

2.5 La aplicación final puede ser modificada por alguien externo al equipo de desarrollo.

El objetivo de este documento es que cualquier persona con conocimientos de matemática, computación y del problema que resuelve la aplicación pueda ser capaz de modificar el código fuente para adaptarlo a nuevas necesidades.

Con este documento pretendemos informar a alguien externo como poder realizar cualquier modificación en la aplicación, para lograr esto explicaremos cual es el flujo de trabajo de esta y que funcionalidades tienen cada una de sus partes. Para realizar cualquier cambio es necesario tener conocimientos básicos del lenguaje de programación *python* y del framework *flask* que se utiliza en el mismo lenguaje para el trabajo con la página web. Para trabajar con documentos *excel* se utilizó *xls* por tanto si se desea realizar algún cambio en la forma de leer y exportar datos con *excel* puede ver el *.py* pertinente

Para empezar tomaremos de referencia el *.py main* y hablaremos de los distintos módulos que esta utiliza a medida que estos se van cargando, explicando las funcionalidades de los mismos y el uso que se les dan, por si se desea realizar algún cambio en estos o agregar algo nuevo el programador sepa donde encontrar lo que necesita para modificarlo si ya esta creado.

Brindaremos un orden de como se va ejecutando el programa y pondremos imágenes del código fuente explicando que realiza cada función. En el caso del *html* mostraremos el código de este y como se visualiza el mismo en pantalla.

main.py

Iniciaremos explicando las funciones que se encuentran en este *.py*

```
from utils.flask import *
from utils.file_work import get_values_from_excel

from flask import Flask, request, render_template
import webbrowser
```

Estos *import* se utilizan para cargar todo lo relacionado con la librería de *flask*, lo que se encuentra en *utils* puede ser modificado fácilmente pues son funciones creados por nosotros para facilitar el trabajo con *Flask*

Al inicio del *.py* seleccionamos quien será el host, el puerto, asignamos quien guardará la estructura de *Flask*.

```
# VARIABLES
_host = 'localhost'
_port = 3000

# APP
app = Flask(__name__)
```

Seguido tenemos las funciones que utiliza el framework para saber que trabajo realizar para cada ruta.

```

@app.route('/', methods=["GET"])
def home():
    return render_template('home.html')

@app.route('/index', methods=["GET", "POST"])
def index():
    if request.method == 'POST':
        if request.form['excel'] == 'false':
            n, l, a, b, variable = find_val_to_model(request)

            params = create_demand(request, variable)

            vars_q, vars_t = get_results(n, a, b, l, variable, params)
            return render_template('results.html', vars_q=vars_q, vars_t=vars_t, n=n, l=l, a=a, b=b, variable=variable)
        else:
            n = int(get_values_from_excel(0))
            a, b = get_values_from_excel(1)
            l = get_values_from_excel(2)
            variable, params1, params2 = get_values_from_excel(3)

            params = create_demand_excel(variable, params1, params2)

            vars_q, vars_t = get_results(n, a, b, l, variable, params)
            return render_template('results.html', vars_q=vars_q, vars_t=vars_t, n=n, l=l, a=a, b=b, variable=variable)

    return render_template('index.html')

@app.route('/about', methods=["GET"])
def about():
    return render_template('about.html')

```

La 1ra función es un *GET* de la página que se muestra al acceder al sitio.

La 2da función trabaja con *index* que es la página que muestra las opciones para rellenar los datos y muestra los resultados obtenidos. Para mostrar la página utiliza un *GET* y una vez se insertan los datos y se desea obtener una solución se realiza un *POST*

Si esa función se llama con un *GET* `request.method == "POST"` será falso y por tanto solo mostrará la página *index.html*, en caso de ser un *POST* si pasará el primer *if* y según si los datos se pasaron con un *excel* o se entraron a mano es que tomará los valores, mostrando luego una nueva página llamada *result.html* donde se presentan los resultados obtenidos de la resolución del modelo. Las diferentes funciones que se utilizan en cada caso se presentarán más adelante en las secciones dedicadas a sus respectivos *.py*

