Numerical Analysis Computational Assignment C

Name: Nate Stemen (20906566)
Email: nate.stemen@uwaterloo.ca
Due: Mon, Dec 7, 2020 5:00 PM
Course: AMATH 740

Problem 3. Adaptive RK45 method.

Solution. First, the code. Note: I put everything in one file because it was simple enough that breaking it into multiple files felt unnecessary, and overcomplicated.

adaptiveRK.py

```
import numpy as np
import matplotlib.pyplot as plt

plt.style.use("ggplot")


alpha = [0, 1 / 4, 3 / 8, 12 / 13, 1, 1 / 2]
beta = [
    [0, 1 / 4, 3 / 32, 1932 / 2197, 439 / 216, -8 / 27],
    [0, 0, 9 / 32, -7200 / 2197, -8, 2],
    [0, 0, 0, 7296 / 2197, 3680 / 513, -3544 / 2565],
    [0, 0, 0, 0, -845 / 4104, 1859 / 4104],
    [0, 0, 0, 0, 0, -11 / 40],
    [0, 0, 0, 0, 0, 0],
]
wA = [25 / 216, 0, 1408 / 2565, 2197 / 4104, -1 / 5, 0]
wB = [16 / 135, 0, 6656 / 12825, 28561 / 56430, -9 / 50, 2 / 55]


def brusselator(time, U):
    u1, u2 = U[0], U[1]
    f1 = 1.0 + (u1 ** 2) * u2 - 4.0 * u1
    f2 = 3.0 * u1 - (u1 ** 2) * u2
    return np.array([f1, f2])


def computeK(previousU, time, step_size, previousK):
    j = len(previousK)
    previous_k_sum = sum((beta[i][j] * previousK[i] for i in range(j)))
    return brusselator(
        time + step_size * alpha[j], previousU + step_size *
            previous_k_sum
    )


def computeIteration(previous, step_size, w, K):
    return previous + step_size * sum(w[i] * K[i] for i in range(6))


tolerance = 1e-6

initial = np.array([1.5, 3.0])
U = [initial]
```

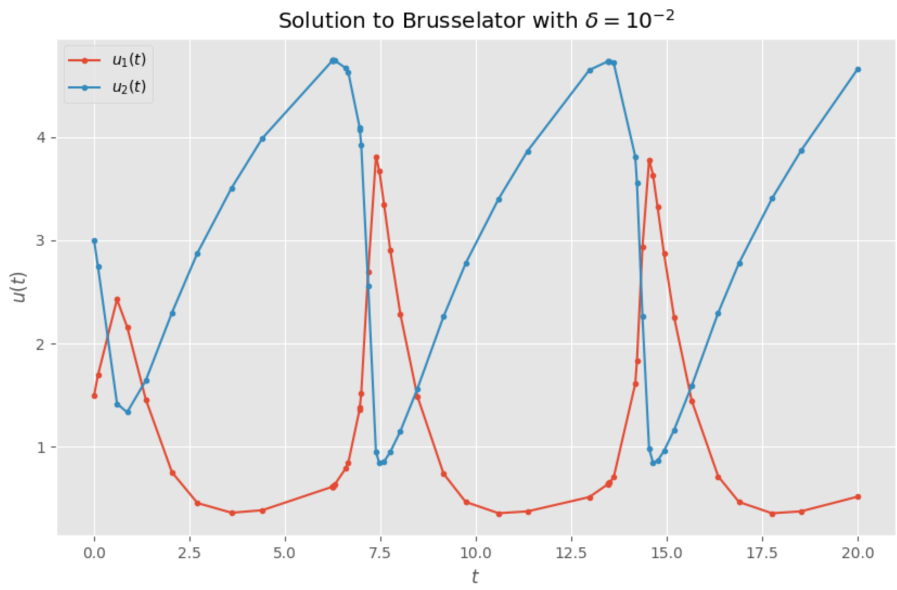
```
h = 0.1
tn = 0.0 + h
t = [0.0]
t_stop = 20

reached_end = False
iterations = 0

while True:
    iterations += 1
    Un = U[-1]

    k1 = computeK(Un, tn, h, [])
    k2 = computeK(Un, tn, h, [k1])
    k3 = computeK(Un, tn, h, [k1, k2])
    k4 = computeK(Un, tn, h, [k1, k2, k3])
    k5 = computeK(Un, tn, h, [k1, k2, k3, k4])
    k6 = computeK(Un, tn, h, [k1, k2, k3, k4, k5])
    ks = [k1, k2, k3, k4, k5, k6]

    U_next_A = computeIteration(Un, h, wA, ks)
    U_next_B = computeIteration(Un, h, wB, ks)
    diff = np.linalg.norm(U_next_A - U_next_B)

    gamma = min(0.8 * (tolerance / diff) ** (1 / 5), 5) if diff else 5
    h *= gamma
    if diff >= tolerance and not reached_end:
        tn = t[-1] + h
        continue

    t.append(tn)
    U.append(U_next_B)

    if reached_end:
        break

    tn += h
    if tn > t_stop:
        reached_end = True
        tn = t_stop

print(f"Total iterations:     {iterations}")
print(f"Function Evaluations: {iterations} * 6 = {iterations * 6}")
smallest_step = min(np.diff(t))
print(f"Smallest step taken:  {smallest_step}")
print(f"Total evaluations needed for all small steps: {6 * 20 /
    smallest_step}")


u1, u2 = [u[0] for u in U], [u[1] for u in U]
plt.plot(t, u1, ".-", label=r"$u_1(t)$")
plt.plot(t, u2, ".-", label=r"$u_2(t)$")
plt.title(r"Solution to Brusselator with $\delta = 10^{-6}$")
plt.xlabel(r"$t$")
plt.ylabel(r"$u(t)$")
plt.legend()
plt.show()
```

