Projekt 1: Styrning av obemannad raket

Numeriska metoder och Simulering

Tuva Björnberg, Nora Reneland, Matilda Stenbaek

September 2024

1 Bakgrund

Uppgiften är att styra en obemannad raket mot ett givet mål med en begränsad mängd bränsle att tillgå. Följande differentialekvation beskriver raketens bana:

$$m(t)\bar{a}(t) = \bar{F} + m'(t)\bar{u}(t)$$

Summan av de krafter som verkar på raketen kan ses genom ekvationen nedan. Förutom tyngdkraften $m(t)\bar{g}$ verkar också luftmotståndet, som är omvänt proportionellt mot hastigheten i kvadrat, på raketen. $m'(t)\bar{m}(t)$ beskriver kraften raketen utsätts för genom att partiklar (bränsle) skjuts ut från raketen. Hastighetsvektorn $\bar{u}(t)$ är en funktion av t och anger riktning och fart av bränslet som skjuts ut från raketen.

$$F = m(t)\bar{g} - c||\bar{v}(t)||\bar{v}(t)$$

Initiala raketmassan är 8 kg, varav 4 kg är bränsle. Under tiden som motorn är igång förbränner den 0.4 kg bränsle per sekund. Detta gör att motorn kan vara igång som mest 10 sekunder. Om vi startar raketmotorn vid t = 0 och håller motorn igång tills bränslet är slut kommer raketens massfunktion $m(t)$ att se ut enligt följande:

$$m(t) = \begin{cases} 8 - 0.4t, & t \leq 10 \\ 4, & t > 10 \end{cases}$$

Hastighetsvektorn för bränslet $\bar{u}(t)$ är beroende av masspartiklarna och deras konstanta fart, k_m , ut från motorn. Med motorns riktning $\theta(t)$ kan vi nu styra fritt.

$$\bar{u}(t) = \begin{pmatrix} u_x(t) \\ u_y(t) \end{pmatrix} = \begin{pmatrix} k_m \cos(\theta(t)) \\ k_m \sin(\theta(t)) \end{pmatrix}$$

$\theta(t)$ anger motorns vinkel från den positiva x-axeln. För att raketen ska åka rakt upp krävs då alltså att $\theta(t) = -\frac{\pi}{2}$. Vi antar att raketen börjar i stillastående läge, dvs. $\bar{v}(0) = (0, 0)$. Raketen måste börja färdas rakt upp tills den når en höjd på 20 meter, först därefter kan den styras mot målet.

Uppgiften är att experimentellt implementera olika strategier för att ta fram en styrfunktion, $\theta(t)$, som kan användas för att styra raketen mot målet. Ekvationerna ska lösas både med Pythons solve_ivp och med en egen Runge-Kutta-lösare.

2 Matematisk modell

Följande konstanter användes vid samtliga beräkningar:

- $g = 9.82$ m/s^2 (gravitationskonstanten)
- $c = 0.05$ kg/m

- k_m = 700 m/s (masspartiklarnas konstant fart)

De matematiska modellerna som utvecklades är alla beroende av ett state vi skapade för att hålla reda på varje tidsstegs x-position, y-position, hastighet i x-led och hastighet i y-led. Alla initierades till 0:

```
initial_state = [0, 0, 0, 0]
```

State genomgår ODE-beräkningarna där hastigheterna i x- och y-riktningarna samt accelerationerna returneras som derivatan till positioner respektive hastigheter.

```
state_derivatives = [vx, vy, acc[0], acc[1]]
```

2.1 Styrfunktioner

För att modellera raketens bana beräknas två kraftvektorer. Den ena är baserad på raketens styrkraft, den kraft bränslet ger, i kombination med styrfunktionen. Den andra vektorn representerar de externa krafterna, som gravitationen och luftmotståndet. Dessa två kraftvektorer adderas och används för att räkna ut den totala kraften som verkar på raketen och på så sätt även raketens acceleration.