La 3ra función es para cuando se acceda a página *about*, la cual solo muestra en pantalla información y por tanto es un *GET*

demand.py

El otro módulo que utilizamos es el *demand.py*, en este se manejan las herramientas que se utilizan en el *frontend*, empezando por una clase que hereda de *Enum* llamada *distribution* que se utiliza para el *dropbox* del *index.html* que permite al usuario seleccionar que tipo de variable aleatoria desea elegir.

```

class distribution(Enum):
    UNIFORM = 'uniform',
    EXP = 'exponential',
    GAMMA = 'gamma',
    NORMAL = 'normal',
    BINARY = 'binary',
    GEOMETRIC = 'geometric',
    ESCENARIOS = 'scenarios'

```

Luego tiene la función *demand* que recibe el tipo de distribución que se seleccionó desde el *frontend*, los parámetros de esa distribución y un N que representa la cantidad de casos si es por escenarios y a partir de aquí se genera la demanda basándose en las funciones del *random_variables.py*

```
# generacion de la demanda
def demand(_distribution, params, N):
    dist = None
    if _distribution == distribution.UNIFORM.value[0]:
        dist = uniform(params[0], params[1])

    elif _distribution == distribution.EXP.value[0]:
        dist = exponential(params[0])

    elif _distribution == distribution.GAMMA.value[0]:
        dist = gamma(params[0], params[1])

    elif _distribution == distribution.NORMAL.value[0]:
        dist = normal(params[0], params[1])

    elif _distribution == distribution.BINARY.value[0]:
        dist = binary(params[0], params[1])

    elif _distribution == distribution.GEOMETRIC.value[0]:
        dist = geometric(params[0])

    else:
        dist = escenarios(params[0], params[1])

    demands = []
    for _ in range(N):
        demands.append(dist.get())

    return demands
```

Por tanto si se desea agregar alguna otra forma de calcular la demanda y acceder a ella desde el *frontend* se deberá agregar su nombre en el *Enum* y especificar como se calcula en la función *demand*. Si se desea agregar alguna otra forma de generar variables aleatorias puede hacerlo aquí pero para mantener la estructura de la aplicación sería mejor que las agregase en *random_variables.py*

file_work.py

En este módulo tenemos las funciones relacionadas con el trabajo con documentos *excel*, por eso se importa *xlrd*. Aquí tenemos las funciones encargadas de abrir el *excel* (con la función *read_excel*), *parsear* sus valores a *python* (con las funciones *get_values_from_excel* y *parse_variable*) y la función *write_output* que se encarga de exportar la salida. Si se desea cambiar la forma en la que se toman los datos de *excel* o la forma en que estos se exportan deberá cambiar las funciones pertinentes.

Recordar que este modulo se carga en *main.py*, y se utiliza en la función *index*, por tanto si se desea hacer algún cambio en la entrada de alguna función puede encontrar su llamado en *main.py*

flask.py

En este módulo se encuentran las funciones que se llaman directamente utilizando las rutas de *flask* en el *main.py*, la principal tarea consiste en enlazar los datos que se agregan desde el *frontend* y traerlos al *backend*

find_val_to_model conecta la entrada y devuelve esta como valores en *python*. Si se desea agregar más entradas puede hacerlo en esta función o crear una nueva función en este *.py* y hacer el llamado a partir de la ruta correspondiente.

create_demand y *create_demand_excel* crean la demanda a partir de la entrada de valores dados por el usuario o a partir de un documento *excel* respectivamente.

Con *get_result* se genera la demanda *d* y se intenta resolver el modelo, luego se hace *return* a los dos listados de variables las cuales tienen el valor con el que se alcanza el óptimo de la función objetivo. Si se quisiera hacer algún cambio en la forma de generar la demanda o en la estructura del modelo puede ir a las funciones que se encargan de realizar esa tarea y en esta solo se aseguraría de recibir la entrada necesaria y dar la salida correcta.

```
def get_results(n, a, b, l, variable, params):
    d = demand(variable, params, n)

    try:
        vars_q, vars_t = solveModel(n, l, a, b, d)
    except:
        vars_q = []
        vars_t = []

    write_output(n, a, b, l, variable, vars_q, vars_t)

    return vars_q, vars_t
```

random_variables.py

En este módulo tenemos todas las fórmulas que utilizamos para calcular variables aleatorias a partir de un random. Como variables aleatorias continuas tenemos:

- uniforme
- exponencial
- gamma
- normal

```

class uniform:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def get(self):
        U = r.random()
        return (self.b-self.a)*U + self.a

class exponential:
    def __init__(self, _lambda):
        self._lambda = _lambda

    def get(self):
        U = r.random()
        return -(1/self._lambda)*math.log(U)

class gamma:
    def __init__(self, n, _lambda):
        self.n = n
        self._lambda = _lambda

    def get(self):
        Un = math.prod([r.random() for _ in range(self.n)])
        return -(1/self._lambda)*math.log(Un)

class normal:
    def __init__(self, mu, o_2):
        self.mu = mu
        self.o_2 = o_2

    def get(self):
        exp = exponential(1)
        while True:
            Y = exp.get()
            U = r.random()
            if U <= math.exp((-1/2)*(Y-1)**2):
                break

        U = r.random()
        Z = Y if U < 0.5 else -Y

        # Z = (X - miu)/o ~ N(0,1) => X = Zo + miu ~ N(miu,o^2)
        return Z*math.sqrt(self.o_2) + self.mu

```

Y como variables aleatorias discretas tenemos:

- binaria
- geométrica

```
# discretas
class binary:
    def __init__(self, n, p):
        self.n = n
        self.p = p

    def get(self):
        X = 0
        for _ in range(self.n):
            U = r.random()
            if U <= self.p:
                X += 1
            else:
                X += 0
        return X

class geometric:
    def __init__(self, p):
        self.p = p

    def get(self):
        U = r.random()
        return math.ceil(math.log(U) / math.log(1-self.p))
```

Y por ultimo tenemos el caso por escenarios

```
# por escenarios
class escenarios:
    def __init__(self, demands, probs):
        self.probabilities = probs
        self.demands = demands
        self.order()

    def order(self):
        for i in range(len(self.probabilities)):
            for j in range(i+1, len(self.probabilities)):
                if self.probabilities[i] > self.probabilities[j]:
                    self.probabilities[i], self.probabilities[j] = self.probabilities[j], self.probabilities[i]
                    self.demands[i], self.demands[j] = self.demands[j], self.demands[i]

    def get(self):
        u = r.random()
        sum_p, i = 0, len(self.probabilities)

        # recorre el array en reverso, comenzado con la
        # probabilidad mayor
        while i >= 0:
            i -= 1
            sum_p += self.probabilities[i]
            if u <= sum_p:
                return self.demands[i]
```

Si se desea agregar alguna otra forma de generar variables aleatorias puede agregarla aquí para mantener la estructura del proyecto pero para utilizar dicha variable para calcular la demanda se deben modificar los métodos del *demand.py*

solveModel.py

```
from gekko import GEKKO
```

Con esta línea cargamos el módulo de *GEKKO* el cual utilizamos para representar el modelo y resolverlo. Para ello utilizamos las funciones que veremos a continuación:

La función *solveModel*

Primero se crean las variables que utiliza *GEKKO* para representar el modelo q_i y t_{ij} y se guardan en los listados *vars_q* y *vars_t* (para realizar este proceso están las funciones *assignVar_q* y *assignVar_t* respectivamente).

En *m* se asigna la estructura *GEKKO* la cual recibirá el modelo con la función objetivo y las condicionales.

conditionals(m, L, vars_q, vars_t, D) coloca en *m* las restricciones del modelo(se verá la función más adelante), se le pasa como parámetros la estructura *GEKKO*, la matriz *L*, los dos listados de variables y la Demanda

Luego se le especifica a *m* cual será su función objetivo mediante un llamado a *obj_fun* que recibe como parámetro el listado de variables *vars_q* y los listados *a* y *b*

Ya con esto tenemos el modelo con su función objetivo y sus condicionales así que se llama al método *solve* de la estructura *GEKKO*, el cual resuelve el modelo y en cada variable deja el valor que debe tener esta para conseguir el óptimo (recordar que en este caso esas variables se encuentran en los listados *vars_q* y *vars_t*) .

Como esta función representa la construcción del modelo y la solución de este con el método *solve* es preferible mantenerlo de esta forma, si se desea cambiar las condicionales del modelo o la función objetivo aquí solo se deben editar los parámetros que estas reciben de entrada y retornar las variables correspondientes.

```
def solveModel(N,L,a,b,D):
    m = GEKKO(remote=False) #Initialize gekko
    vars_q = assignVar_q(m,N)
    vars_t = assignVar_t(m,L)

    conditionals(m,L,vars_q,vars_t,D)
    m.Obj(obj_function(vars_q,a,b))
    m.options.IMODE = 3 # Steady state optimization
    m.solve()

    return vars_q, vars_t
```

obj_function será la función objetivo del modelo que se desea resolver, en caso de tener otra función objetivo basta con cambiar la implementación de este método y devolver la función que se desea. En este caso *q*, *a* y *b* son listados con variables, pero se puede modificar a conveniencia.

```
#funcion objetivo
def obj_function(q,a,b):
    f = 0
    for i in range(len(q)):
        f += a[i]*q[i] + b[i]*q[i]**2
    return f
```

En *conditionals* se especifican las ecuaciones o inecuaciones que se utilizaran como restricciones del modelo, note que cualquier ecuación que se desee agregar se debe colocar en *m.Equation(< EquationBody >)*, donde *m* es la estructura *GEKKO* que representa el modelo. En nuestro caso se agregan tantas condicionales como largo tenga la lista *L* de entrada.

Nótese que la condicional debe de satisfacer que devuelve *Bool*, o sea debe existir una relación de igualdad o desigualdad al final. Si se desean utilizar otras condicionales basta con especificar:

m.Equation(Conditional1)

m.Equation(Conditional2)

m.Equation(Conditional3)

...

m.Equation(Conditionaln)

donde m es la estructura *GEKKO* que representa el modelo.

```
#restricciones
def conditionals(m:GEKKO,L:list,assignVar_q:list,assignVar_t:list,D:list):
    count = len(L)
    for i in range(count):
        m.Equation(assignVar_q[i] - sum_T_Out(i,assignVar_t,L) + sum_T_In(i,assignVar_t,L) >= D[i])
```

En los métodos *assignVar_q* y *assignVar_t* se crean las variables a evaluar y se colocan en listados. Estas variables se crean con el método *m.Var* y se van guardando en el listado correspondiente. Al finalizar se devuelve dicho listado. (Nótese que en *assignVar_t* se utiliza un listado de listas xq se está trabajando con una matriz bidimensional)

```
#Generacion de lista con las q_i
def assignVar_q(m:GEKKO,count:int):
    varList_q = []
    for i in range(count):
        varList_q.append(m.Var(lb=0))
    return varList_q

#Generacion de matriz con las t_ij donde (i,j) es una arista de L a partir de la matriz L
def assignVar_t(m:GEKKO,L:list):
    rows = columns = len(L)
    varList_t = []
    for i in range(rows):
        t_row = []
        for j in range(columns):
            if L[i][j] >= 0:
                t_row.append(m.Var(lb=0))
            else:
                t_row.append(-1)
        varList_t.append(t_row)
    return varList_t
```

Por ultimo tenemos los métodos auxiliares *sum_T_Out* y *sum_T_In* que computan la energía que sale y entra a cada nodo a partir de la fórmula, donde *node* es el id del nodo del que se desea computar la energía que sale o entra.


```

# energia que sale del nodo i
def sum_T_Out(node,assignVar_t,L):
    count = len(L)
    sum = 0
    for j in range(count):
        termicalLoss = L[node][j]
        if(termicalLoss >= 0):
            tvar = assignVar_t[node][j]
            sum += tvar + (termicalLoss / 2) * tvar**2
    return sum

# energia que entra al nodo i
def sum_T_In(node,assignVar_t,L):
    count = len(L)
    sum = 0
    for i in range(count):
        termicalLoss = L[i][node]
        if(termicalLoss >= 0):
            tvar = assignVar_t[i][node]
            sum += tvar - (termicalLoss / 2) * tvar**2
    return sum

