Project 2

December 22nd, 2020

1 Setting up the problem

1.1 Physics assumptions

We model an incompressible, subsonic, steady-state flow with no heat addition/removal and negligible change in height. Additional assumptions that were used in calculations are stated throughout assignment as they were made.

I used numerical lifting line analysis to calculate the solution to this problem. Unlike Prandt's lifting line theory that I used in homework 5, numerical lifting line analysis does not assume a linear gradient of the lift curve across the span. Therefore, I am able to use the NACA0012 lift coefficient lookup table, which accounts for stall effects.

For viscous effects, the coefficient of lift curve of the NACA0012 was obtained experimentally and therefore includes the effect of viscousity on the lift generated by the airfoil & hence wing as it includes the stall effect. The airfoil used in the wing was NACA0012.

All units were converted to metric for standardization and to reduce potential calculation errors.

Facts about the wing:

Wing span:	$b = 10$ ft	$= 3.0480$ m
Wing root chord:	$c = 1.5$ ft	$= 0.4572$ m
Cruise speed:	$V_\infty = 100$ knots	$= 51.4444$ m/s
Angle of Attack:	$\alpha = 8°$	$= 0.139626$ (0.044444π) radians
Airfoil lift curve slope (NACA0012):	given by lookup table	
Propeller diameter:	$D = 4$ ft	$= 1.2192$ m

1.2 Methodology high-level overview

1. We can re-use some of the Wing class from Homework 5 for this assignment to store information about the wing. In Homework 5, the Wing class also calculated its own lift distribution using Prandt's lifting line theory. Since I'm using numerical lifting line methods for this assignment, I will remove that portion of code and instead create a new NumericalLiftLineSolver class to handle that. Additionally, the Wing class now needs to store a lookup table for the coefficient of lift vs. α curve (previously I simply provided a linear gradient).
2. Create a new NumericalLiftLineSolver class which uses numerical lifting line method to solve for the lift distribution and hence wing performance of the simple wing (without propellers).

3. Expand the NumericalLiftLineSolver class to account for the effects of the propeller. We will also need to somehow pass information to the LiftLineSolver class defining where the propellers are (TBD: Aircraft class? Propeller class?)

1.3 Numerical non-linear lifting line method

This methodology is based on the numerical nonlinear lifting line method described in lecture_video_20201116.mp4 and chapter 5.4 of the Anderson textbook, page 465.

0. Divide the wing into N elements, and hence $K = N + 1$ splits.

1. For all the elements, calculate the effective angle of attack:

$$\alpha_{\text{eff}}(n) = \alpha_{\text{geom}}(n) - \alpha_i(n)$$

   For the first iteration, assume $\alpha_i = 0$

2. For all elements, denoted by index $n = 1, 2, \ldots, N$, calculated the bound vortex $\Gamma_b(n)$:

$$\begin{aligned} c_l(n) &= c_{l\alpha}(n) \times \alpha_{\text{eff}}(n) \\ L'(n) &= c_l(n) \times \frac{1}{2}\rho_\infty V_\infty C(n) \\ &= \rho_\infty V_\infty \Gamma_b(n) \\ \implies \Gamma_b(n) &= \frac{c_l(n) V_\infty C(n)}{2} \end{aligned}$$

3. Calculate the trailed vortices, denoted by index $k = 1, 2, \ldots, K$:

$$\Gamma_t(k) = \Gamma_b(k) - \Gamma_b(k-1)$$

For the first and last trailing vortices at the wing tips:

$$\begin{aligned} \Gamma_t(1) &= \Gamma_b(1) \\ \Gamma_t(K) &= -\Gamma_b(K-1) \end{aligned}$$

4. Calculate the induced velocity V_{in} at each element n:

$$V_{\text{in}}(n) = \sum_{k=1}^{K} \frac{-\Gamma_t(k)}{4\pi(y_b(n) - y_t(k))}$$

5. Calculate the induced angle of attack $\alpha_i(n)$ for each element n:

$$\begin{aligned} \alpha_i(n) &= \tan^{-1}\left(\frac{-V_{\text{in}}(n)}{V_\infty}\right) \\ &\approx \frac{-V_{\text{in}}(n)}{V_\infty} \qquad \text{(for small angles)} \end{aligned}$$

6. Repeat steps 1 and 2 using the calculated induced angle of attack.
7. Compare the new distribution $\Gamma_{b_{\text{new}}}$ with the previous values $\Gamma_{b_{\text{old}}}$ fed into step 3. If $\Gamma_{b_{\text{new}}}$ does not agree with $\Gamma_{b_{\text{old}}}$ within an acceptable level of accuracy e_n, then steps X-X are run again.
8. If run again, the new input to step 3 is:

$$\Gamma_{b_{\text{input}}} = \Gamma_{b_{\text{old}}} + D(\Gamma_{b_{\text{new}}} - \Gamma_{b_{\text{old}}})$$

Where D is a damping factor for the iterations; typical D values are on the order of $0.01 \leq D \leq 0.05$.

9. If the solution converged in step 8, then we now have a converged $\Gamma(y)$. During the process of calculating the solution, we calculated the local coefficient of lift. Additionally, from the converged $\Gamma(y)$, the lift and induced drag can be obtained from the following equations:

$$L = \rho_\infty V_\infty \int_{-\frac{b}{2}}^{\frac{b}{2}} \Gamma(y)\, dy$$

$$D_i = \int_{-\frac{b}{2}}^{\frac{b}{2}} L'(y) \sin\alpha_i\, dy$$

Note that before the above forces are integrated, the Lift and Drag force for every spanwise section needs to be rectified from the effective angle of attack in that spanwise section to the overall wing angle of attack, otherwise the lift force is in a slightly different direction for each spanwise section as the effective angle of attack changes.

1.4 Modelling the effects of the propellers' wake

The effect of the propeller wake needs to be accounted for in several of the above steps.

1. We assume the propellers are aligned with the aircraft longtitudinal axis, therefore if the aircraft is at an 8° angle of attack, the propellers will also be at an 8° angle of attack relative to the freestream velocity. We are assuming the wake is a constant flow behind the propeller, i.e., the swirl of the rotor wake is ignored.
2. The propeller wake velocity vector v_i is calculated using the equations given in the assignment.
3. The propeller wake velocity vector v_i is divided into a vertical component and a horizontal component.
4. The freestream velocity V_∞ is no longer constant across the span of the wing, but rather for the sections of the airfoil in the propeller's wake the freestream velocity is V_∞ plus the horizontal component of v_i, accounting for the angle of attack of the aircraft.
5. The propeller wake also has a vertical component. This can either be accounted for when calculating the effective angle of attack α_{eff} for each wing section, or it can be added to the calculated downwash induced velocity (which is then used to calculate α_i and hence α_{eff}. I chose to add the propeller wake velocity vertical component vector to the downwash calculated for the wing.

1.5 Implementing the above method using Python

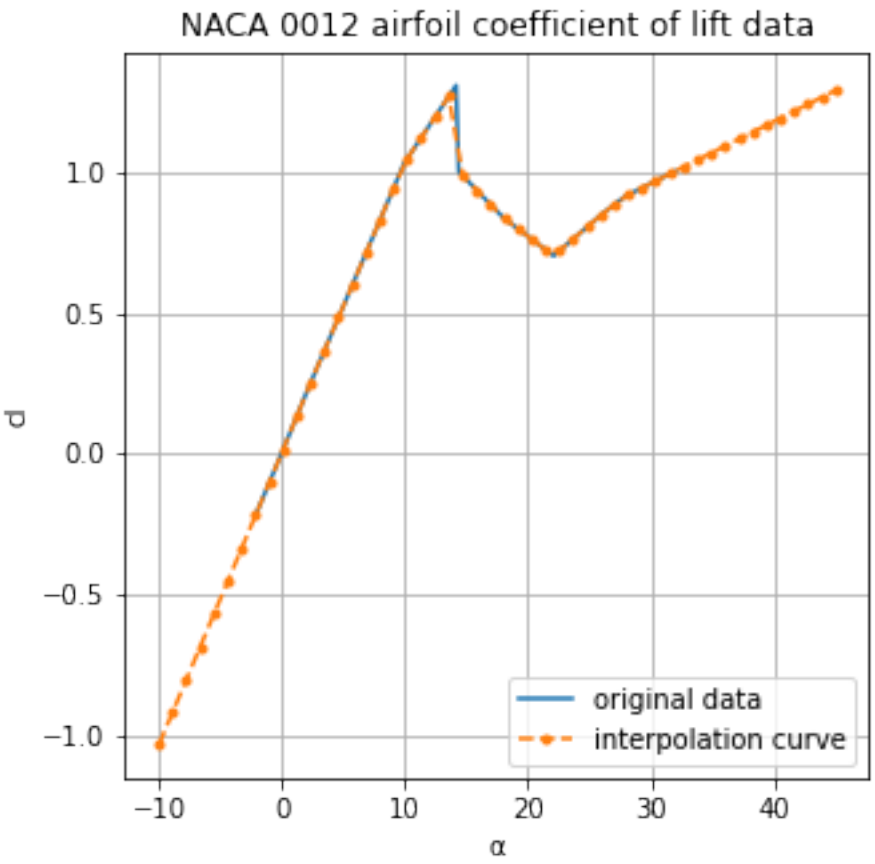
1.5.1 Various Python imports

[1]:
```
import numpy as np
import itertools
import math
from scipy import interpolate as interp
import matplotlib
from matplotlib import pyplot
from IPython.display import clear_output
import multiprocess as mp
from multiprocess import pool
import functools
%matplotlib inline
matplotlib.rcParams['axes.formatter.useoffset'] = False

π = math.pi
# Some Greek letters: α Γ δ Δ ρ ∞
```

1.5.2 Define our NACA0012 coefficient of lift interpolation table

The following code setups functions to interpolate the coefficient of lift using the lookup table given in the assignment. It uses scipy's interp1d function.

[2]:
```
α_deg_arr = [-2.025902479, 0.038772741, 2.05982177, 6.017645346, 9.932239783, 14.16254093, 14.36464087, 17.98442351, 22.
 ↪07932961, 26.01754869, 27.87918097, 31.99647093]
cl_arr = [-0.20812679, 0.00652805, 0.22331494, 0.62902511, 1.03258364, 1.30984164, 1.00359647, 0.83948681, 0.70332889, 0.
 ↪85200745, 0.91454208, 1.00542989]

α_arr = np.deg2rad(α_deg_arr)

NACA0012_cl = interp.interp1d(α_arr, cl_arr, fill_value="extrapolate")


def plot_cl_interpolation():

    # α_test = np.linspace(α_deg_arr[0], α_deg_arr[-1], 50)

    α_test = np.linspace(-10, 45, 50)

    size_X, size_Y = 5,5

    pyplot.figure(figsize=(size_X, size_Y))
    # pyplot.axis("equal")
    pyplot.grid(zorder=0)

    pyplot.title("NACA 0012 airfoil coefficient of lift data")
    pyplot.ylabel('cl')
    pyplot.xlabel('α')

    pyplot.plot(α_deg_arr, cl_arr, "-", label="original data")
    pyplot.plot(α_test, NACA0012_cl(np.deg2rad(α_test)), ".--", label="interpolation curve")

    pyplot.legend()

plot_cl_interpolation()
```



1.5.3 Create a Wing and Propeller class to define the parameters for the wing & propellers

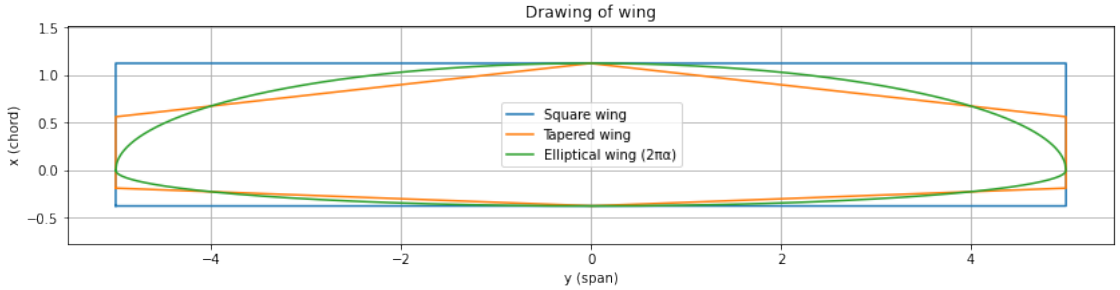
The Wing class below is a heavily modified version of the Wing class from Homework 5. It stores data about the wing and the airfoil used, and calcualtes values parameters as needed, such as the chord at a given location y along the span of the wing.

