Новосибирский государственный технический университет
Факультет прикладной математики и информатики
Кафедра теоретической и прикладной информатики

Лабораторная работа № 2

«Активная параметрическая идентификация моделей линейных дискретных динамических стохастических систем»

по дисциплине

«Математические методы планирования эксперимента»

Группа: ПММ-61
Студент: Горбунов К. К.
Преподаватель: Чубич В. М.

Новосибирск, 2017 г.

Цель работы

Реализовать процедуру активной параметрической идентификации, экспериментально подтвердить её эффективность.

Порядок выполнения лабораторной работы

1. Изучить соответствующий теоретический материал.
2. Последовательно выполнить все задания к лабораторной работе.
3. Проверить правильность реализации алгоритмов и работоспособность программ.

Задание к лабораторной работе

1. Связать разработанные ранее программные модули оценивания параметров и планирования оптимальных входных сигналов.
2. Для некоторой модели стохастической линейной дискретной системы получить начальные оценки параметров по некоторому произвольному начальному плану эксперимента.
3. При полученных начальных оценках параметров синтезировать оптимальный непрерывный план идентификационного эксперимента, округлить полученный непрерывный план до дискретного.
4. Провести идентификационный эксперимент согласно полученному дискретному оптимальному плану.
5. Сравнить точность оценивания параметров по исходному и оптимальному планам.

Теоретический материал

Основы активной параметрической идентификации

Предоположим, что экспериментатор может произвести ν повторных запусков системы, причем сигнал α_1 он подает на вход системы k_1 раз, сигнал α_2 — k_2 раза и так далее, наконец, сигнал α_q — k_q раз. В этом случае дискретный (точный) нормированный план эксперимента ξ_ν представляет собой совокупность точек $\alpha_1, \alpha_2, \ldots, \alpha_q$, называемых спектром плана, и соответствующих им долей повторных запусков:

$$\xi_\nu = \left\{ \begin{array}{c} \alpha_1, \alpha_2, \ldots, \alpha_q \\ \frac{k_1}{\nu}, \frac{k_2}{\nu}, \ldots, \frac{k_q}{\nu} \end{array} \right\}, \ \alpha_i \in \Omega_\alpha, \ i = 1, 2, \ldots, q.$$

Каждая точка α_i спектра плана представляет собой последовательность импульсов, «развернутую во времени», т.е.

$$\alpha_i^T = U_i^T = \left\{ [u^i(t_0)]^T, [u^i(t_1)]^T, \ldots, [u^i(t_{N-1})]^T \right\}, \ i = 1, 2, \ldots, q.$$

Множество планирования Ω_α определяется ограничениями на условия проведения эксперимента.

Под непрерывным планом ξ понимается совокупность величин

$$\xi = \left\{ \begin{array}{c} \alpha_1, \alpha_2, \ldots, \alpha_q \\ p_1, p_2, \ldots, p_q \end{array} \right\}, \ p_i \geq 0, \ \sum_{i=1}^{q} p_i = 1, \ \alpha_i \in \Omega_\alpha, \ i = 1, 2, \ldots, q.$$

В отличие от дискретного нормированного плана в непрерывном нормированном плане снимается условие рациональности весов p_i.

Из непрерывного плана можно получить дискретный путем его округления.

Если $L(Y_1^N; \Theta)$ — плотность совместного распределения вероятностей по совокупности измерений $Y_1^N = \{y(t_1), y(t_2), \ldots, y(t_N)\}$ при фиксированном значении вектора параметров Θ, то информационная матрица Фишера одноточечного плана определяется выражениями [1]:

$$M(\alpha) = \left\| \underset{Y}{E} \left[\frac{\partial \ln L(Y_1^N;\Theta)}{\partial \theta_i} \frac{\partial \ln L(Y_1^N;\Theta)}{\partial \theta_j} \right] \right\| = \left\| \underset{Y}{E} \left[-\frac{\partial^2 \ln L(Y_1^N;\Theta)}{\partial \theta_i \partial \theta_j} \right] \right\|$$

Тогда нормированная информационная матрица $M(\xi)$ плана ξ определяется соотношением:

$$M(\xi) = \sum_{i=1}^{q} p_i M(\alpha_i),$$

где $M(\alpha_i)$ — информационные матрицы точек спектра плана.

Задача построения оптимального плана эксперимента ξ^* сводится к задаче:

$$\xi^* = \arg\min_{\xi \in \Omega_\xi} X[M(\xi)],$$

где Ω_ξ — область планирования, X — критерий оптимальности. Данную задачу можно решать путем прямой и двойственной процедур.

Если через $Y_{i,j}$ обозначить j-ю реализацию выходного сигнала ($j = 1, 2, \ldots, k_i$), соответствующему i-му входному сигналу U_i ($i = 1, 2, \ldots, q$), то в результате проведения по плану ξ_ν идентификационных экспериментов будет сформировано множество

$$\Xi = \{(U_i, Y_{i,j}),\ j = 1, 2, \ldots, k_i,\ i = 1, 2, \ldots, q\}$$

Уточним структуру $Y_{i,j}$:

$$Y_{i,j} = \left\{[y^{i,j}(t_1)]^T, [y^{i,j}(t_2)]^T, \ldots, [y^{i,j}(t_N)]^T\right\},\ j = 1, 2, \ldots, k_i,\ i = 1, 2, \ldots, q$$

Задача оценивания параметров θ сводится к задаче:

$$\hat{\theta} = \arg\min_{\theta \in \Omega_\theta} \chi(\Xi, \theta),$$

где χ — критерий идентификации.

Алгоритм процедуры активной параметрической идентификации:

1. Взять произвольный (можно случайный) план ξ_0, тогда в результате проведения идентификационных экспериментов по данному плану будет сформировано множество Ξ_0;

2. Получить начальные оценки $\hat{\theta}_0$ параметров модели по плану ξ_0, решая задачу:

$$\hat{\theta}_0 = \arg\min_{\theta \in \Omega_\theta} \chi(\Xi_0, \theta);$$

3. Провести процедуру (прямую или двойственную) оптимального планирования входных сигналов — получить план ξ^*, решая следующую задачу, используя найденные найденные ранее оценки $\hat{\theta}_0$:

$$\xi^* = \arg\min_{\xi \in \Omega_\xi} X[M(\xi), \hat{\theta}_0].$$

Множество, сформированное в результате проведения идентификационных экспериментов по плану ξ^* обозначим как Ξ^*.

