Computational Methods

Assignment 3

Rasmus Hammar

2025-11-27

Imports

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
from time import process_time
```

a)

Reproducing Figure 2, a and b from the article.

```
def ode_rhs(t, y, alpha, beta, gamma, delta, theta):
    # Unpacking y
    D_A = y[0]  # dD_A/dt
    D_R = y[1]  # dD_R/dt
    Dprim_A = y[2]  # dDprim_A/dt
    Dprim_R = y[3]  # dDprim_R/dt
    M_A = y[4]  # dM_A/dt
    A = y[5]  # dA/dt
    M_R = y[6]  # dM_R/dt
    R = y[7]  # dR/dt
    C = y[8]  # dC/dt

    # Unpacking constants
    a_A, a_R, a_prim_A, a_prim_R = alpha
    b_A, b_R = beta
    g_A, g_R, g_C = gamma
    d_A, d_R, d_MA, d_MR = delta
    t_A, t_R = theta

    # Equation system
    yt = [
        t_A * Dprim_A - g_A * D_A * A,  # dD_A/dt
```

```
        t_R * Dprim_R - g_R * D_R * A,  # dD_R/dt
        g_A * D_A * A - t_A * Dprim_A,  # dDprim_A/dt
        g_R * D_R * A - t_R * Dprim_R,  # dDprim_R/dt
        a_prim_A * Dprim_A + a_A * D_A - d_MA * M_A,  # dM_A/dt
        b_A * M_A
        + t_A * Dprim_A
        + t_R * Dprim_R
        - A * (g_A * D_A + g_R * D_R + g_C * R + d_A),  # dA/dt
        a_prim_R * Dprim_R + a_R * D_R - d_MR * M_R,  # dM_R/dt
        b_R * M_R - g_C * A * R + d_A * C - d_R * R,  # dR/dt
        g_C * A * R - d_A * C,  # dC/dt
    ]

    return yt
```

```
# Constants
alpha = [
    50,  # a_A
    0.01,  # a_R
    500,  # a_prim_A
    50,  # a_prim_R
]
beta = [
    50,  # b_A
    5,  # b_R
]
gamma = [
    1,  # g_A
    1,  # g_R
    2,  # g_C
]
delta = [
    1,  # d_A
    0.2,  # d_R
    10,  # d_MA
    0.5,  # d_MR
]
theta = [
    50,  # t_A
    100,  # t_R
]

# Time span
t_0 = 0
t_stop = 400
times = np.arange(t_0, t_stop, 0.1)
```

```python
# y-initial
y_0 = [
    1,  # D_A
    1,  # D_R
    0,  # Dprim_A
    0,  # Dprim_R
    0,  # M_A
    0,  # A
    0,  # M_R
    0,  # R
    0,  # C
]
```

```python
sol = solve_ivp(
    ode_rhs,
    (t_0, t_stop),
    y_0,
    args=(alpha, beta, gamma, delta, theta),
    t_eval=times,
)
```

```python
plt.figure(1)
plt.subplot(2, 1, 1)
plt.plot(sol.t, sol.y[5], color="blue")
plt.ylabel("A")

plt.subplot(2, 1, 2)
plt.plot(sol.t, sol.y[7], color="orange")
plt.ylabel("R")
plt.xlabel("Time [hr]")

