VIP Labbook

Daniel Maidment

CONTENTS

I. WING MODELING

A. NACA Airfoils

1) Symmetrical 4-digit NACA foil: The formula for the shape of a NACA 00xx foil, with "x" being replaced by the percentage of thickness to chord, is:

$$y_t = 5t\left[0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4\right] \tag{1}$$

* x is the position along the chord from 0 to 1.00, (0 to 100* y_t is the half thickness at a given value of x (centerline to surface), and * t is the maximum thickness as a fraction of the chord (so t gives the last two digits in the NACA 4-digit denomination divided by 100).

```
#NACA 4-digit airfoil generator function
from jupyter_code_template import *
def gen_NACA(series = [6, 4, 12], N = 1000):
    """
        series is a list [a,b,x]
        -a = 100*m is the maximum camber,
        -b = 100*p is the location of maximum camber,
        -t = x/100 is the maximum thickness as a fraction of the chord
        -N is the number of x coordinates in total

        returns: N, x, y_c , y_U, y_L
        -N is the length of X
        -x is the x coordinate array
        -y_c is the mean camber line
        -y_U is the upper edge coorndinate array
        -y_L is the lower edge coordinate array

        Notes:
        -If a = 0 OR b = 0, then the foil is assumed to be symmertical
        - in this case y_c is set to None
    """
    x = np.linspace(0, 1, N, endpoint=True)
    m = series[0]/100
    p = series[1]/10
    t = series[2]/100
    c = 1

    y_t = 5*t*(0.2969*sqrt(x)-0.1260*x-0.3516*x**2+0.2844*x**3-0.1015*x
        **4)

    if(m == 0 or p == 0):
        print("symmetric")
        return N, x, None, y_t, -1*y_t
    else:
        print("cambered")
        y_c = np.zeros(N)
        y_c[0:int(p*N)] = (m/p**2)*(2*p*(x[0:int(p*N)]/c)-(x[0:int(p*N)]/
            c)**2)
        y_c[int(p*N):] = (m/(1-p)**2)*((1-2*p)+2*p*(x[int(p*N):]/c)-(x[
            int(p*N):]/c)**2)

        dy_c = np.zeros(N)
        dy_c[0:int(p*N)] = (m/p**2)*(p-(x[0:int(p*N)]/c))
        dy_c[int(p*N):] = (m/(1-p)**2)*(p-(x[int(p*N):]/c))
```

```
        theta = np.arctan(dy_c)

        # x_U = x-y_t*sin(theta)
        # x_L= x+y_t*sin(theta)
        y_U = y_c+y_t*cos(theta)
        y_L = y_c-y_t*cos(theta)

        return N, x, y_c, y_U, y_L

def cambered_foil_plt(fig, ax, N, x, y_c, y_U, y_L, NACA):
    NACA_str = ''.join(map(str, NACA))
    ax = config_axis(ax, x_lim = (0, 1), Eng = False)

    ax.plot(x, y_c, label = "Mean camber line: "+r"$y_c$")
    ax.plot(x, y_U, label = "Upper edge: " + r"$y_U$")
    ax.plot(x, y_L, label = "Lower edge: " + r"$y_L$")
    ax.fill_between(x, y_U, y_L, facecolor = "grey", alpha = 0.4, label =
        "Foil section")

    ax.legend(loc = "upper left", bbox_to_anchor = (1, 1))
    figcaption("Cambered 4-digit NACA " + NACA_str +" airfoil", label="
        fig:series4cam_gen")
    plt.show()

def symmetric_foil_plt(fig, ax, N, x, y_c, y_U, y_L, NACA):
    NACA_str = ''.join(map(str, NACA))

    ax = config_axis(ax, x_lim = (0, 1), Eng = False)
    ax.plot(x, y_U, label = "Upper edge: " + r"$y_U$")
    ax.plot(x, y_L, label = "Lower edge: " + r"$y_L$")
    ax.fill_between(x, y_U, y_L, facecolor = "grey", alpha = 0.4, label =
        "Foil section")
    ax.legend(loc = "upper left", bbox_to_anchor = (1, 1))
    figcaption("Symmetric 4-digit NACA " + NACA_str +" airfoil", label="
        fig:series4sym")
    plt.show()

def xfoil_foilgen(series = [6, 4, 12], N = 100):
    """
        Takes in a 4-digit NACA code, and number of data points,
        it then generates the corresponding x and y cooridinates
        for use in XFOIL
    """
```