Den generella strategin för styrfunktionen är att räkna ut vinkeln mellan raketens nuvarande position och målet. De två första versionerna av styrfunktionen räknar ut vinkeln genom att ta sträckan mellan raketen och målet i x- och y-led och stoppa in dem i en tangens- respektive arctangens2-funktion. Den första strategin med tangens fungerar endast för vissa mål och endast om raketen och målet befinner sig i den första kvadranten. Vi bytte därför till arctan2 för att göra vår lösning mer flexibel och robust. Ingen av dessa två strategier tar dock hänsyn till de externa krafter som verkar på raketen, och den missar därmed sitt mål.

Den tredje strategin gör en ansats att anpassa vinkeln som fås av arctan2() genom att minska vinkeln baserat på avståndet. Ju längre bort från målet raketen befinner sig, desto snävare görs vinkeln. Beslutet om just denna strategi togs baserat på tidigare körningar där raketen kontinuerligt sköts över målet. Detta sker för att på raketen verkar en stark kraft uppåt, som följd av den ursprungliga kraftiga accelerationen uppåt. Det krävs därför att raketen svänger snävare i början för att motverka den kraften. Vinkeln planas ut desto närmre målet vi kommer.

Den sista strategin använder sig av två vinklar. Dels vinkeln relativt mot marken som raketen för tillfället åker i, det vill säga raketens nuvarande riktning, och dels vinkeln mellan raketen och målet relativt mot marken. Skillnaden mellan dessa två vinklar, raketens nuvarande riktning och den optimala riktningen mot målet, räknas ut och adderas sedan till vinkeln för raketens optimala riktning. Detta gör att raketens riktning justeras mer drastiskt om raketens nuvarande bana är långt ifrån den optimala, och mindre drastiskt om den nuvarande riktningen är nära den optimala för att träffa målet.

Den sista strategin kommer inte att vända tillbaka mot målet om den inte träffar eftersom corrected_angle kommer att förbli den samma. Detta eftersom skillnaden mellan den idealla riktningen och den nuvarande riktningen blir ca 180° och den idealla riktningen + skillnaden då blir ca 360°, alltså ingen skillnad.

Raketens beteende efter att den passerat målet är inte kodat för att vända tillbaka och försöka komma närmare målet. Styrfunktionen prioriterar istället att försöka komma så nära målet som möjligt första gången raketen passerar.

Styrfunktionerna förutsätter att motorn kan justeras och styras idealt, vilket innebär att den inte påverkas av några utomstående krafter såsom luftmotståndet eller mekaniska trögheter. Vinklar som exempelvis corrected_angle gör att motorn svänger direkt efter beräkningen.

Hela simuleringen kommer också att avslutas när raketen når marken, alltså när y_pos är 0. Med den sista strategin krashar raketen ungefär när tidspannet går upp till 40 sekunder.

3 Körexempel

Tre olika mål valdes för att illustrera att styrfunktionen fungerar även när exempelvis målet befinner sig till vänster om raketens ursprungliga position och när det är en snäv vinkel. Vi undersökte huvudsakligen målen (x-, y-punkterna):

- $(80, 60)$, Röd
- $(-40, 80)$, Grön
- $(20, 90)$, Gul

Figur 1 och 2 beskriver den sista och mest optimerade strategin medan figur 3 och 4 beskriver vår första iteration av styrfunktion.

