Itinerant electrons on a half-filled square lattice

Part 2

Luigi Matera

December 6, 2022

Abstract

This document is the second part of the project about itinerant electrons on an half-filled square lattice. It is divided in the two main parts: the first one related to the code used to compute the DOS, the second one related to the DOS error estimation.
All the following code can be found on https://github.com/giggi22/square_lattice_DOS; the most important parts are also reported in this document.
For the sake of brevity and simplicity of graphs, all quantities should be viewed in arbitrary units.

1 Computing the density of states

In order to obtain the density of states of the half-filled square lattice, the function defined through integral obtained in the first part must be computed:

$$DOS(E) = \frac{1}{a^2\beta\pi^2}\int_{arccos(\frac{E_o-E}{2\beta}+1)}^{arccos(\frac{E_o-E}{2\beta}-1)} 1\Big/\sqrt{1-\big[\frac{E_o-E}{2\beta}-cos(\xi)\big]^2}d\xi$$

In order to do so, some Python functions have been defined; those can be found in the file named "functions_DOS.py".

Python code

```python
import numpy as np
from scipy import integrate
from tqdm import tqdm

```

The following two functions are defined to obtain the upper and lower limit of the integral as a function of E, E_0 and β; i.e. $arccos(\frac{E_o-E}{2\beta}\pm 1)$.
As pointed out in the first part, for each value of E, except for $E = E_o$, one of the two functions is not well defined. In this case the integral limits are set at the "lower" or "higher" frontier of the FBZ. For example, for $E = E_0 - 2\beta$, the upper limit is $arccos(0) = \frac{\pi}{2}$, while the lower limit is not defined: $arccos(2)$ is not defined; in this case the lower limit is set to 0.

```python
def integral_lower_limit(
  energy: float,
  energy_0: float,
  beta: float
) -> float:

"""
Parameters
----------
energy (float): current energy
energy_0 (float): energy of isolated atom
beta (float): hopping energy

Returns
-------
(float) lower limit for the DOS integral

Notes
-----
This functions returns the lower limit for the DOS integral. In case of
  not valid energy, it would
return 0 which correspond to the lower border of the first quarter of the
   FBZ.
"""

if not energy_0 <= energy <= energy_0 + 4 * beta:
  return 0
else:
  return np.arccos((energy_0 - energy) / (2 * beta) + 1)

```

```python

def integral_upper_limit(
  energy: float,
  energy_0: float,
  beta: float
) -> float:

"""
Parameters
----------
energy (float): current energy
energy_0 (float): energy of isolated atom
beta (float): hopping energy

Returns
-------
(float) lower limit for the DOS integral

Notes
-----
This functions returns the upper limit for the DOS integral. In case of
  not valid energy, it would
return pi which correspond to the upper border of the first quarter of
  the FBZ.
"""

if not energy_0 - 4 * beta <= energy <= energy_0:
  return np.pi
else:
  return np.arccos((energy_0 - energy) / (2 * beta) - 1)

