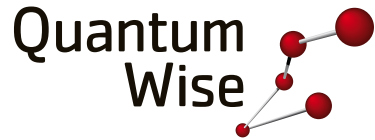


Slater-Koster Tight Binding Models in ATK-SE

Implementing Your Own Models

Version 12.2.0

Slater-Koster Tight Binding Models in ATK-SE: Implementing Your Own Models

Version 12.2.0

Copyright © 2008–2012 QuantumWise A/S

Atomistix ToolKit Copyright Notice

All rights reserved.

This publication may be freely redistributed in its full, unmodified form. No part of this publication may be incorporated or used in other publications without prior written permission from the publisher.

Table of Contents

1. Introduction 1
2. Pi model for carbon 2
    Introduction 2
    Onsite element 2
    Offsite elements 3
    Usage examples 6
3. Nearest-neighbor model for silicon with dummy hydrogen atoms 13
    Introduction 13
    Onsite element 13
    Offsite elements 14
    Adding hydrogen 16
    A slightly different take of the script 17
    Usage examples 19
Bibliography 23

Chapter 1. Introduction

ATK-SemiEmpirical (ATK-SE) comes preloaded with Slater-Koster parameters for simple systems like Si and GaAs, using a couple of different nearest-neighbor models like sp3s* and sp3d5s*. One of the very important features of ATK-SE is however that the user can define new Slater-Koster models for different systems, e.g. found in articles, and this will be demonstrated in this tutorial.

The tutorial begins with a simple but powerful model for pure carbon. The simplicity of this model allows it to be applied to very large structures, like carbon nanotubes, graphite, graphene, diamond, etc., although it is limited to situations where the system can be assumed to be perfect, i.e. it does not take into account any effects of edge relaxations for instance. Next, a slightly more complex model, for Si-H, but still nearest-neighbor, will be introduced. This can be used for instance to simulate H-terminated Si nanowires.

There are two primary ingredients in a Slater-Koster model for the electronic structure:

- the values of the onsite matrix elements, for each element (or atom types) in the model,
- and the values of the offsite matrix elements, for each pair of atom types in the model, and how these depend on the distance between the atoms.

These points will be discussed extensively in the following chapters.

Chapter 2. Pi model for carbon

Introduction

In this chapter a simple π basis set for carbon will be implemented, based on the paper by Y. Hancock et al. [1].

In this model the entire basis set for carbon is reduced to a single s-like orbital. Interactions with up to third nearest neighbors are however included. The model is also spin-polarized.

Tip

This model is already available as a built-in basis set for carbon in ATK (as of version 12.2). The purpose of this tutorial is however to show how you can implement your own tight-binding models in ATK, and the carbon π basis set provides an excellent example for this.

If you are only interested in *using* this model for tight-binding calculations of graphene and other carbon system, it is recommended to instead follow the corresponding tutorial where the built-in basis set is used; you can find it alongside the other tutorials on the website.

Onsite element

First, consider the onsite matrix element. Here you need to specify the **angular momenta** and **occupations** of the basis orbitals. Generally you also have to provide the **ionization potentials** of the orbitals, but since there is only a single orbital this can be set to zero, since the absolute energy zero is arbitrary.

The model is self-consistent, and therefore it is also necessary to specify the **onsite Hartree shift**; this can however be taken from the built-in database in ATK.

Finally, since this is a spin-polarized model, the **onsite spin splitting** must be defined. Ref. [1] uses a value of 2 eV, however their model is slightly different from that in ATK, and therefore a better approach is to fit the value to a DFT calculation of a graphene nanoribbon. Specifically, this is done by looking at the net magnetization of the edge atoms, as given by the Mulliken populations, and adjusting the onsite spin split parameter to give the same result as with LSDA. This is obtained with a value of the spin split of -1.0 eV.

Note

There is a built-in database of spin split parameters in ATK as well. To use that, you would set `onsite_spin_split = ATK_W(Carbon,['2p'])`, which would yield the value -0.6 eV. The difference is caused by the fact that there is not a full shell of 2p orbitals in the

model considered here, only a single π orbital. For more common sp3 models, the database values should work fine.

The resulting onsite matrix element definition becomes

```
C_onsite = SlaterKosterOnsiteParameters(
                element = Carbon,
                angular_momenta      = [0],
                occupations          = [1],
                ionization_potential = [0.0*eV],
                onsite_hartree_shift = [ATK_U(Carbon,['2p'])],
                onsite_spin_split    = [[-1.0]]*eV
           )
```

Offsite elements

For the offsite elements, the values given in the article [1] will be used: -2.7 eV, -0.2 eV, -0.18 eV, for the first, second and third nearest neighbors, respectively (note the sign!).

In general, however, a Slater-Koster model is not based around the concept of neighbors, but rather each offsite parameter requires a scaling function which determines the distance-dependence of the Hamiltonian matrix element. A cut-off distance should also be specified, above which atoms do not interact (the matrix element is set to zero at this distance).

To implement an Nth nearest-neighbor model, you can use a piecewise constant function, which has the same value at and around each respective neighbor distance. The function should look as in the plot below.

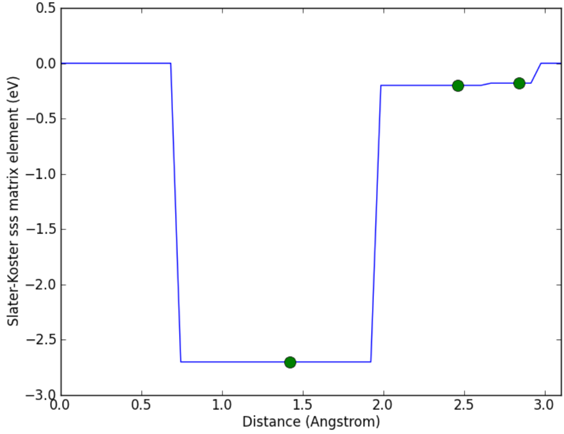


Figure 2.1: The sss matrix element, as a function of distance. The green circles mark the 1st, 2nd, and 3rd nearest neighbor distances.

This can be achieved in many ways in Python; for instance like this:

```
# Specify the cut-off, outside which all matrix elements for sure will be zero
rcut = 3.1

# The 1st, 2nd, and 3rd nearest neighbor distances,
# including low (0.0) and high cut-off distances
nn = [0., 1.42, 2.46, 2.84, rcut ]

# The matrix elements from the paper; the first and last elements
# will be used for very large and very small distances
```

```
energies = [0., -2.7, -0.2, -0.18, 0.]

# Generate a sequence of distances, from zero to the cut-off
distances = numpy.linspace(0.0,rcut,51)

# Compute the matrix elements
hamiltonian = piecewise_constant_function(distances,nn,energies)
```

In this code,

- the nearest neighbor distances are defined, and a cut-off above which the matrix element becomes zero (the variable **nn** should also contain a low-end cut-off, which can always be taken at 0.0 Ang);
- the values the function will take in the different intervals are listed, i.e. the matrix elements from the article; note that the function is set to zero for very small and very large distances;
- a sequence of distances is generated, at which the function will be evaluated;
- finally, the Python function which generates the piecewise constant function is called.

