Open Optical Networks - Network

November 4, 2020

The aim of these exercises is to build a software abstraction of the network represented in Fig.1 and simulate the signal propagation from an input node to an output node. In the whole exercise, define the 'constructor', the 'getter' and 'setter' methods for all the class's attributes, for any class.

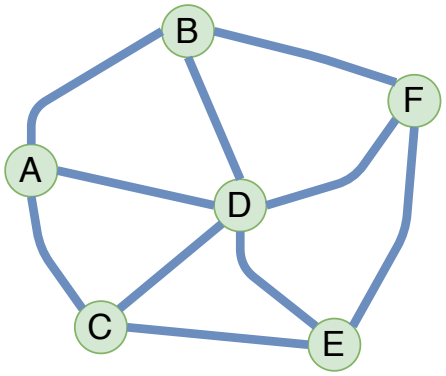

Figure 1: Network sketch.

Necessary library imports:

```
import json
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.constants import c
```

1. Define the class **Signal_information** that has the following attributes:
   - **signal_power**: float
   - **noise_power**: float
   - **latency**: float
   - **path**: list[string]

such that its constructor initializes the signal power to a given value, the noise power and the latency to zero and the path as a given list of letters that represents the labels of the nodes the signal has to travel through. The attribute **latency** is the total time delay due to the signal propagation through any network element along the path. Define the methods to update the signal and noise powers and the latency given an increment of these quantities. Define a method to update the path once a node is crossed.

```
class SignalInformation(object):
    def __init__(self,power,path):
        self._signal_power = power
        self._path = path
        self._noise_power = 0
        self._latency = 0

    @property
    def signal_power(self):
        return self._signal_power

    @property
    def path(self):
        return self._path

    @path.setter
    def path(self, path):
        self._path = path

    @property
    def noise_power(self):
        return self._noise_power

    @noise_power.setter
    def noise_power(self, noise):
        self._noise_power = noise

    @property
    def latency(self):
        return self._latency

    @latency.setter
    def latency(self, latency):
        self._latency = latency

    def add_noise(self,noise):
        self.noise_power += noise

    def add_latency(self,latency):
        self.latency += latency
```

```
    def next(self):
        self.path = self.path[1:]
```

2. Define the class **Node** that has the following attributes:

   - **label**: string
   - **position**: tuple(float, float)
   - **connected_nodes**: list[string]
   - **successive**: dict[Line]

   such that its constructor initializes these values from a python dictionary input. The attribute **successive** has to be initialized to an empty dictionary. Define a propagate method that update a signal_information object modifying its path attribute and call the successive element propagate method, accordingly to the specified path.

```
class Node(object):
    def __init__(self, node_dict):
        self._label = node_dict['label']
        self._position = node_dict['position']
        self._connected_nodes = node_dict['connected_nodes']
        self._successive = {}

    @property
    def label(self):
        return self._label

    @property
    def position(self):
        return self._position

    @property
    def connected_nodes(self):
        return self._connected_nodes

    @property
    def successive(self):
        return self._successive

    @successive.setter
    def successive(self, successive):
        self._successive = successive

    def propagate(self,signal_information):
        path = signal_information.path
        if len(path)>1:
            line_label = path[:2]
            line = self.successive[line_label]
            signal_information.next()
```

```
            signal_information = line.propagate(signal_information)

        return signal_information
```

3. Define the class **Line** that has the following attributes:

   - **label**: string
   - **length**: float
   - **successive**: dict[Node]

   The attribute **successive** has to be initialized to an empty dict. Define the following methods that update an instance of the **signal_information**:

   - **latency_generation**(): float
   - **noise_generation**(signal_power): 1e-3 * signal_power * length

   The light travels through the fiber at around 2/3 of the speed of light in the vacuum. Define the line method **latency_generation** accordingly. Define a propagate method that updates the signal information modifying its noise power and its latency and call the successive element propagate method, accordingly to the specified path.

```
class Line(object):
    def __init__(self, line_dict):
        self._label = line_dict['label']
        self._length = line_dict['length']
        self._successive = {}

    @property
    def label(self):
        return self._label

    @property
    def length(self):
        return self._length

    @property
    def successive(self):
        return self._successive

    @successive.setter
    def successive(self, successive):
        self._successive = successive

    def latency_generation(self):
        latency = self.length / (c * 2 / 3)
        return latency

    def noise_generation(self,signal_power):
```

```
        noise = 1e-3 * signal_power * self.length
        return noise

    def propagate(self,signal_information):
        # Update latency
        latency = self.latency_generation()
        signal_information.add_latency(latency)

        # Update noise
        signal_power = signal_information.signal_power
        noise = self.noise_generation(signal_power)
        signal_information.add_noise(noise)

        node = self.successive[signal_information.path[0]]
        signal_information = node.propagate(signal_information)
        return signal_information
```

4. Define the class **Network** that has the attributes:

   - **nodes**: dict[Node]
   - **lines**: dict[Lines]

   both the dictionaries have to contain one key for each network element that coincide with the element **label**. The value of each key has to be an instance of the network element (**Node** or **Line**). The constructor of this class has to read the given JSON file 'nodes.json', it has to create the instances of all the nodes and the lines. The line labels have to be the concatenation of the node labels that the line connects (for each couple of connected nodes, there would be two lines, one for each direction, e.g. for the nodes 'A' and 'B' there would be line 'AB' and 'BA'). The lengths of the lines have to be calculated as the minimum distance of the connected nodes using their positions. Define the following methods:

   - **connect()**: this function has to set the successive attributes of all the network elements as dictionaries (i.e., each node must have a dict of lines and each line must have a dictionary of a node);
   - **find_paths**(string, string): given two node labels, this function returns all the paths that connect the two nodes as list of node labels. The admissible paths have to cross any node at most once;
   - **propagate**(signal_information): this function has to propagate the signal_information through the path specified in it and returns the modified spectral_information;
   - **draw**(): this function has to draw the network using matplotlib (nodes as dots and connection as lines).

```
class Network(object):
    def __init__(self,json_path):
```

```
        node_json = json.load(open(json_path,'r'))
        self._nodes = {}
        self._lines = {}
        for node_label in node_json:
            # Create the node instance
            node_dict = node_json[node_label]
            node_dict['label'] = node_label
            node = Node(node_dict)
            self._nodes[node_label] = node

            # Create the line instances
            for connected_node_label in node_dict['connected_nodes']:
                line_dict ={}
                line_label = node_label + connected_node_label
                line_dict['label'] = line_label
                node_position = np.array(node_json[node_label]['position'])
                connected_node_position =
                    np.array(node_json[connected_node_label]['position'])
                line_dict['length'] =
                    np.sqrt(
                      np.sum((node_position - connected_node_position)**2)
                    )
                line = Line(line_dict)
                self._lines[line_label] = line

    @property
    def nodes(self):
        return self._nodes

    @property
    def lines(self):
        return self._lines

    def draw(self):
        nodes = self.nodes
        for node_label in nodes:
            n0 = nodes[node_label]
            x0 = n0.position[0]
            y0 = n0.position[1]
            plt.plot(x0,y0,'go',markersize=10)
            plt.text(x0+20,y0+20,node_label)
            for connected_node_label in n0.connected_nodes:
                n1 = nodes[connected_node_label]
                x1 = n1.position[0]
                y1 = n1.position[1]
                plt.plot([x0,x1],[y0,y1],'b')
        plt.title('Network')
        plt.show()

    def find_paths(self,label1,label2):
```

```
        cross_nodes = [key for key in self.nodes.keys()
                       if ((key!=label1) & (key!=label2))]
        cross_lines = self.lines.keys()
        inner_paths = {'0': label1}
        for i in range(len(cross_nodes)+1):
            inner_paths[str(i+1)] = []
            for inner_path in inner_paths[str(i)]:
                inner_paths[str(i+1)]
                    += [inner_path + cross_node
                        for cross_node in cross_nodes
                        if ((inner_path[-1]+cross_node in cross_lines) &
                            (cross_node not in inner_path))]
        paths = []
        for i in range(len(cross_nodes)+1):
            for path in inner_paths[str(i)]:
                if path[-1] + label2 in cross_lines:
                    paths.append(path + label2)
        return paths

    def connect(self):
        nodes_dict = self.nodes
        lines_dict = self.lines
        for node_label in nodes_dict:
            node = nodes_dict[node_label]
            for connected_node in node.connected_nodes:
                line_label = node_label+connected_node
                line = lines_dict[line_label]
                line.successive[connected_node] = nodes_dict[connected_node]
                node.successive[line_label] = lines_dict[line_label]

    def propagate(self,signal_information):
        path = signal_information.path
        start_node = self.nodes[path[0]]
        propagated_signal_information =
                start_node.propagate(signal_information)
        return propagated_signal_information
```

5. For all possible paths between all possible node couples, create a pandas dataframe that contains the path string as "A->B-> ...", the total accumulated latency, the total accumulated noise and the signal to noise ratio obtained with the propagation through the paths of a spectral_information with a signal_power of 1 mW. Calculate the signal to noise ratio in dB using the formula $10\log(signal\_power/noise\_power)$

```
network = Network('nodes.json')
network.connect()
node_labels = network.nodes.keys()
pairs = []
for label1 in node_labels:
    for label2 in node_labels:
```

```
        if label1!=label2:
            pairs.append(label1+label2)

columns = ['path','latency','noise','snr']
df = pd.DataFrame()
paths = []
latencies = []
noises = []
snrs = []

for pair in pairs:
    for path in network.find_paths(pair[0],pair[1]):
        path_string = ''
        for node in path:
            path_string += node + '->'
        paths.append(path_string[:-2])

        # Propagation
        signal_information = SignalInformation(1,path)
        signal_information = network.propagate(signal_information)
        latencies.append(signal_information.latency)
        noises.append(signal_information.noise_power)
        snrs.append(
            10*np.log10(
                signal_information.signal_power/signal_information.noise_power
            )
        )
df['path'] = paths
df['latency'] = latencies
df['noise'] = noises
df['snr'] = snrs
```

Additional Notes All power values have to be considered in Watts, all lengths in meters and all latencies in seconds.