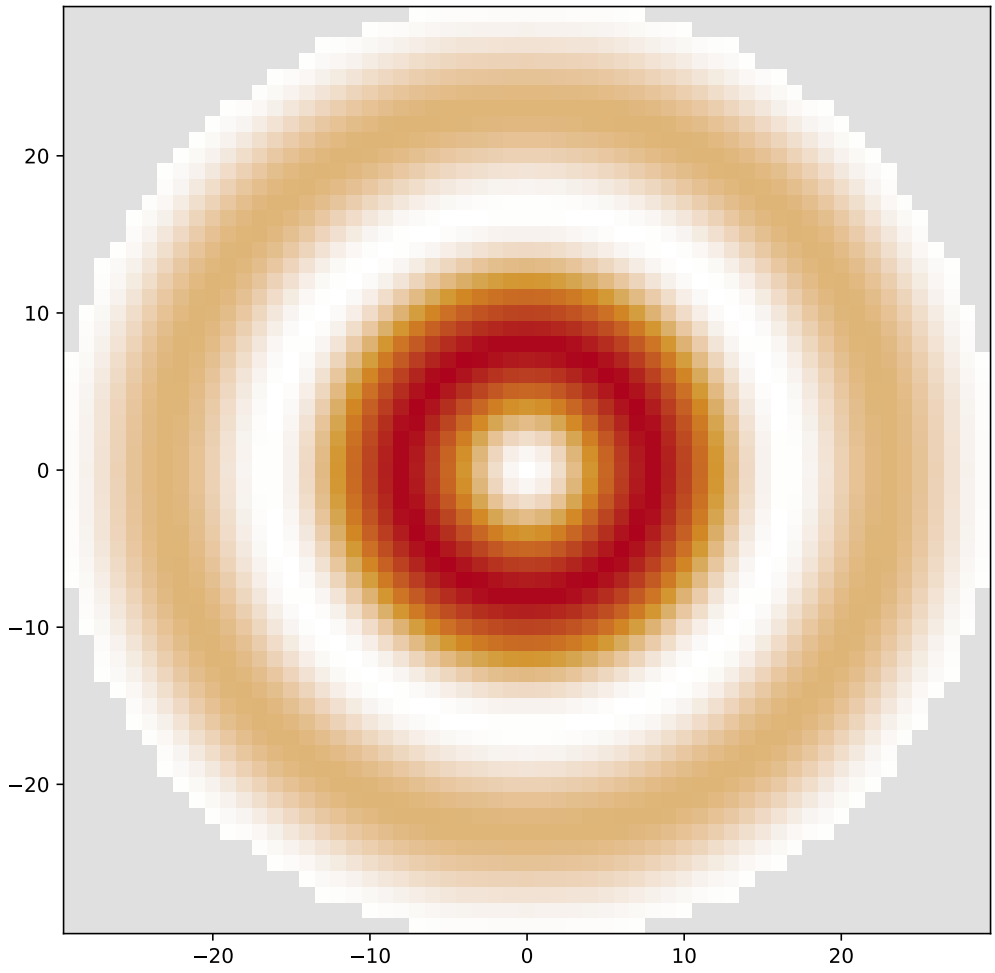


At zero magnetic field several energy levels are degenerate (since our quantum dot is rather symmetric). These degeneracies are split by the magnetic field, and the eigenenergies flow towards the Landau level energies at higher magnetic fields[1].

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

```
def plot_wave_function(syst, B=0.001):
    # Calculate the wave functions in the system.
    ham_mat = syst.hamiltonian_submatrix(sparse=True, params=dict(B=B))
    evals, evecs = sorted_eigs(sla.eigsh(ham_mat.tocsc(), k=20, sigma=0))

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

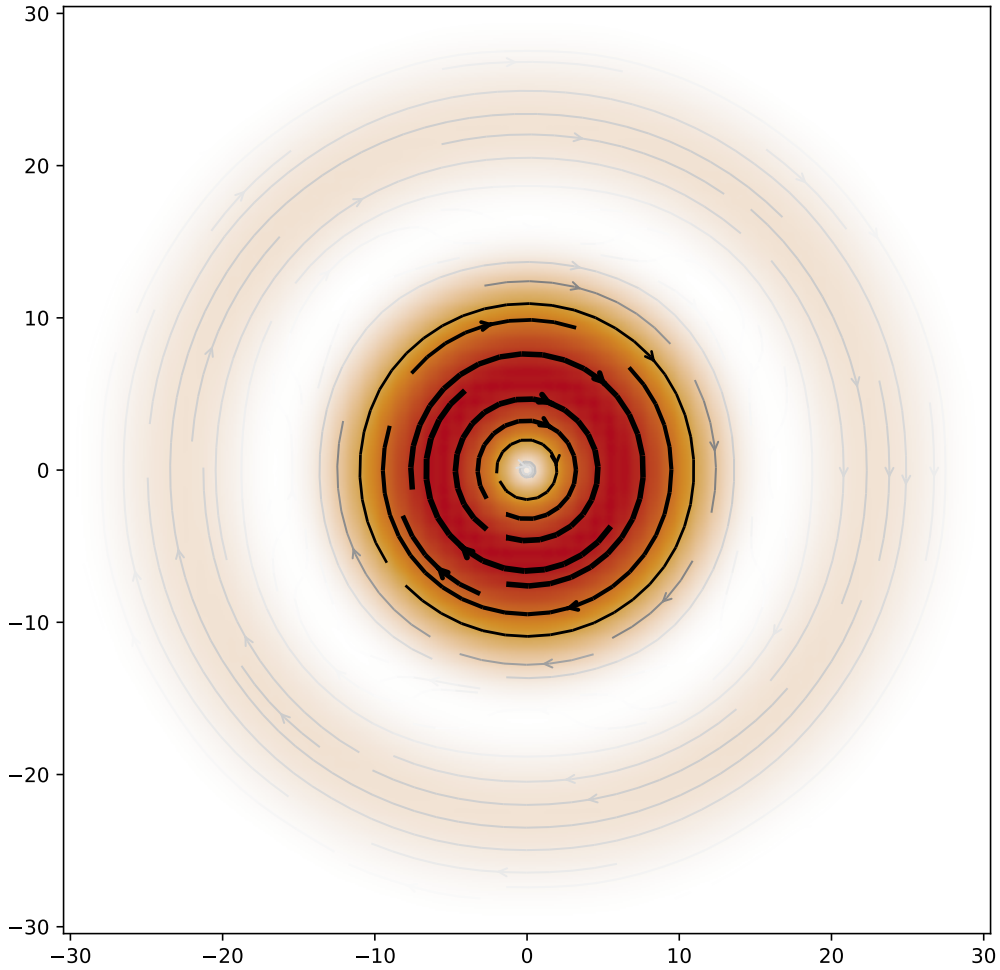
[1] Again, in this tutorial example no care was taken into choosing appropriate material parameters or units. For this reason, magnetic field is given only in "arbitrary units".



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.

As our model breaks time reversal symmetry (because of the applied magnetic field) we can also see an interesting property of the eigenstates, namely that they can have *non-zero* local current. We can calculate the local current due to a state by using *kwant.operator.Current* and plotting it using *kwant.plotter.current*:

```
def plot_current(syst, B=0.001):
    # Calculate the wave functions in the system.
    ham_mat = syst.hamiltonian_submatrix(sparse=True, params=dict(B=B))
    evals, evecs = sorted_eigs(sla.eigsh(ham_mat.tocsc(), k=20, sigma=0))

    # Calculate and plot the local current of the 10th eigenmode.
    J = kwant.operator.Current(syst)
    current = J(evecs[:, 9], params=dict(B=B))
    kwant.plotter.current(syst, current, colorbar=False)
```



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 `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

    syst = 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)

    syst[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:

```
    syst[[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 syst[a(0, 0)]
    syst[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 syst, [lead0, lead1]
```

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

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

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

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:

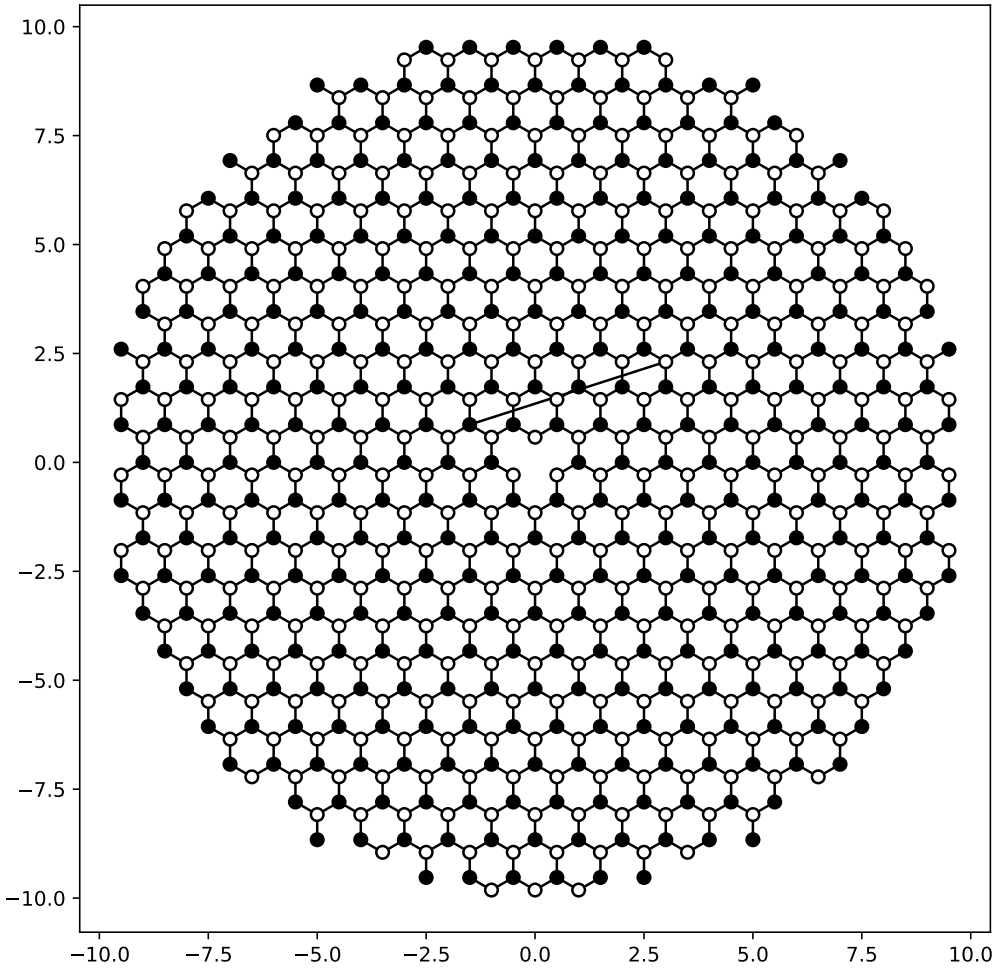
```
def main():
    pot = 0.1
    syst, leads = make_system(pot=pot)

    # 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(syst, 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:

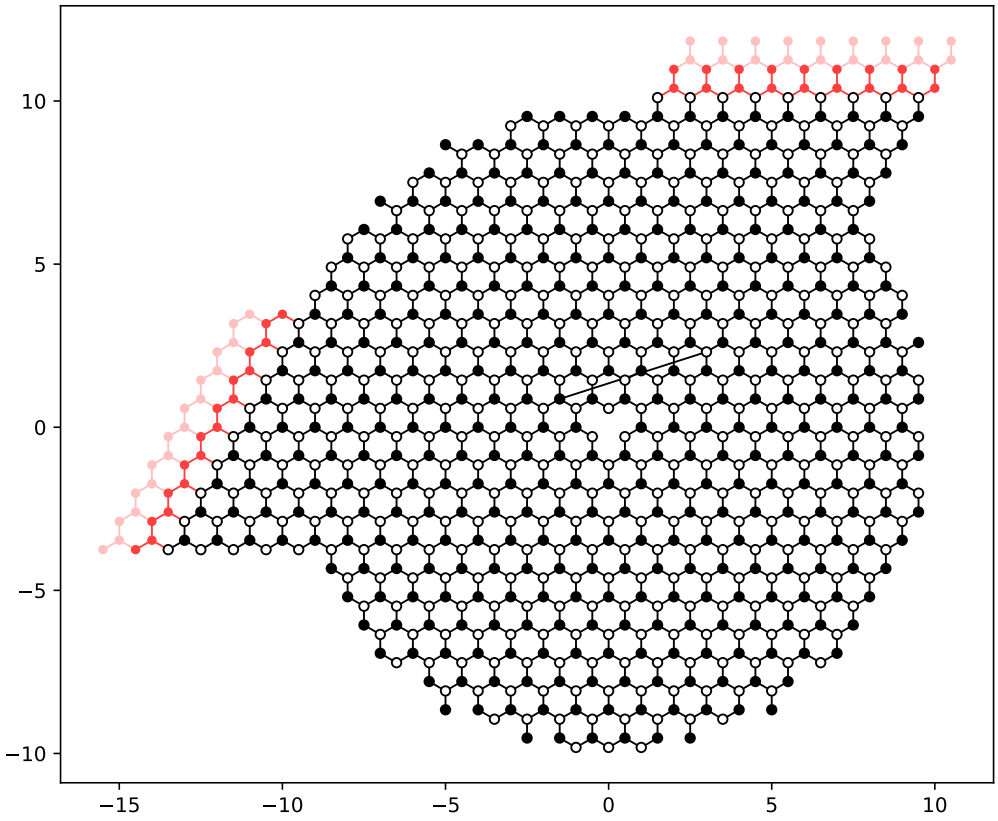


Computing the eigenvalues of largest magnitude,

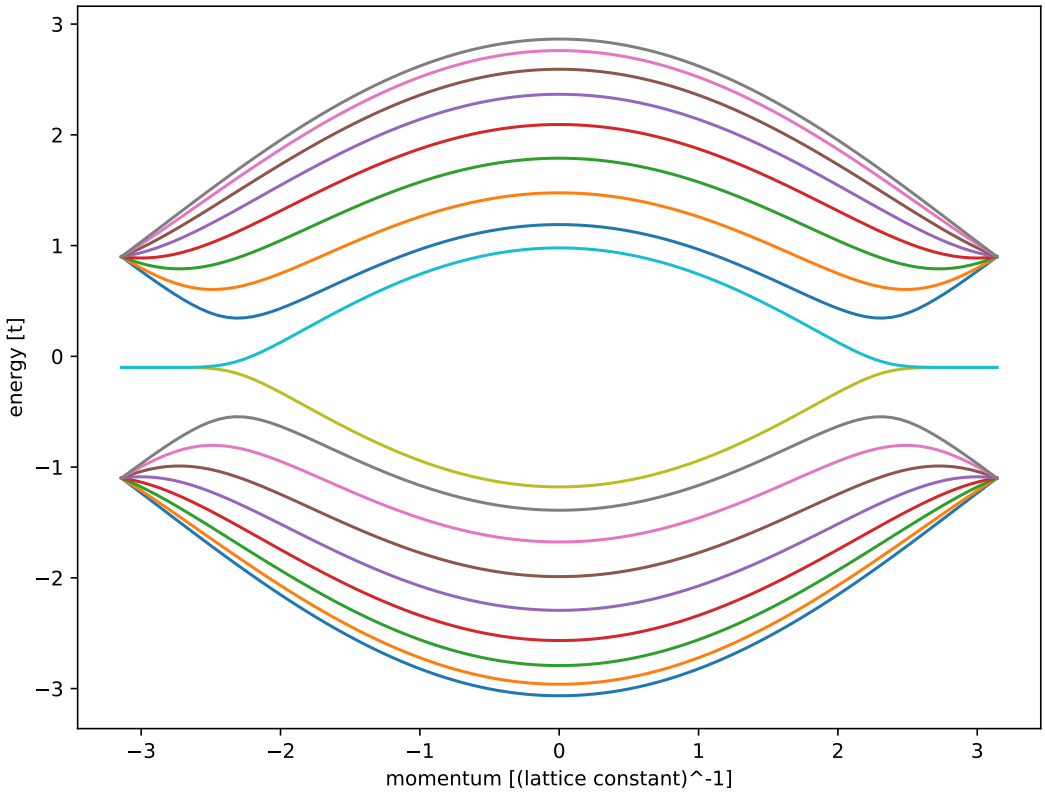
```
    compute_evs(syst.finalized())
```

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:

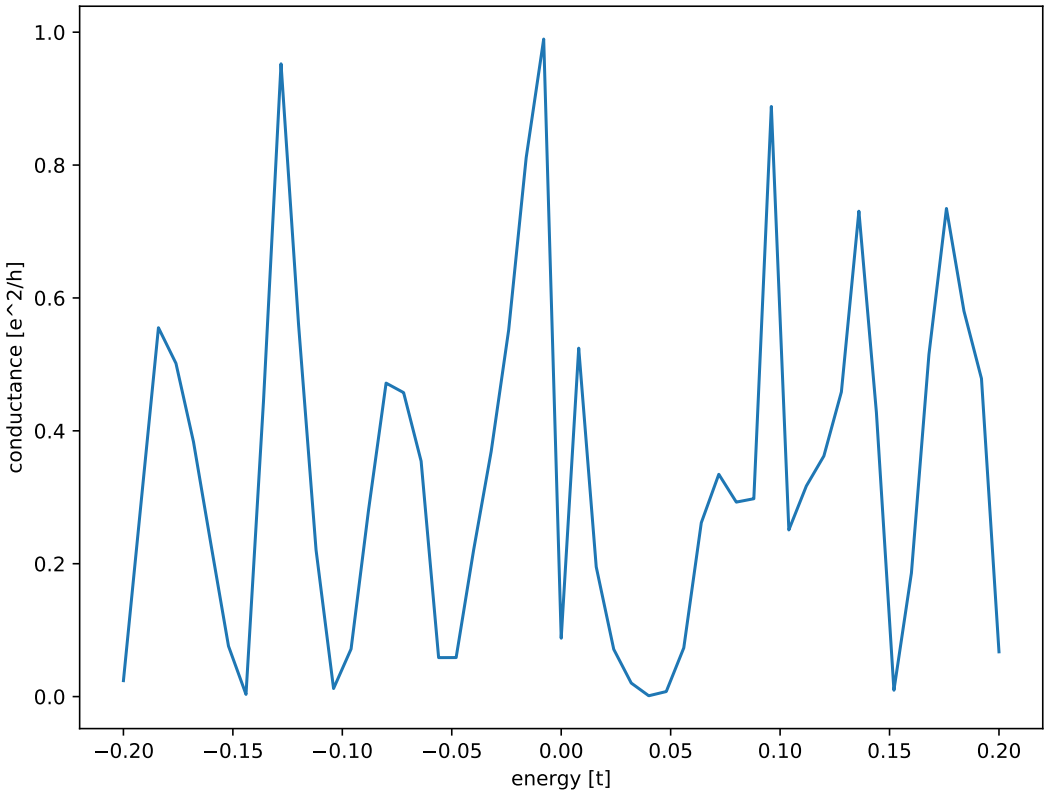


It computes the band structure of one of lead 0:



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:

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

  or just for a single sublattice:

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

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

```
del syst[graphene.shape(...)]
del syst[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 degrees of freedom, conservation laws and symmetries

See also:

The complete source code of this example can be found in `superconductor.py`

This example deals with superconductivity on the level of the Bogoliubov-de Gennes (BdG) equation.

In this framework, the Hamiltonian is given as

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

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 will need to include in our model with Kwant.

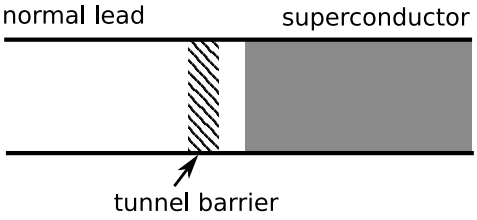
For this we restrict ourselves to a simple spinless system without magnetic field, so that Δ is just a number (which we choose real), and $\mathcal{T} H_0 \mathcal{T}^{-1} = H_0^* = H_0$. Furthermore, note that the Hamiltonian has particle-hole symmetry $\mathcal{P}$, i. e. $\mathcal{P} H \mathcal{P}^{-1} = -H$.

Care must be taken when transport calculations are done with the BdG equation. Electrons and holes carry charge with opposite sign, such that it is necessary to separate the electron and hole degrees of freedom in the scattering matrix. In particular, the conductance of a N-S-junction is given as

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

where N is the number of electron 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. Fortunately, in Kwant it is straightforward to partition the scattering matrix in these two degrees of freedom.

Let us consider a system that consists of a normal lead on the left, a superconductor on the right, and a tunnel barrier in between:



We implement the BdG Hamiltonian in Kwant using a 2x2 matrix structure for all Hamiltonian matrix elements, as we did previously in the *spin example*. We declare the square lattice and construct the scattering region with the following:

```
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. On each site, there
    # are now electron and hole orbitals, so we must specify the
    # number of orbitals per site. The orbital structure is the same
    # as in the Hamiltonian.
    lat = kwant.lattice.square(norbs=2)
    syst = kwant.Builder()

    #### Define the scattering region. ####
    # The superconducting order parameter couples electron and hole orbitals
    # on each site, and hence enters as an onsite potential.
    # The pairing is only included beyond the point 'Deltapos' in the scattering region.
    syst[(lat(x, y) for x in range(Deltapos) for y in range(W))] = (4 * t - mu) * tau_z
    syst[(lat(x, y) for x in range(Deltapos, L) for y in range(W))] = (4 * t - mu) * tau_z +␣
↪Delta * tau_x

    # The tunnel barrier
    syst[(lat(x, y) for x in range(barrierpos[0], barrierpos[1])
         for y in range(W))] = (4 * t + barrier - mu) * tau_z

    # Hoppings
    syst[lat.neighbors()] = -t * tau_z
```

Note the new argument `norbs` to *square*. This is the number of orbitals per site in the discretized BdG Hamiltonian - of course, `norbs = 2`, since each site has one electron orbital and one hole orbital. It is necessary to specify `norbs` here, such that we may later separate the scattering matrix into electrons and holes. Aside from this, creating the system is syntactically equivalent to *spin example*. The only difference is that the Pauli matrices now act in electron-hole space. Note that the tunnel barrier is added by overwriting previously set on-site matrix elements.

The superconducting order parameter is nonzero only in a part of the scattering region - the part to the right of the tunnel barrier. Thus, the scattering region is split into a superconducting part (the right side of it), and a normal part where the pairing is zero (the left side of it). The next step towards computing conductance is to attach leads. Let's attach two leads: a normal one to the left end, and a superconducting one to the right end. Starting with the left lead, we have:

```
    #### Define the leads. ####
    # Left lead - normal, so the order parameter is zero.
    sym_left = kwant.TranslationalSymmetry((-a, 0))
    # Specify the conservation law used to treat electrons and holes separately.
    # We only do this in the left lead, where the pairing is zero.
    lead0 = kwant.Builder(sym_left, conservation_law=-tau_z, particle_hole=tau_y)
    lead0[(lat(0, j) for j in range(W))] = (4 * t - mu) * tau_z
    lead0[lat.neighbors()] = -t * tau_z
```

Note the two new new arguments in *Builder*, `conservation_law` and `particle_hole`. For the purpose of computing conductance, `conservation_law` is the essential one, as it allows us to separate the electron and hole degrees of freedom. Note that it is not necessary to specify `particle_hole` in *Builder* to correctly compute the conductance in this example. We will discuss the argument `particle_hole` later on. First, let us discuss `conservation_law` in more detail.

Observe that electrons and holes are uncoupled in the left (normal) lead, since the superconducting order parameter that couples them is zero. Consequently, we may view the electron and hole degrees of freedom as being conserved, and may therefore separate them in the Hamiltonian.

In more technical terms, the conservation law implies that the Hamiltonian can be block diagonalized into uncoupled electron and hole blocks. Since the blocks are uncoupled, we can construct scattering states in each block independently. Of course, any scattering state from the electron (hole) block is entirely electron (hole) like. As a result, the scattering matrix separates into blocks that describe the scattering between different types of carriers, such as electron to electron, hole to electron, et cetera.

As we saw above, conservation laws in Kwant are specified with the `conservation_law` argument in *Builder*. Specifically, `conservation_law` is a matrix that acts on a single *site* and it must in addition have integer eigenvalues. Of course, it must also commute with the onsite Hamiltonian and hoppings to adjacent sites. Internally, Kwant then uses the eigenvectors of the conservation law to block diagonalize the Hamiltonian. Here, we've specified the conservation law $-\sigma_z$, such that the eigenvectors with eigenvalues -1 and 1 pick out the electron and hole blocks, respectively. Internally in Kwant, the blocks are stored in the order of ascending eigenvalues of the conservation law.

In order to move on with the conductance calculation, let's attach the second lead to the right side of the scattering region:

```
    # Right lead - superconducting, so the order parameter is included.
    sym_right = kwant.TranslationalSymmetry((a, 0))
    lead1 = kwant.Builder(sym_right)
    lead1[(lat(0, j) for j in range(W))] = (4 * t - mu) * tau_z + Delta * tau_x
    lead1[lat.neighbors()] = -t * tau_z

    #### Attach the leads and return the system. ####
    syst.attach_lead(lead0)
    syst.attach_lead(lead1)

    return syst
```

The second (right) lead is superconducting, such that the electron and hole blocks are coupled. Of course, this means that we can not separate them into uncoupled blocks as we did before, and therefore

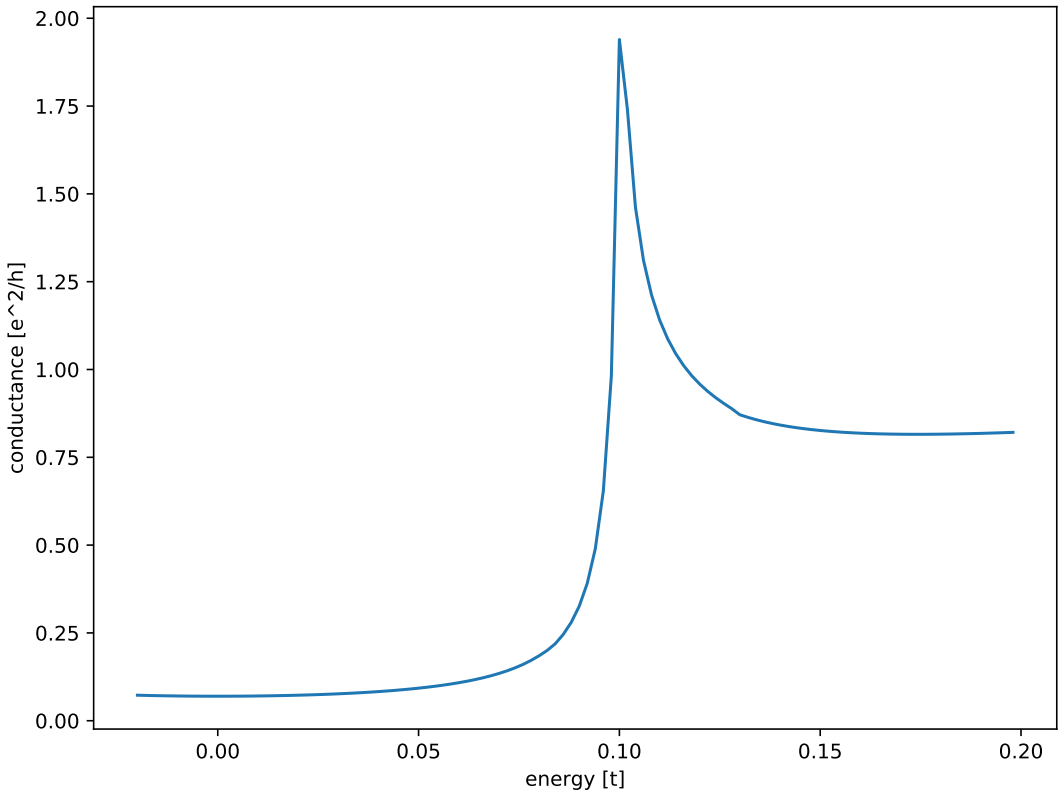
no conservation law is specified.

Kwant is now aware of the block structure of the Hamiltonian in the left lead. This means that we can extract transmission and reflection amplitudes not only into the left lead, but also between different conservation law blocks in the left lead. Generally if leads i and j both have a conservation law specified, `smatrix.transmission((i, a), (j, b))` gives us the scattering probability of carriers from block b of lead j, to block a of lead i. In our example, reflection from electrons to electrons in the left lead is thus `smatrix.transmission((0, 0), (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 is `smatrix.transmission((0, 1), (0, 0))`:

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

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

For the default parameters, we obtain the following conductance:



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.

Remember that when we defined *Builder* for the left lead above, we not only declared an electron-hole conservation law, but also that the Hamiltonian has the particle-hole symmetry $\mathcal{P} = \sigma_y$ which

anticommutes with the Hamiltonian, using the argument `particle_hole`. In Kwant, whenever one or more of the fundamental discrete symmetries (time-reversal, particle-hole and chiral) are present in a lead Hamiltonian, they can be declared in *Builder*. Kwant then automatically uses them to construct scattering states that obey the specified symmetries. In this example, we have a discrete symmetry declared in addition to a conservation law. For any two conservation law blocks that are transformed to each other by the discrete symmetry, Kwant then automatically computes the scattering states of one block by applying the symmetry operator to the scattering states of the other.

Now, $\mathcal{P}$ relates electrons and holes at *opposite* energies. However, a scattering problem is always solved at a fixed energy, so generally $\mathcal{P}$ does not give a relation between the electron and hole blocks. The exception is of course at zero energy, in which case particle-hole symmetry transforms between the electron and hole blocks, resulting in a symmetric scattering matrix. We can check the symmetry explicitly with

```
def check_PHS(syst):
    # Scattering matrix
    s = kwant.smatrix(syst, energy=0)
    # Electron to electron block
    s_ee = s.submatrix((0,0), (0,0))
    # Hole to hole block
    s_hh = s.submatrix((0,1), (0,1))
    print('s_ee: \n', np.round(s_ee, 3))
    print('s_hh: \n', np.round(s_hh[::-1, ::-1], 3))
    print('s_ee - s_hh^*: \n',
          np.round(s_ee - s_hh[::-1, ::-1].conj(), 3), '\n')
    # Electron to hole block
    s_he = s.submatrix((0,1), (0,0))
    # Hole to electron block
    s_eh = s.submatrix((0,0), (0,1))
    print('s_he: \n', np.round(s_he, 3))
    print('s_eh: \n', np.round(s_eh[::-1, ::-1], 3))
    print('s_he + s_eh^*: \n',
          np.round(s_he + s_eh[::-1, ::-1].conj(), 3))
```

which yields the output

