demo

May 2, 2020

[1]:
```
import numpy as np
import numpy.linalg as la
from scipy.spatial import ConvexHull
from scipy.spatial.distance import pdist
from scipy.integrate import solve_ivp
from scipy.interpolate import interp1d
from math import pi
import matplotlib.pyplot as plt
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
from IPython.core.interactiveshell import InteractiveShell

# InteractiveShell.ast_node_interactivity = "all"
# %matplotlib notebook
%matplotlib inline
# mpl.rc('text', usetex=True)
# mpl.rc('font', size=12)
```

[354]:
```
class Trajectory():
    def __init__(self, lp, t0, theta, ms):
        self.t = np.array([t0])
        psi0 = np.array([np.cos(theta), np.sin(theta)])
        self.theta = theta
        self.y = lp.support_X0(psi0)[1]
        self.psi = np.array(psi0).reshape(1, -1)
        self.t_term = None
        self.y_term = None
        self.is_internal = False
        self.is_terminal = False
        self.lp = lp
        self.max_step = ms
        self.is_optimal = False

    def extend(self, t_max_new):
        if (self.is_internal):
            return False
```

```
        if (self.is_terminal):
            return True

        self.t_max = t_max_new
        t_new = np.arange(self.t_max, self.t[-1], -self.max_step)[::-1]
        psi_sol = solve_ivp(lp.conj_func,
                        (self.t[-1], self.t_max),
                        self.psi[-1],
                        method='RK45',
                        t_eval=t_new,
                        dense_output=True,
                        max_step=self.max_step)

        psi_new = psi_sol.y
        t_eval_kostyl = np.hstack((np.array([self.t[-1]]), t_new))
        y_sol = solve_ivp(lambda t, y: lp.func(t, y, psi_sol.sol),
                        (self.t[-1], self.t_max),
                        self.y[-1],
                        method='RK45',
                        t_eval=t_eval_kostyl,
                        events=(lp.reached_target, lp.back_inside),
                        max_step=self.max_step)

        y_new = y_sol.y[:, 1:]

        self.t = np.hstack((self.t, t_new))
        self.y = np.vstack((self.y, y_new.T))
        self.psi = np.vstack((self.psi, psi_new.T))

        if (y_sol.t_events[1].size > 0):
            self.is_internal = True
            self.t = self.t[:self.y.shape[0]]
            self.psi = self.psi[:self.y.shape[0]]
            return False

        if (y_sol.t_events[0].size > 0):
            self.is_terminal = True
            self.t_term = y_sol.t_events[0][0]
            self.y_term = y_sol.y_events[0][0]
            self.t = self.t[:self.y.shape[0]]
            self.psi = self.psi[:self.y.shape[0] + 1]
            self.y = np.vstack((self.y, self.y_term))
            self.t = np.hstack((self.t, self.t_term))
            return True

        return False
```

```
    def plot(self, fig, ax, s, t_max=np.inf):

        idx = self.t <= t_max
        idx = idx[:self.y.shape[0]]
        col, zor = ('r', 2) if self.is_optimal else ('g', 1)

        sy, sx = s
        j = int(sy[1]) - 1
        if (sy[0] == 'u'):
            u = np.empty(self.psi.shape)
            for i in range(u.shape[0]):
                B = lp.B(self.t[i])
                p = self.psi[i]
                while (la.norm(p @ B) < 1e-3):
                    B = B + np.random.randn(B.shape[0], B.shape[1]) * 3e-2
                u[i] = lp.support_P(p @ B)[1]
            d2 = u[idx, j]
            yl = '$u_' + str(j + 1) + '$'
        elif (sy[0] == 'x'):
            d2 = self.y[idx, j]
            yl = '$x_' + str(j + 1) + '$'
        elif (sy[0] == 'p'):
            d2 = self.psi[idx, j]
            yl = '$\psi_' + str(j + 1) + '$'
        else:
            print("ERROR!")

        if (sx[0] == 'u'):
            d1 = u[idx, 0]
            xl = '$u_1$'
        elif (sx[0] == 'x'):
            d1 = self.y[idx, 0]
            xl = '$x_1$'
        elif (sx[0] == 'p'):
            d1 = self.psi[idx, 0]
            xl = '$\psi_1$'
        elif (sx[0] == 't'):
            d1 = self.t[idx]
            xl = '$t$'
        else:
            print("ERROR!")

        if (sx[0] == 'u'):
            ax.scatter(d1, d2, 10, c=col, zorder=zor)
        else:
            ax.plot(d1, d2, col, zorder=zor)
        ax.plot(d1, d2, col, zorder=zor)
```

```
        if (self.is_optimal and sx[0] == 'x' and sy[0] == 'x'):
            ax.scatter(self.y_term[0], self.y_term[1], 15, c='r', zorder=3)
        ax.set_xlabel(xl)
        ax.set_ylabel(yl)
```

