python™
www.python.org

B-factor Plot in Python

By Prof. Walter F. de Azevedo Jr.

© 2016 Dr. Walter F. de Azevedo Jr.

python™
www.python.org

The atomic scattering factor (f) of an atom depends on the electron density and can be found in the International Tables for X-ray Crystallography (1974, Vol. III). The thermal motion of the atoms affects this factor, and it is called B factor. In the simple case in which the components of vibration are the same in all directions, the vibration is called isotropic and the atomic scattering factor is

$$f = f_0 \exp\left[-B\left(\sin^2\theta_{hkl}\right)/\lambda^2\right] \qquad \text{(equation 1)}$$

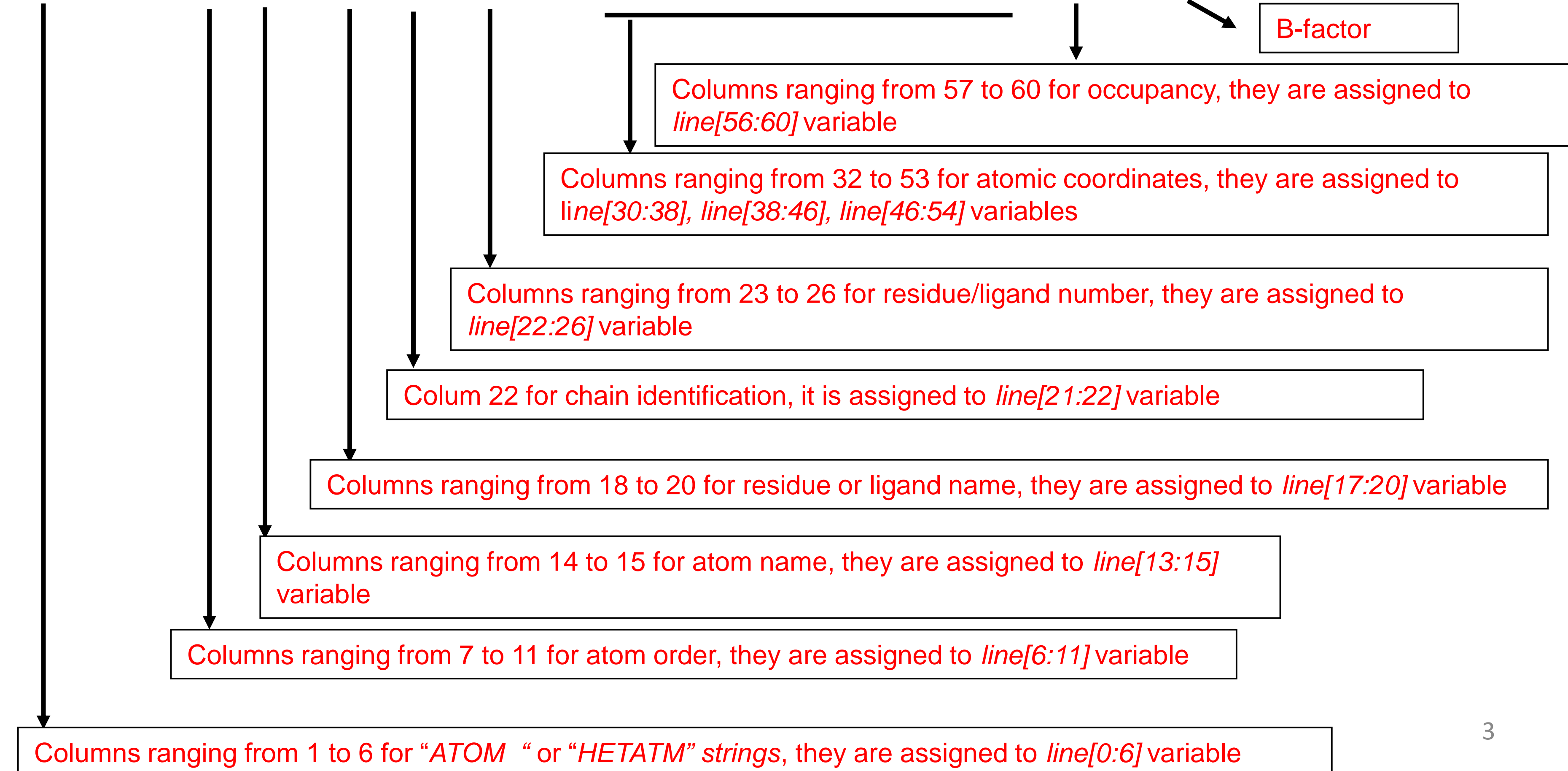
where θ_{hkl} is the scattering angle, f_o is the scattering power of a given atom for a given reflection, λ is the wavelength, and B is related to the mean square amplitude ($\overline{u^2}$) of atomic vibration by