4. Получить конечные оценки $\hat{\theta}$ параметров модели по ранее синтезированному оптимальному плану ξ^*:

$$\hat{\theta} = \arg\min_{\theta \in \Omega_\theta} \chi(\Xi^*, \theta);$$

и закончить процесс.

Описание модельной структуры

Модель стохастической динамической линейной дискретной системы в пространстве состояний в виде [1]:

$$\begin{cases} x(t_{k+1}) &= Fx(t_k) + Cu(t_k) + Gw(t_k), \\ y(t_{k+1}) &= Hx(t_{k+1}) + v(t_{k+1}), \qquad\qquad k = 0, \ldots, N-1 \end{cases} \tag{1}$$

Здесь:

$x(t_k)$ – n-вектор состояния;

F – матрица перехода состояния;

$u(t_k)$ – r-вектор управления (входного воздействия);

C – матрица управления;

$w(t_k)$ – p-вектор возмущений;

G – матрица влияния возмущений;

H – матрица наблюдения;

$v(t_{k+1})$ – m-вектор шума измерений;

$y(t_{k+1})$ – m-вектор наблюдений (измерений) отклика;

F, C, G, H — матрицы соответствующих размеров.

Априорные предположения:

- F устойчива;
- пары (F, C) и (F, G) управляемы;
- пара (F, H) — наблюдаема;

- $w(t_k)$ и $v(t_{k+1})$ — случайные векторы, образующие стационарные белые гауссовские последовательности, причем:

$$E[w(t_k)] = 0,\ E[w(t_k)w^T(t_l)] = Q\delta_{k,l}\ ;$$

$$E[v(t_{k+0}) = 0,\ E[v(t_{k+1})v^T(t_{l+1})] = R\delta_{k,l}$$

$$E[v(t_k)w^T(t_k)] = 0,$$

для любых $k, l = 0, 1, \ldots, N-1$ ($\delta_{k,l}$ — символ Кронекера);

- начальное состояние $x(0)$ имеет нормальное распределение с параметрами $\overline{x}(0)$ и $P(0)$ и не коррелирует с $w(t_k)$ и v_{k+1} при любых значениях k.

Будем считать, что подлежащие оцениванию параметры $\theta = (\theta_1, \theta_2, \ldots, \theta_s)$ могут входить в элементы матриц $F, C, G, H, Q, R, P(0)$ и в вектор $\overline{x}(0)$ в различных комбинациях.

Критерий идентификации

В качестве критерия идентификации используется логарифмическая функция правдоподобия. Для одноточечного плана эксперимента она имеет вид [1]:

$$\chi(\theta, U_i, Y_{i,j}) = -\ln L(\theta) = \frac{Nm}{2}\ ln2\pi + \frac{1}{2}\sum_{k=0}^{N-1}\left[[\varepsilon^{i,j}(t_{k+1})]^T B^{-1}(t_{k+1})\varepsilon^{i,j}(t_{k+1})\right] + \\ + \frac{1}{2}\sum_{k=0}^{N-1}\ln\det B^{-1}(t_{k+1}).$$

Критерий идентификации для многоточечного плана имеет вид:

$$\chi(\Xi, \theta) = \sum_{i=1}^{q}\sum_{j=1}^{k_i}\chi(\theta, U_i, Y_{i,j})$$

Градиент критерия идентификации

Выражение для градиента критерия для одноточечного плана имеет вид [1]:

$$\frac{\partial \chi(\theta)}{\partial \theta_l} = \sum_{k=0}^{N-1} \left[\frac{\partial \varepsilon(t_{k+1})}{\partial \theta_l}\right]^T B^{-1}(t_{k+1}) \left[\varepsilon(t_{k+1})\right] -$$

$$-\frac{1}{2}\sum_{k=0}^{N-1} \left[\varepsilon(t_{k+1})\right]^T B^{-1}(t_{k+1}) \frac{\partial B(t_{k+1})}{\partial \theta_i} B^{-1}(t_{k+1}) \varepsilon(t_{k+1}) +$$

$$+\frac{1}{2}\sum_{k=0}^{N-1} \mathrm{Sp}\left[B^{-1}(t_{k+1}) \frac{\partial B(t_{k+1})}{\partial \theta_l}\right],\ l = 1, 2, \ldots, s.$$

Выражение градиента критерия для многоточечного плана можно легко получить по правилу дифференцирования суммы.

Пример активной идентификации

$$\begin{pmatrix} x_1(t_{k+1}) \\ x_2(t_{k+1}) \end{pmatrix} = \begin{pmatrix} \theta_1 & 1 \\ 0 & 0.5 \end{pmatrix} \begin{pmatrix} x_1(t_k) \\ x_2(t_k) \end{pmatrix} + \begin{pmatrix} \theta_2 \\ 1 \end{pmatrix} u(t_k) + \begin{pmatrix} w_1(t_k) \\ w_2(t_k) \end{pmatrix}$$

$$y(t_{k+1}) = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} x_1(t_{k+1}) \\ x_2(t_{k+1}) \end{pmatrix} + \begin{pmatrix} v_1(t_{k+1}) \\ v_2(t_{k+1}) \end{pmatrix}$$

$$\overline{x}(0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix},\ P(0) = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix},\ Q = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix},\ R = 0.1$$

Истинные значения параметров $\theta_{true} = \begin{pmatrix} 0.5 & 0.5 \end{pmatrix}$

Оценивание параметров по произвольному начальному плану

Спектр непрерывного начального плана:

$$U = \left\{ \begin{bmatrix} -0.95 \\ 0.72 \\ -0.31 \\ 0.35 \\ 0.09 \\ -0.89 \\ 0.91 \\ -0.82 \\ 0.13 \\ -0.01 \\ 0.92 \\ -0.77 \\ 0.32 \\ 1.0 \\ -0.34 \\ 0.63 \\ -0.78 \\ -0.35 \\ -0.98 \\ -0.84 \end{bmatrix}, \begin{bmatrix} -0.74 \\ 0.98 \\ -0.85 \\ -0.26 \\ 0.92 \\ 0.44 \\ -0.5 \\ 0.86 \\ -0.1 \\ -0.98 \\ -0.11 \\ -0.08 \\ -0.53 \\ 0.65 \\ -0.69 \\ -0.95 \\ -0.83 \\ -0.69 \\ -0.58 \\ -0.04 \end{bmatrix}, \begin{bmatrix} 0.09 \\ 0.76 \\ 0.71 \\ 0.73 \\ -0.93 \\ -0.72 \\ 0.26 \\ -0.13 \\ 0.57 \\ -0.62 \\ -0.75 \\ 0.78 \\ 0.58 \\ -0.13 \\ -0.84 \\ 0.87 \\ 0.47 \\ -0.92 \\ 0.17 \\ 0.27 \end{bmatrix}, \begin{bmatrix} -0.0 \\ -1.0 \\ 0.0 \\ -0.0 \\ 1.0 \\ -0.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ -0.0 \\ 0.0 \\ 0.0 \\ 1.0 \\ 0.0 \\ -0.0 \\ -0.0 \\ -0.0 \\ 0.0 \\ -0.0 \\ -1.0 \end{bmatrix} \right\}$$

Веса непрерывного начального плана: $p = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$

Длина точки плана $N = 20$.

Область планирования: $u(t_k) \in [-1,\ 1],\ k = 0, 1, \ldots, N$.

Область оценивания: $0.25 \leq \theta_i \leq 0.75,\ i = 1, 2$.

Общее число запусков $\nu = 10$.

Веса соответствующего дискретного начального плана: $p = \begin{bmatrix} 0.3 & 0.3 & 0.2 & 0.2 \end{bmatrix}$

Полученные оценки параметров: $\hat{\theta} = \begin{bmatrix} 0.342 & 0.559 \end{bmatrix}$.

Относительная погрешность в пространстве параметров:

$$\delta_\theta = \frac{||\hat{\theta} - \hat{\theta}_{true}||}{||\hat{\theta}_{true}||} = 23.9\%.$$

Средняя относительная погрешность в пространстве откликов:

$$\overline{\delta}_Y = \frac{1}{\nu} \sum_{i=1}^{q} \sum_{j=1}^{k_i} \frac{||\hat{Y}^{i,j} - \hat{Y}^{i,j}_{true}||}{||\hat{Y}^{i,j}_{true}||} = 11.8\%.$$

Синтез оптимального плана и оценивание параметров по плану

По проведению двойственной процедуры планирования входных сигналов был получен некоторый оптимальный план. В результате проведения идентификационных экспериментов по данному плану были получены оценки параметров.

Спектр оптимального плана:

$$U = \left\{ \begin{bmatrix} -1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ -1.0 \\ -1.0 \\ -1.0 \\ -1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} \right\}$$

То есть в плане одна точка, соответственно, все запуски ν будут приходиться на неё.

Полученные оценки параметров: $\hat{\theta} = \begin{bmatrix} 0.46 & 0.47 \end{bmatrix}$.

Относительная погрешность в пространстве параметров:

$$\delta_\theta = \frac{||\hat{\theta} - \hat{\theta}_{true}||}{||\hat{\theta}_{true}||} = 6.7\%.$$

Средняя относительная погрешность в пространстве откликов:

$$\overline{\delta}_Y = \frac{1}{\nu} \sum_{i=1}^{q} \sum_{j=1}^{k_i} \frac{||\hat{Y}^{i,j} - \hat{Y}^{i,j}_{true}||}{||\hat{Y}^{i,j}_{true}||} = 2.6\%.$$

Выводы

По результатам эксперимента видно уменьшение погрешности оценок при оценивании по оптимальному плану. Таким образом подтверждается эффективность процедуры активной параметрической идентификации.

Список литературы

[1] Активная параметрическая идентификация стохастических линейных систем: монография / В.И. Денисов, В.М. Чубич, О.С. Черникова, Д.И. Бобылева. — Новосибирск : Изд-во НГТУ, 2009. — 192 с. (Серия «Монографии НГТУ»).

Исходные тексты программ

```
import os
import math
import tensorflow as tf
import control
import autograd.numpy as np
import autograd
from tensorflow.contrib.distributions import MultivariateNormalFullCovariance
import scipy
import itertools
import copy
import operator

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'


class Plan(object):
    def __init__(self, q, r, N):
        pass

    def clean(self):
        pass

    def add(self, x, p):
        pass

    def rand(self):
        pass

    def round(self, v):
        pass


class Model(object):
    # TODO: introduce some more default argument values, check types, cast if
    # neccessary
    def __init__(self, F, C, G, H, x0_mean, x0_cov, w_cov, v_cov, th):
        """
        Arguments are all callables (functions) of 'th' returning python lists
        except for 'th' itself (of course)
        """

        # TODO: check if there are extra components in 'th'
        # TODO: evaluate and cast everything to numpy matrices first
        # TODO: cast scalars to numpy matrices
        # TODO: allow both constant matrices and callables

        def wrap_np(f):
            return lambda th: np.array(f(th), ndmin=2)

        self.__tf_F = F
        self.__tf_C = C
        self.__tf_G = G
        self.__tf_H = H
        self.__tf_x0_mean = x0_mean
        self.__tf_x0_cov = x0_cov
        self.__tf_w_cov = w_cov
        self.__tf_v_cov = v_cov
```

```
    # store arguments, after that check them
    F = self.__F = wrap_np(F)
    C = self.__C = wrap_np(C)
    G = self.__G = wrap_np(G)
    H = self.__H = wrap_np(H)
    x0_mean = self.__x0_mean = wrap_np(x0_mean)
    x0_cov = self.__x0_cov = wrap_np(x0_cov)
    w_cov = self.__w_cov = wrap_np(w_cov)
    v_cov = self.__v_cov = wrap_np(v_cov)

    th = self.__th = np.array(th, dtype=np.float64)

    # evaluate all functions
    F = F(th)
    C = C(th)
    H = H(th)
    G = G(th)
    w_cov = w_cov(th)    # Q
    v_cov = v_cov(th)    # R
    x0_m = x0_mean(th)
    x0_cov = x0_cov(th)  # P_0

    # get dimensions and store them as well
    self.__n = n = F.shape[0]
    self.__m = m = H.shape[0]
    self.__p = p = G.shape[1]
    self.__r = r = C.shape[1]

    x0_m = x0_m.reshape([n, 1])

    # generate means
    w_mean = np.zeros([p, 1], np.float64)
    v_mean = np.zeros([m, 1], np.float64)

    # and store them
    self.__w_mean = w_mean
    self.__v_mean = v_mean

    # check conformability
    u = np.ones([r, 1])
    # generate random vectors
    # squeeze, because mean must be one dimensional
    x = np.random.multivariate_normal(x0_m.flatten(), x0_cov)
    w = np.random.multivariate_normal(w_mean.flatten(), w_cov)
    v = np.random.multivariate_normal(v_mean.flatten(), v_cov)

    # shape them as column-vectors
    x = x.reshape([n, 1])
    w = w.reshape([p, 1])
    v = v.reshape([m, 1])

    # if model is not conformable, exception would be raised (thrown) here
    F @ x + C @ u + G @ w
    H @ x + v

    # check controllability, stability, observability
    # self.__validate()

    # if the execution reached here, all is fine so
    # define corresponding computational tensorflow graphs
    self.__define_observations_simulation()
    self.__define_likelihood_computation()

    self.__d_crit_to_opt_grad_f = autograd.grad(self.__d_crit_to_optimize)

def __define_observations_simulation(self):
    # TODO: reduce code not to create extra operations

    self.__sim_graph = tf.Graph()
    sim_graph = self.__sim_graph

    r = self.__r
    m = self.__m
    n = self.__n
    p = self.__p
    s = len(self.__th)

    x0_mean = self.__tf_x0_mean
    x0_cov = self.__tf_x0_cov

    with sim_graph.as_default():

        # FIXME: shape must be [1 x s]
        th = tf.placeholder(tf.float64, shape=(s), name='th')

        # TODO: this should be continuous function of time
        # but try to let pass array also
        u = tf.placeholder(tf.float64, shape=[r, None], name='u')

        t = tf.placeholder(tf.float64, shape=[None], name='t')

        # TODO: refactor

        # FIXME: gradient of py_func is None
        # TODO: embed function itself in the graph, must rebuild the graph
        # if the structure of the model change
        # use tf.convert_to_tensor
        F = tf.convert_to_tensor(self.__tf_F(th), tf.float64)
        F.set_shape([n, n])

        C = tf.convert_to_tensor(self.__tf_C(th), tf.float64)
        C.set_shape([n, r])

        G = tf.convert_to_tensor(self.__tf_G(th), tf.float64)
```

```
            G.set_shape([n, p])

            H = tf.convert_to_tensor(self.__tf_H(th), tf.float64)
            H.set_shape([m, n])

            x0_mean = tf.convert_to_tensor(x0_mean(th), tf.float64)
            x0_mean = tf.squeeze(x0_mean)

            x0_cov = tf.convert_to_tensor(x0_cov(th), tf.float64)
            x0_cov.set_shape([n, n])

            x0_dist = MultivariateNormalFullCovariance(x0_mean, x0_cov,
                                                       name='x0_dist')

            Q = tf.convert_to_tensor(self.__tf_w_cov(th), tf.float64)
            Q.set_shape([p, p])

            w_mean = self.__w_mean.flatten()
            w_dist = MultivariateNormalFullCovariance(w_mean, Q, name='w_dist')

            R = tf.convert_to_tensor(self.__tf_v_cov(th), tf.float64)
            R.set_shape([m, m])
            v_mean = self.__v_mean.flatten()
            v_dist = MultivariateNormalFullCovariance(v_mean, R, name='v_dist')

            def sim_obs(x):
                v = v_dist.sample()
                v = tf.reshape(v, [m, 1])
                y = H @ x + v  # the syntax is valid for Python >= 3.5
                return y

            def sim_loop_cond(x, y, t, k):
                N = tf.stack([tf.shape(t)[0]])
                N = tf.reshape(N, ())
                return tf.less(k, N-1)

            def sim_loop_body(x, y, t, k):

                # TODO: this should be function of time
                u_t_k = tf.slice(u, [0, k], [r, 1])

                def state_propagate(x):
                    w = w_dist.sample()
                    w = tf.reshape(w, [p, 1])
                    Fx = tf.matmul(F, x, name='Fx')
                    Cu = tf.matmul(C, u_t_k, name='Cu')
                    Gw = tf.matmul(G, w, name='Gw')
                    x = Fx + Cu + Gw
                    return x

                tk = tf.slice(t, [k], [2], 'tk')

                x_k = x[:, -1]
                x_k = tf.reshape(x_k, [n, 1])

                x_k = state_propagate(x_k)

                y_k = sim_obs(x_k)

                # TODO: stack instead of concat
                x = tf.concat([x, x_k], 1)
                y = tf.concat([y, y_k], 1)

                k = k + 1

                return x, y, t, k

            x = x0_dist.sample(name='x0_sample')
            x = tf.reshape(x, [n, 1], name='x')

            # this zeroth measurement should be thrown away
            y = sim_obs(x)
            k = tf.constant(0, name='k')

            shape_invariants = [tf.TensorShape([n, None]),
                                tf.TensorShape([m, None]),
                                t.get_shape(),
                                k.get_shape()]

            sim_loop = tf.while_loop(sim_loop_cond, sim_loop_body,
                                     [x, y, t, k], shape_invariants,
                                     name='sim_loop')

            self.__sim_loop_op = sim_loop

    # defines graph
    def __define_likelihood_computation(self):

        self.__lik_graph = tf.Graph()
        lik_graph = self.__lik_graph

        r = self.__r
        m = self.__m
        n = self.__n
        p = self.__p

        x0_mean = self.__tf_x0_mean
        x0_cov = self.__tf_x0_cov

        with lik_graph.as_default():
            # FIXME: Don't Repeat Yourself (in simulation and here)
            th = tf.placeholder(tf.float64, shape=[None], name='th')
            u = tf.placeholder(tf.float64, shape=[r, None], name='u')
            t = tf.placeholder(tf.float64, shape=[None], name='t')
            y = tf.placeholder(tf.float64, shape=[m, None], name='y')
```

```
N = tf.stack([tf.shape(t)[0]])
N = tf.reshape(N, ())

F = tf.convert_to_tensor(self.__tf_F(th), tf.float64)
F.set_shape([n, n])

C = tf.convert_to_tensor(self.__tf_C(th), tf.float64)
C.set_shape([n, r])

G = tf.convert_to_tensor(self.__tf_G(th), tf.float64)
G.set_shape([n, p])

H = tf.convert_to_tensor(self.__tf_H(th), tf.float64)
H.set_shape([m, n])

x0_mean = tf.convert_to_tensor(x0_mean(th), tf.float64)
x0_mean.set_shape([n, 1])

P_0 = tf.convert_to_tensor(x0_cov(th), tf.float64)
P_0.set_shape([n, n])

Q = tf.convert_to_tensor(self.__tf_w_cov(th), tf.float64)
Q.set_shape([p, p])

R = tf.convert_to_tensor(self.__tf_v_cov(th), tf.float64)
R.set_shape([m, m])

I = tf.eye(n, n, dtype=tf.float64)

def lik_loop_cond(k, P, S, t, u, x, y, yhat):
    return tf.less(k, N-1)

def lik_loop_body(k, P, S, t, u, x, y, yhat):

    # TODO: this should be function of time
    u_t_k = tf.slice(u, [0, k], [r, 1])

    # k+1, cause zeroth measurement should not be taken into account
    y_k = tf.slice(y, [0, k+1], [m, 1])

    t_k = tf.slice(t, [k], [2], 't_k')

    # TODO: extract Kalman filter to a separate class
    def state_predict(x):
        Fx = tf.matmul(F, x, name='Fx')
        Cu = tf.matmul(C, u_t_k, name='Cu')
        x = Fx + Cu
        return x

    def covariance_predict(P):
        GQtG = tf.matmul(G @ Q, G, transpose_b=True)
        PtF = tf.matmul(P, F, transpose_b=True)
        P = tf.matmul(F, P) + PtF + GQtG
        return P

    x = state_predict(x)

    P = covariance_predict(P)

    yh = H @ x

    yhat = tf.concat([yhat, yh], axis=1)

    E = y_k - yh

    B = tf.matmul(H @ P, H, transpose_b=True) + R
    invB = tf.matrix_inverse(B)

    K = tf.matmul(P, H, transpose_b=True) @ invB

    S_k = tf.matmul(E, invB @ E, transpose_a=True)
    S_k = 0.5 * (S_k + tf.log(tf.matrix_determinant(B)))

    S = S + S_k

    # state update
    x = x + tf.matmul(K, E)

    # covariance update
    P = (I - K @ H) @ P

    k = k + 1

    return k, P, S, t, u, x, y, yhat

k = tf.constant(0, name='k')
P = P_0
S = tf.constant(0.0, dtype=tf.float64, shape=[1, 1], name='S')
x = x0_mean
yhat = H @ x

shape_invariants = [k.get_shape(), P.get_shape(), S.get_shape(),
                    t.get_shape(), u.get_shape(), x.get_shape(),
                    y.get_shape(), tf.TensorShape([m, None])]

# TODO: make a named tuple of named list
lik_loop = tf.while_loop(lik_loop_cond, lik_loop_body,
                         [k, P, S, t, u, x, y, yhat],
                         shape_invariants,
                         name='lik_loop')

dS = tf.gradients(lik_loop[2], th)

self.__lik_loop_op = lik_loop
```

```
        self.__dS = dS

    def __isObservable(self, th=None):
        if th is None:
            th = self.__th
        F = np.array(self.__F(th))
        H = np.array(self.__H(th))
        n = self.__n
        obsv_matrix = control.obsv(F, H)
        rank = np.linalg.matrix_rank(obsv_matrix)
        return rank == n

    def __isControllable(self, th=None):
        if th is None:
            th = self.__th
        F = np.array(self.__F(th))
        C = np.array(self.__C(th))
        n = self.__n
        ctrb_matrix = control.ctrb(F, C)
        rank = np.linalg.matrix_rank(ctrb_matrix)
        return rank == n

    def __isStable(self, th=None):
        if th is None:
            th = self.__th
        F = np.array(self.__F(th))
        eigv = np.linalg.eigvals(F)
        abs_vals = np.abs(eigv)
        return np.all(abs_vals < 1)

    def __validate(self, th=None):
        # FIXME: do not raise exceptions
        # TODO: prove, print matrices and their criteria
        if not self.__isControllable(th):
            # raise Exception('''Model is not controllable. Set different
            #                 structure or parameters values''')
            pass

        if not self.__isStable(th):
            # raise Exception('''Model is not stable. Set different structure or
            #                 parameters values''')
            pass

        if not self.__isObservable(th):
            # raise Exception('''Model is not observable. Set different
            #                 structure or parameters values''')
            pass

    def sim(self, u, th=None):
        if th is None:
            th = self.__th

        r = self.__r
        u = np.array(u).reshape([r, -1])
        k = u.shape[1]
        t = np.linspace(0, k-1, k)

        self.__validate(th)
        g = self.__sim_graph

        if t.shape[0] != u.shape[1]:
            raise Exception('''t.shape[0] != u.shape[1]''')

        # run simulation graph
        with tf.Session(graph=g) as sess:
            t_ph = g.get_tensor_by_name('t:0')
            th_ph = g.get_tensor_by_name('th:0')
            u_ph = g.get_tensor_by_name('u:0')
            rez = sess.run(self.__sim_loop_op, {th_ph: th, t_ph: t, u_ph: u})

        return rez[1]

    def yhat(self, u, y, th=None):
        if th is None:
            th = self.__th

        k = u.shape[1]
        t = np.linspace(0, k-1, k)

        # to numpy 1D array
        th = np.array(th).squeeze()

        # self.__validate(th)
        g = self.__lik_graph

        if t.shape[0] != u.shape[1]:
            raise Exception('''t.shape[0] != u.shape[1]''')

        # run lik graph
        with tf.Session(graph=g) as sess:
            t_ph = g.get_tensor_by_name('t:0')
            th_ph = g.get_tensor_by_name('th:0')
            u_ph = g.get_tensor_by_name('u:0')
            y_ph = g.get_tensor_by_name('y:0')
            rez = sess.run(self.__lik_loop_op, {th_ph: th, t_ph: t, u_ph: u,
                                                y_ph: y})

        # TODO: make rez namedtuple
        yhat = rez[-1]
        return yhat

    def lik(self, u, y, th=None):

        # hack continuous to discrete system
        k = u.shape[1]
```

```
        t = np.linspace(0, k-1, k)

        if th is None:
            th = self.__th

        # to numpy 1D array
        th = np.array(th).squeeze()

        # self.__validate(th)
        g = self.__lik_graph

        # TODO: check for y also
        if t.shape[0] != u.shape[1]:
            raise Exception('''t.shape[0] != u.shape[1]''')

        # run lik graph
        with tf.Session(graph=g) as sess:
            t_ph = g.get_tensor_by_name('t:0')
            th_ph = g.get_tensor_by_name('th:0')
            u_ph = g.get_tensor_by_name('u:0')
            y_ph = g.get_tensor_by_name('y:0')
            rez = sess.run(self.__lik_loop_op, {th_ph: th, t_ph: t, u_ph: u,
                                                y_ph: y})

        # hack to discrete
        N = len(t)
        m = y.shape[0]
        S = rez[2]
        S = S + N*m * 0.5 + np.log(2*math.pi)

        return np.squeeze(S)

    def __L(self, th, u, y):
        return self.lik(u, y, th)

    def __dL(self, th, u, y):
        return self.dL(u, y, th)

    def dL(self, u, y, th=None):
        if th is None:
            th = self.__th

        # hack continuous to discrete system
        k = u.shape[1]
        t = np.linspace(0, k-1, k)

        # to 1D numpy array
        th = np.array(th).squeeze()

        # self.__validate(th)
        g = self.__lik_graph

        if t.shape[0] != u.shape[1]:
            raise Exception('''t.shape[0] != u.shape[1]''')

        # run lik graph
        with tf.Session(graph=g) as sess:
            t_ph = g.get_tensor_by_name('t:0')
            th_ph = g.get_tensor_by_name('th:0')
            u_ph = g.get_tensor_by_name('u:0')
            y_ph = g.get_tensor_by_name('y:0')
            rez = sess.run(self.__dS, {th_ph: th, t_ph: t, u_ph: u, y_ph: y})

        return rez[0]

    # TODO: return results as a named tuple or dictionary
    # TODO: bounds
    def mle_fit(self, th, u, y, bounds=None):
        # TODO: call slsqp, check u.shape
        th0 = th
        u = np.array(u, ndmin=2)
        rez = scipy.optimize.minimize(self.__L, th0, args=(u, y),
                                      bounds=bounds, method='SLSQP',
                                      jac=self.__dL)
        return rez


    def round_weights(self, p, v):
        return np.array(self.round_plan([0, p], v)[1])

    def round_plan(self, plan, v):
        plan = copy.deepcopy(plan)
        p = np.array(plan[1])
        q = len(p)
        sigmaI = np.ceil((v - q) * p)  # 1
        sigmaII = np.floor(v * p)
        vI = v - np.sum(sigmaI)  # 2
        vII = v - np.sum(sigmaII)
        if vI < vII:
            sigma = sigmaI
            v1 = int(vI)
        else:
            sigma = sigmaII
            v1 = int(vII)

        s = np.zeros(q)

        vps = v * p - sigma
        vps_id = [i for i in range(len(vps))]
        vps_t = [(val, key) for val, key in zip(vps, vps_id)]

        vps_t = sorted(vps_t, key=operator.itemgetter(0))
        sorted_id = [elem[1] for elem in vps_t]

        for j in range(q):
```

```
            if vps_id[j] in sorted_id[:v1]:
                s[j] = 1
            else:
                s[j] = 0

        p = (sigma + s) / v
        plan[1] = p.tolist()
        return plan

    def grad_lik_plan(self, th, plan, Y, v):
        U, p = plan
        dS = 0
        for i in range(len(U)):
            k_i = int(p[i] * v)
            for j in range(k_i):
                dS += self.__dL(th, U[i], Y[i][j])
        return dS

    def lik_plan(self, th, plan, Y, v):
        U, p = plan
        S = 0
        for i in range(len(U)):
            k_i = int(p[i] * v)
            for j in range(k_i):
                S += self.lik(U[i], Y[i][j], th)
        return S

    def mle_fit_plan(self, plan, v, th0, bounds=None):
        plan = self.round_plan(plan, v)
        U, p = plan
        q = U.shape[0]
        Y = list()
        for i in range(q):
            Y.append(list())
        for i in range(q):
            k_i = int(p[i] * v)
            for j in range(k_i):
                y = self.sim(U[i])
                Y[i].append(y)

        th = self.__th
        bounds = [(th_i-th_i*0.5, th_i+th_i*0.5) for th_i in th]

        rez = scipy.optimize.minimize(fun=self.lik_plan, x0=th0,
                                      args=(plan, Y, v),
                                      method='SLSQP', jac=self.grad_lik_plan,
                                      bounds=bounds)

        # calculate th rel tolerance
        th_e = rez['x']
        rtol_th = np.linalg.norm(th - th_e) / np.linalg.norm(th)
        rez['rtol_th'] = rtol_th

        # calc y rel tol
        y_rtols = list()
        for i in range(q):
            k_i = int(p[i] * v)
            for j in range(k_i):
                yh_e = self.yhat(U[i], Y[i][j], th_e)
                yh = self.yhat(U[i], Y[i][j])
                y_rtol = np.linalg.norm(yh_e - yh) / np.linalg.norm(yh)
                y_rtols.append(y_rtol)

        avg_y_rtol = sum(y_rtols) / len(y_rtols)
        rez['avg_y_rtol'] = avg_y_rtol
        return rez

    def fim(self, u, x0=None, th=None):
        """
        'u' is 2d numpy array [r x N]
        """
        if th is None:
            th = self.__th
        else:
            th = np.array(th)

        s = len(th)
        n = self.__n

        lst = list()
        lst.append(self.__F)
        lst.append(self.__C)
        lst.append(self.__G)
        lst.append(self.__H)
        lst.append(self.__w_cov)
        lst.append(self.__v_cov)
        lst.append(self.__x0_mean)
        lst.append(self.__x0_cov)

        # eval
        jlst = [autograd.jacobian(f)(th) for f in lst]

        # TODO: refactor?
        jlst = [[np.squeeze(j, 2) for j in np.dsplit(jel, s)] for jel in jlst]

        dF, dC, dG, dH, dQ, dR, dX0, dP0 = jlst

        dX0 = [dX0_i.reshape([n, 1]) for dX0_i in dX0]

        # eval
        F, C, G, H, Q, R, X0, P0 = [f(th) for f in lst]

        X0 = X0.reshape([n, 1])

        if x0 is not None:
```

```
    # on reshape fail exception will be raised
    X0 = np.array(x0).reshape([n, 1])

C_A = np.vstack(dC)
C_A = np.vstack([C, C_A])

M = np.zeros([s, s])

t = np.transpose
inv = np.linalg.inv
Sp = np.trace
Pe = P0
dPe = dP0
Inn = np.eye(n)
On1 = np.zeros([n, 1])

# TODO: cast every thing to np.matrix and use '*' multiplication syntax

def F_A_f(F, dF, H, K_):
    _1st_col = [dF_i - K_ @ dH_i for dF_i, dH_i in zip(dF, dH)]
    _1st_col = np.vstack(_1st_col)

    bdiag = scipy.linalg.block_diag(*[F - K_ @ H] * s)
    rez = np.hstack([_1st_col, bdiag])

    _1st_row = np.hstack([np.zeros([n, n])] * s)
    _1st_row = np.hstack([F, _1st_row])

    rez = np.vstack([_1st_row, rez])
    return rez

def X_Ap_f(F_A, X_Ap, u, k):
    if k == 0:
        F_ = np.vstack(dF)
        F_ = np.vstack([F, F_])

        # force dX0_i to be 2D array
        FdX0 = [F @ np.array(dX0_i, ndmin=2) for dX0_i in dX0]
        FdX0 = np.vstack(FdX0)
        OFdX0 = np.vstack([On1, FdX0])

        # u[:,[k]] - get k-th column as column vector
        return F_ @ X0 + OFdX0 + C_A @ u[:, [0]]
    elif k > 0:
        return F_A @ X_Ap + C_A @ u[:, [k]]

def Cf(i):
    i = i + 1
    zeros = [np.zeros([n, n])] * i
    zeros = np.hstack(zeros) if i else []
    C = np.hstack([zeros, np.eye(n)]) if i else np.eye(n)
    zeros = [np.zeros([n, n])] * (s-i)
    zeros = np.hstack(zeros) if s-i else []
    C = np.hstack([C, zeros]) if s-i else C
    return C

u = np.array(u, ndmin=2)
N = u.shape[1]

if u.shape[0] != C.shape[1]:
    raise Exception('invalid shape of \'u\'')

for k in range(N):
    if k == 0:
        E_A = np.zeros([n*(s+1), n*(s+1)])
        X_Ap = X_Ap_f(None, None, u, k)
        F_A = None
        K_A = None
        B = None
    elif k > 0:
        E_A = F_A @ E_A @ t(F_A) + K_A @ B @ t(K_A)
        X_Ap = X_Ap_f(F_A, X_Ap, u, k)

    # Pp, B, K, Pu, K_
    Pp = F @ Pe @ t(F) + G @ Q @ t(G)
    B = H @ Pp @ t(H) + R
    invB = inv(B)
    K = Pp @ t(H) @ invB
    Pu = (Inn - K @ H) @ Pp
    K_ = F @ K

    F_A = F_A_f(F, dF, H, K_)

    # TODO: numba jit it
    dPp = [dF_i @ Pe @ t(F) + F @ dPe_i @ t(F) + F @ Pe @ t(dF_i) +
           dG_i @ Q @ t(G) + G @ dQ_i @ t(G) + G @ Q @ t(dG_i)
           for dF_i, dPe_i, dG_i, dQ_i in zip(dF, dPe, dG, dQ)]

    dB = [dH_i @ Pp @ t(H) + H @ dPp_i @ t(H) + H @ Pp @ t(dH_i) + dR_i
          for dH_i, dPp_i, dR_i in zip(dH, dPp, dR)]

    dK = [(dPp_i @ t(H) + Pp @ t(dH_i) - Pp @ t(H) @ invB @ dB_i) @ invB
          for dPp_i, dH_i, dB_i in zip(dPp, dH, dB)]

    dPu = [(Inn - K @ H) @ dPp_i - (dK_i @ H + K @ dH_i) @ Pp
           for dPp_i, dK_i, dH_i in zip(dPp, dK, dH)]

    dK_ = [dF_i @ K + F @ dK_i for dF_i, dK_i in zip(dF, dK)]

    K_A = np.vstack(dK_)
    K_A = np.vstack([K_, K_A])

    # 8: AM
    AM = list()
```

```
            EXX = E_A + X_Ap @ t(X_Ap)

            C0 = Cf(0)

            for i, j in itertools.product(range(s), range(s)):
                S1 = Sp(C0 @ EXX @ t(C0) @ t(dH[j]) @ invB @ dH[i])
                S2 = Sp(C0 @ EXX @ t(Cf(j)) @ t(H) @ invB @ dH[i])
                S3 = Sp(Cf(i) @ EXX @ t(C0) @ t(dH[j]) @ invB @ H)
                S4 = Sp(Cf(i) @ EXX @ t(Cf(j)) @ t(H) @ invB @ H)
                S5 = 0.5 * Sp(dB[i] @ invB @ dB[j] @ invB)
                AM.append(S1 + S2 + S3 + S4 + S5)

            AM = np.array(AM).reshape([s, s])
            M = M + AM

            # update P, dP etc.
            Pe = Pu
            dPe = dPu

        return M

    def norm_fim(self, plan, th=None):
        ''' plan: list of 'x' and 'p' '''
        ''' x: 3d np array [q x r x N] '''
        ''' p: list or 1d np array '''
        x, p = plan
        x = np.array(x, ndmin=2)
        p = np.array(p)

        # FIXME, TODO: validate plan
        # for x_i, p_i in zip(x, p):
        #     if len(x_i) != self.__n:
        #         raise Exception('invalid plan: len(x_i) != n')

        Mn = 0

        # extract p, x and compute fim for every point of the plan
        for x_i, p_i in zip(x, p):
            # TODO: compute in parallel
            Mn += p_i * self.fim(u=x_i, x0=None, th=th)
        return Mn

    def d_opt_crit(self, plan, th=None):
        ''' plan: list of 'x' and 'p' '''
        ''' x: 3d np array [q x r x N] '''
        ''' p: list or 1d np array '''
        Mn = self.norm_fim(plan, th)
        sign, logdet = np.linalg.slogdet(Mn)
