

UNIVERZA V LJUBLJANI

Fakulteta za matematiko in fiziko

Finančni praktikum

Največja konveksna množica znotraj
konveksne množice

Avtorji:

Jure Sternad

Rok Rozman

Jaša Pozne

Mentorja:

prof. dr. Sergio CABELLO

doc. dr. Janoš VIDALI

Ljubljana, 9. januar 2022

Kazalo

1	Navodilo	2
2	Opis problema	2
3	Definiranje konveksne množice Q	2
4	Eksperimenti	3
4.1	Enakostraničen trikotnik	4
4.1.1	Enakostraničen trikotnik - brez rotacij	4
4.1.2	Enakostraničen trikotnik - z rotacijami	5
4.2	Kvadrat	6
4.2.1	Kvadrat - brez rotacij	6
4.2.2	Kvadrat - z rotacijami	6
4.2.3	Primerjava eksperimentov brez in z rotacijami	7
4.3	Krog	8

1 Navodilo

Če imamo podana konveksna mnogokotnika P in Q v koordinatni ravnini, potem je problem odločanja ali se P lahko preslika v Q linearen program (izvedljivosti). Poleg tega je problem odločanja za koliko lahko P največ povečamo, da je lahko v Q , tudi linearen program. V primeru, da je P disk, je to tudi linearen program.

2 Opis problema

Naša naloga je, da naredimo eksperimente, v katerih bomo poiskali največje možne kvadrate, diske, enakostranične trikotnike ..., ki jih lahko preslikamo tako, da so znotraj danega konveksnega mnogokotnika. Eksperimente bomo reševali s pomočjo linearnega programiranja. Poleg tega bomo ločili primere, ko P lahko rotiramo; v tem primeru bomo ločili več različnih rotacij. Za reševanje problema bomo uporabili programski jezik Sage.

Za lažje razumevanje so spodaj navedene še definicije.

Definicija 1. *Konveksen poligon P je tak poligon, za katerega velja, da pri poljubni izbiri dveh točk p in q iz poligona P , daljica pq , ki povezuje omenjeni točki v celoti leži v poligonu P .*

Definicija 2. *Translacije so preslikave oblike $\tau(\vec{x}) = \vec{x} + \vec{a}$ za nek $a \in \mathbb{R}$.*

Definicija 3. *Rotacije so preslikave oblike $\tau(\vec{x}) = R_\phi \vec{x} + \vec{a}$ za nek $\phi \in (0, 2\pi)$ in $\vec{a} \in \mathbb{R}$. Takšna preslikava ustreza rotaciji za kot ϕ okoli točke v ravnini, ki je določena z enačbo $\tau(\vec{x}) = x$.*

3 Definiranje konveksne množice Q

Projekta smo se lotili tako, da smo najprej s pomočjo linearnega programiranja definirali konveksno množico Q s predpisom

$$\begin{aligned} a_i x + b_i y &\leq c_i \\ i &= 1, \dots, n \end{aligned}$$

n predstavlja število pogojev s katerimi definiramo množico Q .

4 Eksperimenti

Kot navodilo zahteva, smo izvedeli eksperimente za tri like - krog, enakostraničen trikotnik in kvadrat. Pri slednjih smo upoštevali možnost rotacije likov.

Tako je bil cilj naših eksperimentov najti tak že v naprej natančno določen lik, da bo le-ta znotraj množice Q imel kar se da veliko ploščino.

Pri eksperimentih na enakostraničnem trikotniku in kvadratu (brez rotacij), smo uporabili algoritem

```
# Nastavimo p za linearen program v katerem iscemo največjo skalo.
p = MixedIntegerLinearProgram(maximization=True)
p.set_objective(p['k'])
# Z dvema for zankama se sprehodimo po tockah zrcaljenega lika
# in polravninah lika v katerega zrcalimo.
for xi, yi in tocke:
    for ai, bi, ci in zip(a, b, c):
        # Dodamo pogoj v p.
        p.add_constraint(ai * (p['k'] * xi + p['x']) +
                        bi * (p['k'] * yi + p['y']) <= ci)
p.solve()
```

ki za vhodne podatke prejme podatke v obliki $\vec{a} = [a_1, a_2, \dots, a_n]$ za a,b,c in krajišča likov $tocke = [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ ter vrne skalar k , ki predstavlja količino za koliko se bo lik povečal/zmanjšal, in koordinati x in y , ki predstavljata preslikavo lika v množico Q .