$$B = 8\pi^2\overline{u^2} \qquad \text{(equation 2)}$$

In the PDB files, B, generally called B facfor, is shown in lines that start with “ATOM “ or “HETATM”.

python™ www.python.org

Below we have a typical line for atomic coordinates in a Protein Data Bank (PDB) file. Each field brings a specific information related to the atoms in the structure. These lines starts either with "ATOM " or "HETATM" strings. Additional information for each field is given red. The B-factor, also called B-value, comes right after occupancy.



B-factor plot from a PDB file

Program: *bfactor_plot.py*

Abstract
Program to generate B-factor plot using information from a PDB (Berman, Westbrook, Feng *et al*. 2000; Berman, Battistuz, Bhat *et al*. 2002; Westbrook *et al*., 2003) file. The program generates a plot where we have the mean B-factor per residue in a PDB file. In the plot we have three lines, one for all atoms in the residue, the second for main-chain atoms and the last for side-chain atoms. Only protein atoms are considered for the plot. In the plot, x-axis is for residue number and y-axis is for mean B-factor in $Å^2$.

In the main program we call *read_PDB*() function, which reads a PDB file and returns a list with all lines that starts with "ATOM ". To calculate mean B-factors, we call *calc_prot_bfactors()* function, which returns a list with all B-factors, then we create a list with the legends to be used in plot *(list_legends*), and finally call *plot_mult_array()* function. Below we have the main program.

```
# Main program
# Calls function to read PDB
my_list_of_atoms = read_PDB()
# Calls function to calculate mean B-factor for each residue (protein atoms only)
my_bfactors = calc_prot_bfactors(my_list_of_atoms)
# Creates a list of legends
list_legends = ["All","Main-chain","Side-chain"]
# Calls function to plot B-factor
plot_mult_array(my_bfactors,"Residue Number","B-factor(A**2)",list_legends)
```

Initially, we set up an empty list, name *my_atoms.* Then we call *read_file_name()* function, which reads the input file name. At this point, we open this file, and assigns its content to a file object, named *my_fo.*

```
def read_PDB():
    """Function to read a PDB file"""
    # Sets initial list
    my_atoms = []
    # Calls function read_file_name()
    input_file_name = read_file_name()
    # Opens PDB file
    my_fo = open(input_file_name,"r")
    # Looping through PDB file content
    for line in my_fo:
        if line[0:6] == "ATOM  " or line[0:6] == "HETATM":
            my_atoms.append(line)
    my_fo.close()
    return my_atoms
```

python™
www.python.org

After we have for loop, which loops through *my_fo* and assigns all coordinate lines to *my_atoms* list.

```
def read_PDB():
    """Function to read a PDB file"""
    # Sets initial list
    my_atoms = []
    # Calls function read_file_name()
    input_file_name = read_file_name()
    # Opens PDB file
    my_fo = open(input_file_name,"r")
    # Looping through PDB file content
    for line in my_fo:
        if line[0:6] == "ATOM  " or line[0:6] == "HETATM":
            my_atoms.append(line)
    my_fo.close()
    return my_atoms
```

python™
www.python.org

Then, we close the file and return *my_atoms* list..

```
def read_PDB():
    """Function to read a PDB file"""
    # Sets initial list
    my_atoms = []
    # Calls function read_file_name()
    input_file_name = read_file_name()
    # Opens PDB file
    my_fo = open(input_file_name,"r")
    # Looping through PDB file content
    for line in my_fo:
        if line[0:6] == "ATOM  " or line[0:6] == "HETATM":
            my_atoms.append(line)
    my_fo.close()
    return my_atoms
```

python™
www.python.org

This funciton is quite simple, it only reads the input file name and returns it, as shown below.

```
def read_file_name():
    """Function to read file name"""
    my_file = input("Type input file name => ")

    # Returns input file name
    return my_file
```

Function: *calc_prot_bfactor()*

```
def calc_prot_bfactors(list_of_atoms_in):
    """Function to calculate average B-factor for each residue in a protein PDB"""
    import numpy as np
    list_mc =  ["CA","C ","O ","N ","OX"] # List of main-chain atoms
    # Sets lists for all, main-chain and side-chain atoms
    all_atoms = []
    mc_atoms = []
    sc_atoms = []
    for line in list_of_atoms_in:       # looping through all atoms in the list
        if line[0:6] == "ATOM  ":
            all_atoms.append(line)      # Picks all atoms
            if line[13:15] in list_mc:  # Picks main-chain atoms
                mc_atoms.append(line)
            else:
                sc_atoms.append(line)   # Picks side-chain atoms
    nres, bf_all = calculate_bfactors(all_atoms) # Calls function calculate_bfactors()
    nres, bf_mc = calculate_bfactors(mc_atoms)   # Calls function calculate_bfactors()
    nres, bf_sc = calculate_bfactors(sc_atoms)   # Calls function calculate_bfactors()
    columns = 4
    rows = len(nres)
    bf = np.array([[0]*columns]*rows,float) # Sets initial matrix as a NumPy array
    bf[:,0] = nres               # Gets residue number
    bf[:,1] = bf_all[:rows]      # Gets all atom B-factors
    bf[:,2] = bf_mc[:rows]       # Gets main-chain atom B-factor
    bf[:,3] = bf_sc[:rows]       # Gets side-chain atom B-factor
    return bf
```

Here we have the code for *calc_prot_bfactors()* function. We have a list, named *list_mc*, for all main-chain atoms. This list is used in a for loop to filter information in a PDB file, in order to split in main-chain (*mc_atoms* list), side-chain (*sc_atoms* list), and all atoms (*all_atoms* list). These lists are passed to *calculate_bfactors* function, which returns a list of residue numbers in each list and an array with mean B-factor for each residue.

python™ www.python.org

Function: *calc_prot_bfactor()*

```
def calc_prot_bfactors(list_of_atoms_in):
    """Function to calculate average B-factor for each residue in a protein PDB"""
    import numpy as np
    list_mc =  ["CA","C ","O ","N ","OX"] # List of main-chain atoms
    # Sets lists for all, main-chain and side-chain atoms
    all_atoms = []
    mc_atoms = []
    sc_atoms = []
    for line in list_of_atoms_in:        # looping through all atoms in the list
        if line[0:6] == "ATOM  ":
            all_atoms.append(line)       # Picks all atoms
            if line[13:15] in list_mc:   # Picks main-chain atoms
                mc_atoms.append(line)
            else:
                sc_atoms.append(line)    # Picks side-chain atoms
    nres, bf_all = calculate_bfactors(all_atoms) # Calls function calculate_bfactors()
    nres, bf_mc = calculate_bfactors(mc_atoms)   # Calls function calculate_bfactors()
    nres, bf_sc = calculate_bfactors(sc_atoms)   # Calls function calculate_bfactors()
    columns = 4
    rows = len(nres)
    bf = np.array([[0]*columns]*rows,float) # Sets initial matrix as a NumPy array
    bf[:,0] = nres              # Gets residue number
    bf[:,1] = bf_all[:rows]     # Gets all atom B-factors
    bf[:,2] = bf_mc[:rows]      # Gets main-chain atom B-factor
    bf[:,3] = bf_sc[:rows]      # Gets side-chain atom B-factor
    return bf
```

The returned B-factor arrays are used to build a new NumPy array with four columns, named *bf*. In the first column of this new array, we have the residue numbers, in the second, third, and fourth columns, we have mean B-factors for main-chain, side-chain and all atoms, respectively. This mean B-factors are calculated for each residue. The *bf* array is returned by the function.

python™
www.python.org

The *calculate_bfactors()* function has as parameter a list of atoms, named *list_of_atoms_in*. Since the code is long, it is divided in several slides.

```
def calculate_bfactors(list_of_atoms_in):
    """Function to calculate average B-factor for each residue"""
    import numpy as np

    # Calculates the number of atoms in the list_of_atoms_in
    count_atoms = len(list_of_atoms_in)

    bfactor = np.zeros(count_atoms)
    residues = np.zeros(count_atoms,int)
    bfactor_res = np.zeros(999)
    # To have different number of residue for the first iteration
    former_res = -9999
    new_res = -9998
    # Sets count_res and count_atoms to zero
    count_res = 0
    count_atoms = 0
```

python™
www.python.org

Initially, we set up lists and counts, as shown below in red.

```
def calculate_bfactors(list_of_atoms_in):
    """Function to calculate average B-factor for each residue"""
    import numpy as np

    # Calculates the number of atoms in the list_of_atoms_in
    count_atoms = len(list_of_atoms_in)

    bfactor = np.zeros(count_atoms)
    residues = np.zeros(count_atoms,int)
    bfactor_res = np.zeros(999)
    # To have different number of residue for the first iteration
    former_res = -9999
    new_res = -9998
    # Sets count_res and count_atoms to zero
    count_res = 0
    count_atoms = 0
```

python™
www.python.org

Then we have a for loop, to read atoms. Inside the loop, we test if the residue number changes, which indicates that we have a new residue.

```
# Looping through list_of_atoms_in
   for line in list_of_atoms_in:
       if former_res == new_res:
           bfactor_res[count_atoms] =  float(line[61:65])
           new_res = int(line[22:26])
           count_atoms += 1
       else:
           new_res = int(line[22:26])
           former_res = new_res
           if count_res > 0:
               bfactor[count_res] = bfactor_res[0:count_atoms].mean()
               residues[count_res] = count_res
               count_res += 1
               count_atoms = 0
           else:
               count_res = int(line[22:26])    # Allows to get first residue
   # Returns arrays
   return residues[0:count_res],bfactor[0:count_res]
```

python™
www.python.org

Once we finished reading a residue, we calculate the mean value calling *.mean* function from NumPy library. This function is applied to *bfactor_res[0:count_res]* array.

```
# Looping through list_of_atoms_in
    for line in list_of_atoms_in:
        if former_res == new_res:
            bfactor_res[count_atoms] =  float(line[61:65])
            new_res = int(line[22:26])
            count_atoms += 1
        else:
            new_res = int(line[22:26])
            former_res = new_res
            if count_res > 0:
                bfactor[count_res] = bfactor_res[0:count_atoms].mean()
                residues[count_res] = count_res
                count_res += 1
                count_atoms = 0
            else:
                count_res = int(line[22:26])    # Allows to get first residue
    # Returns arrays
    return residues[0:count_res],bfactor[0:count_res]
```

python™
www.python.org

We restricted *bfactor_res[0:count_res]* array, ranging from 0 to *count_res-1*.

```
# Looping through list_of_atoms_in
   for line in list_of_atoms_in:
       if former_res == new_res:
           bfactor_res[count_atoms] =  float(line[61:65])
           new_res = int(line[22:26])
           count_atoms += 1
       else:
           new_res = int(line[22:26])
           former_res = new_res
           if count_res > 0:
               bfactor[count_res] = bfactor_res[0:count_atoms].mean()
               residues[count_res] = count_res
               count_res += 1
               count_atoms = 0
           else:
               count_res = int(line[22:26])    # Allows to get first residue
   # Returns arrays
   return residues[0:count_res],bfactor[0:count_res]
```

Function: *calculate_bfactors()*

python
www.python.org

This function returns the following array: *residues[0:count_res]* and *bfactor[0:count_res]* .

```
# Looping through list_of_atoms_in
    for line in list_of_atoms_in:
        if former_res == new_res:
            bfactor_res[count_atoms] =  float(line[61:65])
            new_res = int(line[22:26])
            count_atoms += 1
        else:
            new_res = int(line[22:26])
            former_res = new_res
            if count_res > 0:
                bfactor[count_res] = bfactor_res[0:count_atoms].mean()
                residues[count_res] = count_res
                count_res += 1
                count_atoms = 0
            else:
                count_res = int(line[22:26])    # Allows to get first residue
    # Returns arrays
    return residues[0:count_res],bfactor[0:count_res]
```

Function: plot_mult_array()

python™ www.python.org

```
def plot_mult_array(x,x_label_in,y_label_in,list_legends_in):
    """Function to plot two multi-dimensional arrays"""

    import matplotlib.pyplot as plt

    num_var = len(x[0,:])          # Number of variables
    x1 = x[1:,0]                   # Gets array for number of residues(column 0)

    # Looping variables to get B-factor arrays
    for i in range(1,num_var):
        x2 = x[1:,i]               # Gets each column (1-3) for B-factors
        plt.plot(x1,x2,label=list_legends_in[i-1])   # Generates plot

    plt.legend(loc='upper left')     # Positioning the legends
    plt.xlabel(x_label_in)           # Adds axis label
    plt.ylabel(y_label_in)           # Adds axis label
    plt.grid(True)                   # Adds grid to the plot

    plt.show()                       # Shows plot

    plt.savefig("bfactor.png")       # Saves plot on png file
```

The *plot_mult_array()* function generates a plot for a given multidimensional array, assigned to *x* variable. Here, the number of columns in the input array (*len(x[0,:])*) is assigned to *num_var* variable. The column with the number of residues is assigned to variable *x1*, as follows: *x1* = *x[1:,0]*. The number 1 means to start in the second row and goes all the way to the last row 1: . The "0" means first column data.

```
def plot_mult_array(x,x_label_in,y_label_in,list_legends_in):
    """Function to plot two multi-dimensional arrays"""

    import matplotlib.pyplot as plt

    num_var = len(x[0,:])        # Number of variables
    x1 = x[1:,0]                 # Gets array for number of residues(column 0)

    # Looping variables to get B-factor arrays
    for i in range(1,num_var):
        x2 = x[1:,i]             # Gets each column (1-3) for B-factors
        plt.plot(x1,x2,label=list_legends_in[i-1])  # Generates plot

    plt.legend(loc='upper left')     # Positioning the legends
    plt.xlabel(x_label_in)           # Adds axis label
    plt.ylabel(y_label_in)           # Adds axis label
    plt.grid(True)                   # Adds grid to the plot

    plt.show()                       # Shows plot

    plt.savefig("bfactor.png")       # Saves plot on png file
```

The for loop assigns each B-factor column to the *x2* array. The *x1* and *x2* arrays are used in the *plt.plot()* function, which generates line plots for the data. The *plt.plot()* is part of the *Matplotlib* library. The x-axis is for *x1* and the y-axis is for *x2.*
The rest of the code is for labels, legends, grid, and finally showing and saving the plot. The plot is save in the *bfactor.png* file.

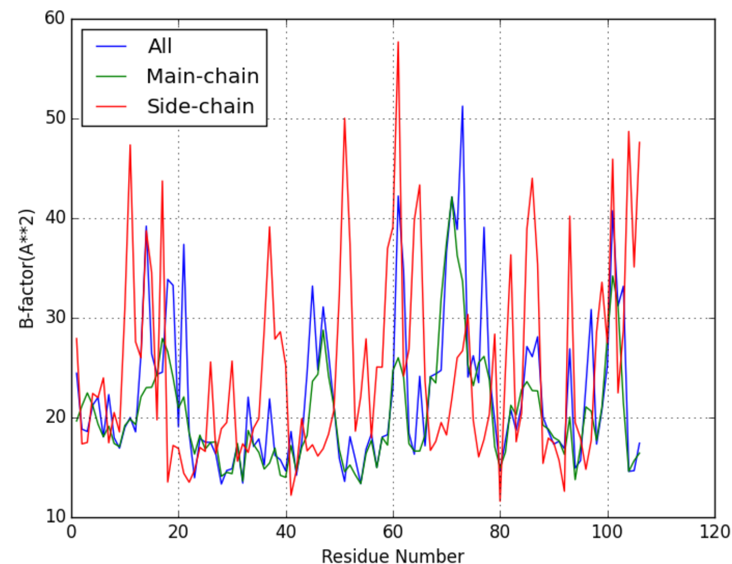
python™
www.python.org

To run *b_factor_plot.py,* type *python bfactor_plot.py*, and then type the PDB file name, as shown below.

```
C:\Users\Walter>python bfactor_plot.py
Type input file name => 1KXY.pdb

C:\Users\Walter>
```

The plot is generated and shown on the screen, as we can see below.
To run this tutorial, you need the following files: *bfactor_plot.py and 1KXY.pdb.*
All files should be in the same folder.



python™
www.python.org

Berman HM, Westbrook J, Feng Z, *et al.* The Protein Data Bank. Nucleic Acids Res 2000; 28: 235-42.

Berman HM, Battistuz T, Bhat TN, *et al.* The Protein Data Bank. Acta Crystallogr D Biol Crystallogr 2002; 58(Pt 6 No 1): 899-907.

Westbrook J, Feng Z, Chen L, Yang H, Berman HM. The Protein Data Bank and structural genomics. Nucleic Acids Res 2003; 31(1): 489-491.

Last update on July 6th 2016.

python™
www.python.org

This text was produced in a DELL Inspiron notebook with 6GB of memory, a 750 GB hard disk, and an Intel® Core® i5-3337U CPU @ 1.80 GHz running Windows 8.1. Text and layout were generated using PowerPoint 2013 and graphical figures were generated by *bfactor_plot.py*. This tutorial uses Arial font.

Author

I graduated in Physics (BSc in Physics) at University of Sao Paulo (USP) in 1990. I completed a Master Degree in Applied Physics also at USP (1992), working under supervision of Prof. Yvonne P. Mascarenhas, the founder of crystallography in Brazil.
My dissertation was about X-ray crystallography applied to organometallics compounds (De Azevedo Jr. *et al.*,1995).
During my PhD I worked under supervision of Prof. Sung-Hou Kim (University of California, Berkeley. Department of Chemistry), on a split PhD program with a fellowship from Brazilian Research Council (CNPq)(1993-1996). My PhD was about the crystallographic structure of CDK2 (Cyclin-Dependent Kinase 2) (De Azevedo Jr. *et al.*,1996). In 1996, I returned to Brazil. In April 1997, I finished my PhD and moved to Sao Jose do Rio Preto (SP, Brazil) (UNESP) and worked there from 1997 to 2005. In 1997, I started the Laboratory of Biomolecular Systems- Department of Physics-UNESP - São Paulo State University. In 2005, I moved to Porto Alegre/RS (Brazil), where I am now. My current position is coordinator of the Laboratory of Computational Systems Biology at Pontifical Catholic University of Rio Grande do Sul (PUCRS). My research interests are focused on application of computer simulations to analyze protein-ligand interactions. I'm also interested in the development of biological inspired computing and machine learning algorithms. We apply these algorithms to molecular docking simulations, protein-ligand interactions and other scientific and technological problems. I published over 160 scientific papers about protein structures and computer simulation methods applied to the study of biological systems (H-index: 36). These publications have over 4000 citations. I am editor for the following journals: