TECHNISCHE HOCHSCHULE OSTWESTFALEN-LIPPE
UNIVERSITY OF APPLIED SCIENCES AND ARTS

Research project:

Finite Element Method

In Python

Group 3: Loads

Autors:

Mahan Mashayekh

Ardalan Mirhadi

Tutors:

Prof. Dipl.-Ing. Jens-Uwe Schulz

Thomaz Da Silva Lopes Vieira

Winter Semester 2020-21

Table of Contents

1. Project Description:

FEM(Finite Element Method) is a numerical approach to predicting the behavior of structural systems. Our task is part of the project with the purpose of programming FEM using Python. The user defines the loads, moments, and values; then, the program visualizes them in Rhinoceros. The received data will store in a JSON file and transfer to the following groups for the next steps.

In order to gain these operations, three classes are defined for point loads, linear loads & moments, and users can choose different methods of inputting data through the Rhino Dialogue Box. The overall process of the project is shown in (figure 1).

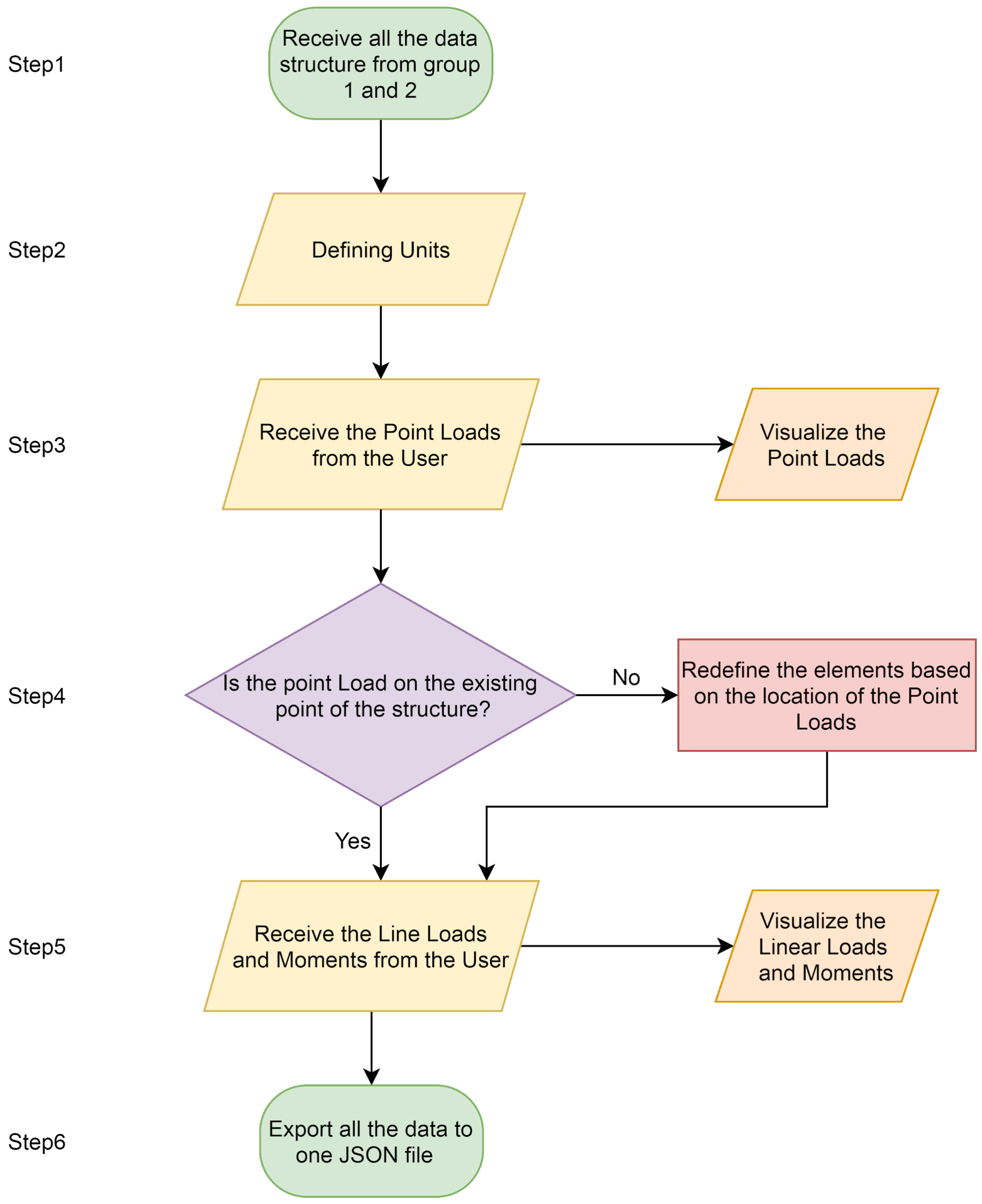

Step1
Receive all the data structure from group 1 and 2
Step2
Defining Units
Step3
Receive the Point Loads from the User
Visualize the Point Loads
Step4
Is the point Load on the existing point of the structure?
No
Redefine the elements based on the location of the Point Loads
Yes
Step5
Receive the Line Loads and Moments from the User
Visualize the Linear Loads and Moments
Step6
Export all the data to one JSON file

Figure 1: Diagram of the overall process

2. Scientific Theory:

Three types of load can act on the beam. these are:

1. Point load is also called a concentrated load[1]
2. Distributed load
3. Coupled load

2.1. Point Load

Point load acts over a small distance. This load can be considered as acting on a point. Point load is indicated by **P** and symbol of point load is an arrowhead.

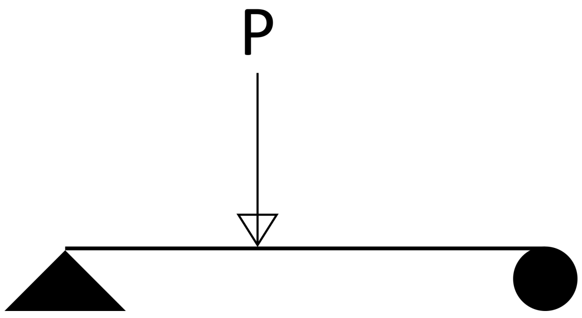


Figure 2: Point Load

2.2. Distributed Load

Distributed load acts over a considerable length, or you can say over a measurable distance. Distributed load is measured as per unit length.

Two types of distributed load

1. Uniformly Distributed load (UDL)
2. Uniformly Varying load (Non-uniformly distributed load).

Uniformly Distributed Load (UDL)

In this type of distributed load the magnitude remains uniform throughout the length.

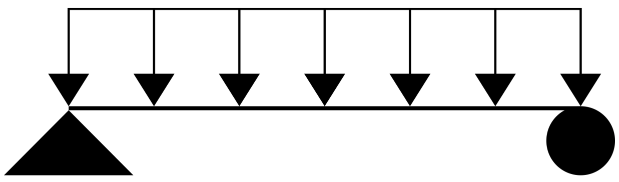


Figure 3: Uniformly Distributed Load (UDL)

[1] Podut, Alex. (2018). Strength of Materials Supplement for Power Engineering

Non–Uniformly Distributed Load or Uniformly Varying Load

in this type of load, the magnitude varies along the loading length with a constant rate.
there are two types of Uniformly Varying Load:

1. Triangular Load
2. Trapezoidal Load

In the **Triangular Load,** the magnitude starts from zero at one end of the span and increases constantly till the second end of the span.

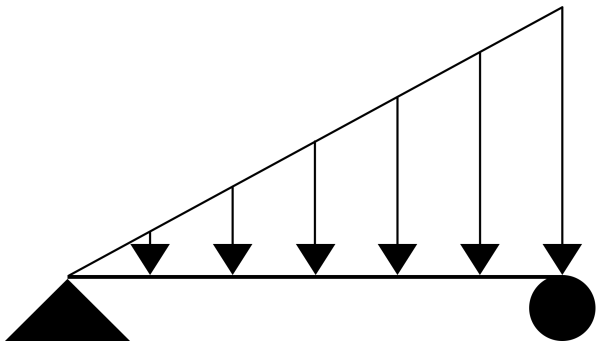


Figure 4: Triangular Load

In the **Trapezoidal load**, the load acts as a form of a trapezoid. Trapezoidal loads are the combination of uniformly distributed loads (UDL) and triangular loads.

