06 November, 2026

BERN01:
Cellular reprogramming

Joseph A. Binns (jo4752bi-s@lu.se)
Group 15

Contents

1 Introduction 1
2 Theory 1
3 Methodology 2
4 Results and discussion 3
5 Conclusion 5
6 AI usage 6
7 References 6
8 Appendix 6

1 Introduction

Pluripotent stem cells can develop into essentially any cell type in the body, including neurons, muscle cells and liver cells. This versatility makes them medicinally valuable in both research and treatment. Abundant differentiated cells, such as skin cells, can be reprogrammed into induced pluripotent stem cells by altering the activity of genes that control cell identity.

In this report, we use a simple model of a gene regulatory network to explore the mechanisms underlying cellular reprogramming. Our aim is to determine which temporary interventions establish a stable model state associated with pluripotency that persists after stimulation is removed. We compare interventions targeting each aspect of the gene regulatory network, examining the role of regulatory feedback with different numerical methods.

2 Theory

Our gene regulatory model comprises proteins which help to regulate cell identity [1]. These proteins are Nanog, Oct4 and Tet1, with respective concentrations N, O and T. Nanog and Tet1 bind to form a protein complex C, whose concentration is assumed to adjust instantaneously to the binding equilibrium set by the current Nanog and Tet1 concentrations. Its concentration and the two activation functions are

$$C = S - \sqrt{S^2 - NT}, \quad S = \frac{K_d + N + T}{2}, \tag{1}$$

$$A(C) = \frac{\left(\frac{C}{K_{NT}}\right)^n}{1 + \left(\frac{C}{K_{NT}}\right)^n}, \quad f_O = \frac{O}{K_O + O}. \tag{2}$$

The equations implemented in our simulations are

$$\begin{aligned} \frac{dN}{dt} &= N_{\text{over}} + \text{LIF} + p_N f_O - N, \\ \frac{dO}{dt} &= O_{\text{over}} + \text{LIF} + p_O A(C) f_O - O, \\ \frac{dT}{dt} &= T_{\text{over}} + p_T A(C) f_O - T. \end{aligned} \tag{3}$$

Here, leukaemia inhibitory factor (LIF) is a signalling protein which supports pluripotency [1]. The promoter activities p_N, p_O and p_T together with f_O set the maximum production rate of the respective proteins, and the overexpressions $N_{\text{over}}, O_{\text{over}}$ and T_{over} artificially increase production of the respective proteins beyond the baseline. Since time is scaled such that all decay coefficients equal one, dimensionless quantities are given to avoid misinterpretation. The gene regulatory network is illustrated in Figure 1.

