```
import os
from typing import List, Optional, Union

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy.integrate
import scipy.optimize
import scipy.stats as st
import seaborn as sns
from matplotlib.lines import Line2D
from sklearn.mixture import GaussianMixture, BayesianGaussianMixture
from tqdm import tqdm


mpl.rcParams["font.family"] = "sans-serif"
plt.rcParams["figure.figsize"] = (5, 3)  # type: ignore
mpl.rcParams["pdf.fonttype"] = 42
sns.set_style(
    "ticks",
    {
        "xtick.major.size": 4,
        "ytick.major.size": 4,
        "font_color": "k",
        "axes.edgecolor": "k",
        "xtick.color": "k",
        "ytick.color": "k",
    },
)
sns.set_context("talk", font_scale=1)

DEBUG = False
min_cit = 5.5
max_cit = 9.5
max_mrna = 600 * (max_cit - min_cit)


def printdb(s: str) -> None:
    if DEBUG:
        print(s)


def calculate_a(t: np.ndarray, ks: float, tlag: float) -> np.ndarray:
    """
    calculate_a computes the active fraction using the 3-state model

    Args:
        t (np.ndarray): time since recruitment began
        ks (float): rate of reversible silencing
        tlag (float): lag time before silencing

    Returns:
        np.ndarray: array of fractions on between 0 and 1 of active cells
    """
    return 1.0 * (t < tlag) + np.exp(-1 * ks * (t - tlag)) * (t >= tlag)


def prob_cit_on_simple(
    log_cits: np.ndarray,
    times: np.ndarray,
    on_frac: float,
    tlag: float,
    lmbda: float,
    sigma_on: float,
    gamma: float,
    mu: float,
    sigma_off: float,
) -> np.ndarray:
    """
```

```
    prob_cit_on_simple computes the pdf of citrine in the ON population
    (either active or silent)

    Args:
        log_cits (np.ndarray): log 10 of citrine levels
        times (np.ndarray): time in days corresponding to citrine levels
        on_frac (float): fraction of cells that don't silence
        tlag (float): lag time before silencing begins
        lmbda (float): net ratio of mRNA production to degradation
        sigma_on (float): standard deviation of the ON population
        gamma (float): rate of mRNA degradation + dilution
        mu (float): mean of OFF population
        sigma_off (float): standard deviation of OFF population

    Returns:
        np.ndarray: array of probability densities corresponding to citrines
    """
    mrna_levels = np.arange(0, 600 * (max_cit - mu))
    mrna_means = lmbda * (times < tlag) + lmbda * np.exp(-gamma * (times - tlag)) * (
        times >= tlag
    )

    sigmas = np.array(
        [sigma_off if t >= 4 and on_frac < 0.15 else sigma_on for t in times]
    )

    p_m = st.poisson(mrna_means).pmf(mrna_levels.reshape(-1, 1))
    p_c = st.norm(((mrna_levels.reshape(-1, 1) / 600) + mu).reshape(-1, 1), sigmas).pdf(
        log_cits
    )
    res = np.sum(p_m * p_c, axis=0)
    return res  # asumes unif distribution over mrna levels


def telegraph_3sm_pdf(
    xdata: np.ndarray,
    bg_frac: float,
    on_frac: float,
    ks: float,
    tlag: float,
    bprime: float,
    lmbda: float,
    s_on: float,
    gamma: float,
    u_sil: float,
    f_sil: float,
    u_off: float,
    s_off: float,
) -> np.ndarray:
    """
    telegraph_3sm_pdf computes the pdf using the 3-state model

    Args:
        xdata (np.ndarray): Mx2 array of time being the 1st col and cit being the 2nd
        bg_frac (float): fraction of cells background silenced
        on_frac (float): fraction of cells permanently on
        ks (float): rate of silencing
        tlag (float): lag time before silencing begins
        bprime (float): fraction of cells ON at equilibrium
        lmbda (float): ratio of mRNA production to degradation
        s_on (float): standard deviation of ON population