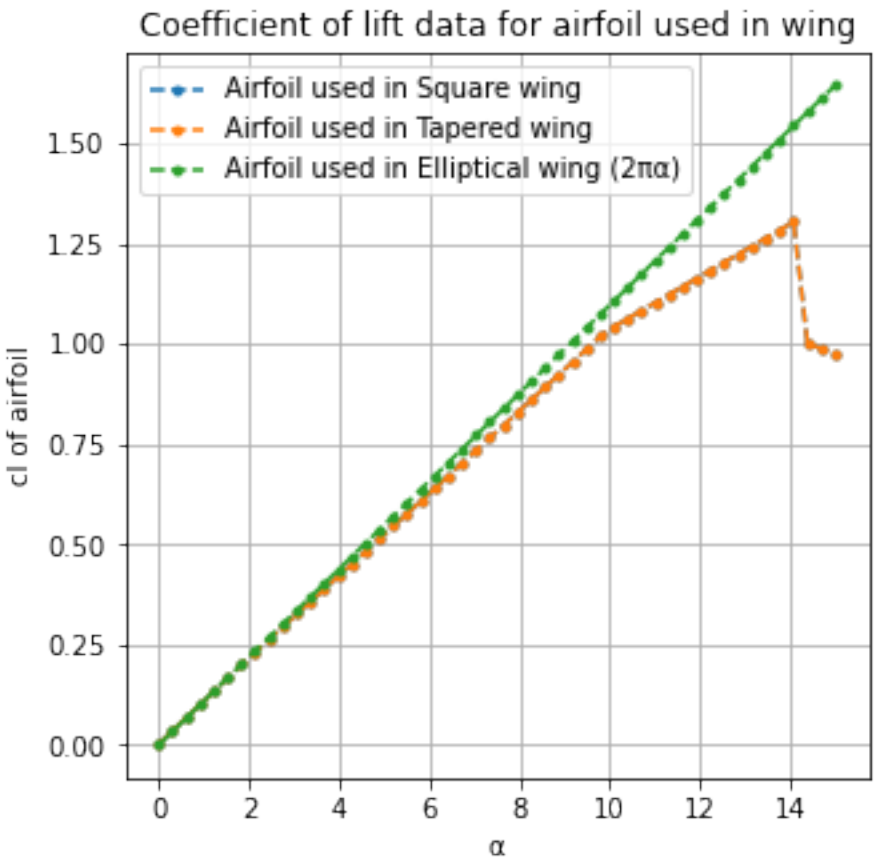
The propeller class below is new. It stores information about the two propellers on the UAV (the data on the two propellers is combined into a single object for simplicity). It also calculates parameters relating to the wake of the propellers, such as the vertical and horizontal components relative to the freestream velocity vector.

[3]:
```
class Wing:

    def __init__(self, cl_curve, span, c_root, c_tip=None, ellipse=False, label=""):
        if c_tip == None:
            c_tip = c_root
        self.cl_f = cl_curve
        self.c_root = c_root
        self.c_tip = c_tip
        self.b = span
        # calculate area and aspect ratio
        self.ellipse=ellipse

        if ellipse:
            self.area = π * self.b / 2 * self.c_root / 2
        else:
            self.area = self.b * (self.c_root + self.c_tip) / 2

        self.AR = (self.b ** 2) / self.area
```

```
        self.label=label

    '''
    def divide(sections):
        wing_sections = np.empty(sections)
        dividers = np.linspace(-self.b/2, self.b/2, sections+1)
        for n in sections:

        return wing_sections
    '''

    def calc_y(self, θ):
        return self.b / 2 * np.cos(θ)

    def calc_c_from_y(self, y):
        '''
        if y < -self.b / 2:
            raise Exception("Sorry, y is outside wing span to left")
        if y > self.b / 2:
            raise Exception("Sorry, y is outside wing span to right")
        '''
        if self.ellipse:
            return self.c_root * np.sqrt(1 - (2*y/self.b) ** 2)
        return self.c_root + abs(y - 0) * (self.c_tip - self.c_root) / (self.b/2 - 0) # abs(y) since assume symmetric

    def calc_c(self, θ):
        return self.calc_c_from_y(self.calc_y(θ))

    def get_cl(self, α):
        return self.cl_f(α)

    def draw(self, size_X=14, size_Y=3., N=1000, init_figure=False, detail="all"):
        if init_figure:
            pyplot.figure(figsize=(size_X, size_Y))
            pyplot.axis("equal")
            pyplot.grid(zorder=0)
            pyplot.title("Drawing of wing")
            pyplot.ylabel('x (chord)')
            pyplot.xlabel('y (span)')

        y_arr = np.linspace(-self.b/2, self.b/2, N)
        chord_arr = self.calc_c_from_y(y_arr)

        y_arr = np.append(y_arr, y_arr[::-1])
        y_arr = np.append(y_arr, y_arr[0])

        x_arr = chord_arr * -0.25
        x_arr = np.append(x_arr, chord_arr*0.75)
        x_arr = np.append(x_arr, chord_arr[0]*-0.25)

        pyplot.plot(y_arr, x_arr, label=self.label)

        pyplot.legend()

    def plot_cl(self, size_X=5, size_Y=5, N=50, init_figure=False, detail="all"):
        if init_figure:
            pyplot.figure(figsize=(size_X, size_Y))
            # pyplot.axis("equal")
            pyplot.grid(zorder=0)
            pyplot.title("Coefficient of lift data for airfoil used in wing")
            pyplot.ylabel('cl of airfoil')
            pyplot.xlabel('α')

        α_test = np.linspace(0, 15, N)

        # pyplot.plot(α_deg_arr, cl_arr, "-", label="original data")
        pyplot.plot(α_test, self.cl_f(np.deg2rad(α_test)), ".--", label="Airfoil used in " + self.label)

        pyplot.legend()

class Propellers():

    '''
        D: diameter
        y: Position of center of propeller on wing y axis (span)
        # spin_direction: Clockwise or counter-clockwise spin (Clockwise = -1, Counter-clockwise = 1)
            But spin_direction is not actually used
    '''
    def __init__(self, D, y1=0, y2=0):
        self.D = D
        self.R = self.D / 2
        self.y1 = y1
        self.y2 = y2
        # self.spin_dir = spin_dir
```

```
    def calc_vi(self, T, V_inf, ρ):
        vh = self.calc_vh(T, V_inf, ρ)
        vi_vh = - (V_inf / (2 * vh)) + np.sqrt( ((V_inf / (2 * vh)) ** 2) + 1 )
        vi = vi_vh * vh
        return vi

    def calc_vh(self, T, V_inf, ρ):
        return np.sqrt(T / (2 * ρ * π * (self.R ** 2)))

    def calc_h_comp(self, α, vi, deg=False):
        return np.cos(α) * vi

    def calc_v_comp(self, α, vi, deg=False):
        return -np.sin(α) * vi

def draw_wing():
    test_wing = [None] * 3
    test_wing[0] = Wing(NACA0012_cl, 10, 1.5, 1.5, ellipse=False, label="Square wing")
    test_wing[1] = Wing(NACA0012_cl, 10, 1.5, 0.75, ellipse=False, label="Tapered wing")
    test_wing[2] = Wing((lambda α: 2*π*α), 10, 1.5, 0.75, ellipse=True, label="Elliptical wing (2πα)")
    test_wing[0].draw(init_figure=True)
    for wing in test_wing[1:]:
        wing.draw()
    test_wing[0].plot_cl(init_figure=True)
    for wing in test_wing[1:]:
        wing.plot_cl()

draw_wing()
```





The above charts are just used for testing. You can see the square UAV wing that is used in the UAV, plus of the coefficient of lift, comparing $2\pi\alpha$ to the coefficient of lift data given in the assignment for NACA0012 as a sanity check.

1.5.4 Define wings to test

These are just some definitions of wings I used to test my Numerical Lifting-Line Solver to make sure it works properly. Only the square wing is actually used in the final analysis requested by the lead UAV engineer.

[4]:
```
wings = np.empty(5, dtype=Wing)
wings[0] = Wing(NACA0012_cl, 3.0480, 0.4572, label="Square wing")
wings[1] = Wing(NACA0012_cl, 3.0480, 0.4572, c_tip=0.4572/2, label="Tapered wing to 0.5c")
wings[2] = Wing(NACA0012_cl, 3.0480, 0.4572, ellipse=True, label="Elliptical wing")
wings[3] = Wing(NACA0012_cl, 3.0480 / 1.415, 0.4572, c_tip=0.4572/2, label="1926 wing with AR/a0 = 1.0")
wings[4] = Wing(NACA0012_cl, 100, 0.4572, c_tip=0.4572, label="Insane AR")
```

1.5.5 Create Numerical Lift Line Solver class to solve the lift distribution across the wing

This is an implementation of the Numerical Lifting Line Solver methodology described above.