[383]:
```
class LinearProblem():
    def __init__(self, params):

        # require A(t), B(t), f(t), t0, a, b, c, d, r, Q, alpha, beta, max_t1,␣
↪max_ngrid

        for key, value in params.items():
            setattr(self, key, value)

        self.eps_reg = 2e-2
        self.hull_X1 = ConvexHull(self.Q, qhull_options='QJ')
        self.reg_X1()
        self.k1 = np.sqrt(self.alpha / self.c)
        self.k2 = np.sqrt(self.beta / self.c)
        self.s = self.d - self.a + self.b
        self.T = np.array([[1 / self.k1, 0],
                           [0, 1 / self.k2]])
        self.center_P = np.array([[self.a],
                                  [self.b]])
        self.ngrid = 20
        self.dt = 0.5
        self.clar_bnd = None
        self.clar_step = None

        self.theta_grid = np.arange(0.0, 2 * pi, 2 * pi / self.ngrid)
        self.traj_list = [Trajectory(self, self.t0, self.theta_grid[i], self.
↪eps_reg)
                          for i in range(self.ngrid)]
        self.found = False

        self.opt_traj = None

    def reg_X1(self):
        S = self.hull_X1.volume
        d = np.max(pdist(self.Q))
        factor = np.max(la.norm(self.Q, axis=1))
        if (d < self.eps_reg):
            self.Q = 2 * (np.random.rand(10, 2) - 0.5) * factor * self.eps_reg␣
↪+ self.hull_X1.points[0]
            self.hull_X1 = ConvexHull(self.Q)
        elif (S / d < self.eps_reg):
```

```
            p = np.tile(self.Q, (10, 1))
            self.Q = p + 2 * (np.random.rand(40, 2) - 0.5) * factor * self.
↪eps_reg
            self.hull_X1 = ConvexHull(self.Q)


    def support_X0(self, l):
        l = l.reshape(-1, 2)
        rho = self.r * la.norm(l, axis=1)
        vec = l * self.r / np.tile(la.norm(l, axis=1), (2, 1)).T
        return (rho, vec)

    def support_X1(self, l):
        l = l.reshape(-1, 2)
        prods = (self.Q @ l.T)
        rho = np.max(prods, axis=0)
        vec = self.Q[np.argmax(prods, axis=0)]
        return (rho, vec)

    def support_P(self, l):

        q = 4

        ll= l / la.norm(l)
        if (self.k1 ** 2 + self.k2 ** 2 - self.s ** 2 < 0 or
                self.k1 * l[0] + self.k2 * l[1] <= self.s * la.norm(l)):
            vec = ll
            vec_reg = vec
        else:
            norm_len_sq = self.k1 ** 2 + self.k2 ** 2

            z1_pos = (self.k1 * self.s + self.k2 *
                      np.sqrt(norm_len_sq - self.s ** 2)) / norm_len_sq
            z2_pos = (self.k2 * self.s - self.k1 *
                      np.sqrt(norm_len_sq - self.s ** 2)) / norm_len_sq
            z1_neg = (self.k1 * self.s - self.k2 *
                      np.sqrt(norm_len_sq - self.s ** 2)) / norm_len_sq
            z2_neg = (self.k2 * self.s + self.k1 *
                      np.sqrt(norm_len_sq - self.s ** 2)) / norm_len_sq

            points = np.array([[z1_pos, z2_pos],
                               [z1_neg, z2_neg]])
            z_mid = 0.5 * (points[0] + points[1])
            vec = points[np.argmax(l @ points.T)]

            norm = np.array([self.k1, self.k2])
            coeff = (ll.dot(norm) - self.s) / norm_len_sq
```

```
            vec_reg = ll - coeff * norm
            vec_reg = q * vec_reg + (1 - q) * z_mid

            if (la.norm(vec_reg) > 1):
                vec_reg = vec

        return (vec.dot(l @ la.inv(self.T) + l.dot(self.center_P)),
                (la.inv(self.T) @ vec_reg.reshape(-1, 1) + self.center_P).
↪flatten())

    def func(self, t, y, psi):
        B = self.B(t)
        p = psi(t)
        while (la.norm(p @ B) < 1e-3):
            B = B + np.random.randn(B.shape[0], B.shape[1]) * 1e-2

        u = self.support_P(p @ B)[1]
        return y @ self.A(t).T + u @ self.B(t).T + self.f(t)

    def conj_func(self, t, y):
        return y @ -self.A(t)

    def reached_target(self, t, y):
        dirs = self.hull_X1.equations[:, :-1]
        biases = self.hull_X1.equations[:, -1]
        dists = y @ dirs.T + biases
        return np.max(dists)

    def back_inside(self, t, y):
        return np.sum(y ** 2) - self.r ** 2 * 0.9

    reached_target.terminal = True
    back_inside.terminal = True

    def solve(self):

        term_traj_list = []

        while (not self.found):
            t_max = self.t0 + self.dt

            if (t_max + self.dt > self.max_t1 and 2 * self.ngrid > self.
↪max_ngrid):
                print("Seems like not reachable, mm?")
                return

            if (t_max + self.dt <= self.max_t1):
```

```
                t_max += self.dt

                self.dt *= 2
                for traj in self.traj_list:
                    res = traj.extend(t_max)
                    if (res):
                        term_traj_list.append(traj)
                    self.found = res or self.found

            if ((not self.found) and 2 * self.ngrid <= self.max_ngrid):
                self.ngrid *= 2
                theta_densed = self.theta_grid + (2 * pi) / self.ngrid
                self.theta_grid = np.arange(0.0, 2 * pi, 2 * pi / self.ngrid)
                traj_densed = [Trajectory(self, self.t0, theta_densed[i], self.
↪eps_reg)
                     for i in range(self.ngrid // 2)]

                for traj in traj_densed:
                    res = traj.extend(t_max)
                    if (res):
                        term_traj_list.append(traj)
                    self.found = res or self.found

                old_traj = self.traj_list
                self.traj_list = [None] * self.ngrid
                for i in range(self.ngrid // 2):
                    self.traj_list[2 * i] = old_traj[i]
                    self.traj_list[2 * i + 1] = traj_densed[i]

        self.opt_traj = min(term_traj_list, key=lambda traj: traj.t[-1])
        self.opt_idx = self.traj_list.index(self.opt_traj)
        self.opt_traj.is_optimal=True
        print('OK!')
        print('Suboptimal time: ' + str(self.opt_traj.t[-1]))

    def clarify_opt(self, criterion, eps, max_iters=5):

        res = self.check_trans(criterion)
        while (res > eps):

            max_iters -= 1
            t_opt = self.opt_traj.t_term
            self.opt_traj.is_optimal = False

            if (self.clar_bnd is None):
                self.clar_bnd = 4 * pi / self.ngrid
            else:
```

```
                self.clar_bnd /= 2

            if (self.clar_step is None):
                self.clar_step = self.eps_reg / 1.5
            else:
                self.clar_step /= 1.5

            theta_cl = np.linspace(-self.clar_bnd, self.clar_bnd, 9) + self.
↪opt_traj.theta
            for i in range(9):
                if (i % 2 == 0):
                    idx = (self.opt_idx + i - 2) % len(self.traj_list)
                    self.traj_list[idx] = Trajectory(self, self.t0,␣
↪theta_cl[i], self.clar_step)
                    self.traj_list[idx].extend(t_opt)
                else:
                    idx = (self.opt_idx + i - 2) % len(self.traj_list)
                    traj = Trajectory(self, self.t0, theta_cl[i], self.
↪clar_step)
                    traj.extend(t_opt)
                    self.traj_list.insert(idx, traj)

            tl = [t for t in self.traj_list if t.is_terminal]
            self.opt_traj = min(tl, key=lambda traj: traj.t_term)
            self.opt_idx = self.traj_list.index(self.opt_traj)
            self.opt_traj.is_optimal = True

            res = self.check_trans(criterion)
            print(res)
            if (max_iters <= 0):
                break

    def check_trans(self, metric):
        psi = self.opt_traj.psi[-1]
        sup, x_trans = self.support_X1(-psi)
        x = self.opt_traj.y[-1]
        if (metric == 'distance'):
            return la.norm(x - x_trans[0])
        elif (metric == 'cosine'):
            n = np.mean([la.norm(x), la.norm(x_trans[0])]) * la.norm(psi)
            return (psi.dot(x) + sup[0]) / n

    def plot_sets(self, fig, ax):
        # plotting X0
        N = 100
        phi = np.arange(N) / (2 * pi)
        x = self.r * np.cos(phi)
```

```
        y = self.r * np.sin(phi)
        ax.fill(x, y, 'y')

        # plotting X1
        x = self.Q[self.hull_X1.vertices].T[0]
        y = self.Q[self.hull_X1.vertices].T[1]
        ax.fill(x, y, 'b')

        return (fig, ax)

    def plot_trans(self, fig, ax):

        psi = self.opt_traj.psi[-1]
        rho, vec = self.support_X1(-psi)
        vec = self.opt_traj.y[-1]

        if (psi[0] > psi[1]):
            w = np.array([-psi[1] / psi[0], 1])
        else:
            w = np.array([1, -psi[0] / psi[1]])
        w = w / la.norm(w)

        x = np.array([vec[0] - w[0], vec[0] + w[0]])
        y = np.array([vec[1] - w[1], vec[1] + w[1]])
        ax.plot(x, y, 'k', zorder=5)

        dirs = self.hull_X1.equations[:, :-1]
        biases = self.hull_X1.equations[:, -1]
        dists = self.opt_traj.y[-1] @ dirs.T + biases
        w = dirs[np.argmin(np.abs(dists))] / 2

        x = np.array([vec[0], vec[0] + w[0]])
        y = np.array([vec[1], vec[1] + w[1]])
        ax.plot(x, y, 'k', zorder=5)

        psi = self.opt_traj.psi[0]
        rho, vec = self.support_X0(psi)
        vec = vec[0]

        if (psi[0] > psi[1]):
            w = np.array([-psi[1] / psi[0], 1])
        else:
            w = np.array([1, -psi[0] / psi[1]])
        w = w / la.norm(w)

        x = np.array([vec[0] - w[0], vec[0] + w[0]])
        y = np.array([vec[1] - w[1], vec[1] + w[1]])
```

```
        ax.plot(x, y, 'k', zorder=5)

        x = np.array([vec[0], vec[0] + psi[0] / la.norm(psi) / 2])
        y = np.array([vec[1], vec[1] + psi[1] / la.norm(psi) / 2])
        ax.plot(x, y, 'k', zorder=5)

    def plot(self, fig, ax, s):
        t_max = self.opt_traj.t[-1] if (self.opt_traj is not None) else np.inf
        for traj in self.traj_list:
            traj.plot(fig, ax, s, np.inf)
```