```

The following functions returns the integrand function, where the input parameters are E, E_0, β, the number of points and a border parameter. The output of the function is a set of two arrays, one contains the ξ values between the upper and lower limit of the integral, and the second one contains the corresponding values of the integrand function. If the border parameter is set to 0, the length of the ξ array is equal to the number of points given as input, with points equally spaced. Being the integrand function divergent in at least one of the FBZ boundaries, it can be made denser at the borders by using the border parameter. The border parameter defines the number of points to add at each border respect to the overall number of points. For example, if the number of points is set to 100, and the border parameter is set to 1, the ξ vector will present 100 point linearly spaced between the upper and lower limit plus 100 points near each border. The distance between the points near the edges is equal to $\frac{upper\ limit\ -\ lower\ limit}{number\ of\ points^{border\ parameter}}$, while in the middle is $\frac{upper\ limit\ -\ lower\ limit}{number\ of\ points}$ (see Fig. 3 and Fig. 4).

```python
def integrand_function(
  energy: float,
  energy_0: float,
  beta: float,
  num_points: int,
  bord_param: int = 0
) -> (np.ndarray, np.ndarray):

"""
Parameters
----------
energy (float): current energy
energy_0 (float): energy of isolated atom
beta (float): hopping energy
num_points (int): array length of the outputs
bord_param (int): defines the amount of points at the borders of xi-array

Returns
-------
xi_array (np.ndarray): array containing the xi values for the specific
  value of energy
y_array (np.ndarray): array containing the y values for the specific
  value of energy

Notes
-----
This function returns two arrays, which correspond to the value of xi
and the corresponding ones of the integrand function (y).
"""

"defining the integration limits"
lower_limit = integral_lower_limit(energy, energy_0, beta)
upper_limit = integral_upper_limit(energy, energy_0, beta)

"creating the output arrays"
xi_array = np.linspace(lower_limit, upper_limit, num_points)
step = upper_limit - lower_limit

# xi_array is denser at the border, where singularities occur, based on
  bord_param
for i in range(bord_param):
  xi_array = np.concatenate((
    np.linspace(lower_limit, lower_limit + step / num_points ** (i + 1),
  num_points),
    xi_array[1:-1],
    np.linspace(upper_limit - step / num_points ** (i + 1), upper_limit,
  num_points)
  ))
y_array = np.zeros_like(xi_array)

"computing the y-values for appropriate values of energy"
if np.abs((energy_0 - energy) / (4 * beta)) <= 1:
  for idx, val in enumerate(xi_array):
    differ = ((energy_0 - energy) / (2 * beta) - np.cos(val)) ** 2
    "in order to avoid a divergence in the integrand function, the"
    "next value would be equal to the previous one"
    if differ < 1:
      y_array[idx] = 1 / np.sqrt(1 - differ)
    else:
      y_array[idx] = y_array[idx - 1]

return xi_array, y_array

```

The following function returns the DOS, where the input parameter are E_0, β, a, the number of points for ξ, the number of points for the energy, the energy interval measured in β and a border parameter.
The energy interval parameter is used to give the interval of energy respect to E_0 measured in β. Being the DOS different from 0 in the region $E_o - 4\beta < E < E_o + 4\beta$, by setting the parameter slightly higher than 4, than the full DOS is obtained. Instead, by using a lower value (i.e. close to 0) the central divergence of the DOS can be studied.
The Simpson technique was used to integrate. The Simpson algorithm simply approximates the function with a parabola and then compute the subjacent area.
The number of points of the energy are denser near E_0, by using the same principle explained for the ξ array.

```
def gross_dos(
  energy_0: float,
  beta: float,
  lattice_constant: float,
  num_points_energy: int,
  num_points_xi: int,
  interval_energy: float = 4.2,
  bord_param: int = 0
) -> (np.ndarray, np.ndarray):

  """
  Parameters
  ----------
  energy_0 (float): energy of isolated atom
  beta (float): hopping energy
  lattice_constant (float): lattice constant length
  num_points_energy (int): array length of the outputs (energy and DOS)
  num_points_xi (int): array length of the xi-array for the integration
    step
  interval_energy (float): interval of energy respect to E0 measured in
    beta, suggested value is slightly above 4
  bord_param (int): defines the amount of points at the borders of xi-array

  Returns
  -------
  energy_array (np.ndarray): array containing the energy values
  dos_array (np.ndarray): dos array

```

```
    Notes
    -----
    This function returns two arrays, which correspond to the value of the
      energy
    and the corresponding ones of the density of states.
    """

    "creating energy and dos arrays; being energy_0 a singularity point, the
      two arrays are denser in its proximity."
    num_points = int((num_points_energy+2)/2)
    step = interval_energy * beta
    energy_array = np.concatenate((
      np.linspace(energy_0 - step, energy_0, num_points)[:-1],
      np.linspace(energy_0 - step / num_points, energy_0, num_points)[:-1],
      np.linspace(energy_0, energy_0 + step / num_points, num_points)[1:],
      np.linspace(energy_0, energy_0 + step, num_points)[1:]
    ))
    dos_array = np.zeros_like(energy_array)

    "tqdm allows to have the progress bar"
    for idx in tqdm(range(len(energy_array))):
      energy_val = energy_array[idx]
      x_data, y_data = integrand_function(energy_val, energy_0, beta,
      num_points_xi, bord_param)
      dos_array[idx] = 1 / (lattice_constant ** 2 * beta * np.pi ** 2) *
      integrate.simps(y_data, x_data)

    return energy_array, dos_array