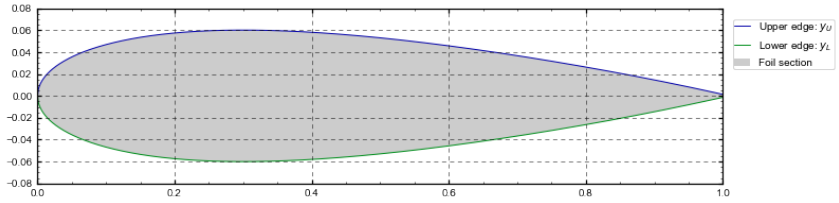


Fig. 1: Symmetric 4-digit NACA 0012 airfoil

```
    N, x_i, y_c, y_U_i, y_L_i = gen_NACA(series, N = N)

    x_U = x_i[::-1]
    x_L = x_i
    x = np.concatenate((x_U, x_L))

    y_U = y_U_i[::-1]
    y_L = y_L_i
    y = np.concatenate((y_U, y_L))
    N = len(x)
    return N, x, y
```

```
from jupyter_code_template import *
NACA = [0, 0, 12]
NACA_str = ''.join(map(str, NACA))
N = 1000
N, x, y_c, y_U, y_L = gen_NACA(series=NACA, N = N)

fig1, ax1 = plt.subplots(1, 1, figsize = (12, 3))
symmetric_foil_plt(fig1, ax1, N, x, y_c, y_U, y_L, NACA)
```

`symmetric`

2) Cambered 4-digit NACA foil: The simplest asymmetric foils are the NACA 4-digit series foils, which use the same formula as that used to generate the 00xx symmetric foils, but with the line of mean camber bent. The formula used to calculate the mean camber line is:

$$y_c = \begin{cases} \frac{m}{p^2}\left(2p\left(\frac{x}{c}\right) - \left(\frac{x}{c}\right)^2\right), & 0 \leq x \leq pc \\ \frac{m}{(1-p)^2}\left((1-2p) + 2p\left(\frac{x}{c}\right) - \left(\frac{x}{c}\right)^2\right), & pc \leq x \leq c \end{cases} \tag{2}$$

* m is the maximum camber ($100m$ is the first of the four digits), * p is the location of maximum camber ($10p$ is the second digit in the NACA xxxx description).

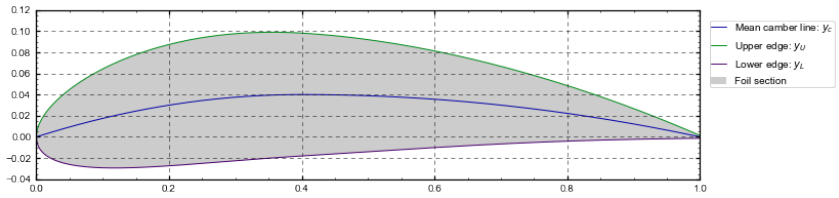


Fig. 2: Cambered 4-digit NACA 4412 airfoil

For this cambered airfoil, because the thickness needs to be applied perpendicular to the camber line, the coordinates (x_U, y_U) and (x_L, y_L), of respectively the upper and lower airfoil surface, become:

$$x_U = x - y_t \sin(\theta), \qquad y_U = y_c + y_t \cos(\theta) \tag{3}$$

$$x_L = x + y_t \sin(\theta), \qquad y_L = y_c - y_t \cos(\theta) \tag{4}$$

$$\tag{5}$$

, where

$$\theta = \arctan\left(\frac{dy_c}{dx}\right) \tag{6}$$

$$\frac{dy_c}{dx} = \begin{cases} \frac{2m}{p^2}\left(p - \frac{x}{c}\right), & 0 \leq x \leq pc \\ \frac{m}{(1-p)^2}\left(p - \frac{x}{c}\right), & pc \leq x \leq c \end{cases} \tag{7}$$

```
from jupyter_code_template import *
NACA = [4, 4, 12]
N = 1000

N, x, y_c, y_U, y_L = gen_NACA(series=NACA, N = N)

fig2, ax2 = plt.subplots(1, 1, figsize = (12, 3))

cambered_foil_plt(fig2, ax2, N, x, y_c, y_U, y_L, NACA)
```

```
from jupyter_code_template import *
NACA = [4.5, 4, 12]
N = 1000

N, x, y_c, y_U, y_L = gen_NACA(series=NACA, N = N)

fig2, ax2 = plt.subplots(1, 1, figsize = (12, 3))

cambered_foil_plt(fig2, ax2, N, x, y_c, y_U, y_L, NACA)
```

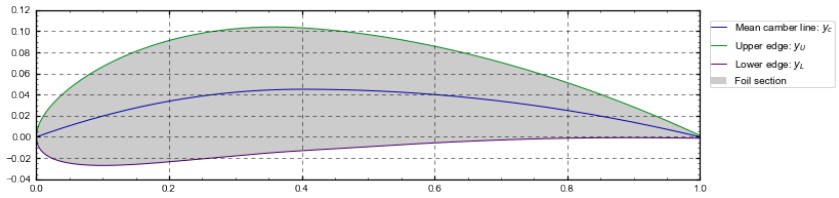
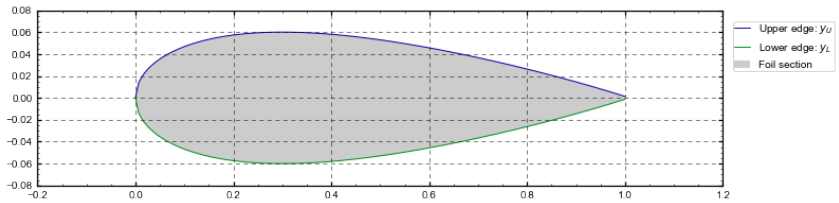


Fig. 3: Cambered 4-digit NACA 4.5412 airfoil



`cambered`

```
N, x, y = xfoil_foilgen(series = [0, 0, 12], N = 160)
fig, ax = plt.subplots(1, 1, figsize = (12, 3))
ax = config_axis(ax, Eng = False)

x1 = x[int(N/2):]
y_U = y[:int(N/2)]
y_U = y_U[::-1]
y_L = y[int(N/2):]

ax.plot(x1, y_U, label = "Upper edge: " + r"$y_U$")
ax.plot(x1, y_L, label = "Lower edge: " + r"$y_L$")
ax.fill_between(x1, y_U, y_L, facecolor = "grey", alpha = 0.4, label = "
    Foil section")
ax.legend(loc = "upper left", bbox_to_anchor = (1, 1))
plt.show()
```

`symmetric`

3) 5-digit NACA:

B. AeroPy

```
import aeropy.xfoil_module as xf
# xf.find_coefficients('naca0015', 5)
```

```
NACA = [4, 4, 12]
NACA_str = 'naca'+''.join(map(str, NACA))
# N, x, y_c, y_U, y_L = gen_NACA(NACA)
# filename = xf.file_name(NACA_str,  alfas=5, output = 'Polar')
# print(filename)
# xf.create_input(x, y_U, y_L, filename, True)
print(xf.find_coefficients('24112',5,10e4))
# xf.call(NACA_str, 5, 'Coordinates',  10e4, 0, False)
```