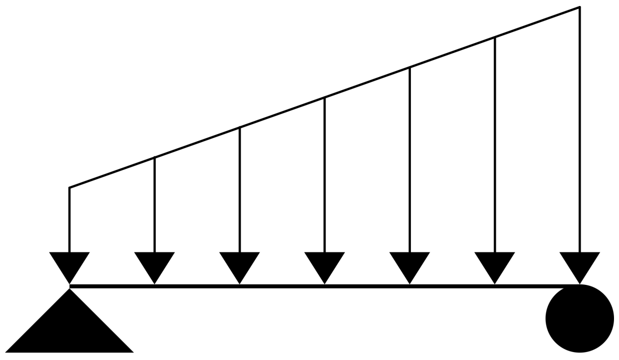


Figure 5: Trapezoidal Load

3. Outsource Data Inputs:

As shown in Step1 of the diagram, we receive the structure's geometry as a rhino file and a dictionary of data from group 2 as a JSON file consisting of elements, points, profiles, and conditions. Therefore, different dictionaries have been created with the same name to organize the received data.

Code 1

```
#Import the dictionaries from the data_file
structure_data = {}
points = {}
elements = {}
conditions = {}
profile = {}
with open('data_file.json') as json_file:
    structure_data = json.load(json_file)
    points = structure_data["Points"]
    elements = structure_data["Elements"]
    conditions = structure_data["Conditions"]
    profile = structure_data["Profile"]
```

4. Units:

In the second step, the user is asked to choose the Units of Load (figure 6). And the Units of the Dimension are taken automatically from the rhino system.

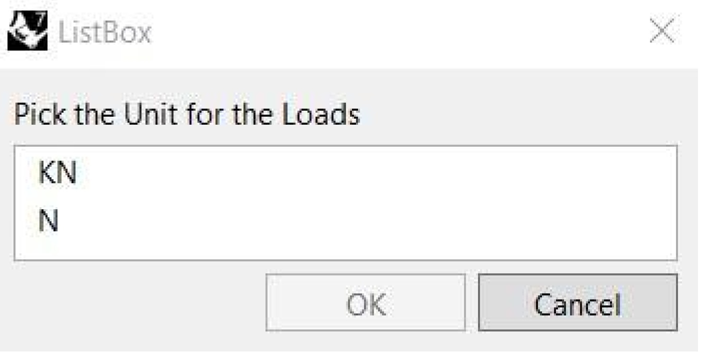


Figure 6: Unit of the Loads

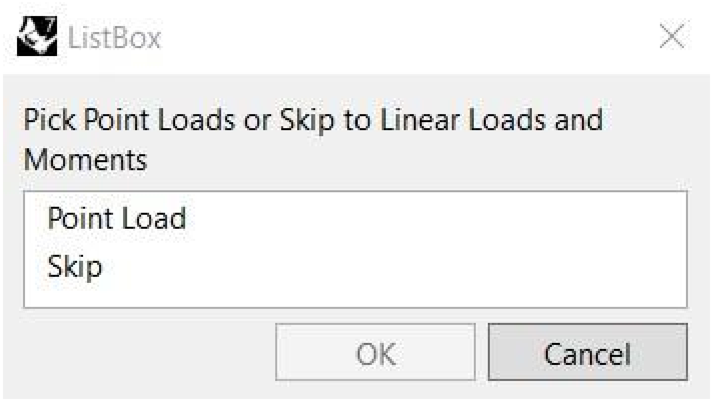
Two different variables, **unit_l**, and **unit_f** are defined in the code. The unit_l takes the dimension unit from the unit system of the RhinoScript. The user establishes the unit_f for the Loads. The Units of the dimensions could be meter, centimeter, or millimeter, so they are abbreviated as "**m**," "**cm**," and "**mm**". The units of the loads are also abbreviated as "**KN**" for *kilonewtons, and "**N**" for Newtons.*

Code 2

```
#UNITS
unit_f = rs.ListBox(["KN", "N"], "Pick the Unit for the Loads")
if rs.UnitSystem() == 2:
    unit_l = "mm"
if rs.UnitSystem() == 3:
    unit_l = "cm"
if rs.UnitSystem() == 4:
    unit_l = "m"
```

5. Point Loads:

In the third step, the user can choose whether they will pick a Point Load or skip this step to define Linear Loads or Moments (figure 7).
A class is defined for the Point Loads. Every time the user wants to pick a point load, an object is determined to collect the point load variables in the class and return those parameters to store in a list. The object is defined as “obj” in the code, and the returning variables from the class are “point” and “dir,” which are the location and the direction of the point load respectively. They are all stored in a list named “P” in this step (Code3).



Pick Point Loads or Skip to Linear Loads and Moments

Point Load
Skip

Figure 7: Picking the point loads

5.1. Point load methods:

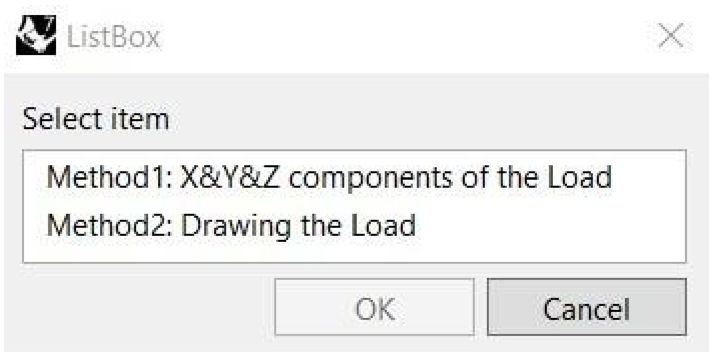
There are two methods for defining point loads (figure 8)(code 3):

Method 1: Load direction based on x,y,z value

The program asks the user to choose the load location and the values of x ,y, and z.

Method 2: drawing the load in Rhino

The program asks the user to choose the load location and draw a line in the desired direction.



Select item

Method1: X&Y&Z components of the Load
Method2: Drawing the Load

Figure 8: choosing a method for picking point loads

After the user picks a method, the class's related functions are called to take the user's point load. These two functions also have an argument “i” which shows the index of the point loads.

```
219   obj.Point_Load_Method1(i)
221   obj.Point_Load_Method2(i)
```

5.2. Point load Inputs:

- Selecting point location by the user
- Determining load direction by the user

Code 3

```
213 while True:
214   Pick_Load = rs.ListBox(["Point Load", "Skip"], "Pick Point Loads or Skip to
Linear Loads and Moments")
215   if Pick_Load == "Point Load" :
216         obj = Point_Load (0,0)
217         Point_Method = rs.ListBox(["Method1: X&Y&Z components of the Load",
"Method2: Drawing the Load"])
218         if Point_Method == "Method1: X&Y&Z components of the Load" :
219               obj.Point_Load_Method1(i)
220         if Point_Method == "Method2: Drawing the Load" :
221               obj.Point_Load_Method2(i)
224 to 251 (Group 2 code): #Checks the points with the element dictionary and update
the element dictionary
252         P.append(obj)
253         i= i+1
254   if Pick_Load == "Skip" :
255         break
```

5.3. Point load Class:

In the class point loads, there are two main functions:

1. `def Point_Load_Method1(self, i):`
2. `def Point_Load_Method2(self, i):`

In the first function, the user is asked to select the Point Load's location and then Enter the Load components in the X, Y, and Z Direction, respectively. Based on the point load's coordinates and the fx, fy, and fz parameters, the direction of the point load is then defined, and an arrowhead and the name of the point load by its index are added as the visualization of the point load (figure 9). As mentioned, the returning variables from this function are the point(point) and direction(dir) of the point load.
The second function acts the same as the first function. It takes “i” as an input and returns point load coordinates and the direction of the point load. The only difference in this function is the method of taking the load’s direction from the user. Here the user picks the direction of the point load by drawing a line. The visualization is also the same as the first function.

