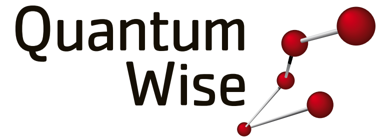


Exploring Graphene with NanoLanguage

Tutorial

Version 12.2.0

Exploring Graphene with NanoLanguage: Tutorial

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. Graphene builder modules 3
    NanoLanguage graphene builder commands 3
    Building an (8,0) graphene sheet 3
    Custom Builder for wrapping a graphene sheet into a carbon nano-tube 4
3. Twisting graphene structures 8
    Transmission spectrum of an ideal (narrow) graphene ribbon 8
    Conductance of a twisted ribbon 11
4. Going further with NanoLanguage 17
    Möbius nanoribbon 17
    Buckling a graphene sheet 19

Chapter 1. Introduction

The purpose of this tutorial is to present the functionality for building graphene structures with NanoLanguage, the Python interface to Atomistix ToolKit. A number of built-in functions allow for easy construction of graphene nanoribbons, sheets – and also carbon nanotubes, although that will be dealt with in a different tutorial.

The tutorial show how you can easily twist and stretch the structures. Using the Custom Builder component in Virtual NanoLab, the resulting scripts can be turned into plug-in modules where parameters are easily changed and the resulting structures visualized directly in the graphical user interface.

After building the structures it is possible to calculate their electronic properties with ATK. You will learn to automate calculations on large sets of structures using Python scripting.

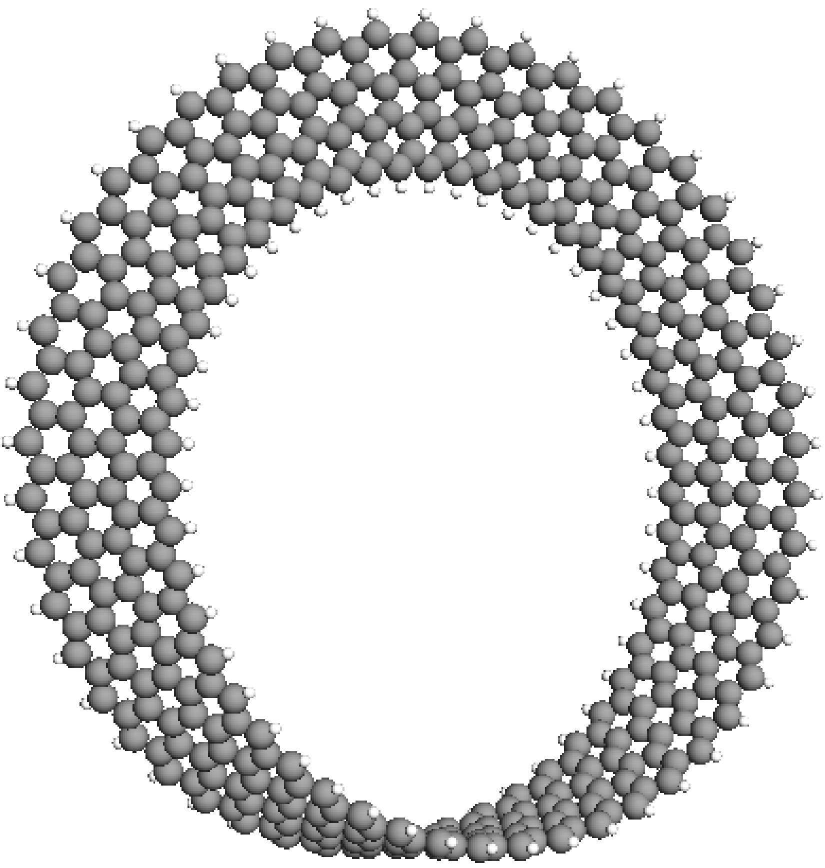


Figure 1.1: A graphene nanoribbon wrapped into a Möbius strip.

Note

VNL is used for setting up and analyzing the results. To familiarize yourself with VNL it is recommended to run the VNL Tutorial.

The **Atomistix ToolKit Semi Empirical** model is the computational engine used in this tutorial. A complete description of all the parameters, and in many cases a longer discussion about their physical relevance, can be found in the ATK Reference Manual.

Finally, you are ready to begin the Tutorial.

Chapter 2. Graphene builder modules

NanoLanguage graphene builder commands

NanoLanguage has three functions for building graphene-related structures.

`NanoSheet(n,m)`	Generates an (n,m) graphene sheet, with periodic boundary conditions in the y and z directions.
`NanoRibbon(n,m)`	Generates an (n,m) graphene ribbon with periodic boundary conditions in the z direction, and a hydrogen-passivated edge in the y direction.
`NanoTube(n,m)`	Generates an (n,m) carbon nanotube with periodic boundary conditions in the z direction.

- Each command takes as input the chiral indices (n,m), which must fulfill $n, m \geq 1$ and $n \geq m$.
- The commands return a **`BulkConfiguration`** with the corresponding structure.

Note

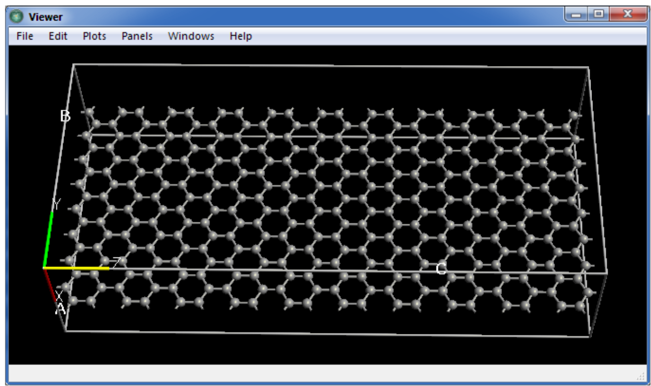
In the ATK Reference Manual you will find that the above commands also take a number of other parameters, related e.g. to the bond lengths in the generated structures. They can also be used to build e.g. boron-nitride nanotubes. This tutorial focus on carbon systems and you will use default bond lengths, which will give structures in close agreement with experimental data.

Building an (8,0) graphene sheet

The very simple script below builds a graphene (8,0) sheet, repeated 10 times in the z direction.

```
# generating (8,0) sheet repeated 10 times
sheet = NanoSheet(8,0)
sheet = sheet.repeat(1,1,10)
```

To view the structure **left**-click the `sheet80.py` link (only seen in the HTML edition of this tutorial) and save the script to your file system. Next, drop the file on the **3D Viewer** tool in VNL. You should now see the following structure:



All VNL tools take Python scripts as input, and the **3D Viewer** will automatically parse the script and display the first structure generated in it. A good methodology for building scripts is to first make sure that the script is valid syntactically by first running it on the command line

```
atkpython sheet80.py
```

If the script runs without any error messages, you may then drop the script on the **3D Viewer** to check that the geometry is set up as intended.

Custom Builder for wrapping a graphene sheet into a carbon nanotube

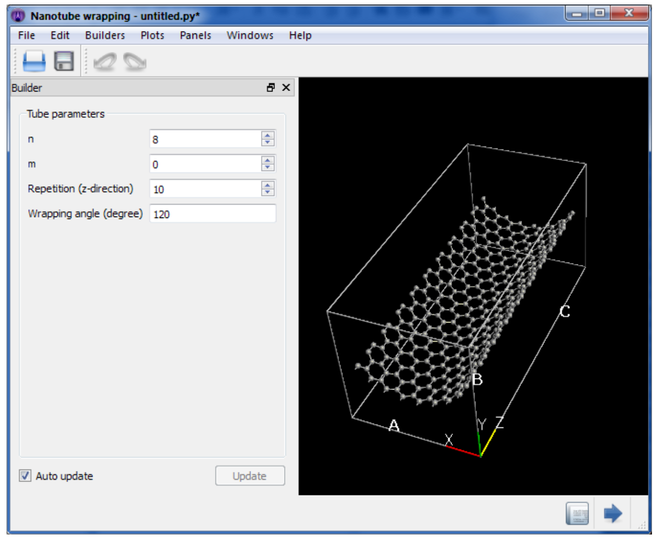
When you have made a script for building a particular structure, you might want to vary certain parameters to modify the structure. In this case you can conveniently turn the script into a **Custom Builder** with a GUI interface for tuning the parameters. This section illustrates how to do this, using the graphene sheet as an example, and wrapping it into a carbon nanotube.

Below is given the complete Custom Builder script for building the wrapped graphene sheet.

```
import math

def displacement(x, width, wrapping_angle):
    """
       Function for converting a nanosheet coordinate into a partly wrapped nanotube
       @param x : Coordinates of nanosheet atom
       @param width : Width of the nano-sheet
       @param wrapping_angle : maximum wrapping angle of the nanotube in radians
    """
    # calculate the radius of the incomplete wrapped tube
    radius = width/wrapping_angle
    # find the center of the tube, a full tube has radius = width/6.28
    center = numpy.array([ width/2.,width/3.+radius,0])
    # find the angle of the current atom
    angle = (x[1]-width/2.)/radius
    # return atom position of the wrapped atom
    return center + numpy.array([radius*math.sin(angle),
                                 -radius*math.cos(angle),x[2]])

def configuration(n, m, repetition, wrapping_angle):
    """
       Function for generating a (partly) wrapped nanotube
       @param n              : Chiral vector n
       @param m              : Chiral vector m
       @param repetition     : Repetition along z
       @param wrapping_angle : Wrapping angle of the nanotube (in degrees)
```

```
    """

    # find width of tube
    configuration = NanoSheet(n,m)
    lattice = configuration.bravaisLattice()
    tube_width = numpy.linalg.norm(lattice.primitiveVectors()[1].inUnitsOf(Angstrom))
    # now make new sheet with correct vacuum
    configuration = NanoSheet(n,m, vacuum = 0.5*tube_width * Angstrom)
    # repeat the structure in the z direction
    configuration = configuration.repeat(1,1,repetition)

    # now extract the information about the configuration
    lattice = configuration.bravaisLattice()
    elements = configuration.elements()
    cartesian_coordinates=configuration.cartesianCoordinates().inUnitsOf(Angstrom)

    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : displacement(c,tube_width,wrapping_angle/360*2.*math.pi)
    # apply the function to find new displaced coordinates
    new_coordinates = numpy.apply_along_axis(f,1,cartesian_coordinates)

    # generate the bulk configuration
    new_configuration = BulkConfiguration(
            bravais_lattice=lattice,
            elements=elements,
            cartesian_coordinates=new_coordinates * Angstrom
            )
    return new_configuration

# Instantiate the builder object and choose our title
builder = Builder()
builder.title('Nanotube wrapping')

# Set the configuration generator
builder.setConfigurationGenerator(configuration)

# Tube parameters group
builder.newGroup('Tube parameters')
builder.integer( 'n', 8, 'n', min=1, max=1000)
builder.integer( 'm', 0, 'm', min=0, max=1000)
builder.integer( 'repetition', 10, 'Repetition (z-direction)', min=1, max=1000)
builder.double(  'wrapping_angle',  120.0, 'Wrapping angle (degree)',     min=0.0)
```

Save the script to your file system. Then, drop the file on the **Custom Builder** tool in VNL. In result, the Custom Builder should open with an interface as shown below.



Try to increase the wrapping angle to 360 degrees, and see that the sheet now is wrapped completely into a nanotube. You may also change the chiral indices of the tube on the fly, as well as the repetition along the tube axis.

Details of the TubeBuilder script

This section discuss some of the details behind the TubeBuilder script, to show you how to create your own Custom Builder.

The builder itself is initialized with a call to the constructor **`Builder()`**, and you may also give a title to the builder.

```
builder.title('Nanotube wrapping')
```

The central part of any Custom Builder script is the function which generates the geometry. In this case, this function is relatively complicated, but the basic principle is simple: the input arguments to the function are the parameters which the user controls in the Custom Builder interface.

```
def configuration(n, m, repetition, wrapping_angle):
    """
       Function for generating a (partly) wrapped nanotube
       @param n              : Chiral vector n
       @param m              : Chiral vector m
       @param repetition     : Repetition along z
       @param wrapping_angle : Wrapping angle of the nanotube (in degrees)
    """
```

The function returns the generated geometry, and this is what will be displayed in the 3D window.

```
    new_configuration = BulkConfiguration(
            bravais_lattice=lattice,
            elements=elements,
            cartesian_coordinates=new_coordinates * Angstrom
            )
    return new_configuration
```

This function must then be attached to the builder so it can be called whenever the parameters are updated.

```
# Set the configuration generator
builder.setConfigurationGenerator(configuration)
```

Finally, you must define which parameters the user can control in the Custom Builder interface, and for each parameter specify its type and certain parameters like the default value etc. For a better overview of the parameters, they can be grouped.

```
# Tube parameters group
builder.newGroup('Tube parameters')
builder.integer( 'n', 8, 'n', min=1, max=1000)
builder.integer( 'm', 0, 'm', min=0, max=1000)
builder.integer( 'repetition', 10, 'Repetition (z-direction)', min=1, max=1000)
builder.double(  'wrapping_angle',  120.0, 'Wrapping angle (degree)',     min=0.0)
```

Each parameter is defined by a variable name (a string; these must match the variables passed to the **`configuration()`** function), a default value, and a label. The label will be shown in the Custom Builder interface. Besides integers and doubles, as used here, the parameter can also be an element, a Boolean (to turns things on/off), a drop-down list, or a configuration drop-zone, as you can see in many of the predefined builders in VNL.

For more details about making Custom Builder scripts, see the corresponding section in the Reference Manual..

Chapter 3. Twisting graphene structures

The Custom Builder tool in Virtual NanoLab comes with a set of built-in graphene and nanotube builders. These tools are based on the corresponding NanoLanguage commands presented in the previous chapter. In this chapter you will learn how to modify structures built with these tools into more complex graphene nanostructures, and how scripting can be used to automate electronic structure calculations of these systems.

Transmission spectrum of an ideal (narrow) graphene ribbon

You will first compute the transport properties of a perfect (1,1) graphene nanoribbon. This will be used as a reference that can be compared with the distorted structure you will make later. ATK contains a special functionality for computing the transmission spectrum of a perfect periodic structure, which you will use.

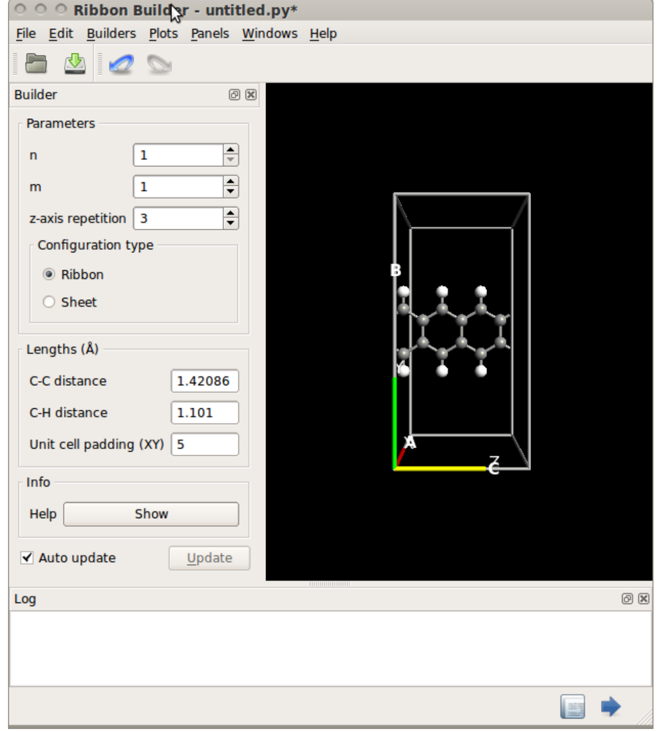
You will use the **Custom Builder** tool to build the ribbon.

1. Open the **Custom Builder** tool from the main toolbar
2. Select, from the menu, the `Builders` → `Graphene Ribbon` builder
3. Set (n,m) to (1,1) and use a repetition of 3 to make the structure sufficiently long in the C-direction to be a valid electrode cell.

Note

In order to correctly calculate the Transmission Spectrum from a bulk system, the bulk system must qualify as a valid device electrode. This requires that the C axis is perpendicular to the A,B plane, and the C-axis is sufficiently long that the atoms in the unit cell only have matrix elements with atoms in the nearest neighbour cells along C. Typically this condition is full filled if the C-vector is longer than 7 Å.

You should now see the following structure



Next, transfer the structure to the **Script Generator** using the “Send to” icon in the lower right-hand corner.

Double-click the **New Calculator** icon and observe that a icon is added to the middle panel. Double-click this icon. This launches a window for setting the parameters of the calculator.

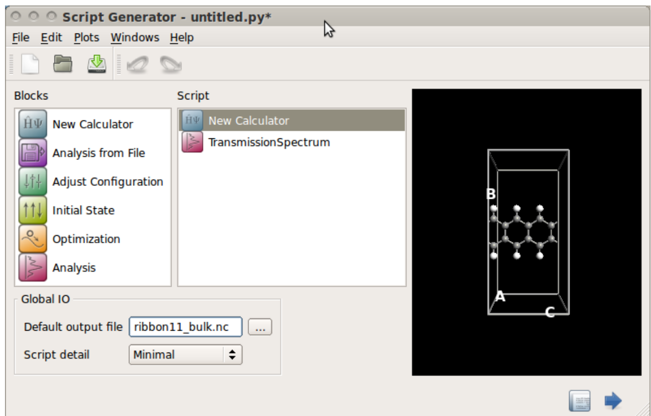
Select the **Extended Huckel** calculator, and set the number of **n**$_C$ points to 100. Then, under **Huckel basis set**, select

- C: `Cerda.Carbon [graphite]`
- H: `Cerda.Hydrogen [C2H4]`

Double-click the icon and select **TransmissionSpectrum** from the menu.

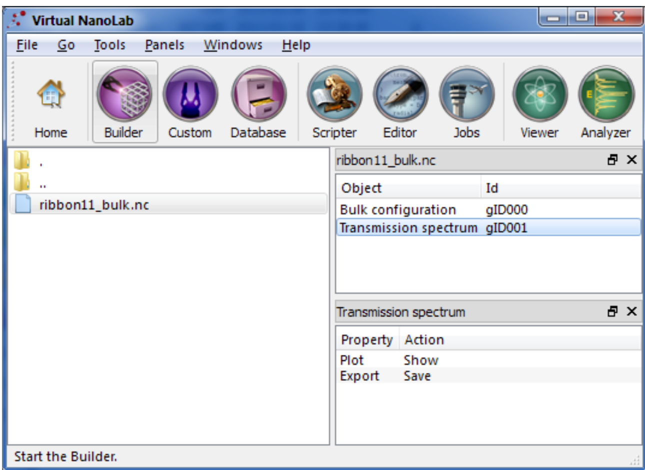
Finally, under **Global IO**, set the **Default output file** to `ribbon11_bulk.nc`

The Script Generator should now look like this



Now run the file using the "Send to" icon to transfer the script to the **Job Manager** . There, push the "Start" button.

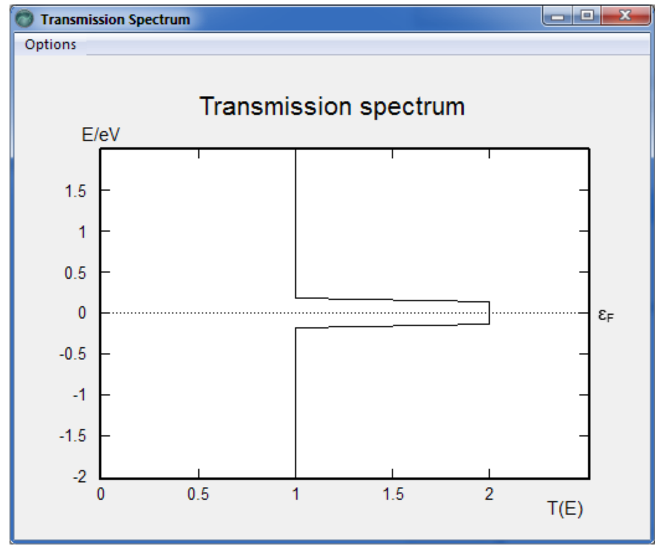
When the calculation is finished (it will run extremely fast), locate the output file `ribbon11_twist0_nscf.nc` in the VNL file browser window. Select the file and notice that the content of the file is displayed in the panel to the right.



Note

The NetCDF file will reside in the directory where you launched VNL from; click the **Home** icon on the toolbar to go to this directory. For more precise control of the location of the file, specify the full path of the file in the Script Generator before running the calculation.

Left-click the "Transmission spectrum" entry in the **Result Browser** panel; this will give a list of actions in the **Result Viewer** panel below. Select "Show Plot" to visualize the results.



The transmission coefficient takes integer values in this perfect one-dimensional system. At the Fermi energy there are two channels. In the next section you will investigate how the transmission is affected when the ribbon is distorted, more specifically *twisted*.

Conductance of a twisted ribbon

Twisting the ribbon using a custom builder

The script below defines a Custom Builder for twisting a one-dimensional system. Save the script to your file system, and drop the file on the **Custom Builder** tool .

```
from NL.Math.Utilities import rotationMatrix
import math
from NanoLanguage import *

def displacement(x, rotation_angle_per_z, rotation_axis,
                 rotation_axis_center, z_start, z_end):
    """
       Function for twisting a 1-d structure
       @param x : Coordinates of 1-d structure
       @param rotation_angle_per_z : size of twist in angle/length
       @param rotation_axis : axis to apply twist along
       @param rotation_axis_center : center of the rotation axis
       @param z_start : z value for starting the twist
       @param z_end : z value for ending the twist
    """

    # do not twist for z > z_end
    z = x[2]
    z = min(z,z_end)
    # do not twist for z < z_start
    z = z - z_start
    z = max (z,0.0)
    # find twist angle
    theta = z*rotation_angle_per_z
    # calculate the rotation matrix
    rotation_matrix = rotationMatrix(theta, *rotation_axis)
    # apply rotation
    return rotation_axis_center+ numpy.dot(rotation_matrix, x - rotation_axis_center)
```

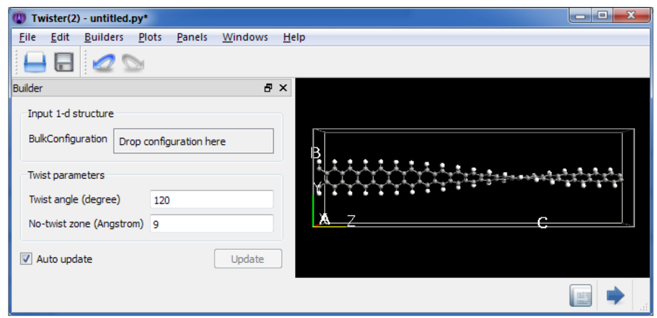
```
def configuration_twist(bulk_configuration, twist_angle, z_padding):
    """
       Function for generating a twisted 1-d structure
       @param bulk_configuration : the structure to be twisted
       @param twist_angle : the twist of the entire structure in degrees
       @param z_padding : the structure is not twisted for z < z_padding or z > length-z_padding
    """

    # Make sure we got a BulkConfiguration
    if not isinstance(bulk_configuration, BulkConfiguration):
        return None

    # get properties of the bulk configuration
    lattice = bulk_configuration.bravaisLattice()
    elements = bulk_configuration.elements()
    cartesian_coordinates=bulk_configuration.cartesianCoordinates().inUnitsOf(Angstrom)

    # calculate the length of the 1-d structure
    z_length = numpy.linalg.norm(lattice.primitiveVectors()[2].inUnitsOf(Angstrom))

    # calculate twist parameters
    rotation_angle_per_z = twist_angle/180. * math.pi /(z_length-2*z_padding)
    rotation_axis = numpy.array([0,0,1])
    rotation_axis_center = numpy.sum(cartesian_coordinates,axis=0)/len(cartesian_coordinates)

    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : displacement(c, rotation_angle_per_z, rotation_axis,
                                rotation_axis_center, z_padding,z_length-z_padding)
    # apply the function to find new displaced coordinates
    new_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)

    new_configuration = BulkConfiguration(
            bravais_lattice=lattice,
            elements=elements,
            cartesian_coordinates=new_coordinates * Angstrom
            )

    return new_configuration

# Instantiate the builder object and choose our title
builder = Builder()
builder.title('Twister')

# Set the configuration generator
builder.setConfigurationGenerator(configuration_twist)

# define the input configuration
builder.newGroup('Input 1-d structure')
builder.configuration('bulk_configuration', 'BulkConfiguration')

# Set up twist parameters
builder.newGroup('Twist parameters')
builder.double(  'twist_angle',  120.0, 'Twist angle (degree)',     min=0.0)
builder.double(  'z_padding',  9.0, 'No-twist zone (Angstrom)',     min=0.0)
```

Return to the Custom Builder used to build the (1,1) graphene ribbon in the section above and change the repetition to 20.

Tip

From the `Windows` menu at the top of each tool, you have quick access to all other open tools. When a structure is sent to another tool, using the “Send to” menu, the original window is minimized, not closed.

Drag and drop the (1,1,20) graphene ribbon geometry onto the **`BulkConfiguration`** drop zone of the **Twister** Custom Builder. You should now see the following:



A certain part of the system to the left and right of the structure are not twisted; the length of the non-twisted part is specified by the value of **`No-twist zone`**. The remainder of the structure is twisted by the **`Twist angle`**.

The implementation of the Twister Custom Builder follows the same framework as the Nanotube Wrapper in the previous chapter. The main functionality is implemented in the function **`displacement`** which introduces the twist using the built-in NanoLanguage command **`rotationMatrix`** which applies a rotation about an axis, in this case the z-direction.

Conductance as a function of the twist angle

In this section you will calculate the conductance of the graphene ribbon as a function of the twist angle. Instead of building the structures and performing the calculations one by one, by hand, you will use scripting to perform all calculations in one go.

Below is given the script for performing the calculation. It contains a lot of useful things, and you are recommended to study it carefully if you are interested in learning more about how scripting in NanoLanguage can be used in a very powerful way to automate tasks.

```
import Twister as Twister
import NL.CommonConcepts.Configurations.DeviceFromBulk as DeviceFromBulk

# Calculator settings
basis_set = [
    CerdaHuckelParameters.Carbon_graphite_Basis,
    CerdaHuckelParameters.Hydrogen_C2H4_Basis,
    ]

calculator = DeviceHuckelCalculator(
    basis_set=basis_set,
    iteration_control_parameters=NonSelfconsistent,
    )

# Build (1,1) ribbon, repeat 20 times along Z
ribbon = NanoRibbon(1,1).repeat(1,1,20)
# Pick the electrode period that is about 7 Angstrom
d = DeviceFromBulk.repetitionMatch(ribbon)
electrode_period = d[(d>6.5) & (d<7.5)][0]*Angstrom

# Loop through angles
for angle in numpy.linspace(0, 400.,11):
    # Twist the ribbon
    twisted_configuration = Twister.configuration_twist(ribbon,angle,9.0)

    # Use DeviceFromBulk custom builder to make configuration
    twisted_device = DeviceFromBulk.deviceFromBulk(
        electrode_period,electrode_period,twisted_configuration
        )

    # Attach the calculator to the configuration
```

```
    twisted_device.setCalculator(calculator())

    # Calculate transmission spectrum
    transmission_spectrum = TransmissionSpectrum(
        configuration=twisted_device,
        energies=numpy.linspace(-0.1,0.1,20)*eV,
        kpoints=MonkhorstPackGrid(1,1),
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=KrylovSelfEnergy(),
        )

    # Save the results
    nlsave('ribbon11_twist.nc', transmission_spectrum, object_id='trans'+str(angle))
```

Save the script to your file system, in the same directory you saved `Twister.py`, and run the calculation by opening a terminal window in the directory where the scripts reside, and give the command

```
atkpython ribbon11_twist.py
```

While the calculation runs, inspect the parts of the script.

One part of the script sets up the calculation method. It uses a non self-consistent Huckel method, and change the basis set:

```
# Calculator settings
basis_set = [
    CerdaHuckelParameters.Carbon_graphite_Basis,
    CerdaHuckelParameters.Hydrogen_C2H4_Basis,
    ]

calculator = DeviceHuckelCalculator(
    basis_set=basis_set,
    iteration_control_parameters=NonSelfconsistent,
    )
```

The other part of the script sets up the geometry. A lot of the functionality in VNL is also available from within the scripting language, and you can use methods from the Builder and Custom builders (the built-in one for the nanoribbon, and our own "Twister"). So, the script actually mimic the work flow you could do manually in VNL, by drag-and-drop, but in an automated way.

First, the script creates a (1,1) nanoribbon, repeated 20 times along Z.

```
ribbon = NanoRibbon(1,1).repeat(1,1,20)
```

It use a convenient function to determine the periodicty of the ribbon. The second line selects a repetition which is about 7 Angstrom long:

```
d = DeviceFromBulk.repetitionMatch(ribbon)
electrode_period = d[(d>6.5) & (d<7.5)][0]*Angstrom
```

Then the script loops over the twist angles, apply the twist to the ribbon, and convert it into a device structure:

```
# Loop through angles
for angle in numpy.linspace(0, 400.,11):
```

```
    # Twist the ribbon
    twisted_configuration = Twister.configuration_twist(ribbon,angle,9.0)

    # Use DeviceFromBulk custom builder to make configuration
    twisted_device = DeviceFromBulk.deviceFromBulk(
        electrode_period,electrode_period,twisted_configuration
        )
```

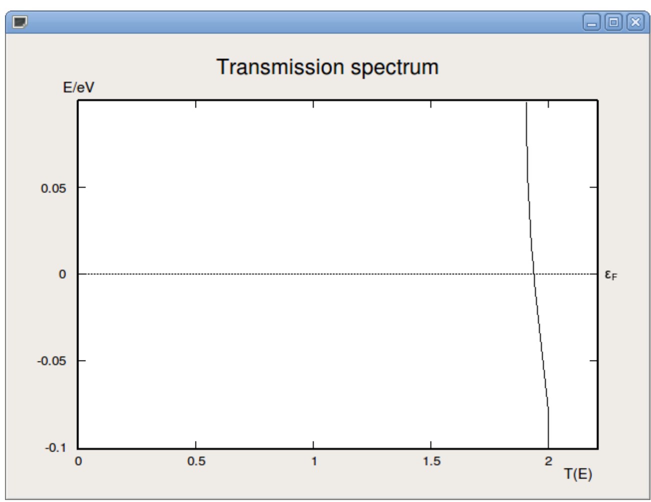
Finally the calculator is attached to the configuration, and the transmission spectrum is computed. The result is saved in a NetCDF file, and by using a "label" each spectrum can be identified by the angle:

```
    # Attach the calculator to the configuration
    twisted_device.setCalculator(calculator())

    # Calculate transmission spectrum
    transmission_spectrum = TransmissionSpectrum(
        configuration=twisted_device,
        energies=numpy.linspace(-0.1,0.1,20)*eV,
        kpoints=MonkhorstPackGrid(1,1),
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=KrylovSelfEnergy(),
        )

    # Save the results
    nlsave('ribbon11_twist.nc', transmission_spectrum, object_id='trans'+str(angle))
```

Note

The use of “()” after **calculator** has the effect of generating a *copy* of the **nscf_calculator**, rather than just a reference to it. This is a safe method to use a calculator from a previous calculation in a new calculation.

Once the calculation has finished you can inspect the file `ribbon11_twist.nc` to see the results. Below is shown the transmission coefficient for an applied twist of 360 degrees.

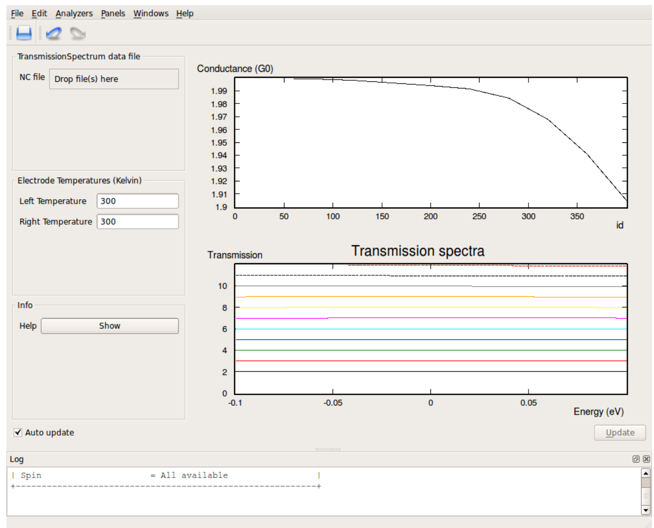


The transmission coefficient is now less than 2 at the Fermi level – the ribbon is no longer a perfect quantum conductor. Remember that the ribbons extends to infinity to both sides; however, the electrodes are created from the non-twisted part of the ribbon, and are therefore perfectly flat ribbons. The twist thus represents a source of scattering, since it breaks the translational symmetry of the electrodes along the ribbon.

Analyzing the results with a Custom Analyzer

To get an overview of the calculations performed in the previous section it is convenient to use the Custom Analyzer tool. Open the **Custom Analyzer** tool and from the menu select `Analyzers` → `Conductance`

Now drag and drop the file `ribbon11_twist.nc` onto the "NC file" drop zone. You should now see the following



To learn more about Custom Analyzers and how to make your own analyzers, you are referred to the Reference Manual.

Chapter 4. Going further with NanoLanguage

The previous chapters have illustrated the strength of combining NanoLanguage and the graphical user interface by using custom – and built-in – builders and analyzer tools. Below is provided a few examples on how these ideas can be expanded to construct some more exotic graphene structures.

Möbius nanoribbon

The following script applies a wrapping and a twist to a nanoribbon, turning it into a Möbius strip.

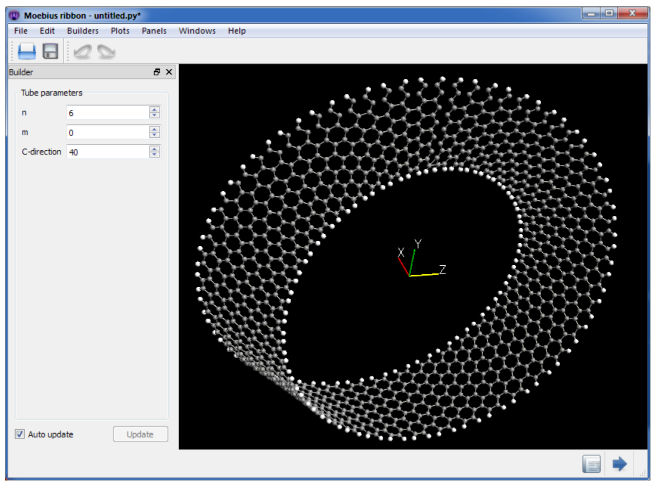
```
import Twister as Twister
import math

def displacement(x, width, wrapping_angle):
    """
       Function for converting a nanosheet coordinate into a partly wrapped nanotube
       @param x : Coordinates of nanosheet atom
       @param width : Width of the nano-sheet
       @param wrapping_angle : maximum wrapping angle of the nanotube in radians
    """
    # calculate the average radius of the incomplete wrapped tube
    radius = width/wrapping_angle
    # find the angle of the current atom
    angle = (x[2]-width/2.)/radius
    # calculate the radius of the current atom
    atom_radius = radius+x[1]

    # return atom position of the wrapped atom
    return numpy.array([x[0], atom_radius*math.cos(angle),atom_radius*math.sin(angle)])

def configuration(n, m, repetition):
    """
       Function for generating a moebius molecule
       @param n : Chiral vector index
       @param m : Chiral vector index
       @param repetition : Repetition along z
    """

    # build n,m ribbon
    ribbon = NanoRibbon(n,m)
    ribbon = ribbon.repeat(1,1,repetition)

    # get properties of the ribbon
    lattice = ribbon.bravaisLattice()
    elements = ribbon.elements()
    cartesian_coordinates=ribbon.cartesianCoordinates().inUnitsOf(Angstrom)
```

```
    # calculate the length of the 1-d structure
    z_length = numpy.linalg.norm(lattice.primitiveVectors()[2].inUnitsOf(Angstrom))

    # calculate twist parameters
    rotation_angle_per_z =  math.pi /z_length
    rotation_axis = numpy.array([0,0,1])
    rotation_axis_center = numpy.sum(cartesian_coordinates,axis=0)/len(cartesian_coordinates)

    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : Twister.displacement(c, rotation_angle_per_z, rotation_axis,
                                        rotation_axis_center, 0.,z_length)
    # apply the function to find new displaced coordinates
    cartesian_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)
    cartesian_center = numpy.sum(cartesian_coordinates,axis=0)/len(cartesian_coordinates)
    cartesian_coordinates = cartesian_coordinates - cartesian_center


    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : displacement(c, z_length,2.0*math.pi)
    # apply the function to find new displaced coordinates
    cartesian_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)

    return MoleculeConfiguration(
            elements=elements,
            cartesian_coordinates=cartesian_coordinates * Angstrom
            )

# Instantiate the builder object and choose our title
builder = Builder()
builder.title('Moebius ribbon')

# Set the configuration generator
builder.setConfigurationGenerator(configuration)

# Tube properties group
builder.newGroup('Tube parameters')
builder.integer( 'n', 6, 'n', min=1, max=1000)
builder.integer( 'm', 0, 'm', min=0, max=1000)
builder.integer( 'repetition', 40, 'C-direction', min=1, max=1000)
```

Important

You must save both this script and `Twister.py` in the VNL "home" directory.