V primeru, ko smo pri eksperimentih upoštevali še rotacije smo uporabili sledeč algoritem.

```
# Nastavimo p za linearen program v katerem iscemo največjo skalo.
p = MixedIntegerLinearProgram(maximization=True)

# S kot in t belezimo največjo skalo in kot
# pri katerem se ta zgodi.
kot = None
t = 0
# Shranimo si stevilo tock in definiramo nov seznam tocke2.
dolzina = len(tocke)
tocke2=list(range(dolzina))

p.set_objective(p['k'])

# S for zanko se sprehodimo po kotih od 1 do 360 stopinj.
for j in range(1,360):
    # Pri vsakem kotu si shranimo nova oglišca zarotiranega lika
    # v tocke2.
    for k in range(len(tocke)):
        xii = (cos(j*pi/180)*tocke[k][0] - sin(j*pi/180)*tocke[k][1])
```

```

        yii = (sin(j*pi/180)*tocke[k][0]+cos(j*pi/180)*tocke[k][1])
        tocke2[k] =[xii,yii]
        # Za vsako točko dodamo pogoje.
    for xi, yi in tocke2:
        for ai, bi, ci in zip(a, b, c):
            p.add_constraint(ai * (p['k'] * xi + p['x']) +
                             bi * (p['k'] * yi + p['y']) <= ci)
    r = p.solve()
    # Če naletimo na vecjo skalo, si jo shranimo skupaj s kotom
    # in oglisci prezrcaljenega lika.
    if r > t:
        t = r
        kot = j
        k, x, y = p.get_values(p['k']), p.get_values(p['x']),
        p.get_values(p['y'])
        nove_tocke=[]
        for tocka in tocke2:
            tocka[0] = (tocka[0]*k+x).numerical_approx()
            tocka[1] = (tocka[1]*k+y).numerical_approx()
            nove_tocke.append(tocka)
        p.remove_constraints(range(p.number_of_constraints()))
    print(t, kot, k, x, y, nove_tocke)

```

Ta poleg vseh že omenjenih lastnosti prvega algoritma hkrati upošteva še vse možne rotacije.

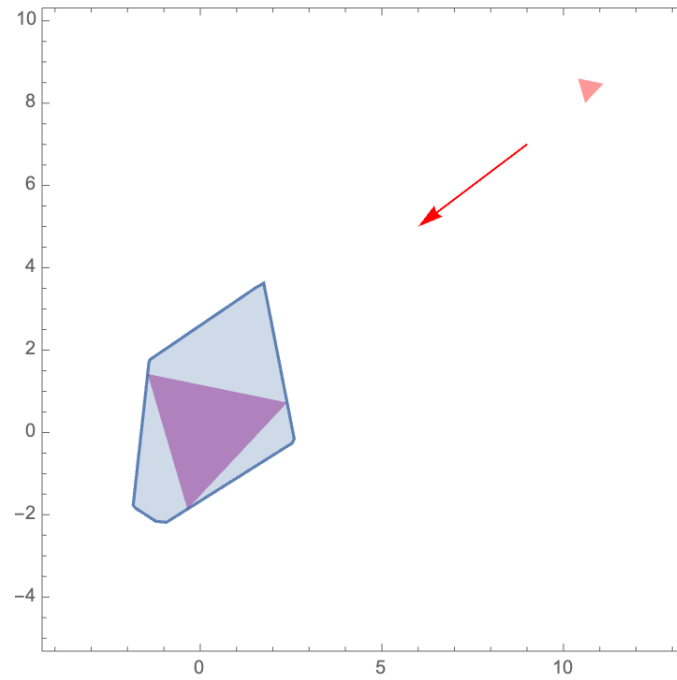
4.1 Enakostraničen trikotnik

4.1.1 Enakostraničen trikotnik - brez rotacij

Prvi eksperiment smo opravili za enakostranični trikotnik brez rotacij. Tako potrebne pogoje kot točke smo si izmislili.