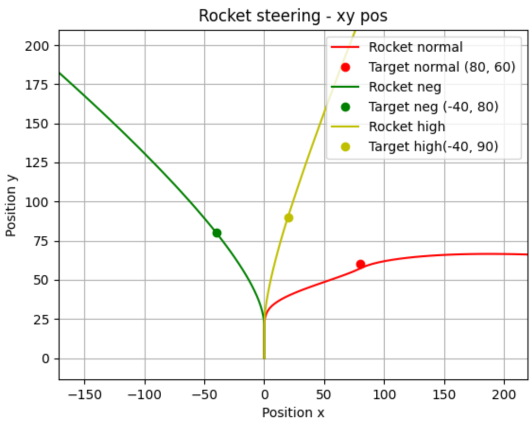

Figure 1: Den sista strategin med tre olika mål, löst med solve_ivp

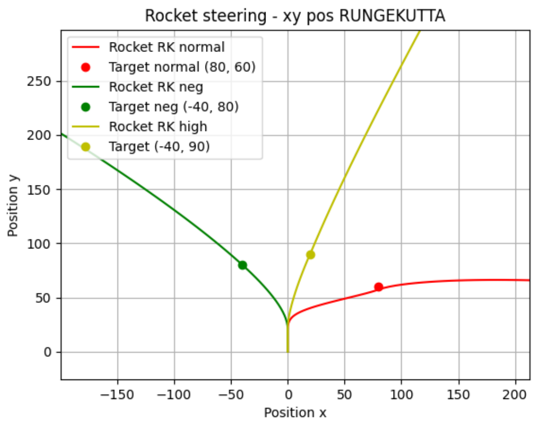

Figure 2: Den sista strategin med tre olika mål, löst med egen Runge-Kuttalösare

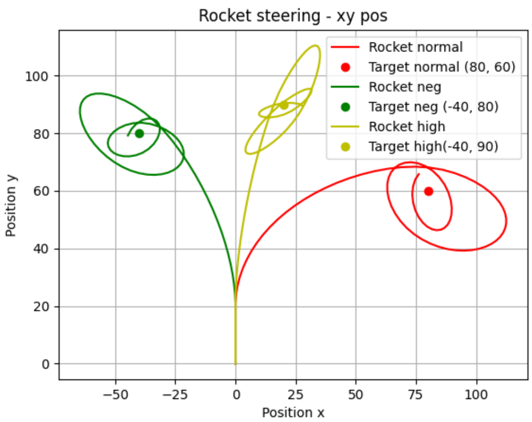

Figure 3: Andra strategin (arctan2) med tre olika mål, löst med solve-ivp

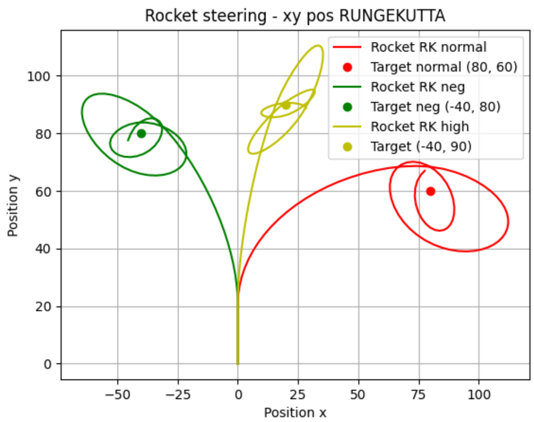

Figure 4: Andra strategin (arctan2) med tre olika mål, löst med egen Runge-Kuttalösare