[5]:
```
# Some Greek letters: π α Γ δ Δ ρ ∞

class WingSolution:

    def __init__(self, wing, α, V_inf, ρ_inf, y_b, y_t, y_b_Δ, chord_b, Γ_b, Γ_t, α_eff, α_i, V_in, V_inf_b, T):
```

```
        self.wing = wing
        self.α = α
        self.V_inf = V_inf
        self.ρ_inf = ρ_inf
        self.y_b = y_b
        self.y_t = y_t
        self.y_b_Δ = y_b_Δ
        self.chord_b = chord_b
        self.Γ_b = Γ_b
        self.Γ_t = Γ_t
        self.α_eff = α_eff
        self.α_i = α_i
        self.V_in = V_in
        self.V_inf_b = V_inf_b
        self.T = T
        # calculated values
        self.L_b = ρ_inf * V_inf * Γ_b
        self.Lift = np.sum(self.L_b*y_b_Δ*np.cos(α - α_eff))
        self.cl_b = 2 * self.L_b / (chord_b * ρ_inf * (V_inf_b ** 2))
        self.CL = 2 * self.Lift / (self.wing.area * ρ_inf * (V_inf ** 2))

        self.Di_b = self.L_b * np.sin(α_i)
        self.Drag_i = np.sum(self.Di_b*y_b_Δ)
        self.cd_b = 2 * self.Di_b / (y_b_Δ * chord_b * ρ_inf * (V_inf_b ** 2))
        self.CDi = 2 * self.Drag_i / (self.wing.area * ρ_inf * (V_inf ** 2))

    def plot_f(self, x, y, title="", xlabel="", ylabel='y (span)', xlim=None, ylim=None, size_X=10, size_Y=10, initialize=True):
        if initialize:
            pyplot.figure(figsize=(size_X, size_Y))
            pyplot.grid(zorder=0)
            pyplot.title(title)
            pyplot.ylabel(xlabel)
            pyplot.xlabel(ylabel)
        # Draw the plot
        pyplot.plot(x, y)
        if xlim is not None:
            pyplot.xlim(xlim)
        if ylim is not None:
            print("set Y")
            pyplot.ylim(ylim)
        pyplot.legend()
        pyplot.show()
        return

    def plot_all(self):
            self.plot_f(self.y_b, self.y_b_Δ, r"Elem size vs wing span", r'y_b_Δ', ylim=(0,1))
            self.plot_f(self.y_b, self.chord_b, r"Chord vs wing span", r'Chord')
            self.plot_f(self.y_b, self.Γ_b, r"Circulation $Γ_b$ vs wing span", r'$L$')
            self.plot_f(self.y_b, self.cl_b, r"Coefficient of lift $c_{l}$ vs wing span", r'$c_{l}$')#, ylim=(-0.5,1.5))
            self.plot_f(self.y_b, self.V_in, r"Induced velocity $V_{in}$ vs wing span", r'V_{in}')
            self.plot_f(self.y_b, np.rad2deg(self.α_i), r"Induced angle of attack $α_{i}$ vs wing span", 'α_i')

class NumericalLiftLineSolver:

    def __init__(self, wing, propellers = None, N=100, max_iterations=10000, e_n=0.00001, D=0.05, polar=True):
        self.wing = wing
        self.propellers = propellers
        self.N = N
        self.max_iterations = max_iterations
        self.e_n = e_n
        self.D = D
        self.polar = polar

    def get_params(self):
        return self.N, self.max_iterations, self.e_n, self.D, self.polar

    def calc_Γ(V_inf, chord, cl):
        return 1/2 * V_inf * chord * cl

    # Assumes propellers don't overlap, because that's not in the assignment
    def calc_prop_bool_array(self, y_b, propellers):
        # check if y_b is behind each propeller
        y_b_prop = np.empty(shape=y_b.size, dtype=bool)
        for i in range(y_b.size):
                # print(np.abs(y_b[i] - propellers.y1))
                y_b_prop[i] = (np.abs(y_b[i] - propellers.y1) <= propellers.R) or (np.abs(y_b[i] - propellers.y2) <= propellers.
↪R)
        return y_b_prop

    # T is thrust of EACH propeller, so TWO propellers = 2x the thrust!
    def solve(self, α, V_inf, T=0, ρ_inf=1.225, α_deg=True, draw=False, debug=True):
        N, max_iterations, e_n, D, polar = self.get_params()
        if α_deg:
            α = np.deg2rad(α)
```

```
        # 1. Divide wing into k elements (k+1 cuts), each station will be represented by its position on the y axis
        N_min = 1
        if N < N_min:
            raise Exception("Elements must be at least %d" % k_min)
        if max_iterations < 0:
            raise Exception("Max iterations must be at least 0.")
        K = N+1
        if polar:
            θ_t = np.linspace(0, π, K)
            y_t = - self.wing.b / 2 * np.cos(θ_t)
        else:
            y_t = np.linspace(-self.wing.b/2, self.wing.b/2, K)

        y_b = (y_t[1:] + y_t[:-1]) / 2
        y_t_chords = self.wing.calc_c_from_y(y_t)

        y_b_Δ = np.diff(y_t)
        np.set_printoptions(formatter={'float': lambda x: "{0:0.10f}".format(x)})
        # print(y_t)
        # print(y_b_Δ)

        # Each element has bound vortex Γ_b, cl(α_eff), chord, α_geom
        # Each cut has trailing vortex Γ_t
        if (not self.propellers == None) and not T==0.0:
            calc_prop = True
            # print("Calculating with thrust")
            # For each element, we need to determine if it is behind the propeller or not, using its midpoint
            # Then store this in a boolean array
            y_b_in_prop_wake = self.calc_prop_bool_array(y_b, propellers)
            # Let's also calculate the propeller vi and thus it's contribution to freestream velocity and induced velocity
            prop_vi = propellers.calc_vi(T, V_inf, ρ_inf)

            #prop_vi_h = self.propellers.calc_h_comp(α, prop_vi, deg=False)
            #prop_vi_v = self.propellers.calc_v_comp(α, prop_vi, deg=False)

            prop_vi_h = self.propellers.calc_h_comp(α, prop_vi, deg=False)
            prop_vi_v = self.propellers.calc_v_comp(α, prop_vi, deg=False)

            # print("prop_vi_h: %0.1f" % prop_vi_h)
            # print("prop_vi_v: %0.1f" % prop_vi_v)

            # print(y_b_in_prop_wake)

            α_prop = np.arctan(- (prop_vi_v / (prop_vi_h + V_inf) * y_b_in_prop_wake))
            # print(α_prop)

        else:
            calc_prop = False
            # print("Ignoring thrust")
            y_b_in_prop_wake = np.array([False] * y_b.size)
            prop_vi = 0
            prop_vi_h = 0
            prop_vi_v = 0

            α_prop = 0

        # 2. Initialize the bound Γ_b vortex distribution to an elliptical one (c_l constant, so α_i is 0)
        chord_b = self.wing.calc_c_from_y(y_b)
        α_eff = np.array([α] * N) - α_prop
        cl_b = self.wing.get_cl(α_eff)

        # Make V_inf an array using prop_vi_h and V_inf
        V_inf_b_arr = V_inf + prop_vi_h * y_b_in_prop_wake

        Γ_b = 1/2 * V_inf_b_arr * chord_b * cl_b
        # Bool variable to check if we have converged solution of if we hit max iterations; total error
        converged = False
        Γ_b_diff = 0
        α_i = np.zeros(Γ_b.size)
        Γ_b_diff_arr = np.array([])

        # V_in is initialized to 0 plus propeller downwash
        V_in = np.zeros(Γ_b.shape) # + prop_vi_v * y_b_in_prop_wake
        for iteration in range(max_iterations):
            if debug:
                # Calculate lift distribution from Γ_b, total lift and coefficient of lift
                L_b = ρ_inf * V_inf_b_arr * Γ_b
                Lift = np.sum(L_b*y_b_Δ)
                CL = 2 * Lift / (self.wing.area * ρ_inf * (V_inf ** 2))
                print("Running iteration %d out of %d. Last error: %0.10f. Input CL: %0.3f" % (iteration+1, max_iterations,␣
↪Γ_b_diff, CL), end="\r")
            # 3. Calculate the trailed vortices using the input bounded vortices
            Γ_t = np.diff(Γ_b)
            Γ_t = np.concatenate(([Γ_b[0]], Γ_t,  [-Γ_b[-1]]))
            # print(Γ_t)
```

```
            # 4. Calculate the induced velocity V_in at each element n
            # V_in_λ_no_prop = lambda n : np.sum([-Γ_t[k]/(4*π*(y_b[n] - y_t[k])) for k in range (K)]) # + prop_vi_h *␣
↪y_b_in_prop_wake[n]
            # V_in_no_prop = np.array([V_in_λ_no_prop(n) for n in range(N)])

            V_in_λ = lambda n : np.sum([-Γ_t[k]/(4*π*(y_b[n] - y_t[k])) for k in range (K)]) + prop_vi_v * y_b_in_prop_wake[n]
            V_in = np.array([V_in_λ(n) for n in range(N)])
            # print("V_in NO PROP " + str(V_in_no_prop) + "\n" + "V_in WITH PROP" + str(V_in))

            # 5. Calculate the induced angle of attack
            α_i = np.arctan(- V_in / V_inf_b_arr)
            # α_i = -V_in / V_inf
            # 6. Calculate effective angle of attack
            # α_eff = α - α_i
            α_eff = α - α_i # - α_prop
            # print(α_eff)
            # 7. Calculate the new Γ_b
            cl_b = self.wing.get_cl(α_eff)
            Γ_b_new = 1/2 * V_inf_b_arr * chord_b * cl_b
            # 8. Compare difference between old and new Γ_b; if difference less than or equal to e_n then finished
            Γ_b_diff = np.sum(np.abs(Γ_b_new - Γ_b))
            Γ_b_diff_arr = np.append(Γ_b_diff_arr, Γ_b_diff)


            # plot the error
            draw_step = 20
            if draw and (iteration % draw_step == 0):
                size_X = 10
                clear_output(wait=True)
                # Now plot the error convergence
                pyplot.figure(figsize=(size_X, size_X))
                pyplot.axis("equal")
                pyplot.grid(zorder=0)
                pyplot.title("Error convergence")
                pyplot.ylabel('Error')
                pyplot.xlabel('Iterations')
                # Draw the divisions
                pyplot.hlines(e_n, 0, Γ_b_diff_arr.size, colors="red", label="Max error") # linestyles="dashed"
                pyplot.plot(range(Γ_b_diff_arr.size), Γ_b_diff_arr, label="Error of each iteration")
                pyplot.legend()
                pyplot.show()


            if np.abs(Γ_b_diff) <= e_n:
                if debug:
                    print("%s: solver has converged for α=%0.1f after %d iterations with error %0.10f." % (self.wing.label, np.
↪rad2deg(α), iteration+1, Γ_b_diff))
                converged = True
                break
            # 9. If solution has not converged re-run again and we're not at the last iteration
            elif iteration+1 < max_iterations:
                Γ_b = Γ_b + D * (Γ_b_new - Γ_b)
            else:
                break
                print("You have a bug in your code.")

        if not converged:
            print("")
            print("%s: solver has NOT converged for α=%0.1f after %d iterations!" % (self.wing.label, np.rad2deg(α),␣
↪max_iterations))
            print("Max error is: %0.10f Last error was: %0.10f" %(e_n, Γ_b_diff))

        # Calculate lift distribution from Γ_b, total lift and coefficient of lift
        L_b = ρ_inf * V_inf_b_arr * Γ_b
        Lift = np.sum(L_b*y_b_Δ)
        # print(Lift)
        # print(self.wing.area)
        # cl_b = 2 * L_b / (chord_b * ρ_inf * (V_inf_b_arr ** 2))
        CL = 2 * Lift / (self.wing.area * ρ_inf * (V_inf ** 2))
        # print(CL)

        Di_b = L_b * np.sin(α_i)
        Drag_i = np.sum(L_b*y_b_Δ)
        cd_b = 2 * Di_b / (y_b_Δ * chord_b * ρ_inf * (V_inf ** 2))
        CDi = 2 * Drag_i / (self.wing.area * ρ_inf * (V_inf ** 2))

        # Plot how we cut up the wing
        if draw:
            size_X = 10
            size_Y = size_X * (self.wing.c_root / self.wing.b + 0.2)
            pyplot.figure(figsize=(size_X, size_Y))
            pyplot.axis("equal")
            pyplot.grid(zorder=0)
            pyplot.title("Sections of wing")
```

```
            pyplot.ylabel('x (chord)')
            pyplot.xlabel('y (span)')
            # Draw the wing
            self.wing.draw()
            # Draw the divisions
            pyplot.vlines(y_t, -y_t_chords * 0.25, y_t_chords * 0.75, colors="gray", linestyles="dashed", label="Wing cuts")
            pyplot.legend()
            pyplot.show()

        self.last_solution = WingSolution(self.wing, α, V_inf, ρ_inf, y_b, y_t, y_b_Δ, chord_b, Γ_b, Γ_t, α_eff, α_i, V_in,␣
↪V_inf_b_arr, T)

        if draw:
            # print(y_b_Δ)
            self.last_solution.plot_all()

        # print(self.last_solution)
        return self.last_solution

    def plot_f(self, x, y, title="", xlabel="", ylabel='y (span)', xlim=None, ylim=None, size_X=10, size_Y=10):
        pyplot.figure(figsize=(size_X, size_Y))
        pyplot.grid(zorder=0)
        pyplot.title(title)
        pyplot.ylabel(xlabel)
        pyplot.xlabel(ylabel)
        # Draw the divisions
        pyplot.plot(x, y)
        if xlim is not None:
            pyplot.xlim(xlim)
        if ylim is not None:
            print("set Y")
            pyplot.ylim(ylim)
        pyplot.legend()
        pyplot.show()
        return

    def solve_multi_α(self, α_min=0, α_max=30, α_res=2, α_deg=True, V_inf=1, ρ_inf=1.225, T=0.0):
        α_arr = np.linspace(α_min, α_max, round ((α_max - α_min) / α_res) + 1)
        if α_deg:
            α_arr = np.deg2rad(α_arr)

        print("Parallelizing")
        pool = mp.Pool(mp.cpu_count())
        # f = lambda α : self.solve(α, V_inf=100, α_deg=False)
        f = functools.partial(self.solve, V_inf=V_inf, T=T, α_deg=False, ρ_inf=ρ_inf, draw=False, debug=False)
        solutions = list(pool.map(f, α_arr)) # [result[2] for result in list(pool.map(f, α_arr))]
        # print(cl_arr)
        pool.close()
        print("Done parallelizing " + self.wing.label)

        return solutions

    def print_solution(self):
        Lift, Drag_i, CL, CDi, L_b, Di_b, cl_b, cd_b, V_in = self.last_solution
        print(self.wing.label + " solution")
        print("Calculated Lift: %0.5f" % Lift)
        print("Calculated CL: %0.5f" % CL)
        print("Calculated induced drag: %0.5f" % Drag_i)
        print("Calculated CDi: %0.5f" % CDi)
        print()

def test_solver():
    wing = Wing(NACA0012_cl, 3.0480, 0.4572, 0.4572, ellipse=False, label="Square UAV wing")
    solver = NumericalLiftLineSolver(wing, N=50, max_iterations=1500, e_n=0.0005, D=0.01, polar=True)
    solution = solver.solve(8, 51.4, draw=True)

    solution.plot_all()
    # solver.print_solution()
    # solver.plot_polars(N=300, max_iterations=1500, e_n=0.0005, D=0.025, polar=False)

# test_solver()


def solve_multiple_wings_vs_α(wings, propellers=None, V_inf=1, T=0.0):

    α_min, α_max, α_res, α_deg = -2, 30, 1, True
    # α_min, α_max, α_res, α_deg = 1, 30, 1, True
    α_arr = np.linspace(α_min, α_max, round ((α_max - α_min) / α_res) + 1)
    if α_deg:
        α_arr = np.deg2rad(α_arr)
    wing_solution_array = []

    pool = mp.pool.ThreadPool(mp.cpu_count())
```

```
    # f = lambda α : self.solve(α, V_inf=100, α_deg=False)
    def solve_wing(wing):
        solver = NumericalLiftLineSolver(wing, propellers, N=200, max_iterations=5000, e_n=0.001, D=0.025, polar=False)
        α_solution_array = solver.solve_multi_α(α_min=α_min, α_max=α_max, α_res=α_res, α_deg=α_deg, V_inf=V_inf, ρ_inf=1.225,␣
↪T=T)
        return α_solution_array
    # f = functools.partial(solve_wing)
    wing_solution_array = list(pool.map(solve_wing, wings)) # [result[2] for result in list(pool.map(f, α_arr))]
    # print(cl_arr)
    pool.close()

    '''
    for wing in wings:
        solver = NumericalLiftLineSolver(wing, N=200, max_iterations=5000, e_n=0.001, D=0.05, polar=False)
        α_solution_array = solver.solve_multi_α(α_min=α_min, α_max=α_max, α_res=α_res, α_deg=α_deg, V_inf=1, ρ_inf=1.225)
        wing_solution_array.append(α_solution_array)
    '''

    return α_arr, wing_solution_array

def plot_wings_vs_α(α_arr, wing_solution_array, var_name, title, xlabel, ylabel, legend_label, setup=True):
    if setup:
        # Setup up the plot
        size_X, size_Y = 10, 10
        # Now plot the error convergence
        pyplot.figure(figsize=(size_X, size_X))
        pyplot.grid(zorder=0)
        pyplot.title(title)
        pyplot.xlabel(xlabel)
        pyplot.ylabel(ylabel)

        # Plot the airfoil lift curve
        if var_name == "CL":
            pyplot.plot(np.rad2deg(α_arr), wings[0].cl_f(α_arr), label=r"c_l curve for airfoil")

    # print(wing_solution_array)
    for α_solution_array in wing_solution_array:
        # print(α_solution_array)
        y_arr = [eval("solution."+var_name) for solution in α_solution_array]
        # print(y_arr)
        pyplot.plot(np.rad2deg(α_arr), y_arr, ".-", label=legend_label + " for " + α_solution_array[0].wing.label + " at␣
↪thrust = %0.1fN" % α_solution_array[0].T)
    pyplot.legend()

def plot_wings_vs_span(wing_solution_array, index, rad2deg, var_name, title, xlabel, ylabel, legend_label, setup=True):
    # Setup up the plot
    if setup:
        size_X, size_Y = 10, 10
        # Now plot the error convergence
        pyplot.figure(figsize=(size_X, size_X))
        pyplot.grid(zorder=0)
        pyplot.title(title)
        pyplot.xlabel(xlabel)
        pyplot.ylabel(ylabel)

    # Plot the airfoil lift curve
    # if var_name == "CL":
    #    pyplot.plot(np.rad2deg(α_arr), wings[0].cl_f(α_arr), label=r"c_l curve for airfoil")

    # print(wing_solution_array)
    for α_solution_array in wing_solution_array:
        # print(α_solution_array)
        x_arr = α_solution_array[0].y_b
        y_arr = eval(str("α_solution_array[%d]." % index)+var_name )
        # print(y_arr)
        if rad2deg:
            y_arr = np.rad2deg(y_arr)
        α = np.rad2deg( α_solution_array[index].α )
        pyplot.plot(x_arr, y_arr, ".-", label=legend_label + " for " + α_solution_array[0].wing.label + str(" at α=%0.1f°" %␣
↪α) + str(" and thrust=%0.1fN" % α_solution_array[index].T))
        pyplot.legend()

# plot_polars(wings)


AR_wings = np.empty(5, dtype=Wing)
AR_wings[0] = Wing(NACA0012_cl, 10, 1, label="AR = 10:1")
AR_wings[1] = Wing(NACA0012_cl, 20, 1, label="AR = 20:1")
AR_wings[2] = Wing(NACA0012_cl, 30, 1, label="AR = 35:1")
AR_wings[3] = Wing(NACA0012_cl, 40, 1, label="AR = 50:1")
AR_wings[4] = Wing(NACA0012_cl, 50, 1, label="AR = 100:1")

# α_arr, wing_solution_array = plot_multiple_wings_α(AR_wings)
```

```
# α_arr, wing_solution_array = solve_multiple_wings_vs_α(wings)
```