        return -logdet

    def __d_crit_to_opt_grad(self, plan, q, th=None):
        ''' plan is 1D np.array or list '''
        plan = np.array(plan)
        grad = self.__d_crit_to_opt_grad_f(plan, q, th)
        return grad

    # this wraps self.d_opt_crit() above
    # for scipy.optimize.minimize
    def __d_crit_to_optimize(self, plan, q, th=None):
        # unflatten plan
        r = self.__r
        p = plan[-q:]
        x = plan[:-q]
        x = np.reshape(x, [q, r, -1])
        plan = [x, p]
        crit = self.d_opt_crit(plan, th)
        return crit

    # TODO: take bounds
    def direct(self, plan0, th=None):
        ''' plan0: list of: list (or 3D np.array) of 'u' and list of 'p' '''
        r = self.__r

        x, p = plan0
        x = np.array(x)
        p = np.array(p)

        q = len(p)
        N = x.shape[-1]

        x_bounds = [(-1, 1)] * q * r * N
        p_bounds = [(0, 1)] * q
        bounds = x_bounds + p_bounds  # concat lists

        def heq(x):
            p = x[-q:]
            return np.sum(p) - 1

        constraints = {'type': 'eq', 'fun': heq}

        x0 = x.flatten()
        x0 = np.hstack([x0, p])

        rez = scipy.optimize.minimize(fun=self.__d_crit_to_optimize, x0=x0,
                                      jac=self.__d_crit_to_opt_grad,
                                      args=(q, th), method='SLSQP',
                                      constraints=constraints, bounds=bounds)
        new_plan = rez['x']

        pn = new_plan[-q:]
        xn = new_plan[:-q].reshape([q, r, N])

        # TODO: return loss and its jacobian values
```

```
        # return dictionary
        return [xn, pn]

    def clean(self, plan, dn=0.5, dp=0.1):
        ''' plan = [x, p], x is 3D array, p is list or 1D np array '''
        x, p = plan
        p = list(p)

        # clean by weight
        while True:
            indices = [i for i in range(len(p)) if p[i] < dp]
            if len(indices) == 0:
                break

            i = indices[0]
            x = np.delete(x, i, 0)
            p_i = p.pop(i)
            p = [p_j + p_i / len(p) for p_j in p]
            p = [p_i / sum(p) for p_i in p]  # make sure sum(p) = 1

        q, r, N = x.shape
        x = x.reshape([q, -1])

        # clean by distance
        while True:
            tree = scipy.spatial.cKDTree(x)
            bt = tree.query_ball_tree(tree, dn)
            lengths = [len(bt_i) for bt_i in bt]
            max_length = max(lengths)
            if max_length == 1:
                break  # nothing to clean
            else:  # clean
                i = lengths.index(max_length)
                indices = bt[i]  # get close points indices

                # merge points to new one
                new_point = [p[i] * x[i] for i in indices]
                px = sum([p[i] for i in indices])
                new_point = sum(new_point) / px

                x = np.delete(x, indices, 0)   # delete merged points
                x = np.vstack([x, new_point])  # append new point

                # merge corresponding weights to new one
                pn = sum([p[i] for i in range(len(p)) if i in indices])

                # delete old weights
                p = [p[i] for i in range(len(p)) if i not in indices]

                p.append(pn)  # add new weight

        q = len(p)
        x = x.reshape([q, r, N])

        return [x, p]

    # wraps fim()
    def __mu(self, u, M_plan, th):
        M = self.fim(u=u, th=th)
        return -np.trace(np.linalg.inv(M_plan) @ M)

    def __crit_tau(self, tau, a, plan, th):
        x, p = plan
        p = list(p)
        x = np.array(x)
        a = np.expand_dims(a, 0)  # TODO: set shape explicitly
        x = np.concatenate([x, a])
        p = [p_i * (1 - tau) for p_i in p]
        p.append(tau)
        plan = [x, p]
        crit = self.d_opt_crit(plan, th)
        return crit

    def rand_plan(self, N, q=None, bounds=None):
        r = self.__r
        s = len(self.__th)
        if q is None:
            q = int((s + 1) * s / 2 + 1)
        x = np.random.uniform(-1, 1, [q, r, N])
        p = [1 / q] * q
        return [x, p]

    def dual(self, plan, th=None, d=0.05):
        ''' plan '''
        dmu = autograd.grad(self.__mu)  # this is *not* time consuming

        plan = copy.deepcopy(plan)

        if th is None:
            th = self.__th
        else:
            th = np.array(th)

        eta = len(th)
        r = self.__r
        X, p = plan  # TODO: make plan class
        N = X.shape[-1]

        crit_tau_grad = autograd.grad(self.__crit_tau)

        x_bounds = [(-1, 1)] * r * N

        while True:
            M_plan = self.norm_fim(plan, th)
```

```
while True:
    x_guess = np.random.uniform(-1, 1, [r, N])  # FIXME

    # nlopts <- list(xtol_rel=1e-3, maxeval=1e3)
    rez = scipy.optimize.minimize(fun=self.__mu, x0=x_guess,
                                  args=(M_plan, th),
                                  method='SLSQP', jac=dmu,
                                  bounds=x_bounds, tol=None,
                                  options=None)
    x_opt = rez['x'].reshape([r, N])
    mu = -rez['fun']

    if abs(mu - eta) <= d:
        return list(plan)

    if mu > eta:
        break

while True:
    # XXX: this was needed to get non singular tau value,
    # not sure if it is still needed
    tau_guess = np.random.uniform(size=1)
    tau_crit = self.__crit_tau(tau_guess, x_opt,
                               copy.deepcopy(plan), th)
    if not np.isnan(tau_crit):
        break

rez = scipy.optimize.minimize(fun=self.__crit_tau, x0=tau_guess,
                              args=(x_opt, copy.deepcopy(plan), th),
                              bounds=[(0, 1)],
                              method='SLSQP', jac=crit_tau_grad)

tau_opt = rez['x']

# add x_opt, tau_opt to plan
X, p = plan
x_opt = np.expand_dims(x_opt, 0)
X = np.concatenate([X, x_opt])
tau_opt = tau_opt[0]
p = [p_i - tau_opt / len(p) for p_i in p]
p.append(tau_opt)
plan = [X, p]

# clean plan
plan = self.clean(copy.deepcopy(plan))

# continue
```