A Koden

```
import numpy as np
import scipy
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Rocket constants
g = 9.82
c = 0.05
k = 700
burn_rate = 0.4
initial_rocket_mass = 8
initial_fuel_mass = 4

# Time constants
t_max = initial_fuel_mass / burn_rate
t_span = (0, 10 + 1.0e-14)  # time intervall

# Target constants
norm_target = (80, 60)
neg_target = (-40, 80)
high_target = (20, 90)

initial_state = [0, 0, 0, 0]  # [x_pos, y_pos, x_velocity, y_velocity]


# Mass of the rocket with the burning of fuel in regards
def mass(t):
    if t <= t_max:
        return initial_rocket_mass - burn_rate * t
    else:
        return initial_rocket_mass - initial_fuel_mass


# Rate of which the mass changes, the fuel is consumed
def mass_der(t):
    if t <= t_max:
        return -burn_rate
    else:
        return 0


# First strategy - only works when rocket is to the left and below the target,
# and in the first quadrant
def engine_dir_tan(t, state, target):
    x_pos, y_pos, vx, vy = state
    x_t, y_t = target
    if y_pos < 20:
        return -np.pi / 2
    else:
        angle = np.tan((y_t - y_pos) / (x_t - x_pos))
        return angle + np.pi


# Second strategy - simple implementation, just steer rocket continuously toward the target.
# Corrected the calculation of the angle from 1st implementation, using arctan2 instead of tan.
# Works in all quadrants now.
def engine_dir_arctan(t, state, target):
    x_pos, y_pos, vx, vy = state

    x_t, y_t = target
```

```
    if y_pos < 20:
        return -np.pi / 2
    else:
        angle = np.arctan2((y_t - y_pos), (x_t - x_pos))
        return angle + np.pi


# The third strategy - adjustment based on the distance between the rocket and its target
# The distance is multiplied by 0.05, which is a magic number that was deduced through
# trial and error
def engine_dir_corrected(t, state, target):
    x_pos, y_pos, vx, vy = state

    x_t, y_t = target
    dx = x_t - x_pos
    dy = y_t - y_pos

    distance = np.sqrt(dx**2 + dy**2)  # distance to target

    if y_pos < 20:
        return -np.pi / 2
    else:
        angle = np.arctan2(dy, dx)
        if x_pos < x_t:
            corrected_angle = angle / (
                distance * 0.05
            )  # 0.05 explanation in func desc.
        else:
            corrected_angle = angle + angle / (distance * 0.05)

        return corrected_angle + np.pi


# The final strategy - the difference between the rocket's current direction
# and the ideal direction is used to adjust the direction of the rocket
def engine_dir(t, state, target):
    x_pos, y_pos, vx, vy = state

    x_t, y_t = target
    dx = x_t - x_pos
    dy = y_t - y_pos

    current_dir = np.arctan2(vy, vx)  # direction of rocket

    if y_pos < 20:
        return -np.pi / 2
    else:
        angle = np.arctan2(dy, dx)
        diff = angle - current_dir
        corrected_angle = angle + diff

        return corrected_angle + np.pi


# Function for calculating the velocity of the engine particles
def fuel_velocity(t, state, target, steering_func):
    x_pos, y_pos, vx, vy = state

    if t > t_max:
        return np.array([0, 0])
    else:
```

```python
        direction = steering_func(t, state, target)
        vx = k * np.cos(direction)
        vy = k * np.sin(direction)
        return np.array([vx, vy])


# Function for calculating the external forces and their effect on the rocket
def external_forces(t, v):
    m = mass(t)
    gravity_F = m * np.array([0, -g])  # Gravity vector: x, y direction
    air_res = -c * np.linalg.norm(v) * v  # Air resistance: c||v(t)||v(t)
    return gravity_F + air_res


# Defining the system of equations
def rocket_ODE(t, y, steering_func, target):
    x_pos, y_pos, vx, vy = y

    if (y_pos < 0): return 0