plt.show()
```

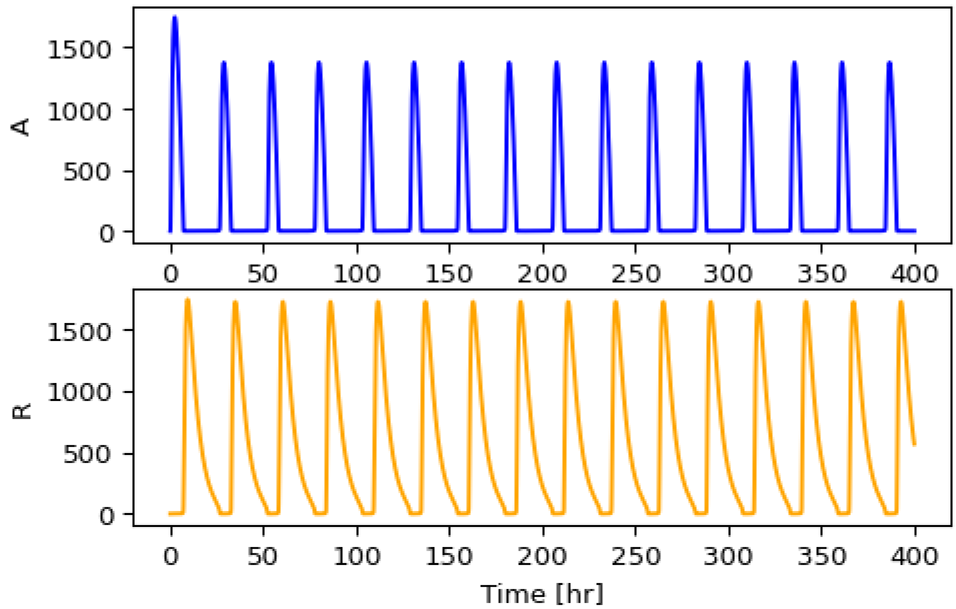

Figure 1: Recreation of Figure 2 a & b from research article.

b)

Benchmarking of explicit (RK45) vs implicit (BDF) methods to determine if this is a stiff problem.

```
## Iterations
n = 20
## Empty time vectors
time_explicit = np.zeros(n)
time_implicit = np.zeros(n)

## Benchmark methods
for i in range(n):
    ## Explicit method (RK45)
    start_time = process_time()
    solve_ivp(
        ode_rhs,
        (t_0, t_stop),
        y_0,
        args=(alpha, beta, gamma, delta, theta),
        t_eval=times,
    )
    time_explicit[i] = process_time() - start_time
    ## Implicit method (BDF)
    start_time = process_time()
    solve_ivp(
        ode_rhs,
        (t_0, t_stop),
```

```
        y_0,
        args=(alpha, beta, gamma, delta, theta),
        t_eval=times,
        method="BDF",  # Implicit solver for stiff ODEs
    )
    time_implicit[i] = process_time() - start_time
```

```
## Plot
plt.figure(2)
plt.plot(
    range(1, n + 1),
    time_explicit,
    label="Explicit (RK45)",
    color="purple",
)
plt.plot(
    range(1, n + 1),
    time_implicit,
    label="Implicit (BDF)",
    color="green",
)
plt.legend()
plt.xlabel("Iterations")
plt.ylabel("Time [s]")
plt.show()
```

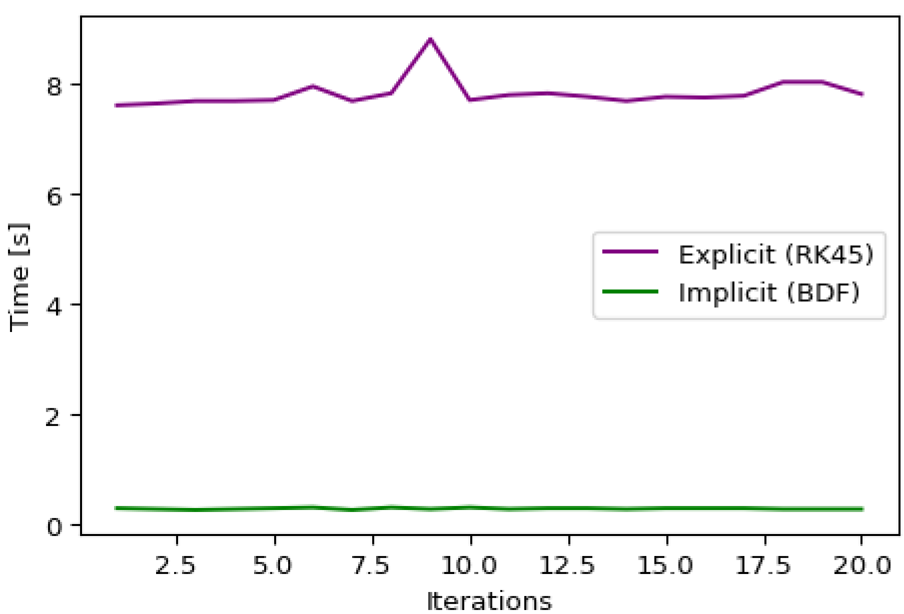

Figure 2: Processing time for solving the ODE using explicit vs implicit methods. The implicit method consistently performed considerably better.