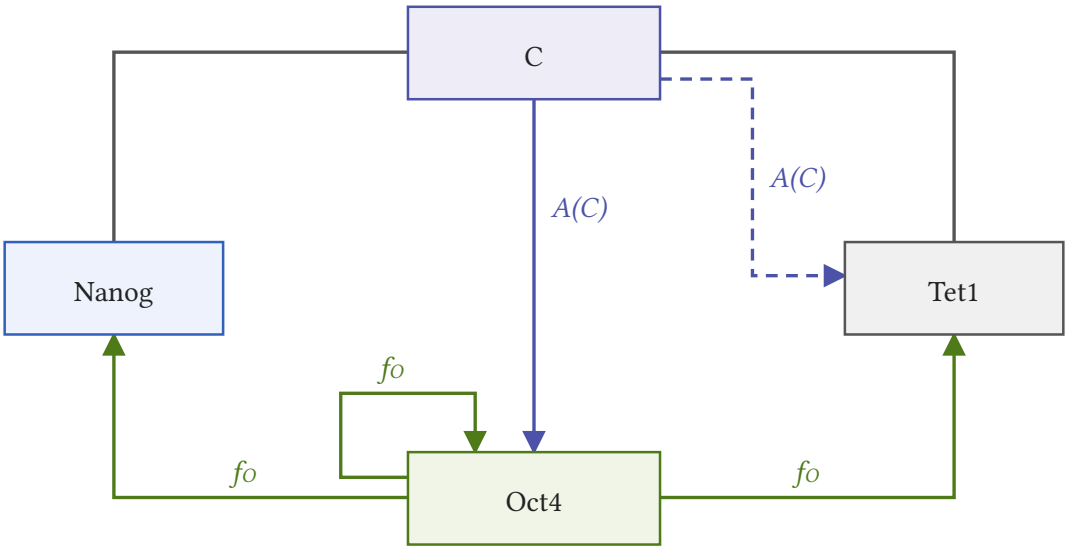


Figure 1: Transcription factor circuit.

3 Methodology

3.1 Parameters and simulation protocol

Each simulation follows an OFF-ON-OFF sequence to examine how the gene regulatory network responds to the overexpression of a selected protein and its subsequent removal. Each phase runs for 20 dimensionless time units, a duration which has been empirically deemed sufficient for protein concentrations to approach a steady state in the cases considered. The dimensionless initial protein concentrations are $(N, O, T) = (0, 0, 0)$. We solve the intervals separately so that an integration step does not cross a change in input.

Condition	N_{over}	O_{over}	T_{over}	**LIF**
OFF	0	0	0.05	0
Oct4 ON	0	0.3	0.05	0.06
Nanog ON	0.3	0	0.05	0
Tet1 ON	0	0	0.3	0.06

Table 1: Conditional model parameters.

The parameters were chosen to reproduce the results reported by Olariu et al. [1]. Here, $T_{\text{over}} = 0.05$ represents background production of Tet1, and LIF is absent during the Nanog ON condition. All other parameters are fixed as shown in Table 2.

$p_N = p_O = p_T$	**Decay rates**	n	K_d	K_O	K_{NT}
1	1	2	0.1	0.3	0.2

Table 2: Model constants.

3.2 Forward Euler

Let $y = (N, O, T)$ denote the protein concentrations. The derivative dy/dt then describes their rates of change as calculated from Equation 3. The forward Euler method estimates changes over discrete time steps h using the derivative evaluated at the current state y_k. Each step introduces a local error of order $O(h^2)$, which accumulates to a global error of order $O(h)$ over the interval. The approximation is given by the update rule

$$y_{k+1} = y_k + h\left(\frac{dy}{dt}\right)_k. \tag{4}$$

3.3 Runge-Kutta 45

We compared the results of the forward Euler method with those of the higher-order Runge–Kutta 45 (RK45) method. Each accepted step uses a fifth-order approximation. A fourth-order approximation is calculated alongside it, and the difference is used to estimate the local error and adjust the step size accordingly to balance accuracy and computation time [2].

We used SciPy's `scipy.integrate.solve_ivp` function with `method="RK45"` [3]. We set `max_step=0.1` and retained the default error tolerances, `rtol=1e-3` and `atol=1e-6`.

3.4 Error estimation

We used Richardson extrapolation to estimate the global error at the end of each interval. For each simulation, we compared runs using $h = 0.1$ and $2h = 0.2$ to estimate the global error at time t:

$$\varepsilon_{h(t)} \approx \frac{\left|y_{h(t)} - y_{2h}(t)\right|}{2^p - 1}, \tag{5}$$

where $y = (N, O, T)$ and the absolute value is taken componentwise. The parameter p is the order of accuracy of the numerical method such that $p = 1$ for forward Euler and $p = 5$ for RK45. We report the largest of the three error estimates. The error estimation assumes the leading error scales as h^p. For adaptive RK45, changing the maximum step size does not guarantee this scaling, so the estimate is only indicative.

3.5 Varying LIF during Nanog overexpression

We repeated the Nanog experiment at various LIF values during the ON phase in order to determine whether LIF can enable persistent pluripotency after Nanog stimulation. Each OFF-ON-OFF phase was extended from 20 to 40 dimensionless time units to allow sufficient time for the system to equilibrate.