    m = mass(t)
    v = np.array([vx, vy])

    ext_forces = external_forces(t, v)
    engine_forces = mass_der(t) * fuel_velocity(t, y, target, steering_func)
    total_force = ext_forces + engine_forces

    acc = total_force / m

    state_derivatives = [vx, vy, acc[0], acc[1]]
    return state_derivatives  # is derived to [x_pos, y_pos, x_velocity, y_velocity]


# Runge-Kutta solver
def RK4(f, tspan, u0, dt, *args):
    t_vec = np.arange(tspan[0], tspan[1] + 1.0e-14, dt)
    dt_vec = dt * np.ones_like(t_vec)
    if t_vec[-1] < tspan[1]:
        t_vec = np.append(t_vec, tspan[1])
        dt_vec = np.append(dt_vec, t_vec[-1] - t_vec[-2])

    u = np.zeros((len(t_vec), len(u0)))
    u[0, :] = u0
    for i in range(len(t_vec) - 1):
        h = dt_vec[i]
        k1 = np.array(f(t_vec[i], u[i, :], *args))
        k2 = np.array(f(t_vec[i] + 0.5 * h, u[i, :] + 0.5 * h * k1, *args))
        k3 = np.array(f(t_vec[i] + 0.5 * h, u[i, :] + 0.5 * h * k2, *args))
        k4 = np.array(f(t_vec[i + 1], u[i, :] + h * k3, *args))
        u[i + 1, :] = u[i, :] + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return t_vec, u


tt = np.arange(t_span[0], t_span[1], 0.1)

# Target = (80, 60)
sol_norm = solve_ivp(
    rocket_ODE, t_span, initial_state, args=(engine_dir, norm_target), t_eval=tt
)
t_norm, u_norm = RK4(rocket_ODE, t_span, initial_state, 0.1, engine_dir, norm_target
    )

# Target = (-40, 80)
```

```python
sol_neg = solve_ivp(
    rocket_ODE, t_span, initial_state, args=(engine_dir, neg_target), t_eval=tt
)
t_neg, u_neg = RK4(rocket_ODE, t_span, initial_state, 0.1, engine_dir, neg_target)

# Target = (20, 80)
sol_high = solve_ivp(
    rocket_ODE, t_span, initial_state, args=(engine_dir, high_target), t_eval=tt
)
t_high, u_high = RK4(rocket_ODE, t_span, initial_state, 0.1, engine_dir, high_target
    )

# Target = (80, 60), with first iteration of steering function, to compare (arctan)
sol_arctan = solve_ivp(
    rocket_ODE, t_span, initial_state, args=(engine_dir_arctan, norm_target), t_eval
        =tt
)
t_arctan, u_arctan = RK4(
    rocket_ODE, t_span, initial_state, 0.1, engine_dir_arctan, norm_target
)


# --------- X-Y TO TIME AXIS ---------
# plt.plot(sol.t, sol.y[0], label="x position")
# plt.plot(sol.t, sol.y[1], label="y position")
# plt.title("Rocket steering")
# plt.xlabel("Position x ()")
# plt.ylabel("Position y ()")
# plt.grid()
# plt.legend()
# plt.show()


# --------- X-Y POS AXES ---------
plt.plot(sol_norm.y[0], sol_norm.y[1], "r", label="Rocket normal")
plt.plot(norm_target[0], norm_target[1], "ro", label="Target normal (80, 60)")

plt.plot(sol_neg.y[0], sol_neg.y[1], "g", label="Rocket neg")
plt.plot(neg_target[0], neg_target[1], "go", label="Target neg (-40, 80)")

plt.plot(sol_high.y[0], sol_high.y[1], "y", label="Rocket high")
plt.plot(high_target[0], high_target[1], "yo", label="Target high(-40, 90)")

plt.plot(sol_arctan.y[0], sol_arctan.y[1], "b", label="Rocket old arctan")

plt.title("Rocket steering - xy pos")
plt.xlabel("Position x")
plt.ylabel("Position y")
plt.grid()
plt.prism()
plt.legend()
plt.show()

# --------- RUNGEKUTTA ---------
plt.plot(u_norm[:, 0], u_norm[:, 1], "r", label="Rocket RK normal")
plt.plot(norm_target[0], norm_target[1], "ro", label="Target normal (80, 60)")

plt.plot(u_neg[:, 0], u_neg[:, 1], "g", label="Rocket RK neg")
plt.plot(neg_target[0], neg_target[1], "go", label="Target neg (-40, 80)")

plt.plot(u_high[:, 0], u_high[:, 1], "y", label="Rocket RK high")
plt.plot(high_target[0], high_target[1], "yo", label="Target (-40, 90)")

```

```
plt.plot(u_arctan[:, 0], u_arctan[:, 1], "b", label="Rocket old arctan")

plt.title("Rocket steering - xy pos RUNGEKUTTA")
plt.xlabel("Position x")
plt.ylabel("Position y")
plt.grid()
plt.legend()
plt.show()
```