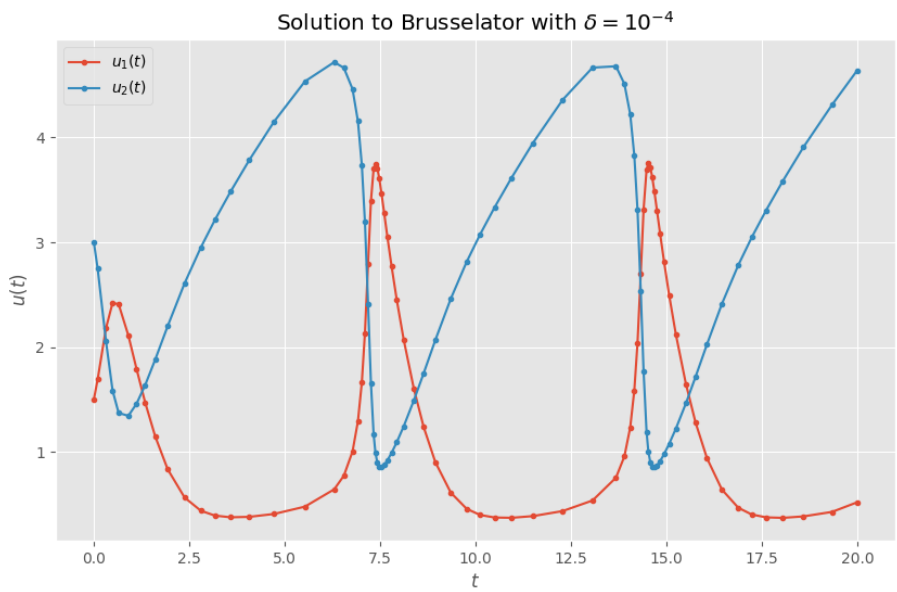
```
plt.plot(u1, u2, ".-")
plt.xlabel(r"$u_1(t)$")
plt.ylabel(r"$u_2(t)$")
plt.title(r"Phase Space of $u_1(t)$ and $u_2(t)$")
plt.show()
```