Then, drop it onto the Custom Builder tool and you should see the following:



Play around with the parameters and enjoy!

Buckling a graphene sheet

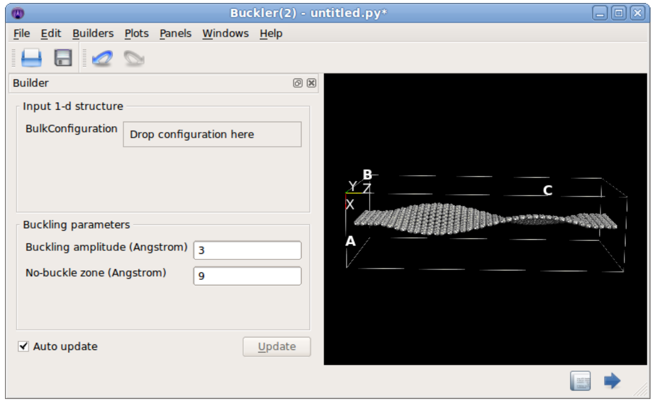
The following script applies a wobbling, or buckling, in the x-direction to a one-dimensional structure.

```
import math

def displacement(x, width, length, delta, z_start, z_end):
    """
       Function for converting a nanosheet coordinate into a wobbled graphene nano-sheet
       @param x      : Coordinates of an atom
       @param width  : wave length of the buckling in the y-direction
       @param length : wave length of the buckling in the z-direction
       @param delta  : Buckling in units of Angstrom
       @param z_start : z value for starting the buckling
       @param z_end : z value for ending the buckling
    """
    # do not buckle for z > z_end
    z = x[2]
    z = min(z,z_end)
    # do not buckle for z < z_start
    z = z - z_start
    z = max (z,0.0)
    # find the buckling phase of the current atom in the y direction
    phase_y = 2.*math.pi*(x[1]-width/2.)/width
    # find the buckling phase of the current atom in the z direction
    phase_z = 2.*math.pi*(z-length/2.)/length

    # return atom position of the wrapped atom
    return x + numpy.array([delta* math.sin(phase_y)*math.sin(phase_z),0.0,0.0])

def configuration(bulk_configuration, buckling_amplitude, z_padding):
    """
       Function for generating a buckled 1-d structure
       @param bulk_configuration : the structure to be buckled
       @param buckling_amplitude : the buckling of the structure applied in the x direction
       @param z_padding : the structure is not buckled for z < z_padding or z > length-z_padding
    """

    # Make sure we got a BulkConfiguration
    if not isinstance(bulk_configuration, BulkConfiguration):
        return None

    # get properties of the bulk configuration
    lattice = bulk_configuration.bravaisLattice()
    elements = bulk_configuration.elements()
    cartesian_coordinates=bulk_configuration.cartesianCoordinates().inUnitsOf(Angstrom)

    # calculate the length of the 1-d structure
    y_length = numpy.linalg.norm(lattice.primitiveVectors()[1].inUnitsOf(Angstrom))
    z_length = numpy.linalg.norm(lattice.primitiveVectors()[2].inUnitsOf(Angstrom))


    # define a function of one variable, f(c),  for displacing the atoms
    f = lambda c : displacement(c, y_length, z_length-z_padding*2.,
                            buckling_amplitude, z_padding,z_length-z_padding)
    # apply the function to find new displaced coordinates
    new_coordinates = numpy.apply_along_axis(f, 1, cartesian_coordinates)

    new_configuration = BulkConfiguration(
            bravais_lattice=lattice,
            elements=elements,
            cartesian_coordinates=new_coordinates * Angstrom
            )
```

```
    return new_configuration

# Instantiate the builder object and choose our title
builder = Builder()
builder.title('Buckler')

# Set the configuration generator
builder.setConfigurationGenerator(configuration)

# define the input configuration
builder.newGroup('Input 1-d structure')
builder.configuration('bulk_configuration', 'BulkConfiguration')

# Set up buckle parameters
builder.newGroup('Buckling parameters')
builder.double(  'buckling_amplitude',  3.0, 'Buckling amplitude (Angstrom)',     min=0.0)
builder.double(  'z_padding',  9.0, 'No-buckle zone (Angstrom)',     min=0.0)
```

Save the script and drop the file onto the Custom Builder tool. You will then see a tool similar to the Twister from the previous chapter. Next open a “Graphene Ribbon” Custom Builder tool, and make a (10,0,20) graphene sheet (not ribbon, this time, but a 2D sheet). In addition, increase the unit cell X/Y padding to 12 Angstrom.

Drop the sheet onto the drop zone of the “Buckler” and you will obtain a buckled sheet.



Tip

The structure looks best if you turn off the bonds, and use the covalent radius to plot the atoms.