```python
"""Computational analysis for the proposed plant nickel biosensor.

This script implements the mechanistic ODE model described in the paper and
generates the dose-response, response-time, model-comparison, uncertainty, and
sensitivity outputs. All quantities are normalized design-study values rather
than fitted experimental measurements.
"""

from dataclasses import dataclass, replace
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from scipy.stats import spearmanr


ROOT = Path(__file__).resolve().parent
DATA_DIR = ROOT / "data"
FIGURE_DIR = ROOT / "figures"
DATA_DIR.mkdir(exist_ok=True)
FIGURE_DIR.mkdir(exist_ok=True)


@dataclass(frozen=True)
class Parameters:
    """Normalized baseline parameters for the design-study model."""

    k_in: float = 0.35
    k_out: float = 0.08
    k_seq: float = 0.05
    alpha_R: float = 1.0
    delta_R: float = 0.08
    k_f: float = 0.04
    k_r: float = 0.15
    delta_C: float = 0.06
    K_C: float = 5.0
    h: float = 2.0
    alpha_0: float = 3.0e-4
    alpha_1: float = 1.0
    delta_m: float = 0.70
    beta: float = 1.0
    k_mat: float = 0.45
    delta_G: float = 0.08
    sensitivity: float = 1.0
    background: float = 0.40


BASE = Parameters()


def promoter_occupancy(active_nikr: np.ndarray | float, p: Parameters):
    """Hill occupancy of the synthetic promoter by active NikR."""
    numerator = np.asarray(active_nikr) ** p.h
    return numerator / (p.K_C**p.h + numerator)


def ode_system(_time, state, nickel_external, p):
    """Right-hand side of the coupled nickel-sensor/reporter model."""
    nickel_internal, nikr, active_nikr, mrna, immature, mature = state
    occupancy = promoter_occupancy(active_nikr, p)

    return (
        p.k_in * nickel_external
        - (p.k_out + p.k_seq) * nickel_internal,
        p.alpha_R
        - p.delta_R * nikr
        - p.k_f * nikr * nickel_internal
        + p.k_r * active_nikr,
        p.k_f * nikr * nickel_internal
        - (p.k_r + p.delta_C) * active_nikr,
        p.alpha_0 + p.alpha_1 * occupancy - p.delta_m * mrna,
        p.beta * mrna - (p.k_mat + p.delta_G) * immature,
        p.k_mat * immature - p.delta_G * mature,
    )


def simulate(nickel_external, times, p=BASE):
    """Simulate one fixed external-nickel exposure condition."""
    times = np.asarray(times, dtype=float)
    if np.any(np.diff(times) < 0) or times[0] < 0:
        raise ValueError("times must be nonnegative and sorted")

    # NikR is constitutively expressed before exposure; reporter states begin at
    # zero so that maturation and accumulation remain visible in the time course.
```

```
    initial_state = np.array([0.0, p.alpha_R / p.delta_R, 0.0, 0.0, 0.0, 0.0])
    requested = np.unique(np.concatenate(([0.0], times)))
    solution = solve_ivp(
        ode_system,
        (0.0, float(requested[-1])),
        initial_state,
        args=(float(nickel_external), p),
        t_eval=requested,
        method="LSODA",
        rtol=1e-7,
        atol=1e-9,
    )
    if not solution.success:
        raise RuntimeError(solution.message)

    lookup = {time: i for i, time in enumerate(solution.t)}
    indices = [lookup[time] for time in times]
    states = solution.y[:, indices].T
    occupancy = promoter_occupancy(states[:, 2], p)
    fluorescence = p.sensitivity * states[:, 5] + p.background
    return states, occupancy, fluorescence


def make_dose_response(p=BASE):
    """Evaluate fluorescence over dose and sampling time."""
    nickel = np.concatenate(([0.0], np.logspace(-2, 2.3, 100)))
    sample_times = np.array([6.0, 12.0, 24.0, 48.0])
    rows = []

    _, _, baseline = simulate(0.0, sample_times, p)
    for dose in nickel:
        _, occupancy, fluorescence = simulate(dose, sample_times, p)
        for time, signal, promoter, blank in zip(
            sample_times, fluorescence, occupancy, baseline
        ):
            rows.append(
                {
                    "N_e": dose,
                    "time_h": time,
                    "F": signal,
                    "P": promoter,
                    "fold_over_baseline": signal / blank,
                }
            )

    frame = pd.DataFrame(rows)
    frame.to_csv(DATA_DIR / "simulated_dose_response.csv", index=False)
    return frame


def make_time_courses(p=BASE):
    """Save dense trajectories used to estimate response half-times."""
    doses = np.array([0.0, 0.3, 1.0, 3.0, 10.0, 30.0, 100.0])
    times = np.linspace(0.0, 48.0, 481)
    rows = []

    for dose in doses:
        states, occupancy, fluorescence = simulate(dose, times, p)
        for time, state, promoter, signal in zip(
            times, states, occupancy, fluorescence
        ):
            rows.append(
                {
                    "N_e": dose,
                    "time_h": time,
                    "N_i": state[0],
                    "R": state[1],
                    "C": state[2],
                    "P": promoter,
                    "m": state[3],
                    "G_u": state[4],
                    "G": state[5],
                    "F": signal,
                }
            )

    frame = pd.DataFrame(rows)
    frame.to_csv(DATA_DIR / "simulated_time_courses.csv", index=False)
    return frame


def calculate_half_times(time_courses, p=BASE):
    """Find the time needed to reach half of the 48-hour signal increase."""
    rows = []
    for dose, group in time_courses.groupby("N_e"):
        if dose == 0:
            continue
```

```
        group = group.sort_values("time_h")
        final_signal = group["F"].iloc[-1]
        target = p.background + 0.5 * (final_signal - p.background)
        signals = group["F"].to_numpy()
        times = group["time_h"].to_numpy()
        crossing = np.flatnonzero(signals >= target)[0]
        half_time = np.interp(
            target,
            signals[max(crossing - 1, 0) : crossing + 1],
            times[max(crossing - 1, 0) : crossing + 1],
        )
        rows.append({"N_e": dose, "F_48h": final_signal, "t_half_h": half_time})

    frame = pd.DataFrame(rows)
    frame.to_csv(DATA_DIR / "response_half_times.csv", index=False)
    return frame


def compare_with_instantaneous_hill(dose_response):
    """Compare the 24-hour ODE prediction with a direct Hill mapping."""
    doses = np.logspace(-2, 2.3, 140)
    at_24h = dose_response.query("time_h == 24").sort_values("N_e")
    mechanistic = np.interp(doses, at_24h["N_e"], at_24h["fold_over_baseline"])
    instantaneous = 1.0 + 13.0 * doses**2 / (3.0**2 + doses**2)

    frame = pd.DataFrame(
        {
            "N_e": doses,
            "mechanistic_24h": mechanistic,
            "instant_hill": instantaneous,
            "ratio_mech_to_hill": mechanistic / instantaneous,
        }
    )
    frame.to_csv(DATA_DIR / "model_comparison.csv", index=False)
    return frame


def fold_response(dose, p=BASE, time=24.0):
    """Return fluorescence normalized to the matched zero-nickel control."""
    _, _, signal = simulate(dose, [time], p)
    _, _, blank = simulate(0.0, [time], p)
    return float(signal[0] / blank[0])


def local_sensitivity(p=BASE):
    """Perturb selected parameters one at a time by 0.5x and 2x."""
    names = ["k_in", "k_seq", "K_C", "alpha_1", "k_mat", "delta_G"]
    baseline = fold_response(10.0, p)
    rows = []
    for name in names:
        for factor in (0.5, 2.0):
            changed = replace(p, **{name: getattr(p, name) * factor})
            response = fold_response(10.0, changed)
            rows.append(
                {
                    "parameter": name,
                    "factor": factor,
                    "fold_response_24h": response,
                    "relative_to_baseline": response / baseline,
                }
            )

    frame = pd.DataFrame(rows)
    frame.to_csv(DATA_DIR / "local_sensitivity.csv", index=False)
    return frame


def monte_carlo_analysis(p=BASE, samples=350, seed=2026):
    """Propagate parameter uncertainty and calculate rank sensitivities."""