Algoritem je poiskal največji možen trikotnik znotraj konveksne množice Q tako, da je vsa tri oglišča enakostraničnega trikotnika preslikal v Q za (x, y) , pri tem pa celoten lik povečal/zmanjšal za skalar k .

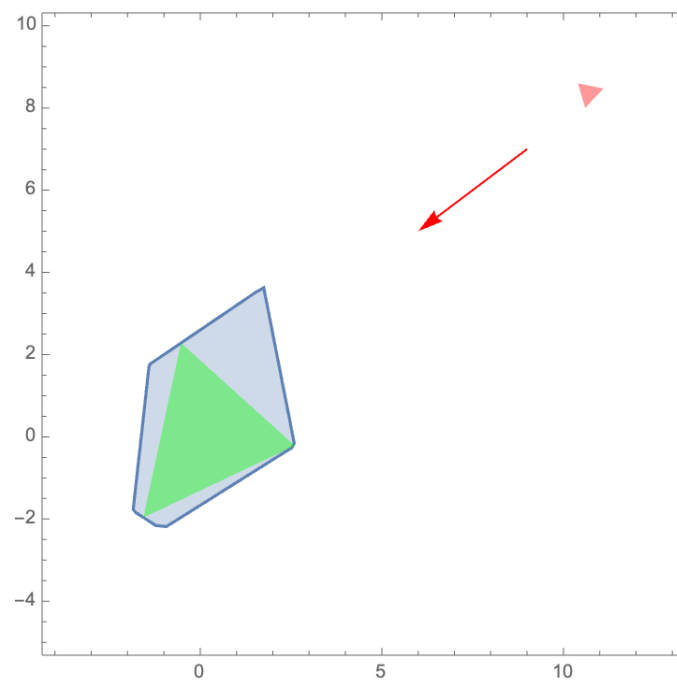
Vhodni podatki prvega eksperimenta so bili $\vec{a} = [-8, 9, -4, -3, 5]$, $\vec{b} = [1, 2, -7, 5, -9]$, $\vec{c} = [13, 23, 20, 13, 15]$ in $tocke = [(10.4, 8.6), (10.6, 8), (11.1019615, 8.4732052)]$.



4.1.2 Enakostraničen trikotnik - z rotacijami

Drugi eksperiment smo ponovno izvedli za enakostraničen trikotnik, le da tokrat z rotacijami. Začetne pogoje smo si ponovno izmislili ter z že prej opisanim algoritmom iskali optimalno rešitev.

Vhodni podatki drugega eksperimenta so bili $\vec{a} = [-8, 9, -4, -3, 5]$, $\vec{b} = [1, 2, -7, 5, -9]$, $\vec{c} = [13, 23, 20, 13, 15]$ in $tocke = [(10.4, 8.6), (10.6, 8), (11.1019615, 8.4732052)]$.

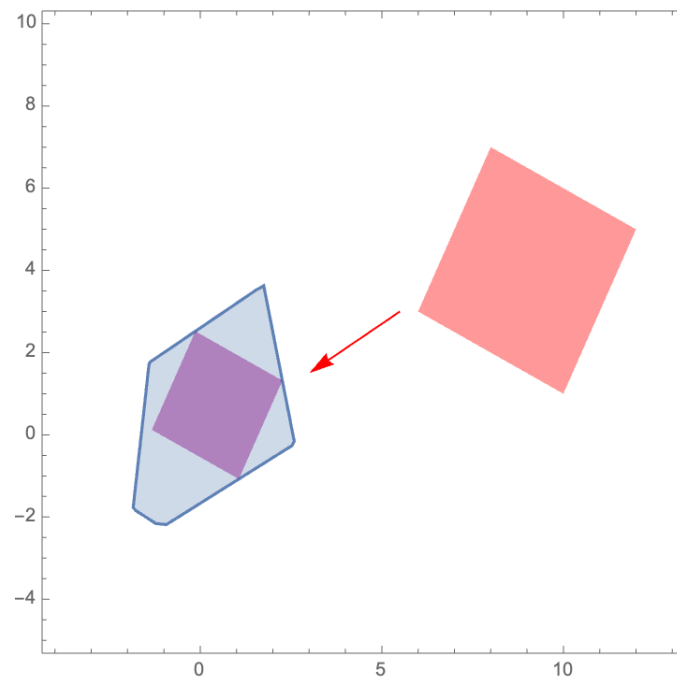


4.2 Kvadrat

Tudi pri eksperimentih s kvadratom smo izvedli dva preizkusa - brez in z rotacijami. Ker je linearni program enak tako za enakostraničen trikotnik kot kvadrat, smo tudi tukaj uporabili že opisana algoritma.

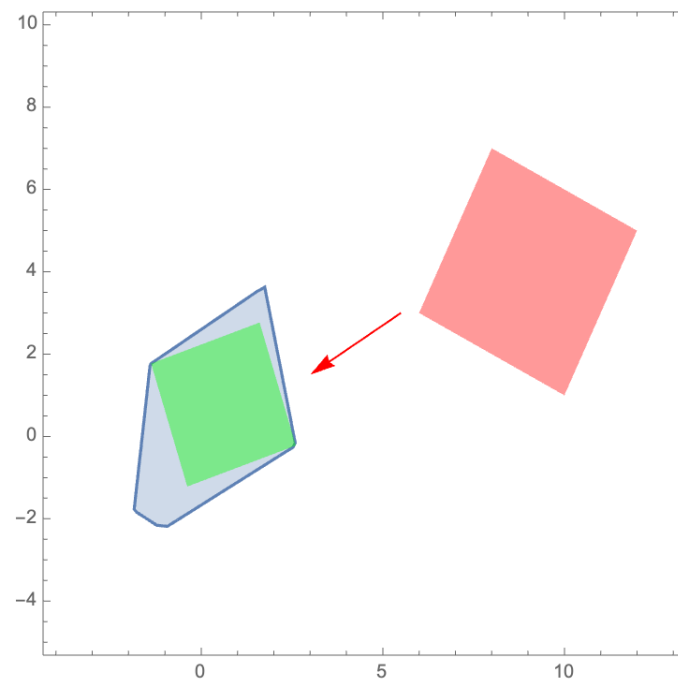
4.2.1 Kvadrat - brez rotacij

Vhodni podatki tako že tretjega eksperimenta so bili $\vec{a} = [-8, 9, -4, -3, 5]$, $\vec{b} = [1, 2, -7, 5, -9]$, $\vec{c} = [13, 23, 20, 13, 15]$ in $tocke = [(10, 1), (6, 3), (8, 7), (12, 5)]$.