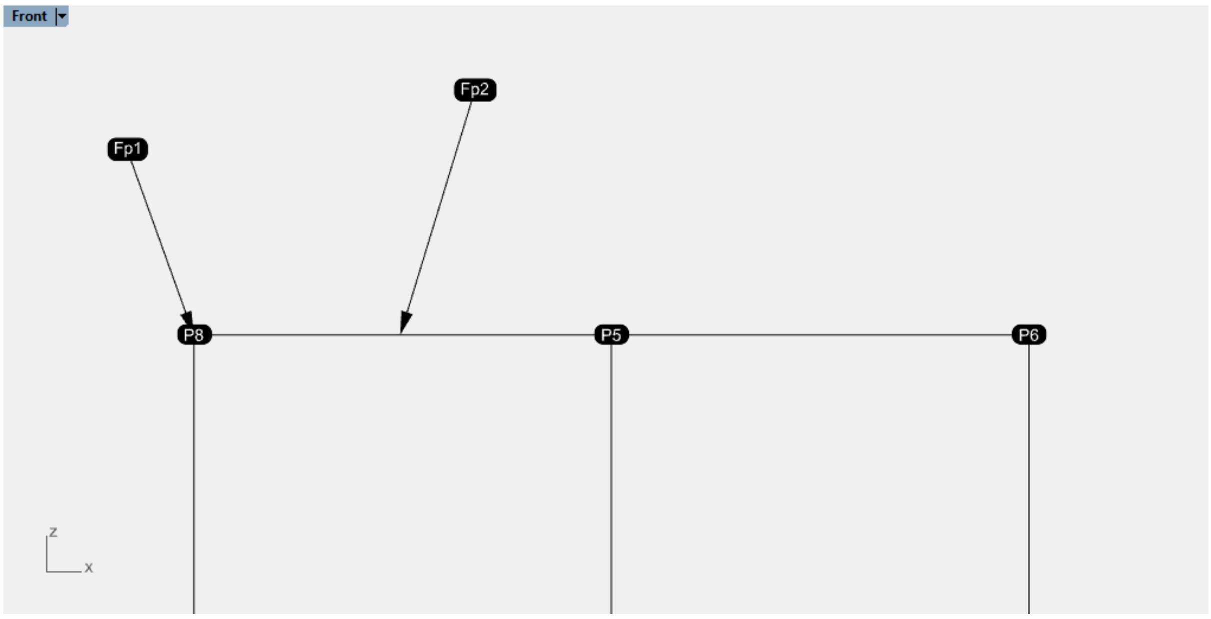


Figure 9: Visualization of the point loads

Code 4

```
class Point_Load:
      def __init__(self,_point, _dir):
            self.point = _point
            self.dir = _dir
      def Point_Load_Method1(self, i):
            self.point = rs.GetPoint("Select Point Load Location")
            fx = rs.GetReal("Enter the Load component in the X Direction")
            fy = rs.GetReal("Enter the Load component in the Y Direction")
            fz = rs.GetReal("Enter the Load component in the Z Direction")
            AB = rs.CreatePoint(fx*-1, fy*-1, fz*-1)
            B = self.point + AB
            line = rg.Line(B, self.point)
            self.dir = line.Direction
            l = rs.AddLine( self.point , B)
            rs.CurveArrows(l, 1)
            rs.AddTextDot("Fp" + str(i+1) ,B)

            return self.point, self.dir

      def Point_Load_Method2(self, i):
            self.point = rs.GetPoint("Select Point Load Location")
            B = rs.GetPoint("Start of the load")
            line = rg.Line(B,self.point)
            self.dir = line.Direction
            l = rs.AddLine(self.point,B)
            rs.CurveArrows(l, 1)
            rs.AddTextDot("Fp" + str(i+1) ,B)
            return self.point, self.dir
```

6. Group 2 Function:

In Step 4, group 2 has defined a function, after receiving the point loads from the user, to see if the point loads are on the structure's existing points or are between the nodes. Therefore if they are between the nodes, the elements would be split into two parts, and a new list of points and elements would be defined. For example, in figure 9, the fp2 is a point load between the nodes p8 and p5. Therefore it split the element between the point p8 and p5 into two elements. And also add a point named P9 to the end of the points dictionary.

Code 5

```
#create coordinations variables
PX = str(obj.point[0])
PY = str(obj.point[1])
PZ = str(obj.point[2])
#assign a name to the chosen element
#element_name = get_Element([PX,PY,PZ],points)
#create a condition to replace the element with two new one and update the points dict
if getKeyByValue(points,[PX,PY,PZ]) == -1:
  element_name = get_Element([PX,PY,PZ],points)
  #calculate how many points
  lenPointDict = len(points)
  #name the new point
  name_Point = "P" + str(lenPointDict)
  #update it in the dict
  points.update({name_Point : [PX,PY,PZ]})
  #create the first element from the new point to the old one
  elem1 = (elements[element_name][0] ,name_Point)
  #create the first element from the old point to new point (the other way around)
  elem2 = (name_Point,elements[element_name][1])
   #Insert the new elements in the dict
  elements.update({element_name : [elem1[0],elem1[1]]})
   #calculate how many elements
   lenElementDict = len(elements)
   #assign a new name to the new elements and add them to the dict
   name_Element = "E" + str(lenElementDict)
   elements.update({name_Element : [elem2[0],elem2[1]]})
```

```
#Definition to get the Element where the point load is
def get_Element(point,Point_dict):
   #create a dictionary to calculate the nodes distances from the point load
   d = {}
   #transform the points from the points dict to floats
   point_f = (float(point[0]),float(point[1]),float(point[2]))
   #loop in the points dict to search for the points
   for p in Point_dict:
         point_dict_f = (float(Point_dict[p][0]),float(Point_dict[p][1]),float(Point_dict[p][2]))
         #If the rest of the division of the point angel is 10 then choose the point to eliminate the NOT perpendicular angels
         state = rs.Angle(point_f, point_dict_f, True)[1]%10
         if rs.Angle2((point_f,point_dict_f),((0,0,0),(0,0,10)))[0] == 90:
               state = rs.Angle(point_f, point_dict_f, True)[0]%10
         if state == 0:
               d.update({p: rs.Distance(point_f,point_dict_f)})
               #get the closest point from the dict and delete it
```

```
          P1 = getKeyByValue(d,min(d.values()))
          d.pop(P1)
          #get the second closest point
          P2 = getKeyByValue(d,min(d.values()))
          #detect the element that connects the two closest points to the new point and get the name
    if getKeyByValue(elements,[P1,P2]) != -1:
          elem_Name = getKeyByValue(elements,[P1,P2])
    if getKeyByValue(elements,[P2,P1]) != -1:
          elem_Name = getKeyByValue(elements,[P2,P1])
          return elem_Name
```

7. Linear loads:

After updating the dictionary, the user is asked to pick a Linear Load, Moment or Exit the dialog box. In this project, the term linear load refers to distributed load.

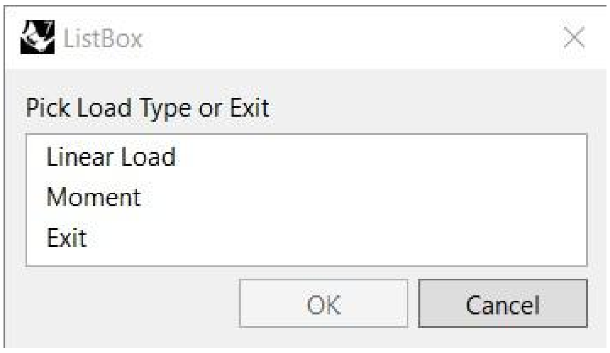


Figure 10: picking load type

7.1. Linear load Inputs:

There are three steps for the user to define a linear load.

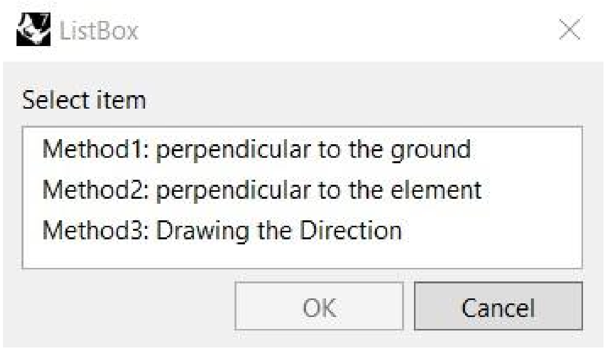
1. Selecting start and end point of the load
2. Selecting load direction
3. Choosing load magnitudes

First:

Choosing the start point and end point of the linear load.

Second:

Choosing one of the predefined directions for the load or drawing the direction as a line. Three different methods are defined in the code. Based on the selected methods, one of the functions from the class Linear_Load() would be called. The returning of these functions would be a direction as “dir” which will be stored in a list named “L”.



Select item

Method1: perpendicular to the ground
Method2: perpendicular to the element
Method3: Drawing the Direction

Figure 11: choosing linear load method

Method 1: perpendicular to the ground

Code 6

```
def Linear_Load_Method1(self):
  vec = rs.CreateVector(0, 0, -1)
  line = rg.Line(self.point1,self.point1 + vec)
  self.dir = line.Direction/line.Length
  return self.dir
```

Method 2: perpendicular to the element

Code 7

```
def Linear_Load_Method2(self):
  line = rg.Line(self.point1 ,self.point2)
  vec = line.Direction/line.Length
  self.dir = rs.CreateVector(vec.Z, 0, -1*vec.X)
  return self.dir
```

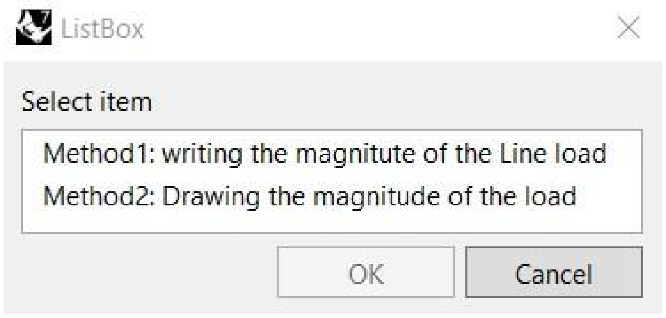
Method 3: drawing the load direction

Code 8

```
def Linear_Load_Method3(self):
  line = rg.Line(rs.GetPoint("draw the Direction of Load"),rs.GetPoint ())
  self.dir = line.Direction/line.Length
  return self.dir
```

Third:

After selecting the direction of the load, the user is asked to choose a method for defining the load's magnitude. Two methods have been defined. These functions return one magnitude for the start point of the linear load as “f1” and one for its end as “f2” which will be stored in the list named “L”. So if the linear load has the same magnitude as the start point and the endpoint is a Uniformly Distributed Load. If it has different values for the two magnitudes is a Non Uniformly Distributed Load.


ListBox

Select item

Method1: writing the magnitute of the Line load
Method2: Drawing the magnitude of the load

Figure 12: choosing linear load magnitude method

Method 1: Writing the magnitude of the line load

Code 9

```
def Line_Load_Magn1(self):
  self.f1 = rs.GetReal("Enter the Load Magnitude at the Start point"+" ("+
 unit_f + "/"+ unit_l + ")")
  self.f2 = rs.GetReal("Enter the Load Magnitude at the End point"+"(" +
 unit_f + "/"+ unit_l + ")")
   return self.f1, self.f2
```

Method 2: Drawing the magnitude of the line load

Code 10

```
def Line_Load_Magn2(self):
   draw1 = rg.Line(rs.GetPoint("Draw the Load Magnitude for start point "+"("+
 unit_f + "/"+ unit_l + ")"),rs.GetPoint())
   draw2 = rg.Line(rs.GetPoint("Draw the Load Magnitude for End point"+ "("+
 unit_f + "/"+ unit_l + ")"),rs.GetPoint())
   self.f1 = draw1.Length
   self.f2 = draw2.Length
   return self.f1, self.f2
```

7.2. Linear load visualization:

After taking all the Linear Load parameters from the user in step 5, the linear load class's last function, Visualize_Line_Load(j), is called with an argument, j, the index of the linear loads. In this function, different parameters related to the linear load, such as the integral of the load function, are being calculated. This resulting force is also called the **Equivalent Concentrated Load.** The centroid of the linear load is being calculated, which defines the Equivalent Concentrated Load's acting point**.** These are the parameters that seem to be necessary for calculating a Non Uniformly Distributed Load. But here, it is considered that only two types of distributed loads are being calculated; Rectangular and triangular loads. So the returning parameters of this function are not added to the final dictionary.

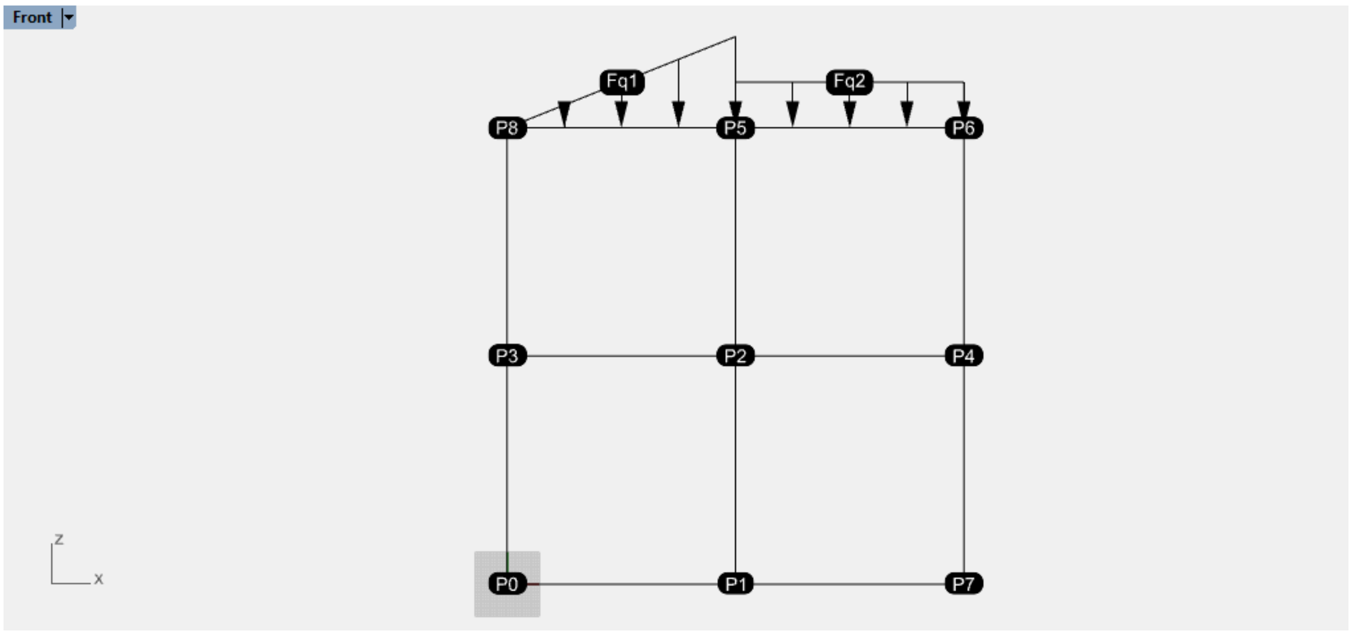


Figure 13: Visualization of the Linear Loads

Code 11

```python
def Visualize_Line_Load(self,j):
  #polyLine from 4 point
  #AREA
  #Centroid
  #centroid closest point on line
  p1p2 = rg.Line(self.point1,self.point2)
  vecp1p2= p1p2.Direction/4
  p1 = self.point1
  p2 =self.point2
  p3 = p1+(self.dir*-1*self.f1)
  p4 = p2+(self.dir*-1*self.f2)
  poly = rs.AddPolyline([p1,p2,p4,p3,p1])
  self.area = rs.Area(poly)
  c = rs.CurveAreaCentroid(poly)
  l = rg.Line(c[0] , c[0]+ self.dir * -1)
  intersect = rs.LineLineIntersection(p1p2, l)
  self.cent = intersect[0]


  dif= (self.f2-self.f1)/4
  if (p1 != p3):
        line1 = rs.AddLine(p1,p3)
        rs.CurveArrows(line1, 1)

  line2 = rs.AddLine(p1+vecp1p2,p1+vecp1p2+(self.dir*-1*(self.f1+dif)) )
  rs.CurveArrows(line2, 1)
  line3 = rs.AddLine(p1+2*vecp1p2,p1+2*vecp1p2+(self.dir*-1*(self.f1+2 *dif)))
  rs.AddTextDot("Fq" + str(j+1) ,p1+2*vecp1p2+(self.dir*-1*(self.f1+2* dif)))
  rs.CurveArrows(line3, 1)
  line4 = rs.AddLine(p1+3*vecp1p2,p1+3*vecp1p2+(self.dir*-1*(self.f1+3 *dif)))
  rs.CurveArrows(line4, 1)
  line5 = rs.AddLine(p2,p4)
```

```
rs.CurveArrows(line5, 1)

return self.cent , self.area
```

7.3. Adding Linear Load to the JSON Dictionary:

For adding the linear loads to the JSON dictionary, two if conditions have been defined that compare the coordinates of the start and endpoint of the linear load with the structure's Points dictionary, and if they are the same, it returns the name of the points as strings. Then it compares the name of these two points with the existing Elements dictionary, then assigns the name of the element to the related load when updating the dictionary. There are two conditions for checking the points with the elements, because the user might select the points of the linear load in the reverse order of the points of the element.(figure 14)(Code 12)
In the if conditions there is a function called getKeyByValue(List,item) that checks if an item exists in the list or not. If it exist, the returning parameter is the Key of the related item in the list but if not it returns -1.(Code 13)

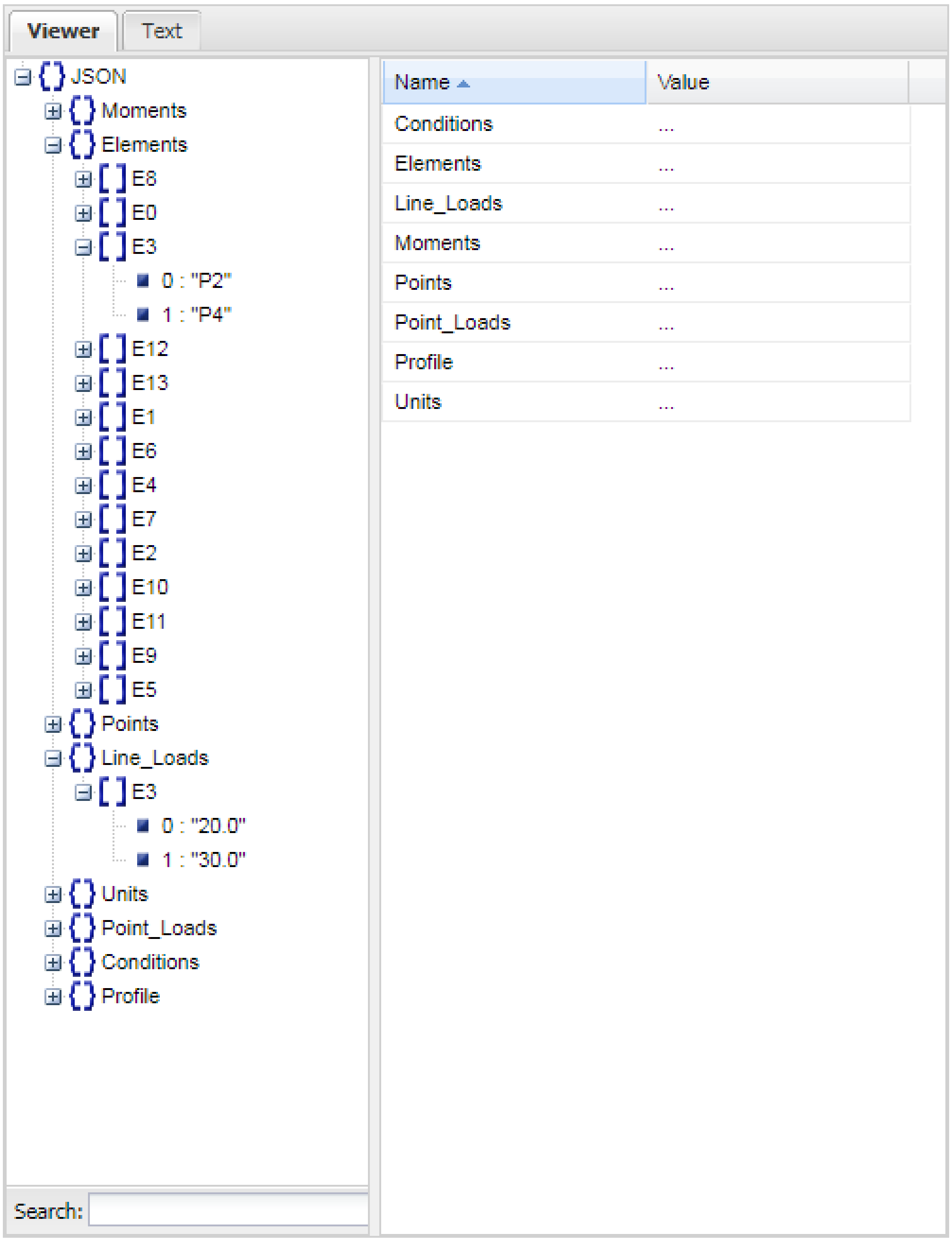
Code 12

```
PX1 = str(obj.point1[0])
PY1 = str(obj.point1[1])
PZ1 = str(obj.point1[2])
PX2 = str(obj.point2[0])
PY2 = str(obj.point2[1])
PZ2 = str(obj.point2[2])

if getKeyByValue(points,[PX1,PY1,PZ1]) != -1:
  P1_name = getKeyByValue(points,[PX1,PY1,PZ1])
if getKeyByValue(points,[PX2,PY2,PZ2]) != -1:
  P2_name = getKeyByValue(points,[PX2,PY2,PZ2])

if getKeyByValue(elements,[P1_name,P2_name]) != -1:
  E_Name = getKeyByValue(elements,[P1_name,P2_name])
  Line_Loads.update({E_Name : (str(obj.f1) ,str(obj.f2))})

if getKeyByValue(elements,[P2_name,P1_name]) != -1:
  E_Name = getKeyByValue(elements,[P2_name,P1_name])
  Line_Loads.update({E_Name : ( str(obj.f2), str(obj.f1))})
```



```
JSON
  Moments
  Elements
    E8
    E0
    E3
      0 : "P2"
      1 : "P4"
    E12
    E13
    E1
    E6
    E4
    E7
    E2
    E10
    E11
    E9
    E5
  Points
  Line_Loads
    E3
      0 : "20.0"
      1 : "30.0"
  Units
  Point_Loads
  Conditions
  Profile
```

Name	Value
Conditions	...
Elements	...
Line_Loads	...
Moments	...
Points	...
Point_Loads	...
Profile	...
Units	...

Figure 14: Linear Loads exported to JSON Dictionary

Code 13

```python
#Search key by the values and the indexes
def getKeyByValue(list,search_pointName):
    try:
            return list.keys()[list.values().index(search_pointName)]
    except ValueError:
            return -1
```

8. Defining Moments:

For the moments, it is necessary to specify the moment direction which is asked from the user (Figure 15). A class is defined to receive the value of the moment from the user. It has two main functions for the direction of the moments, which also visualize the user's defining moment (figure 16). This class returns the value and the acting point of the moment as “m” and “mpt” respectively. All the moments and the points are then added to the list name M.