        gamma (float): net rate of mRNA degradation + dilution
        u_sil (float): minor steady-state of decaying fluorescence peak
        f_sil (float): fraction of non-ON population that stays at u_sil
        u_off (float): mean of OFF population
        s_off (float): standard deviation of OFF population

    Returns:
        np.ndarray: probability density of citrine, time tuples
    """
```

```
    times = xdata[:, 0]
    log_cits = xdata[:, 1]

    # cit_range = (np.arange(0, max_mrna) / 600) + min_cit
    # est_mrnas = np.rint(u_off + ((log_cits - u_off) * 600))

    vuln_frac = 1 - bg_frac - on_frac
    p_active = on_frac + vuln_frac * bprime * calculate_a(times, ks, tlag)
    p_silent = vuln_frac * bprime * (1 - calculate_a(times, ks, tlag))
    p_off = bg_frac + vuln_frac * (1 - bprime)

    # p_active = (1 - bg_frac) * bprime * calculate_a(times, ks, tlag)
    # p_silent = (1 - bg_frac) * bprime * (1 - p_active)
    # p_off = bg_frac + (1 - bg_frac) * (1 - bprime)

    p_cit_active = prob_cit_on_simple(
        log_cits,
        np.tile(0, len(log_cits)),
        on_frac,
        tlag,
        lmbda,
        s_on,
        gamma,
        u_off,
        s_off,
    )
    if ks != 0:
        # p_cit_silent = prob_cit_on_silent(
        #     log_cits, times, ks, lmbda, s_on, gamma, u_off, s_off
        # )
        p_cit_silent_quiet = prob_cit_on_simple(
            log_cits,
            times,
            on_frac,
            tlag,
            max(lmbda - (600 * (u_sil - u_off)), 0),
            s_on,
            gamma,
            u_sil,
            s_off,
        )
        p_cit_silent_off = prob_cit_on_simple(
            log_cits,
            times,
            on_frac,
            tlag,
            lmbda,
            s_on,
            gamma,
            u_off,
            s_off,
        )
        p_cit_silent = f_sil * p_cit_silent_quiet + (1 - f_sil) * p_cit_silent_off
    else:
        p_cit_silent = np.array([0 for _ in log_cits])
    p_cit_off = st.norm(u_off, s_off).pdf(log_cits)

    return p_cit_active * p_active + p_cit_silent * p_silent + p_off * p_cit_off


def get_ks_tlag_gamma_peak_offratio(
    times: np.ndarray,
    log_cits: np.ndarray,
    u_off: float,
    plasmid: float,
    description: str,
    on_frac: float,
) -> List[float]:
    """
    get_ks_tlag_gamma computes ks, tlag, and gamma from citrine data
```

```
    Args:
        times (np.ndarray): list of times in days
        log_cits (np.ndarray): list of log citrine values
        u_off (float): mean of OFF population
        plasmid (float): number of plasmid to be fit
        on_frac (float): fraction of cells ON at end of recruitment

    Returns:
        List[float]: list of [ks, tlag, gamma, sil_peak]
    """
    dq = pd.DataFrame.from_dict({"time": times, "citrine": log_cits})

    # get fractions on
    dq["on"] = dq["citrine"] >= 8

    def get_locs_fracs_gmm(cits: np.ndarray, d: float) -> pd.DataFrame:
        bgm = BayesianGaussianMixture(n_components=3).fit(cits.reshape(-1, 1))
        poff, psil, pon = sorted(list(bgm.means_))
        ps = [poff[0], psil[0], pon[0]]
        sds = list(np.sqrt([bgm.covariances_[i][0][0] for i in range(3)]))
        sds = [sds[list(bgm.means_).index(p)] for p in ps]
        ws = [list(bgm.weights_)[list(bgm.means_).index(p)] for p in ps]
        return pd.DataFrame.from_dict(
            {
                "peak": ps,
                "std": sds,
                "weight": ws,
                "pop": ["off", "sil", "on"],
                "t": [d, d, d],
            }
        )

    def get_gmm_df() -> pd.DataFrame:
        return pd.concat(
            [
                get_locs_fracs_gmm(np.array(dq[dq["time"] == d]["citrine"]), d)