```

En este módulo se encuentra todo lo relacionado con la estructura del modelo y las funciones relacionadas con este, por tanto si no se desea cambiar nada del modelo en cuestión no se debe modificar ninguna función de aquí, si se desea cambiar el modelo, no es necesario acceder a nada externo excepto si se desea cambiar algún parámetro de entrada, en ese caso deberá ir al *flask.py* en la función *get_results* y especificar esos parámetros donde se llama a la función *solveModel*

about.html

Este *html* contiene la información de los pasos a realizar para utilizar la aplicación y resolver el problema, cualquier cambio que se desee realizar en como utilizar la aplicación debería dejarlo reflejado en el about.

home.html

Es la presentación de la app, si desea cambiar la presentación puede cambiarla aquí o modificar la aplicación para que abra directamente en el index.html, está solo por motivos estéticos.

index.html

A continuación pasaremos a explicar como está distribuido el *html* usando *flask* por si se desea hacer algún cambio sobre este se entienda como afecta a la aplicación.

Primero se muestra un *textbox* para números donde se coloca la cantidad de nodos del problema y un botón Create matrix que genera una matriz a partir de la cantidad de nodos que se insertan. A continuación podemos ver como se ve en el navegador y el código correspondiente

A screenshot of a web form. At the top, there is a number input field with a placeholder 'N'. Below it is a button labeled 'Create matrix'.

```
<input type="number" required="true" min="1" id="n" name="n" placeholder="N" /><br />
<input type="text" hidden id="excel1" name="excel" value="false" /> <br />
<input type="button" id="btn_create" value="Create matrix" onclick="create()"/> <br />
```

Al pulsar el botón se nos muestran dos listados y una matriz para rellenar, el código es el que podemos ver a continuación en el *html*

A screenshot of the web form after clicking the 'Create matrix' button. It displays three matrices, each with a header row and a header column. The matrices are labeled 'a_matrix', 'b_matrix', and 'l_matrix'.

a_matrix	
	1 2 3
1	1 2 1

b_matrix	
	1 2 3
1	3 1 2

l_matrix	
	1 2 3
1	4 3 5
2	3 2 1
3	2 2 1

```
<div id="a_matrix" required="true"></div> <br />
<div id="b_matrix" required="true"></div> <br />
<div id="l_matrix" required="true"></div> <br />
```

Estos valores pasan al *backend* como las variables *l* (para *l_matrix*), *a* (para *a_matrix*) y *b*, (para *b_matrix*) en la función *index* del *main.py* cuando se le haga *post*.

Luego tenemos la selección de variable y su respectivo código en *html*. Esta selección es un *dropbox* con las distintas distribuciones separadas por tipo y luego uno o dos *textbox* aparecerán para seleccionar los parámetros de la variable aleatoria que se utilizará para generar la Demanda. En el caso de *por_escenarios* aparecerá una lista para rellenar la probabilidad de cada escenario.

The image contains two screenshots of a web form interface. The top screenshot shows a dropdown menu titled "Choose a distribution for demand:" with "binary" selected. The menu is open, displaying categories: "None" (with "none" below it), "Discrete" (with "binary" and "geometric" below it), "Continua" (with "uniform", "exponential", "gamma", and "normal" below it), and "Scenarios" (with "scenarios" below it). The "binary" option is highlighted. The bottom screenshot shows the same form after selection, with the dropdown set to "binary" and two input fields labeled "n" and "p" appearing below it.

Choose a distribution for demand:

binary ▾

None
none

Discrete
binary
geometric

Continua
uniform
exponential
gamma
normal

Scenarios
scenarios

Choose a distribution for demand:

binary ▾

n

p

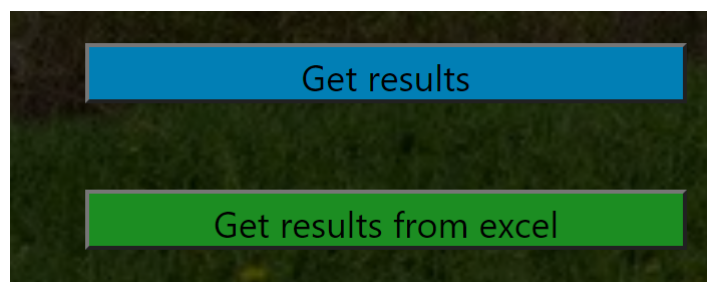
```

<label for="variable">Choose a distribution for demand:</label>
<select name="variable" id="variable" value="" onclick="create_params()">
  <optgroup label="None">
    <option value="none">none</option>
  <optgroup label="Discrete">
    <option value="binary">binary</option>
    <option value="geometric">geometric</option>
  </optgroup>
  <optgroup label="Continua">
    <option value="uniform">uniform</option>
    <option value="exponential">exponential</option>
    <option value="gamma">gamma</option>
    <option value="normal">normal</option>
  </optgroup>
  <optgroup label="Scenarios">
    <option value="scenarios">scenarios
    </option>
  </optgroup>
</select>

<div id="params"></div> <br />
<div id="intervals"></div> <br />

```

Y para finalizar tenemos el botón *send* que al ser de tipo *submit*, hará un llamado a la función *index* del *main.py* pero con *request.method* siendo *POST* por lo que esta vez pasará por *solve_model* y devolverá las listas con las soluciones del modelo, mostrándolas en otra página. También tenemos otro botón para cargar los datos a partir de un *Excel* que llama a la función encargada de transformar los datos desde *Excel* a *python*. El código de los botones es el siguiente:



```

<button type="submit" style="background-color: #017fb5;" onclick="parse_data(event)">
  Get results
</button>
</form>

<form action="/index" method="POST" class="form">
  <input type="text" hidden id="excel" name="excel" value="false" /> <br />
  <button type="submit" style="background-color: rgb(28, 141, 34);" onclick="parse_data_from_excel()">
    Get results from excel
  </button>

```

results.htm

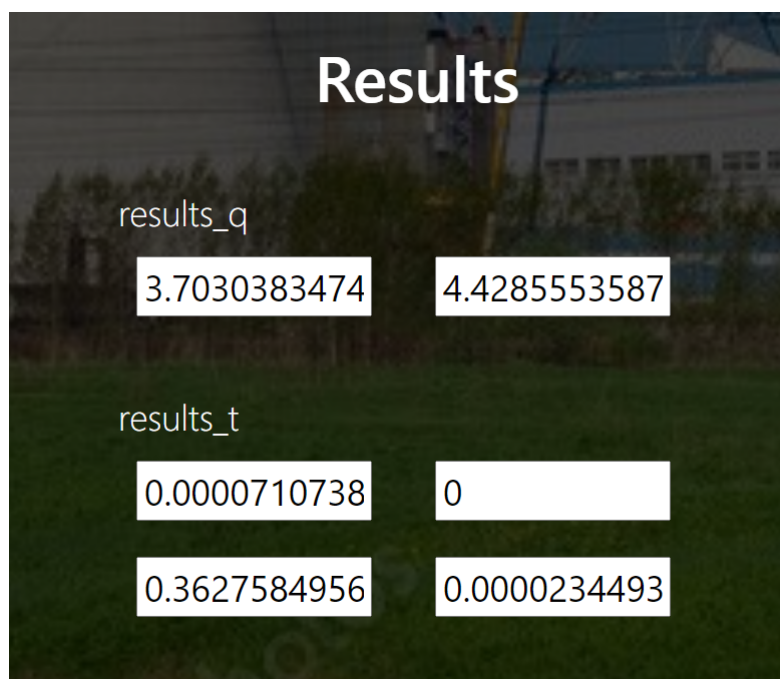
En esta página damos los resultados del modelo, el script para dibujarlo en forma de matriz se encuentra en el propio *html* con el nombre *draw_matrix*. Puede cambiarlo para mostrar los resultados de la forma en que guste, los listados con la solución del modelo se encuentra en:

```
var vars_q = {{vars_q}}  
var vars_t = {{vars_t}}
```

Y se muestran en la página con el siguiente código:

```
<h2>Results</h2>  
</div> <br />  
  
<div id="results_q"></div> <br />  
  
<div id="results_t"></div> <br />
```

Quedando de la siguiente forma:



Results	
results_q	
3.7030383474	4.4285553587
results_t	
0.0000710738	0
0.3627584956	0.0000234493