```

The following two functions were created but never used in the second part of this document, they can be used to "clean" the DOS from points that are divergent due to numerical approximation. For example, they can be used in case the previously explained border parameter is set to 0, in fact in that case is probable to obtain few point that are way above the average value of the DOS.

```
def low_dos(
  energy: np.ndarray,
  dos: np.ndarray
) -> (np.ndarray, np.ndarray):

"""
Parameters
----------
energy (np.ndarray): energy array
dos (np.ndarray): density of states array

Returns
-------
energy_low (np.ndarray): array containing the new energy values
dos_low (np.ndarray): dos array containing the dos values below the
  average value

Notes
-----
This function returns two arrays, which correspond to the value of the
  energy
and the corresponding ones of the density of states with the values below
   the average.
"""

```

```python
    dos_low = dos[[n for n, i in enumerate(dos) if i < np.average(dos)]]
    energy_low = energy[[n for n, i in enumerate(dos) if i < np.average(dos)
    ]]
    return energy_low, dos_low

```

```python

def high_dos(
    energy: np.ndarray,
    dos: np.ndarray
) -> (np.ndarray, np.ndarray):

    """
    Parameters
    ----------
    energy (np.ndarray): energy array
    dos (np.ndarray): density of states array

    Returns
    -------
    energy_low (np.ndarray): array containing the new energy values
    dos_low (np.ndarray): dos array containing the dos values above the
    average value

    Notes
    -----
    This function returns two arrays, which correspond to the value of the
    energy
    and the corresponding ones of the density of states with the values above
     the average.
    """

    dos_high = dos[[n for n, i in enumerate(dos) if i > np.average(dos)]]
    energy_high = energy[[n for n, i in enumerate(dos) if i > np.average(dos)
    ]]
    return energy_high, dos_high

```

2 DOS and error estimation

Python code

The following code is the one used to obtain the square lattice DOS. It simply compute the DOS using the previously explained functions, it saves the data and finally plots the full DOS.
Moreover, for the error estimation, a new parameter θ is defined ("goodness_parameter" is the name in the code):

$$\theta = \frac{a^2}{2}\int_{E_min}^{E_max} DOS(E)dE \tag{1}$$

The theoretical value of θ is 1, so by computing it numerically, an estimate of the error can be obtained. The integration over the DOS is computed using a trapezoidal approximation.

```python
import numpy as np
import functions_DOS as fD
import matplotlib.pyplot as plt
from scipy import integrate


"""defining lattice parameters"""
lattice_constant = 1  # lattice constant
energy_0 = 0  # atomic energy level
beta = 5  # hopping energy

"""defining plot parameters"""
points_energy_div_2 = 500  # number of half the energy points
interval_energy = 5  # interval over which the energy is computed, measured in beta

"""defining integration parameters"""
num_points_xi = 1000  # number of points over which the integral is computed
energy_border_parameter = 3  # number of points added to the integral tails, measured in num_points_xi

"""file name where the DOS is stored, keep it empty to not save it"""
file_name = ""

"""computing the DOS"""
E, DOS = fD.gross_dos(energy_0, beta, lattice_constant, points_energy_div_2,
num_points_xi, interval_energy, energy_border_parameter)

"""saving the file if file_name is not empty"""
if not file_name == "":
np.save(file_name, [E, DOS])


"""plotting the DOS"""
fig, axs = plt.subplots(1)
axs.plot(E, DOS, linewidth=2.5)
axs.set_xlabel("Energy")
axs.set_ylabel("DOS")
axs.grid()
fig.tight_layout()

"""defining goodness_parameter, the closer it is to 1, the better the DOS is"""
goodness_parameter = lattice_constant**2*integrate.trapz(DOS[~np.isnan(DOS)], E[~np.isnan(DOS)])/2

"""inserting the lattice parameters in the legend"""
axs.plot([], [], ' ', label=r"$E_0$ = {}".format(energy_0))
axs.plot([], [], ' ', label=r"$\beta$ = {}".format(beta))
axs.plot([], [], ' ', label="a = {}".format(lattice_constant))
axs.plot([], [], ' ', label=r"$\Theta$ = {:.5f}".format(goodness_parameter))
plt.legend()

plt.show()

```

Code output

In the following figure is shown the obtained density of states. Note that the value of θ is slightly larger than 1. The values of E, E_0 and a are reported in the legend. The "integration" and "plotting" parameters are the ones written in the code.

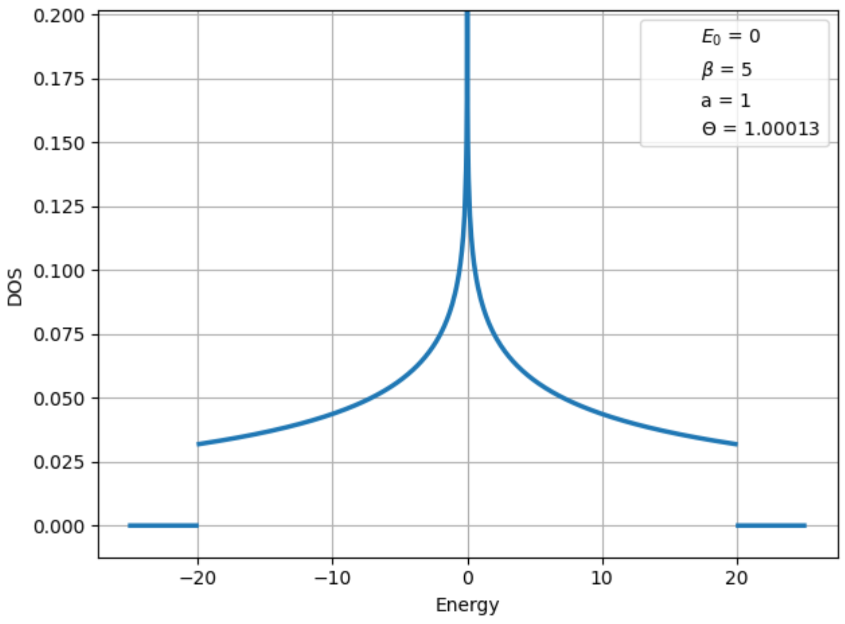

Figure 1: DOS of half-filled square lattice. The "integration" and "plotting" parameters are the ones written in the previous code.

Error estimation

Python code: https://github.com/giggi22/square_lattice_DOS/blob/main/theta_estimation.py

In order to have an idea about the error involved in the calculation, we can start by varying the border parameter value and see what happens to the θ value. In the following figure there are the DOS computed for values of the border parameter ranging from 0 to 4.

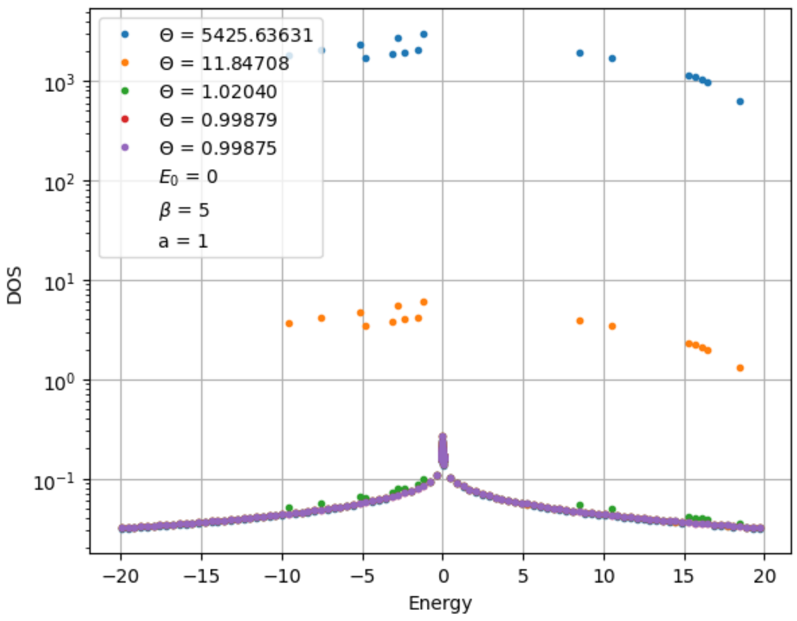

Figure 2: DOS with different values of the border parameter. Blue = 0; Orange = 1; Green = 2; Red = 3; Purple = 4. The corresponding θ values are written in the legend.

From a quick view, it is clear how the action of the border parameter allows to have a better final result, with lower numerical divergences (which are later discussed). Moreover, it can be seen how the θ parameter converges to a value really close to 0, with a deviation of $\approx 0.2\%$.
Another thing that must be pointed out is the fact that when the θ value is computed in an energy range larger than $E_o - 4\beta < E < E_o + 4beta$, it converges to a value slightly larger than 1. This happens because near the energy edges (so at $E = E_0 \pm 4\beta$), the function goes from a finite value to 0, which leads to an overestimation of the DOS integral. Thus, the θ value obtained in Fig. 1 can not be used directly as indication of the error because is it due to an overestimate of the integral over the energies and an underestimate of the DOS. Thus, the error that can be used is $\approx 0.2\%$, because it is computed only in the region $E_0 - 4\beta \leq E \leq E_0 + 4\beta$, so avoiding the overestimation due to the band edges.

Origin of numerical divergences in the DOS

Python code: `https://github.com/giggi22/square_lattice_DOS/blob/main/Integrand_function.py`

In order to understand the presence of the anomalous numerical divergences in DOS in Fig. 2, we have to give a look at the discretized integrand function for different values of the energy:

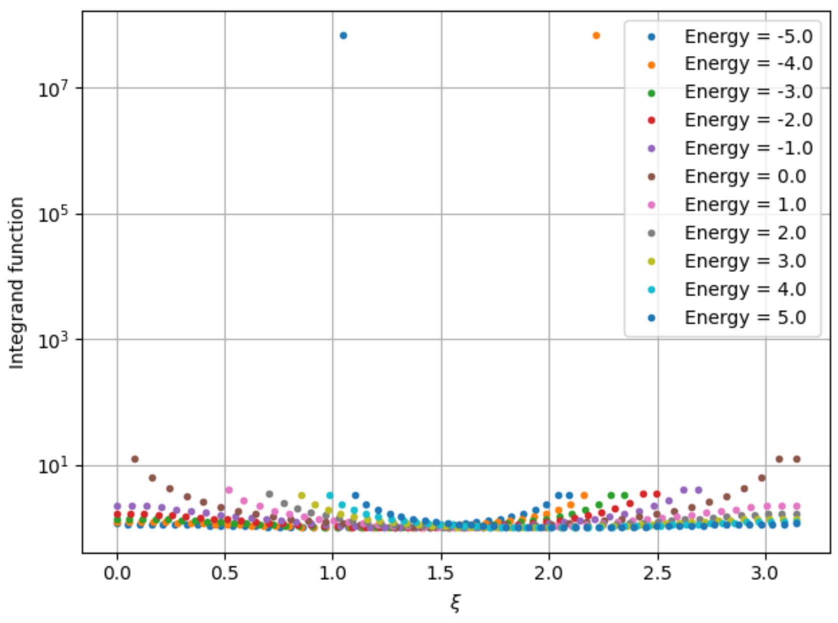

Figure 3: Integrand function for different values of energy. The border parameter is set to 0 for each curve.

It can be noticed how the function discretization leads to the presence of few points with a really high value for certain energies. Those points determine the presence of anomalous peaks in the DOS in Fig. 2.
Instead, by using the border parameter, it can be noted how those anomalous points become less relevant for the integral calculation, being their "width" decreased:

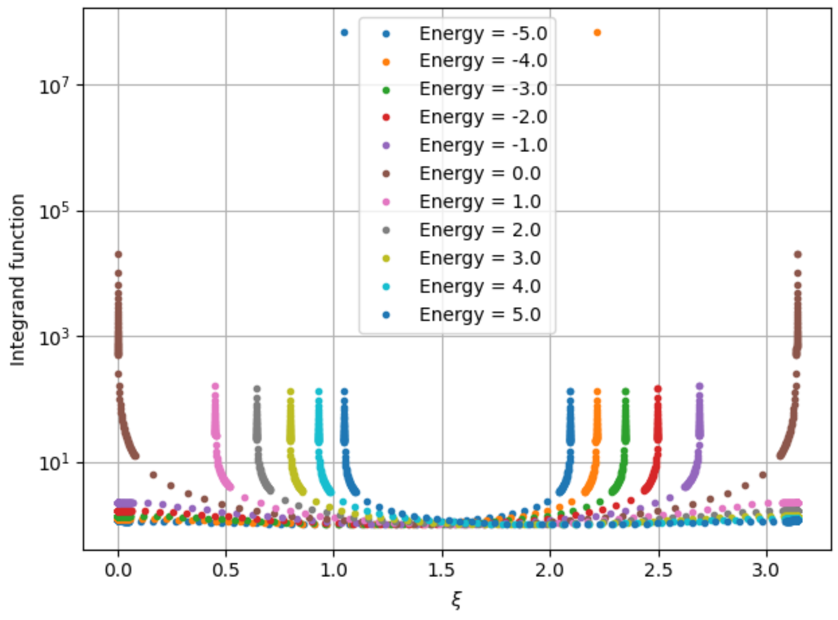

Figure 4: Integrand function for different values of energy. The border parameter is set to 2 for each curve.

By comparing the two figures, it is easy to understand how the integral value can be either overestimated, when an anomalous peak appears; or underestimated when the divergences of the function are not properly discretized. Nevertheless, the overestimation can be strongly reduced by using the border parameter (like in Fig. 2).

DOS value at band edges

Python code: https://github.com/giggi22/square_lattice_DOS/blob/main/DOS_beta_variable.py

While the DOS value at the band center has been evaluated in the first part of the project, here we try to compute the DOS value at the band edges.
First of all, we want to understand how β determines the DOS value at the band edges. Starting from the density of state function

$$DOS(E) = \frac{1}{a^2\beta\pi^2}\int_{arccos(\frac{E_o-E}{2\beta}+1)}^{arccos(\frac{E_o-E}{2\beta}-1)} 1\Big/\sqrt{1-[\frac{E_o-E}{2\beta}-cos(\xi)]^2}d\xi$$

we can substitute at $E = E_0 - 4*\beta + \epsilon$, where ϵ is a number really close to 0, we get:

$$DOS(E_0 - 4*\beta + \epsilon) = \frac{1}{a^2\beta\pi^2}\int_0^{\delta} 1\Big/\sqrt{1-[2-\frac{\epsilon}{2\beta}-cos(\xi)]^2}d\xi \tag{2}$$

where $\delta = arccos(1+\frac{\epsilon}{2\beta})$, which in the limit of ϵ going to 0, tends to 0. The same can be done with the opposite edge, in that case $E = E_0 + 4\beta - \epsilon$ and the integral limits

would be $\pi - \delta$ and π.
In the following figure are shown the computed values of the DOS$*\beta * a^2$ ad the band edges for different values of β:

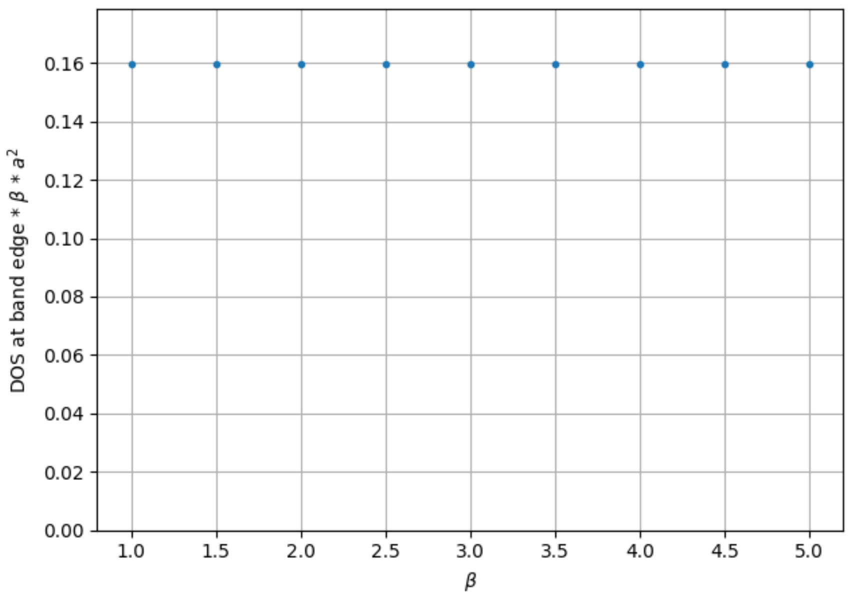

Figure 5: DOS$*\beta * a^2$ for different values of β, the points distribution can be considered flat with a constant value of ≈ 0.1595.

It can be seen from Fig. 5 that there is no dependence on β and the value is approximately 0.1595.
Thus the DOS at the band edges can be written as:

$$DOS(E = E_0 \pm 4\beta) \approx \frac{0.1595}{\beta a^2}$$

.