```
s_ee:
 [[ 0.538+0.823j  0.   +0.j   ]
 [-0.   -0.j     0.515-0.856j]]
s_hh:
 [[ 0.538-0.823j  0.   -0.j   ]
 [-0.   +0.j     0.515+0.856j]]
s_ee - s_hh^*:
 [[ 0.-0.j  0.+0.j]
 [-0.-0.j  0.+0.j]]

s_he:
 [[-0.   +0.j  0.054+0.j]
 [ 0.179-0.j  0.   -0.j]]
s_eh:
 [[ 0.   -0.j -0.054-0.j]
 [-0.179+0.j -0.   -0.j]]
s_he + s_eh^*:
 [[ 0.+0.j -0.+0.j]
 [-0.-0.j  0.+0.j]]
```

Note that $\mathcal{P}$ flips the sign of momentum, and for the parameters we consider here, there are two electron and two hole modes active at zero energy. We thus reorder the matrix elements of the scattering matrix blocks above, to ensure that the same matrix elements in the electron and hole blocks relate scattering states and their particle hole partners. **Technical details**

- If you are only interested in particle (thermal) currents you do not need to separate the electron and hole degrees of freedom. Still, separating them using a conservation law makes the lead calculation in the solving phase more efficient.

2.7 Computing local quantities: densities and currents

In the previous tutorials we have mainly concentrated on calculating *global* properties such as conductance and band structures. Often, however, insight can be gained from calculating *locally-defined* quantities, that is, quantities defined over individual sites or hoppings in your system. In the *Closed systems* tutorial we saw how we could visualize the density associated with the eigenstates of a system using *kwant.plotter.map*.

In this tutorial we will see how we can calculate more general quantities than simple densities by studying spin transport in a system with a magnetic texture.

See also:

The complete source code of this example can be found in `magnetic_texture.py`

2.7.1 Introduction

Our starting point will be the following spinful tight-binding model on a square lattice:

$$H = -\sum_{\langle ij\rangle}\sum_{\alpha}|i\alpha\rangle\langle j\alpha| + J\sum_{i}\sum_{\alpha\beta}\mathbf{m}_i\cdot{}_{\alpha\beta}|i\alpha\rangle\langle i\beta|,$$

where latin indices run over sites, and greek indices run over spin. We can identify the first term as a nearest-neighbor hopping between like-spins, and the second as a term that couples spins on the same site. The second term acts like a magnetic field of strength J that varies from site to site and that, on site i, points in the direction of the unit vector $\mathbf{m}_i$. ${}_{\alpha\beta}$ is a vector of Pauli matrices. We shall take the following form for $\mathbf{m}_i$:

$$\begin{aligned}\mathbf{m}_i &= \left(\frac{x_i}{x_i^2+y_i^2}\sin\theta_i,\ \frac{y_i}{x_i^2+y_i^2}\sin\theta_i,\ \cos\theta_i\right)^T,\\ \theta_i &= \frac{\pi}{2}(\tanh\frac{r_i-r_0}{\delta}-1),\end{aligned}$$

where x_i and y_i are the x and y coordinates of site i, and $r_i = \sqrt{x_i^2+y_i^2}$.

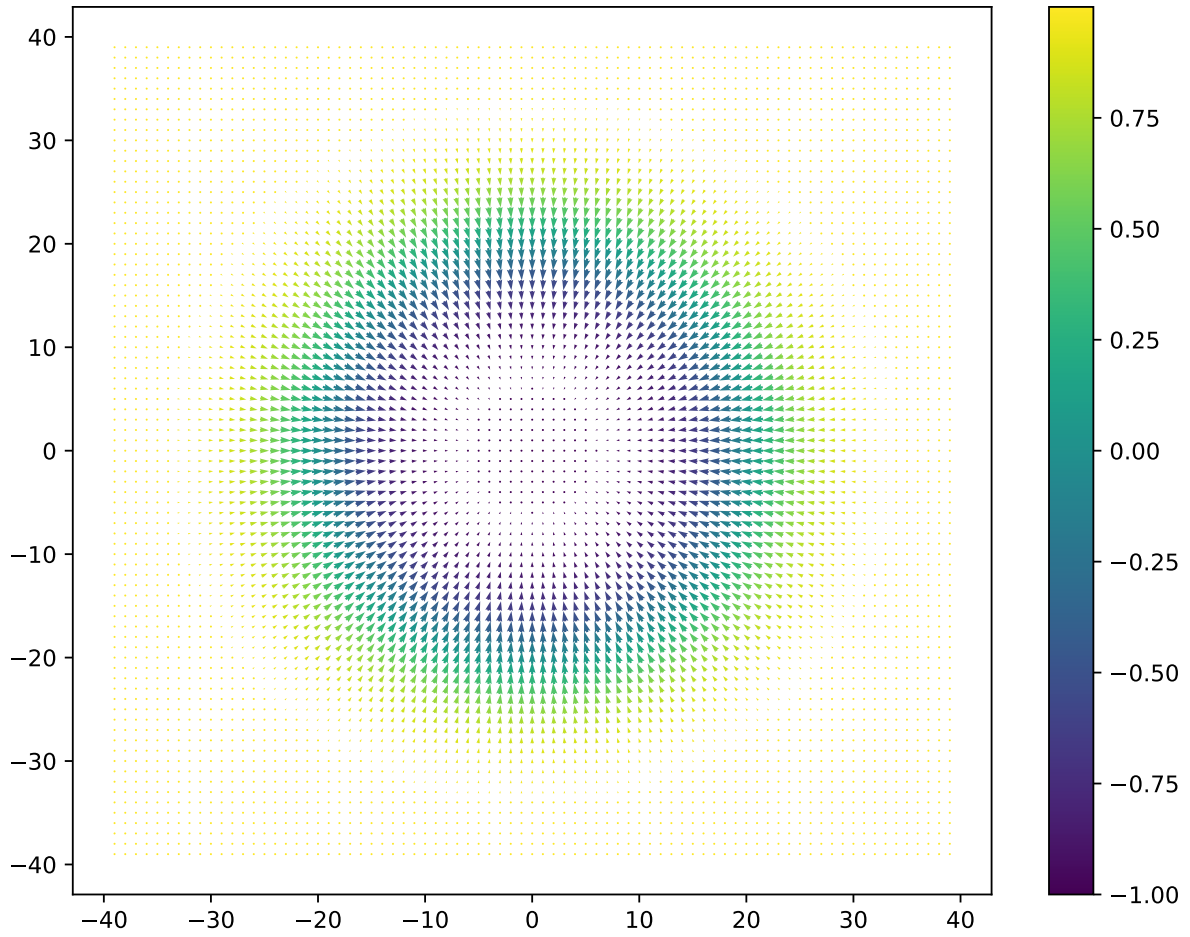
To define this model in Kwant we start as usual by defining functions that depend on the model parameters:

```
def field_direction(pos, r0, delta):
    x, y = pos
    r = np.linalg.norm(pos)
    r_tilde = (r - r0) / delta
    theta = (tanh(r_tilde) - 1) * (pi / 2)

    if r == 0:
        m_i = [0, 0, -1]
    else:
        m_i = [
            (x / r) * sin(theta),
            (y / r) * sin(theta),
            cos(theta),
        ]

    return np.array(m_i)


def scattering_onsite(site, r0, delta, J):
    m_i = field_direction(site.pos, r0, delta)
    return J * np.dot(m_i, sigma)


def lead_onsite(site, J):
    return J * sigma_z
```

and define our system as a square shape on a square lattice with two orbitals per site, with leads attached on the left and right:

```
lat = kwant.lattice.square(norbs=2)

def make_system(L=80):

    syst = kwant.Builder()

    def square(pos):
        return all(-L/2 < p < L/2 for p in pos)

    syst[lat.shape(square, (0, 0))] = scattering_onsite
    syst[lat.neighbors()] = -sigma_0

    lead = kwant.Builder(kwant.TranslationalSymmetry((-1, 0)),
                         conservation_law=-sigma_z)

    lead[lat.shape(square, (0, 0))] = lead_onsite
    lead[lat.neighbors()] = -sigma_0

    syst.attach_lead(lead)
    syst.attach_lead(lead.reversed())

    return syst
```

Below is a plot of a projection of $\mathbf{m}_i$ onto the x-y plane inside the scattering region. The z component is shown by the color scale:



We will now be interested in analyzing the form of the scattering states that originate from the left lead:

```
    params = dict(r0=20, delta=10, J=1)
    wf = kwant.wave_function(syst, energy=-1, params=params)
    psi = wf(0)[0]
```

2.7.2 Local densities

If we were simulating a spinless system with only a single degree of freedom, then calculating the density on each site would be as simple as calculating the absolute square of the wavefunction like:

```
density = np.abs(psi)**2
```

When there are multiple degrees of freedom per site, however, one has to be more careful. In the present case with two (spin) degrees of freedom per site one could calculate the per-site density like:

```
    # even (odd) indices correspond to spin up (down)
    up, down = psi[::2], psi[1::2]
    density = np.abs(up)**2 + np.abs(down)**2
```

With more than one degree of freedom per site we have more freedom as to what local quantities we can meaningfully compute. For example, we may wish to calculate the local z-projected spin density. We could calculate this in the following way:

```
    # spin down components have a minus sign
    spin_z = np.abs(up)**2 - np.abs(down)**2
```

If we wanted instead to calculate the local y-projected spin density, we would need to use an even more complicated expression:

```
    # spin down components have a minus sign
    spin_y = 1j * (down.conjugate() * up - up.conjugate() * down)
```

The *kwant.operator* module aims to alleviate somewhat this tedious book-keeping by providing a simple interface for defining operators that act on wavefunctions. To calculate the above quantities we would use the *Density* operator like so:

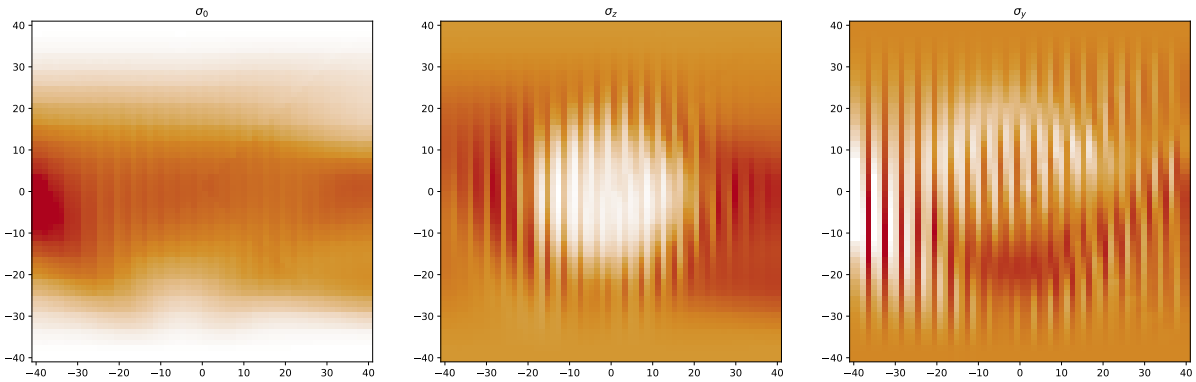
```
    rho = kwant.operator.Density(syst)
    rho_sz = kwant.operator.Density(syst, sigma_z)
    rho_sy = kwant.operator.Density(syst, sigma_y)

    # calculate the expectation values of the operators with 'psi'
    density = rho(psi)
    spin_z = rho_sz(psi)
    spin_y = rho_sy(psi)
```

Density takes a *System* as its first parameter as well as (optionally) a square matrix that defines the quantity that you wish to calculate per site. When an instance of a *Density* is then evaluated with a wavefunction, the quantity

$$\rho_i = {}_i^\dagger\mathbf{M}\,{}_i$$

is calculated for each site i, where ${}_i$ is a vector consisting of the wavefunction components on that site and $\mathbf{M}$ is the square matrix referred to previously.

Below we can see colorplots of the above-calculated quantities. The array that is returned by evaluating a *Density* can be used directly with *kwant.plotter.density*:



Technical Details

Although we refer loosely to "densities" and "operators" above, a *Density* actually represents a *collection* of linear operators. This can be made clear by rewriting the above definition of ρ_i in the following way:

$$\rho_i = \sum_{\alpha\beta} \psi^*_\alpha \mathcal{M}_{i\alpha\beta} \psi_\beta$$

where greek indices run over the degrees of freedom in the Hilbert space of the scattering region and latin indices run over sites. We can this identify $\mathcal{M}_{i\alpha\beta}$ as the components of a rank-3 tensor and can represent them as a "vector of matrices":

$$\mathcal{M} = \left[\begin{pmatrix} \mathbf{M} & 0 & \dots \\ 0 & 0 & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}, \begin{pmatrix} 0 & 0 & \dots \\ 0 & \mathbf{M} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}, \dots \right]$$

where $\mathbf{M}$ is defined as in the main text, and the 0 are zero matrices of the same shape as $\mathbf{M}$.

2.7.3 Local currents

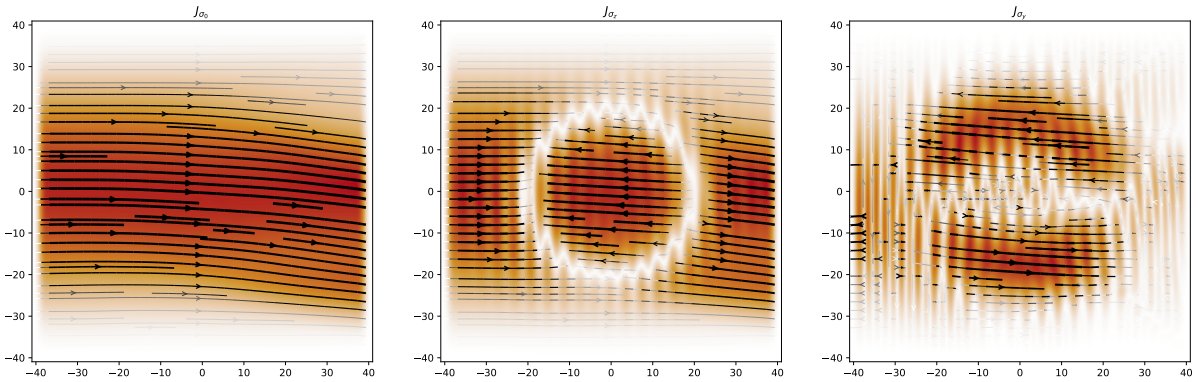
kwant.operator also has a class *Current* for calculating local currents, analogously to the local "densities" described above. If one has defined a density via a matrix $\mathbf{M}$ and the above equation, then one can define a local current flowing from site b to site a:

$$J_{ab} = i \left({}^{\dagger}_{b}(\mathbf{H}_{ab})^{\dagger} \mathbf{M}_{\ a} - {}^{\dagger}_{a}\mathbf{M}\mathbf{H}_{ab\ b} \right),$$

where $\mathbf{H}_{ab}$ is the hopping matrix from site b to site a. For example, to calculate the local current and spin current:

```
J_0 = kwant.operator.Current(syst)
J_z = kwant.operator.Current(syst, sigma_z)
J_y = kwant.operator.Current(syst, sigma_y)

# calculate the expectation values of the operators with 'psi'
current = J_0(psi)
spin_z_current = J_z(psi)
spin_y_current = J_y(psi)
```

Evaluating a *Current* operator on a wavefunction returns a 1D array of values that can be directly used with *kwant.plotter.current*:



Note: Evaluating a *Current* operator on a wavefunction returns a 1D array of the same length as the number of hoppings in the system, ordered in the same way as the edges in the system's graph.

Technical Details

Similarly to how we saw in the previous section that *Density* can be thought of as a collection of operators, *Current* can be defined in a similar way. Starting from the definition of a "density":

$$\rho_a = \sum_{\alpha\beta} \psi^*_\alpha \mathcal{M}_{a\alpha\beta} \psi_\beta,$$

we can define *currents* J_{ab} via the continuity equation:

$$\frac{\partial \rho_a}{\partial t} - \sum_b J_{ab} = 0$$

where the sum runs over sites b neigboring site a. Plugging in the definition for ρ_a, along with the Schrödinger equation and the assumption that $\mathcal{M}$ is time independent, gives:

$$J_{ab} = \sum_{\alpha\beta} \psi^*_\alpha \left(i \sum_\gamma \mathcal{H}^*_{ab\gamma\alpha} \mathcal{M}_{a\gamma\beta} - \mathcal{M}_{a\alpha\gamma} \mathcal{H}_{ab\gamma\beta} \right) \psi_\beta,$$

where latin indices run over sites and greek indices run over the Hilbert space degrees of freedom, and

$$\mathcal{H}_{ab} = \begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \ddots & 0 & \mathbf{H}_{ab} & \cdots \\ \cdots & 0 & \ddots & 0 & \cdots \\ \cdots & 0 & 0 & \ddots & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

i.e. $\mathcal{H}_{ab}$ is a matrix that is zero everywhere except on elements connecting *from* site b *to* site a, where it is equal to the hopping matrix $\mathbf{H}_{ab}$ between these two sites.

This allows us to identify the rank-4 quantity

$$\mathcal{J}_{ab\alpha\beta} = i \sum_\gamma \mathcal{H}^*_{ab\gamma\alpha} \mathcal{M}_{a\gamma\beta} - \mathcal{M}_{a\alpha\gamma} \mathcal{H}_{ab\gamma\beta}$$

as the local current between connected sites.

The diagonal part of this quantity, $\mathcal{J}_{aa}$, represents the extent to which the density defined by $\mathcal{M}_a$ is not conserved on site a. It can be calculated using *Source*, rather than *Current*, which only computes the off-diagonal part.

2.7.4 Spatially varying operators

The above examples are reasonably simple in the sense that the book-keeping required to manually calculate the various densities and currents is still manageable. Now we shall look at the case where we

wish to calculate some projected spin currents, but where the spin projection axis varies from place to place. More specifically, we want to visualize the spin current along the direction of $\mathbf{m}_i$, which changes continuously over the whole scattering region.

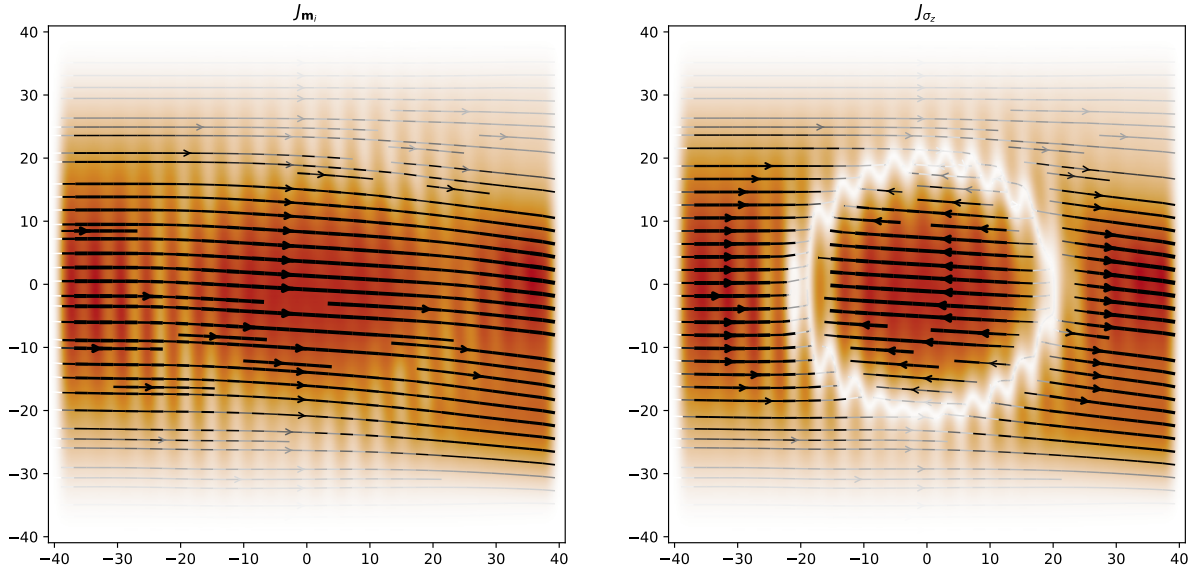
Doing this is as simple as passing a *function* when instantiating the *Current*, instead of a constant matrix:

```
def following_m_i(site, r0, delta):
    m_i = field_direction(site.pos, r0, delta)
    return np.dot(m_i, sigma)

J_m = kwant.operator.Current(syst, following_m_i)

# evaluate the operator
m_current = J_m(psi, params=dict(r0=25, delta=10))
```

The function must take a *Site* as its first parameter, and may optionally take other parameters (i.e. it must have the same signature as a Hamiltonian onsite function), and must return the square matrix that defines the operator we wish to calculate.

Note: In the above example we had to pass the extra parameters needed by the `following_operator` function via the `param` keyword argument. In general you must pass all the parameters needed by the Hamiltonian via `params` (as you would when calling *smatrix* or *wave_function*). In the previous examples, however, we used the fact that the system hoppings do not depend on any parameters (these are the only Hamiltonian elements required to calculate currents) to avoid passing the system parameters for the sake of brevity.

Using this we can see that the spin current is essentially oriented along the direction of m_i in the present regime where the onsite term in the Hamiltonian is dominant:



Note: Although this example used exclusively *Current*, you can do the same with *Density*.

2.7.5 Defining operators over parts of a system

Another useful feature of *kwant.operator* is the ability to calculate operators over selected parts of a system. For example, we may wish to calculate the total density of states in a certain part of the system, or the current flowing through a cut in the system. We can do this selection when creating the operator by using the keyword parameter `where`.

Density of states in a circle

To calculate the density of states inside a circle of radius 20 we can simply do:

```
    def circle(site):
        return np.linalg.norm(site.pos) < 20

    rho_circle = kwant.operator.Density(syst, where=circle, sum=True)

    all_states = np.vstack((wf(0), wf(1)))
    dos_in_circle = sum(rho_circle(p) for p in all_states) / (2 * pi)
    print('density of states in circle:', dos_in_circle)
```

```
density of states in circle: 859.7665263179026
```

note that we also provide `sum=True`, which means that evaluating the operator on a wavefunction will produce a single scalar. This is semantically equivalent to providing `sum=False` (the default) and running `numpy.sum` on the output.

Current flowing through a cut

Below we calculate the probability current and z-projected spin current near the interfaces with the left and right leads.

```
    def left_cut(site_to, site_from):
        return site_from.pos[0] <= -39 and site_to.pos[0] > -39

    def right_cut(site_to, site_from):
        return site_from.pos[0] < 39 and site_to.pos[0] >= 39

    J_left = kwant.operator.Current(syst, where=left_cut, sum=True)
    J_right = kwant.operator.Current(syst, where=right_cut, sum=True)

    Jz_left = kwant.operator.Current(syst, sigma_z, where=left_cut, sum=True)
    Jz_right = kwant.operator.Current(syst, sigma_z, where=right_cut, sum=True)

    print('J_left:', J_left(psi), ' J_right:', J_right(psi))
    print('Jz_left:', Jz_left(psi), ' Jz_right:', Jz_right(psi))
```

```
J_left: 0.9798539026381679  J_right: 0.9798539026346825
Jz_left: 0.9714656811973775  Jz_right: 0.9840499358260828
```

We see that the probability current is conserved across the scattering region, but the z-projected spin current is not due to the fact that the Hamiltonian does not commute with σ_z everywhere in the scattering region.

Note: `where` can also be provided as a sequence of *Site* or a sequence of hoppings (i.e. pairs of *Site*), rather than a function.

2.7.6 Advanced Topics

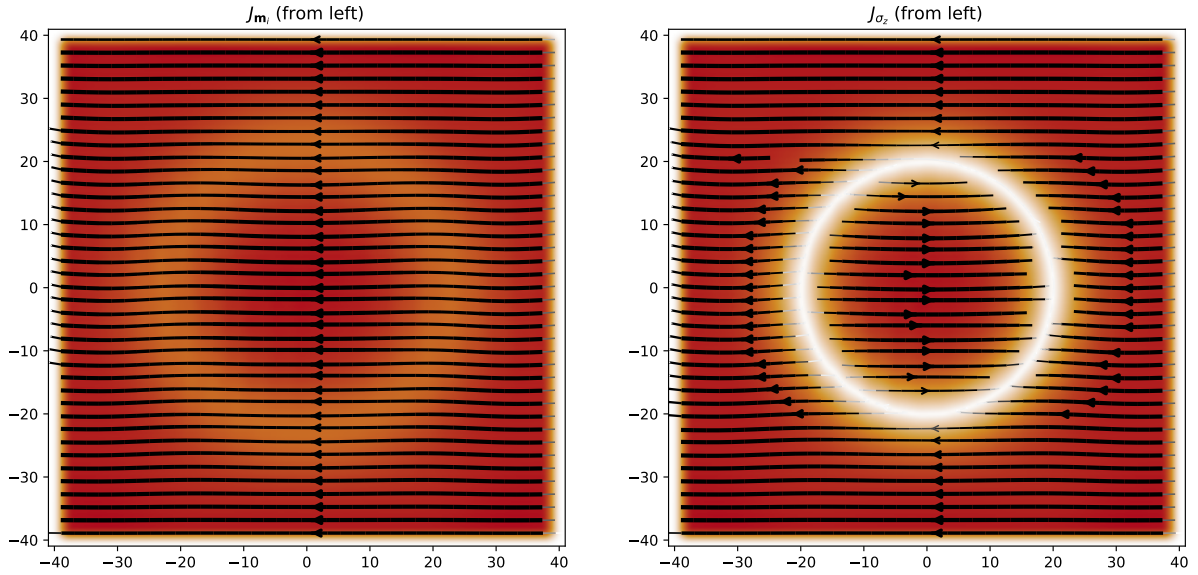
Using `bind` for speed

In most of the above examples we only used each operator *once* after creating it. Often one will want to evaluate an operator with many different wavefunctions, for example with all scattering wavefunctions at a certain energy, but with the *same set of parameters*. In such cases it is best to tell the operator to precompute the onsite matrices and any necessary Hamiltonian elements using the given set of parameters, so that this work is not duplicated every time the operator is evaluated.

This can be achieved with *bind*:

Warning: Take care that you do not use an operator that was bound to a particular set of parameters with wavefunctions calculated with a *different* set of parameters. This will almost certainly give incorrect results.

```
    J_m = kwant.operator.Current(syst, following_m_i)
    J_z = kwant.operator.Current(syst, sigma_z)

    J_m_bound = J_m.bind(params=dict(r0=25, delta=10, J=1))
    J_z_bound = J_z.bind(params=dict(r0=25, delta=10, J=1))

    # Sum current local from all scattering states on the left at energy=-1
    wf_left = wf(0)
    J_m_left = sum(J_m_bound(p) for p in wf_left)
    J_z_left = sum(J_z_bound(p) for p in wf_left)
```



2.8 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.

2.8.1 2D example: graphene quantum dot

See also:

The complete source code of this example can be found in `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):
```

(continues on next page)

(continued from previous page)

```
        x, y = pos
        return x**2 + y**2 < r**2

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

    return syst
```

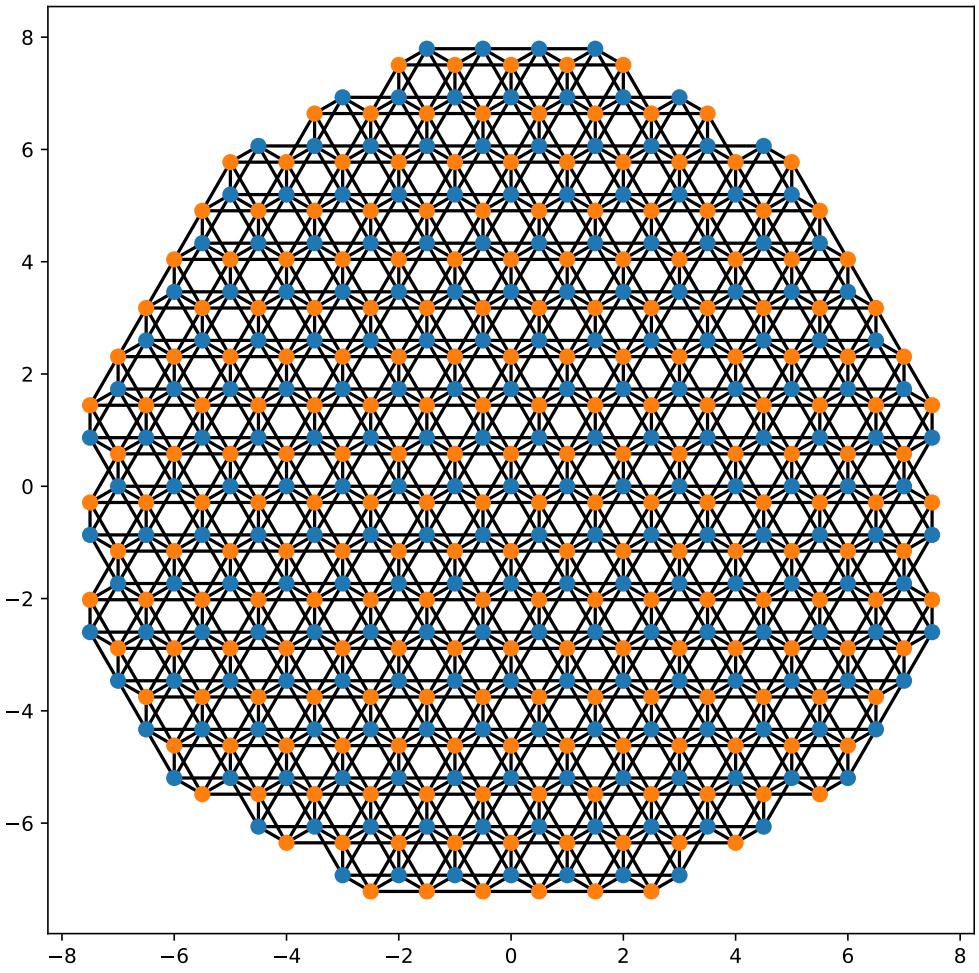
Note that adding hoppings hoppings to the n-th nearest neighbors can be simply done by passing n as an argument to *neighbors*. Also note that we use the method *eradicate_dangling* to get rid of single atoms sticking out of the shape. It is necessary to do so *before* adding the next-nearest-neighbor hopping[1].

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

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

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

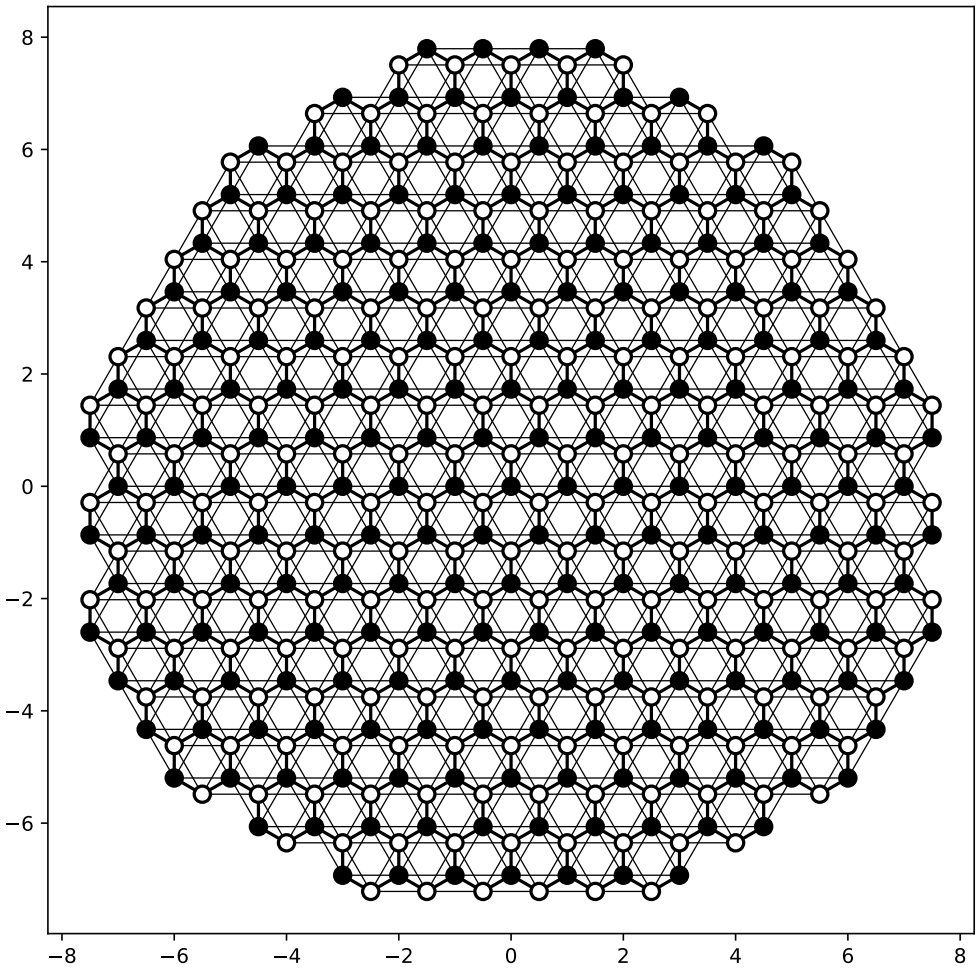
[1] A dangling site is defined as having only one hopping connecting it to the rest. With next-nearest-neighbor hopping also all sites that are dangling with only nearest-neighbor hopping have more than one hopping.



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(syst, 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.

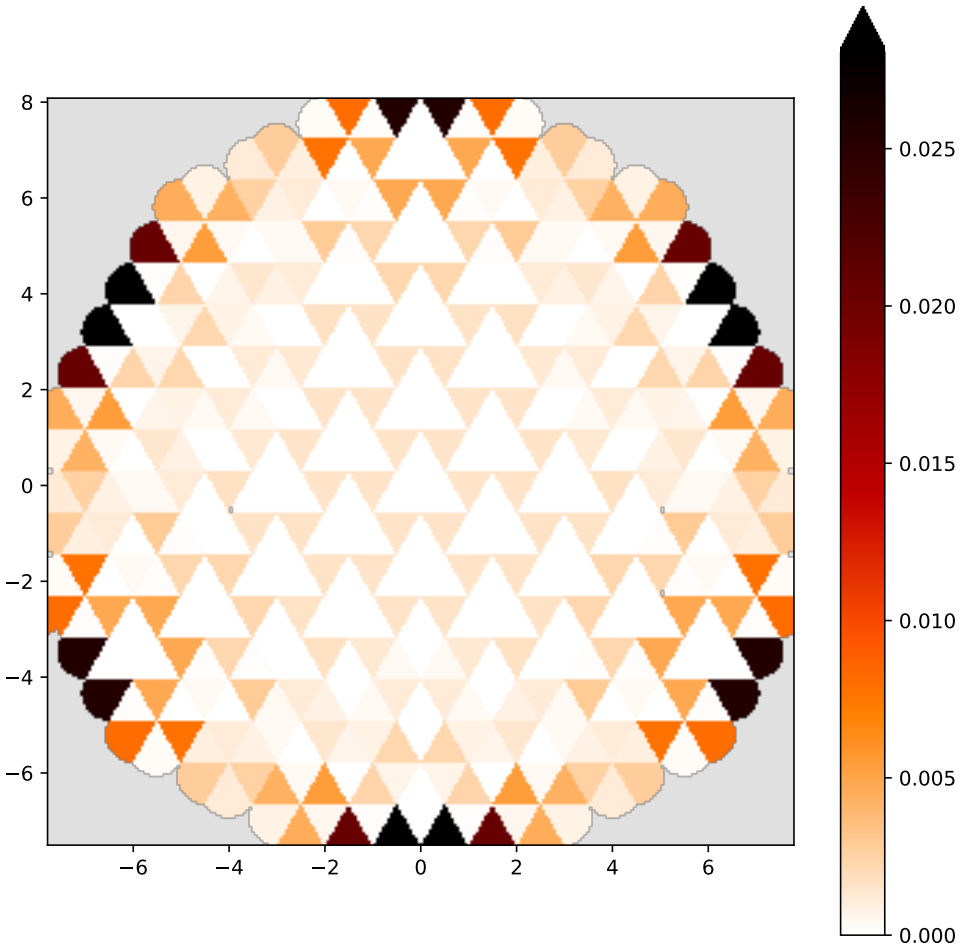
As an example, we now compute the eigenstates of the graphene quantum dot and intend to plot the wave function probability in the quantum dot. For aesthetic reasons (the wave functions look a bit nicer), we restrict ourselves to nearest-neighbor hopping. Computing the wave functions is done in the usual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

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

    syst = syst.finalized()
    ham = syst.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(syst, 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 = syst.sites[i]
        return ('p', 3, 180) if site.family == a else ('p', 3, 0)

    def family_color(i):
        return 'black' if syst.sites[i].family == a else 'white'

    kwant.plot(syst, site_color=wf, site_symbol=family_shape,
```

(continues on next page)

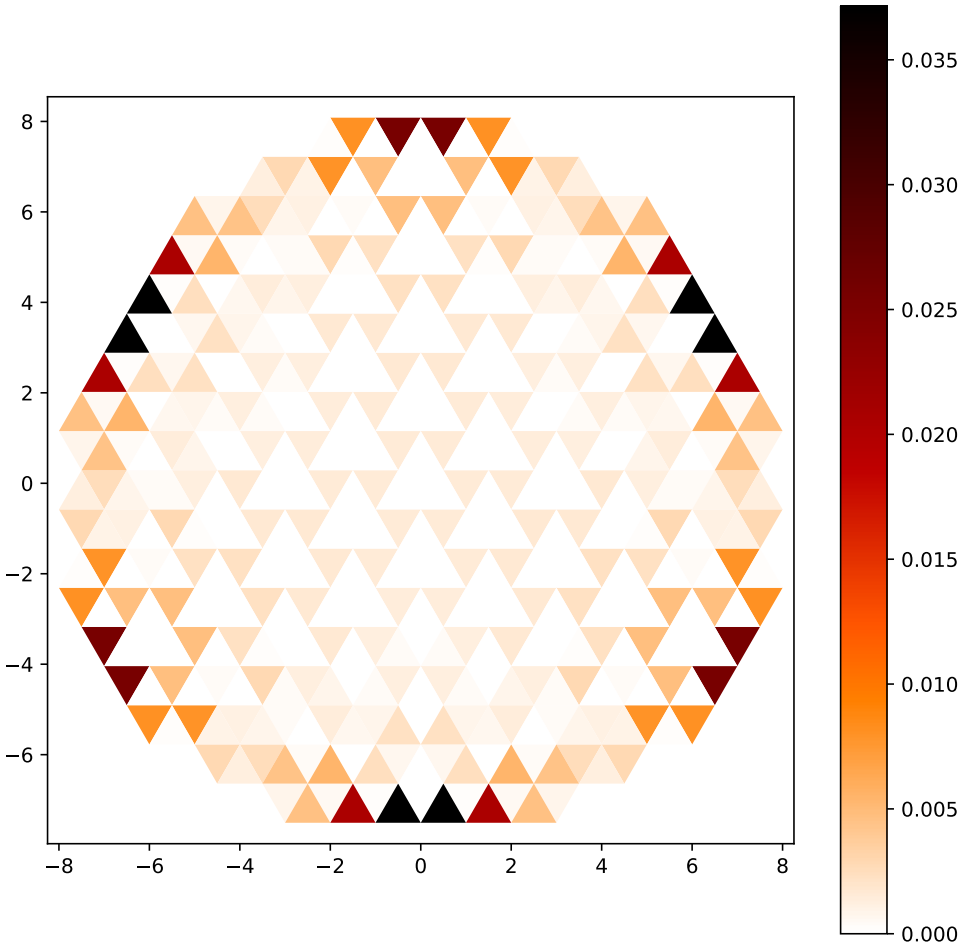
(continued from previous page)

```
                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). Moreover, `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*, `syst.sites[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()
```

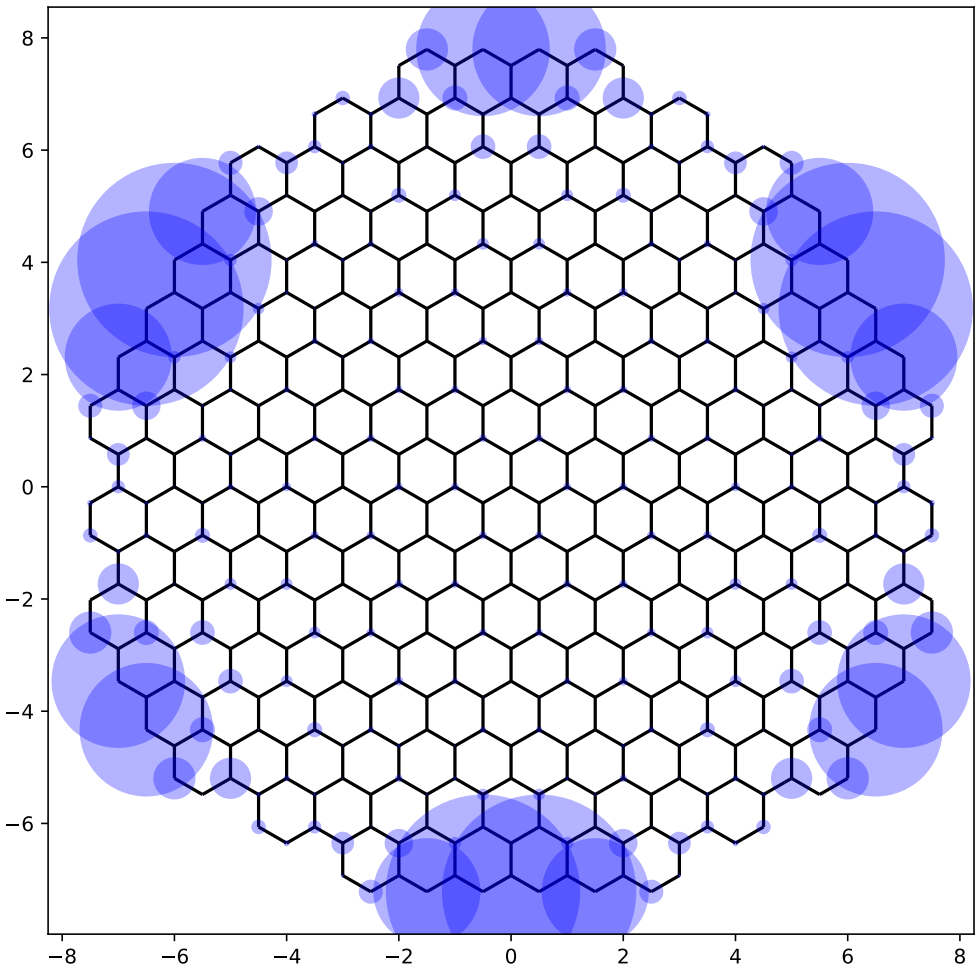
(continues on next page)

(continued from previous page)

```
    kwant.plot(syst, 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.

2.8.2 3D example: zincblende structure

See also:

The complete source code of this example can be found in `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:

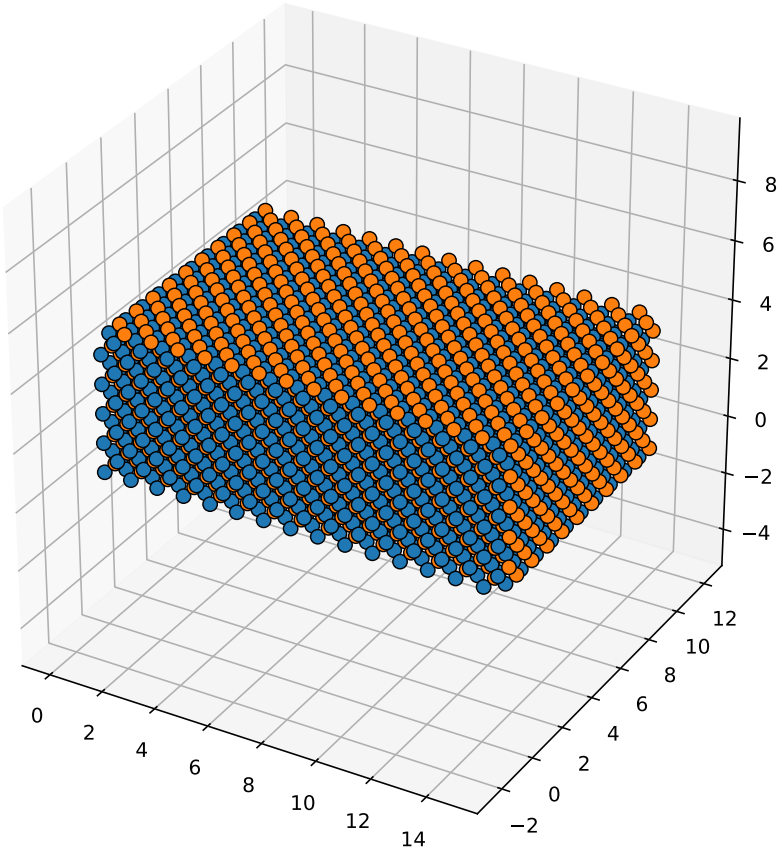
```
def make_cuboid(a=15, b=10, c=5):
    def cuboid_shape(pos):
        x, y, z = pos
        return 0 <= x < a and 0 <= y < b and 0 <= z < c

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

    return syst
```

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:

```
    syst = make_cuboid()

    kwant.plot(syst)
```

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:

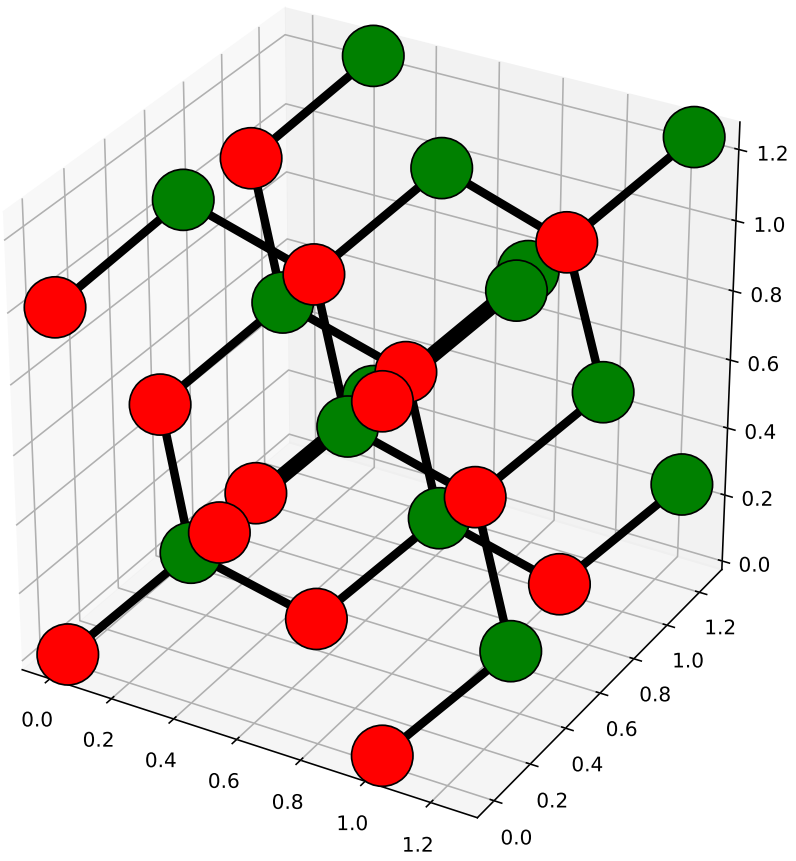
```
syst = make_cuboid(a=1.5, b=1.5, c=1.5)

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

(continues on next page)

(continued from previous page)

```
kwant.plot(syst, 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.

2.9 Calculating spectral density with the kernel polynomial method

We have already seen in the "*Closed systems*" tutorial that we can use Kwant simply to build Hamiltonians, which we can then directly diagonalize using routines from Scipy.

This already allows us to treat systems with a few thousand sites without too many problems. For larger systems one is often not so interested in the exact eigenenergies and eigenstates, but more in the *density of states*.

The kernel polynomial method (KPM), is an algorithm to obtain a polynomial expansion of the density of states. It can also be used to calculate the spectral density of arbitrary operators. Kwant has an implementation of the KPM method `kwant.kpm`, that is based on the algorithms presented in Ref.[1].

See also:

The complete source code of this example can be found in `kernel_polynomial_method.py`

2.9.1 Introduction

Our aim is to use the kernel polynomial method to obtain the spectral density $\rho_A(E)$, as a function of the energy E, of some Hilbert space operator A. We define

$$\rho_A(E) = \rho(E)A(E),$$

where $A(E)$ is the expectation value of A for all the eigenstates of the Hamiltonian with energy E, and the density of states is

$$\rho(E) = \sum_{k=0}^{D-1} \delta(E - E_k) = \mathrm{Tr}\left(\delta(E - H)\right),$$

where H is the Hamiltonian of the system, D the Hilbert space dimension, and E_k the eigenvalues.

In the special case when A is the identity, then $\rho_A(E)$ is simply $\rho(E)$, the density of states.

2.9.2 Calculating the density of states

Roughly speaking, KPM approximates the density of states, or any other spectral density, by expanding the action of the Hamiltonian and operator of interest on a small set of *random vectors* (or *local vectors* for local density of states), as a sum of Chebyshev polynomials up to some order, and then averaging. The accuracy of the method can be tuned by modifying the order of the Chebyshev expansion and the number of vectors. See notes on *accuracy* below for details.

Performance and accuracy

The KPM method is especially well suited for large systems, and in the case when one is not interested in individual eigenvalues, but rather in obtaining an approximate spectral density.

The accuracy in the energy resolution is dominated by the number of moments. The lowest accuracy is at the center of the spectrum, while slightly higher accuracy is obtained at the edges of the spectrum. If we use the KPM method (with the Jackson kernel, see Ref.[1]) to describe a delta peak at the center of the spectrum, we will obtain a function similar to a Gaussian of width $\sigma = \pi a/N$, where N is the number of moments, and a is the width of the spectrum.

On the other hand, the random vectors will *explore* the range of the spectrum, and as the system gets bigger, the number of random vectors that are necessary to sample the whole spectrum reduces. Thus, a small number of random vectors is in general enough, and increasing this number will not result in a visible improvement of the approximation.

The global *spectral density* $\rho_A(E)$ is approximated by the stochastic trace, the average expectation value of random vectors r

$$\rho_A(E) = \mathrm{Tr}\left(A\delta(E - H)\right) \sim \frac{1}{R}\sum_r \langle r|A\delta(E - H)|r\rangle,$$

[1] Rev. Mod. Phys., Vol. 78, No. 1 (2006).

while the *local spectral density* for a site i is

$$\rho_A^i(E) = \langle i|A\delta(E-H)|i\rangle,$$

which is an exact expression.

Global spectral densities using random vectors

In the following example, we will use the KPM implementation in Kwant to obtain the (global) density of states of a graphene disk.

We start by importing kwant and defining our system.

```
# necessary imports
import kwant
import numpy as np


# define the system
def make_syst(r=30, t=-1, a=1):
    syst = kwant.Builder()
    lat = kwant.lattice.honeycomb(a, norbs=1)

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

    syst[lat.shape(circle, (0, 0))] = 0.
    syst[lat.neighbors()] = t
    syst.eradicate_dangling()

    return syst
```

After making a system we can then create a *SpectralDensity* object that represents the density of states for this system.

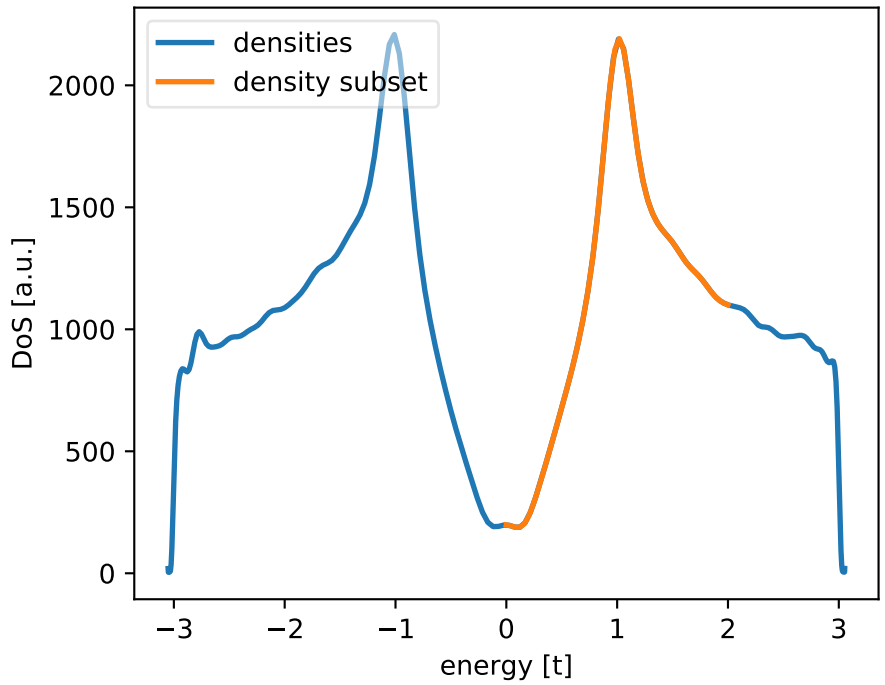
```
    fsyst = make_syst().finalized()

    spectrum = kwant.kpm.SpectralDensity(fsyst)
```

The *SpectralDensity* can then be called like a function to obtain a sequence of energies in the spectrum of the Hamiltonian, and the corresponding density of states at these energies.

```
    energies, densities = spectrum()
```

When called with no arguments, an optimal set of energies is chosen (these are not evenly distributed over the spectrum, see Ref.[1] for details), however it is also possible to provide an explicit sequence of energies at which to evaluate the density of states.

```
    energy_subset = np.linspace(0, 2)
    density_subset = spectrum(energy_subset)
```



In addition to being called like functions, *SpectralDensity* objects also have a method *integrate* which can be used to integrate the density of states against some distribution function over the whole spectrum. If no distribution function is specified, then the uniform distribution is used:

```
print('identity resolution:', spectrum.integrate())
```

```
identity resolution: (6493+0j)
```

We see that the integral of the density of states is normalized to the total number of available states in the system. If we wish to calculate, say, the number of states populated in equilibrium, then we should integrate with respect to a Fermi-Dirac distribution:

```
# Fermi energy 0.1 and temperature 0.2
fermi = lambda E: 1 / (np.exp((E - 0.1) / 0.2) + 1)

print('number of filled states:', spectrum.integrate(fermi))
```

```
number of filled states: (3270.802599135651-2.458110650324e-15j)
```

Stability and performance: spectral bounds

The KPM method internally rescales the spectrum of the Hamiltonian to the interval `(-1, 1)` (see Ref.[1] for details), which requires calculating the boundaries of the spectrum (using `scipy.sparse.linalg.eigsh`). This can be very costly for large systems, so it is possible to pass this explicitly as via the `bounds` parameter when instantiating the *SpectralDensity* (see the class documentation for details).

Additionally, *SpectralDensity* accepts a parameter `epsilon`, which ensures that the rescaled Hamiltonian (used internally), always has a spectrum strictly contained in the interval `(-1, 1)`. If bounds are not provided then the tolerance on the bounds calculated with `scipy.sparse.linalg.eigsh` is set to `epsilon/2`.

Local spectral densities using local vectors

The *local density of states* can be obtained without using random vectors, and using local vectors instead. This approach is best when we want to estimate the local density on a small number of sites of the system. The accuracy of this approach depends only on the number of moments, but the computational cost increases linearly with the number of sites sampled.

To output local densities for each local vector, and not the average, we set the parameter `mean=False`, and the local vectors will be created with the *`LocalVectors`* generator (see *advanced_topics* for details).

The spectral density can be restricted to the expectation value inside a region of the system by passing a `where` function or list of sites to the *`RandomVectors`* or *`LocalVectors`* generators.

In the following example, we compute the local density of states at the center of the graphene disk, and we add a staggering potential between the two sublattices.

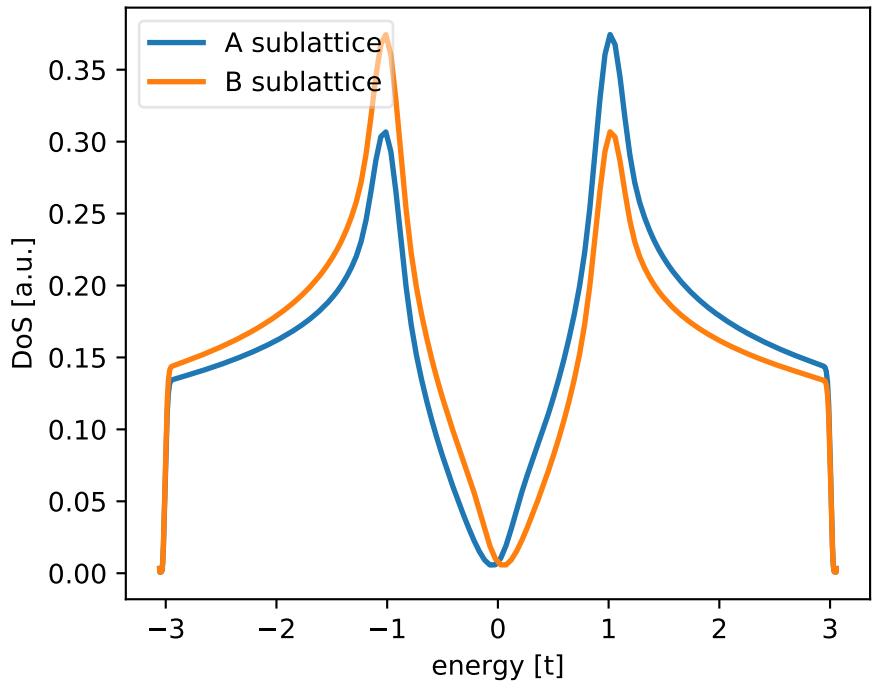
```
# define the system
def make_syst_staggered(r=30, t=-1, a=1, m=0.1):
    syst = kwant.Builder()
    lat = kwant.lattice.honeycomb(a, norbs=1)

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

    syst[lat.a.shape(circle, (0, 0))] = m
    syst[lat.b.shape(circle, (0, 0))] = -m
    syst[lat.neighbors()] = t
    syst.eradicate_dangling()

    return syst
```

Next, we choose one site of each sublattice "A" and "B",

```
    # find 'A' and 'B' sites in the unit cell at the center of the disk
    center_tag = np.array([0, 0])
    where = lambda s: s.tag == center_tag
    # make local vectors
    vector_factory = kwant.kpm.LocalVectors(fsyst, where)
```

and plot their respective local density of states.

```
    # 'num_vectors' can be unspecified when using 'LocalVectors'
    local_dos = kwant.kpm.SpectralDensity(fsyst, num_vectors=None,
                                          vector_factory=vector_factory,
                                          mean=False)
    energies, densities = local_dos()
    plot_dos([
        ('A sublattice', (energies, densities[:, 0])),
        ('B sublattice', (energies, densities[:, 1])),
    ])
```



Note that there is no noise comming from the random vectors.

2.9.3 Increasing the accuracy of the approximation

SpectralDensity has two methods for increasing the accuracy of the method, each of which offers different levels of control over what exactly is changed.

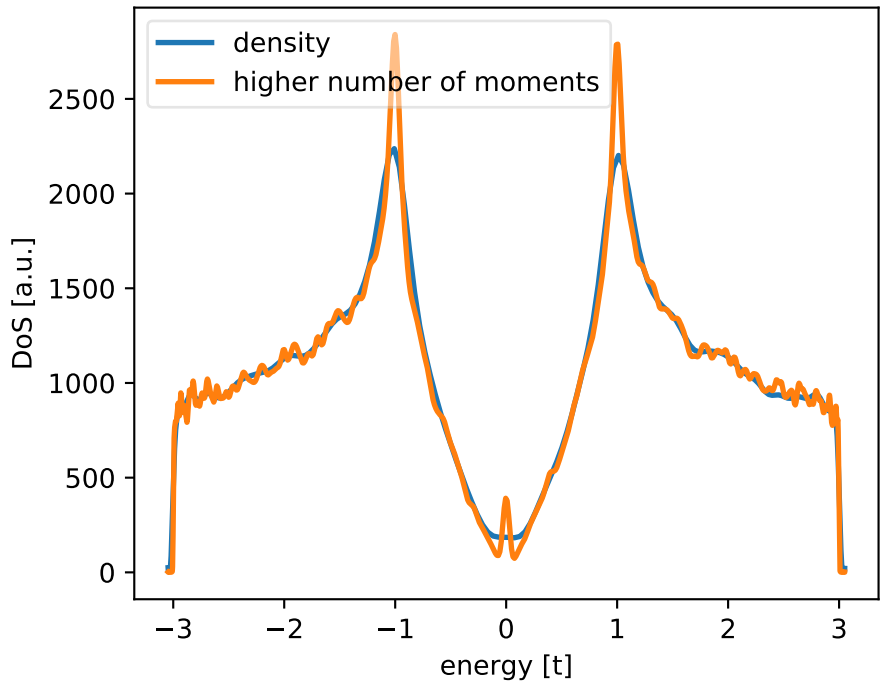
The simplest way to obtain a more accurate solution is to use the `add_moments` method:

```
spectrum.add_moments(energy_resolution=0.03)
```

This will update the number of calculated moments and also the default number of sampling points such that the maximum distance between successive energy points is `energy_resolution` (see notes on *accuracy*).

Alternatively, you can directly increase the number of moments with `add_moments`, or the number of random vectors with `add_vectors`. The added vectors will be generated with the `vector_factory`.

```
spectrum.add_moments(100)
spectrum.add_vectors(5)
```



2.9.4 Calculating the spectral density of an operator

Above, we saw how to calculate the density of states by creating a *SpectralDensity* and passing it a finalized Kwant system. When instantiating a *SpectralDensity* we may optionally supply an operator in addition to the system. In this case it is the spectral density of the given operator that is calculated.

SpectralDensity accepts the operators in a few formats:

- *explicit matrices* (numpy array or scipy sparse matrices will work)
- *operators* from *kwant.operator*

If an explicit matrix is provided then it must have the same shape as the system Hamiltonian.

```
# identity matrix
matrix_op = scipy.sparse.eye(len(fsyst.sites))
matrix_spectrum = kwant.kpm.SpectralDensity(fsyst, operator=matrix_op)
```

Or, to do the same calculation using *kwant.operator.Density*:

```
# 'sum=True' means we sum over all the sites
kwant_op = kwant.operator.Density(fsyst, sum=True)
operator_spectrum = kwant.kpm.SpectralDensity(fsyst, operator=kwant_op)
```

Spectral density with random vectors

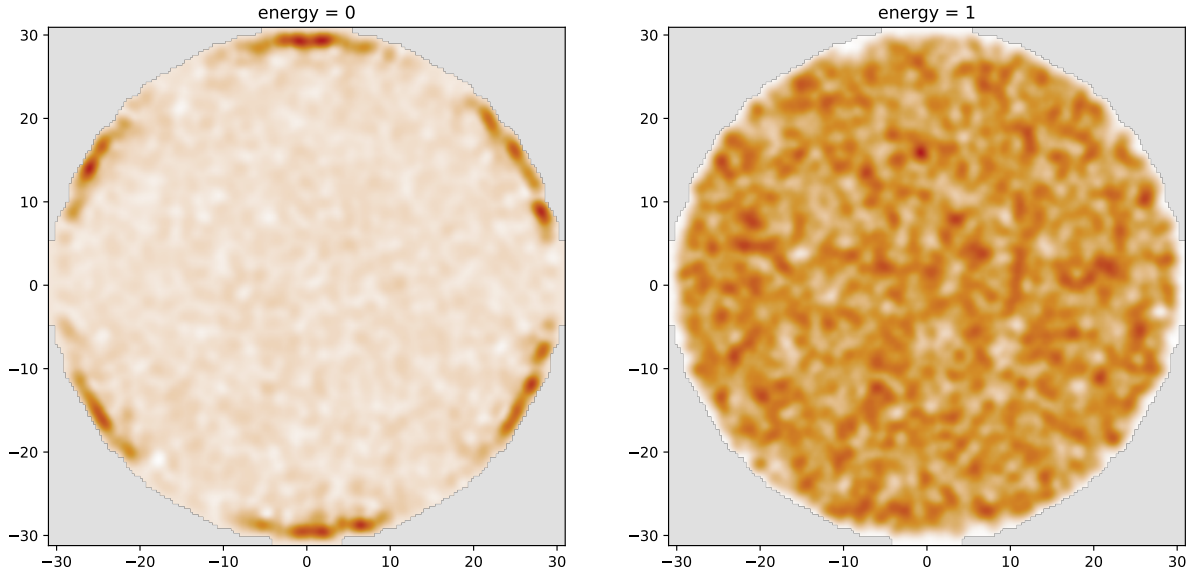
Using operators from *kwant.operator* allows us to calculate quantities such as the *local* density of states by telling the operator not to sum over all the sites of the system:

```
# 'sum=False' is the default, but we include it explicitly here for clarity.
kwant_op = kwant.operator.Density(fsyst, sum=False)
local_dos = kwant.kpm.SpectralDensity(fsyst, operator=kwant_op)
```

SpectralDensity will properly handle this vector output, and will average the local density obtained with random vectors.

The accuracy of this approximation depends on the number of random vectors used. This allows us to plot an approximate local density of states at different points in the spectrum:

```
    zero_energy_ldos = local_dos(energy=0)
    finite_energy_ldos = local_dos(energy=1)
    plot_ldos(fsyst, [
        ('energy = 0', zero_energy_ldos),
        ('energy = 1', finite_energy_ldos)
    ])
```



2.9.5 Calculating Kubo conductivity

The Kubo conductivity can be calculated for a closed system with two KPM expansions. In *Conductivity* we implemented the Kubo-Bastin formula of the conductivity and any temperature (see Ref.[2]). With the help of *Conductivity*, we can calculate any element of the conductivity tensor $\sigma_{\alpha\beta}$, that relates the applied electric field to the expected current.

$$j_\alpha = \sigma_{\alpha,\beta} E_\beta$$

In the following example, we will calculate the longitudinal conductivity σ_{xx} and the Hall conductivity σ_{xy}, for the Haldane model. This model is the first and one of the most simple ones for a topological insulator.

The Haldane model consist of a honeycomb lattice, similar to graphene, with nearest neigbours hoppings. To turn it into a topological insulator we add imaginary second nearest neigbours hoppings, where the sign changes for each sublattice.

```
# define a Haldane system
def make_syst_topo(r=30, a=1, t=1, t2=0.5):
    syst = kwant.Builder()
    lat = kwant.lattice.honeycomb(a, norbs=1, name=['a', 'b'])

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

    syst[lat.shape(circle, (0, 0))] = 0.
    syst[lat.neighbors()] = t
    # add second neighbours hoppings
    syst[lat.a.neighbors()] = 1j * t2
    syst[lat.b.neighbors()] = -1j * t2
    syst.eradicate_dangling()
```

(continues on next page)

[2] Phys. Rev. Lett. 114, 116602 (2015).

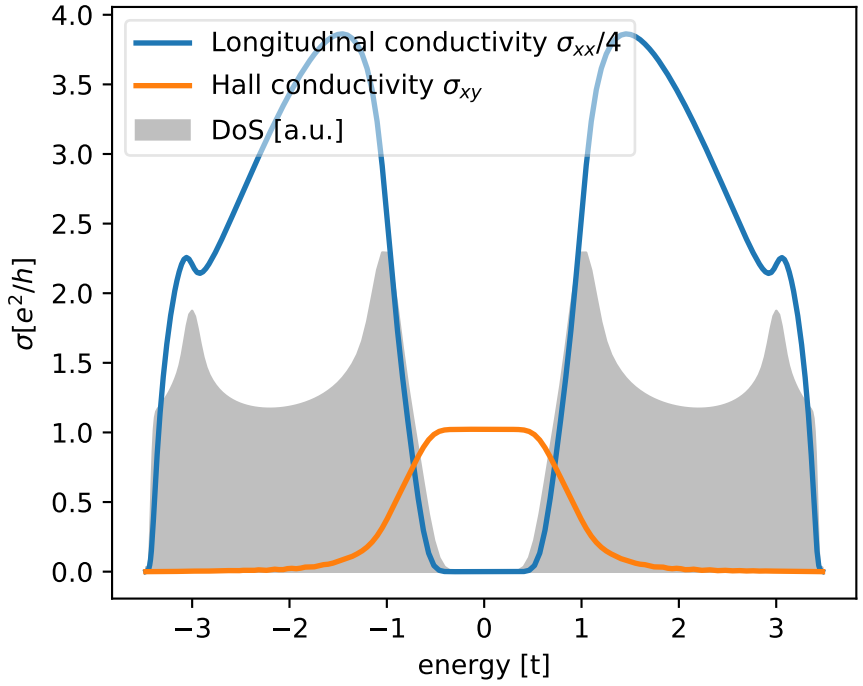
(continued from previous page)

```
    return lat, syst.finalized()
```

To calculate the bulk conductivity, we will select sites in the unit cell in the middle of the sample, and create a vector factory that outputs local vectors

```
# construct the Haldane model
lat, fsyst = make_syst_topo()
# find 'A' and 'B' sites in the unit cell at the center of the disk
where = lambda s: np.linalg.norm(s.pos) < 1

# component 'xx'
s_factory = kwant.kpm.LocalVectors(fsyst, where)
cond_xx = kwant.kpm.conductivity(fsyst, alpha='x', beta='x', mean=True,
                                 num_vectors=None, vector_factory=s_factory)
# component 'xy'
s_factory = kwant.kpm.LocalVectors(fsyst, where)
cond_xy = kwant.kpm.conductivity(fsyst, alpha='x', beta='y', mean=True,
                                 num_vectors=None, vector_factory=s_factory)

energies = cond_xx.energies
cond_array_xx = np.array([cond_xx(e, temperature=0.01) for e in energies])
cond_array_xy = np.array([cond_xy(e, temperature=0.01) for e in energies])

# area of the unit cell per site
area_per_site = np.abs(np.cross(*lat.prim_vecs)) / len(lat.sublattices)
cond_array_xx /= area_per_site
cond_array_xy /= area_per_site
```

Note that the Kubo conductivity must be normalized with the area covered by the vectors. In this case, each local vector represents a site, and covers an area of half a unit cell, while the sum covers one unit cell. It is possible to use random vectors to get an average expectation value of the conductivity over large parts of the system. In this case, the area that normalizes the result, is the area covered by the random vectors.



2.9.6 Advanced topics

Custom distributions of vectors

By default *SpectralDensity* will use random vectors whose components are unit complex numbers with phases drawn from a uniform distribution. The generator is accesible through *RandomVectors*.

For large systems, one will generally resort to random vectors to sample the Hilbert space of the system. There are several reasons why you may wish to make a different choice of distribution for your random vectors, for example to enforce certain symmetries or to only use real-valued vectors.

To change how the random vectors are generated, you need only specify a function that takes the dimension of the Hilbert space as a single parameter, and which returns a vector in that Hilbert space:

```
# construct a generator of vectors with n random elements -1 or +1.
n = fsyst.hamiltonian_submatrix(sparse=True).shape[0]
def binary_vectors():
    while True:
        yield np.rint(np.random.random_sample(n)) * 2 - 1

custom_factory = kwant.kpm.SpectralDensity(fsyst,
                                           vector_factory=binary_vectors())
```

Aditionally, a *LocalVectors* generator is also available, that returns local vectors that correspond to the sites passed. Note that the vectors generated match the sites in order, and will be exhausted after all vectors are drawn from the factory.

Both *RandomVectors* and *LocalVectors* take the argument `where`, that restricts the non zero values of the vectors generated to the sites in `where`.

Reproducible calculations

Because KPM internally uses random vectors, running the same calculation twice will not give bit-for-bit the same result. However, similarly to the funcions in *rmt*, the random number generator can be directly manipulated by passing a value to the `rng` parameter of *SpectralDensity*. `rng` can itself be a random number generator, or it may simply be a seed to pass to the numpy random number generator (that is used internally by default).

Defining operators as sesquilinear maps

Above, we showed how *SpectralDensity* can calculate the spectral density of operators, and how we can define operators by using *kwant.operator*. If you need even more flexibility, *SpectralDensity* will also accept a *function* as its `operator` parameter. This function must itself take two parameters, `(bra, ket)` and must return either a scalar or a one-dimensional array. In order to be meaningful the function must be a sesquilinear map, i.e. antilinear in its first argument, and linear in its second argument. Below, we compare two methods for computing the local density of states, one using *kwant.operator.Density*, and the other using a custom function.

```
rho = kwant.operator.Density(fsyst, sum=True)

# sesquilinear map that does the same thing as `rho`
def rho_alt(bra, ket):
    return np.vdot(bra, ket)

rho_spectrum = kwant.kpm.SpectralDensity(fsyst, operator=rho)
rho_alt_spectrum = kwant.kpm.SpectralDensity(fsyst, operator=rho_alt)
```

References

2.10 Discretizing continuous Hamiltonians

2.10.1 Introduction

In "*Discretization of a Schrödinger Hamiltonian*" we have learnt that Kwant works with tight-binding Hamiltonians. Often, however, one will start with a continuum model and will subsequently need to discretize it to arrive at a tight-binding model. Although discretizing a Hamiltonian is usually a simple process, it is tedious and repetitive. The situation is further exacerbated when one introduces additional on-site degrees of freedom, and tracking all the necessary terms becomes a chore. The *continuum* sub-package aims to be a solution to this problem. It is a collection of tools for working with continuum models and for discretizing them into tight-binding models.

See also:

The complete source code of this tutorial can be found in `discretize.py`

2.10.2 Discretizing by hand

As an example, let us consider the following continuum Schrödinger equation for a semiconducting heterostructure (using the effective mass approximation):

$$\left(k_x \frac{\hbar^2}{2m(x)} k_x\right) \psi(x) = E\,\psi(x).$$

Replacing the momenta by their corresponding differential operators

$$k_\alpha = -i\partial_\alpha,$$

for $\alpha = x, y$ or z, and discretizing on a regular lattice of points with spacing a, we obtain the tight-binding model

$$H = -\frac{1}{a^2}\sum_i A\left(x+\frac{a}{2}\right)\left(|i\rangle\langle i+1| + h.c.\right) + \frac{1}{a^2}\sum_i \left(A\left(x+\frac{a}{2}\right) + A\left(x-\frac{a}{2}\right)\right)|i\rangle\langle i|,$$

with $A(x) = \frac{\hbar^2}{2m(x)}$.

2.10.3 Using `discretize` to obtain a template

First we must explicitly import the *kwant.continuum* package:

```
import kwant.continuum
```

The function *kwant.continuum.discretize* takes a symbolic Hamiltonian and turns it into a *Builder* instance with appropriate spatial symmetry that serves as a template. (We will see how to use the template to build systems with a particular shape later).

```
    template = kwant.continuum.discretize('k_x * A(x) * k_x')
    print(template)
```

It is worth noting that `discretize` treats `k_x` and `x` as non-commuting operators, and so their order is preserved during the discretization process.

The builder produced by `discretize` may be printed to show the source code of its onsite and hopping functions (this is a special feature of builders returned by `discretize`):

```
# Discrete coordinates: x

# Onsite element:
def onsite(site, A):
```

(continues on next page)

(continued from previous page)

```
    (x, ) = site.pos
    _const_0 = (A(0.5 + x))
    _const_1 = (A(-0.5 + x))
    return (1.0*_const_0 + 1.0*_const_1)

# Hopping from (1,):
def hopping_1(site1, site2, A):
    (x, ) = site1.pos
    _const_0 = (A(0.5 + x))
    return (-1.0*_const_0)
```

Technical details

- `kwant.continuum` uses `sympy` internally to handle symbolic expressions. Strings are converted using *kwant.continuum.sympify*, which essentially applies some Kwant-specific rules (such as treating `k_x` and `x` as non-commutative) before calling `sympy.sympify`
- The builder returned by `discretize` will have an N-D translational symmetry, where `N` is the number of dimensions that were discretized. This is the case, even if there are expressions in the input (e.g. `V(x, y)`) which in principle *may not* have this symmetry. When using the returned builder directly, or when using it as a template to construct systems with different/lower symmetry, it is important to ensure that any functional parameters passed to the system respect the symmetry of the system. Kwant provides no consistency check for this.
- The discretization process consists of taking input $H(k_x, k_y, k_z)$, multiplying it from the right by $\psi(x, y, z)$ and iteratively applying a second-order accurate central derivative approximation for every $k_\alpha = -i\partial_\alpha$:

  $$\partial_\alpha \psi(\alpha) = \frac{1}{a}\left(\psi\left(\alpha + \frac{a}{2}\right) - \psi\left(\alpha - \frac{a}{2}\right)\right).$$

  This process is done separately for every summand in Hamiltonian. Once all symbols denoting operators are applied internal algorithm is calculating `gcd` for hoppings coming from each summand in order to find best possible approximation. Please see source code for details.
- Instead of using `discretize` one can use *discretize_symbolic* to obtain symbolic output. When working interactively in Jupyter notebooks it can be useful to use this to see a symbolic representation of the discretized Hamiltonian. This works best when combined with `sympy` Pretty Printing.
- The symbolic result of discretization obtained with `discretize_symbolic` can be converted into a builder using *build_discretized*. This can be useful if one wants to alter the tight-binding Hamiltonian before building the system.

2.10.4 Building a Kwant system from the template

Let us now use the output of `discretize` as a template to build a system and plot some of its energy eigenstate. For this example the Hamiltonian will be

$$H = k_x^2 + k_y^2 + V(x, y),$$

where $V(x, y)$ is some arbitrary potential.

First, use `discretize` to obtain a builder that we will use as a template:

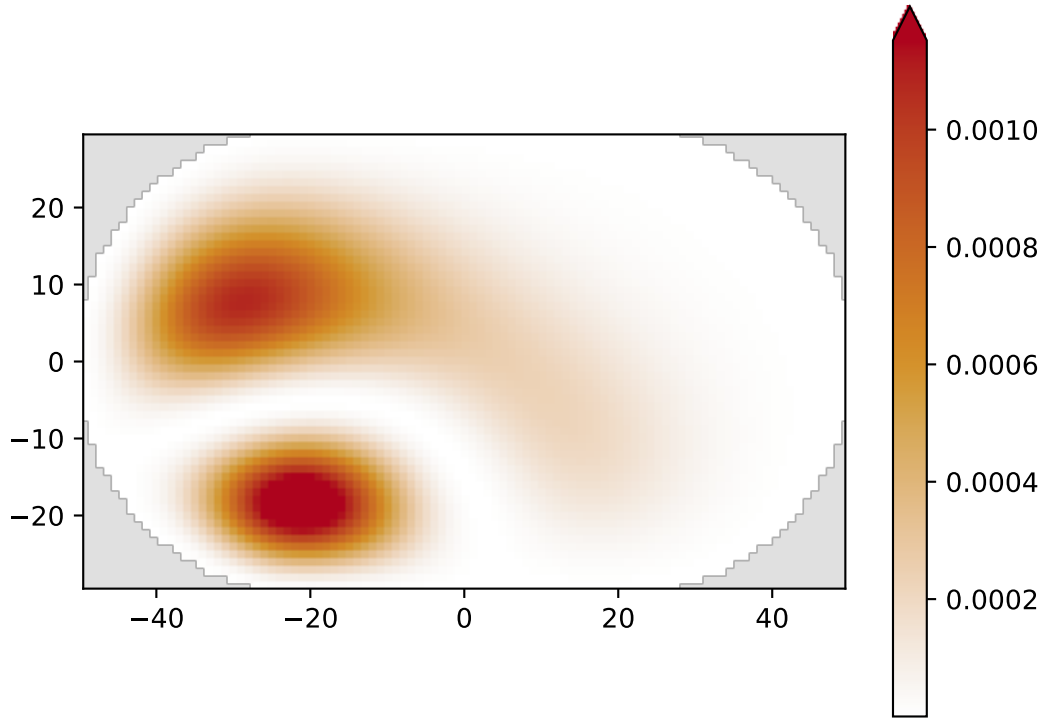
```
hamiltonian = "k_x**2 + k_y**2 + V(x, y)"
template = kwant.continuum.discretize(hamiltonian)
print(template)
```

We now use this system with the *fill* method of *Builder* to construct the system we want to investigate:

```
    def stadium(site):
        (x, y) = site.pos
        x = max(abs(x) - 20, 0)
        return x**2 + y**2 < 30**2

    syst = kwant.Builder()
    syst.fill(template, stadium, (0, 0));
    syst = syst.finalized()
```

After finalizing this system, we can plot one of the system's energy eigenstates:

```
def plot_eigenstate(syst, n=2, Vx=.0003, Vy=.0005):

    def potential(x, y):
        return Vx * x + Vy * y

    ham = syst.hamiltonian_submatrix(params=dict(V=potential), sparse=True)
    evecs = scipy.sparse.linalg.eigsh(ham, k=10, which='SM')[1]
    kwant.plotter.map(syst, abs(evecs[:, n])**2, show=False)
```



Note in the above that we pass the spatially varying potential *function* to our system via a parameter called `V`, because the symbol V was used in the initial, symbolic, definition of the Hamiltonian.

In addition, the function passed as `V` expects two input parameters `x` and `y`, the same as in the initial continuum Hamiltonian.

2.10.5 Models with more structure: Bernevig-Hughes-Zhang

When working with multi-band systems, like the Bernevig-Hughes-Zhang (BHZ) model[1][2], one can provide matrix input to *discretize* using `identity` and `kron`. For example, the definition of the BHZ model can be written succinctly as:

[1] Science, 314, 1757 (2006).
[2] Phys. Rev. B 82, 045122 (2010).

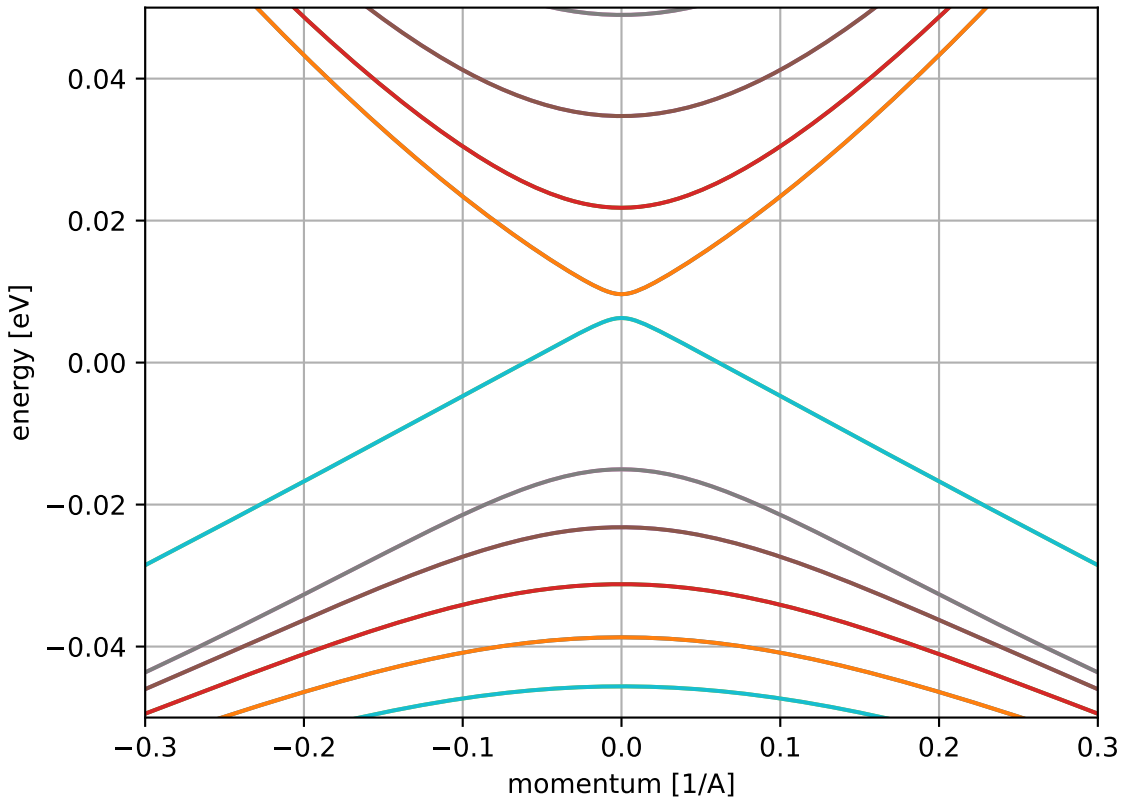
```
    hamiltonian = """
       + C * identity(4) + M * kron(sigma_0, sigma_z)
       - B * (k_x**2 + k_y**2) * kron(sigma_0, sigma_z)
       - D * (k_x**2 + k_y**2) * kron(sigma_0, sigma_0)
       + A * k_x * kron(sigma_z, sigma_x)
       - A * k_y * kron(sigma_0, sigma_y)
    """

    template = kwant.continuum.discretize(hamiltonian, grid=a)
```

We can then make a ribbon out of this template system:

```
    def shape(site):
        (x, y) = site.pos
        return (0 <= y < W and 0 <= x < L)

    def lead_shape(site):
        (x, y) = site.pos
        return (0 <= y < W)

    syst = kwant.Builder()
    syst.fill(template, shape, (0, 0))

    lead = kwant.Builder(kwant.TranslationalSymmetry([-a, 0]))
    lead.fill(template, lead_shape, (0, 0))

    syst.attach_lead(lead)
    syst.attach_lead(lead.reversed())

    syst = syst.finalized()
```

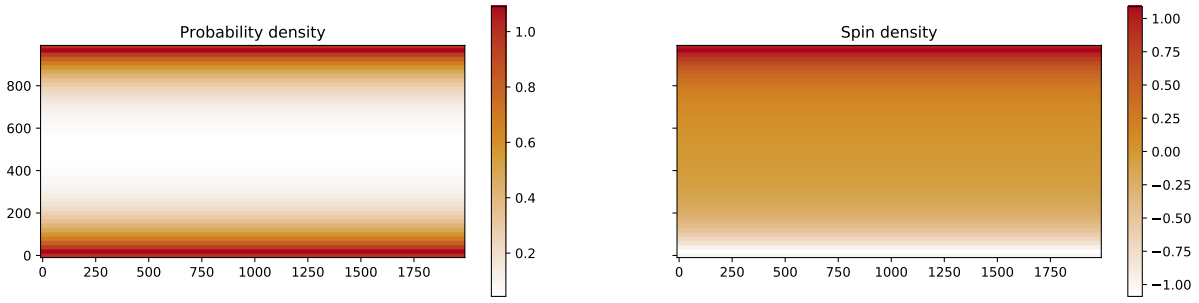
and plot its dispersion using *kwant.plotter.bands*:

```
    kwant.plotter.bands(syst.leads[0], params=params,
                        momenta=np.linspace(-0.3, 0.3, 201), show=False)
```



In the above we see the edge states of the quantum spin Hall effect, which we can visualize using *kwant.plotter.map*:

```
# get scattering wave functions at E=0
wf = kwant.wave_function(syst, energy=0, params=params)

# prepare density operators
sigma_z = np.array([[1, 0], [0, -1]])
prob_density = kwant.operator.Density(syst, np.eye(4))
spin_density = kwant.operator.Density(syst, np.kron(sigma_z, np.eye(2)))

# calculate expectation values and plot them
wf_sqr = sum(prob_density(psi) for psi in wf(0))  # states from left lead
rho_sz = sum(spin_density(psi) for psi in wf(0))  # states from left lead

fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(16, 4))
kwant.plotter.map(syst, wf_sqr, ax=ax1)
kwant.plotter.map(syst, rho_sz, ax=ax2)
```



2.10.6 Limitations of discretization

It is important to remember that the discretization of a continuum model is an *approximation* that is only valid in the low-energy limit. For example, the quadratic continuum Hamiltonian

$$H_{\text{continuous}}(k_x) = \frac{\hbar^2}{2m}k_x^2$$

and its discretized approximation

$$H_{\text{tight-binding}}(k_x) = 2t(1 - \cos(k_x a)),$$

where $t = \frac{\hbar^2}{2ma^2}$, are only valid in the limit $E < t$. The grid spacing a must be chosen according to how high in energy you need your tight-binding model to be valid.

It is possible to set a through the `grid` parameter to *`discretize`*, as we will illustrate in the following example. Let us start from the continuum Hamiltonian

$$H(k) = k_x^2 \mathbb{1}_{2\times 2} + \alpha k_x \sigma_y.$$

We start by defining this model as a string and setting the value of the α parameter:

```
hamiltonian = "k_x**2 * identity(2) + alpha * k_x * sigma_y"
params = dict(alpha=.5)
```

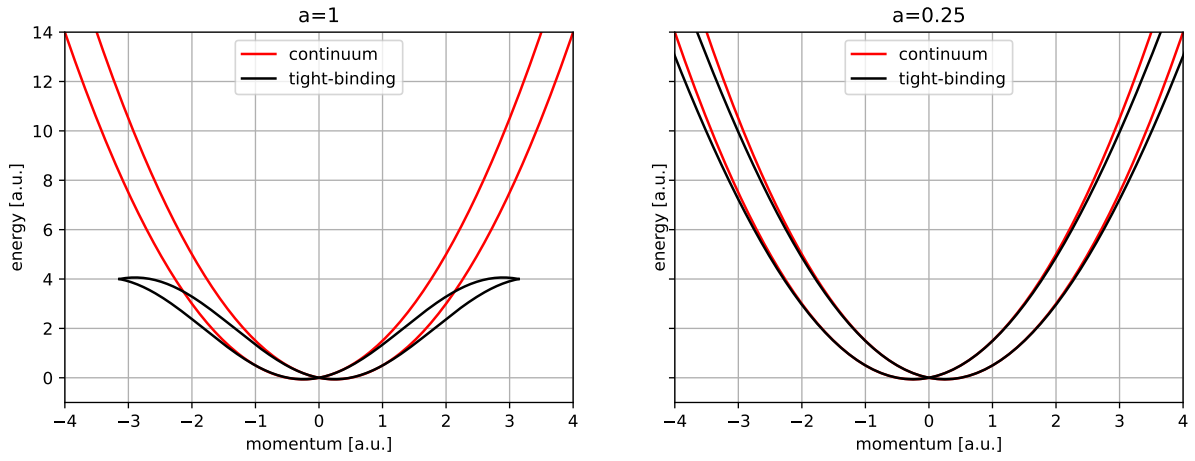
Now we can use *`kwant.continuum.lambdify`* to obtain a function that computes $H(k)$:

```
h_k = kwant.continuum.lambdify(hamiltonian, locals=params)
k_cont = np.linspace(-4, 4, 201)
e_cont = [scipy.linalg.eigvalsh(h_k(k_x=ki)) for ki in k_cont]
```

We can also construct a discretized approximation using *`kwant.continuum.discretize`*, in a similar manner to previous examples:

```
template = kwant.continuum.discretize(hamiltonian, grid=a)
syst = kwant.wraparound.wraparound(template).finalized()

def h_k(k_x):
    p = dict(k_x=k_x, **params)
    return syst.hamiltonian_submatrix(params=p)

k_tb = np.linspace(-np.pi/a, np.pi/a, 201)
e_tb = [scipy.linalg.eigvalsh(h_k(k_x=a*ki)) for ki in k_tb]
```

Below we can see the continuum and tight-binding dispersions for two different values of the discretization grid spacing a:



We clearly see that the smaller grid spacing is, the better we approximate the original continuous dispersion. It is also worth remembering that the Brillouin zone also scales with grid spacing: $[-\frac{\pi}{a}, \frac{\pi}{a}]$.

2.10.7 Advanced topics

The input to *kwant.continuum.discretize* and *kwant.continuum.lambdify* can be not only a `string`, as we saw above, but also a `sympy` expression or a `sympy` matrix. This functionality will probably be mostly useful to people who are already experienced with `sympy`.

It is possible to use `identity` (for identity matrix), `kron` (for Kronecker product), as well as Pauli matrices `sigma_0`, `sigma_x`, `sigma_y`, `sigma_z` in the input to *lambdify* and *discretize*, in order to simplify expressions involving matrices. Matrices can also be provided explicitly using square `[]` brackets. For example, all following expressions are equivalent:

```
    sympify = kwant.continuum.sympify
    subs = {'sx': [[0, 1], [1, 0]], 'sz': [[1, 0], [0, -1]]}

    e = (
        sympify('[[k_x**2, alpha * k_x], [k_x * alpha, -k_x**2]]'),
        sympify('k_x**2 * sigma_z + alpha * k_x * sigma_x'),
        sympify('k_x**2 * sz + alpha * k_x * sx', locals=subs),
    )

    print(e[0] == e[1] == e[2])
```

```
True
```

We can use the `locals` keyword parameter to substitute expressions and numerical values:

```
    subs = {'A': 'A(x) + B', 'V': 'V(x) + V_0', 'C': 5}
    print(sympify('k_x * A * k_x + V + C', locals=subs))
```

```
V_0 + 5 + k_x*(B + A(x))*k_x + V(x)
```

Symbolic expressions obtained in this way can be directly passed to all `discretizer` functions. **Technical details**

Because of the way that `sympy` handles commutation relations all symbols representing position and momentum operators are set to be non commutative. This means that the order of momentum and position operators in the input expression is preserved. Note that it is not possible to define individual commutation relations within `sympy`, even expressions such xk_yx will not be simplified, even though mathematically $[x, k_y] = 0$.

References

2.11 Frequently asked questions

This FAQ complements the regular Kwant tutorials and thus does not cover questions that are discussed there. The Kwant paper also digs deeper into Kwant's structure.

2.11.1 What is a system, and what is a builder?

A Kwant system represents a particular tight-binding model. It contains a graph whose edges and vertices are assigned values, and that corresponds to the Hamiltonian matrix of the model being simulated.

In Kwant the creation of the system is separated from its use in numerical calculations. First an instance of the *Builder* class is used to construct the model, then the *finalize* method is called, which produces a so-called low-level *System* that can be used by Kwant's solvers.

The interface of builders mimics Python mappings (e.g. dictionaries). The familiar square-bracket syntax allows to set, get and delete items that correspond to elements of the system graph, e.g. `syst[key] = value`. An item consists of a key and an associated value. Keys are *sites* and *hoppings*. Values can be numbers, arrays of numbers, or functions that return numbers or arrays.

Finalizing a builder returns a copy of the system with the graph *structure* frozen. (This can be equivalently seen as freezing the system geometry or the sparsity structure of the Hamiltonian.) The associated *values* are taken over verbatim. Note that finalizing does not freeze the Hamiltonian matrix: only its structure is fixed, values that are functions may depend on an arbitrary number of parameters.

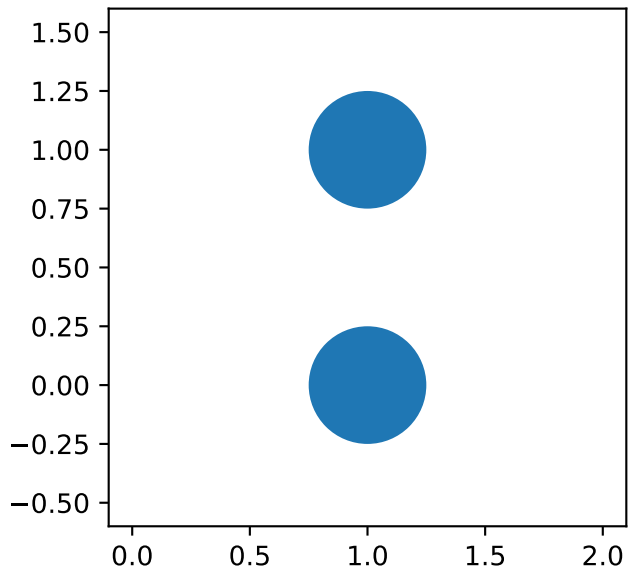
In the documentation and in mailing list discussions, the general term "system" can refer either to a `Builder` or to a low-level `System`, and the context will determine which specific class is being referred to. The terms "builder" and "low-level system" (or "finalized system") refer respectively to `Builder` and `System`.

2.11.2 What is a site?

Kwant is a tool for working with tight-binding models, which can be viewed as a graph composed of edges and vertices. Sites are Kwant's labels for the vertices. Sites have two attributes: a *family* and a *tag*. The combination of family and tag uniquely defines a site.

For example let us create an empty tight binding system and add two sites:

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

syst[lat(1, 0)] = 4
syst[lat(1, 1)] = 4

kwant.plot(syst)
```



In the above snippet we added 2 sites: `lat(1, 0)` and `lat(0, 1)`. Both of these sites belong to the same family, `lat`, but have different tags: `(1, 0)` and `(0, 1)` respectively.

Both sites were given the value 4 which means that the above system corresponds to the Hamiltonian matrix

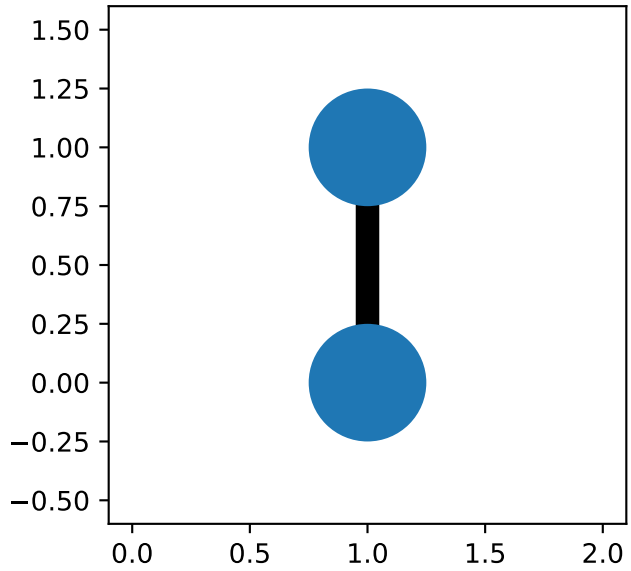
$$H = \begin{pmatrix} 4 & 0 \\ 0 & 4 \end{pmatrix}.$$

2.11.3 What is a hopping?

A hopping is simply a tuple of two of sites, which defines an edge of the graph that makes up a tight-binding model. Other sequences of sites that are not tuples, for example lists, are not treated as hoppings.

Starting from the example code from *What is a site?*, we can add a hopping to our system in the following way:

```
syst[(lat(1, 0), lat(1, 1))] = 1j
```



Visually, a hopping is represented as a line that joins two sites.

The Hamiltonian matrix is now

$$H = \begin{pmatrix} 4 & i \\ -i & 4 \end{pmatrix}.$$

Note how adding `(site_a, site_b)` to a system and assigning it a value `v`, implicitly adds the hopping `(site_b, site_a)` with the Hermitian conjugate of `v` as value.

2.11.4 What is a site family, and what is a tag?

A site family groups related sites together, and a tag serves as a unique identifier for a site within a given family.

In the previous example we saw a family that was suggestively called `lat`, which had sites whose tags were pairs of integers. In this specific example the site family also happens to be a regular Bravais lattice, and the tags take on the meaning of lattice coordinates for a site on this lattice.

The concept of families and tags is, however, more general. For example, one could implement a mesh that can be locally refined in certain areas, by having a family where sites belong to a quadtree, or an amorphous blob where sites are tagged by letters of the alphabet.

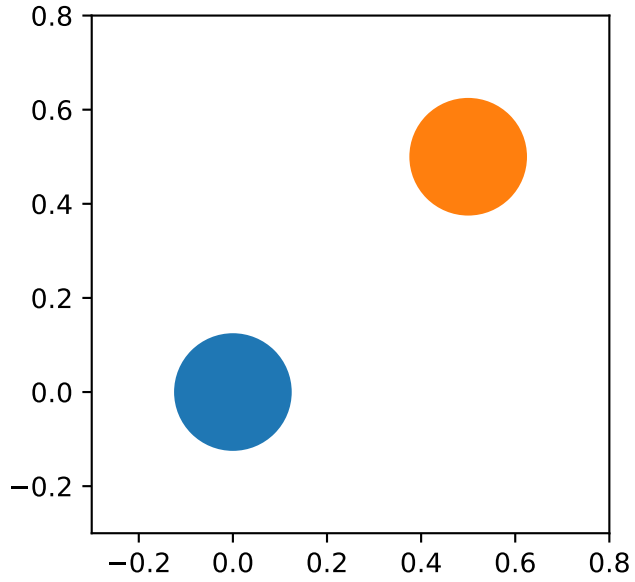
2.11.5 What is a lattice?

Kwant allows to define and use Bravais lattices for dealing with collections of regularly placed sites. They know about things like what sites are neighbors, or what sites belong to a given region of real space. *`Monatomic`* lattices have a single site in their basis, while *`Polyatomic`* lattices have more than one site in their basis.

Monatomic lattices in Kwant *are also site families*, with sites that are tagged by tuples of integers: the site's coordinates in the basis of primitive vectors of the lattice. Polyatomic lattices, however, are *not* site families, since lattice coordinates are not enough information to uniquely identify a site if there is more than one site in the basis. Polyatomic lattices do, however, have an attribute `sublattices` that is a list of monatomic lattices that together make up the whole polyatomic lattice.

Let's create two monatomic lattices (`lat_a` and `lat_b`). `(1, 0)` and `(0, 1)` will be the primitive vectors and `(0, 0)` and `(0.5, 0.5)` the origins of the two lattices:

```
# Two monatomic lattices
primitive_vectors = [(1, 0), (0, 1)]
lat_a = kwant.lattice.Monatomic(primitive_vectors, offset=(0, 0))
lat_b = kwant.lattice.Monatomic(primitive_vectors, offset=(0.5, 0.5))
# lat1 is equivalent to kwant.lattice.square()

syst = kwant.Builder()

syst[lat_a(0, 0)] = 4
syst[lat_b(0, 0)] = 4

kwant.plot(syst)
```



We can also create a `Polyatomic` lattice with the same primitive vectors and two sites in the basis:

```
# One polyatomic lattice containing two sublattices
lat = kwant.lattice.Polyatomic([(1, 0), (0, 1)], [(0, 0), (0.5, 0.5)])
sub_a, sub_b = lat.sublattices
```

The two sublattices `sub_a` and `sub_b` are nothing else than `Monatomic` instances, and are equivalent to `lat_a` and `lat_b` that we created previously. The advantage of the second approach is that there is now a `Polyatomic` object that is aware of both of its sublattices, and we can do things like calculate neighboring sites, even between sublattices, which would not be possible with the two separate `Monatomic` lattices.

The *`kwant.lattice`* module also defines several convenience functions, such as *`square`* and *`honeycomb`*, for creating lattices of common types, without having to explicitly specify all of the lattice vectors and basis vectors.

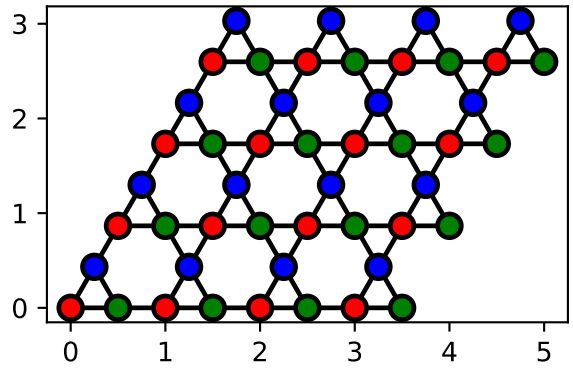
2.11.6 When plotting, how to color the different sublattices differently?

In the following example we shall use a kagome lattice, which has three sublattices.

```
lat = kwant.lattice.kagome()
syst = kwant.Builder()

a, b, c = lat.sublattices  # The kagome lattice has 3 sublattices
```

As we can see below, we create a new plotting function that assigns a color for each family, and a different size for the hoppings depending on the family of the two sites. Finally we add sites and hoppings to our system and plot it with the new function.

```
# Plot sites from different families in different colors
def family_color(site):
    if site.family == a:
        return 'red'
    if site.family == b:
        return 'green'
    else:
        return 'blue'

def plot_system(syst):
    kwant.plot(syst, site_lw=0.1, site_color=family_color)

## Add sites and hoppings.
for i in range(4):
    for j in range (4):
        syst[a(i, j)] = 4
        syst[b(i, j)] = 4
        syst[c(i, j)] = 4

syst[lat.neighbors()] = -1

## Plot the system.
plot_system(syst)
```



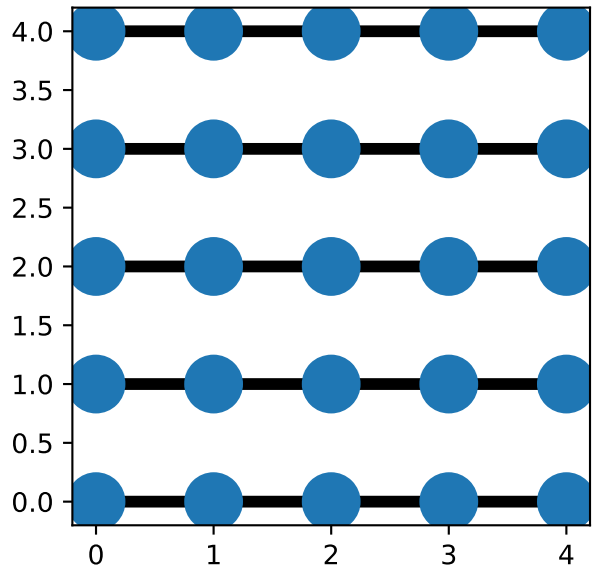
2.11.7 How to create many similar hoppings in one go?

This can be achieved with an instance of the class *kwant.builder.HoppingKind*. In fact, sites and hoppings are not the only possible keys when assigning values to a *Builder*. There exists a mechanism to *expand* more general keys into these simple keys.

A `HoppingKind`, the most comonly used general key, is a way of specifying all hoppings of a particular "kind", between two site families. For example `HoppingKind((1, 0), lat_a, lat_b)` represents all hoppings of the form `(lat_a(x + (1, 0)), lat_b(x))`, where `x` is a tag (here, a pair of integers).

The following example shows how this can be used:

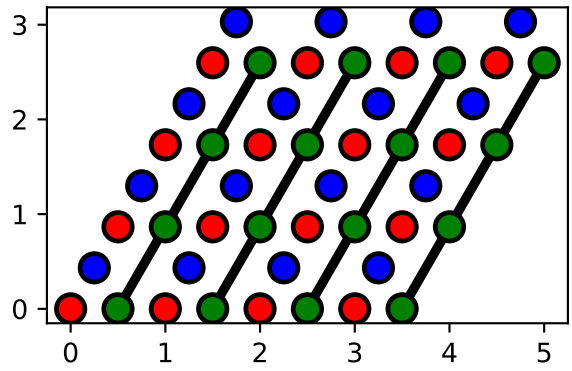
```
# Create hopping between neighbors with HoppingKind
a = 1
syst = kwant.Builder()
lat = kwant.lattice.square(a)
syst[ (lat(i, j) for i in range(5) for j in range(5)) ] = 4

syst[kwant.builder.HoppingKind((1, 0), lat)] = -1
kwant.plot(syst)
```



Note that `HoppingKind` only works with site families so you cannot use them directly with `Polyatomic` lattices; you have to explicitly specify the sublattices when creating a `HoppingKind`:

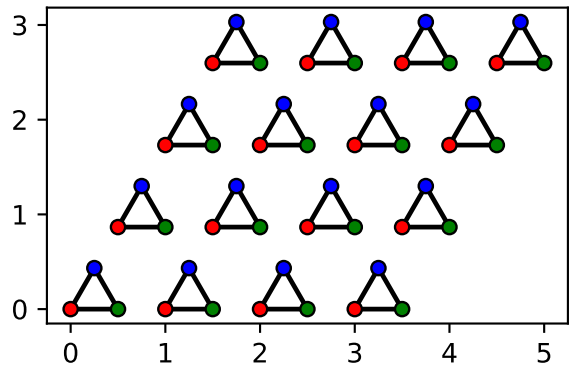
```
# equivalent to syst[kwant.builder.HoppingKind((0, 1), b)] = -1
syst[kwant.builder.HoppingKind((0, 1), b, b)] = -1
```

Here, we want the hoppings between the sites from sublattice b with a direction of (0,1) in the lattice coordinates.




```
syst[kwant.builder.HoppingKind((0, 0), a, b)] = -1
syst[kwant.builder.HoppingKind((0, 0), a, c)] = -1
syst[kwant.builder.HoppingKind((0, 0), c, b)] = -1
```

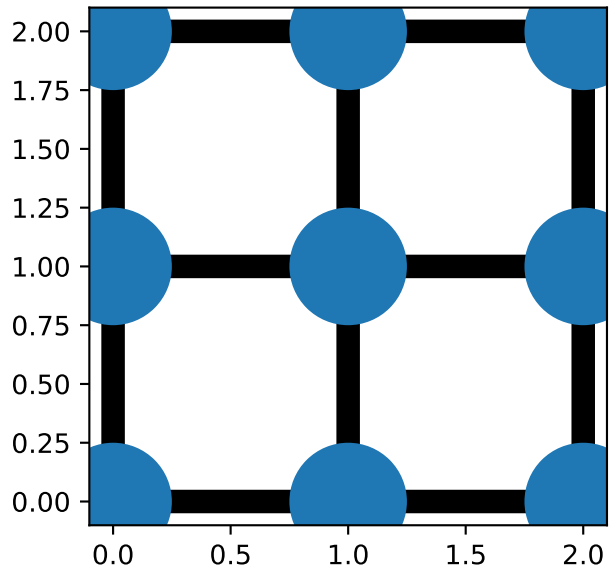
Here, we create hoppings between the sites of the same lattice coordinates but from different families.

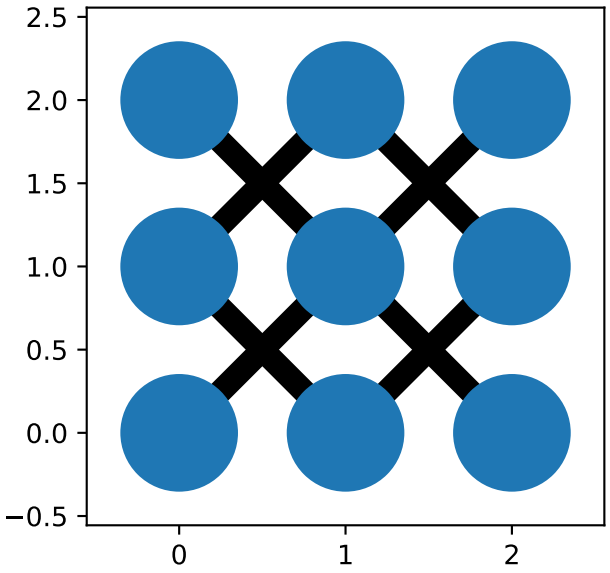


2.11.8 How to set the hoppings between adjacent sites?

`Polyatomic` and `Monatomic` lattices have a method *`neighbors`* that returns a list of `HoppingKind` instances that connect sites with their (n-nearest) neighors:

```
# Create hoppings with lat.neighbors()
syst = kwant.Builder()
lat = kwant.lattice.square()
syst[(lat(i, j) for i in range(3) for j in range(3))] = 4

syst[lat.neighbors()] = -1  # Equivalent to lat.neighbors(1)
kwant.plot(syst)

del syst[lat.neighbors()]  # Delete all nearest-neighbor hoppings
syst[lat.neighbors(2)] = -1

kwant.plot(syst)
```

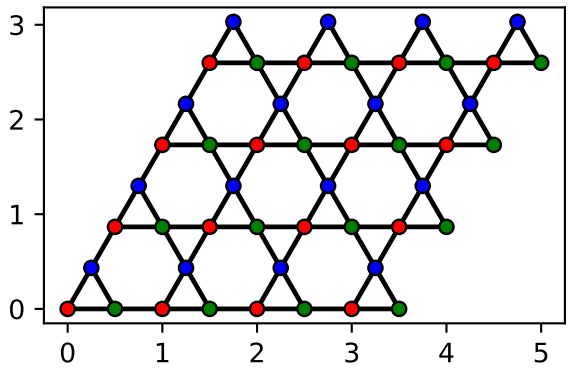




As we can see in the figure above, `lat.neighbors()` (on the left) returns the hoppings between the first nearest neighbors and `lat.neighbors(2)` (on the right) returns the hoppings between the second nearest neighbors.

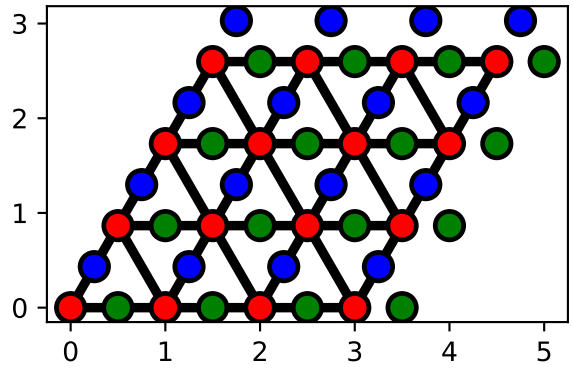
When using a `Polyatomic` lattice `neighbors()` knows about the different sublattices:

```
# Create the system
lat = kwant.lattice.kagome()
syst = kwant.Builder()
a, b, c = lat.sublattices  # The kagome lattice has 3 sublattices

for i in range(4):
    for j in range (4):
        syst[a(i, j)] = 4  # red
        syst[b(i, j)] = 4  # green
        syst[c(i, j)] = 4  # blue

syst[lat.neighbors()] = -1
```



However, if we use the `neighbors()` method of a single sublattice, we will only get the neighbors *on that sublattice*:

```
syst[a.neighbors()] = -1
```

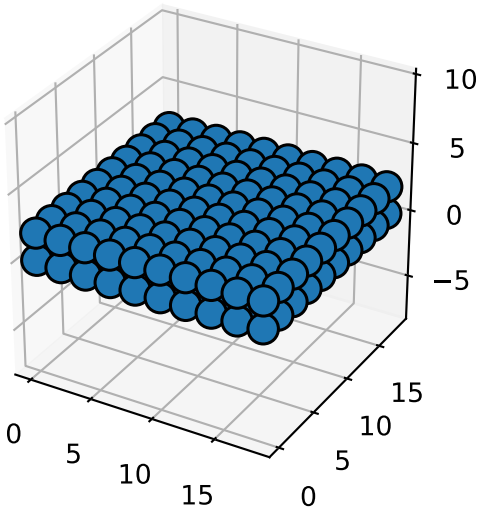


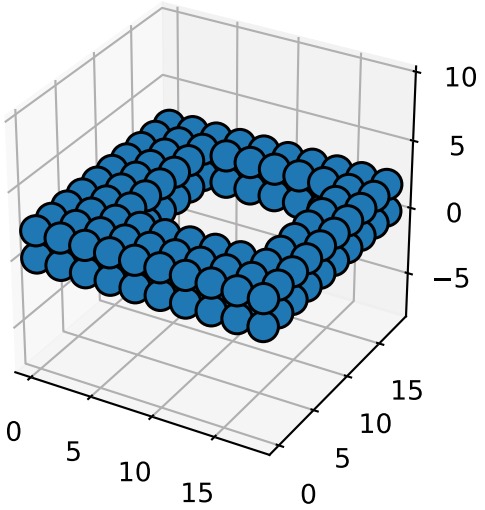
Note in the above that there are *only* hoppings between the red sites. This is an artifact of the visualisation: the blue and green sites just happen to lie in the path of the hoppings, but are not connected by them.

2.11.9 How to make a hole in a system?

To make a hole in the system, use `del syst[site]`, just like with any other mapping. In the following example we remove all sites inside some "hole" region:

```
# Define the lattice and the (empty) system
a = 2
lat = kwant.lattice.cubic(a)
syst = kwant.Builder()

L = 10
W = 10
H = 2

# Add sites to the system in a cuboid

syst[(lat(i, j, k) for i in range(L) for j in range(W) for k in range(H))] = 4
kwant.plot(syst)

# Delete sites to create a hole

def in_hole(site):
    x, y, z = site.pos / a - (L/2, W/2, H/2)  # position relative to centre
    return abs(x) < L / 4 and abs(y) < W / 4

for site in filter(in_hole, list(syst.sites())):
    del syst[site]

kwant.plot(syst)
```





`del syst[site]` also works after hoppings have been added to the system. If a site is deleted, then all the hoppings to/from that site are also deleted.

2.11.10 How to access a system's sites?

The ways of accessing system sites is slightly different depending on whether we are talking about a `Builder` or `System` (see *What is a system, and what is a builder?* if you do not know the difference).

We can access the sites of a `Builder` by using its *`sites`* method:

```
# Before finalizing the system

sites = list(builder.sites())  # sites() doe *not* return a list
```

The `sites()` method returns an *iterator* over the system sites, and in the above example we create a list from the contents of this iterator, which contains all the sites. At this stage the ordering of sites is not fixed, so if you add more sites to the `Builder` and call `sites()` again, the sites may well be returned in a different order.

After finalization, when we are dealing with a `System`, the sites themselves are stored in a list, which can be accessed via the `sites` attribute:

```
# After finalizing the system
syst = builder.finalized()
sites = syst.sites  # syst.sites is an actual list
```

The order of sites in a `System` is fixed, and also defines the ordering of the system Hamiltonian, system wavefunctions etc. (see *How does Kwant order components of an individual wavefunction?* for details).

`System` also contains the inverse mapping, `id_by_site` which gives us the index of a given site within the system:

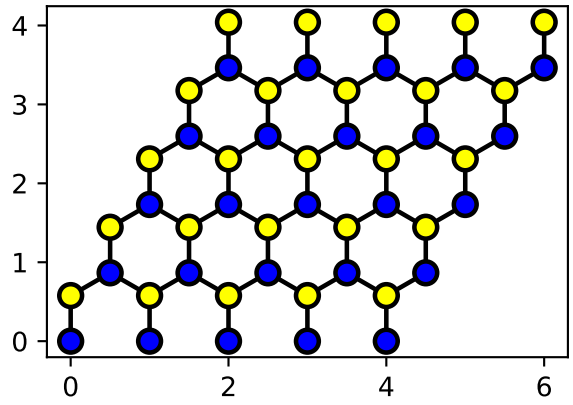
```
i = syst.id_by_site[lat(0, 2)]  # we want the id of the site lat(0, 2)
```

2.11.11 How to use different lattices for the scattering region and a lead?

Let us take the example of a system containing sites from a honeycomb lattice, which we want to connect to leads that contain sites from a square lattice.

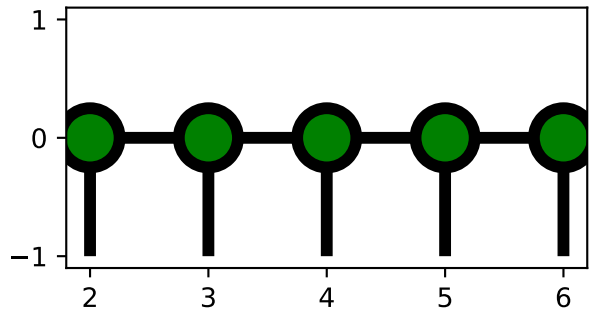
First we construct the central system:

```
# Define the scattering Region
L = 5
W = 5

lat = kwant.lattice.honeycomb()
subA, subB = lat.sublattices

syst = kwant.Builder()
syst[(subA(i, j) for i in range(L) for j in range(W))] = 4
syst[(subB(i, j) for i in range(L) for j in range(W))] = 4
syst[lat.neighbors()] = -1
```



and the lead:

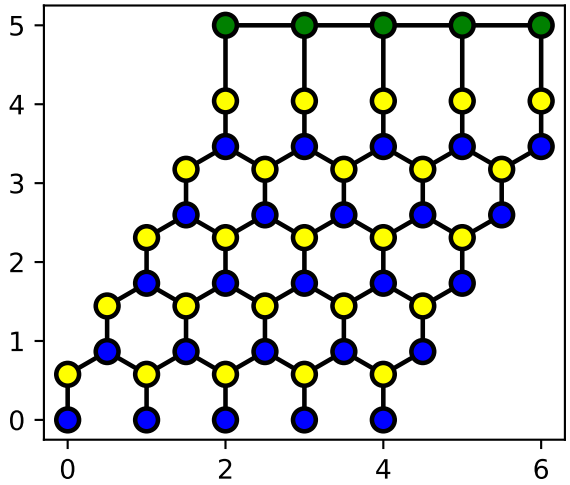
```
# Create a lead
lat_lead = kwant.lattice.square()
sym_lead1 = kwant.TranslationalSymmetry((0, 1))

lead1 = kwant.Builder(sym_lead1)
lead1[(lat_lead(i, 0) for i in range(2, 7))] = 4
lead1[lat_lead.neighbors()] = -1
```



We cannot simply use *attach_lead* to attach this lead to the system with the honeycomb lattice because Kwant does not know how sites from these two lattices should be connected.

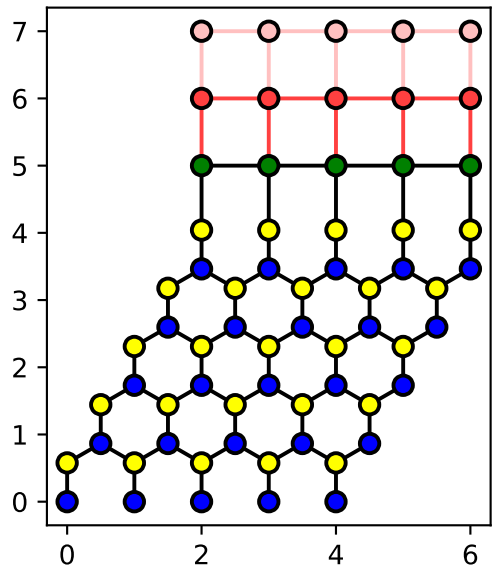
We must first add a layer of sites from the square lattice to the system and manually add the hoppings from these sites to the sites from the honeycomb lattice:

```
syst[(lat_lead(i, 5) for i in range(2, 7))] = 4
syst[lat_lead.neighbors()] = -1

# Manually attach sites from graphene to square lattice
syst[((lat_lead(i+2, 5), subB(i, 4)) for i in range(5))] = -1
```



`attach_lead()` will now be able to attach the lead:

```
syst.attach_lead(lead1)
```



2.11.12 How to cut a finite system out of a system with translational symmetries?

This can be achieved using the *fill* method to fill a `Builder` with a `Builder` with higher symmetry.

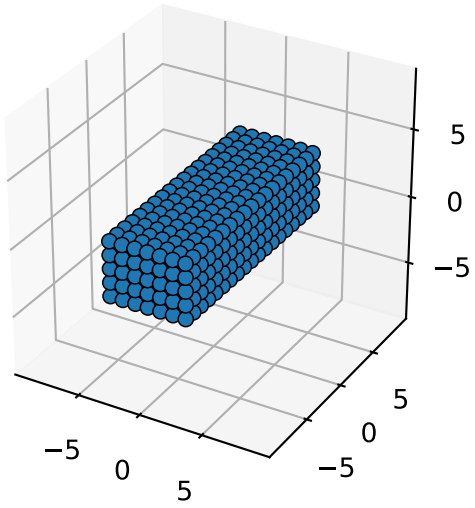
When using the `fill()` method, we need two systems: the template and the target. The template is a `Builder` with some translational symmetry that will be repeated in the desired shape to create the final system.

For example, say we want to create a simple model on a cubic lattice:

```
# Create 3d model.
cubic = kwant.lattice.cubic()
sym_3d = kwant.TranslationalSymmetry([1, 0, 0], [0, 1, 0], [0, 0, 1])
model = kwant.Builder(sym_3d)
model[cubic(0, 0, 0)] = 4
model[cubic.neighbors()] = -1
```

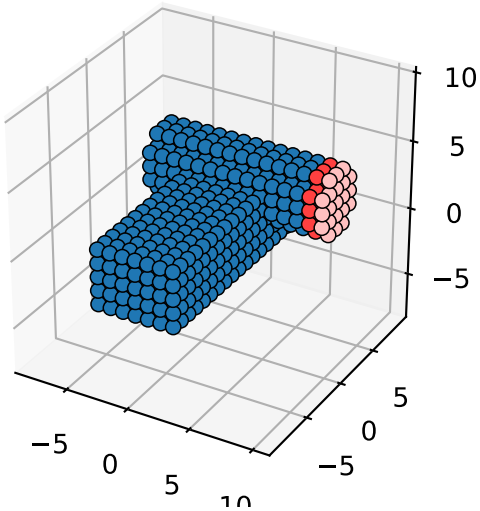
We have now created our "template" `Builder` which has 3 translational symmetries. Next we will fill a system with no translational symmetries with sites and hoppings from the template inside a cuboid:

```
# Build scattering region (white).
def cuboid_shape(site):
    x, y, z = abs(site.pos)
    return x < 4 and y < 10 and z < 3

cuboid = kwant.Builder()
cuboid.fill(model, cuboid_shape, (0, 0, 0));
```



We can then use the original template to create a lead, which has 1 translational symmetry. We can then use this lead as a template to fill another section of the system with a cylinder of sites and hoppings:

```
# Build electrode (black).
def electrode_shape(site):
    x, y, z = site.pos - (0, 5, 2)
    return y**2 + z**2 < 2.3**2

electrode = kwant.Builder(kwant.TranslationalSymmetry([1, 0, 0]))
electrode.fill(model, electrode_shape, (0, 5, 2))  # lead

# Scattering region
cuboid.fill(electrode, lambda s: abs(s.pos[0]) < 7, (0, 5, 4))

cuboid.attach_lead(electrode)
```

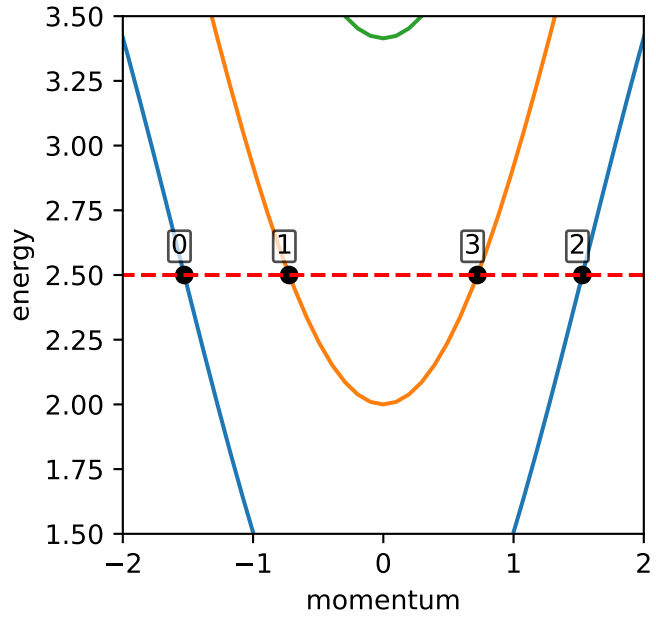


2.11.13 How does Kwant order the propagating modes of a lead?

A very useful feature of kwant is to calculate the transverse wavefunctions of propagating modes in a system with 1 translational symmetry. This can be achieved with the *modes* method, which returns a pair of objects, the first of which contains the propagating modes of the system in a *PropagatingModes* object:

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

lead = kwant.Builder(kwant.TranslationalSymmetry((-1, 0)))
lead[(lat(0, i) for i in range(3))] = 4
lead[lat.neighbors()] = -1

flead = lead.finalized()

E = 2.5
prop_modes, _ = flead.modes(energy=E)
```

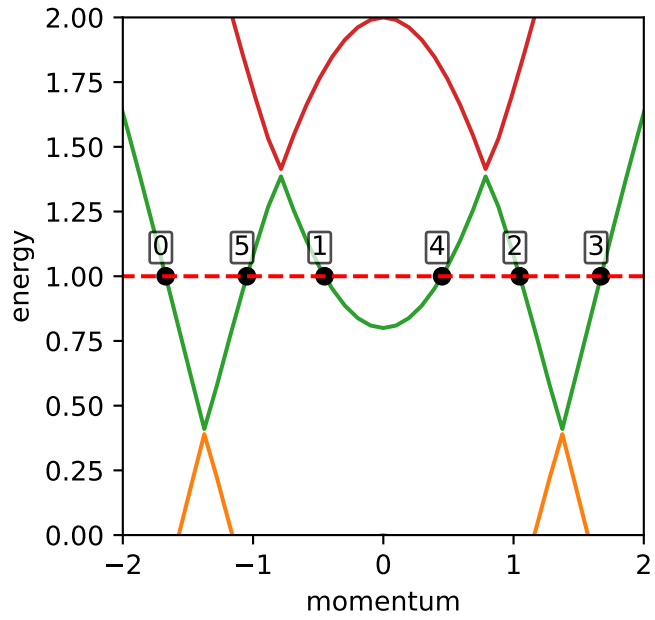
`PropagatingModes` contains the wavefunctions, velocities and momenta of the modes at the requested energy (2.5 in this example). In order to understand the order in which these quantities are returned it is often useful to look at the a section of the band structure for the system in question:



On the above band structure we have labelled the 4 modes in the order that they appear in the output of `modes()` at energy 2.5. Note that the modes are sorted in the following way:

- First all the modes with negative velocity, then all the modes with positive velocity
- Negative velocity modes are ordered by *increasing* momentum
- Positive velocity modes are ordered by *decreasing* momentum

For more complicated systems and band structures this can lead to some unintuitive orderings:



2.11.14 How does Kwant order scattering states?

Scattering states calculated using *wave_function* are returned in the same order as the "incoming" modes of *modes*. Kwant considers that the translational symmetry of a lead points "towards infinity" (*not* towards the system) which means that the incoming modes are those that have *negative* velocities.

This means that for a lead attached on the left of a scattering region (with symmetry vector $(-1, 0)$, for example), the positive k direction (when inspecting the lead's band structure) actually corresponds to the *negative* x direction.

2.11.15 How does Kwant order components of an individual wavefunction?

In *How to access a system's sites?* we saw that the sites of a finalized system are available as a list through the `sites` attribute, and that one can look up the index of a site with the `id_by_site` attribute.

When all the site families present in a system have only 1 degree of freedom per site (i.e. the all the onsites are scalars) then the index into a wavefunction defined over the system is exactly the site index:

```
lat = kwant.lattice.square(norbs=1)
syst = make_system(lat)
scattering_states = kwant.wave_function(syst, energy=1)
wf = scattering_states(0)[0]  # scattering state from lead 0 incoming in mode 0

idx = syst.id_by_site[lat(0, 0)]  # look up index of site

print('wavefunction on lat(0, 0): ', wf[idx])
```

```
wavefunction on lat(0, 0):  (0.12535832151852414-0.36344642498268875j)
```

We see that the wavefunction on a single site is a single complex number, as expected.

If a site family have more than 1 degree of freedom per site (e.g. spin or particle-hole) then Kwant places degrees of freedom on the same site adjacent to one another. In the case where all site families in the system have the *same* number of degrees of freedom, we can then simply *reshape* the wavefunction into a matrix, where the row number indexes the site, and the column number indexes the degree of freedom on that site:

```
lat = kwant.lattice.square(norbs=2)
syst = make_system(lat)
```

(continues on next page)

(continued from previous page)

```
scattering_states = kwant.wave_function(syst, energy=1)
wf = scattering_states(0)[0]  # scattering state from lead 0 incoming in mode 0

idx = syst.id_by_site[lat(0, 0)]  # look up index of site

# Group consecutive degrees of freedom from 'wf' together; these correspond
# to degrees of freedom on the same site.
wf = wf.reshape(-1, 2)

print('wavefunction on lat(0, 0): ', wf[idx])
```

```
wavefunction on lat(0, 0):  [-0.11048367+0.35420532j -0.06211986-0.07925331j]
```

We see that the wavefunction on a single site is a *vector* of 2 complex numbers, as we expect.

If there are different site families present in the system that have *different* numbers of orbitals per site, then the situation becomes much more involved, because we cannot simply "reshape" the wavefunction like we did in the preceding example.

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__`. This string respects PEP 440 and has the following format

- Released version: '1.3.0', '1.3.1', etc.
- Alpha version: '1.2.0a0', '1.2.0a1', etc.
- Beta version: '1.1.0b0', '1.1.0b1', etc.
- Development version (derived from `git describe --first-parent --dirty`): '1.3.2.dev27+gdecf6893', '1.1.1.dev10+gabcd012.dirty', etc.
- Development version with incomplete information: 'unknown', 'unknown+g0123abc', etc.

KwantDeprecationWarning	Class of warnings about a deprecated feature of Kwant.
UserCodeError	Class for errors that occur in user-provided code.

kwant.KwantDeprecationWarning

exception kwant.KwantDeprecationWarning

Bases: `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.

kwant.UserCodeError

exception kwant.UserCodeError

Bases: `Exception`

Class for errors that occur in user-provided code.

Usually users will define value functions that Kwant calls in order to evaluate the Hamiltonian. If one of these function raises an exception then it is caught and this error is raised in its place. This makes it clear that the error is from the user's code (and not a bug in Kwant) and also makes it possible for any libraries that wrap Kwant to detect when a user's function causes an error.

3.1.2 From `kwant.builder`

Builder([symmetry, conservation_law, ...])	A tight binding system defined on a graph.
HoppingKind	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 scattering region.

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

3.2.1 Types

Builder([symmetry, conservation_law, ...])	A tight binding system defined on a graph.
Site	A site, member of a *SiteFamily*.
HoppingKind	A pattern for matching hoppings.
SimpleSiteFamily([name, norbs])	A site family used as an example and for testing.
BuilderLead(builder, interface[, padding])	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, parameters)	A general lead defined by its modes wave functions.
FiniteSystem(builder)	Finalized *Builder* with leads.
InfiniteSystem(builder[, interface_order])	Finalized infinite system, extracted from a *Builder*.

kwant.builder.Builder

class `kwant.builder.Builder`(*symmetry=None*, ***, *conservation_law=None*, *time_reversal=None*, *particle_hole=None*, *chiral=None*)

Bases: `object`

A tight binding system defined on a graph.

An alias exists for this common name: `kwant.Builder`.

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 spatial symmetry of the system.

conservation_law : 2D array, dictionary, function, or *None*

An onsite operator with integer eigenvalues that commutes with the Hamiltonian. The ascending order of eigenvalues corresponds to the selected ordering of the Hamiltonian subblocks. If a dict is given, it maps from site families to such matrices. If a function is given it must take the same arguments as the onsite Hamiltonian functions of the system and return the onsite matrix.

time_reversal : scalar, 2D array, dictionary, function, or *None*

The unitary part of the onsite time-reversal symmetry operator. Same format as that of *conservation_law*.

particle_hole : scalar, 2D array, dictionary, function, or *None*

The unitary part of the onsite particle-hole symmetry operator. Same format as that of *conservation_law*.

chiral : 2D array, dictionary, function or *None*

The unitary part of the onsite chiral symmetry operator. Same format as that of *conservation_law*.

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, `syst[general_key] = value` is equivalent to

```
for simple_key in syst.expand(general_key):
    syst[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: 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 the list of leads accessbile as the *leads* attribute.

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.

This method first adds sites from 'lead_builder' until the interface where the lead will attach is "smooth". Then it appends the 'lead_builder' and the interface sites as a *BuilderLead* to the 'leads' of this builder.

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 raises an error, there is no guarantee that the system stayed unaltered.

The system must "interrupt" the lead that is being attached. This means that for each site in the lead that has a hopping to a neighbnoring unit cell there must be at least one site in the system that is an image of the lead site under the action of the lead's translational symmetry. In order to interrupt the lead, the system must contain sites from the same site family as the sites in the lead. Below are three examples of leads being attached to systems:

```
Successful          Successful          Unsuccessful
----------          ----------          ------------
Lead   System        Lead   System        Lead  System
x-  …      o         x   …                x-  …
|          |         |                    |
x-  …    o-o         x-  …    o-o         x-  …  o-o
|        | |         |        | |         |      | |
x-  …  o-o-o         x-  …  o-o-o         x-  …  o-o
```

The second case succeeds, as even though the top site has no image in the system, because the top site has no hoppings to sites in other unit cells.

Sites may be added to the system when the lead is attached, so that the interface to the lead is "smooth". Below we show the system after having attached a lead. The 'x' symbols in the system indicate the added sites:

```
Lead   System        Lead   System
x-  …  x-x-o         x   …  x
|      | | |         |      |
x-  …  x-o-o         x-  …  x-o-o
|      | | |         |      | | |
x-  …  o-o-o         x-  …  o-o-o
```

static cell_hamiltonian(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

closest(*pos*)
Return the site that is closest to the given position.

This function takes into account the symmetry of the builder. It is assumed that the symmetry is a translational symmetry.

This function executes in a time proportional to the number of sites, so it is not efficient for large builders. It is especially slow for builders with a symmetry, but such systems often contain only a limited number of sites.

dangling()
Return an iterator over all dangling sites.

Sites are considered as dangling when less than two hoppings lead to them.

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

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

Sites are considered as dangling when less than two hoppings lead to them.

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. `syst[key] = value` or `del syst[key]`).

`fill`(*template*, *shape*, *start*, *, *max_sites=10000000*)
Populate builder using another one as a template.

Starting from one or multiple sites, traverse the graph of the template builder and copy sites and hoppings to the target builder. The traversal stops at sites that are already present in the target and on sites that are not inside the provided shape.

This function takes into account translational symmetry. As such, typically the template will have a higher symmetry than the target.

Newly added sites are connected by hoppings to sites that were already present. This facilitates construction of a system by a series of calls to 'fill'.

Parameters **template** : *Builder* instance

The builder used as the template. The symmetry of the target builder must be a subgroup of the symmetry of the template.

shape : callable

A boolean function of site returning whether the site should be added to the target builder or not. The shape must be compatible with the symmetry of the target builder.

start : *Site* instance or iterable thereof or iterable of numbers

The site(s) at which the the flood-fill starts. If start is an iterable of numbers, the starting site will be `template.closest(start)`.

max_sites : positive number

The maximal number of sites that may be added before `RuntimeError` is raised. Used to prevent using up all memory.

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

Raises **ValueError**

If the symmetry of the target isn't a subgroup of the template symmetry.

RuntimeError

If more than *max_sites* sites are to be added. The target builder will be left in an unusable state.

Notes

This function uses a flood-fill algorithm. If the template builder consists of disconnected parts, the fill will stop at their boundaries.

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

Returns **finalized_system** : *kwant.builder.FiniteSystem*

If there is no symmetry.

finalized_system : *kwant.builder.InfiniteSystem*

If a symmetry is present.

Notes

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

Upon finalization, it is implicitly assumed that **every** function assigned as a value to a builder with a symmetry possesses the same symmetry.

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.

static hamiltonian(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

static hamiltonian_submatrix(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

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).

static inter_cell_hopping(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

static modes(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

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

Technical note: This method respects the symmetry of the builder, i.e. the returned sites are really connected to the given site (and not to its image in the fundamental domain).

static precalculated(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

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.

static selfenergy(**args*, ***kwargs*)
You need a finalized system; Use Builder.finalized() first.

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).

substituted(***subs*)
Return a copy of this Builder with modified parameter names.

update(*other*)
Update builder from *other*.

All sites and hoppings of *other*, together with their values, are written to *self*, overwriting already existing sites and hoppings. The leads of *other* are appended to the leads of the system being updated.

This method requires that both builders share the same symmetry.

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`(*$self*, *value*, */*)
Return number of occurrences of value.

`index`(*$self*, *value*, *start=0*, *stop=sys.maxsize*, */*)
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.

This relies on `family` having a `pos` method (see *SiteFamily*).

`tag`
The tag of the site.

kwant.builder.HoppingKind

class kwant.builder.HoppingKind
Bases: tuple

A pattern for matching hoppings.

An alias exists for this common name: `kwant.HoppingKind`.

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)
syst[kind(syst)] = 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)]]
syst[kinds] = 1
```

Methods

count(*$self*, *value*, */*)
Return number of occurrences of value.

index(*$self*, *value*, *start=0*, *stop=sys.maxsize*, */*)
Return first index of value.

Raises ValueError if the value is not present.

Attributes

delta
The difference between the tags of the hopping's sites

family_a
The family of the first site in the hopping

family_b
The family of the second site in the hopping

kwant.builder.SimpleSiteFamily

class kwant.builder.**SimpleSiteFamily**(*name=None*, *norbs=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*)
Return a normalized version of the tag.

Raises TypeError or ValueError if the tag is not acceptable.

kwant.builder.BuilderLead

`class kwant.builder.BuilderLead`(*builder*, *interface*, *padding=None*)
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).

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

Attributes

builder	(*Builder*) The tight-binding system of a lead.
interface	(list of *Site* instances) A sorted list of interface sites.
padding	(list of *Site* instances) A sorted list of sites that originate from the lead, have the same onsite Hamiltonian, and are connected by the same hoppings as the lead sites.

Methods

`finalized`()
Return a *kwant.builder.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*, *parameters*)
Bases: *kwant.builder.Lead*

A general lead defined by its self energy.

Parameters **selfenergy_func** : function

Has the same signature as *selfenergy* (without the `self` parameter) and returns the self energy matrix for the interface sites.

interface : sequence of *Site* instances

parameters : sequence of strings

The parameters on which the lead depends.

Methods

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

`selfenergy`(*energy*, *args=()*, ***, *params=None*)

kwant.builder.ModesLead

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

A general lead defined by its modes wave functions.

Parameters **modes_func** : function

Has the same signature as *modes* (without the `self` parameter) and returns the modes of the lead as a tuple of *PropagatingModes* and *StabilizedModes*.

interface :

sequence of *Site* instances

parameters : sequence of strings

The parameters on which the lead depends.

Methods

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

`modes`(*energy*, *args=()*, ***, *params=None*)

`selfenergy`(*energy*, *args=()*, ***, *params=None*)

kwant.builder.FiniteSystem

`class kwant.builder.FiniteSystem`(*builder*)
Bases: `kwant.builder._FinalizedBuilderMixin`, *kwant.system.FiniteSystem*

Finalized *Builder* with leads.

Usable as input for the solvers in *kwant.solvers*.

Attributes

sites	(sequence) `sites[i]` is the *Site* instance that corresponds to the integer-labeled site `i` of the low-level system. The sites are ordered first by their family and then by their tag.
id_by_site	(dict) The inverse of `sites`; maps from `i` to `sites[i]`.

Methods

`discrete_symmetry`(*args=()*, ***, *params=None*)

`hamiltonian`(*i*, *j*, **args*, *params=None*)

`hamiltonian_submatrix`(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*, ***, *params=None*)
Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method. Mutually exclusive with 'params'.

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`.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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.

`pos`(*i*)

`precalculate`(*energy=0*, *args=()*, *leads=None*, *what='modes'*, ***, *params=None*)

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. Deprecated in favor of 'params' (and mutually exclusive with it).

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'.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns **syst** : 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.

`validate_symmetries`(*args=()*, ***, *params=None*)
Check that the Hamiltonian satisfies discrete symmetries.

Applies *`validate`* to the Hamiltonian, see its documentation for details on the return format.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

kwant.builder.InfiniteSystem

`class kwant.builder.InfiniteSystem`(*builder*, *interface_order=None*)
Bases: `kwant.builder._FinalizedBuilderMixin`, *`kwant.system.InfiniteSystem`*

Finalized infinite system, extracted from a *`Builder`*.

Notes

In infinite systems `sites` consists of 3 parts: sites in the fundamental domain (FD) with hoppings to neighboring cells, sites in the FD with no hoppings to neighboring cells, and sites in FD+1 attached to the FD by hoppings. Each of these three subsequences is individually sorted.

Attributes

sites	(sequence) `sites[i]` is the *`Site`* instance that corresponds to the integer-labeled site `i` of the low-level system.
id_by_site	(dict) The inverse of `sites`; maps from `i` to `sites[i]`.

Finalize a builder instance which has to have exactly a single symmetry direction.

If interface_order is not set, the order of the interface sites in the finalized system will be arbitrary. If interface_order is set to a sequence of interface sites, this order will be kept.

Methods

`cell_hamiltonian`(*args=()*, *sparse=False*, ***, *params=None*)
Hamiltonian of a single cell of the infinite system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`discrete_symmetry`(*args=()*, ***, *params=None*)

`hamiltonian`(*i*, *j*, **args*, *params=None*)
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.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

hamiltonian_submatrix(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*, ***, *params=None*)

Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method. Mutually exclusive with 'params'.

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`.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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*, ***, *params=None*)

Hopping Hamiltonian between two cells of the infinite system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

modes(*energy=0*, *args=()*, ***, *params=None*)

Return mode decomposition of the lead

See documentation of *PropagatingModes* and *StabilizedModes* for the return format details.

The wave functions of the returned modes are defined over the *unit cell* of the system, which corresponds to the degrees of freedom on the first `cell_sites` sites of the system (recall that infinite systems store first the sites in the unit cell, then connected sites in the neighboring unit cell).

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

pos(*i*)

selfenergy(*energy=0*, *args=()*, ***, *params=None*)

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))`.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

validate_symmetries(*args=()*, ***, *params=None*)
Check that the Hamiltonian satisfies discrete symmetries.

Returns *validate* applied to the onsite matrix and the hopping. See its documentation for details on the return format.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

3.2.2 Abstract base classes

SiteFamily(canonical_repr, name, norbs)	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*, *norbs*)
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. `norbs` defines the number of orbitals on sites associated with this site family; it may be *None*, in which case the number of orbitals is not specified.

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.

If the `norbs` of a site family are provided, and sites of this family are used to populate a *Builder*, then the associated Hamiltonian values must have the correct shape. That is, if a site family has `norbs = 2`, then any on-site terms for sites belonging to this family should be 2x2 matrices. Similarly, any hoppings to/from sites belonging to this family must have a matrix structure where there are two rows/columns. This condition applies equally to Hamiltonian values that are given by functions. If this condition is not satisfied, an error will be raised.

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.

The type of the domain objects as handled by the methods of this class is not specified. The only requirement is that it must support the unary minus operation. The reference implementation of *to_fd()* is hence *self.act(-self.which(a), a, b)*.

Methods

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

has_subgroup(*other*)
Test whether *self* has the subgroup *other*…

or, in other words, whether *other* is a subgroup of *self*. The reason why this is the abstract method (and not *is_subgroup*) is that in general it's not possible for a subgroup to know its supergroups.

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

subgroup(**generators*)
Return the subgroup generated by a sequence of group elements.

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.

Equivalent to *self.act(-self.which(a), a, b)*.

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*, i.e. an instance of *kwant.system.InfiniteSystem*. Typically it will be a finalized builder: *kwant.builder.InfiniteSystem*.

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, norbs])	Create a Bravais lattice of any dimensionality, with any number of sites.
Monatomic(prim_vecs[, offset, name, norbs])	A Bravais lattice with a single site in the basis.
Polyatomic(prim_vecs, basis[, name, norbs])	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.

An alias exists for this common name: kwant.TranslationalSymmetry.

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*)
Act with a symmetry group element on a site or hopping.

`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.

`has_subgroup`(*other*)
Test whether *self* has the subgroup *other*…

or, in other words, whether *other* is a subgroup of *self*. The reason why this is the abstract method (and not *is_subgroup*) is that in general it's not possible for a subgroup to know its supergroups.

`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.

`subgroup`(**generators*)
Return the subgroup generated by a sequence of group elements.

Parameters ***generators: sequence of int**

Each generator must have length `self.num_directions`.

`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.

Equivalent to *self.act(-self.which(a), a, b)*.

`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.lattice.general

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

norbs : int or sequence of ints, optional

The number of orbitals per site on the lattice, or a sequence of the number of orbitals of sites on each of the sublattices.

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=''*, *norbs=None*)
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*, *group_by_length=False*, *rtol=1e-09*)
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. Note that the zeroth neighbor is the site itself or any other sites with the same position.

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*)
Return a normalized version of the tag.

Raises TypeError or ValueError if the tag is not acceptable.

`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 *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()
>>> syst = kwant.Builder()
>>> syst[lat.shape(circle, (0, 0))] = 0
>>> syst[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 *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=”*, *norbs=None*)
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, optional

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.

norbs : int or sequence of ints, optional

The number of orbitals per site on the lattice, or a sequence of the number of orbitals of sites on each of the sublattices.

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. Note that the zeroth neighbor is the site itself or any other sites with the same position.

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 *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()
>>> syst = kwant.Builder()
>>> syst[lat.shape(circle, (0, 0))] = 0
>>> syst[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 *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, norbs])	Make a one-dimensional lattice.
square([a, name, norbs])	Make a square lattice.
cubic([a, name, norbs])	Make a cubic lattice.
triangular([a, name, norbs])	Make a triangular lattice.
honeycomb([a, name, norbs])	Make a honeycomb lattice.
kagome([a, name, norbs])	Make a kagome lattice.

kwant.lattice.chain

kwant.lattice.**chain**(*a=1*, *name=''*, *norbs=None*)
Make a one-dimensional lattice.

kwant.lattice.square

kwant.lattice.**square**(*a=1*, *name=''*, *norbs=None*)
Make a square lattice.

kwant.lattice.cubic

kwant.lattice.**cubic**(*a=1*, *name=''*, *norbs=None*)
Make a cubic lattice.

kwant.lattice.triangular

kwant.lattice.**triangular**(*a=1*, *name=''*, *norbs=None*)
Make a triangular lattice.

kwant.lattice.honeycomb

kwant.lattice.**honeycomb**(*a=1*, *name=''*, *norbs=None*)
Make a honeycomb lattice.

kwant.lattice.kagome

kwant.lattice.**kagome**(*a=1*, *name=''*, *norbs=None*)
Make a kagome lattice.

3.4 kwant.plotter – Plotting of systems

3.4.1 Plotting routines

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.
density(syst, density[, relwidth])	Show an interpolated density defined on the sites of a system.
current(syst, current[, relwidth])	Show an interpolated current defined for the hoppings of a system.
bands(sys[, args, momenta, file, show, dpi, ...])	Plot band structure of a translationally invariant 1D system.
spectrum(syst, x[, y, params, mask, file, ...])	Plot the spectrum of a Hamiltonian as a function of 1 or 2 parameters

Continued on next page

Table 10 – continued from previous page

streamplot(field, box[, cmap, bgcolor, ...])	Draw streamlines of a flow field in Kwant style
scalarplot(field, box[, cmap, colorbar, ...])	Draw a scalar field in Kwant style

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.

An alias exists for this common name: `kwant.plot`.

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. By default sites are colored by their site family, using the current matplotlib color cycle.

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*, *background='#e0e0e0'*)

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.

This function is similar to *density*, but is more suited to the case where you want site-level resolution of the quantity that you are plotting. If your system has many sites you may get more appealing plots by using *density*.

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.

background : matplotlib color spec

Areas without sites are filled with this color.

Returns **fig** : matplotlib figure

A figure with the output if *ax* is not set, else None.

See also:

kwant.plotter.density

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.density

`kwant.plotter.density`(*syst*, *density*, *relwidth=0.05*, ***kwargs*)

Show an interpolated density defined on the sites of a system.

The system sites, together with a scalar per site defines a "discrete" density field that is non-zero only on the sites.

To make this scalar field easier to visualize and interpret at different length scales, it is smoothed by convoluting it with the bell-shaped bump function `f(r) = max(1 - (2*r / width)**2, 0)**2`. The bump width is determined by the `relwidth` parameter.

This routine samples the smoothed field on a regular (square or cubic) grid and displays it using matplotlib's imshow.

This function is similar to *map*, but generally gives more appealing visual results when used on systems with many sites. If you want site-level resolution you may be better off using *map*.

This is a convenience function that is equivalent to `scalarplot(*interpolate_density(syst, density, relwidth), **kwargs)`. The longer form makes it possible to tweak additional options of *interpolate_density*.

Parameters **syst** : *kwant.system.FiniteSystem*

The system for which to plot `density`.

density : sequence of float

Sequence of values defining density on each site of the system. Ordered in the same way as `syst.sites`. This typically will be the result of evaluating a *Density* operator.

relwidth : float or *None*

Relative width of the bumps used to smooth the field, as a fraction of the length of the longest side of the bounding box.

****kwargs** : various

Keyword args to be passed verbatim to *scalarplot*.

Returns **fig** : matplotlib figure

A figure with the output if `ax` is not set, else None.

See also:

kwant.plotter.current, *kwant.plotter.map*

kwant.plotter.current

`kwant.plotter.current`(*syst*, *current*, *relwidth=0.05*, ***kwargs*)

Show an interpolated current defined for the hoppings of a system.

The system graph together with current intensities defines a "discrete" current density field where the current density is non-zero only on the straight lines that connect sites that are coupled by a hopping term.

To make this scalar field easier to visualize and interpret at different length scales, it is smoothed by convoluting it with the bell-shaped bump function `f(r) = max(1 - (2*r / width)**2, 0)**2`. The bump width is determined by the `relwidth` parameter.

This routine samples the smoothed field on a regular (square or cubic) grid and displays it using an enhanced variant of matplotlib's streamplot.

This is a convenience function that is equivalent to `streamplot(*interpolate_current(syst, current, relwidth), **kwargs)`. The longer form makes it possible to tweak additional options of *interpolate_current*.

Parameters **syst** : *kwant.system.FiniteSystem*

The system for which to plot the `current`.

current : sequence of float

Sequence of values defining currents on each hopping of the system. Ordered in the same way as `syst.graph`. This typically will be the result of evaluating a *Current* operator.

relwidth : float or *None*

Relative width of the bumps used to smooth the field, as a fraction of the length of the longest side of the bounding box.

****kwargs** : various

Keyword args to be passed verbatim to *kwant.plotter.streamplot*.

Returns **fig** : matplotlib figure

A figure with the output if `ax` is not set, else None.

See also:

kwant.plotter.density

kwant.plotter.bands

`kwant.plotter.bands`(*sys*, *args=()*, *momenta=65*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*, ***, *params=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. Deprecated in favor of 'params' (and mutually exclusive with it).

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.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns **fig** : matplotlib figure

A figure with the output if *ax* is not set, else None.

Notes

See *Bands* for the calculation of dispersion without plotting.

kwant.plotter.spectrum

`kwant.plotter.spectrum`(*syst*, *x*, *y=None*, *params=None*, *mask=None*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*)

Plot the spectrum of a Hamiltonian as a function of 1 or 2 parameters

Parameters **syst** : *kwant.system.FiniteSystem* or callable

If a function, then it must take named parameters and return the Hamiltonian as a dense matrix.

x : pair `(name, values)`

Parameter to `ham` that will be varied. Consists of the parameter name, and a sequence of parameter values.

y : pair `(name, values)`, optional

Used for 3D plots (same as `x`). If provided, then the cartesian product of the `x` values and these values will be used as a grid over which to evaluate the spectrum.

params : dict, optional

The rest of the parameters to `ham`, which will be kept constant.

mask : callable, optional

Takes the parameters specified by `x` and `y` and returns True if the spectrum should not be calculated for the given parameter values.

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.

kwant.plotter.streamplot

kwant.plotter.**streamplot**(*field*, *box*, *cmap=None*, *bgcolor=None*, *linecolor='k'*, *max_linewidth=3*, *min_linewidth=1*, *density=0.2222222222222222*, *colorbar=True*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*, *vmax=None*)

Draw streamlines of a flow field in Kwant style

Solid colored streamlines are drawn, superimposed on a color plot of the flow speed that may be disabled by setting *bgcolor*. The width of the streamlines is proportional to the flow speed. Lines that would be thinner than *min_linewidth* are blended in a perceptually correct way into the background color in order to create the illusion of arbitrarily thin lines. (This is done because some plot backends like PDF do not support lines of arbitrarily thin width.)

Internally, this routine uses matplotlib's streamplot.

Parameters **field** : 3d arraylike of float

2d array of 2d vectors.

box : 2-sequence of 2-sequences of float

the extents of *field*: ((x0, x1), (y0, y1))

cmap : colormap, optional

Colormap for the background color plot. When not set the colormap "kwant_red" is used by default, unless *bgcolor* is set.

bgcolor : color definition, optional

The solid color of the background. Mutually exclusive with *cmap*.

linecolor : color definition

Color of the flow lines.

max_linewidth : float

Width of lines at maximum flow speed.

min_linewidth : float

Minimum width of lines before blending into the background color begins.

density : float

Number of flow lines per point of the field. The default value of 2/9 is chosen to show two lines per default width of the interpolation bump of *interpolate_current*.

colorbar : bool

Whether to show a colorbar if a 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.

vmax : float or *None*

The upper saturation limit for the colormap; flows higher than this will saturate. Note that there is no corresponding vmin option, vmin being fixed at zero.

Returns fig : matplotlib figure

A figure with the output if *ax* is not set, else None.

kwant.plotter.scalarplot

`kwant.plotter.scalarplot`(*field*, *box*, *cmap=None*, *colorbar=True*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*, *vmin=None*, *vmax=None*, *background='#e0e0e0'*)

Draw a scalar field in Kwant style

Internally, this routine uses matplotlib's imshow.

Parameters field : 2d arraylike of float

2d scalar field to plot.

box : pair of pair of float

the realspace extents of `field`: ((x0, x1), (y0, y1))

cmap : colormap, optional

Colormap for the background color plot. When not set the colormap "kwant_red" is used by default.

colorbar : bool, default: True

Whether to show a colorbar if a colormap is used. Ignored if *ax* is provided.

file : string or file object, optional

The output file. If not provided, output will be shown instead.

show : bool, default: True

Whether `matplotlib.pyplot.show()` is to be called, and the output is to be shown immediately.

dpi : float, optional

Number of pixels per inch. If not set the `matplotlib` default is used.

fig_size : tuple, optional

Figure size `(width, height)` in inches. If not set, the default `matplotlib` value is used.

ax : `matplotlib.axes.Axes` instance, optional

If `ax` is provided, 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.

vmin, vmax : float, optional

The lower/upper saturation limit for the colormap.

background : matplotlib color spec

Areas outside the system are filled with this color.

Returns fig : matplotlib figure

A figure with the output if `ax` is not set, else None.

3.4.2 Helper functions

interpolate_current(syst, current[, ...])	Interpolate currents in a system onto a regular grid.
interpolate_density(syst, density[, ...])	Interpolate density in a system onto a regular grid.
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.interpolate_current

`kwant.plotter.interpolate_current`(*syst*, *current*, *relwidth=None*, *abswidth=None*, *n=9*)

Interpolate currents in a system onto a regular grid.

The system graph together with current intensities defines a "discrete" current density field where the current density is non-zero only on the straight lines that connect sites that are coupled by a hopping term.

To make this vector field easier to visualize and interpret at different length scales, it is smoothed by convoluting it with the bell-shaped bump function `f(r) = max(1 - (2*r / width)**2, 0)**2`. The bump width is determined by the *relwidth* and *abswidth* parameters.

This routine samples the smoothed field on a regular (square or cubic) grid.

Parameters **syst** : A finalized system

The system on which we are going to calculate the field.

current : '1D array of float'

Must contain the intensity on each hoppings in the same order that they appear in syst.graph.

relwidth : float or *None*

Relative width of the bumps used to generate the field, as a fraction of the length of the longest side of the bounding box. This argument is only used if *abswidth* is not given.

abswidth : float or *None*

Absolute width of the bumps used to generate the field. Takes precedence over *relwidth*. If neither is given, the bump width is set to four times the length of the shortest hopping.

n : int

Number of points the grid must have over the width of the bump.

Returns **field** : n-d arraylike of float

n-d array of n-d vectors.

box : sequence of 2-sequences of float

the extents of *field*: ((x0, x1), (y0, y1), ...)

kwant.plotter.interpolate_density

kwant.plotter.interpolate_density(*syst*, *density*, *relwidth=None*, *abswidth=None*, *n=9*, *mask=True*)

Interpolate density in a system onto a regular grid.

The system sites together with a scalar for each site defines a "discrete" density field where the density is non-zero only at the site positions.

To make this vector field easier to visualize and interpret at different length scales, it is smoothed by convoluting it with the bell-shaped bump function `f(r) = max(1 - (2*r / width)**2, 0)**2`. The bump width is determined by the *relwidth* and *abswidth* parameters.

This routine samples the smoothed field on a regular (square or cubic) grid.

Parameters **syst** : A finalized system

The system on which we are going to calculate the field.

density : 1D array of float

Must contain the intensity on each site in the same order that they appear in syst.sites.

relwidth : float, optional

Relative width of the bumps used to smooth the field, as a fraction of the length of the longest side of the bounding box. This argument is only used if `abswidth` is not given.

abswidth : float, optional

Absolute width of the bumps used to smooth the field. Takes precedence over `relwidth`. If neither is given, the bump width is set to four times the length of the shortest hopping.

n : int

Number of points the grid must have over the width of the bump.

mask : Bool

If True, this function returns a masked array that masks positions that are too far away from any sites. This is useful for showing an approximate outline of the system when the field is plotted.

Returns **field** : n-d arraylike of float

n-d array of n-d vectors.

box : sequence of 2-sequences of float

the extents of `field`: ((x0, x1), (y0, y1), …)

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 *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 *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 *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 *BuilderLead* leads are returned with an unfinalized system, and hoppings of *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 scattering region.
`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*, ***, *params=None*)

Compute the scattering matrix of a system.

An alias exists for this common name: `kwant.smatrix`.

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. Deprecated in favor of 'params' (and mutually exclusive with it).

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.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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*, ***, *params=None*)

Compute the retarded Green's function of the system between its leads.

An alias exists for this common name: `kwant.greens_function`.

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. Deprecated in favor of 'params' (and mutually exclusive with it).

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.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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*, ***, *params=None*)

Return a callable object for the computation of the wave function inside the scattering region.

An alias exists for this common name: `kwant.wave_function`.

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. Deprecated in favor of 'params' (and mutually exclusive with it).

check_hermiticity : `bool`

Check if the Hamiltonian matrices are Hermitian.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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 the negative velocity modes in *kwant.physics.PropagatingModes*. In Kwant's convention leads are attached so that their translational symmetry points *away* from the scattering region:

```
left lead    SR   right lead
/---------\ /---\ /---------\
...-3-2-1-0-X-X-X-0-1-2-3-...
```

This means that incoming modes (coming from infinity towards the scattering region) have *negative* velocity with respect to the lead's symmetry direction.

Examples

```
>>> wf = kwant.solvers.default.wave_function(some_syst, some_energy)
>>> wfs_of_lead_2 = wf(2)
```

kwant.solvers.default.ldos

`kwant.solvers.default.ldos(`*sys*, *energy=0*, *args=()*, *check_hermiticity=True*, ***, *params=None*`)`

Calculate the local density of states of a system at a given energy.

An alias exists for this common name: `kwant.ldos`.

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. Deprecated in favor of 'params' (and mutually exclusive with it).

check_hermiticity : `bool`

Check if the Hamiltonian matrices are Hermitian.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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*.

SMatrix also respects the conservation laws present in the lead, such as spin conservation, if they are declared during system construction. If queried with length-2 sequence the first number is the number of the lead, and the second number is the index of the corresponding conserved quantity. For example `smatrix.transmission((1, 3), (0, 2))` is transmission from block 2 of the conserved quantity in lead 0 to the block 3 of the conserved quantity in lead 1.

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 *lead_in* or *lead_out* are a length-2 sequence, the first number is the number of the lead, and the second number indexes the eigenvalue of the conserved quantity in that lead (e.g. spin) if it was specified.

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**(*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 `kwant.linalg.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**()

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.operator` – Operators and Observables

3.6.1 Observables

Density(syst[, onsite, where, ...])	An operator for calculating general densities.
Current(syst[, onsite, where, ...])	An operator for calculating general currents.
Source(syst[, onsite, where, ...])	An operator for calculating general sources.

kwant.operator.Density

`class kwant.operator.Density`(*syst*, *onsite=1*, *where=None*, *check_hermiticity=True*, ***, *sum=False*)

Bases: `kwant.operator._LocalOperator`

An operator for calculating general densities.

An instance of this class can be called like a function to evaluate the expectation value with a wavefunction. See *__call__* for details.

Parameters **syst** : *System*

onsite : scalar or square matrix or dict or callable

The onsite matrix that defines the operator. If a dict is given, it maps from site families to square matrices. If a function is given it must take the same arguments as the onsite Hamiltonian functions of the system.

where : sequence of *int* or *Site*, or callable, optional

Where to evaluate the operator. If `syst` is not a finalized Builder, then this should be a sequence of integers. If a function is provided, it should take a single *int* or *Site* (if `syst` is a finalized builder) and return True or False. If not provided, the operator will be calculated over all sites in the system.

check_hermiticity: bool

Check whether the provided `onsite` is Hermitian. If it is not Hermitian, then an error will be raised when the operator is evaluated.

sum : bool, default: False

If True, then calling this operator will return a single scalar, otherwise a vector will be returned (see *__call__* for details).

Notes

In general, if there is a certain "density" (e.g. charge or spin) that is represented by a square matrix M_i associated with each site i then an instance of this class represents the tensor $Q_{i\alpha\beta}$ which is equal to M_i when α and β are orbitals on site i, and zero otherwise.

Methods

`act`(*self*, *ket*, *args=()*, ***, *params=None*)

Act with the operator on a wavefunction.

For an operator $Q_{i\alpha\beta}$ and `ket` ψ_β this computes $\sum_{i\beta} Q_{i\alpha\beta}\psi_\beta$.

Parameters **ket** : sequence of complex

Wavefunctions defined over all the orbitals of the system.

args : tuple

The extra arguments to the Hamiltonian value functions and the operator `onsite` function. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns Array of *complex*.

`bind`(*self*, *args=()*, ***, *params=None*)

Bind the given arguments to this operator.

Returns a copy of this operator that does not need to be passed extra arguments when subsequently called or when using the `act` method.

Providing positional arguments via 'args' is deprecated, instead provide named parameters as a dictionary via 'params'.

`tocoo`(*self*, *args=()*, ***, *params=None*)

Convert the operator to coordinate format sparse matrix.

Providing positional arguments via 'args' is deprecated, instead provide named parameters as a dictionary via 'params'.

`__call__`()

Return the matrix elements of the operator.

An operator `A` can be called like

```
>>> A(psi)
```

to compute the expectation value $\langle\psi| A |\psi\rangle$, or like

```
>>> A(phi, psi)
```

to compute the matrix element $\langle\varphi| A |\psi\rangle$.

If `sum=True` was provided when constructing the operator, then a scalar is returned. If `sum=False`, then a vector is returned. The vector is defined over the sites of the system if the operator is a *`Density`*, or over the hoppings if it is a *`Current`* or *`Source`*. By default, the returned vector is ordered in the same way as the sites (for *`Density`*) or hoppings in the graph of the system (for *`Current`* or *`Density`*). If the keyword parameter `where` was provided when constructing the operator, then the returned vector is instead defined only over the sites or hoppings specified, and is ordered in the same way as `where`.

Alternatively stated, for an operator $Q_{i\alpha\beta}$, `bra` φ_α and `ket` ψ_β this computes $q_i = \sum_{\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$ if `self.sum` is False, otherwise computes $q = \sum_{i\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$. where i runs over all sites or hoppings, and α and β run over all the degrees of freedom.

Parameters bra, ket : sequence of complex

Must have the same length as the number of orbitals in the system. If only one is provided, both `bra` and `ket` are taken as equal.

args : tuple, optional

The arguments to pass to the system. Used to evaluate the `onsite` elements and, possibly, the system Hamiltonian. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns *float* if `check_hermiticity` is True, and `ket` is `None`,

otherwise *complex*. If this operator was created with `sum=True`,

then a single value is returned, otherwise an array is returned.

Attributes

`check_hermiticity`

`onsite`

`sum`

`syst`

`where`

kwant.operator.Current

class kwant.operator.**Current**(*syst*, *onsite=1*, *where=None*, *check_hermiticity=True*, ***, *sum=False*)

Bases: `kwant.operator._LocalOperator`

An operator for calculating general currents.

An instance of this class can be called like a function to evaluate the expectation value with a wavefunction. See *__call__* for details.

Parameters **syst** : *System*

onsite : scalar or square matrix or dict or callable

The onsite matrix that defines the density from which this current is derived. If a dict is given, it maps from site families to square matrices (scalars are allowed if the site family has 1 orbital per site). If a function is given it must take the same arguments as the onsite Hamiltonian functions of the system.

where : sequence of pairs of *int* or *Site*, or callable, optional

Where to evaluate the operator. If `syst` is not a finalized Builder, then this should be a sequence of pairs of integers. If a function is provided, it should take a pair of integers or a pair of *Site* (if `syst` is a finalized builder) and return True or False. If not provided, the operator will be calculated over all hoppings in the system.

check_hermiticity : bool

Check whether the provided `onsite` is Hermitian. If it is not Hermitian, then an error will be raised when the operator is evaluated.

sum : bool, default: False

If True, then calling this operator will return a single scalar, otherwise a vector will be returned (see *__call__* for details).

Notes

In general, if there is a certain "density" (e.g. charge or spin) that is represented by a square matrix M_i associated with each site i and H_{ij} is the hopping Hamiltonian from site j to site i, then an instance of this class represents the tensor $J_{ij\alpha\beta}$ which is equal to $i\left[(H_{ij})^\dagger M_i - M_i H_{ij}\right]$ when α and β are orbitals on sites i and j respectively, and zero otherwise.

The tensor $J_{ij\alpha\beta}$ will also be referred to as $Q_{n\alpha\beta}$, where n is the index of hopping (i, j) in `where`.

Methods

`act`(*self*, *ket*, *args=()*, ***, *params=None*)
Act with the operator on a wavefunction.

For an operator $Q_{i\alpha\beta}$ and `ket` ψ_β this computes $\sum_{i\beta} Q_{i\alpha\beta}\psi_\beta$.

Parameters **ket** : sequence of complex

Wavefunctions defined over all the orbitals of the system.

args : tuple

The extra arguments to the Hamiltonian value functions and the operator `onsite` function. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns Array of *complex*.

`bind`(*self*, *args=()*, ***, *params=None*)
Bind the given arguments to this operator.

Returns a copy of this operator that does not need to be passed extra arguments when subsequently called or when using the `act` method.

Providing positional arguments via 'args' is deprecated, instead provide named parameters as a dictionary via 'params'.

`__call__`()
Return the matrix elements of the operator.

An operator `A` can be called like

```
>>> A(psi)
```

to compute the expectation value $\langle\psi| A |\psi\rangle$, or like

```
>>> A(phi, psi)
```

to compute the matrix element $\langle\varphi| A |\psi\rangle$.

If `sum=True` was provided when constructing the operator, then a scalar is returned. If `sum=False`, then a vector is returned. The vector is defined over the sites of the system if the operator is a *`Density`*, or over the hoppings if it is a *`Current`* or *`Source`*. By default, the returned vector is ordered in the same way as the sites (for *`Density`*) or hoppings in the graph of the system (for *`Current`* or *`Density`*). If the keyword parameter `where` was provided when constructing the operator, then the returned vector is instead defined only over the sites or hoppings specified, and is ordered in the same way as `where`.

Alternatively stated, for an operator $Q_{i\alpha\beta}$, `bra` φ_α and `ket` ψ_β this computes $q_i = \sum_{\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$ if `self.sum` is False, otherwise computes $q = \sum_{i\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$. where i runs over all sites or hoppings, and α and β run over all the degrees of freedom.

Parameters **bra, ket** : sequence of complex

Must have the same length as the number of orbitals in the system. If only one is provided, both `bra` and `ket` are taken as equal.

args : tuple, optional

The arguments to pass to the system. Used to evaluate the `onsite` elements and, possibly, the system Hamiltonian. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns *float* if `check_hermiticity` is True, and `ket` is `None`,

otherwise *complex*. If this operator was created with `sum=True`,

then a single value is returned, otherwise an array is returned.

Attributes

`check_hermiticity`

`onsite`

`sum`

`syst`

`where`

kwant.operator.Source

class `kwant.operator.Source`(*syst*, *onsite=1*, *where=None*, *check_hermiticity=True*, ***, *sum=False*)

Bases: `kwant.operator._LocalOperator`

An operator for calculating general sources.

An instance of this class can be called like a function to evaluate the expectation value with a wavefunction. See *`__call__`* for details.

Parameters **syst** : *System*

onsite : scalar or square matrix or dict or callable

The onsite matrix that defines the density from which this source is defined. If a dict is given, it maps from site families to square matrices (scalars are allowed if the site family has 1 orbital per site). If a function is given it must take the same arguments as the onsite Hamiltonian functions of the system.

where : sequence of *int* or *Site*, or callable, optional

Where to evaluate the operator. If `syst` is not a finalized Builder, then this should be a sequence of integers. If a function is provided, it should take a single *int* or *Site* (if `syst` is a finalized builder) and return True or False. If not provided, the operator will be calculated over all sites in the system.

check_hermiticity : bool

Check whether the provided `onsite` is Hermitian. If it is not Hermitian, then an error will be raised when the operator is evaluated.

sum : bool, default: False

If True, then calling this operator will return a single scalar, otherwise a vector will be returned (see *`__call__`* for details).

Notes

An example of a "source" is a spin torque. In general, if there is a certain "density" (e.g. charge or spin) that is represented by a square matrix M_i associated with each site i, and H_i is the onsite Hamiltonian on site site i, then an instance of this class represents the tensor $Q_{i\alpha\beta}$ which is equal to $(H_i)^\dagger M_i - M_i H_i$ when α and β are orbitals on site i, and zero otherwise.

Methods

`act`(*self*, *ket*, *args=()*, ***, *params=None*)
Act with the operator on a wavefunction.

For an operator $Q_{i\alpha\beta}$ and `ket` ψ_β this computes $\sum_{i\beta} Q_{i\alpha\beta}\psi_\beta$.

Parameters **ket** : sequence of complex

Wavefunctions defined over all the orbitals of the system.

args : tuple

The extra arguments to the Hamiltonian value functions and the operator `onsite` function. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns Array of *complex*.

`bind`(*self*, *args=()*, ***, *params=None*)
Bind the given arguments to this operator.

Returns a copy of this operator that does not need to be passed extra arguments when subsequently called or when using the `act` method.

Providing positional arguments via 'args' is deprecated, instead provide named parameters as a dictionary via 'params'.

`__call__`()
Return the matrix elements of the operator.

An operator `A` can be called like

```
>>> A(psi)
```

to compute the expectation value $\langle\psi| A |\psi\rangle$, or like

```
>>> A(phi, psi)
```

to compute the matrix element $\langle\varphi| A |\psi\rangle$.

If `sum=True` was provided when constructing the operator, then a scalar is returned. If `sum=False`, then a vector is returned. The vector is defined over the sites of the system if the operator is a *`Density`*, or over the hoppings if it is a *`Current`* or *`Source`*. By default, the returned vector is ordered in the same way as the sites (for *`Density`*) or hoppings in the graph of the system (for *`Current`* or *`Density`*). If the keyword parameter `where` was provided when constructing the operator, then the returned vector is instead defined only over the sites or hoppings specified, and is ordered in the same way as `where`.

Alternatively stated, for an operator $Q_{i\alpha\beta}$, `bra` φ_α and `ket` ψ_β this computes $q_i = \sum_{\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$ if `self.sum` is False, otherwise computes $q = \sum_{i\alpha\beta} \varphi^*_\alpha Q_{i\alpha\beta}\psi_\beta$. where i runs over all sites or hoppings, and α and β run over all the degrees of freedom.

Parameters **bra, ket** : sequence of complex

Must have the same length as the number of orbitals in the system. If only one is provided, both `bra` and `ket` are taken as equal.

args : tuple, optional

The arguments to pass to the system. Used to evaluate the `onsite` elements and, possibly, the system Hamiltonian. Deprecated in favor of 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns *float* if `check_hermiticity` is True, and `ket` is `None`,

otherwise *complex*. If this operator was created with `sum=True`,

then a single value is returned, otherwise an array is returned.

Attributes

`check_hermiticity`

`onsite`

`sum`

`syst`

`where`

3.7 `kwant.physics` – Physics-related algorithms

3.7.1 Leads

Bands(sys[, args, params])	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=()*, ***, *params=None*)

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. Deprecated in favor or 'params' (and mutually exclusive with it).

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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. If *derivative_order* > *0* or *return_eigenvectors* = *True*, additional arrays are returned.

Examples

```
>>> bands = kwant.physics.Bands(some_syst)
>>> 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*, ***, *discrete_symmetry=None*, *projectors=None*, *time_reversal=None*, *particle_hole=None*, *chiral=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.

particle_hole : sparse or dense square matrix

The unitary part of the particle-hole symmetry operator.

time_reversal : sparse or dense square matrix

The unitary part of the time-reversal symmetry operator.

chiral : sparse or dense square matrix

The chiral symmetry operator.

projectors : an iterable of sparse or dense matrices

Projectors that block diagonalize the Hamiltonian in accordance with a conservation law.

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 sorting of the propagating modes is fully described in the documentation for *PropagatingModes*. 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. Within these halves the modes are ordered by the eigenvalues of any declared conservation laws. Within blocks with the same conservation law eigenvalue the modes with negative velocity are ordered by increasing momentum, and the modes with positive velocity are ordered by decreasing momentum. Finally, modes are ordered by the magnitude of their velocity. To summarize, the modes are ordered according to the key *(sign(v), conserved_quantity, sign(v) * k , abs(v))* where v is velocity, k is momentum and *conserved_quantity* is the conservation law eigenvalue.

In the above, the positive velocity and momentum directions are defined with respect to the translational symmetry direction of the system.

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.

If a conservation law is specified to block diagonalize the Hamiltonian to N blocks, then *block_nmodes[i]* is the number of left or right moving propagating modes in conservation law block *i*. The ordering of blocks is the same as the ordering of the projectors used to block diagonalize the Hamiltonian.

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.
block_nmodes: list of integers	Number of left or right moving propagating modes per conservation law block of the Hamiltonian.

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.)

3.7.2 Symmetry

DiscreteSymmetry([projectors, ...])	A collection of discrete symmetries and conservation laws.

kwant.physics.DiscreteSymmetry

`class kwant.physics.DiscreteSymmetry(`*projectors=None*, *time_reversal=None*, *particle_hole=None*, *chiral=None*`)`

Bases: `object`

A collection of discrete symmetries and conservation laws.

Parameters projectors : iterable of rectangular or square sparse matrices

Projectors that block-diagonalize the Hamiltonian.

time_reversal : square sparse matrix

The unitary part of the time-reversal symmetry operator.

particle_hole : square sparse matrix

The unitary part of the particle-hole symmetry operator.

chiral : square sparse matrix

The chiral symmetry operator.

Notes

Whenever one or more discrete symmetry is declared in conjunction with a conservation law, the symmetry operators and projectors must be declared in canonical form. This means that each block of the Hamiltonian is transformed either to itself by a discrete symmetry or to a single other block.

More formally, consider a discrete symmetry S. The symmetry projection that maps from block i to block j of the Hamiltonian with projectors P_i and P_j is $S_{ji} = P_j^+ S P_i$. If S_{ji} is nonzero, a

symmetry relation exists between blocks i and j. Canonical form means that for each j, the block S_{ji} is nonzero at most for one i, while all other blocks vanish.

If the operators are not in canonical form, they can be made so by further splitting the Hamiltonian into smaller blocks, i.e. by adding more projectors.

Methods

`validate`(*matrix*)

Check if a matrix satisfies the discrete symmetries.

Parameters **matrix** : sparse or dense matrix

If rectangular, it is padded by zeros to be square.

Returns **broken_symmetries** : list

List of strings, the names of symmetries broken by the matrix: any combination of "Conservation law", "Time reversal", "Particle-hole", "Chiral". If no symmetries are broken, returns an empty list.

3.7.3 Computation of magnetic field gauge

magnetic_gauge(syst)	Fix the magnetic gauge for a finalized system.

kwant.physics.magnetic_gauge

`class kwant.physics.magnetic_gauge`(*syst*)

Bases: `object`

Fix the magnetic gauge for a finalized system.

This can be used to later calculate the Peierls phases that should be applied to each hopping, given a magnetic field.

This API is currently provisional. Refer to the documentation for details.

Parameters **syst** : *kwant.builder.FiniteSystem* or *kwant.builder.InfiniteSystem*

Examples

The following illustrates basic usage for a scattering region with a single lead attached:

```
>>> import numpy as np
>>> import kwant
>>>
>>> def hopping(a, b, t, peierls):
>>>     return -t * peierls(a, b)
>>>
>>> syst = make_system(hopping)
>>> lead = make_lead(hopping)
>>> lead.substituted(peierls='peierls_lead')
>>> syst.attach_lead(lead)
>>> syst = syst.finalized()
>>>
>>> gauge = kwant.physics.magnetic_gauge(syst)
>>>
>>> def B_syst(pos):
>>>    return np.exp(-np.sum(pos * pos))
>>>
>>> peierls_syst, peierls_lead = gauge(B_syst, 0)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> params = dict(t=1, peierls=peierls_syst, peierls_lead=peierls_lead)
>>> kwant.hamiltonian_submatrix(syst, params=params)
```

Methods

__call__(*syst_field*, **lead_fields*, *tol=1e-08*, *average=False*)

Return the Peierls phase for a magnetic field configuration.

Parameters **syst_field** : scalar, vector or callable

The magnetic field to apply to the scattering region. If callable, takes a position and returns the magnetic field at that position. Can be a scalar if the system is 1D or 2D, otherwise must be a vector. Magnetic field is expressed in units φ_0/l^2, where φ_0 is the magnetic flux quantum and l is the unit of length.

***lead_fields** : scalar, vector or callable

The magnetic fields to apply to each of the leads, in the same format as 'syst_field'. In addition, if a callable is provided, then the magnetic field must have the symmetry of the associated lead.

tol : float, default: 1E-8

The tolerance to which to calculate the flux through each hopping loop in the system.

average : bool, default: False

If True, estimate the magnetic flux through each hopping loop in the system by evaluating the magnetic field at a single position inside the loop and multiplying it by the area of the loop. If False, then `scipy.integrate.quad` is used to integrate the magnetic field. This parameter is only used when 'syst_field' or 'lead_fields' are callable.

Returns **phases** : callable, or sequence of callables

The first callable computes the Peierls phase in the scattering region and the remaining callables compute the Peierls phases in each of the leads. Each callable takes a pair of *Site* (a hopping) and returns a unit complex number (Peierls phase) that multiplies that hopping.

CHAPTER

FOUR

OTHER MODULES

The following modules provide functionality for special applications.

4.1 `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.

4.2 `kwant.rmt` – Random matrix theory Hamiltonians

kwant.rmt.gaussian(*n*, *sym='A'*, *v=1.0*, *rng=None*)
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)*.

rng: int or rng (optional)

Seed or random number generator. If no 'rng' is provided the random number generator from numpy will be used.

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*, *rng=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.

rng: int or rng (optional)

Seed or random number generator. If no 'rng' is passed, the random number generator provided by numpy will be used.

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.

4.3 `kwant.kpm` – Kernel Polynomial Method

`class kwant.kpm.SpectralDensity`(*hamiltonian*, *params=None*, *operator=None*, *num_vectors=10*, *num_moments=None*, *energy_resolution=None*, *vector_factory=None*, *bounds=None*, *eps=0.05*, *rng=None*, *kernel=None*, *mean=True*, *accumulate_vectors=True*)

Bases: `object`

Calculate the spectral density of an operator.

This class makes use of the kernel polynomial method (KPM), presented in *[R1]*, to obtain the spectral density $\rho_A(e)$, as a function of the energy e, of some operator A that acts on a kwant system or a Hamiltonian. In general

$$\rho_A(E) = \rho(E)A(E),$$

where $\rho(E) = \sum_{k=0}^{D-1} \delta(E - E_k)$ is the density of states, and $A(E)$ is the expectation value of A for all the eigenstates with energy E.

Parameters **hamiltonian** : *FiniteSystem* or matrix Hamiltonian

If a system is passed, it should contain no leads.

params : dict, optional

Additional parameters to pass to the Hamiltonian and operator.

operator : operator, dense matrix, or sparse matrix, optional

Operator for which the spectral density will be evaluated. If it is callable, the `densities` at each energy will have the dimension of the result of `operator(bra, ket)`. If it has a `dot` method, such as `numpy.ndarray` and `scipy.sparse.matrices`, the densities will be scalars.

num_vectors : positive int, or None, default: 10

Number of vectors used in the KPM expansion. If `None`, the number of vectors used equals the length of the 'vector_factory'.

num_moments : positive int, default: 100

Number of moments, order of the KPM expansion. Mutually exclusive with `energy_resolution`.

energy_resolution : positive float, optional

The resolution in energy of the KPM approximation to the spectral density. Mutually exclusive with `num_moments`.

vector_factory : iterable, optional

If provided, it should contain (or yield) vectors of the size of the system. If not provided, random phase vectors are used. The default random vectors are optimal for most cases, see the discussions in *[R1]* and *[R2]*.

bounds : pair of floats, optional

Lower and upper bounds for the eigenvalue spectrum of the system. If not provided, they are computed.

eps : positive float, default: 0.05

Parameter to ensure that the rescaled spectrum lies in the interval `(-1, 1)`; required for stability.

rng : seed, or random number generator, optional

Random number generator used for the calculation of the spectral bounds, and to generate random vectors (if `vector_factory` is not provided). If not provided, numpy's rng will be used; if it is an Integer, it will be used to seed numpy's rng, and if it is a random number generator, this is the one used.

kernel : callable, optional

Callable that takes moments and returns stabilized moments. By default, the *jackson_kernel* is used. The Lorentz kernel is also accesible by passing *lorentz_kernel*.

mean : bool, default: `True`

If `True`, return the mean spectral density for the vectors used, otherwise return an array of densities for each vector.

accumulate_vectors : bool, default: `True`

Whether to save or discard each vector produced by the vector factory. If it is set to `False`, it is not possible to increase the number of moments, but less memory is used.

Notes

When passing an operator defined in *operator*, the result of `operator(bra, ket)` depends on the attribute `sum` of such operator. If `sum=True`, densities will be scalars, that is, total densities of the system. If `sum=False` the densities will be arrays of the length of the system, that is, local densities.

Examples

In the following example, we will obtain the density of states of a graphene sheet, defined as a honeycomb lattice with first nearest neighbors coupling.

We start by importing kwant and defining a *FiniteSystem*,

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

and after finalizing the system, create an instance of *SpectralDensity*