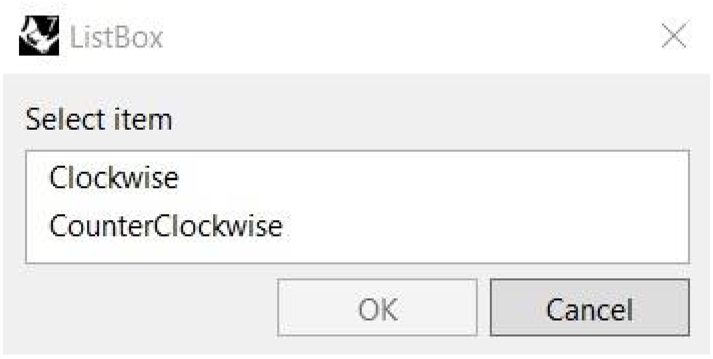


Figure 15: selecting moment direction

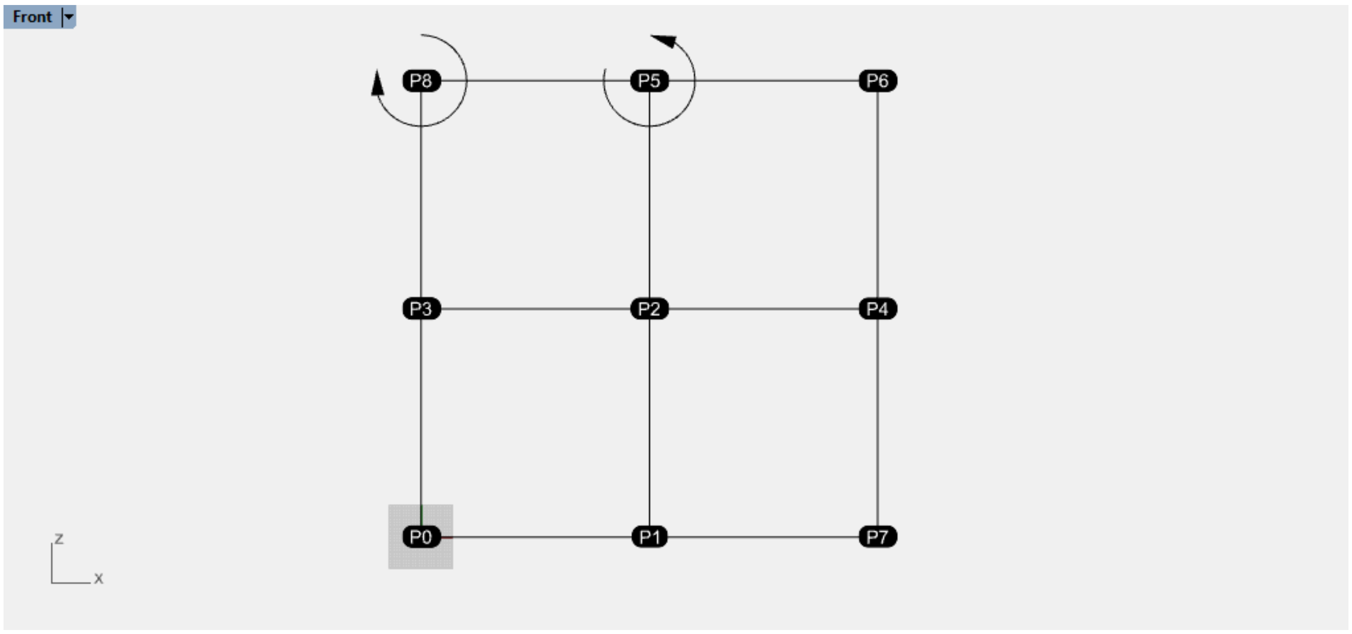


Figure 16: The visualization of the Moments

8.1. Moment Inputs:

- Selecting Moment point by user
- Selecting Moment direction by user
- Defining the moment value by user

Code 14

```
class Moments:
  def __init__(self, _m, _mpt):
        self.m = _m
        self.mpt = _mpt
```

8.2. Choosing moment direction and visualization:

Clockwise:

Code 15

```
def Clockwise(self,k,d):
  self.mpt = rs.GetPoint("Select Moment Location")
  self.m = rs.GetReal("Amount Of Moment?")
  vec= rg.Vector3d(0,1,0)
  pl = rg.Plane(self.mpt, vec)
  arc = rs.AddArc(pl , d , 285)
  rs.CurveArrows(arc, 2)
  return self.m, self.mpt
```

Counterclockwise:

Code 16

```
def Counterwise(self,k,d):
  self.mpt = rs.GetPoint("Select Moment Location")
  self.m = rs.GetReal("Amount Of Moment?")*-1
  vec= rg.Vector3d(0,1,0)
  pl = rg.Plane(self.mpt, vec)
  arc = rs.AddArc(pl , d , 285)
  rs.CurveArrows(arc, 1)
  return self.m, self.mpt
```

9. Exporting data to Json File:

In step 6, Point Loads and Moments are added to dictionaries with the same assigned name as the points dictionary. For this purpose, the coordinates of the point loads and moments are compared with the existing points dictionary then the returning point name is assigned as the Key of point loads and moments. After exporting Moments and Point Loads, the units are also added to the dictionary. (Figure 17)(Code 17)