[6]:
```
# plot_wings_vs_α(α_arr, wing_solution_array, "CL", "Coefficient of lift vs α", "α", "CL", "CL")
# plot_wings_vs_α(α_arr, wing_solution_array, "CDi", "Coefficient of induced drag vs α", "α", "CDi", "CDi")

# plot_wings_vs_span(wing_solution_array, 8+2, True, "α_eff", "Effective angle of attack vs span at α of 8°", "span",␣
↪r"$α_{eff}$", r"$α_{eff}$")
# plot_wings_vs_span(wing_solution_array, 8+2, True, "cl_b", "Coefficient of lift c_l vs span at α of 8°", "span", "cl", "cl_b")
```

2 Problem solution

2.1 Wing performance without propellers

First I analyzed the performance of the wing without modelling the effect of the propellers. The focus was on creating an anlysis to understand at what overall angle of attack the wing starts to stall.

To assess the overall stall angle of attack for the wing, I created two analyses:

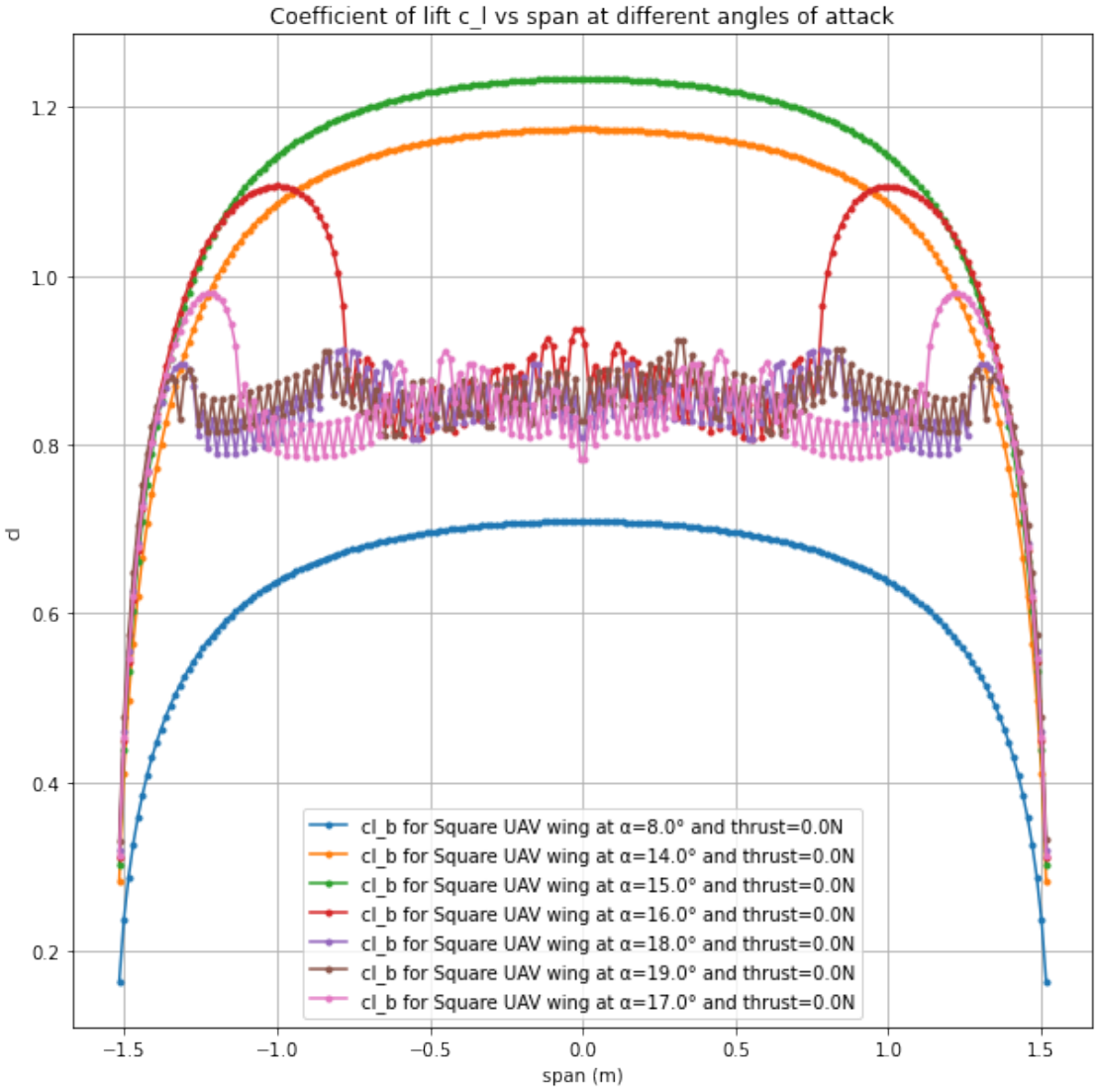
1. Plotting how the local wing cl distribution changes as a function of angle of attack (as requested in the assignment).
2. Calculating and graphing the overall coefficient of lift for the wing. Although this was not specifically requested in the assignment, it is an additional analysis I created to see if more insights could be learned.

2.1.1 Local wing cl distribution vs angle of attack

Based on the chart of the NACA 0012 coefficient of lift vs. angle of attack, it is clear that the airfoil itself stalls around α=14.2°. Therefore, I selected angles of attack for this analysis close to the airfoil's stall angle of attack.

[7]:
```
wing = Wing(NACA0012_cl, 3.0480, 0.4572, 0.4572, ellipse=False, label="Square UAV wing")
# solver = NumericalLiftLineSolver(wing, N=50, max_iterations=1500, e_n=0.0005, D=0.01, polar=True)
wings = np.array([wing])
α_arr, no_propeller_solution_array = solve_multiple_wings_vs_α(wings, V_inf = 51.4444)

plot_wings_vs_span(no_propeller_solution_array, 8+2, False, "cl_b", "Coefficient of lift c_l vs span at different angles of␣
↪attack", "span (m)", "cl", "cl_b")

plot_wings_vs_span(no_propeller_solution_array, 14+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 14°", "span␣
↪(m)", "cl", "cl_b", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 15+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 15°", "span␣
↪(m)", "cl", "cl_b", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 16+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 16°", "span␣
↪(m)", "cl", "cl_b", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 18+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 18°", "span␣
↪(m)", "cl", "cl_b", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 19+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 19°", "span",␣
↪"cl", "cl_b", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 17+2, False, "cl_b", "Coefficient of lift c_l vs span at α of 19°", "span",␣
↪"cl", "cl_b", setup=False)
```

```
Parallelizing
Done parallelizing Square UAV wing
```



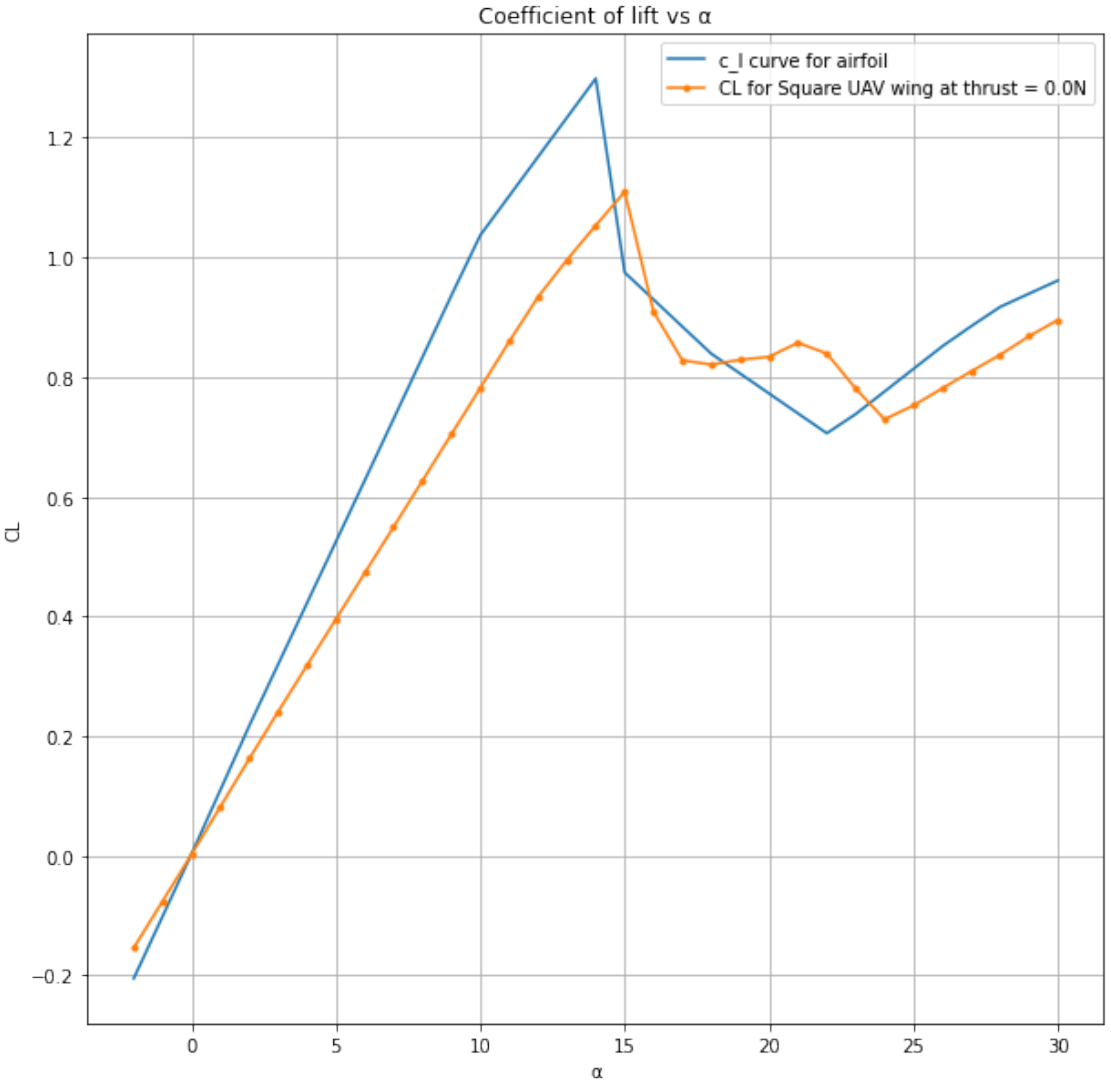
In the above charts we can see how the stall progresses across the airfoil wing as the angle of attack exceeds the stall angle. At $\alpha = 15°$, the aircraft wing still has a normal coefficient of lift distribution, indicating that the wing itself has not stalled yet. At $\alpha = 16°$, we see that the center of the wing starts to stall. The center of the wing stalls first because it has the angle of attack; the angle of attack on the wing is reduced as we approach the wing tips due to the effect of the induced downwash velocity. Therefore, the center of the wing is stalled at this angle of attack, but the wing tips have not stalled yet.

We can see that at $\alpha = 17°$ the stalled region of the wing progresses outwards, until at $\alpha = 18°$ and $\alpha = 19°$ the aircraft wing is now fully stalled, including the wing tips.

2.1.2 Stall angle of attack for wing from overall coefficient of lift

Although not requested in the assignment, I also calculated the overall coefficient of lift for the whole wing to observe at what angle of attack the wing stalls.

[8]:
```
plot_wings_vs_α(α_arr, no_propeller_solution_array, "CL", "Coefficient of lift vs α", "α", "CL", "CL")
```



In the above chart we can see that the coefficient of lift increases linearly with the angle of attack until a peak at $\alpha = 15°$. At $\alpha = 15°$, there is a sharp drop in the coefficient of lift, indicating that this is the stall angle of attack. This is in line with what we saw in the local coefficient of lift distribution: at $\alpha = 15°$, the wing still had a normal coefficient of lift distribution, but beyonf $\alpha = 15°$ the wing started experiencing a stall.

2.2 Wing performance with propellers

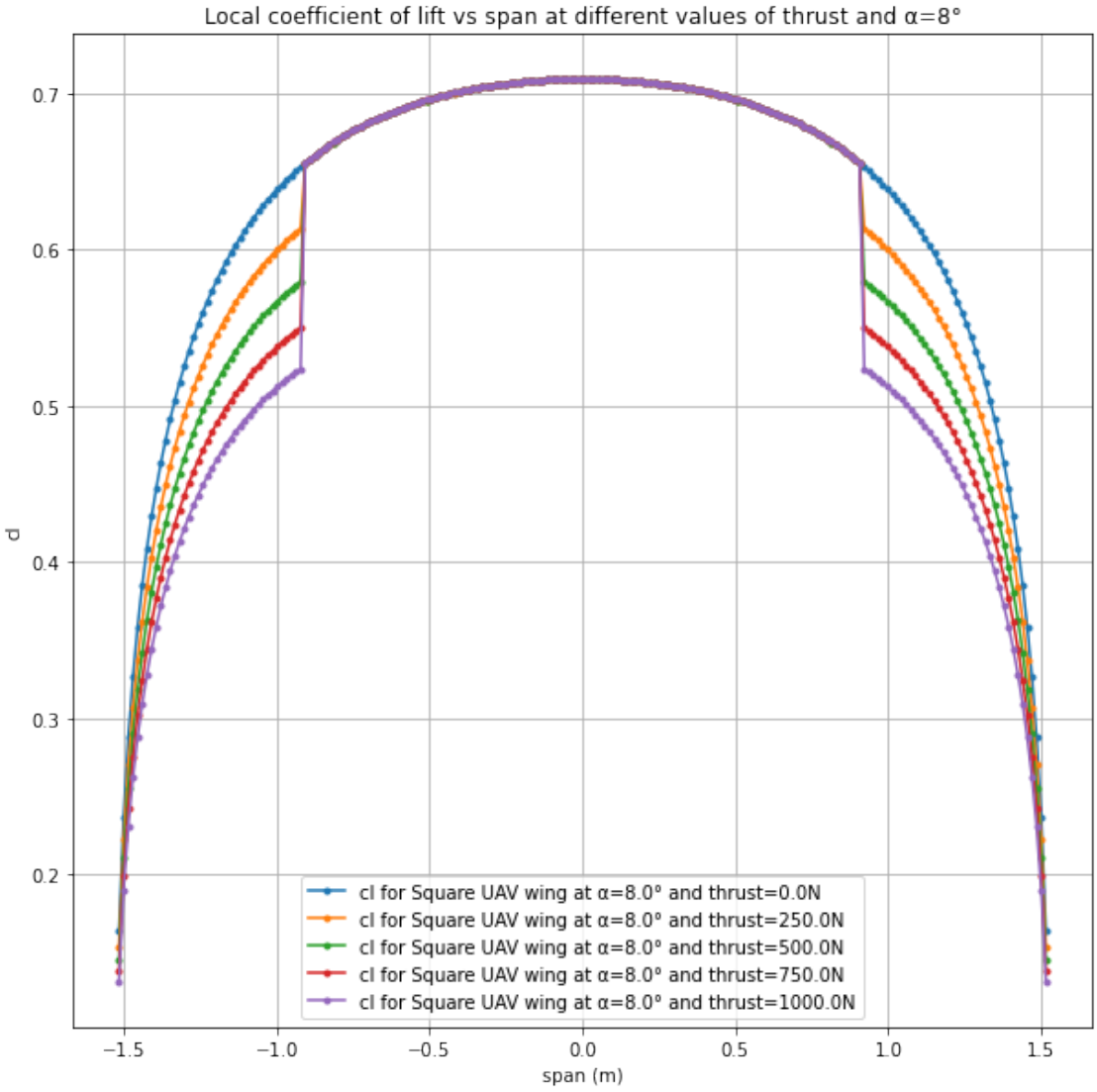
My analysis of the wing performance with propellers focused on evaluating the effect of the propeller wake on the wing performance. Specifically, the portion of the wing behind the propeller will 'see' a different angle of attack and freestream velocity due to the wake of the propeller. Using the equations provided in the assignment and the methodology described in the methodology section above, I estimated how the local wing coefficient of lift varies as a function of propeller thrust and angle of attack.

2.2.1 Evaluating how local wing coefficient of lift varies as a function of propeller thrust

At an angle of attack of $\alpha = 8°$, the total lift force on the UAV wing is 1417N. Therefore, to analysis how the local wing coefficient of lift varies as a function of propeller thrust, I used values of 0, 250N, 500N, 750N and 1000N, i.e., going up to a thrust to weight ratio of ~0.7, which is a reasonable number to capture the full flight envelope of the UAV (the max T/W ratio of 0.7 assumes the UAV lift force is equal to its weight at $\alpha = 8°$ and $V_\infty = 51.444$ m/s, i.e. that it would be in steady cruise flight.

[11]:
```
wing = Wing(NACA0012_cl, 3.0480, 0.4572, 0.4572, ellipse=False, label="Square UAV wing")
# solver = NumericalLiftLineSolver(wing, N=50, max_iterations=1500, e_n=0.0005, D=0.01, polar=True)
wings = np.array([wing])
propellers = Propellers(1.2192, y1=3.0480 / 2, y2=-3.0480 / 2)

print("Propeller vi: %0.5f" % propellers.calc_vi(4000, 51.4444, 1.225))

α_arr, propeller_solution_array_T0 = solve_multiple_wings_vs_α(wings, propellers, V_inf = 51.4444, T=0)
α_arr, propeller_solution_array_T250 = solve_multiple_wings_vs_α(wings, propellers, V_inf = 51.4444, T=250)
α_arr, propeller_solution_array_T500 = solve_multiple_wings_vs_α(wings, propellers, V_inf = 51.4444, T=500)
α_arr, propeller_solution_array_T750 = solve_multiple_wings_vs_α(wings, propellers, V_inf = 51.4444, T=750)
α_arr, propeller_solution_array_T1000 = solve_multiple_wings_vs_α(wings, propellers, V_inf = 51.4444, T=1000)

plot_wings_vs_span(propeller_solution_array_T0, 8+2, False, "cl_b", "Local coefficient of lift vs span at different values of␣
↪thrust and α=8°", "span (m)", "cl", "cl")
plot_wings_vs_span(propeller_solution_array_T250, 8+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 8+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T750, 8+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T1000, 8+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
```

```
Propeller vi: 19.66618
Parallelizing
Done parallelizing Square UAV wing
Parallelizing
Done parallelizing Square UAV wing
Parallelizing
Done parallelizing Square UAV wing
Parallelizing
Done parallelizing Square UAV wing
Parallelizing
Done parallelizing Square UAV wing
```

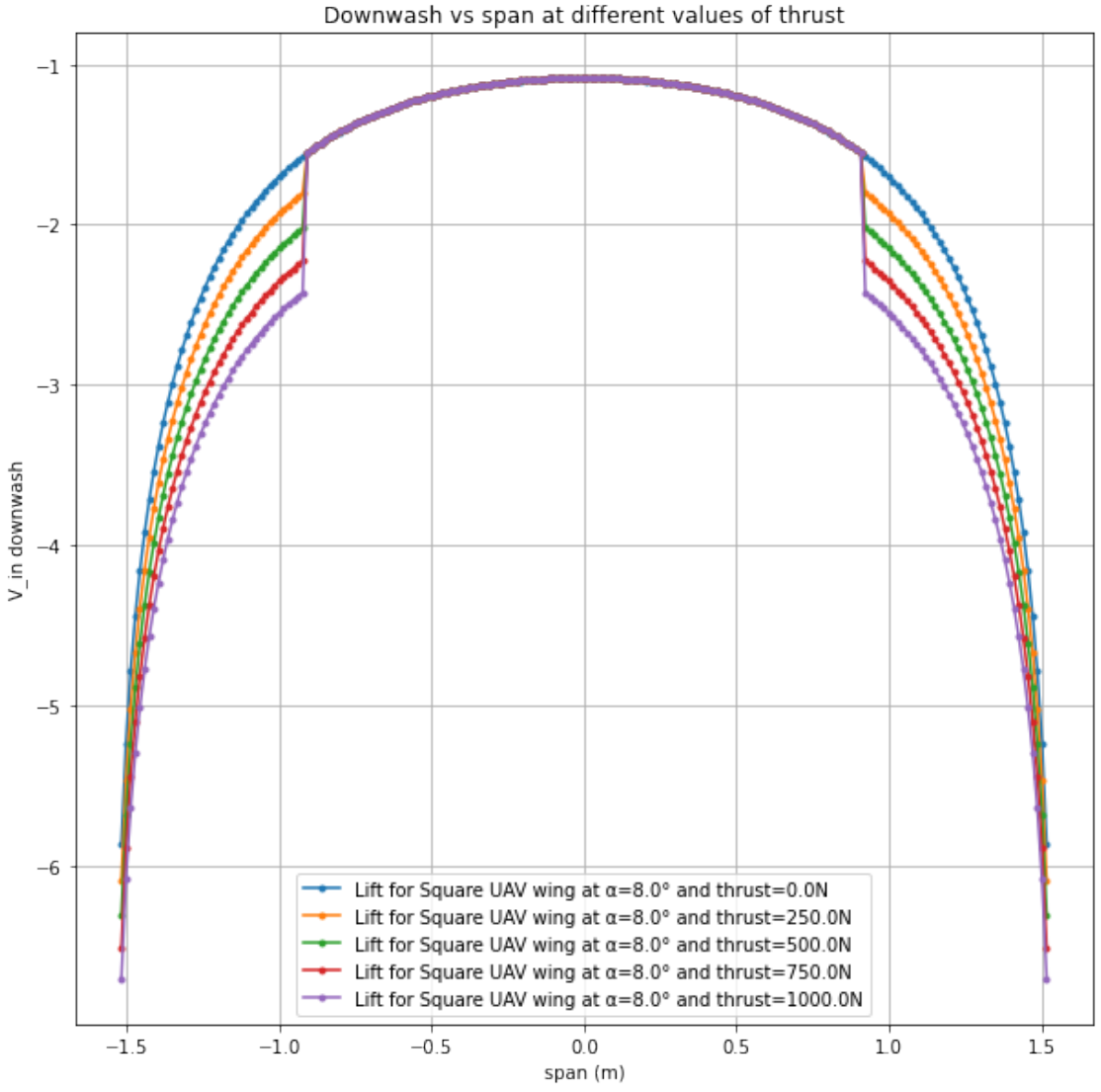


In the above chart we can observe a few effects of the propeller wing on the coefficient of lift across the wing span:

1. The propeller wake reduces the coefficient of lift of the wing behind the propeller. This is because the propeller wake effectively reduces the angle of attack of that section of the wing by creating a flow field parallel to the wing chord.
2. As one would expect, the size of this effect increases as the thrust of the propeller increases.
3. Just because the coefficient of lift decreases, doesn't necessarily mean that the lift generated by that section of the aircraft wing would decrease. The propeller generates increased air flow over that wing section, which offsets some, or perhaps even all, of the effect of the decrease in coefficient of lift.
4. As discussed in office hours with Dr. Mistry, intuitively, one would expect the coefficient of lift to have a smoother transition curve between the wing section behind the propeller and the center wing

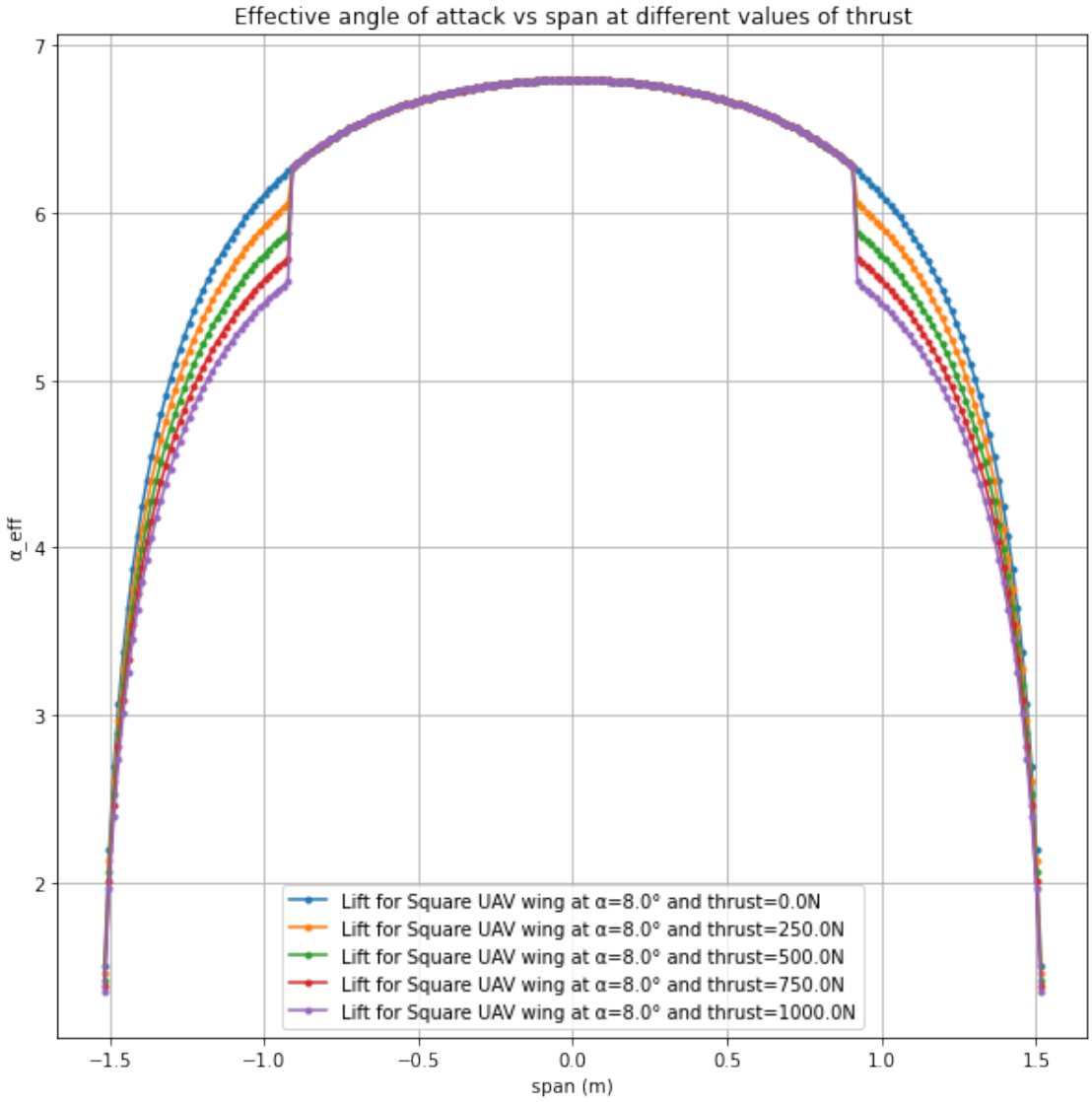
section. The reason for this intuitive hypothesis is that the vorticity at the transition location would be smoothed out due to the sudden change in lift as we go through the iterative loop. But this is not the case here. I went back and triple checked my code, didn't find any errors and even played around with changing the propeller parameters.

[12]:
```
plot_wings_vs_span(propeller_solution_array_T0, 8+2, False, "V_in", "Downwash vs span at different values of thrust", "span
↪(m)", "V_in downwash", "Lift")

plot_wings_vs_span(propeller_solution_array_T250, 8+2, False, "V_in", "", "span (m)", "V_in downwash", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 8+2, False, "V_in", "", "span (m)", "V_in downwash", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T750, 8+2, False, "V_in", "", "span (m)", "V_in downwash", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T1000, 8+2, False, "V_in", "", "span (m)", "V_in downwash", "Lift", setup=False)
```

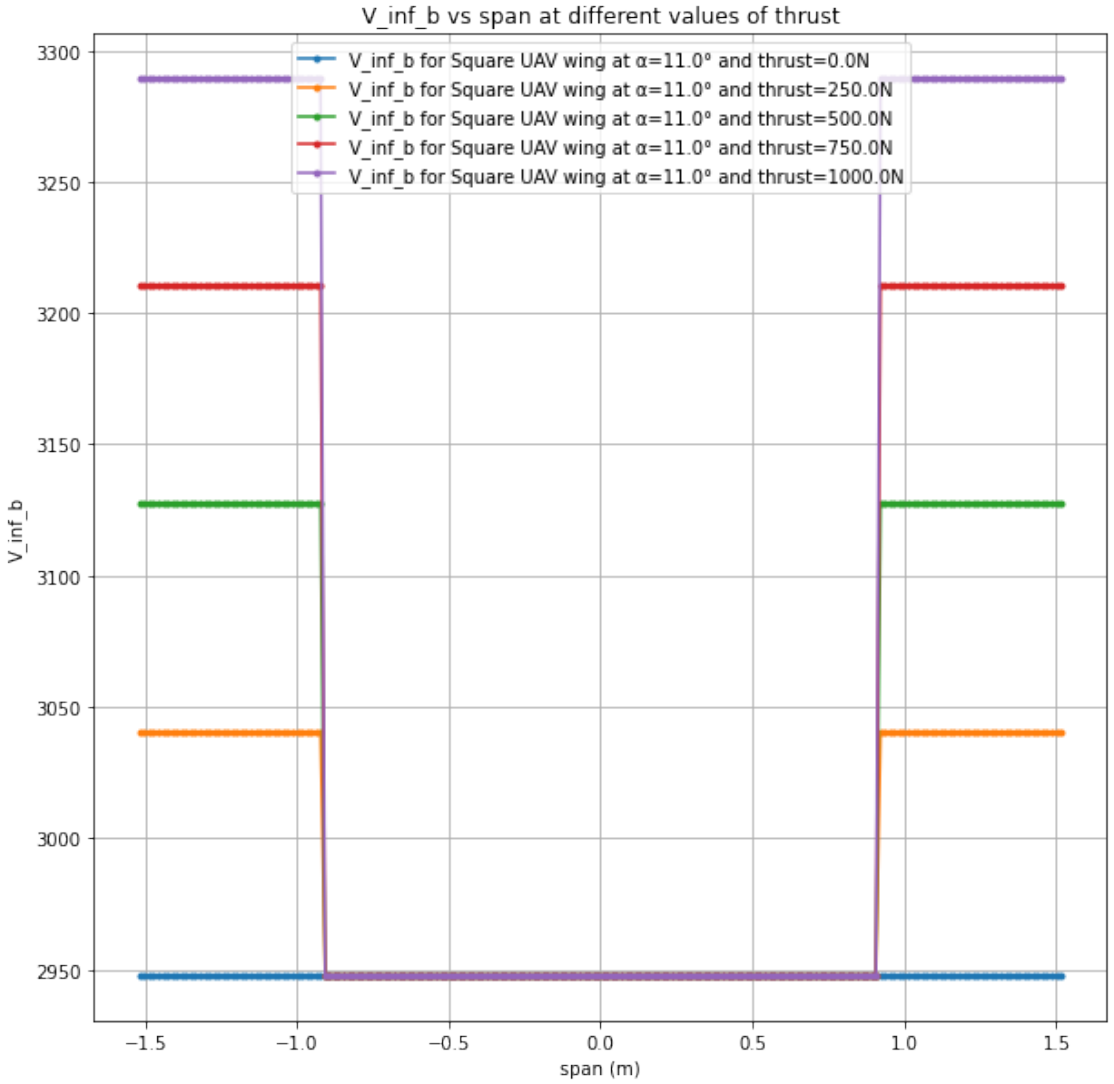


Given that the downwash directly impacts the effective angle of attack, and hence coefficient of lift, the downwash distribution across the span of the wing has the same shape as the local coefficient of lift curve.

[13]:
```
plot_wings_vs_span(propeller_solution_array_T0, 8+2, True, "α_eff", "Effective angle of attack vs span at different values of␣
↪thrust", "span (m)", "α_eff", "Lift")

plot_wings_vs_span(propeller_solution_array_T250, 8+2, True, "α_eff", "", "span (m)", "α_eff", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 8+2, True, "α_eff", "", "span (m)", "α_eff", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T750, 8+2, True, "α_eff", "", "span (m)", "α_eff", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T1000, 8+2, True, "α_eff", "", "span (m)", "α_eff", "Lift", setup=False)
```



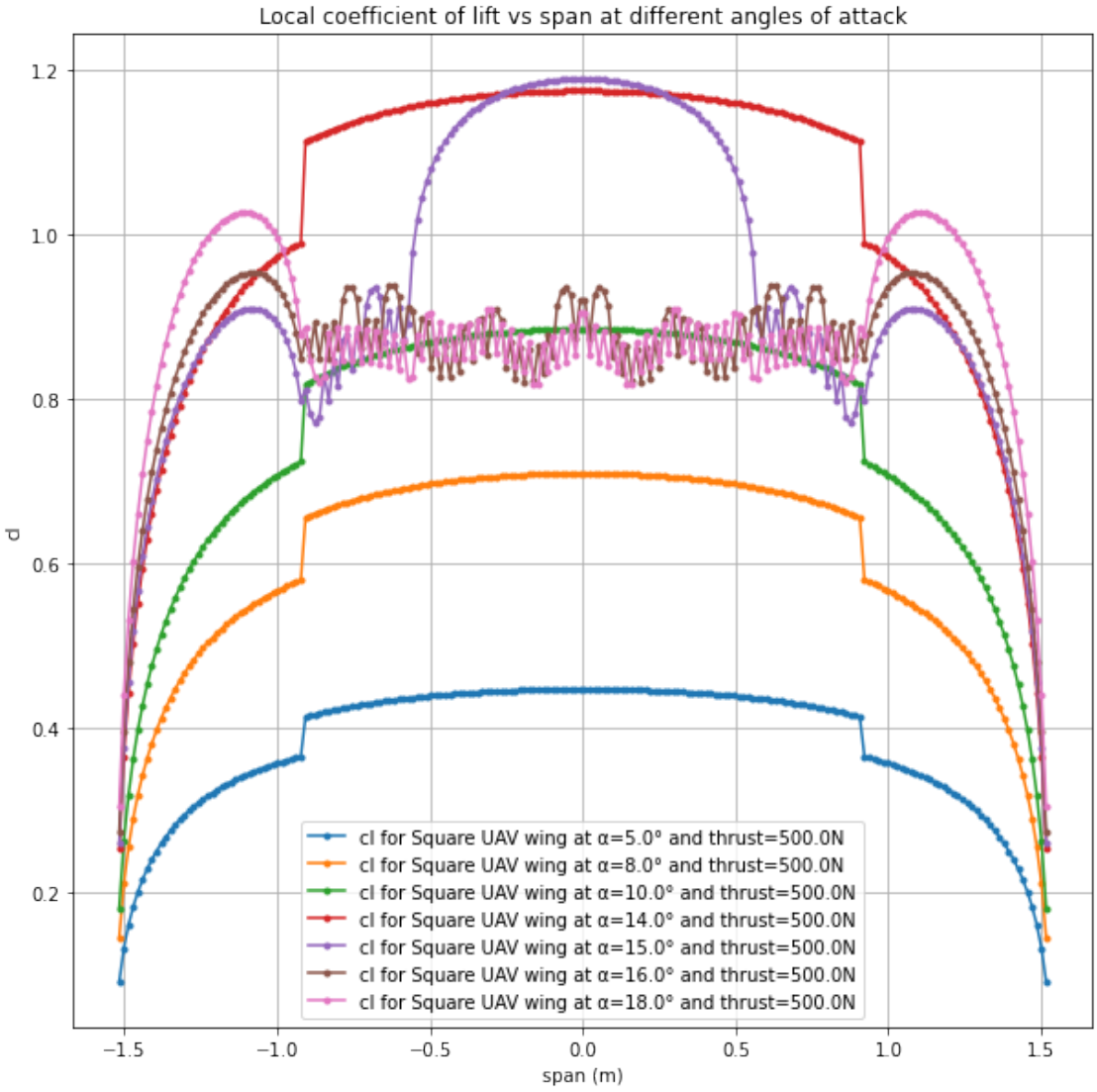
[14]:
```
plot_wings_vs_span(propeller_solution_array_T0, 11+2, True, "V_inf_b", "V_inf_b vs span at different values of thrust", "span␣
↪(m)", "V_inf_b", "V_inf_b")

plot_wings_vs_span(propeller_solution_array_T250, 11+2, True, "V_inf_b", "", "span (m)", "V_inf_b", "V_inf_b", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 11+2, True, "V_inf_b", "", "span (m)", "V_inf_b", "V_inf_b", setup=False)
plot_wings_vs_span(propeller_solution_array_T750, 11+2, True, "V_inf_b", "", "span (m)", "V_inf_b", "V_inf_b", setup=False)
plot_wings_vs_span(propeller_solution_array_T1000, 11+2, True, "V_inf_b", "", "span (m)", "V_inf_b", "V_inf_b", setup=False)
```



2.2.2 Evaluating how local wing coefficient of lift varies as a function of the angle of attack

For this analysis, I kept thrust constant at 500N and changed the angle of attack to observe how the local coefficient of lift changed. Again I focused most of my analysis to angles of attack close to the stall angle, to see if the way in which the stall develops on the wing has been affected by the propeller.

[15]:
```
plot_wings_vs_span(propeller_solution_array_T500, 5+2, False, "cl_b", "Local coefficient of lift vs span at different angles␣
↪of attack", "span (m)", "cl", "cl")

plot_wings_vs_span(propeller_solution_array_T500, 8+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 10+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 14+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 15+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 16+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 18+2, False, "cl_b", "", "span (m)", "cl", "cl", setup=False)
```



In the above chart, we can see how the angle of attack affects the local coefficient of lift. There are several interesting observations to make.

1. The aircraft wing already begins to stall at $\alpha = 15°$, whereas without the propellers $\alpha = 15°$ still had a normal lift distribution. Also, it is interesting to note that this time, the stall develops at the transition point from propeller to non-propeller wing span. Additionally, even though a portion of the wing stalls, the center section of the wing still generates significantly more lift and has not stalled yet. Therefore, it appears that around the transition points between propeller and non-propeller wing spans there is actually a high effective angle of attack.

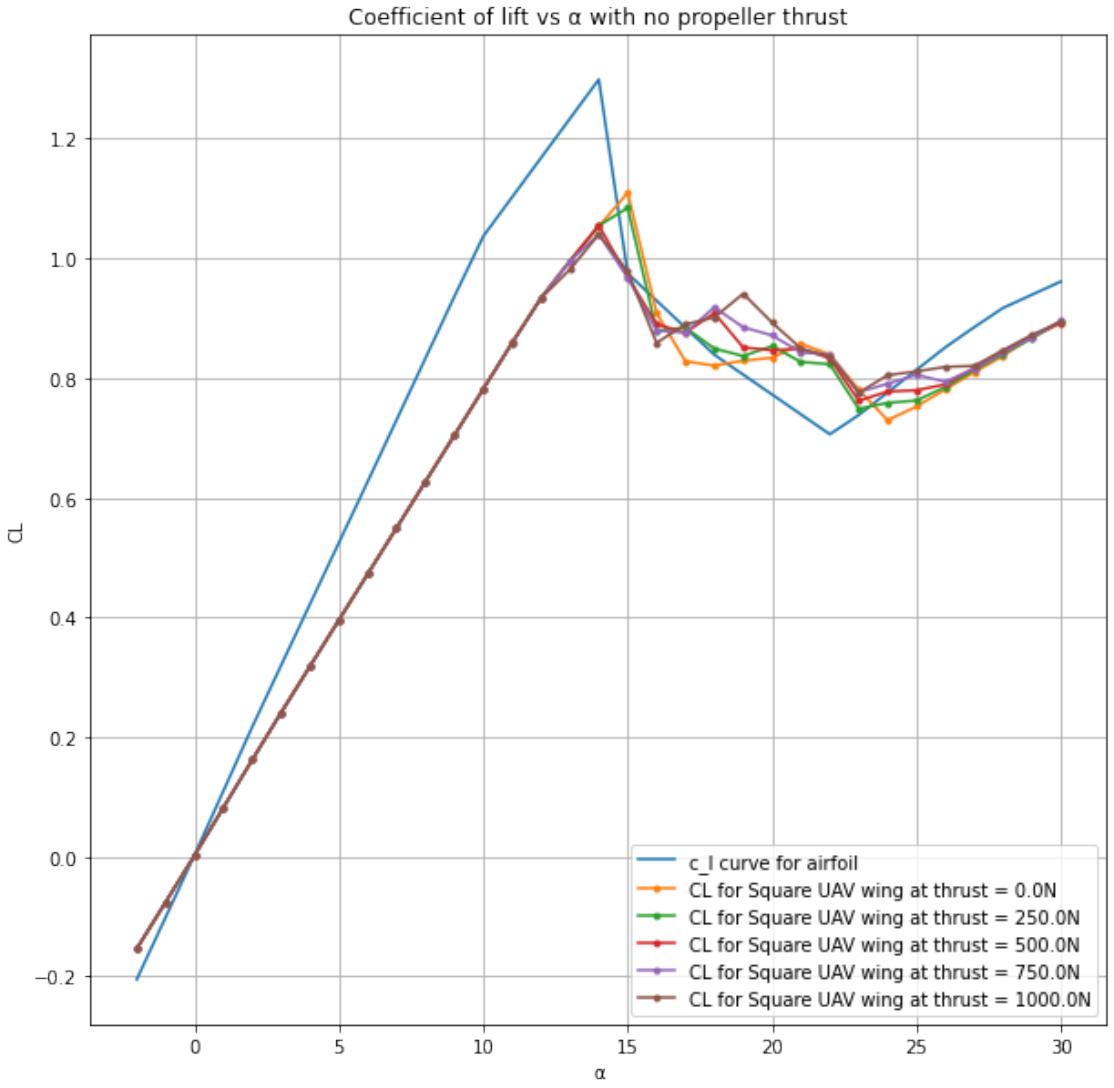
2. The propeller reduces the angle of attack at the wing tips of the wing. As a result, these wing tips experienced a significantly delayed stall characteristic. Previously, we saw that at $\alpha = 18°$, the aircraft wing without propellers had fully stalled. However, in this case, only the non-propeller section of the wing has stalled. The propeller section still has laminar flow, and as we increase the angle of

attack beyond $\alpha = 15°$, the wing tips continue generating more lift, whereas the stalled center section generates approximate the same amount of lift.

3 Appendix

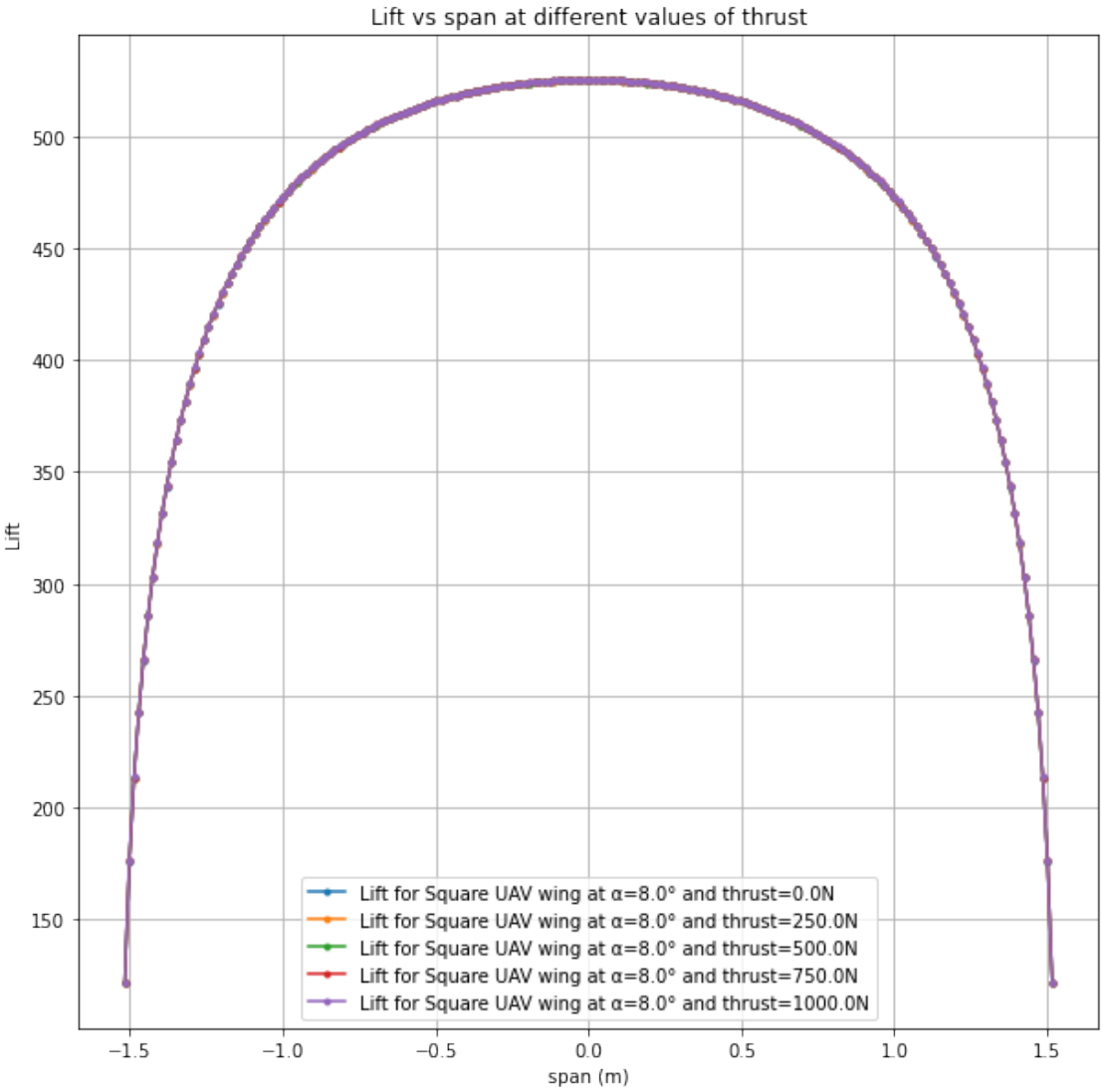
During my analysis, I generated a lot of additional charts and graph to help with understanding the physics behind what is going on in this situation, to cross-check my answers and learn new insights. A lot of the charts are included here in the appendix. I also didn't spend too much time debugging these as they are beyond the scope of the assignment.

[16]:
```
plot_wings_vs_span(no_propeller_solution_array, 8+2, True, "α_eff", "Effective angle of attack vs span at different angles of␣
↪attack", "span (m)", "α_eff", "α_eff")

plot_wings_vs_span(no_propeller_solution_array, 14+2, True, "α_eff", "Coefficient of lift c_l vs span at α of 14°", "span␣
↪(m)", "α_eff", "α_eff", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 15+2, True, "α_eff", "Coefficient of lift c_l vs span at α of 15°", "span␣
↪(m)", "α_eff", "α_eff", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 16+2, True, "α_eff", "Coefficient of lift c_l vs span at α of 16°", "span␣
↪(m)", "α_eff", "α_eff", setup=False)

plot_wings_vs_span(no_propeller_solution_array, 18+2, True, "α_eff", "Coefficient of lift c_l vs span at α of 18°", "span␣
↪(m)", "α_eff", "α_eff", setup=False)

# plot_wings_vs_span(no_propeller_solution_array, 20+2, True, "cl_b", "Coefficient of lift c_l vs span at α of 18°", "span",␣
↪"cl", "cl_b", setup=False)
```



[18]:
```
plot_wings_vs_span(propeller_solution_array_T0, 8+2, False, "L_b", "Lift vs span at different values of thrust", "span (m)",␣
↪"Lift", "Lift")

plot_wings_vs_span(propeller_solution_array_T250, 8+2, False, "L_b", "", "span (m)", "Lift", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T500, 8+2, False, "L_b", "", "span (m)", "Lift", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T750, 8+2, False, "L_b", "", "span (m)", "Lift", "Lift", setup=False)
plot_wings_vs_span(propeller_solution_array_T1000, 8+2, False, "L_b", "", "span (m)", "Lift", "Lift", setup=False)

print(propeller_solution_array_T0[0][8+2].Lift)
```

```
1416.1928492365314
```


Lift vs span at different values of thrust
Lift
span (m)
500
450
400
350
300
250
200
150
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
Lift for Square UAV wing at α=8.0° and thrust=0.0N
Lift for Square UAV wing at α=8.0° and thrust=250.0N
Lift for Square UAV wing at α=8.0° and thrust=500.0N
Lift for Square UAV wing at α=8.0° and thrust=750.0N
Lift for Square UAV wing at α=8.0° and thrust=1000.0N