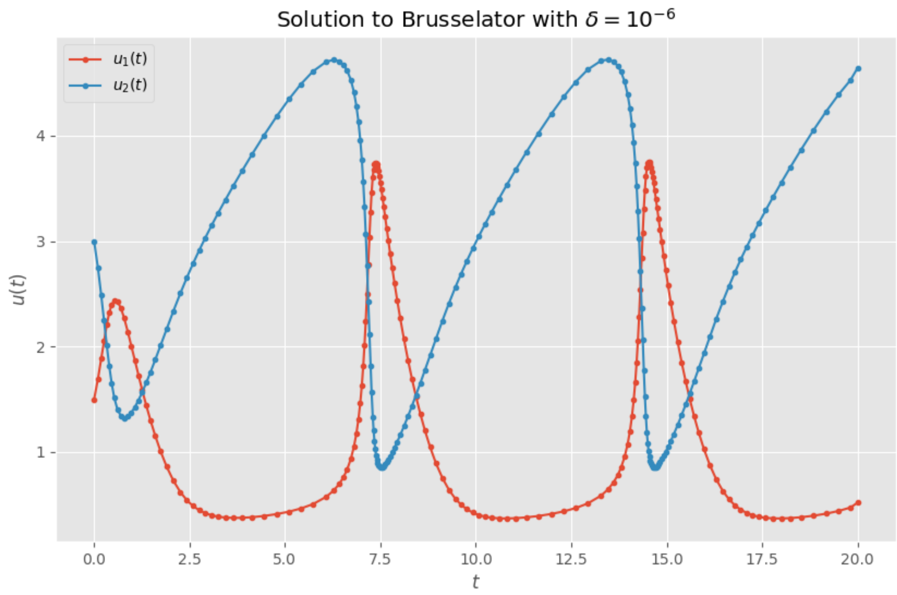
And here's the output of `adaptiveRK.py`:

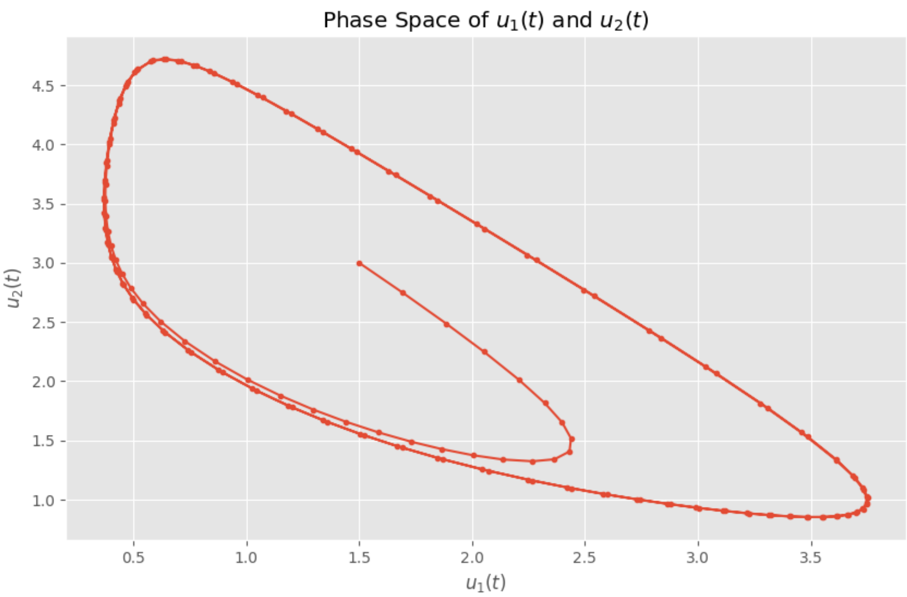
```
Total iterations:      182
Function Evaluations: 182 * 6 = 1092
Smallest step taken:   0.020509068135583064
Total evaluations needed for all small steps: 5851.070326876577
```

From here we can see that the adaptive method is *much* more efficient than had we used a fixed step length method with this smallest step length. Sweet! And now the plots:









We see in the phase space plot that the solution very quickly approaches an orbit, and then stays on that orbit for what seems like the rest of the evolution.