                for d in tvals
            ]
        )

    def get_frac_on(cits: np.ndarray, d: float) -> float:
        """
        get_frac_on computes the fraction of cells on

        Args:
            cits (np.ndarray): list of citrine values
            d (float): day

        Returns:
            float: fraction of cells on
        """
        # i do not care that this code is repeated at the moment
        return np.mean(cits > 8)

        gm2 = GaussianMixture(n_components=2).fit(cits.reshape(-1, 1))
        gm3 = GaussianMixture(n_components=3).fit(cits.reshape(-1, 1))
        n_comps = (
            2 if gm2.aic(cits.reshape(-1, 1)) >= gm3.aic(cits.reshape(-1, 1)) else 3
        )
        gm = BayesianGaussianMixture(n_components=n_comps).fit(cits.reshape(-1, 1))
        means = [x[0] for x in gm.means_]
        mmax = max(means)
        imax = means.index(mmax)
        cmax = gm.covariances_[imax][0][0]
        nmax = st.norm(loc=mmax, scale=cmax)
        wmax = list(gm.weights_)[imax]
        return wmax * (nmax.cdf(9) - nmax.cdf(8))
        if max(means) >= 8:
            return gm.weights_[means.index(max(means))]
```

```
        else:
            return 0.0

    def get_loc_on(cits: np.ndarray, d) -> float:
        """
        get_loc_on returns the location of the ON peak

        Args:
            cits (np.ndarray): list of citrine values
            d (_type_): day

        Returns:
            float: ON peak location
        """

        ddf = get_locs_fracs_gmm(cits, d)
        won = list(ddf[ddf["pop"] == "on"]["weight"])[0]
        wsil = list(ddf[ddf["pop"] == "sil"]["weight"])[0]
        woff = list(ddf[ddf["pop"] == "off"]["weight"])[0]
        lon = list(ddf[ddf["pop"] == "on"]["peak"])[0]
        lsil = list(ddf[ddf["pop"] == "sil"]["peak"])[0]
        loff = list(ddf[ddf["pop"] == "off"]["peak"])[0]
        # don = list(ddf[ddf["pop"] == "on"]["std"])[0]
        # dsil = list(ddf[ddf["pop"] == "sil"]["std"])[0]
        # doff = list(ddf[ddf["pop"] == "off"]["std"])[0]

        ws = [won, wsil, woff]
        ls = [lon, lsil, loff]
        # ds = [don, dsil, doff]

        return ls[ws.index(max(ws))]


        loc = (won * lon + wsil * lsil) / (won + wsil)
        return loc

        gm = GaussianMixture(n_components=2).fit(cits.reshape(-1, 1))
        m1, m2 = gm.means_
        w1, w2 = gm.weights_
        m1 = m1[0]
        m2 = m2[0]
        m_min = min(m1, m2)
        m_max = max(m1, m2)
        w_max = w1 if m_max == m1 else w2
        if w_max < 0.1:
            return m_min
        else:
            return m_max

    tvals = sorted(list(set(list(dq["time"]))))
    fvals = [get_frac_on(np.array(dq[dq["time"] == d]["citrine"]), d) for d in tvals]
    cvals = [get_loc_on(np.array(dq[dq["time"] == d]["citrine"]), d) for d in tvals]

    gdf = get_gmm_df()
    if DEBUG:
        fig, ax = plt.subplots(1, 3, figsize=(14, 4))
        sns.lineplot(data=gdf, x="t", y="peak", hue="pop", palette="Dark2", ax=ax[0])
        sns.lineplot(data=gdf, x="t", y="std", hue="pop", palette="Dark2", ax=ax[1])
        sns.lineplot(data=gdf, x="t", y="weight", hue="pop", palette="Dark2", ax=ax[2])
        plt.tight_layout()
        fig.savefig("./plot_telegraph/" + description + "_gmm.pdf", bbox_inches="tight")

    end_off = list(gdf[(gdf["t"] == 5) & (gdf["pop"] == "off")]["weight"])[0]
    end_sil = list(gdf[(gdf["t"] == 5) & (gdf["pop"] == "sil")]["weight"])[0]
    f = end_sil / (end_sil + end_off)

    start_fon = fvals[0]

    def fraction_off_curve(t, ks, tlag):
        return on_frac + (start_fon - on_frac) * (
```

```
            1 * (t < tlag) + np.exp(-ks * (t - tlag)) * (t >= tlag)
        )

    def peak_decay_curve(t, gamma, u_sil, tlag):
        return u_sil + (np.max(cvals) - u_sil) * np.exp(
            -gamma * np.maximum(np.zeros(t.shape), (t - tlag))
        )

    fopt, _ = scipy.optimize.curve_fit(
        fraction_off_curve,
        tvals,
        fvals,
        p0=[5, 1],
        bounds=[[0, 0], [15, 2]],
        max_nfev=1000,
    )
    ks, t1 = fopt

    p = list(gdf[(gdf["t"] == 5) & (gdf["pop"] == "sil")]["peak"])[0]
    if p <= 7:
        f = 0
        p = u_off

    if cvals[-1] > 7.25:
        copt, _ = scipy.optimize.curve_fit(
            lambda x, y, t: peak_decay_curve(x, y, p, t),
            tvals,
            cvals,
            p0=[0.5, 0.9],
            bounds=[[0.4, 0.0], [1.0, 2.0]],
        )
    else:
        copt, _ = scipy.optimize.curve_fit(
            lambda x, y, t: peak_decay_curve(x, y, p, t),
            [0, 5],
            [cvals[0], u_off],
            p0=[0.5, 0.9],
            bounds=[[0.4, 0.0], [1.0, 2.0]],
        )
    g = copt[0]

    # if fvals[-1] < 0.05:
    #     p = u_off
    #     g = 0.639
    # else:
    #     p = cvals[-1]
    #     #        p = u_off
    #     copt, _ = scipy.optimize.curve_fit(
    #         lambda x, y, t: peak_decay_curve(x, y, p, t),
    #         tvals,
    #         cvals,
    #         p0=[0.5, 0.9],
    #         bounds=[[0.4, 0.0], [1.0, 2.0]],
    #     )
    #     g = copt[0]
    #     if cvals[-1] > 8:
    #         g = 0.639

    if DEBUG:
        fig, ax = plt.subplots(1, 2, figsize=(8, 4))
        ax[0].set_title("Fraction Off Curve")
        ax[0].plot(
            np.linspace(0, 5), fraction_off_curve(np.linspace(0, 5), *fopt), color="red"
        )
        ax[0].scatter(tvals, fvals, color="blue")
        ax[0].set_xlim(0, 5)
        ax[0].set_ylim(0, 1)
        ax[1].set_title("Peak Decay Curve")
        ax[1].plot(
            np.linspace(0, 5),
```

```
            peak_decay_curve(np.linspace(0, 5), g, p, t1),
            color="red",
        )
        ax[1].scatter(tvals, cvals, color="blue")
        ax[1].set_xlim(1, 5)
        ax[1].set_ylim(5.5, 9.5)
        plt.tight_layout()
        fig.savefig("./plot_telegraph/" + description + "_ks_params.pdf")

    return [ks, t1, g, p, f]


def get_fit_params(
    times: np.ndarray,
    log_cits: np.ndarray,
    plasmid: float,
    description: str,
    bg_frac: float,
    on_frac: float,
) -> List[float]:
    """
    get_fit_params returns optimal fit parameters for the telegraph model given data
    and estimated background silenced fraction

    Args:
        times (np.ndarray): numpy array of times
        log_cits (np.ndarray): numpy array of log citrine measurements
        plasmid (float): number corresponding to the plasmid of interest
        bg_frac (float): fraction of cells background silenced
        on_frac (float): fraction of cells permanently on

    Returns:
        List[float]: list of optimal parameters, in the order:
            bg_frac, ks, tlag, bprime, lambda, sigma_on, gamma, mu_off, sigma_off
    """
    xdata = np.transpose(np.array([times, log_cits]))

    # fit mu and sigma to last day data less than 1e7
    lxdata = xdata[xdata[:, 0] == np.max(xdata[:, 0])]
    lxdata = lxdata[lxdata[:, 1] <= 7]
    lkde = st.gaussian_kde(dataset=lxdata[:, 1])
    lydata = lkde.evaluate(lxdata[:, 1])

    def off_fn(x, mu, sigma):
        # here we pick super strong silencing params
        return telegraph_3sm_pdf(
            x, bg_frac, on_frac, 10, 0, 0, 0, 1e-200, 10, mu, 1.0, mu, sigma
        )

    # printdb("Fitting mu, sigma")
    lopt, _ = scipy.optimize.curve_fit(
        f=off_fn,
        xdata=lxdata,
        ydata=lydata,
        p0=[6.3, 0.5],
        bounds=[[5.5, 0.1], [7, 1.0]],
    )
    fit_m, fit_soff = lopt
    printdb("\tFound mu={:.2f} and sigma_off={:.2f}".format(fit_m, fit_soff))

    # fit beta and lambda to day 0 data
    zxdata = xdata[xdata[:, 0] == 0]
    zkde = st.gaussian_kde(dataset=zxdata[:, 1])
    zydata = zkde.evaluate(zxdata[:, 1])

    def zero_fn(x, bp, lmbda, s_on):
        return telegraph_3sm_pdf(
            x, bg_frac, on_frac, 0, 10, bp, lmbda, s_on, 0, fit_m, 0.0, fit_m, fit_soff
        )
```

```
    # printdb("Fitting beta, lambda")
    zopt, _ = scipy.optimize.curve_fit(
        f=zero_fn,
        xdata=zxdata,
        ydata=zydata,
        p0=[0.5, 1200, 0.5],
        bounds=[[0, 600, 0], [1, max_mrna, 2]],
    )
    fit_b, fit_l, fit_son = zopt
    printdb(
        "\tFound beta={:.2f}, lambda={:.2f}, sigma_on={:.2f}".format(
            fit_b, fit_l, fit_son
        )
    )

    # last, fit ks, tlag, and gamma
    # printdb("Fitting ks, tlag, gamma")
    fit_k, fit_t, fit_g, fit_p, fit_f = get_ks_tlag_gamma_peak_offratio(
        times, log_cits, fit_m, plasmid, description, on_frac
    )
    fit_k = fit_k if fit_k > 0.1 else 0

    printdb(
        "\tFound ks={:.2f}, tlag={:.2f}, gamma={:.2f}, u_sil={:.2f}, f_sil={:.2f}".format
(
            fit_k, fit_t, fit_g, fit_p, fit_f
        )
    )

    return [
        bg_frac,
        on_frac,
        fit_k,
        fit_t,
        fit_b,
        fit_l,
        fit_son,
        fit_g,
        fit_p,
        fit_f,
        fit_m,
        fit_soff,
    ]


def estimate_bg_frac(rtetr_log_cits: np.ndarray) -> float:
    """
    estimate_bg_frac computes background silenced fraction

    Args:
        rtetr_log_cits (np.ndarray): array of rTetR-only no dox citrine values

    Returns:
        float: estimated fraction of background silenced cells
    """
    gm = GaussianMixture(n_components=2).fit(rtetr_log_cits.reshape(-1, 1))
    m1, m2 = gm.means_
    w1, w2 = gm.weights_
    m1 = m1[0]
    m2 = m2[0]
    m_min = min(m1, m2)
    w_min = w1 if m_min == m1 else w2
    return w_min


def import_repression_data() -> pd.DataFrame:
    """
    import_repression_data imports repression data from the two measurements

    Returns:
```

```
        pd.DataFrame: DF of all cells measured, gated for mch+ and singlets
    """
    rep_round1 = pd.read_csv("./data/rep_rd1_all_cells_live_mch_gated.csv")
    rep_round1 = rep_round1[rep_round1["treatment"] == "none"]
    rep_round1["date"] = "2021.11.17"

    rep_round2 = pd.read_csv("./data/repadd_stapl_all_cells_mch_live.csv")
    rep_round2 = rep_round2[rep_round2["asv"] == 0]
    rep_round2 = rep_round2.drop("asv", axis=1)
    rep_round2["date"] = "2022.04.22"

    rep_df = pd.concat([rep_round1, rep_round2])
    rep_df = rep_df.replace([np.inf, -np.inf], np.nan)
    rep_df = rep_df.dropna(subset=["mCitrine-A"])
    return rep_df


def get_test_histogram_data(
    plasmid: int = 217, df: Optional[pd.DataFrame] = None, nodox: Optional[bool] = None
) -> pd.DataFrame:
    """
    get_test_histogram_data gets test histogram data for a given plasmid

    Args:
        plasmid (int, optional): plasmid number, defaults to 217.

    Returns:
        pd.DataFrame: dataframe of all +dox cells for that plasmid
    """
    if df is None:
        df = import_repression_data()
    df = df[df["plasmid"] == plasmid]
    if nodox:
        df = df[df["dox"] == 0]
    else:
        df = df[df["dox"] == 1000]
    return df


def fit_and_plot(
    plasmid_df: Union[int, Optional[pd.DataFrame]] = None,
    params: Optional[List[float]] = None,
    bg_frac: Optional[float] = None,
    on_frac: Optional[float] = None,
):
    """
    fit_and_plot fits the telegraph model to plasmid data and plots the fit

    Args:
        plasmid_df (Union[int, Optional[pd.DataFrame]], optional): plasmid dataframe.
                Defaults to None.
        params (Optional[List[float]], optional): list of parameters. Defaults to None.
        bg_frac (Optional[float]): background silent fraction
        on_frac (Optional[float]): permanently active fraction

    Returns:
        [plt.Figure, pd.DataFrame]: dataframe of parameter fits, plot of the fit
    """
    if plasmid_df is None:
        printdb("Reading in data with default plasmid")
        plasmid_df = get_test_histogram_data()
    elif isinstance(plasmid_df, int):
        printdb("Reading in data for plasmid " + str(plasmid_df))  # type: ignore
        plasmid_df = get_test_histogram_data(
            plasmid_df
        )  # enabling some real abuse here
    else:
        printdb("User provided dataframe, no need to load")
    plasmid = list(plasmid_df["plasmid"])[0]
    descr = list(plasmid_df["description"])[0]
```

```
    if bg_frac is None:
        nodox_rtetr_df = get_test_histogram_data(126, None, True)
        nodox_cits = list(nodox_rtetr_df["mCitrine-A"])
        nodox_cits = [n for n in nodox_cits if n > 0]
        nodox_cits = np.log10(nodox_cits)
        bg_frac = estimate_bg_frac(nodox_cits)

    if on_frac is None:
        # on_frac = 0.0
        last_df = plasmid_df[plasmid_df["day"] == 5]
        last_df = last_df[last_df["mCitrine-A"] > 0]
        gm = GaussianMixture(n_components=2).fit(
            np.log10(np.array(last_df["mCitrine-A"])).reshape(-1, 1)
        )
        m = max(list(gm.means_))
        idx = list(gm.means_).index(m)
        on_frac = gm.weights_[idx] if m > 8 else 0
        # on_frac = np.mean(cits > 7.7)
        # on_df = last_df[last_df["mCitrine-A"] > 1e8]
        # on_frac = on_df.shape[0] / last_df.shape[0]

    if params is None:
        printdb("Getting fit parameters for pAXM" + str(plasmid) + ", " + descr)
        printdb("\tFound bg_frac={:.2f}, on_frac={:.2f}".format(bg_frac, on_frac))
        plasmid_df = plasmid_df[plasmid_df["mCitrine-A"] > 0]
        times = np.array(plasmid_df["day"])
        lcits = np.log10(np.array(plasmid_df["mCitrine-A"]))
        popt = get_fit_params(
            times, lcits, plasmid, descr, bg_frac, on_frac  # type: ignore
        )
    else:
        printdb("Given parameters, going directly to plotting")
        popt = params

    bg, on, ks, tl, bp, lm, son, y, us, fs, uo, soff = popt

    daylist = sorted(list(set(list(plasmid_df["day"]))))
    nrows = int(max(1, len(daylist) / 3))
    ncols = 3
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(5 * ncols, 3 * nrows))
    for i, a in enumerate(ax.flat):  # type: ignore
        day = int(daylist[i])
        # print(f"Working on day {day:d}")

        def popt_fun(log_cits: np.ndarray) -> np.ndarray:
            days = np.array([day for _ in log_cits])
            xdata = np.transpose(np.array([days, log_cits]))
            return telegraph_3sm_pdf(xdata, *popt)

        sns.kdeplot(
            data=plasmid_df[plasmid_df["day"] == day],
            x="mCitrine-A",
            color="tab:red",
            log_scale=True,
            ax=a,
        )

        log_cits = np.linspace(min_cit, max_cit, num=100)
        yvals = popt_fun(log_cits)
        xvals = np.power(10, log_cits)
        # print(popt, xvals, yvals)
        a.plot(xvals, yvals, color="tab:blue", linestyle="--")

        a.set_title("Day " + str(int(day)))
        a.set_xscale("log")
        a.set_xlim(3.16e5, 3.16e9)
        a.set_xticks([1e6, 1e7, 1e8, 1e9])

        if DEBUG:
```

```
            a.axvline(x=uo, linestyle="--", lw=2, color="k")

        if i == 0:
            a.text(
                1e6,
                -2.25,
                (
                    "bkgd sil:    {:.2f}   on frac:  {:.2f}      ks:        {:.2f}   tlag
:  {:.2f}\n"
                    + "bprime:      {:.2f}   lambda:   {:.2f}   sigma_on:  {:.2f}   gamma
: {:.2f}\n"
                    + "mean_silent: {:.2f}   frac_silent: {:.2f}   mean_off: {:.2f}    si
gma_off: {:.2f}"
                ).format(bg, on, ks, tl, bp, lm, son, y, us, fs, uo, soff),
                fontdict={"fontfamily": "monospace"},
            )

    custom_lines = [
        Line2D([0], [0], color="tab:red", lw=4),
        Line2D([0], [0], color="tab:blue", lw=4),
    ]
    fig.axes[0].legend(
        custom_lines,
        ["Data", "Fit"],
        loc="upper left",
        #        bbox_to_anchor=(1.15, -7),
    )

    sns.despine(fig)

    fig.suptitle("pAXM" + str(plasmid).zfill(3) + ", " + descr)

    plt.tight_layout()
    # plt.show()

    param_df = pd.DataFrame.from_dict(
        {
            "plasmid": [list(plasmid_df["plasmid"])[0]],
            "bg": [bg],
            "on": [on],
            "ks": [ks],
            "tlag": [tl],
            "bprime": [bp],
            "lambda": [lm],
            "s_on": [son],
            "gamma": [y],
            "u_sil": [us],
            "f_sil": [fs],
            "u_off": [uo],
            "s_off": [soff],
        }
    )
    return [fig, param_df]


def fit_all() -> pd.DataFrame:
    df = import_repression_data()
    plasmid_list = sorted(list(set(list(df["plasmid"]))))
    # plasmid_list = [126, 217] + sorted([73, 80, 83, 84, 61])
    # plasmid_list = [140, 222]
    # plasmid_list = [84, 138]
    # plasmid_list = sorted([84, 138, 73, 80, 83, 61, 126, 217, 140, 222])

    def get_descr(p):
        return list(df[df["plasmid"] == p]["description"])[0]

    descr_list = list(map(get_descr, plasmid_list))

    print("Running telegraph model fits for", len(plasmid_list), "plasmids")
```

```
    # first, estimate background silencing
    nodox_rtetr_df = get_test_histogram_data(126, None, True)
    nodox_cits = list(nodox_rtetr_df["mCitrine-A"])
    nodox_cits = [n for n in nodox_cits if n > 0]
    nodox_cits = np.log10(nodox_cits)
    bg_frac = estimate_bg_frac(nodox_cits)

    dfs = []
    for i, p in tqdm(enumerate(plasmid_list), total=len(plasmid_list)):
        descr = descr_list[i]
        full_p_data = get_test_histogram_data(p, df)
        sample_size = np.min(list(full_p_data.groupby("day").count()["dox"])) - 1
        p_data = full_p_data.groupby("day").sample(n=sample_size)
        fig, pdf = fit_and_plot(p_data, None, bg_frac, None)
        fig.savefig("./plot_telegraph/" + descr + "_fit.pdf", bbox_inches="tight")
        dfs.append(pdf)
        plt.close(fig)

    param_df = pd.DataFrame(pd.concat(dfs))
    param_df.to_csv("./data/telegraph_parameters.csv")

    return param_df


def make_ks_comp_plot(param_df: pd.DataFrame):
    """
    make_ks_comp_plot Makes a plot comparing the telegraph model ks to the prior
    validation fits

    Args:
        param_df (pd.DataFrame): dataframe of ks and prior validation ks parameters

    Returns:
        mpl.Figure: makes plots, returns a Figure object
    """
    fig, ax = plt.subplots(figsize=(3, 3))

    plot_df = param_df.dropna(subset=["ks", "ks_validation"])

    _ = sns.regplot(
        data=plot_df,
        x="ks",
        y="ks_validation",
        scatter=False,
        line_kws={"color": "tab:red", "linestyle": "--", "zorder": -10},
    )

    _ = sns.scatterplot(
        data=plot_df,
        x="ks",
        y="ks_validation",
        color="white",
        edgecolor="tab:blue",
        s=40,
        linewidth=2,
        ax=ax,
    )

    r, p = st.pearsonr(plot_df["ks"], plot_df["ks_validation"])
    ax.text(0.1, 3.5, "Pearson\n$R={:.2f}$".format(r))
    # ax.set_xlim(0, 4.5)
    # ax.set_xticks([0, 2, 4])
    ax.set_xlabel("Telegraph Model $k_s$")
    # ax.set_ylim(0, 4.5)
    # ax.set_yticks([0, 2, 4])
    ax.set_ylabel("3-State Model $k_s$")

    return fig
```

```
def get_param_df() -> List[pd.DataFrame]:
    """
    get_param_df gets dataframe of validation parameter fits with telegraph model

    Note that if the file at `./data/telegraph_parameters.csv` exists, it will
    simply load that. To force re-calculation, delete the file.