The code that generates the piecewise constant function can be of general interest:

```
def make_breakpoints(points):
    '''
    Returns the breakpoints needed for piecewise_constant_function
    based on the input of the midpoints of each constant segment
    '''
    b = [points[0]]
    for i in range(1,len(points)):
        b += [(points[i]+points[i-1])/2]
    b += [points[-1]]
    return b

def piecewise_constant_function(x,points,y):
    '''
    Returns a piecewise constant function f(x) as a numpy array
    f(x) = y[0] if b[0] < x < b[1]
    f(x) = y[1] if b[1] < x < b[2]
    etc
    where the points b lie halfway in-between the points in "points"
    x are the abscissa values, the returned array will have the same length
    '''
    b = make_breakpoints(points)
    f = numpy.zeros(x.shape)
    for i in range(len(b)-1):
        f[(x>b[i]) & (x<b[i+1])] = y[i]
    return f
```

Here, first the breakpoints are generated, at which the function will change its value, based on the points at which the function value is specified (the variable **points** in the function **piecewise_constant_function()**). Then the function is evaluated at all values given by the variable **x**, using the values from **y**, in the various intervals.

This construction can be very easily be adjusted for other situations. If for instance you want a pure nearest-neighbor model for graphene, all you need is to change **nn**, **energies**, and the cut-off:

```
# Specify the cut-off, outside which all matrix elements for sure will be zero
rcut = 3.1

# The 1st, 2nd, and 3rd nearest neighbor distances,
# including low (0.0) and high cut-off distances
```

```
nn = [0., 1.42, rcut ]

# The matrix elements from the paper; the first and last elements
# will be used for very large and very small distances
energies = [0., -2.7, 0.]
```

Anyway, once you have come this far, all that is left is to set up the Slater-Koster table, which is the basis set for a Slater-Koster calculation in ATK-SE. The table includes both the onsite and offsite matrix elements:

```
C_Basis = SlaterKosterTable(C_C_sss = zip(distances*Ang, hamiltonian*eV),
                            Carbon  = C_onsite,
                           )
```

The keywords for **`SlaterKosterTable`** must obey some rules, so that ATK knows how to set up the basis set.

- The **offsite** matrix elements are marked by keywords **`X_Y_ijk`** where
  - **`X`** and **`Y`** are element labels (see below),
  - **`i`** and **`j`** i and j are orbital labels: s, p, d, or f.
  - **`k`** k is "s" or "p" for σ or π.

  Thus, the sdπ matrix element between carbon and silicon (if such a element was present in the model) would be set with the keyword **`si_c_sdp`**.
- The onsite matrix elements are marked by element labels, so for carbon you simply use the keyword **`carbon`**

Note

Some additional remarks:

- The order of the element labels **`X`** and **`Y`** matters, if there is more than one element in the model. The orbital **`i`** is associated with the element **`X`** orbital, and **`j`** with **`Y`**.
- The element labels are not case-sensitive and may be abbreviated. Thus, C, c, carbon, or Carbon are all accepted for carbon.

As mentioned, our simple π basis set for carbon is actually implemented as an s-orbital model, hence i=j=s and k=s.

This completes the definition of the model. The complete script is given below, for reference.

```
from NanoLanguage import *

'''
Generate a Slater-Koster basis set for carbon based on
Hancock et. al., J. Low Temp Phys. 153, 393 (2008)

It's a p-p pi model, which we describe as an s-band model

The onsite_hartree_shift is taken from the ATK parameter sets
The onsite spin split is fitted to LDA calculations for a nanoribbon
(the standard parameter for a Carbon p orbital is -0.6 eV
and the article uses U=2 eV but their model is slightly different)
'''

def make_breakpoints(points):
```

```
    '''
    Returns the breakpoints needed for piecewise_constant_function
    based on the input of the midpoints of each constant segment
    '''
    b = [points[0]]
    for i in range(1,len(points)):
        b += [(points[i]+points[i-1])/2]
    b += [points[-1]]
    return b

def piecewise_constant_function(x,points,y):
    '''
    Returns a piecewise constant function f(x) as a numpy array
    f(x) = y[0] if b[0] < x < b[1]
    f(x) = y[1] if b[1] < x < b[2]
    etc
    where the points b lie halfway in-between the points in "points"
    x are the abscissa values, the returned array will have the same length
    '''
    b = make_breakpoints(points)
    f = numpy.zeros(x.shape)
    for i in range(len(b)-1):
        f[(x>b[i]) & (x<b[i+1])] = y[i]
    return f

# Specify the cut-off, outside which all matrix elements for sure will be zero
rcut = 3.1

# The 1st, 2nd, and 3rd nearest neighbor distances,
# including low (0.0) and high cut-off distances
nn = [0., 1.42, 2.46, 2.84, rcut ]

# The matrix elements from the paper; the first and last elements
# will be used for very large and very small distances
energies = [0., -2.7, -0.2, -0.18, 0.]

# Generate a sequence of distances, from zero to the cut-off
distances = numpy.linspace(0.0,rcut,51)

# Compute the matrix elements
hamiltonian = piecewise_constant_function(distances,nn,energies)

C_onsite = SlaterKosterOnsiteParameters(
                element = Carbon,
                angular_momenta      = [0],
                occupations          = [1],
                ionization_potential = [0.0*eV],
                onsite_hartree_shift = [ATK_U(Carbon,['2p'])],
                onsite_spin_split    = [[-1.0]]*eV
           )

C_Basis = SlaterKosterTable(C_C_sss = zip(distances*Ang, hamiltonian*eV),
                            Carbon  = C_onsite,
                           )
```

Usage examples

Band structure of graphene

As a demonstration of how this basis set can be used, you will now compute the band structure of graphene, using a simple script.

```
# Graphene
lattice = Hexagonal(2.4612*Angstrom, 6.709*Angstrom)
```

```
elements = [Carbon, Carbon]
fractional_coordinates = [[ 0.    ,  0.    ,  0. ],
                          [ 1./3.,  2./3.,  0. ]]
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

from CCPiBasisSet import C_Basis as basis_set

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(9, 9, 1),
    )

# Run the calculation self-consistently, and with spin-polarization
iteration_control_parameters = IterationControlParameters()

calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    iteration_control_parameters=iteration_control_parameters,
    spin_polarization=True,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
nlsave('graphene.nc', bulk_configuration)

# Bandstructure
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=['G', 'M', 'K', 'A'],
    points_per_segment=100,
    bands_above_fermi_level=All
    )
nlsave('graphene.nc', bandstructure)
```

Download and save this script, as well as the script `CCPiBasisSet.py` defining the basis set [5], in the same directoty. The script must be run from the command line, in this directory to manage the import of the basis set properly. It finishes in a few seconds, and yields a simple but accurate band structure of graphene.