```
>>> fsyst = syst.finalized()
>>> rho = kwant.kpm.SpectralDensity(fsyst)
```

The `energies` and `densities` can be accessed with

```
>>> energies, densities = rho()
```

or

```
>>> energies, densities = rho.energies, rho.densities
```

Attributes

energies	(array of floats) Array of sampling points with length `2 * num_moments` in the range of the spectrum.
densities	(array of floats) Spectral density of the `operator` evaluated at the energies.

`__call__`(*energy=None*)
Return the spectral density evaluated at `energy`.

Parameters energy : float or sequence of floats, optional

Returns energies : array of floats

Drawn from the nodes of the highest Chebyshev polynomial. Not returned if 'energy' was not provided

densities : float or array of floats

single `float` if the `energy` parameter is a single `float`, else an array of `float`.

Notes

If `energy` is not provided, then the densities are obtained by Fast Fourier Transform of the Chebyshev moments.

`add_moments`(*num_moments=None*, *, *energy_resolution=None*)
Increase the number of Chebyshev moments.

Parameters num_moments: positive int

The number of Chebyshev moments to add. Mutually exclusive with `energy_resolution`.

energy_resolution: positive float, optional

Features wider than this resolution are visible in the spectral density. Mutually exclusive with `num_moments`.

`add_vectors`(*num_vectors=None*)
Increase the number of vectors

Parameters num_vectors: positive int, optional

The number of vectors to add.

`integrate`(*distribution_function=None*)
Returns the total spectral density.

Returns the integral over the whole spectrum with an optional distribution function. `distribution_function` should be able to take arrays as input. Defined using Gauss-Chebyshev integration.

`class kwant.kpm.Correlator`(*hamiltonian*, *operator1=None*, *operator2=None*, ***kwargs*)
Bases: `object`

Calculates the response of the correlation between two operators.

The response tensor $\chi_{\alpha\beta}$ of an operator O_α to a perturbation in an operator O_β, is defined here based on *[R5]*, and *[R6]*, and takes the form

$$\chi_{\alpha\beta}(,T) = \int_{-\infty}^{\infty} \mathrm{d}E f(-E,T) \left(O_\alpha \rho(E) O_\beta \frac{\mathrm{d}G^+}{\mathrm{d}E} - O_\alpha \frac{\mathrm{d}G^-}{\mathrm{d}E} O_\beta \rho(E) \right)$$

Internally, the correlation is approximated with a two dimensional KPM expansion,

$$\chi_{\alpha\beta}(,T) = \int_{-1}^{1} \mathrm{d}E \frac{f(-E,T)}{(1-E^2)^2} \sum_{m,n} \Gamma_{nm}(E)^{\alpha\beta}_{nm},$$

with coefficients

$$\Gamma_{mn}(E) = (E - in\sqrt{1-E^2})e^{in\arccos(E)}T_m(E) + (E + im\sqrt{1-E^2})e^{-im\arccos(E)}T_n(E),$$

and moments matrix $\overset{\alpha\beta}{}_{nm} = \mathrm{Tr}(O_\alpha T_m(H) O_\beta T_n(H))$.

The trace is calculated creating two instances of *SpectralDensity*, and saving the vectors $\Psi_{nr} = O_\beta T_n(H)|r\rangle$, and $\Omega_{mr} = T_m(H)O_\alpha|r\rangle$, where $|r\rangle$ is a vector provided by the `vector_factory`. The moments matrix is formed with the product $_{mn} = \langle\Omega_{mr}|\Psi_{nr}\rangle$ for every $|r\rangle$.

Parameters **hamiltonian** : *FiniteSystem* or matrix Hamiltonian

If a system is passed, it should contain no leads.

operator1, operator2 : operators, dense matrix, or sparse matrix, optional

Operators to be passed to two different instances of *SpectralDensity*.

****kwargs** : dict

Keyword arguments to pass to *SpectralDensity*.

Notes

The `operator1` must act to the right as $O_\alpha|r\rangle$.

`__call__`(*mu=0*, *temperature=0*)

Returns the linear response $\chi_{\alpha\beta}(, T)$

Parameters **mu** : float

Chemical potential defined in the same units of energy as the Hamiltonian.

temperature : float

Temperature in units of energy, the same as defined in the Hamiltonian.

`add_moments`(*num_moments=None*, ***, *energy_resolution=None*)

Increase the number of Chebyshev moments

Parameters **num_moments: positive int, optional**

The number of Chebyshev moments to add. Mutually exclusive with 'energy_resolution'.

energy_resolution: positive float, optional

Features wider than this resolution are visible in the spectral density. Mutually exclusive with `num_moments`.

`add_vectors`(*num_vectors=None*)

Increase the number of vectors

Parameters **num_vectors: positive int, optional**

The number of vectors to add.

`kwant.kpm.conductivity`(*hamiltonian*, *alpha='x'*, *beta='x'*, *positions=None*, ***kwargs*)

Returns a callable object to obtain the elements of the conductivity tensor using the Kubo-Bastin approach.

A *Correlator* instance is created to obtain the correlation between two components of the current operator

$$\sigma_{\alpha\beta}(, T) = \frac{1}{V}\int_{-\infty}^{\infty} \mathrm{d}E f(-E, T)\left(j_\alpha \rho(E) j_\beta \frac{\mathrm{d}G^+}{\mathrm{d}E} - j_\alpha \frac{\mathrm{d}G^-}{\mathrm{d}E} j_\beta \rho(E)\right),$$

where V is the volume where the conductivity is sampled. In this implementation it is assumed that the vectors are normalized and $V = 1$, otherwise the result of this calculation must be normalized with the corresponding volume.

The equations used here are based on *[R8]* and *[R9]*

Parameters alpha, beta : str, or operators

If `hamiltonian` is a kwant system, or if the `positions` are provided, `alpha` and `beta` can be the directions of the velocities as strings {'x', 'y', 'z'}. Otherwise `alpha` and `beta` should be the proper velocity operators, which can be members of *operator* or matrices.

positions : array of float, optioinal

If `hamiltonian` is a matrix, the velocities can be calculated internally by passing the positions of each orbital in the system when `alpha` or `beta` are one of the directions {'x', 'y', 'z'}.

****kwargs** : dict

Keyword arguments to pass to *Correlator*.

Examples

We will obtain the conductivity of the Haldane model, defined as a honeycomb lattice with first nearest neighbors coupling, and imaginary second nearest neighbors coupling.

We start by importing kwant and defining a *FiniteSystem*,

```
>>> import kwant
...
>>> def circle(pos):
...     x, y = pos
...     return x**2 + y**2 < 100
...
>>> lat = kwant.lattice.honeycomb()
>>> syst = kwant.Builder()
>>> syst[lat.shape(circle, (0, 0))] = 0
>>> syst[lat.neighbors()] = -1
>>> syst[lat.a.neighbors()] = -0.5j
>>> syst[lat.b.neighbors()] = 0.5j
>>> fsyst = syst.finalized()
```

Now we can call *conductivity* to calculate the transverse conductivity at chemical potential 0 and temperature 0.01.

```
>>> cond = kwant.kpm.conductivity(fsyst, alpha='x', beta='y')
>>> cond(mu=0, temperature=0.01)
```

`kwant.kpm.RandomVectors`(*syst*, *where=None*, *rng=None*)

Returns a random phase vector iterator for the sites in 'where'.

Parameters syst : *FiniteSystem* or matrix Hamiltonian

If a system is passed, it should contain no leads.

where : sequence of *int* or *Site*, or callable, optional

Spatial range of the vectors produced. If `syst` is a *FiniteSystem*, where behaves as in *Density*. If `syst` is a matrix, `where` must be a list of integers with the indices where column vectors are nonzero.

`class kwant.kpm.LocalVectors`(*syst*, *where=None*, **args*)

Bases: `object`

Generates a local vector iterator for the sites in 'where'.

Parameters **syst** : *FiniteSystem* or matrix Hamiltonian

If a system is passed, it should contain no leads.

where : sequence of *int* or *Site*, or callable, optional

Spatial range of the vectors produced. If `syst` is a *FiniteSystem*, where behaves as in *Density*. If `syst` is a matrix, `where` must be a list of integers with the indices where column vectors are nonzero.

`kwant.kpm.jackson_kernel(`*moments*`)`
Convolutes `moments` with the Jackson kernel.

Taken from Eq. (71) of Rev. Mod. Phys., Vol. 78, No. 1 (2006).

`kwant.kpm.lorentz_kernel(`*moments*, *l=4*`)`
Convolutes `moments` with the Lorentz kernel.

Taken from Eq. (71) of Rev. Mod. Phys., Vol. 78, No. 1 (2006).

The additional parameter `l` controls the decay of the kernel.

`kwant.kpm.fermi_distribution(`*energy*, *mu*, *temperature*`)`
Returns the Fermi distribution f(e, μ, T) evaluated at 'e'.

Parameters **energy** : float or sequence of floats

Energy array where the Fermi distribution is evaluated.

mu : float

Chemical potential defined in the same units of energy as the Hamiltonian.

temperature : float

Temperature in units of energy, the same as defined in the Hamiltonian.

4.4 `kwant.continuum` – Tools for continuum systems

4.4.1 Discretizer

discretize(hamiltonian[, coords, grid, ...])	Construct a tight-binding model from a continuum Hamiltonian.
discretize_symbolic(hamiltonian[, coords, ...])	Discretize a continuous Hamiltonian into a tight-binding representation.
build_discretized(tb_hamiltonian, coords, *)	Create a template builder from a symbolic tight-binding Hamiltonian.

kwant.continuum.discretize

`kwant.continuum.discretize(`*hamiltonian*, *coords=None*, ***, *grid=None*, *locals=None*, *grid_spacing=None*`)`
Construct a tight-binding model from a continuum Hamiltonian.

If necessary, the given Hamiltonian is sympified using *kwant.continuum.sympify*. It is then discretized symbolically and turned into a *Builder* instance that may be used with *fill*.

This is a convenience function that is equivalent to first calling *discretize_symbolic* and feeding its result into *build_discretized*.

> **Warning:** This function uses `eval` (because it calls `sympy.sympify`), and thus should not be used on unsanitized input.

Parameters **hamiltonian** : str or SymPy expression

Symbolic representation of a continuous Hamiltonian. It is converted to a SymPy expression using *kwant.continuum.sympify*.

coords : sequence of strings, optional

The coordinates for which momentum operators will be treated as differential operators. May contain only "x", "y" and "z" and must be sorted. If not provided, *coords* will be obtained from the input Hamiltonian by reading the present coordinates and momentum operators.

grid : scalar or kwant.lattice.Monatomic instance, optional

Lattice that will be used as a discretization grid. It must have orthogonal primitive vectors. If a scalar value is given, a lattice with the appropriate grid spacing will be generated. If not provided, a lattice with grid spacing 1 in all directions will be generated.

locals : dict, optional

Additional namespace entries for *sympify*. May be used to simplify input of matrices or modify input before proceeding further. For example: `locals={'k': 'k_x + I * k_y'}` or `locals={'sigma_plus': [[0, 2], [0, 0]]}`.

grid_spacing : int or float, optional

(deprecated) Spacing of the discretization grid. If unset the default value will be 1. Cannot be used together with `grid`.

Returns model : *Builder*

The translationally symmetric builder that corresponds to the provided Hamiltonian. This builder instance belongs to a subclass of the standard builder that may be printed to obtain the source code of the value functions. It also holds the discretization lattice (a *Monatomic* instance with lattice constant *grid_spacing*) in the `lattice` attribute.

kwant.continuum.discretize_symbolic

`kwant.continuum.discretize_symbolic`(*hamiltonian*, *coords=None*, *, *locals=None*)

Discretize a continuous Hamiltonian into a tight-binding representation.

If necessary, the given Hamiltonian is sympified using *kwant.continuum.sympify*. It is then discretized symbolically.

The two return values may be used directly as the first two arguments for *build_discretized*.

Warning: This function uses `eval` (because it calls `sympy.sympify`), and thus should not be used on unsanitized input.

Parameters hamiltonian : str or SymPy expression

Symbolic representation of a continuous Hamiltonian. It is converted to a SymPy expression using *kwant.continuum.sympify*.

coords : sequence of strings, optional

The coordinates for which momentum operators will be treated as differential operators. May contain only "x", "y" and "z" and must be sorted. If not provided, *coords* will be obtained from the input Hamiltonian by reading the present coordinates and momentum operators.

locals : dict, optional

Additional namespace entries for *sympify*. May be used to simplify input of matrices or modify input before proceeding further. For example: `locals={'k': 'k_x + I * k_y'}` or `locals={'sigma_plus': [[0, 2], [0, 0]]}`.

Returns tb_hamiltonian : dict

Keys are tuples of integers; the offsets of the hoppings ((0, 0, 0) for the onsite). Values are symbolic expressions for the hoppings/onsite.

coords : list of strings

The coordinates that have been discretized.

kwant.continuum.build_discretized

`kwant.continuum.build_discretized`(*tb_hamiltonian*, *coords*, ***, *grid=None*, *locals=None*, *grid_spacing=None*)

Create a template builder from a symbolic tight-binding Hamiltonian.

The provided symbolic tight-binding Hamiltonian is put on a (hyper) square lattice and turned into Python functions. These functions are used to create a *Builder* instance that may be used with *fill* to construct a system of a desired shape.

The return values of *discretize_symbolic* may be used directly for the first two arguments of this function.

> **Warning:** This function uses `eval` (because it calls `sympy.sympify`), and thus should not be used on unsanitized input.

Parameters tb_hamiltonian : dict

Keys must be the offsets of the hoppings, represented by tuples of integers ((0, 0, 0) for onsite). Values must be symbolic expressions for the hoppings/onsite or expressions that can by sympified with *kwant.continuum.sympify*.

coords : sequence of strings

The coordinates for which momentum operators will be treated as differential operators. May contain only “x”, “y” and “z” and must be sorted.

grid : scalar or kwant.lattice.Monatomic instance, optional

Lattice that will be used as a discretization grid. It must have orthogonal primitive vectors. If a scalar value is given, a lattice with the appropriate grid spacing will be generated. If not provided, a lattice with grid spacing 1 in all directions will be generated.

locals : dict, optional

Additional namespace entries for *sympify*. May be used to simplify input of matrices or modify input before proceeding further. For example: `locals={'k': 'k_x + I * k_y'}` or `locals={'sigma_plus': [[0, 2], [0, 0]]}`.

grid_spacing : int or float, optional

(deprecated) Spacing of the discretization grid. If not provided, the default value will be 1. Cannot be used together with `grid`.

Returns model : *Builder*

The translationally symmetric builder that corresponds to the provided Hamiltonian. This builder instance belongs to a subclass of the standard builder that may be printed to obtain the source code of the value functions.

It also holds the discretization lattice (a *Monatomic* instance with lattice constant *grid_spacing*) in the `lattice` attribute.

4.4.2 Symbolic helpers

sympify(expr[, locals])	Sympify object using special rules for Hamiltonians.
lambdify(expr[, locals])	Return a callable object for computing a continuum Hamiltonian.

kwant.continuum.sympify

`kwant.continuum.sympify`(*expr*, *locals=None*)

Sympify object using special rules for Hamiltonians.

If *'expr'* is already a type that SymPy understands, it will do nothing but return that value. Note that `locals` will not be used in this situation.

Otherwise, it is sympified by `sympy.sympify` with a modified namespace such that

- the position operators "x", "y" or "z" and momentum operators "k_x", "k_y", and "k_z" do not commute,
- all single-letter identifiers and names of Greek letters (e.g. "pi" or "gamma") are treated as symbols,
- "kron" corresponds to `sympy.physics.quantum.TensorProduct`, and "identity" to `sympy.eye`,
- "sigma_0", "sigma_x", "sigma_y", "sigma_z" are the Pauli matrices.

In addition, Python list literals are interpreted as SymPy matrices.

Warning: This function uses `eval` (because it calls `sympy.sympify`), and thus should not be used on unsanitized input.

Parameters **expr** : str or SymPy expression

Expression to be converted to a SymPy object.

locals : dict or `None` (default)

Additional entries for the namespace under which *expr* is sympified. The keys must be valid Python variable names. The values may be strings, since they are all are sent through *continuum.sympify* themselves before use. (Note that this is a difference to how `sympy.sympify` behaves.)

Note: When a value of *locals* is already a SymPy object, it is used as-is, and the caller is responsible to set the commutativity of its symbols appropriately. This possible source of errors is demonstrated in the last example below.

Returns **result** : SymPy object

Examples

```
>>> sympify('k_x * A(x) * k_x + V(x)')
k_x*A(x)*k_x + V(x)     # as valid sympy object
```

```
>>> sympify('k_x**2 + V', locals={'V': 'V_0 + V(x)'})
k_x**2 + V(x) + V_0
```

```
>>> ns = {'sigma_plus': [[0, 2], [0, 0]]}
>>> sympify('k_x**2 * sigma_plus', ns)
Matrix([
[0, 2*k_x**2],
[0,        0]])
```

```
>>> sympify('k_x * A(c) * k_x', locals={'c': 'x'})
k_x*A(x)*k_x
>>> sympify('k_x * A(c) * k_x', locals={'c': sympy.Symbol('x')})
A(x)*k_x**2
```

kwant.continuum.lambdify

`kwant.continuum.lambdify`(*expr*, *locals=None*)

Return a callable object for computing a continuum Hamiltonian.

> **Warning:** This function uses `eval` (because it calls `sympy.sympify`), and thus should not be used on unsanitized input.

If necessary, the given expression is sympified using *kwant.continuum.sympify*. It is then converted into a callable object.

Parameters **expr** : str or SymPy expression

Expression to be converted into a callable object

locals : dict or `None` (default)

Additional definitions for *sympify*.

Examples

```
>>> f = lambdify('a + b', locals={'b': 'b + c'})
>>> f(1, 3, 5)
9
```

```
>>> ns = {'sigma_plus': [[0, 2], [0, 0]]}
>>> f = lambdify('k_x**2 * sigma_plus', ns)
>>> f(0.25)
array([[ 0.   ,  0.125],
       [ 0.   ,  0.   ]])
```

4.5 `kwant.wraparound` – Wrapping around translational symmetries

wraparound(builder[, keep, coordinate_names])	Replace translational symmetries by momentum parameters.
plot_2d_bands(syst[, k_x, k_y, params, ...])	Plot 2D band structure of a wrapped around system.

4.5.1 kwant.wraparound.wraparound

`kwant.wraparound.wraparound`(*builder*, *keep=None*, *, *coordinate_names='xyz'*)

Replace translational symmetries by momentum parameters.

A new Builder instance is returned. By default, each symmetry is replaced by one scalar momentum parameter that is appended to the already existing arguments of the system. Optionally, one symmetry may be kept by using the *keep* argument. The momentum parameters will have names

like 'k_n' where the 'n' are specified by 'coordinate_names'.

Parameters builder : *Builder*

keep : int, optional

Which (if any) translational symmetry to keep.

coordinate_names : sequence of strings, default: 'xyz'

The names of the coordinates along the symmetry directions of 'builder'.

Notes

Wraparound is stop-gap functionality until Kwant 2.x which will include support for higher-dimension translational symmetry in the low-level system format. It will be deprecated in the 2.0 release of Kwant.

4.5.2 kwant.wraparound.plot_2d_bands

`kwant.wraparound.plot_2d_bands`(*syst*, *k_x=31*, *k_y=31*, *params=None*, *mask_brillouin_zone=False*, *extend_bbox=0*, *file=None*, *show=True*, *dpi=None*, *fig_size=None*, *ax=None*)

Plot 2D band structure of a wrapped around system.

This function is primarily useful for systems that have translational symmetry vectors that are non-orthogonal (e.g. graphene). This function will properly plot the band structure in an orthonormal basis in k-space, as opposed to in the basis of reciprocal lattice vectors (which would produce a "skewed" Brillouin zone).

If your system has orthogonal lattice vectors, you are probably better off using *kwant.plotter.spectrum*.

Parameters syst : *kwant.system.FiniteSystem*

A 2D system that was finalized from a Builder produced by *kwant.wraparound.wraparound*. Note that this *must* be a finite system; so *kwant.wraparound.wraparound* should have been called with `keep=None`.

k_x, k_y : int or sequence of float, default: 31

Either a number of sampling points, or a sequence of points at which the band structure is to be evaluated, in units of inverse length.

params : dict, optional

Dictionary of parameter names and their values, not including the momentum parameters.

mask_brillouin_zone : bool, default: False

If True, then the band structure will only be plotted over the first Brillouin zone. By default the band structure is plotted over a rectangular bounding box that contains the Brillouin zone.

extend_bbox : float, default: 0

Amount by which to extend the region over which the band structure is plotted, expressed as a proportion of the Brillouin zone bounding box length. i.e. `extend_bbox=0.1` will extend the region by 10% (in all directions).

file : string or file object, optional

The output file. If None, output will be shown instead.

show : bool, default: False

Whether `matplotlib.pyplot.show()` is to be called, and the output is to be shown immediately. Defaults to *True*.

dpi : float, optional

Number of pixels per inch. If not set the `matplotlib` default is used.

fig_size : tuple, optional

Figure size *(width, height)* in inches. If not set, the default `matplotlib` value is used.

ax : `matplotlib.axes.Axes` instance, optional

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.

See also:

kwant.plotter.spectrum

Notes

This function produces plots where the units of momentum are inverse length. This is contrary to *kwant.plotter.bands*, where the units of momentum are inverse lattice constant.

If the lattice vectors for the symmetry of `syst` are not orthogonal, then part of the plotted band structure will be outside the first Brillouin zone (inside the bounding box of the brillouin zone). Setting `mask_brillouin_zone=True` will cause the plot to be truncated outside of the first Brillouin zone.

4.6 `kwant.qsymm` – Integration with Qsymm

`kwant.qsymm.builder_to_model`(*syst*, *momenta=None*, *real_space=True*, *params=None*)

Make a qsymm.BlochModel out of a *Builder*.

Parameters **syst** : *Builder*

May have translational symmetries.

momenta : list of strings or None

Names of momentum variables. If None, 'k_x', 'k_y', ... is used.

real_space : bool (default True)

If False, use the unit cell convention for Bloch basis, the exponential has the difference in the unit cell coordinates and k is expressed in the reciprocal lattice basis. This is consistent with *kwant.wraparound*. If True, the difference in the real space coordinates is used and k is given in an absolute basis. Only the default choice guarantees that qsymm is able to find nonsymmorphic symmetries.

params : dict, optional

Dictionary of parameter names and their values; used when evaluating the Hamiltonian matrix elements.

Returns **model** : qsymm.BlochModel

Model representing the tight-binding Hamiltonian.

Notes

The sites in the the builder are in lexicographical order, i.e. ordered first by their family and then by their tag. This is the same ordering that is used in finalized kwant systems.

kwant.qsymm.model_to_builder(*model*, *norbs*, *lat_vecs*, *atom_coords*, *, *coeffs=None*)

Make a *Builder* out of qsymm.Models or qsymm.BlochModels.

Parameters **model** : qsymm.Model, qsymm.BlochModel, or an iterable thereof

The Hamiltonian (or terms of the Hamiltonian) to convert to a Builder.

norbs : OrderedDict or sequence of pairs

Maps sites to the number of orbitals per site in a unit cell.

lat_vecs : list of arrays

Lattice vectors of the underlying tight binding lattice.

atom_coords : list of arrays

Positions of the sites (or atoms) within a unit cell. The ordering of the atoms is the same as in norbs.

coeffs : list of sympy.Symbol, default None.

Constant prefactors for the individual terms in model, if model is a list of multiple objects. If model is a single Model or BlochModel object, this argument is ignored. By default assigns the coefficient c_n to element model[n].

Returns **syst** : *Builder*

The unfinalized Kwant system representing the qsymm Model(s).

Notes

Onsite terms that are not provided in the input model are set to zero by default.

The input model(s) representing the tight binding Hamiltonian in Bloch form should follow the convention where the difference in the real space atomic positions appear in the Bloch factors.

kwant.qsymm.find_builder_symmetries(*builder*, *momenta=None*, *params=None*, *spatial_symmetries=True*, *prettify=True*, *sparse=None*)

Finds the symmetries of a Kwant system using qsymm.

Parameters **builder** : *Builder*

momenta : list of strings or None

Names of momentum variables, if None 'k_x', 'k_y', … is used.

params : dict, optional

Dictionary of parameter names and their values; used when evaluating the Hamiltonian matrix elements.

spatial_symmetries : bool (default True)

If True, search for all symmetries. If False, only searches for the symmetries that are declarable in *Builder* objects, i.e. time-reversal symmetry, particle-hole symmetry, chiral symmetry, or conservation laws. This can save computation time.

prettify : bool (default True)

Whether to carry out sparsification of the continuous symmetry generators, in general an arbitrary linear combination of the symmetry generators is returned.

sparse : bool, or None (default None)

Whether to use sparse linear algebra in the calculation. Can give large performance gain in large systems. If None, uses sparse or dense computation depending on the size of the Hamiltonian.

Returns symmetries : list of qsymm.PointGroupElements and/or qsymm.ContinuousGroupElement

The symmetries of the Kwant system.

CHAPTER

FIVE

MODULES MAINLY FOR INTERNAL USE

The following modules contain functionality that is most often used only internally by Kwant itself or by advanced users.

5.1 `kwant.system` – Low-level interface of systems

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.

In practice, very often Kwant systems are finalized *builders* (i.e. *kwant.builder.FiniteSystem* or *kwant.builder.InfiniteSystem*) and offer some additional attributes.

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.

5.1.1 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.

Due to the ordering semantics of sequences, and the fact that a given `first_site` can only appear *at most once* in `site_ranges`, `site_ranges` is ordered according to `first_site`.

Consecutive elements in `site_ranges` are not required to have different numbers of orbitals.

Attributes

graph	(kwant.graph.CGraph) The system graph.
site_ranges	(None or sorted sequence of triples of integers) If provided, encodes ranges of sites that have the same number of orbitals. Each triple consists of `(first_site, norbs, orb_offset)`: the first site in the range, the number of orbitals on each site in the range, and the offset of the first orbital of the first site in the range. In addition, the final triple should have the form `(len(graph.num_nodes), 0, tot_norbs)` where `tot_norbs` is the total number of orbitals in the system.
parameters	(frozenset of strings) The names of the parameters on which the system depends. This attribute is provisional and may be changed in a future version of Kwant

Methods

`discrete_symmetry`(*args*, *, *params=None*)

Return the discrete symmetry of the system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian`(*i*, *j*, **args*, *params=None*)

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.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian_submatrix`(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*, *, *params=None*)

Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method. Mutually exclusive with 'params'.

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`.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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.

5.1.2 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*, ***, *params=None*)

Hamiltonian of a single cell of the infinite system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

discrete_symmetry(*args*, ***, *params=None*)

Return the discrete symmetry of the system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian`(*i*, *j*, **args*, *params=None*)
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.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian_submatrix`(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*, ***, *params=None*)
Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method. Mutually exclusive with 'params'.

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`.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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*, ***, *params=None*)
Hopping Hamiltonian between two cells of the infinite system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`modes`(*energy=0*, *args=()*, ***, *params=None*)
Return mode decomposition of the lead

See documentation of *PropagatingModes* and *StabilizedModes* for the return format details.

The wave functions of the returned modes are defined over the *unit cell* of the system, which corresponds to the degrees of freedom on the first `cell_sites` sites of the system (recall that infinite systems store first the sites in the unit cell, then connected sites in the neighboring unit cell).

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

selfenergy(*energy=0*, *args=()*, ***, *params=None*)
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))`.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

validate_symmetries(*args=()*, ***, *params=None*)
Check that the Hamiltonian satisfies discrete symmetries.

Returns *validate* applied to the onsite matrix and the hopping. See its documentation for details on the return format.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

5.1.3 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` and `lead_paddings`.

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 *PropagatingModes*, *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` that has the same signature as *InfiniteSystem.selfenergy* (without the `self` parameter), and must have property `parameters`: a collection of strings that name the system parameters (though this requirement is provisional and may be removed in a future version of Kwant). It may also provide `modes` that has the same signature as *InfiniteSystem.modes* (without the `self` parameter).
lead_interfaces	(sequence of sequences of integers) Each sub-sequence contains the indices of the system sites to which the lead is connected.
lead_paddings	(sequence of sequences of integers) Each sub-sequence contains the indices of the system sites that belong to the lead, and therefore have the same onsite as the lead sites, and are connected by the same hoppings as the lead sites.
parameters	(frozenset of strings) The names of the parameters on which the system depends. This does not include the parameters for any leads. This attribute is provisional and may be changed in a future version of Kwant

Methods

`discrete_symmetry`(*args*, *, *params=None*)
Return the discrete symmetry of the system.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian`(*i*, *j*, **args*, *params=None*)
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.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

`hamiltonian_submatrix`(*self*, *args=()*, *to_sites=None*, *from_sites=None*, *sparse=False*, *return_norb=False*, *, *params=None*)
Return a submatrix of the system Hamiltonian.

Parameters **args** : tuple, defaults to empty

Positional arguments to pass to the `hamiltonian` method. Mutually exclusive with 'params'.

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`.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

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'*, *, *params=None*)
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. Deprecated in favor of 'params' (and mutually exclusive with it).

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'.

params : dict, optional

Dictionary of parameter names and their values. Mutually exclusive with 'args'.

Returns syst : 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.

`validate_symmetries`(*args=()*, ***, *params=None*)

Check that the Hamiltonian satisfies discrete symmetries.

Applies *`validate`* to the Hamiltonian, see its documentation for details on the return format.

Providing positional arguments via 'args' is deprecated, instead, provide named parameters as a dictionary via 'params'.

5.1.4 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=()*, ***, *params=None*)

`selfenergy`(*energy=0*, *args=()*, ***, *params=None*)

5.2 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*.

5.2.1 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`

5.2.2 Other

`gint_dtype`	Data type used for graph nodes and edges

5.3 `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.

- genindex

BIBLIOGRAPHY

[R1] Rev. Mod. Phys., Vol. 78, No. 1 (2006).

[R2] Phys. Rev. E 69, 057701 (2004)

[R5] Phys. Rev. Lett. 114, 116602 (2015).

[R6] Phys. Rev. B 92, 184415 (2015)

[R8] Phys. Rev. Lett. 114, 116602 (2015).

[R9] Phys. Rev. B 92, 184415 (2015)

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W