C. *UAV design ideas*

- Foam structure
- Skin
- Mould for rapid deployment
  - JIG for foam cutting instead?
- Pixhawk brain
  - Definitely –> I shoud probably source one
- AI autopilot
  - Long term goal, integrate with pixahawk
- Predefined wiring harness
  - Obviously
- Parachute for landing
  - Yes
- Sled takeoff
  - Yes
  - Compressed air?
    - Not right now.
  - Elastic bands seem better.
  - VTOL sled? Seems cool, why not?
- Design challenge:
  - Design for scaling
  - Rapid prototyping
  - Rapid deployment
  - Minimal moving parts
- Pitch control
- Torque vectoring for pitch control

- Tail surface?
- Linear motor over centre of gravity
- Up-cambered wings
- Reaper and Predator drones really caught my eye. These were basically just glider wings, set back on the fuselage to counter the weight of the camera equipment and sensors in the nose. The anhedral tail surfaces on the Predator are intended to assist in the prevention of prop-strike during take-off and landing – plus they look cool.
- Air time: 10˜min
- SDR & GPU-Jetson
  - 15 W to 20 W (achievable)
- GPS
- Mini-airship
- Does it need to move a distance
  - Tethering is difficult
  - Yes we want it to move
- Low transmission power in RF, 100 mW cap (to be revised)
- Try to get better power consumption estimates
  - best GPU?
  - radar
  - SDR

II. Jamming System UAV Requirements

2019/02/28

- SDR Type: Nuand BladeRF 2.0
- SDR Dimensions: 12 cm (17 cm with amplifiers) x 8 cm x 2.5 cm
- SDR Power:(5 V ; 1 A) $\implies$ 5 .0 W
- SDR External Amplifier Power:
  - 2 x PA: (2 x 0.3 W) $\implies$ 0.6 W
  - 2 x LNA: (2 x 0.45 W) $\implies$ 0.9 W
- Jammer Wingspan Requirements: 2.5 m
- SDR Mass (with amplifiers and cover): 350 g
- Odroid Processor Power $\implies$ 20 W
- Odroid Dimensions: 7 cm x 9 cm x 3 cm

- Odroid Weight: 100 g
- RG-58 Coaxial Cable: 37 g / m

A. *Rietfontein Anti-poaching*

- Rietfontein Anti-poaching
  - Need a drone.
  - We can test at the Rietfontein
  - We need to get specs
  - Circumference of 30 km
  - Diameter of 8 km

B. *Design requirements*

- UAV wing tips
- Reflexed Wing tips
- Fibre glass manufacturing don't so small jobs
- They also don't

C. *Build Techniques*

- Balsa
  - Hard to build
  - Fragile
  - Hard to repair
- Folded foam board
  - Quick cheap and easy
  - Not much in the way of accuracy
- Coragated platic board
  - indestrutable
  - inaccurate
- Foam board
  - Hot wire cutter
  - really fragile
    - packing tape
    - laminate sheet
    - fibre glass onto foam (promising)

D. Tutorials

- https://www.instructables.com/id/Black-Eagle-aerial-mapping-UAV/
- https://www.youtube.com/watch?v=Xafg-o83L94
- https://www.robotshop.com/community/tutorials/show/how-to-make-a-drone-uav-lesson-8-airplanes

E. Genetic Algorithm implementation

1) Basic GA tutorial:

- https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

```
def cal_pop_fitness(equation_inputs, pop):
    # Calculating the fitness value of each solution in the current
        population.
    # The fitness function calculates the sum of products between each
        input
    # and its corresponding weight.
    fitness = np.sum(pop*equation_inputs, axis=1)
    return fitness

def select_mating_pool(pop, fitness, num_parents):

    # Selecting the best individuals in the current generation as parents
    # for producing the offspring of the next generation.

    #generate empty parent population array
    parents = np.empty((num_parents, pop.shape[1]))

    #iterate through the number of parents
    for parent_num in range(num_parents):
        #find which memper had max fitness
        max_fitness_idx = np.where(fitness == np.max(fitness))
        max_fitness_idx = max_fitness_idx[0][0]

        #choose the individual from the previous population with the
            maximum
        #fitness to be a parent for the next generation.
        parents[parent_num, :] = pop[max_fitness_idx, :]

        #set the current maximum fitness to be some small value so that
            on the
```

```
        #next iteration of the loop the next best individual will be
            selected
        #i.e. take this one out of the running
        fitness[max_fitness_idx] = -99999999999
    return parents

def crossover(parents, offspring_size):

    offspring = np.empty(offspring_size)
    # The point at which crossover takes place between two parents.
    #Usually, it is at the center.
    crossover_point = int(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        # Index of the first parent to mate.
        parent1_idx = k%parents.shape[0]
        # Index of the second parent to mate.
        parent2_idx = (k+1)%parents.shape[0]
        # The new offspring will have its first half of its genes taken
            from the first parent.
        offspring[k, 0:crossover_point] = parents[parent1_idx, 0:
            crossover_point]
        # The new offspring will have its second half of its genes taken
            from the second parent.
        offspring[k, crossover_point:] = parents[parent2_idx,
            crossover_point:]
    return offspring

def mutation(offspring_crossover):

    # Mutation changes a single gene in each offspring randomly.

    for idx in range(offspring_crossover.shape[0]):

        # The random value to be added to the gene.

        random_value = np.random.uniform(-1.0, 1.0, 1)

        offspring_crossover[idx, 4] = offspring_crossover[idx, 4] +
            random_value

    return offspring_crossover
```

```
from jupyter_code_template import *
```

```
#Inputs to the function to be minimised
equation_inputs = [4,-2,3.5,5,-11,-4.7]

#parameters to be optimised
N_w = len(equation_inputs)

#the number of individuals in a population
N_ind = 8

#the population shape
pop_shape = (N_ind, N_w)

#seed the population
new_pop = np.random.uniform(-4.0, 4.0, size = pop_shape)

#the number of generations
N_gen = 10
new_parents_mating = 4
fit_arr = []
for gen in range(N_gen):
    fitness = cal_pop_fitness(equation_inputs, new_pop)
    parents = select_mating_pool(new_pop, fitness, new_parents_mating)
    offspring_crossover = crossover(parents, offspring_size = (pop_shape
        [0]-parents.shape[0], N_w))
    offspring_mutation = mutation(offspring_crossover)
    new_pop[0:parents.shape[0], :] = parents
    new_pop[parents.shape[0]:, :] = offspring_mutation
    fit = max(sum(new_pop*equation_inputs, axis=1))
    fit_arr = np.append(fit_arr, fit)

# Getting the best solution after iterating finishing all generations.
#At first, the fitness is calculated for each solution in the final
    generation.
fitness = cal_pop_fitness(equation_inputs, new_pop)
# Then return the index of that solution corresponding to the best
    fitness.
best_match_idx = np.where(fitness == max(fitness))

print("Best solution : ", new_pop[best_match_idx, :])
print("Best solution fitness : ", fitness[best_match_idx])

fig, ax = plt.subplots(1, 1, figsize = (12, 3))
ax = config_axis(ax, Eng = False)
ax.plot(fit_arr)
```

```
plt.show()
```

```
Best solution :  [[[ 3.17667176 -0.35600885  1.92396877  2.49634592

-6.41061891 -2.64615371]]]
Best solution fitness :  [115.58805545]
```

```
from jupyter_code_template import *
from numpy.random import randint
np.random.seed(20190321)

N_ind = 10
N_params = 3
pop = np.empty((N_ind, 3), int)
#seed population
for i in range(N_ind):
    pop[i, :] = [randint(0, 7, None, int), randint(0, 7, None, int),
        randint(10, 20, None, int)]
```

```
array([[ 6,  3, 19],
       [ 2,  2, 18],
       [ 4,  2, 16],
       [ 1,  0, 18],
       [ 2,  3, 15],
       [ 5,  1, 17],
       [ 0,  6, 15],
       [ 0,  1, 17],
       [ 3,  4, 15],
       [ 4,  6, 17]])
```