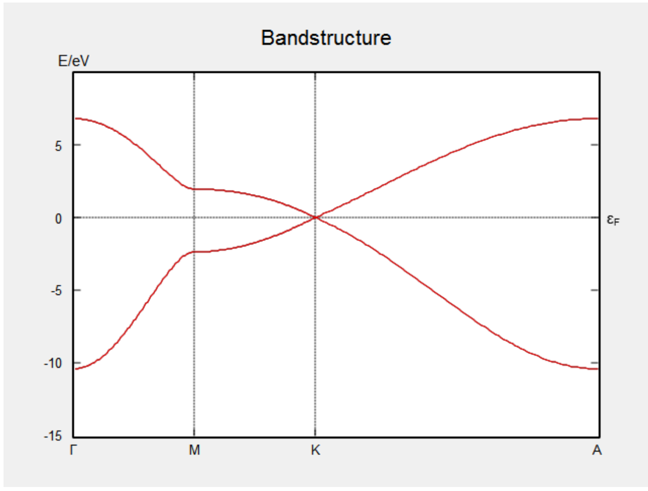


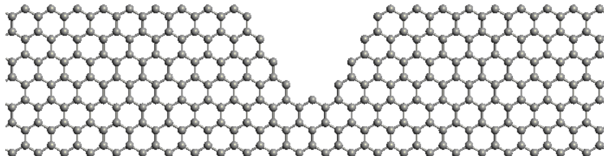
Figure 2.2: The band structure of graphene, computed with a simple π model. Graphene is not magnetic, so there is no difference in the spin up and down band structure, even if we used a spin-polarized model for the calculation.

Spin-polarized graphene nanoribbon with V-shaped notch

A more interesting example, where you also can see how the spin-polarization plays an important role, can be taken from the article where we found this tight-binding model itself, i.e. Ref. [1].

The example will proceed along the following lines:

1. The first step is to build the configuration shown in the figure below. For this you will use Virtual NanoLab, and it will be assumed that you are quite familiar with how to operate the various tools in this program, thus the instructions here will be quite condensed.



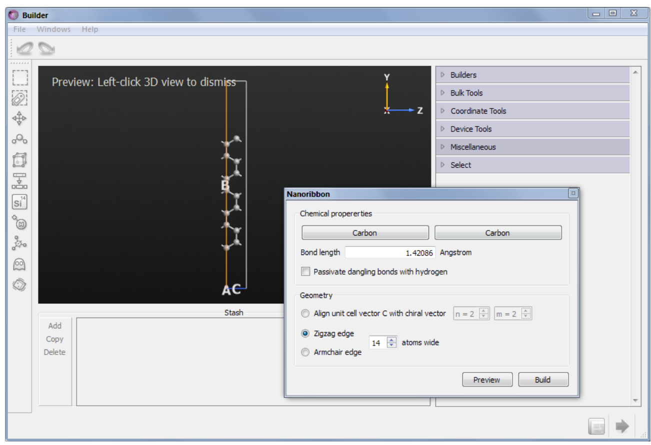
2. For the second step it will be possible to use the Script Generator in VNL to set up the calculation, at least most of it. It will however then be necessary to edit the script, to incorporate the basis set implemented above.

3. You will also have to edit the script to define the initial spin polarization of the structure. A graphene ribbon must be initialized with a different spin polarization on the two edges, otherwise it just converges to a ferromagnetic state.

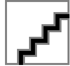
4. Finally, after the calculation has been carried out, you will be introduced to a script that plots the results in the same way as in the article referred to above.

Building the structure

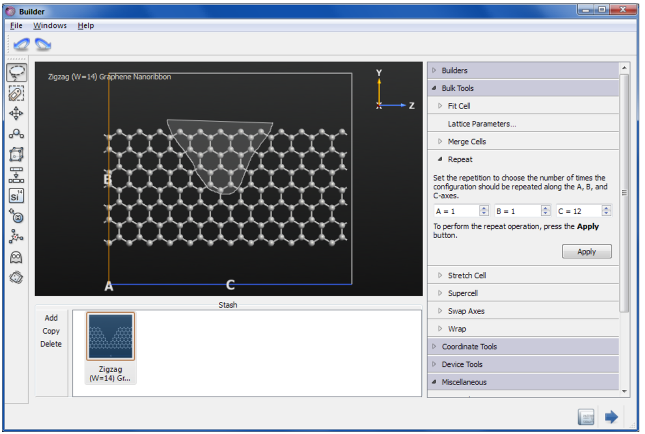
In VNL, open the **Builder** , and click `Add>Add Custom>Nanoribbon`. Choose a zigzag edge, 14 atoms wide, and also untick the checkbox "Passivate dangling bonds with hydrogen".



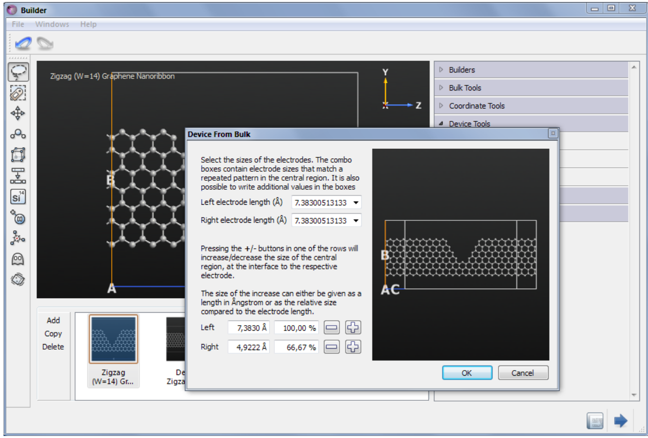
Repeat the structure 12 times in the C-direction.



Delete the carbon atoms indicated in the picture below to create the V-shaped notch. An easy way to select the atoms is to use the lasso tool, and then press Delete on the keyboard.



Now convert the structure to a device using Device From Bulk, under Device Tools. The suggested electrode length matches the system in the article by Hancock et al., but the central region must be modified a bit. Extend it by 100% (of the electrode width) on the left, and by 66.6666% on the right by entering these numbers into the respective field and pressing the icon for both. The resulting setup is shown in the figure below. Press **OK** to add the device configuration to the stash.



This completes the geometry setup.

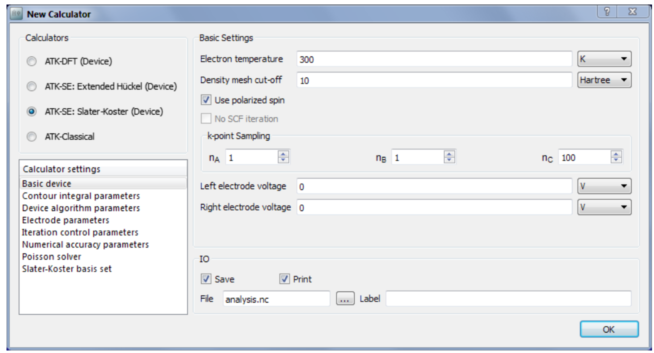
Setting up a template script

Send the structure to the Script Generator, and insert a New Calculator. Open the New Calculator and select the **ATK-SE: Slater-Koster (device)** calculator.

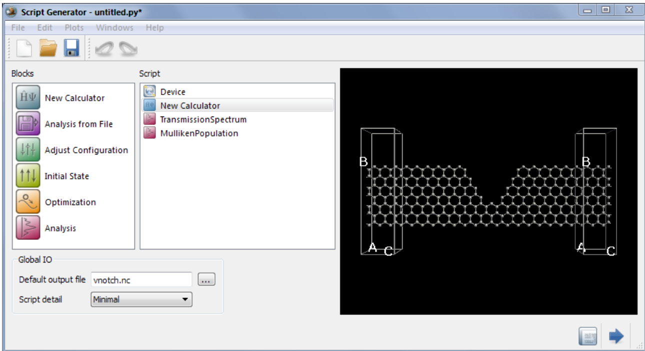
Select **Use polarized spin**. All other options can be left at their default values.

Note

When selecting that the calculation should be spin-polarized, the "No SCF iteration" option, which is the default for Slater-Koster models, is no longer available; a spin-polarized model needs to be self-consistent to produce sensible results.



From Analysis, insert a **TransmissionSpectrum** block (use 301 points to get good resolution, and the **Krylov** self-energy method for optimal speed) and also a MullikenPopulation block. Finally, set the output file name to something suitable, like `vnotch.nc`.



Save the script as `vnotch.py`.

Editing the script

Open the saved script in a text editor; you may use the built-in one in VNL, or your own favorite code editor.

The script needs to be edited for two things: to define the initial spin, and to use the basis set implemented above. Starting with the basis set, locate the line

```
basis_set = Vogl.C_Basis
```

and replace it by

```
from CCPiBasisSet import C_Basis as basis_set
```

Next, immediately after this line, add the following lines:

```
scaled_spins=[0.0, ]*len(central_region_coordinates)
for i in range(len(central_region_coordinates)):
    if central_region_coordinates[i,1]<8.*Ang:
        scaled_spins[i] = -1.
    elif central_region_coordinates[i,1]>12.*Ang:
        scaled_spins[i] = 1.
initial_spin = InitialSpin(scaled_spins=scaled_spins)
```

This sets up an initial state with opposite spin polarization on the two sides of the ribbon. It's not important that this is done very carefully, in terms of which precise atoms that are polarized how much and so on, as long as the two sides are clearly oppositely polarized.

Finally, this initial state needs to be incorporated in the calculation. Locate the line

```
device_configuration.setCalculator(calculator)
```

and change it into

```
device_configuration.setCalculator(calculator,
                                   initial_spin=initial_spin,
                                  )
```

Running the calculation

The script needs to be run from the command line, just as the graphene example above, again in the same directory where the file `CCPiBasisSet.py` resides. It should take no more than a few minutes to complete, even on a laptop (which is rather nice, considering it is a self-consistent calculation with 214 atoms in the central region).

To inspect the results, locate the created file `vnotch.nc` in VNL and plot the transmission spectrum. From the `Options` menu you may choose to not flip the spin-down spectrum, which simplifies the comparison to Fig. 3a in Ref. [1].

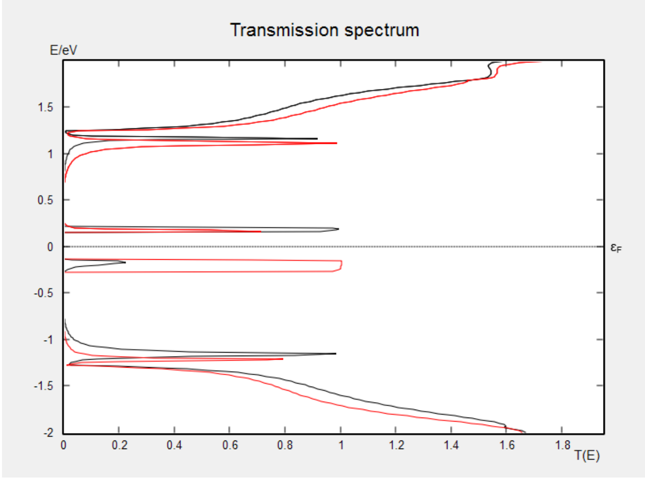


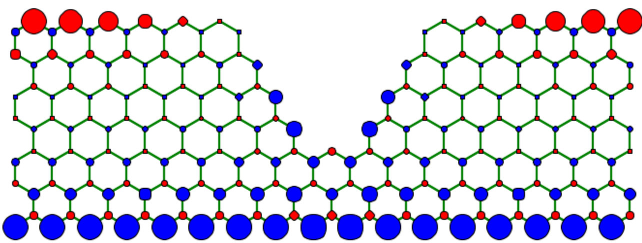
Figure 2.3:In this plot we have also enabled anti aliasing (right-click the plot to select it).

Spin polarization plot

It would also be interesting to visualize the results similar to Fig. 2a in Ref. [1]. To do this, run the script below in a terminal window, and have a look at the image file `spinplot.png` it produces.

```
config = nlread("vnotch.nc",DeviceConfiguration)[0]
coords = config.cartesianCoordinates().inUnitsOf(Ang)
bonds = []
for atom in coords:
    d = numpy.sqrt(numpy.sum((atom - coords)**2,axis=1))
    bonds.append(numpy.where((d<1.5) & (d>0.1))[0])
```

```
m = nlread("vnotch.nc",MullikenPopulation)[0]
diff_m = m.atoms(Spin.Up)-m.atoms(Spin.Down)

import pylab

fig = pylab.figure()
P = fig.gca()

# Bonds
for i in range(len(coords)):
    for j in bonds[i]:
        P.plot([coords[i][2],coords[j][2]],[coords[i][1],coords[j][1]],'g-')

# Mulliken populations plotted as red/blue circles at atom positions
# Area of circle is proportional to net spin population (M_UP-M_DOWN)
for i in range(len(coords)):
    if diff_m[i]<0:
        P.plot(coords[i][2],coords[i][1],'bo',markersize=numpy.sqrt(abs(diff_m[i]))*25)
    else:
        P.plot(coords[i][2],coords[i][1],'ro',markersize=numpy.sqrt(abs(diff_m[i]))*25)

P.set_aspect(1.)
P.set_xticks([100])
P.set_yticks([100])
P.axis([-3,43,-3,43])
pylab.savefig("spinplot.png",dpi=150)
```



In this plot, the two colors indicate a surplus of spin up or down, respectively, while the area of each circle is proportional to the degree of spin polarization at each atom, as measured by the difference in the Mulliken spin up vs. down population. The agreement with Fig. 2a in Ref. [1] is excellent.

Chapter 3. Nearest-neighbor model for silicon with dummy hydrogen atoms

Introduction

Next, it is instructive to consider a model with more than one element, to see how that changes things. A commonly used Slater-Koster parametrization for studying Si nanowires is the $sp^3d^5s^*$ model by Zheng et al. [2] which extends an earlier parametrization of Si and Ge [3], which in turn builds on the original $sp^3d^5s^*$ models by Jancu et al. [4]. (The latter is available as a built-in basis set in ATK.)

It is useful to start by defining the model for Si, since it is a little bit more involved than the simple pi-model of the previous chapter, and also quite useful on its own. Then the hydrogen matrix elements will be added to the model; these have a very specific purpose in this model, namely to remove spurious states due to dangling Si bonds on the surface of the nanowire. So, they actually just constitute a numerical trick, as discussed in detail in Ref. [2].

Onsite element

As before, the onsite and offsite matrix elements need to be defined, and the main question is how the latter scale as a function of the distance.

Starting with the onsite parameters, the ionization potentials can be taken directly from Ref. [3]. There are two occupied orbitals (3s and 3p), and two unoccupied ones (3d and an excited s-type orbital, commonly denoted s* in the literature). Since spin will not be considered (to be precise, the model does contain spin-orbit interaction, but this is not yet implemented in ATK), the spin split parameter can be left out. As before, the onsite Hartree shifts can be extracted from the ATK U database; for the unoccupied orbitals 3d and s* will be use values from occupied the valence orbitals:

```
Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV

onsite = SlaterKosterOnsiteParameters(
            element               = Silicon,
            angular_momenta       = [0,1,2,0],
            occupations           = [2,2,0,0],
            ionization_potential  = [Es,Ep,Ed,Es1],
            onsite_hartree_shift  = ATK_U(Silicon,['3s','3p','3p','3s'])
           )
```

Offsite elements

This is a nearest-neighbor model, but the scaling function for the matrix elements is fitted to allow for at least somewhat irregular structures. It is commonly assumed that the matrix elements ijk(d) follow a power law

$$ijk(d) = ijk(d_0)\left(\frac{d_0}{d}\right)^{n_{ijk}},$$

where d_0 is the ideal neighbor distance. Harrison [5] originally used a general exponent of $n_{ijk} = 2$ for all matrix elements ijk; later work, like Ref. [4], extends this to fit an individual exponent for each matrix element.

The scaling function can be tabulated in an interval around the nearest-neighbor distance using the following recipe:

```
# Nearest and second nearest neighbor distances
# where a is the cubic lattice constant
d_n1 = sqrt(3.0)/4.0*a
d_n2 = sqrt(2.0)/2.0*a
# Interpolate distances and energies
epsilon = linspace(-0.10, 0.10, 21)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]
ijk = [ ijk0/(1.0+x)**n_ijk for x in epsilon ] + [ 0.0*eV ]
```

Here **ijk0** is the matrix element from the article, and the cut-off distance is taken at the midpoint between the first and second nearest neighbors (the separate element added to the list) where the corresponding matrix element is set to zero. This way a power law scaling is obtained around the nearest-neighbor distance **d_n1**.

Now, you just need to collect the parameters from the article:

```
si_si_sss   = [ -1.95933*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1s1s = [ -4.24135*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ss1s  = [ -1.52230*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_sps   = [  3.02562*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ps  = [  3.15565*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_sds   = [ -2.28485*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ds  = [ -0.80993*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pps   = [  4.10364*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ppp   = [ -1.51801*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pds   = [ -1.35554*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pdp   = [  2.38479*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_dds   = [ -1.68136*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddp   = [  2.58880*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddd   = [ -1.81400*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
```

In this code $n_{ijk} = 2$ is used for all parameters, but it is clearly very easy to modify this to have a specific value for the exponent for each parameter [4].

The final, complete script, with also the Slater-Koster table definition (cf. the rules regarding keyword name in the previous chapter [5]) becomes as follows:

```
from NanoLanguage import *

'''
Generate a Slater-Koster basis set for silicon based on
T. B. Boykin et al., PRB 69, 115201 (2004)
```

```
http://dx.doi.org/10.1103/PhysRevB.69.115201
'''

from math import sqrt

# a is the cubic lattice constant
a = 5.430*Ang
# Nearest and second nearest neighbor distances
d_n1 = sqrt(3.0)/4.0*a
d_n2 = sqrt(2.0)/2.0*a

epsilon = numpy.linspace(-0.10, 0.10, 21)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

si_si_sss   = [ -1.95933*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1s1s = [ -4.24135*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ss1s  = [ -1.52230*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_sps   = [  3.02562*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ps  = [  3.15565*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_sds   = [ -2.28485*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ds  = [ -0.80993*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pps   = [  4.10364*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ppp   = [ -1.51801*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pds   = [ -1.35554*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pdp   = [  2.38479*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_dds   = [ -1.68136*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddp   = [  2.58880*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddd   = [ -1.81400*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]

Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV

onsite = SlaterKosterOnsiteParameters(
            element              = Silicon,
            angular_momenta      = [0,1,2,0],
            occupations          = [2,2,0,0],
            ionization_potential = [Es,Ep,Ed,Es1],
            onsite_hartree_shift = ATK_U(Silicon,['3s','3p','3p','3s'])
           )

basis_set = SlaterKosterTable(silicon     = onsite,
                              si_si_sss   = zip(distances,si_si_sss),
                              si_si_s1s1s = zip(distances,si_si_s1s1s),
                              si_si_ss1s  = zip(distances,si_si_ss1s),
                              si_si_sps   = zip(distances,si_si_sps),
                              si_si_s1ps  = zip(distances,si_si_s1ps),
                              si_si_sds   = zip(distances,si_si_sds),
                              si_si_s1ds  = zip(distances,si_si_s1ds),
                              si_si_pps   = zip(distances,si_si_pps),
                              si_si_ppp   = zip(distances,si_si_ppp),
                              si_si_pds   = zip(distances,si_si_pds),
                              si_si_pdp   = zip(distances,si_si_pdp),
                              si_si_dds   = zip(distances,si_si_dds),
                              si_si_ddp   = zip(distances,si_si_ddp),
                              si_si_ddd   = zip(distances,si_si_ddd),
                             )
```

Note

This basis set contains two s-type orbitals (3s and s*). To distinguish orbitals of the same angular momentum, there are two things to keep in mind:

- For the **onsite** elements, the identification is made by the ordering of the ionization potentials vs. the occupations and angular momenta, etc.

- For the **offsite** elements, a running index is added, so the first s-orbital is "s0", the second one "s1", etc. The "0" for the first orbital can be omitted to reduce the notation, so as you see above the s*pσ matrix element will be defined by the keyword "si_si_s1ps", etc.

Adding hydrogen

The goal in this chapter is to enable modeling of H-terminated Si nanowires, and now it is time to include the parameters for hydrogen [2]. As discussed above, the role of the hydrogen atoms is to eliminate the effects of the unsaturated Si bonds on the surface of the wire, not to represent "real" hydrogen atoms. (In an experiment it is of course quite possible that hydrogen atoms, or other atoms, will attach themselves to these dangling bonds.)

There must naturally be an onsite element for hydrogen also:

```
Es_H    =  0.999840*eV
onsite_H = SlaterKosterOnsiteParameters(element             = Hydrogen,
                                        angular_momenta     = [0],      # s-model for H
                                        occupations         = [1],      # 1s
                                        ionization_potential = [Es_H],
                                        onsite_hartree_shift = [0]*eV
                                        )
```

The scaling of the offsite parameters is completely analogous to those for Si:

```
h_si_sss  = [ -3.999720*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_ss1s = [ -1.697700*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_sps  = [  4.251750*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_sds  = [ -2.105520*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
```

The only difference compared to before is that now there are two elements, which introduces keywords of the form "si_h_sss":

```
from NanoLanguage import *

'''
Generate a Slater-Koster basis set for silicon based on
T. B. Boykin et al., PRB 69, 115201 (2004)
http://dx.doi.org/10.1103/PhysRevB.69.115201
with hydrogen parameters, to remove effects of dangling bonds, from
Y. Zheng et al., IEEE Trans Electron Devices 52, 1097 (2005)
http://dx.doi.org/10.1109/TED.2005.848077
'''

from math import sqrt

# a is the cubic lattice constant
a = 5.430*Ang
# Nearest and second nearest neighbor distances
d_n1 = sqrt(3.0)/4.0*a
d_n2 = sqrt(2.0)/2.0*a

epsilon = numpy.linspace(-0.10, 0.10, 21)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

si_si_sss   = [ -1.95933*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1s1s = [ -4.24135*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ss1s  = [ -1.52230*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_sps   = [  3.02562*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ps  = [  3.15565*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
```

```
si_si_sds   = [ -2.28485*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_s1ds  = [ -0.80993*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pps   = [  4.10364*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ppp   = [ -1.51801*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pds   = [ -1.35554*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_pdp   = [  2.38479*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_dds   = [ -1.68136*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddp   = [  2.58880*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]
si_si_ddd   = [ -1.81400*eV / (1.0+x)**2 for x in epsilon ] + [ 0.0*eV]

Es     = -2.15168*eV
Ep     =  4.22925*eV
Ed     = 13.78950*eV
Es1    = 19.11650*eV

onsite = SlaterKosterOnsiteParameters(element              = Silicon,
                                      angular_momenta      = [0,1,2,0],      # sp3d5s*
                                      occupations          = [2,2,0,0],      # 2s^2 2p^2
                                      ionization_potential = [Es,Ep,Ed,Es1],
                                      onsite_hartree_shift = [0,0,0,0]*eV
                                      )

Es_H    =  0.999840*eV
onsite_H = SlaterKosterOnsiteParameters(element              = Hydrogen,
                                        angular_momenta      = [0],      # s-model for H
                                        occupations          = [1],      # 1s
                                        ionization_potential = [Es_H],
                                        onsite_hartree_shift = [0]*eV
                                        )

h_si_sss  = [ -3.999720*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_ss1s = [ -1.697700*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_sps  = [  4.251750*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]
h_si_sds  = [ -2.105520*eV / (1.0+x)**2 for x in epsilon ] + [0.0*eV]

basis_set = SlaterKosterTable(silicon     = onsite,
                              hydrogen    = onsite_H,
                              si_si_sss   = zip(distances,si_si_sss),
                              si_si_s1s1s = zip(distances,si_si_s1s1s),
                              si_si_ss1s  = zip(distances,si_si_ss1s),
                              si_si_sps   = zip(distances,si_si_sps),
                              si_si_s1ps  = zip(distances,si_si_s1ps),
                              si_si_sds   = zip(distances,si_si_sds),
                              si_si_s1ds  = zip(distances,si_si_s1ds),
                              si_si_pps   = zip(distances,si_si_pps),
                              si_si_ppp   = zip(distances,si_si_ppp),
                              si_si_pds   = zip(distances,si_si_pds),
                              si_si_pdp   = zip(distances,si_si_pdp),
                              si_si_dds   = zip(distances,si_si_dds),
                              si_si_ddp   = zip(distances,si_si_ddp),
                              si_si_ddd   = zip(distances,si_si_ddd),
                              h_si_sss    = zip(distances,h_si_sss),
                              h_si_ss1s   = zip(distances,h_si_ss1s),
                              h_si_sps    = zip(distances,h_si_sps),
                              h_si_sds    = zip(distances,h_si_sds),
                             )
```

A slightly different take of the script

The script above [14] for the Si basis set should be straightforward to modify for other elements. For instance, you can easily obtain a basis set for Ge using parameters from Ref. [3], or GaAs and all other III-V binary compounds by taking parameters from Ref. [4].

For precisely these two references the orbital labels etc are basically identical for all systems. If, on the other hand, you find a different model in an article, with quite different orbitals etc, there would be a relatively large risk to make a typo in the conversion process.

To this end, you may want to consider an alternative method to setup the Si basis set:

```
from NanoLanguage import *

element         = Silicon
angular_momenta = [0,1,2,0]      # sp3d5s*
occupations     = [2,2,0,0]      # 2s^2 2p^2

# a is the cubic lattice constant
a = 5.430*Ang

Es    = -2.15168*eV
Ep    =  4.22925*eV
Ed    = 13.78950*eV
Es1   = 19.11650*eV
ionization_potentials = [Es,Ep,Ed,Es1]

offsite = {
    "sss"   : -1.95933*eV,
    "s*s*s" : -4.24135*eV,
    "ss*s"  : -1.52230*eV,
    "sps"   :  3.02562*eV,
    "s*ps"  :  3.15565*eV,
    "sds"   : -2.28485*eV,
    "s*ds"  : -0.80993*eV,
    "pps"   :  4.10364*eV,
    "ppp"   : -1.51801*eV,
    "pds"   : -1.35554*eV,
    "pdp"   :  2.38479*eV,
    "dds"   : -1.68136*eV,
    "ddp"   :  2.58880*eV,
    "ddd"   : -1.81400*eV,
}

# ------------------------------------------------------------------------------
# Usually no need to change anything below

from math import sqrt
from string import lower

# Nearest neighbor distance
d_n1 = sqrt(3.0)/4.0*a
# Second-nearest neighbor distance
d_n2 = sqrt(2.0)/2.0*a

epsilon = numpy.linspace(-0.10, 0.10, 21)
distances = [ d_n1*(1.0+x) for x in epsilon ] + [ 0.5*(d_n1+d_n2) ]

# Element label (Si for Silicon, etc); use for the offsite keywords
e = lower(element.name())

# Generate a keyword argument for each offsite parameter in the dict
args = ""
for p in offsite.keys():
    args += "%s_%s_%s = zip(distances, \
             [ %f*eV /(1.0+x)**2 for x in epsilon ] + [ 0.0*eV]),\n" \
             % (e,e,p.replace("*","1"),offsite[p])

onsite = SlaterKosterOnsiteParameters(element              = element,
                                      angular_momenta      = angular_momenta,
                                      occupations          = occupations,
                                      ionization_potential = ionization_potentials,
                                      onsite_hartree_shift = [0,]*len(occupations)*eV
                                      )

exec("""basis_set = SlaterKosterTable(%s = onsite,%s)""" % (e,args))
```

To some extent the beginning of the script is self-explanatory, and the parts which are below the dashed line are generic, and don't need to be modified, even if the model changes. The script relies on two primary techniques:

- a **dictionary** to store the matrix elements, where the keys (string labels) correspond directly to the Hamiltonian matrix element label (here it is possible to use "*", it is later replaced by "1");
- the use of the **exec** statement to execute code which has been assembled dynamically in the script itself - this demonstrates Python in its glory, as an interpretive language.

Usage examples

Band structure of silicon

An obvious test case is to compute the band structure of graphene in a simple script.

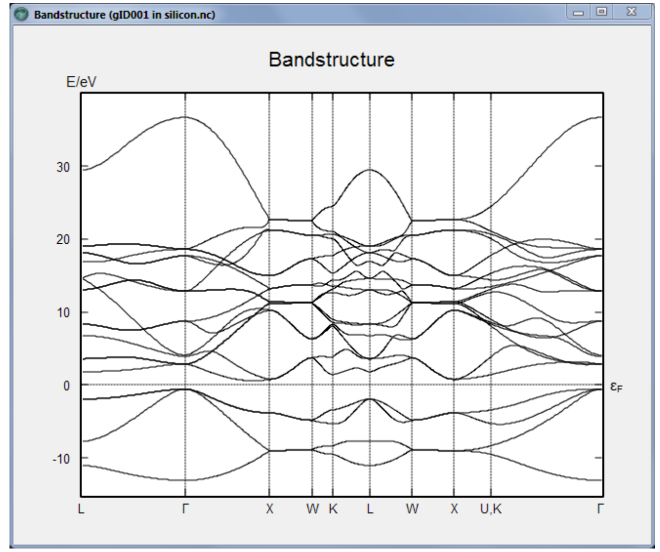
```
# Silicon
lattice = FaceCenteredCubic(5.4306*Angstrom)
elements = [Silicon, Silicon]
fractional_coordinates = [[ 0.    ,  0.    ,  0.    ],
                          [ 0.25 ,  0.25 ,  0.25 ]]
bulk_configuration = BulkConfiguration(
    bravais_lattice=lattice,
    elements=elements,
    fractional_coordinates=fractional_coordinates
    )

from SiBasisSet import basis_set

numerical_accuracy_parameters = NumericalAccuracyParameters(
    k_point_sampling=(9, 9, 9),
    )

calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )

bulk_configuration.setCalculator(calculator)
bulk_configuration.update()
nlsave('silicon.nc', bulk_configuration)

# Bandstructure
bandstructure = Bandstructure(
    configuration=bulk_configuration,
    route=[['L', 'G', 'X', 'W', 'K', 'L', 'W', 'X', 'U'],['K', 'G']],
    points_per_segment=100,
    bands_above_fermi_level=All
    )
nlsave('silicon.nc', bandstructure)
```

Download this script from the link just above, as well as the script `SiBasis.py` defining the Si basis set [14], and save them in the same directory. The script must be run from the command line, in this directory. It finishes in a few seconds, and produces a very accurate band structure of silicon.

Note

The energy bands in Si are degenerate at the U and K points, and by the special way we construct the route in the script, the band structure plot makes a discontinuous break in

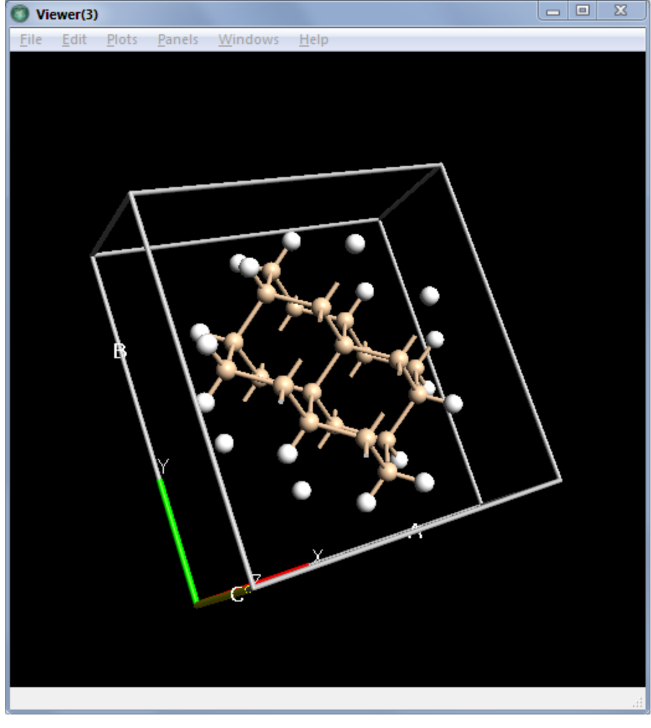
the route between these points, i.e. it goes from X to U, then jumps to K and continues to Γ.



Complex band structure of a narrow Si nanowire

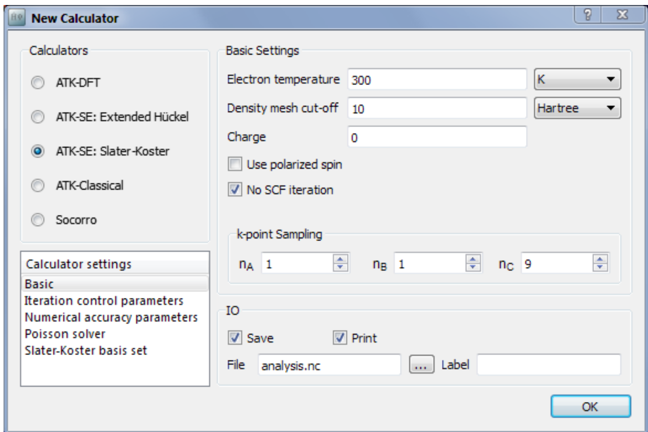
A more interesting example, where you also can see another interesting feature in ATK is to compute the complex band structure of a silicon nanowire. Here you also need to utilize the hydrogen saturation of the surface, to avoid states in the band gap.

The first step is to build the configuration shown in the figure below. The process for this is described in detail in a separate tutorial on **nanowires**, which you can find at the Tutorial page of the QuantumWise website. Follow the instructions there to construct the narrow Si nanowire with a hydrogen terminated surface.

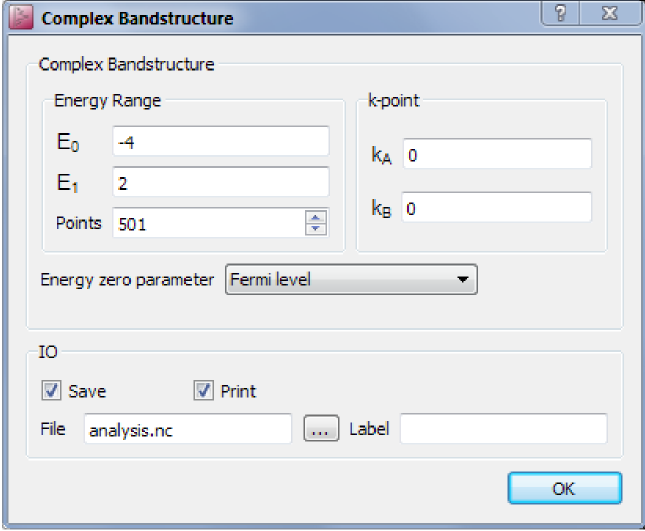


Send this structure to the Script Generator, and insert a New Calculator. Open the New Calculator and select the **ATK-SE: Slater-Koster (device)** calculator.

Select 9 k-points in the C direction. All other options can be left at their default values.

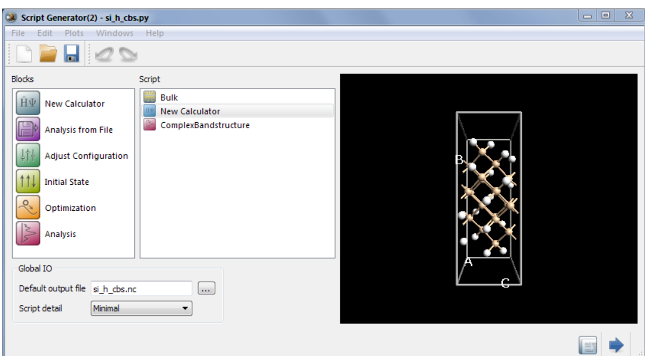


From Analysis, insert a **ComplexBandstructure** block. To cover the large band gap of such a narrow wire, set the energy range from -4 to 2 eV. It is also necessary to use a large number of energy points, to get a detailed plot of the complex band structure. Use 501 points.



Finally, set the output file name to something suitable, like `si_h_cbs.nc` and save the script as `si_h_cbs.py`.

Also download and save the Si-H basis set script `SiHBasis.py` [16] in the same directory.



Open the saved script in a text editor; you may use the built-in one in VNL, or your own favorite code editor.

To use the basis set defined above, locate the line

```
basis_set = DFTBDirectory("cp2k/nonscc/")
```

and replace it by

```
from SiHBasisSet import basis_set
```

Also, remove the line

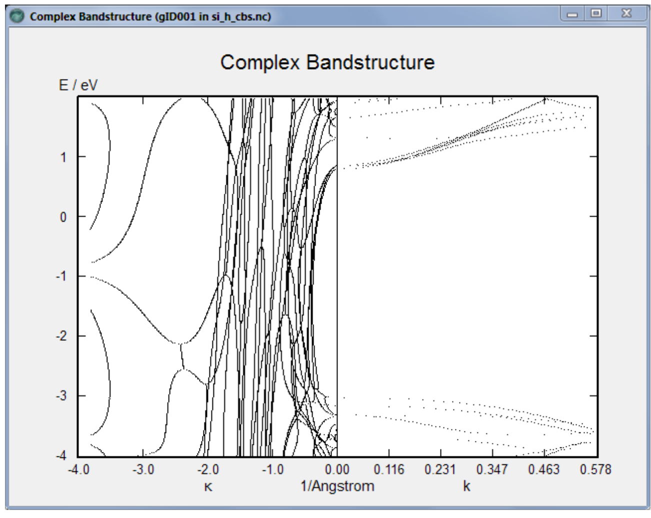
```
pair_potentials = DFTBDirectory("cp2k/nonscc/")
```

as well as the line referring to the **pair_potentials** variable in the SlaterKosterCalculator setup:

```
calculator = SlaterKosterCalculator(
    basis_set=basis_set,
    pair_potentials=pair_potentials,    ### remove this line
    numerical_accuracy_parameters=numerical_accuracy_parameters,
    )
```

The script you have now created needs to be run on the command line, in the directory where `SiHBasis.py` resides. It takes a few minutes to complete.

To inspect the results, locate the created file `si_h_cbs.nc` in VNL and plot the complex band structure.



Bibliography

[1] Y. Hancock, K. Saloriutta, A. Uppstu, A. Harju, and M. J. Puska. *J Low Temp Phys*, **153**, 393-398, 2008 (link).

[2] Y. Zheng, C. Rivas, R. Lake, K. Alam, T. B. Boykin, and G. Klimeck. *IEEE Trans Electron Devices*, **52**, 1097, 2005 (link).

[3] T. B. Boykin, G. Klimeck, and F. Oyafuso. *Phys. Rev. B*, **69**, 115201, 2004 (link).

[4] J.-M. Jancu, R. Scholz, F. Beltram, and F. Bassani. *Phys. Rev. B*, **57**, 6493, 1998 (link).

[5] W. A.. Harrison. *Electronic Structure and Properties of Solids*. Freeman (San Francisco). . 1980. .