4 Results and discussion

4.1 Response to temporary overexpression

Both integrators produce the same qualitative responses as illustrated in Figure 2. During the initial OFF interval, Nanog and Oct4 remain zero, while Tet1 approaches 0.05 as a result of its baseline production. Oct4 and Tet1 overexpression are sufficient to trigger a positive feedback loop in the gene regulatory network, causing the pluripotent state to remain elevated after the stimulation is removed. However, Nanog overexpression produces only a temporary rise in Nanog, followed by a return towards the initial state.

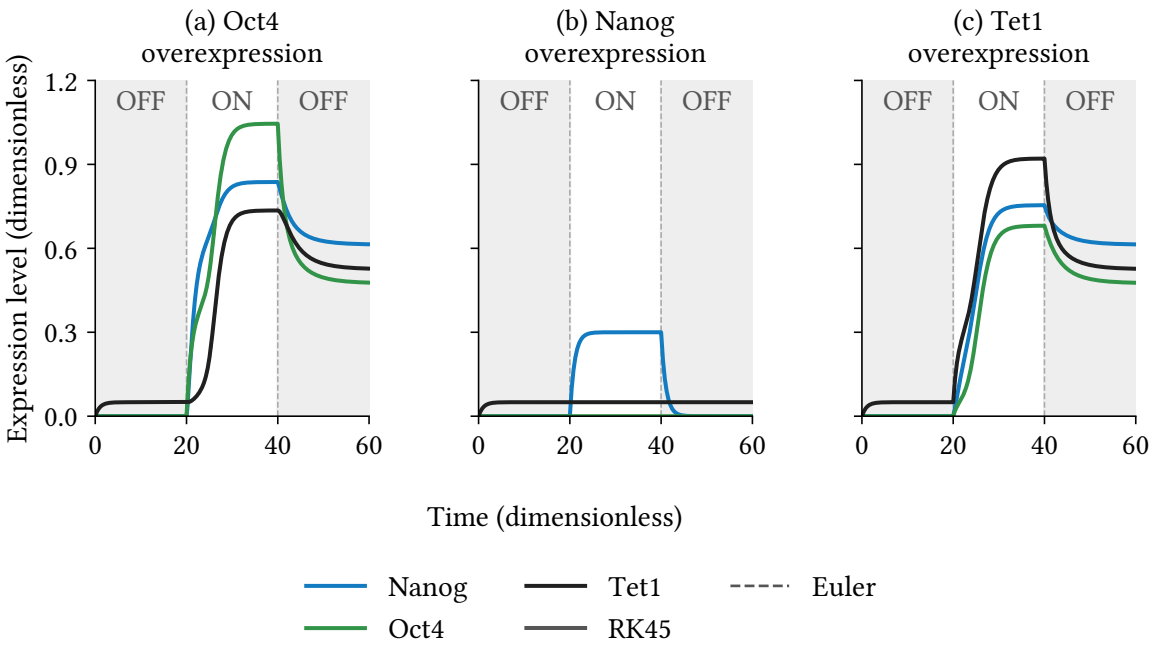


Figure 2: Numerical simulations of protein expression levels with different stimuli.

Table 3, Table 4 and Table 5 show the estimated accumulated error at the end of each OFF or ON interval, taking the largest absolute estimate among the three protein concentrations.

Intervention	$t = 20$	$t = 40$	$t = 60$
Oct4	2.51×10^{-11}	9.69×10^{-6}	1.35×10^{-4}
Nanog	2.51×10^{-11}	1.51×10^{-10}	1.51×10^{-10}
Tet1	2.51×10^{-11}	2.22×10^{-5}	9.66×10^{-5}

Table 3: Forward Euler global error estimates.

Intervention	$t = 20$	$t = 40$	$t = 60$
Oct4	8.06×10^{-18}	1.32×10^{-12}	7.06×10^{-13}
Nanog	8.06×10^{-18}	4.66×10^{-17}	4.76×10^{-17}
Tet1	8.06×10^{-18}	6.05×10^{-14}	3.93×10^{-13}

Table 4: RK45 global error estimates.

Oct4 directly raises its own concentration and activates Nanog. The increased Nanog-Tet1 complex then reinforces Oct4 and Tet1 production, sustaining a positive feedback loop. In the Tet1 scenario, LIF supplies initial Nanog and Oct4 production, allowing additional Tet1 to activate the same feedback loop. The Oct4 and Tet1 overexpressions approach similar elevated expression levels after their removal.

4.2 Varying LIF during Nanog overexpression

Figure 3 shows that a raised LIF level during Nanog stimulation can produce a persistent pluripotent state. With LIF $= 0$, Oct4 remains zero and Nanog decays after withdrawal. LIF $= 0.06$ induces some Oct4 and Tet1 production during stimulation, but the response still returns to the low state without activating the feedback loop. Further increasing LIF to 0.11 supplies enough Oct4 to allow the Nanog-Tet1 complex to reinforce Oct4 and Tet1 production, sustaining the positive feedback loop. The failure of Nanog stimulation alone follows directly from the equations. With $O = 0$, $O_{\text{over}} = 0$ and LIF $= 0$,

the activation f_O remains zero, preventing any increase in Oct4 or Tet1 and thus preventing activation of the feedback loop.

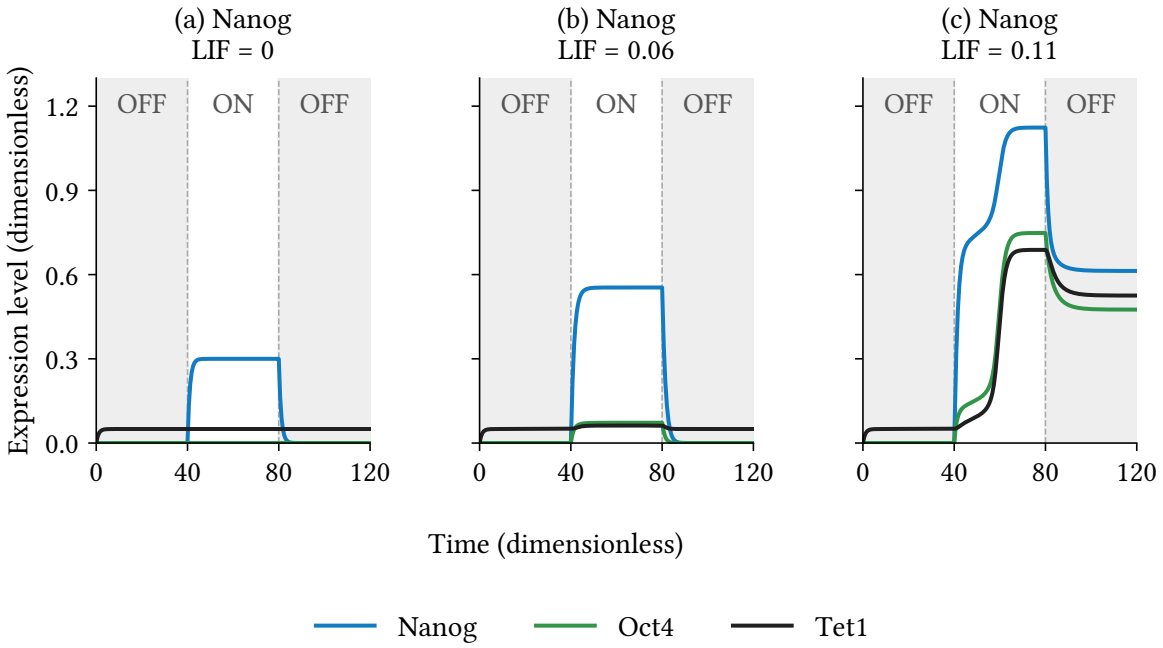


Figure 3: Nanog overexpression with varying LIF.

LIF during ON	$t = 40$	$t = 80$	$t = 120$
0	6.72×10^{-19}	5.37×10^{-18}	6.72×10^{-19}
0.06	6.72×10^{-19}	6.72×10^{-19}	6.72×10^{-19}
0.11	6.72×10^{-19}	2.05×10^{-14}	1.01×10^{-14}

Table 5: RK45 error estimates for Nanog overexpression with varying LIF.

5 Conclusion

The Nanog-Oct4-Tet1 gene regulatory network illustrates how positive feedback can maintain elevated protein concentrations associated with pluripotency in the model after a temporary intervention. Oct4 promotes Nanog production, while the Nanog-Tet1 complex and Oct4 together reinforce Oct4 and Tet1 production, allowing the network to sustain its own activity. Both numerical methods predict persistently elevated expression after Oct4 or Tet1 stimulation with LIF, with RK45 yielding smaller estimated global errors. Under the tested conditions, Nanog stimulation without LIF does not establish sustained expression, whereas stimulation with LIF $= 0.11$ maintains elevated expression throughout the final OFF interval. The RK45 error estimates remain indicative because changing the maximum step size does not guarantee the scaling assumed by Richardson extrapolation.

6 AI usage

The initial core of our code using the RK45 method was implemented by hand, whilst the later addition of the forward Euler method was implemented with the assistance of Codex [4]. The figures in this report were also created and formatted with the use of Codex and underwent several iterations. The risk of poor methodology was mitigated by separating simulation from visualization.

7 References

[1] V. Olariu, C. Lövkvist, and K. Sneppen, "Nanog, Oct4 and Tet1 interplay in establishing pluripotency," *Scientific Reports*, vol. 6, p. 25438, 2016, doi: 10.1038/srep25438.

[2] SciPy developers, "RK45: Explicit Runge–Kutta method of order 5(4)." Accessed: Oct. 25, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.RK45.html

[3] SciPy developers, "scipy.integrate.solve_ivp — SciPy documentation." Accessed: Nov. 01, 2026. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

[4] OpenAI, "Codex." Accessed: Oct. 14, 2026. [Online]. Available: https://developers.openai.com/learn/codex

8 Appendix

Python

```python
import csv
from pathlib import Path
import numpy

RESULTS_DIR = Path(__file__).resolve().parents[1] / "results"

class parameters:
    def __init__(self, N_over, O_over, T_over, Lif):
        self.N_over = N_over
        self.O_over = O_over
        self.T_over = T_over
        self.Lif = Lif
        self.p_N = 1
        self.p_O = 1
        self.p_T = 1
        self.n = 2
        self.K_d = 0.1
        self.K_0 = 0.3
        self.K_NT = 0.2

p_mef = parameters(0, 0, 0.05, 0)
p_nanog = parameters(0.3, 0, 0.05, 0)
p_oct4 = parameters(0, 0.3, 0.05, 0.06)
p_tet1 = parameters(0, 0, 0.3, 0.06)

def slow_gene_regulation(t, y, p):
    [N_total, O_total, T_total] = y
```

```
28
29      NT = (p.K_d + N_total + T_total) / 2 - numpy.sqrt(numpy.square(p.K_d +
        N_total + T_total) / 4 - N_total * T_total)
30      A = numpy.power(NT / p.K_NT, p.n) / (1 + numpy.power(NT / p.K_NT, p.n))
31      f_O = (O_total / p.K_0) / (1 + O_total / p.K_0)
32
33      d_N_total = p.N_over + p.Lif + p.p_N * f_O - N_total
34      d_O_total = p.O_over + p.Lif + p.p_O * A * f_O - O_total
35      d_T_total = p.T_over + p.p_T * A * f_O - T_total
36
37      return [d_N_total, d_O_total, d_T_total]
38
39  def solve_with_error(
40      integrate, order, params, initial_conditions, t_max=20, h=0.1,
41      *, coarse_initial_conditions=None,
42  ):
43      if coarse_initial_conditions is None:
44          coarse_initial_conditions = initial_conditions
45      sol_h = integrate(params, initial_conditions, t_max, h)
46      sol_2h = integrate(params, coarse_initial_conditions, t_max, 2 * h)
47      difference = sol_h(t_max) - sol_2h(t_max)
48      global_error = difference / (2**order - 1)
49      return (sol_h, sol_2h, global_error)
50
51
52  def run_simulations(solve, method, results_dir=RESULTS_DIR, t_max=20, h=0.1):
53      results_dir = Path(results_dir)
54      results_dir.mkdir(parents=True, exist_ok=True)
55      t = numpy.linspace(0, t_max, 100)
56      sol0, sol0_2h, err0 = solve(p_mef, [0, 0, 0], t_max=t_max, h=h)
57      error_rows = []
58
59      for params, gene in ((p_oct4, "oct4"), (p_nanog, "nanog"), (p_tet1,
        "tet1")):
60          sol1, sol1_2h, err1 = solve(
61              params, sol0(t_max), t_max=t_max, h=h,
62              coarse_initial_conditions=sol0_2h(t_max),
63          )
64          sol2, _, err2 = solve(
65              p_mef, sol1(t_max), t_max=t_max, h=h,
66              coarse_initial_conditions=sol1_2h(t_max),
67          )
68          data = numpy.vstack([
69              numpy.column_stack((t + start_time, solution(t).T))
70              for solution, start_time in ((sol0, 0), (sol1, t_max), (sol2, 2 *
                t_max))
```

```
        ])
        filename = results_dir / f"cell-{method}_{gene}.csv"
        numpy.savetxt(filename, data, delimiter=",",
        header="time,Nanog,Oct4,Tet1", comments="")
        print(f"{filename.name}: accumulated endpoint error estimates {err0},
        {err1}, {err2}")
        for phase, endpoint, error in zip(
            ("OFF", "ON", "OFF"), (t_max, 2 * t_max, 3 * t_max), (err0, err1,
            err2)
        ):
            error_rows.append((gene, phase, endpoint, *error))

    with (results_dir / f"cell-{method}-errors.csv").open("w", newline="") as
    output:
        writer = csv.writer(output)
        writer.writerow(("scenario", "phase", "time", "Nanog", "Oct4", "Tet1"))
        writer.writerows(error_rows)
```

Code 1: Gene regulatory model, simulation protocol and error estimation.

```python
import numpy
from cell import slow_gene_regulation, solve_with_error, run_simulations

def integrate(params, initial_conditions, t_max, h):
    t = numpy.append(numpy.arange(0, t_max, h), t_max)
    y = numpy.empty((len(initial_conditions), len(t)))
    y[:, 0] = initial_conditions
    for i, step in enumerate(numpy.diff(t)):
        derivative = numpy.asarray(slow_gene_regulation(t[i], y[:, i], params))
        y[:, i + 1] = y[:, i] + step * derivative

    def solution(time):
        return numpy.array([numpy.interp(time, t, component) for component in
        y])

    return solution

def solve(params, initial_conditions, t_max=20, h=0.1, *,
coarse_initial_conditions=None):
    return solve_with_error(
        integrate, 1, params, initial_conditions, t_max, h,
        coarse_initial_conditions=coarse_initial_conditions,
    )

run_simulations(solve, "euler")
```

Code 2: Forward Euler integration.

```python
from scipy.integrate import solve_ivp
```

```
from cell import slow_gene_regulation, solve_with_error, run_simulations

def integrate(params, initial_conditions, t_max, h):
    solution = solve_ivp(
        fun=slow_gene_regulation,
        t_span=(0, t_max),
        y0=initial_conditions,
        method="RK45",
        args=(params,),
        dense_output=True,
        max_step=h,
    )
    return solution.sol

def solve(params, initial_conditions, t_max=20, h=0.1, *, coarse_initial_conditions=None):
    return solve_with_error(
        integrate, 5, params, initial_conditions, t_max, h,
        coarse_initial_conditions=coarse_initial_conditions,
    )

run_simulations(solve, "rk45")
```

Code 3: RK45 integration.