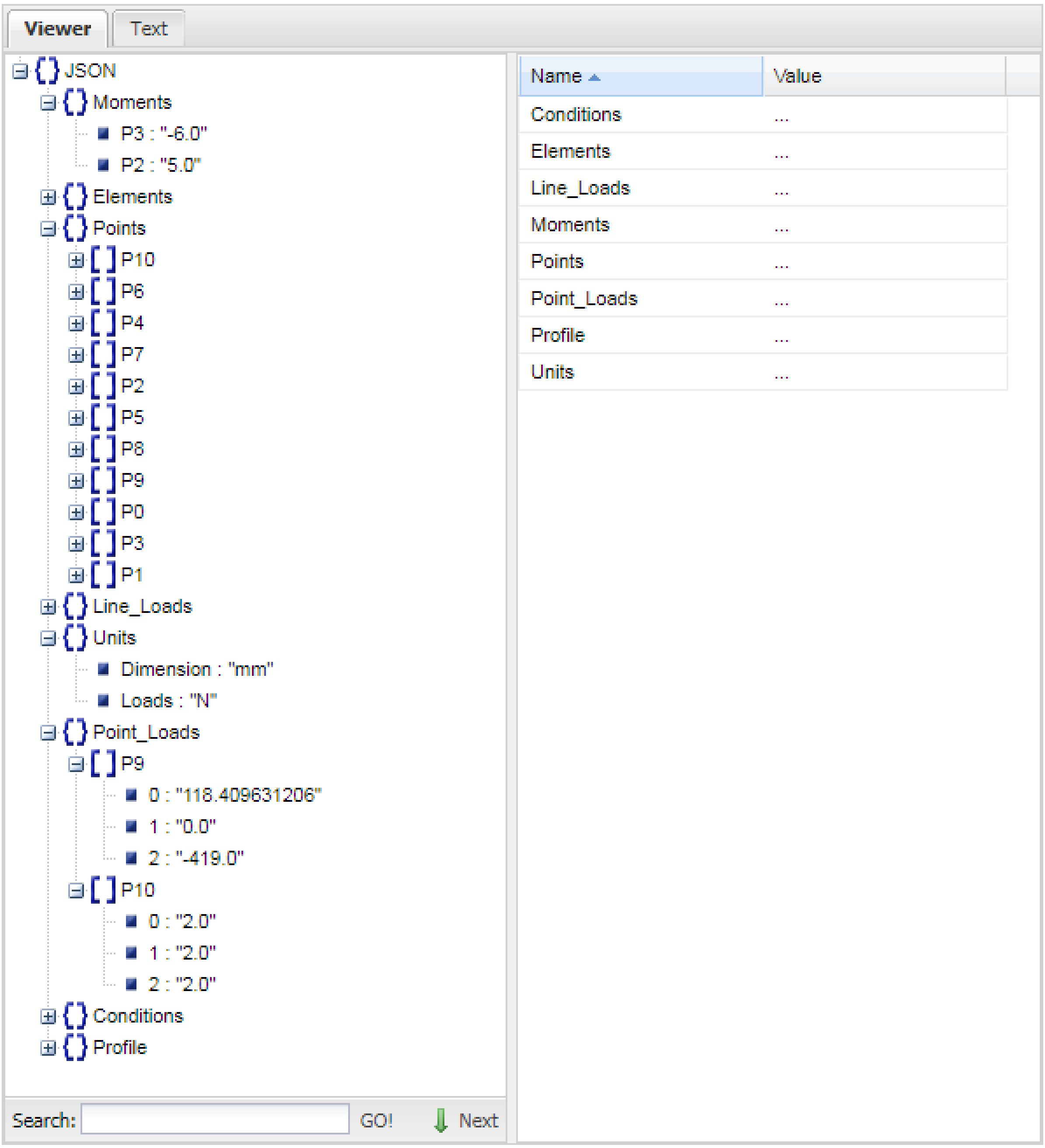
Code 17

```
Point_Loads = {}
for i,d in enumerate(P):
  #add the point loads and assign them to the existed points dict
  name = getKeyByValue(points, [str(d.point.X),str(d.point.Y),str(d.point.Z)])
  Point_Loads.update({name : (str(d.dir.X),str(d.dir.Y),str(d.dir.Z))})

Moments = {}
```

```python
for i,d in enumerate(M):
  #add the point loads and assign them to the existed points dict
  name = getKeyByValue(points, [str(d.mpt.X),str(d.mpt.Y),str(d.mpt.Z)])
  Moments.update({name : (str(d.m))})

Units = {}
Units.update({"Loads" : (unit_f)})
Units.update({"Dimension" : (unit_l)})
json_dict = {"Points": points, "Elements": elements, "Conditions": conditions ,
 "Profile": profile, "Point_Loads": Point_Loads, "Line_Loads": Line_Loads ,
 "Moments": Moments, "Units": Units}
```



```
JSON
  Moments
    P3 : "-6.0"
    P2 : "5.0"
  Elements
  Points
    P10
    P6
    P4
    P7
    P2
    P5
    P8
    P9
    P0
    P3
    P1
  Line_Loads
  Units
    Dimension : "mm"
    Loads : "N"
  Point_Loads
    P9
      0 : "118.409631206"
      1 : "0.0"
      2 : "-419.0"
    P10
      0 : "2.0"
      1 : "2.0"
      2 : "2.0"
  Conditions
  Profile
```

Name ▲	Value
Conditions	...
Elements	...
Line_Loads	...
Moments	...
Points	...
Point_Loads	...
Profile	...
Units	...

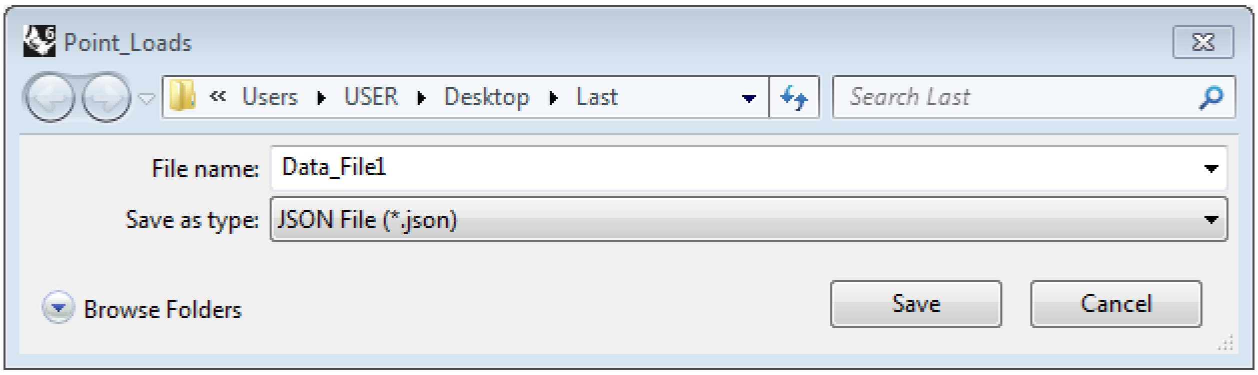
Figure 17: The exported JSON file Containing Point Loads and Moments data

In the end, the user is asked to choose a folder and a name to save the JSON file. (Code 18)(figure 18)

Code 18

```
#Get the file name for the new file to write
filter = "JSON File (*.json)|*.json|All Files (*.*)|*.*||"
filename = rs.SaveFileName("Point_Loads", filter)

# If the file name exists, write a JSON string into the file.
if filename:
  # Writing JSON data
  with open(filename, 'w') as f:
        json.dump(json_dict, f)
```


Point_Loads

« Users ▸ USER ▸ Desktop ▸ Last

File name: Data_File1

Save as type: JSON File (*.json)

Figure 18: The user can save the JSON file

10. Conclusion:

In conclusion, as shown in Figure 19, after receiving the geometry as a rhino file from Group 2, different types of load, including Point Load, Distributed Load, and Moment, were picked by the user in the rhino interface and then have been visualized. The Point Loads are named "fp", and the Linear Loads are named "fq".
As can be seen from figure 20, alongside the visualization, first, all the structure data, including Profile, Conditions, Points, and Elements, has been imported from Group2's JSON file. After defining the Point Loads, the Elements and Points dictionaries have been updated, and new elements have been added to these dictionaries.
The code's output is a JSON file which can be seen on the right side of figure 20. this JSON file consists of old, updated, and new dictionaries including Point_Loads, Line_Loads, Moments, and Units.

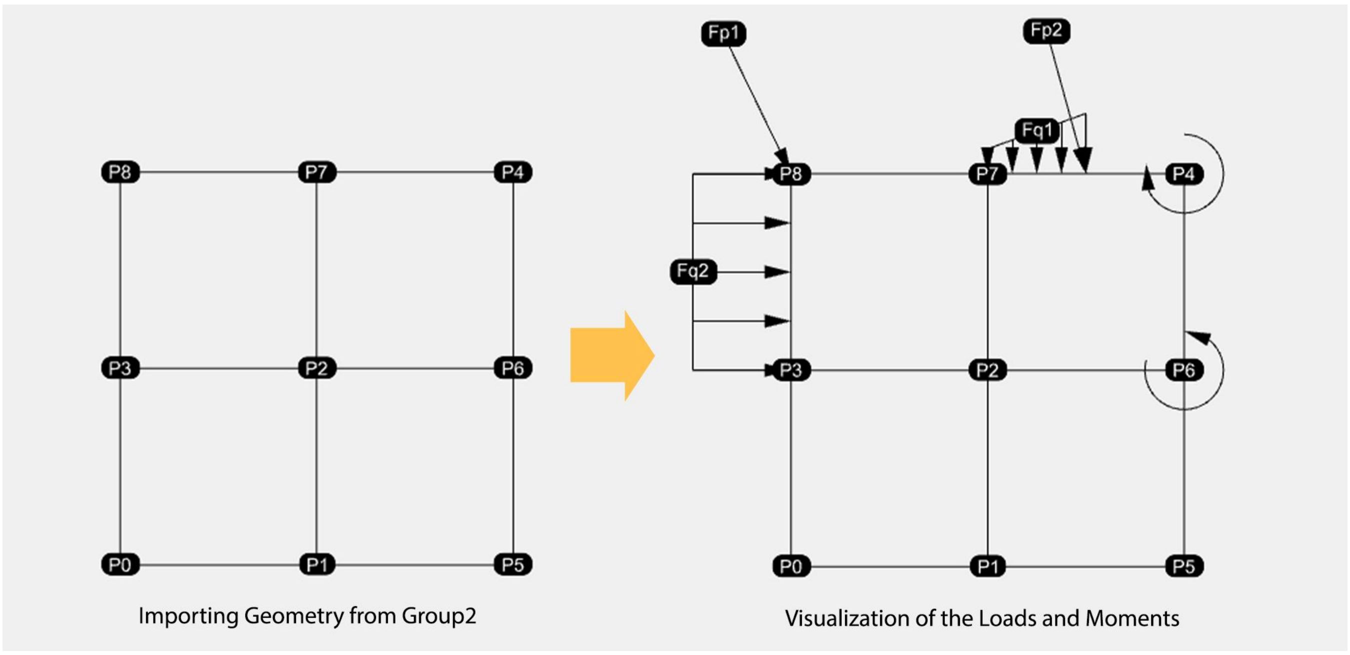
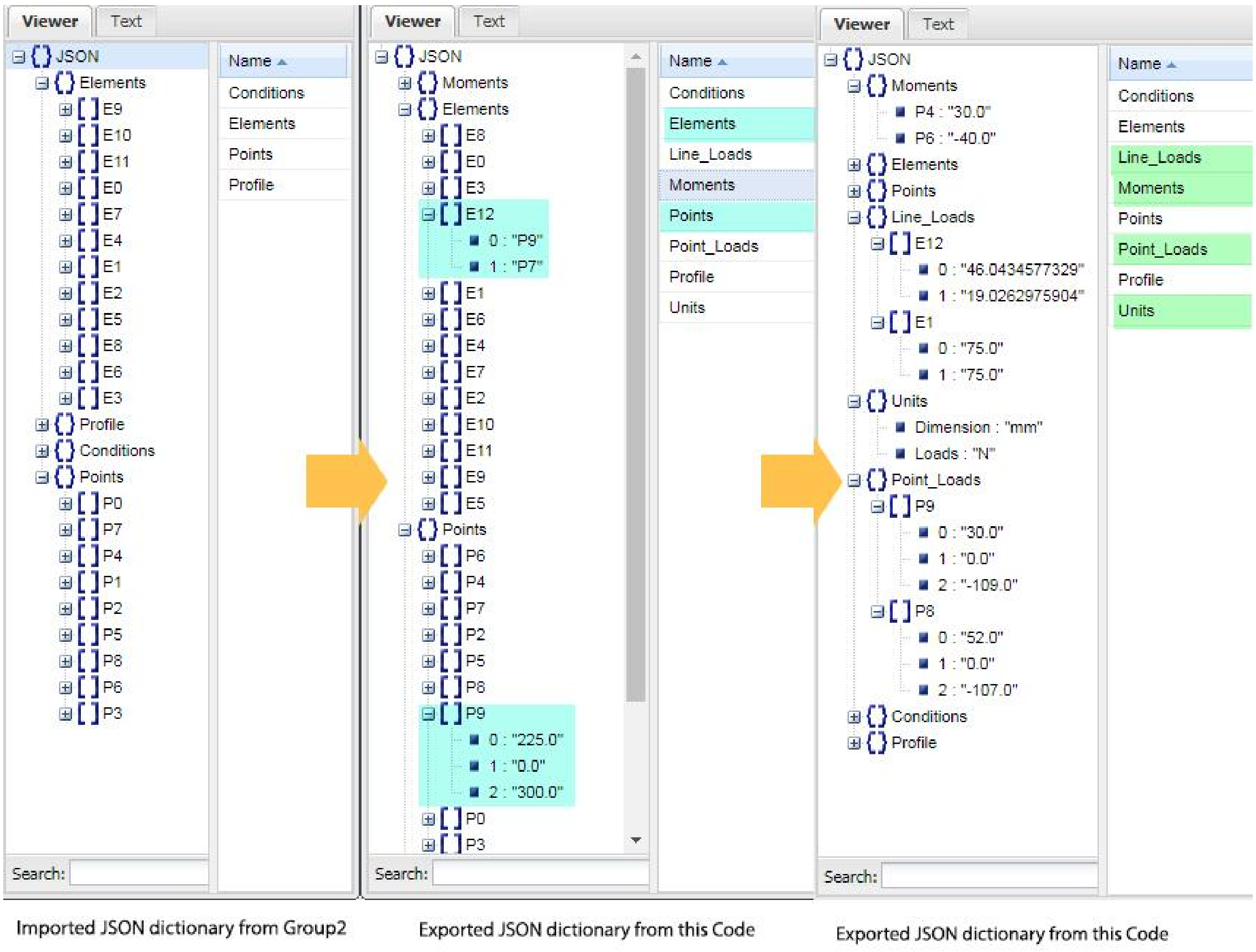

P8
P7
P4
P3
P2
P6
P0
P1
P5
Importing Geometry from Group2
Fp1
Fp2
Fq1
Fq2
Visualization of the Loads and Moments

Figure 19: Imported geometry from group two and the user-defined loads, and visualization



Imported JSON dictionary from Group2

```
Viewer | Text
⊟ { } JSON
  ⊟ { } Elements
    ⊞ [ ] E9
    ⊞ [ ] E10
    ⊞ [ ] E11
    ⊞ [ ] E0
    ⊞ [ ] E7
    ⊞ [ ] E4
    ⊞ [ ] E1
    ⊞ [ ] E2
    ⊞ [ ] E5
    ⊞ [ ] E8
    ⊞ [ ] E6
    ⊞ [ ] E3
  ⊞ { } Profile
  ⊞ { } Conditions
  ⊟ { } Points
    ⊞ [ ] P0
    ⊞ [ ] P7
    ⊞ [ ] P4
    ⊞ [ ] P1
    ⊞ [ ] P2
    ⊞ [ ] P5
    ⊞ [ ] P8
    ⊞ [ ] P6
    ⊞ [ ] P3
Search:

Name ▲
Conditions
Elements
Points
Profile
```

Exported JSON dictionary from this Code

```
Viewer | Text
⊟ { } JSON
  ⊞ { } Moments
  ⊟ { } Elements
    ⊞ [ ] E8
    ⊞ [ ] E0
    ⊞ [ ] E3
    ⊟ [ ] E12
        0 : "P9"
        1 : "P7"
    ⊞ [ ] E1
    ⊞ [ ] E6
    ⊞ [ ] E4
    ⊞ [ ] E7
    ⊞ [ ] E2
    ⊞ [ ] E10
    ⊞ [ ] E11
    ⊞ [ ] E9
    ⊞ [ ] E5
  ⊟ { } Points
    ⊞ [ ] P6
    ⊞ [ ] P4
    ⊞ [ ] P7
    ⊞ [ ] P2
    ⊞ [ ] P5
    ⊞ [ ] P8
    ⊟ [ ] P9
        0 : "225.0"
        1 : "0.0"
        2 : "300.0"
    ⊞ [ ] P0
    ⊞ [ ] P3
Search:

Name ▲
Conditions
Elements
Line_Loads
Moments
Points
Point_Loads
Profile
Units
```

Exported JSON dictionary from this Code

```
Viewer | Text
⊟ { } JSON
  ⊟ { } Moments
      P4 : "30.0"
      P6 : "-40.0"
  ⊞ { } Elements
  ⊞ { } Points
  ⊟ { } Line_Loads
    ⊟ [ ] E12
        0 : "46.0434577329"
        1 : "19.0262975904"
    ⊟ [ ] E1
        0 : "75.0"
        1 : "75.0"
  ⊟ { } Units
      Dimension : "mm"
      Loads : "N"
  ⊟ { } Point_Loads
    ⊟ [ ] P9
        0 : "30.0"
        1 : "0.0"
        2 : "-109.0"
    ⊟ [ ] P8
        0 : "52.0"
        1 : "0.0"
        2 : "-107.0"
  ⊞ { } Conditions
  ⊞ { } Profile
Search:

Name ▲
Conditions
Elements
Line_Loads
Moments
Points
Point_Loads
Profile
Units
```

Figure 20: Adding new Elements and Dictionaries to the Imported JSON file from Group 2