    rng = np.random.default_rng(seed)
    names = ["k_in", "k_seq", "K_C", "alpha_1", "k_mat", "delta_G"]
    doses = [1.0, 3.0, 10.0, 30.0]
    rows = []

    # A lognormal distribution keeps rates positive. sigma=0.4 corresponds to
    # roughly a 2.2-fold interval on either side of the median for 95% of draws.
    draws = {
        name: getattr(p, name) * np.exp(rng.normal(0.0, 0.4, samples))
        for name in names
    }
    for sample in range(samples):
        sampled_values = {name: draws[name][sample] for name in names}
        sampled_parameters = replace(p, **sampled_values)
        for dose in doses:
            row = {"sample": sample, "N_e": dose}
            row.update({f"param_{key}": value for key, value in sampled_values.items()})
            row["fold_response_24h"] = fold_response(dose, sampled_parameters)
```

```
            rows.append(row)

    uncertainty = pd.DataFrame(rows)
    uncertainty.to_csv(DATA_DIR / "monte_carlo_uncertainty.csv", index=False)

    rankings = []
    for dose, group in uncertainty.groupby("N_e"):
        for name in names:
            rho, _ = spearmanr(group[f"param_{name}"], group["fold_response_24h"])
            rankings.append({"N_e": dose, "parameter": name, "spearman_rho": rho})

    sensitivity = pd.DataFrame(rankings)
    sensitivity.to_csv(DATA_DIR / "global_sensitivity_rankings.csv", index=False)
    return uncertainty, sensitivity


def plot_dose_response(frame):
    """Plot fold response at each sampling time."""
    fig, ax = plt.subplots(figsize=(7.2, 4.8))
    for time, group in frame.groupby("time_h"):
        positive = group[group["N_e"] > 0]
        ax.semilogx(positive["N_e"], positive["fold_over_baseline"], label=f"{time:g} h")
    ax.set(xlabel="External nickel, normalized units", ylabel="Fold fluorescence over baseline")
    ax.legend(title="Sampling time", frameon=False)
    ax.grid(alpha=0.25)
    fig.tight_layout()
    fig.savefig(FIGURE_DIR / "dose_response.png", dpi=300)
    plt.close(fig)


def plot_model_comparison(frame):
    """Plot the dynamic and instantaneous calibration curves."""
    fig, ax = plt.subplots(figsize=(7.2, 4.8))
    ax.semilogx(frame["N_e"], frame["mechanistic_24h"], label="Mechanistic model, 24 h", lw=2)
    ax.semilogx(frame["N_e"], frame["instant_hill"], label="Instantaneous Hill model", lw=2, ls="--")
    ax.set(xlabel="External nickel, normalized units", ylabel="Fold fluorescence over baseline")
    ax.legend(frameon=False)
    ax.grid(alpha=0.25)
    fig.tight_layout()
    fig.savefig(FIGURE_DIR / "model_comparison.png", dpi=300)
    plt.close(fig)


def plot_uncertainty_and_sensitivity(uncertainty, sensitivity):
    """Plot Monte Carlo distributions and Spearman rank correlations."""
    fig, axes = plt.subplots(1, 2, figsize=(10.5, 4.5))
    groups = [g["fold_response_24h"].to_numpy() for _, g in uncertainty.groupby("N_e")]
    labels = [f"{dose:g}" for dose in sorted(uncertainty["N_e"].unique())]
    axes[0].boxplot(groups, tick_labels=labels, showfliers=False)
    axes[0].set(xlabel="External nickel", ylabel="24 h fold response", title="Parameter uncertainty")

    selected = sensitivity.query("N_e == 1").sort_values("spearman_rho")
    colors = ["#b04a4a" if value < 0 else "#3f7fba" for value in selected["spearman_rho"]]
    axes[1].barh(selected["parameter"], selected["spearman_rho"], color=colors)
    axes[1].axvline(0, color="black", lw=0.8)
    axes[1].set(xlabel="Spearman rank correlation", title="Global sensitivity at $N_e=1$")
    for ax in axes:
        ax.grid(axis="x", alpha=0.2)
    fig.tight_layout()
    fig.savefig(FIGURE_DIR / "uncertainty_sensitivity.png", dpi=300)
    plt.close(fig)


def main():
    """Run the full reproducible analysis pipeline."""
    dose_response = make_dose_response()
    time_courses = make_time_courses()
    calculate_half_times(time_courses)
    comparison = compare_with_instantaneous_hill(dose_response)
    local_sensitivity()
    uncertainty, sensitivity = monte_carlo_analysis()

    plot_dose_response(dose_response)
    plot_model_comparison(comparison)
    plot_uncertainty_and_sensitivity(uncertainty, sensitivity)
    print("Analysis complete: CSV files are in data/ and figures are in figures/.")


if __name__ == "__main__":
    main()
```