    Returns:
        List[pd.DataFrame]:
            [Dataframe of parameter estimates, Dataframe of prior validation data]
    """
    if not os.path.exists("./data/telegraph_parameters.csv"):
        param_df = fit_all()
    param_df = pd.read_csv("./data/telegraph_parameters.csv")
    val_1_p_df = pd.read_csv("./data/paramed_concats.csv")
    val_2_p_df = pd.read_csv("./data/parameter_df_r372.csv")

    val_df = pd.concat([val_1_p_df, val_2_p_df])
    # val_df["validation_ks"] = val_df["ks"]
    # val_df["validation_tlag"] = val_df["tlag"]
    # val_df = val_df[["plasmid", "validation_ks", "validation_tlag"]]

    param_df = (
        param_df.set_index("plasmid")
        .join(val_df.set_index("plasmid"), how="left", rsuffix="_validation")
        .reset_index()
    )

    print(param_df)
    return [param_df, val_df]


def make_ks_screen_scatter(joined_df):
    """
    make_ks_screen_scatter Builds plot comparing ks to screen score

    Args:
        joined_df (_type_): dataframe containing ks and screen score data

    Returns:
        mpl.Figure: figure object of plot
    """
    fig, ax = plt.subplots(figsize=(3, 3))

    _ = sns.regplot(
        data=joined_df,
        x="avg_enrichment_d5",
        y="ks",
        scatter=False,
        line_kws={"color": "tab:red", "linestyle": "--", "zorder": -10},
    )

    _ = sns.scatterplot(
        data=joined_df,
        x="avg_enrichment_d5",
        y="ks",
        color="white",
        edgecolor="tab:blue",
        s=40,
        linewidth=2,
        ax=ax,
    )

    r, p = st.pearsonr(joined_df["avg_enrichment_d5"], joined_df["ks"])
    ax.text(-2.5, 3.35, "Pearson\nR$={:.2f}$".format(r))
    # ax.set_xlim(-5.5, 2.0)
    # ax.set_xticks([-4, -2, 0, 2])
    ax.set_xlabel("Screen $\log_2$(ON:OFF)")
    # ax.set_ylim(-0.2, 4.5)
    # ax.set_yticks([0, 2, 4])
```

```
    ax.set_ylabel("Telegraph Model $k_s$")

    fig.savefig("./plot_telegraph/ks_screen_comparison.pdf", bbox_inches="tight")

    return fig


if __name__ == "__main__":
    param_df, val_df = get_param_df()
    fig = make_ks_comp_plot(param_df)
    fig.savefig("./plot_telegraph/ks_comparison.pdf", bbox_inches="tight")
    plt.close(fig)

    screen_df = pd.read_csv(
        "../../../github_repo/fig_3/" + "01_repressor_additivity/pairs_baselinesums.csv"
    )
    d1g = list(screen_df["d1_Gene"])
    d2g = list(screen_df["d2_Gene"])
    dpairs = [str(d1) + " - " + str(d2) for d1, d2 in zip(d1g, d2g)]
    screen_df["description"] = dpairs

    joined_df = (
        param_df.set_index("description")
        .join(screen_df.set_index("description"), how="left", rsuffix="screen")
        .reset_index()
    )
    joined_df = joined_df.dropna(
        subset=["description", "avg_enrichment_d5", "ks_validation"]
    )
    joined_df = joined_df[~joined_df["composition"].str.contains("C")]
    f = make_ks_screen_scatter(joined_df)
    plt.close(f)
```