[374]:
```
def A(t):
    return np.array([[3, -2],
                     [2, 1]])
#     return np.zeros((2, 2))
```

[375]:
```
def B(t):
    return np.array([[1, 0],
                     [0, 1]])
```

[376]:
```
def f(t):
    return np.zeros(2)
```

[377]:
```
params = {
    'A': A,
    'B': B,
    'f': f,
    't0': 0.,
    'a': -0.9,
    'b': 0,
    'c': 1.,
    'd': 5.9,
    'r': 1,
    'Q': np.array([[5.4, 1],
                   [5.4, -1],
                   [7, 1],
                   [7, -1]]),
#     'Q': np.array([[-1, 5],
#                    [-1, 3],
#                    [1, 5],
#                    [1, 3]]),
    'alpha': 1.,
    'beta': 1.,
    'max_t1': 4,
    'max_ngrid': 20
}
```

[378]:
```
lp = LinearProblem(params)
```

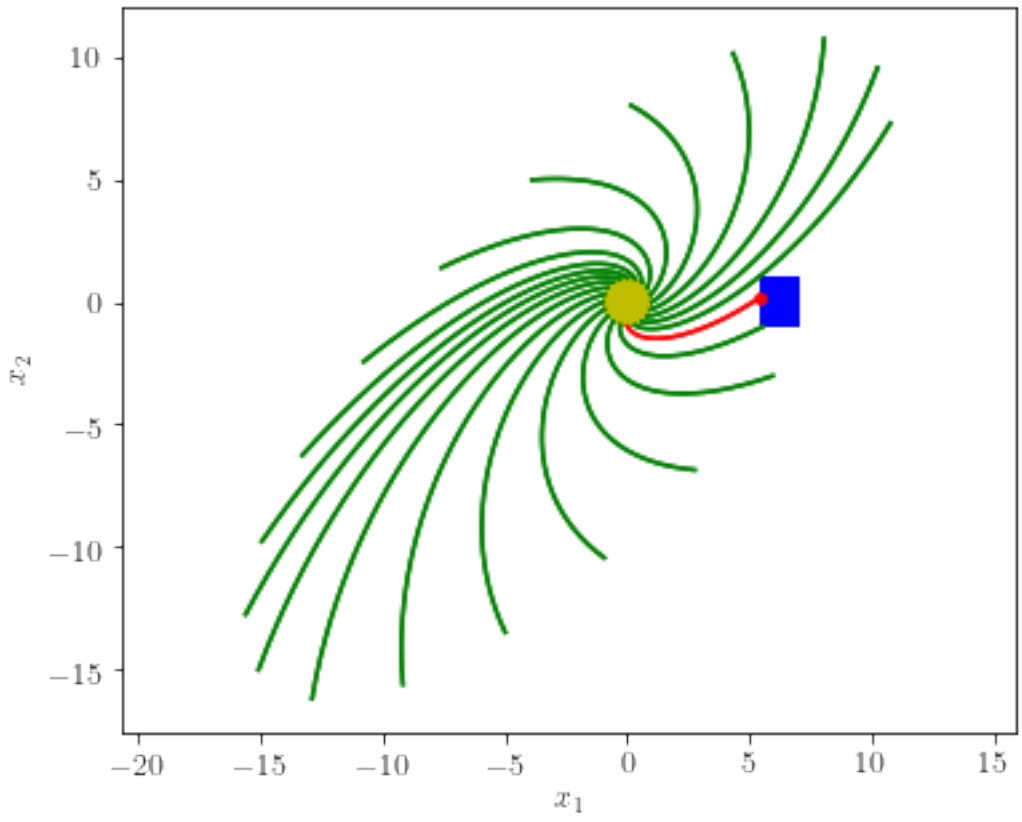
[379]:
```
lp.solve()
```

```
OK!
Suboptimal time: 0.7525540250225108
```

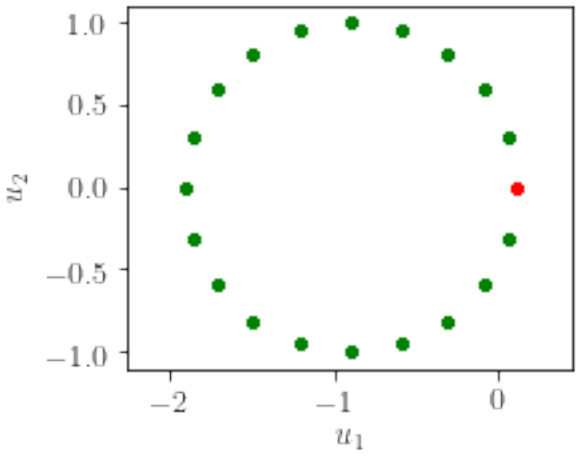
[380]:
```
fig, ax = plt.subplots();
fig.set_size_inches(6, 5)
lp.plot_sets(fig, ax);
lp.plot(fig, ax, ('x2', 'x1'));
# lp.plot_trans(fig, ax)
ax.axis('equal');
# lp.check_trans('cosine')
fig.savefig('report/figures/ex42_x.pdf', bbox_inches='tight')
```

[380]:
```
(<Figure size 432x360 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7ff205a44340>)
```

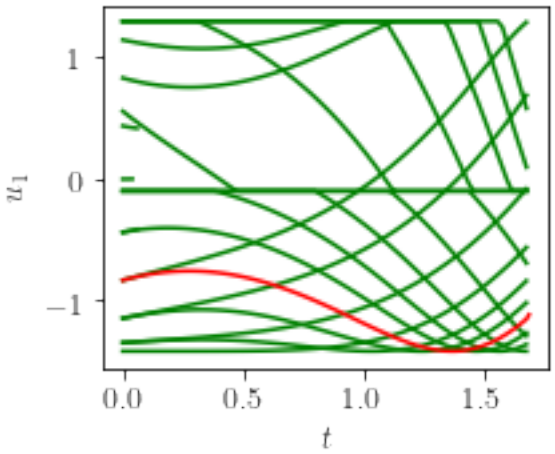
[380]:
```
(-16.902346316459777,
 12.050760047084838,
 -17.538330688997974,
 12.08288156011455)
```



[362]:
```
fig, ax = plt.subplots();
fig.set_size_inches(3, 2.5)
lp.plot(fig, ax, ('u2', 'u1'));
ax.axis('equal');
# fig.savefig('report/figures/ex2_u.pdf', bbox_inches='tight')
```



[79]:
```
fig, ax = plt.subplots();
fig.set_size_inches(3, 2.5)
lp.plot(fig, ax, ('u1', 't'));
fig.savefig('report/figures/ex2_u1t.pdf', bbox_inches='tight')
```



[381]:
```
lp.clarify_opt('cosine', 0.005)
```

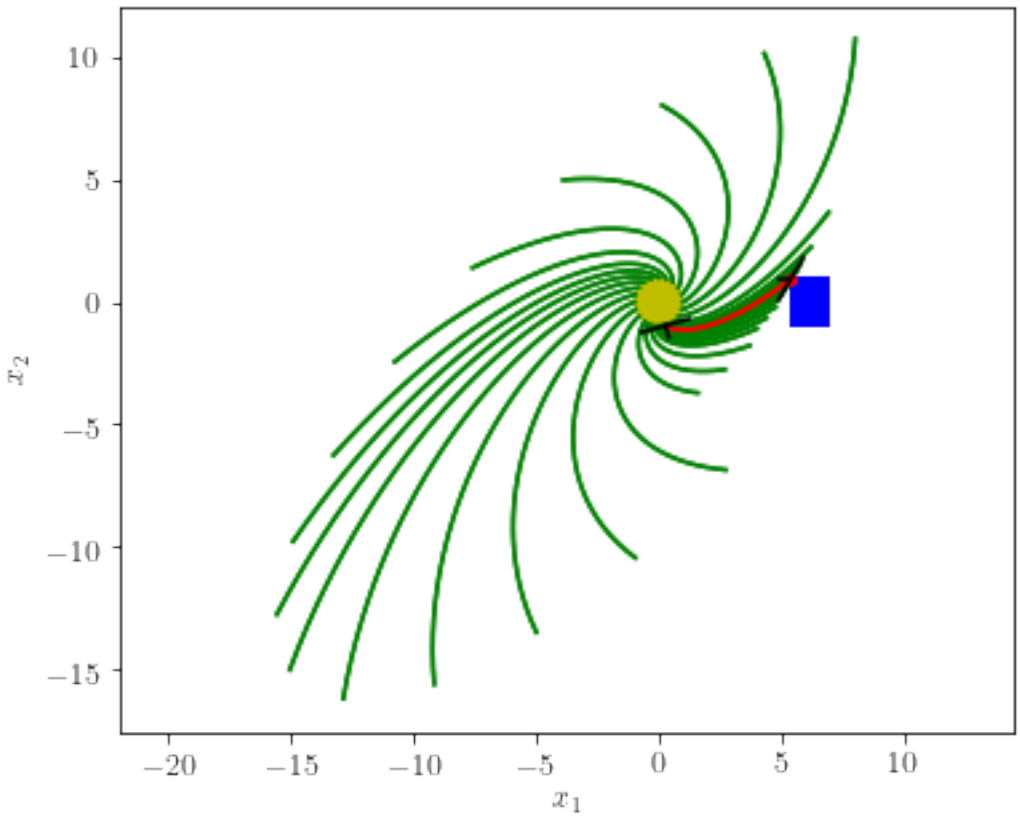
```
0.03672205108034856
0.014625658345814323
0.004575884397517703
```

[382]:
```
fig, ax = plt.subplots();
fig.set_size_inches(6, 5)
lp.plot_sets(fig, ax);
lp.plot(fig, ax, ('x2', 'x1'));
lp.plot_trans(fig, ax)
ax.axis('equal');
lp.check_trans('cosine')
fig.savefig('report/figures/ex2_x_clar.pdf', bbox_inches='tight')
```

[382]:
```
(<Figure size 432x360 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7ff205c0f070>)
```

[382]:
```
(-16.76672842754808, 9.202784379939276, -17.538330688997974, 12.08288156011455)
```

[382]:
```
0.004575884397517703
```



[92]:
```
lp.opt_traj.t_term
```

[92]: 1.6302936086615125

[ ]: