1. For the Motor Design problem, use the First Order Reliability method (FORM) to compute the probably of meeting a requirement that weight < 22 kgs. Use the following uncertain parameters:

D N(7.5,0.5)
L N(9.5,0.5)
ρ_{cu} $N(8.94e3, 100)$
ρ_{fe} $N(7.98e3, 100)$

To set up this problem I began with a u value of 0 for each of the variables.

$$u = [0, 0, 0, 0] \tag{1}$$

I then converted each variable into u space using the following equation.

$$x_i = \mu_{xi} + u_i \sigma_i \tag{2}$$

I then multiplied each of the variable partial derivatives (found in Homework 2) by their respective sigmas to obtain the partial derivatives of the weight function with respect to u.

$$\frac{\partial g}{\partial u} = \frac{\partial g}{\partial x} * \frac{\partial x}{\partial u} \tag{3}$$

The next set of u values was then determined using the following equation.

$$u_{k+1} = \frac{\nabla g(u_k)^T u_k - g(u_k)}{\nabla g(u_k)^T \nabla g(u_k)} \nabla g(u_k) \tag{4}$$

This process was iterated over until the difference between the reliability index of the new and old values were less than 0.001. The code that I wrote completed after 6 iterations and showed a final reliability index of 2.29 and a probability of successfully meeting the constraints of 0.989.

2. Repeat problem 1 but use the Full Tensor Numerical Integration Method. Use 3 nodes per uncertain input. Again the uncertainties are (the variables are independent, i.e. 0 correlation

D N(7.5,0.5)
L N(9.5,0.5)
ρ_{cu} $N(8.94e3, 100)$
ρ_{fe} $N(7.98e3, 100)$

To set up this problem I first converted the Gauss-Hermite nodes and weights to these specific distributions using the numpy.polynomial.hermite.hermgauss function. I then performed a full factorial so that I could obtain every combination of points and weights that were produced and plugged them into the motor weight calculation. Using these values as well as the combined weights for each of the combinations I then found the moments of the new distribution using the following equations.

$$\mu_g = \int_{-\infty}^{\infty} g(x) f_x(x) dx \tag{5}$$

$$\sigma_g = \sqrt{\int_{-\infty}^{\infty} (g(x) - \mu_g)^2 f_x(x) dx} \tag{6}$$

$$\beta_{1g} = \left(\frac{\int_{-\infty}^{\infty} (g(x) - \mu_g)^3 f_x(x) dx}{\sigma_g^3} \right)^2 \tag{7}$$

$$\beta_{2g} = \left(\frac{\int_{-\infty}^{\infty} (g(x) - \mu_g)^4 f_x(x) dx}{\sigma_g^4} \right) \tag{8}$$

Using these equations and the pears_cdf.py file provided, my code produced the following values.

Mean : 16.746
Standard Deviation : 2.115
Skew: 0.064
Kurtosis: 3.076
Probability of Success: 0.978

3. Repeat problem 2, but use the following distributions for uncertainty:

D Uniform(6.5, 8.5)
L Uniform (8.5, 10.5)
ρ_{cu} $Uniform(8840, 9040)$
ρ_{fe} $Uniform(7880, 8080)$

To set up this problem I used the same method as number 2 but used the numpy.polynomial.legendre.leggauss function to find the specific distribution values for the uniform distribution. Using equations 5, 6, 7, and 8 and the pears_cdf.py file provided my code produced the following values.

Mean: 16.766
Standard Deviation: 2.435
Skew: 0.036
Kurtosis: 2.16
Probability of Success: 0.965

ME615 Homework 3

May 7, 2020

Created by: Heather Miller - ME615 HW3 - started 5/6/20

[15]:
```
import numpy as np
from scipy.stats import norm
from numpy.polynomial.hermite import hermgauss
from numpy.polynomial.legendre import leggauss
from pyDOE2 import fullfact
from pears_cdf import *
```

[16]:
```
#####################################################
# Given Constants
# Motor Design Variables
Lwa = 14.12  # armature wire length (m)
Lwf = 309.45  # field wire length (m)
ds = 0.00612  # slot depth (m)

# Coupling Variables b, shared with control problem
n = 122  # rotational speed (rev/s)
v = 40  # design voltage (V)
pmin = 3.94  # minimum required power (kW)
ymin = 5.12e-3  # minimum required torque (kNm)

# Parameter Vector a (constants)
fi = 0.7  # pole arc to pole pitch ratio
p = 2  # number of poles
s = 27  # number of slots (teeth on rotor)
rho = 1.8e-8  # resistivity (ohm-m) of copper at 25C

# Derived parameters and constants
mu0 = 4 * np.pi * 1e-7  # magnetic constant
ap = p  # parallel circuit paths (equals poles)
eff = 0.85  # efficiency
bfc = 40e-3  # pole depth (m)
fcf = 0.55  # field coil spacing factor
Awa = 2.0074e-006  # cross sectional area of armature winding (m^2)
Awf = 0.2749e-6  # cross sectional area of field coil winding (m^2)
```

1 Part 1 : Using FORM

[17]:
```
def calculate_weight(diameter, length, rho_cu, rho_fe, limit=22):
    # calculate the weight of the motor
    # set limit=0 you are not calculating in u space
    weight = (rho_cu * (Awa*Lwa + Awf*Lwf) + rho_fe * length * np.pi *␣
 ↪pow(diameter + ds, 2)) - limit
    return weight
```

[20]:
```
def finite_diff(variables):
    # returns the first derivative of diameter, length, rho_cu, and rho_fe

    # These are the mu and sigmas of each variable
    D = (variables["diameter"][0], variables["diameter"][1])
    L = (variables["length"][0], variables["length"][1])
    rho_cu = (variables["rho_cu"][0], variables["rho_cu"][1])
    rho_fe = (variables["rho_fe"][0], variables["rho_fe"][1])

    # first derivative of each variable
    d_dD = (calculate_weight(D[0] + 0.1 * D[1],
                             L[0],
                             rho_cu[0],
                             rho_fe[0]) -
            calculate_weight(D[0] - 0.1 * D[1],
                             L[0],
                             rho_cu[0],
                             rho_fe[0]))/ (0.2 * D[1])

    d_dL = (calculate_weight(D[0],
                             L[0] + 0.1 * L[1],
                             rho_cu[0],
                             rho_fe[0]) -
            calculate_weight(D[0],
                             L[0] - 0.1 * L[1],
                             rho_cu[0],
                             rho_fe[0]))/ (0.2 * L[1])

    d_dcu = (calculate_weight(D[0],
                              L[0],
                              rho_cu[0] + 0.1 * rho_cu[1],
                              rho_fe[0]) -
             calculate_weight(D[0],
                              L[0],
                              rho_cu[0] - 0.1 * rho_cu[1],
                              rho_fe[0]))/ (0.2 * rho_cu[1])

    d_dfe = (calculate_weight(D[0],
```

```
                                L[0],
                                rho_cu[0],
                                rho_fe[0] + 0.1 * rho_fe[1]) -
            calculate_weight(D[0],
                                L[0],
                                rho_cu[0],
                                rho_fe[0] - 0.1 * rho_fe[1]))/ (0.2 * rho_fe[1])

    return [d_dD, d_dL, d_dcu, d_dfe]
```

[21]:
```
def u_update(u, variables):
    # returns the updated u

    variable_mus = [variables["diameter"][0],
                    variables["length"][0],
                    variables["rho_cu"][0],
                    variables["rho_fe"][0]]
    variable_sigmas = [variables["diameter"][1],
                       variables["length"][1],
                       variables["rho_cu"][1],
                       variables["rho_fe"][1]]

    # this is iterating through each of the variables
    # mu and sigma and converting it into u space
    x = [(variable_mus[i] + u[i]*variable_sigmas[i]) for i in range(len(u))]

    # this is iterating through the first derivatives
    # of each of the variables and multiplying them by
    # their respectivesigmas
    grad_g = [(finite_diff(variables)[i]* variable_sigmas[i]) for i in␣
↪range(len(u))]

    # this is plugging in the u space version of each variable into the equation
    g_u = calculate_weight(x[0], x[1], x[2], x[3])

    # updating u
    u_k = [((np.dot(grad_g, u) - g_u)/(np.dot(grad_g, grad_g))) * i for i in␣
↪grad_g]

    return u_k
```

[22]:
```
def find_reliability(u, diff_threshold, iterations):
    # search for the reliability of function
    # returns the number of iterations until convergence,
    # the final beta, and the probability of success

    diff = 1
```

```
    k = 1
    while diff > diff_threshold and k < iterations:
        uk = u_update(u, variables)
        diff = abs(np.linalg.norm(uk) - np.linalg.norm(u))
        u = uk
        beta = np.linalg.norm(u)
        prob = norm.cdf(np.linalg.norm(beta))

        k += 1
    return k-1, beta, prob
```

2 Part 2 : Using Full Tensor Numerical Integration Method

- 3 nodes per uncertain input
- same uncertainties as problem 1

[23]:
```
def convert_norm(number_of_nodes, variables):
    # convert Gauss-Hermite nodes and weights to integrate specific distributions

    variable_mus = [variables["diameter"][0],
                    variables["length"][0],
                    variables["rho_cu"][0],
                    variables["rho_fe"][0]]
    variable_sigmas = [variables["diameter"][1],
                       variables["length"][1],
                       variables["rho_cu"][1],
                       variables["rho_fe"][1]]

    points, weights = hermgauss(number_of_nodes)
    variable_points =[]
    variable_weights = [i / np.sqrt(np.pi) for i in weights]

    for j in range(len(variables)):
        variable_points.append([(np.sqrt(2) * i * variable_sigmas[j])
                                + variable_mus[j] for i in points])

    return variable_points, variable_weights
```

[24]:
```
def full_factorial(points, weights):
    # perform a full factorial on the converted points and weights

    D = points[0]
    L = points[1]
    cu = points[2]
    fe = points[3]
```

```
    g_points = []
    big_W = []

    indx = fullfact([3, 3, 3, 3])

    for i in indx:
        idx_D = int(i[0])
        idx_L = int(i[1])
        idx_cu = int(i[2])
        idx_fe =int(i[3])
        g_points.append(calculate_weight(D[idx_D],
                                         L[idx_L],
                                         cu[idx_cu],
                                         fe[idx_fe],
                                         limit=0))
        big_W.append(weights[idx_D]*
                     weights[idx_L]*
                     weights[idx_cu]*
                     weights[idx_fe])

    return g_points, big_W
```

[25]:
```
def cal_sys_moments(g_points, big_W):
    # calculate the moments of the function

    g_mean = np.dot(g_points, big_W)
    g_sigma = np.sqrt(np.dot((g_points-g_mean)**2, big_W))
    g_skew = (np.dot((((g_points-g_mean)**3)/g_sigma**3), big_W))**2
    g_kurtosis = np.dot((((g_points-g_mean)**4)/g_sigma**4), big_W)

    return g_mean, g_sigma, g_skew, g_kurtosis
```

3 Part 3 : Same as 2 but with a Uniform Distribution

[28]:
```
def convert_uniform(number_of_nodes, variables):
    # convert Gauss-Laguerre nodes and weights to integrate specific␣
 ↪distributions

    variable_lower = [variables["diameter"][0],
                      variables["length"][0],
                      variables["rho_cu"][0],
                      variables["rho_fe"][0]]
    variable_upper = [variables["diameter"][1],
                      variables["length"][1],
                      variables["rho_cu"][1],
                      variables["rho_fe"][1]]
```

```
    points, weights = leggauss(number_of_nodes)
    variable_points =[]
    variable_weights = [i / 2 for i in weights]

    for j in range(len(variables)):
        variable_points.append([(variable_upper[j] - variable_lower[j]) * i / 2 +
                                (variable_upper[j] + variable_lower[j])
                                / 2 for i in points])

    return variable_points, variable_weights
```

Answers

[27]:
```
# Part 1: Normal Distribution (FORM)
variables = {"diameter": (0.075, 0.005),
                 "length": (0.095, 0.005),
                 "rho_cu": (8.94e3, 100),
                 "rho_fe": (7.98e3, 100)}
u = [0, 0, 0, 0]
k, beta, prob = find_reliability(u, .001, 20)

print("Problem 1:\n# of Iterations:", k, "\nBeta:", beta,
      "\nProbability of Success:", prob)

# Part 2: Normal Distribution (Full Tensor Numerical Integration Method)
points, weights = convert_norm(3, variables)
g_points, big_W = full_factorial(points, weights)
g_mean, g_sigma, g_skew, g_kurtosis = cal_sys_moments(g_points, big_W)
p, type = pearson_fit([-np.inf, 22], g_mean, g_sigma, g_skew, g_kurtosis)

print("\nProblem 2:\nMean:",g_mean,"\nStandard Deviation:", g_sigma,
      "\nSkew:", g_skew, "\nKurtosis:", g_kurtosis,
        "\nProbaility of Success:", p)

# Part 3: Uniform Distribution (Full Tensor Numerical Integration Method)
variables_3 = {"diameter": (0.065, 0.085),
                   "length": (0.085, 0.105),
                    "rho_cu": (8840, 9040),
                    "rho_fe": (7880, 8080)}

points, weights = convert_uniform(3, variables_3)
g_points, big_W = full_factorial(points, weights)
g_mean, g_sigma, g_skew, g_kurtosis = cal_sys_moments(g_points, big_W)
p, type = pearson_fit([-np.inf, 22], g_mean, g_sigma, g_skew, g_kurtosis)

print("\nProblem 3:\nMean:",g_mean,"\nStandard Deviation:", g_sigma,
```

```
      "\nSkew:", g_skew, "\nKurtosis:", g_kurtosis,
      "\nProbaility of Success:", p)
```

```
Problem 1:
# of Iterations: 6
Beta: 2.291138707117728
Probability of Success: 0.9890223037125798

Problem 2:
Mean: 16.74572950275175
Standard Deviation: 2.1154865548942046
Skew: 0.06370777809819958
Kurtosis: 3.075851834633403
Probaility of Success: 0.9778296002644999

Problem 3:
Mean: 16.76557651434081
Standard Deviation: 2.435148669743893
Skew: 0.035671782489359864
Kurtosis: 2.1600721149471838
Probaility of Success: 0.9653058147235051
```