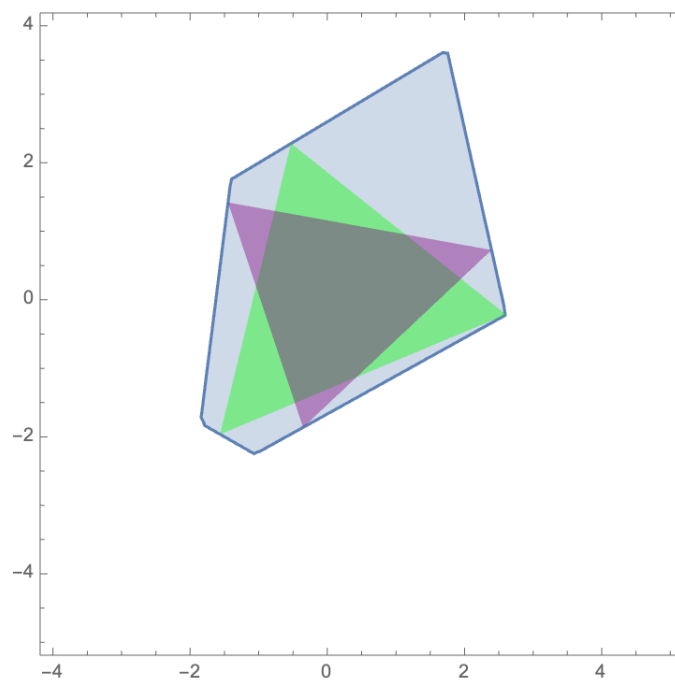
4.2.2 Kvadrat - z rotacijami

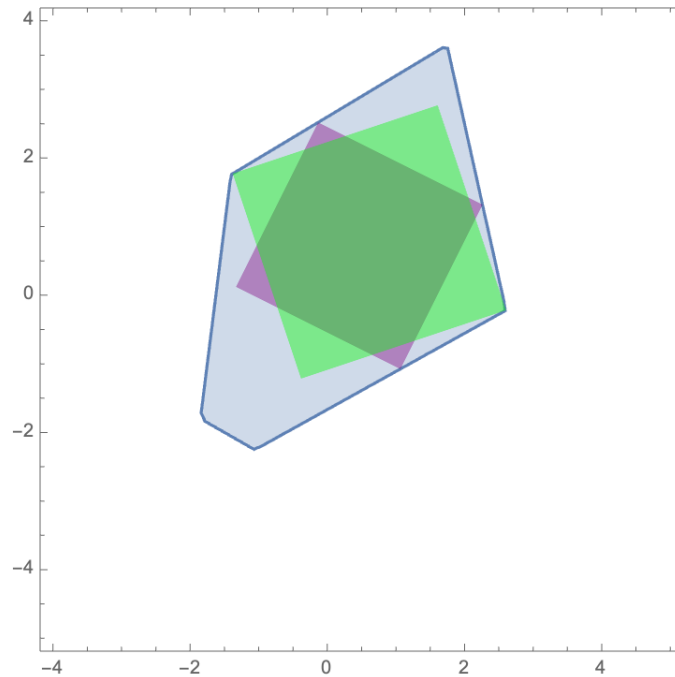
Vhodni podatki četrtega eksperimenta pa $\vec{a} = [-8, 9, -4, -3, 5]$, $\vec{b} = [1, 2, -7, 5, -9]$, $\vec{c} = [13, 23, 20, 13, 15]$ in $tocke = [(10, 1), (6, 3), (8, 7), (12, 5)]$.



4.2.3 Primerjava eksperimentov brez in z rotacijami

Če ponovimo oba eksperimenta z istimi vhodnimi podatki hitro opazimo pričakovano vidno razliko med ploščinama likov. Tako lahko logično sklepamo, da je metoda z rotacijami učinkovitejša.





4.3 Krog

Za eksperiment s krogom smo potrebovali nov algoritem, saj ima krog neskončno število točk, ki jih je potrebno preveriti, posledično pa algoritem, ki je deloval v primeru enakostraničnega trikotnika in kvadrata v tem primeru ne deluje. Ideja algoritma je krožnico razdeliti na 360 točk ter potem za vsako preveriti ali ustreza predpisanemu linearnemu pogoju.

```
# krog = [sredisce, radij]

# Nastavimo p za linearni program.
p = MixedIntegerLinearProgram(maximization=True)
p.set_objective(p['k'])

# Z x0 in y0 oznacimo sredisce kroga.
x0,y0 = krog[0]
radij = krog[1]
# Sredisce kroga postavimo v koordinatno izhodišce in nato točko,
# ki je na poziciji (sredisce,0) zarotiramo za
# 360 stopinj okoli izhodišca,
# za vsako stopinjo posebej.
# Na ta način dobimo dober približek za krog.
tocke=[(0,0),radij]
for j in range(1,360):
    xi = tocke[1]*cos(j*pi/180)
    yi = tocke[1]*sin(j*pi/180)
    # Pri vsaki rotaciji tocke dodamo v linearni program pogoj,
    # ki vključuje to točko.
```

```

    for ai, bi, ci in zip(a, b, c):
        p.add_constraint(ai * (p['k'] * xi + p['x']) +
            bi * (p['k'] * yi + p['y']) <= ci)
p.solve()

```

```

k, x, y = p.get_values(p['k']), p.get_values(p['x']),
    p.get_values(p['y'])
nov_radij = radij * k
premik_sredisca = (x-x0, y-y0)
novo_sredisce = (x, y)
print(k, novo_sredisce, nov_radij)

```

Vhodni podatki petega in tako zadnjega eksperimenta so bili $\vec{a} = [-8, 9, -4, -3, 5]$, $\vec{b} = [1, 2, -7, 5, -9]$, $\vec{c} = [13, 23, 20, 13, 15]$ in $krog = [(7, 7), 3]$.

