Higher order numerical methods

Roylan Martinez

December 27, 2022

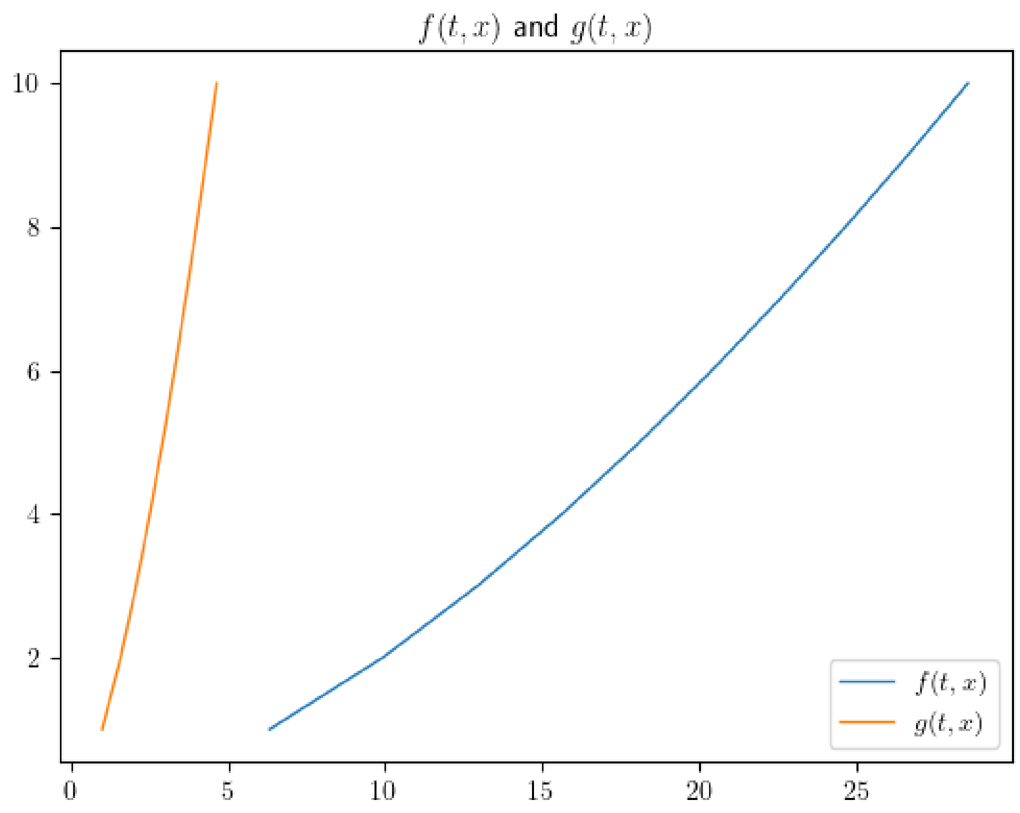
0.0.1 Data needed in first part

[215]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
np.random.seed(1)
```

[216]:
```
# Definitions
x0 = 1
t0 = 0
tfin = 1
dt = 1/100

# Functions
f = lambda t, x: (1/3) * x ** (1/3) + 6 * x ** (2/3)
g = lambda t, x: x ** (2/3)

# Function derivatives
df = lambda t, x: (1/9) * x ** (-2/3) + 4 * x ** (-1/3)
dgdx = lambda t, x: (2/3) * x ** (-1/3)

# Function second derivatives
df2 = lambda t, x: (-2/27) * x ** (-5/3) - (4/3) * x ** (-4/3)
dgdx2 = lambda t, x: (-2/9) * x ** (-4/3)
```

[217]:
```
fig, ax = plt.subplots()
ax.plot(f(0, np.arange(1, 11)), np.arange(1, 11), linewidth=1, label=r"$f(t,␣
 ↪x)$")
ax.plot(g(0, np.arange(1, 11)), np.arange(1, 11), linewidth=1, label=r'$g(t,␣
 ↪x)$')
ax.set_title(r'$f(t, x)$ and $g(t, x)$')
ax.legend()
```

[217]:
```
<matplotlib.legend.Legend at 0x124f38640>
```



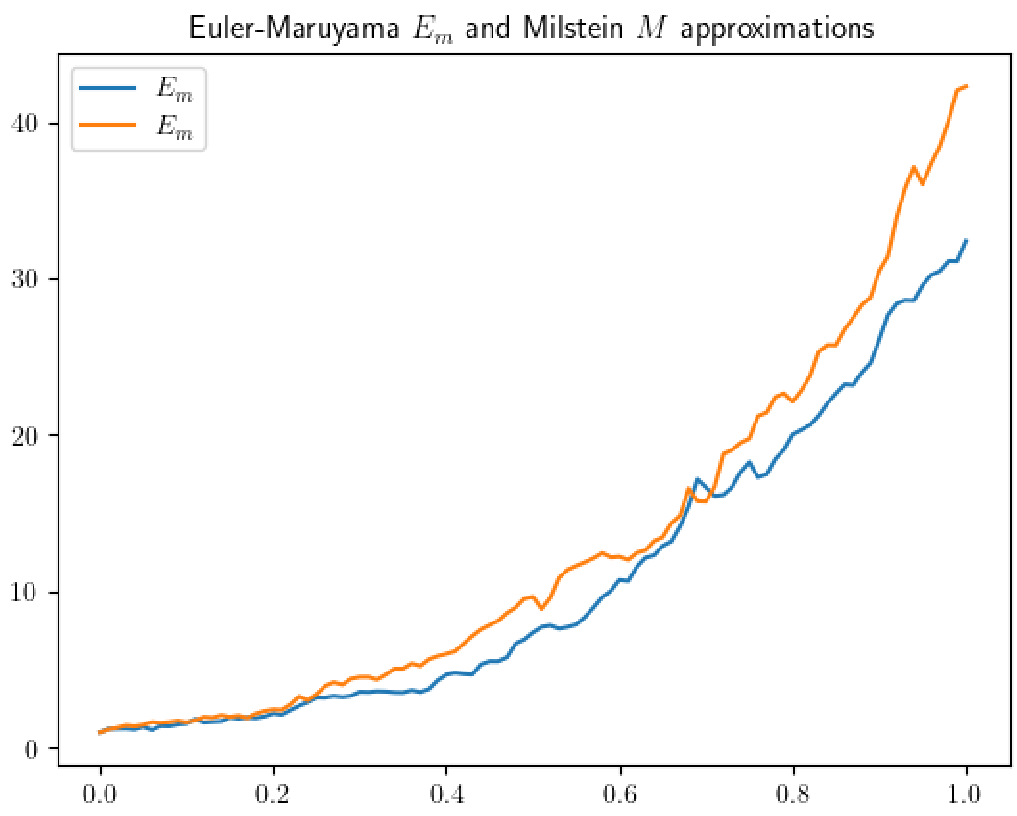
0.1 Function approximation using the Milstein method and the comparisson with the Euler-Maruyama.

[218]:
```
# Define euler maruyama
def euler_maruyama():
    # Vector (0.01, ... , 1) * 0.01
    dt_list = np.arange(0, 101) * dt

    # Vector (0.01, ... , 1)
    deltaw_list = np.random.normal(size=101)

    # Initial list
    xn_list = [x0]

    for t, dt_element in enumerate(dt_list):

        # Compute Xn+1
        xn_1 = xn_list[-1] + f(t, xn_list[-1]) * dt + g(t, xn_list[-1]) *␣
 ↪deltaw_list[t] * np.sqrt(dt)

        # Add it to initial list
        xn_list.append(xn_1)

    # Return final list
    return np.array(xn_list[:-1], dtype='f')
```

[219]:
```
# Define euler maruyama
def milstein():
    # Vector (0.01, ... , 1) * 0.01
    dt_list = np.arange(0, 101) * dt

    # Vector (0.01, ... , 1)
    deltaw_list = np.random.normal(size=101) * np.sqrt(dt)

    # Initial value x_0
    xn_list = [x0]

    for t, dt_element in enumerate(dt_list):

        # Compute Xn+1
        xn_1 = xn_list[-1] + f(t, xn_list[-1]) * dt + g(t, xn_list[-1]) *␣
 ↪deltaw_list[t] + 1/2 * g(t, xn_list[-1]) * dgdx(t, xn_list[-1]) *␣
 ↪((deltaw_list[t])**2 - dt)

        # Add it to initial list
        xn_list.append(xn_1)

    # Return final list
    return np.array(xn_list[:-1], dtype='f')
```

[220]:
```
fig2, ax2 = plt.subplots()
ax2.plot(dt_list, euler_maruyama(), label=r"$E_m$")
ax2.plot(dt_list, milstein(), label=r"$E_m$")
ax2.set_title(r'Euler-Maruyama $E_m$ and Milstein $M$ approximations')
ax2.legend()
```

[220]:
```
<matplotlib.legend.Legend at 0x124f13e80>
```



As we see the approximation seems to coincide in both Euler-Maruyama E_m and Milstein M as the graph above suggests and the approximation does not really improve that much.

0.2 Function approximation using the Milstein method modified with a higher order Runge-Kutta type and comparisson with the 1.5 order with Taylor.

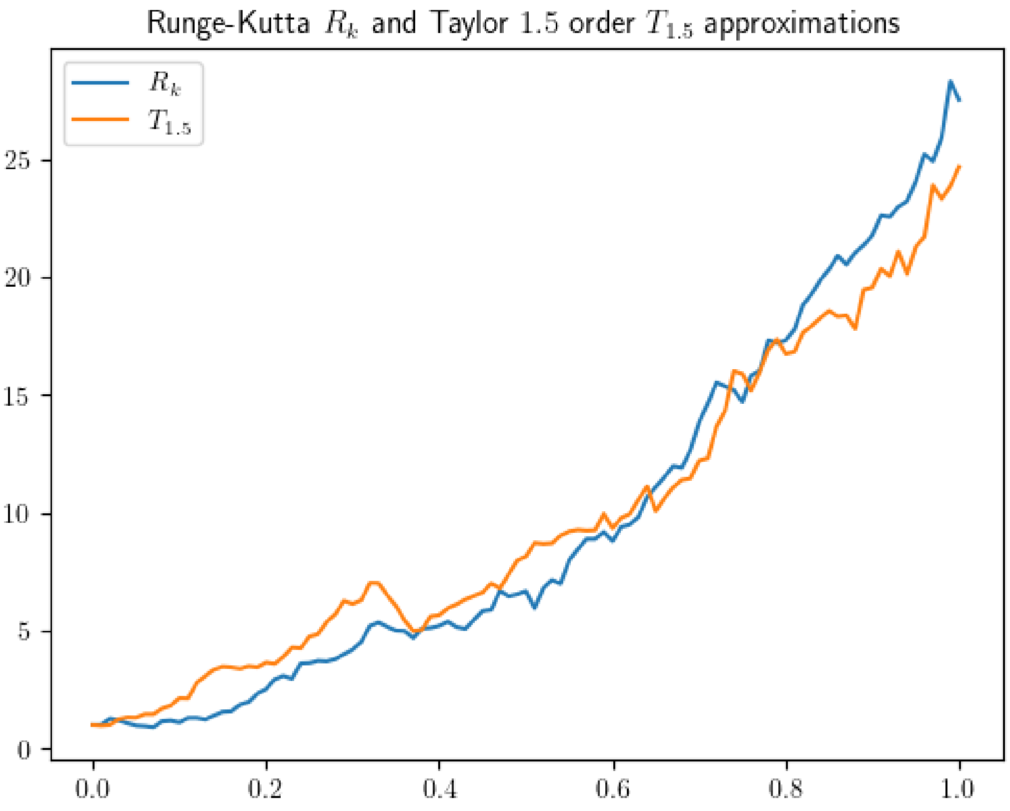
[221]:
```
# Define euler maruyama
def modified_milstein():
    # Vector (0.01, ... , 1) * 0.01
    dt_list = np.arange(0, 101) * dt

    # Vector (0.01, ... , 1)
    deltaw_list = np.random.normal(size=101) * np.sqrt(dt)

    # Initial value x_0
    xn_list = [x0]

    for t, dt_element in enumerate(dt_list):

        # Compute Xn+1
        xn_1 = xn_list[-1] + f(t, xn_list[-1]) * dt + g(t, xn_list[-1]) *␣
↪deltaw_list[t] + 1/2 * (1/dt) * (g(t, xn_list[-1] + dt * g(t, xn_list[-1]))␣
↪- g(t, xn_list[-1])) * ((deltaw_list[t])**2 - dt)

        # Add it to initial list
        xn_list.append(xn_1)

    # Return final list
    return np.array(xn_list[:-1], dtype='f')
```

[222]:
```
# Define euler maruyama
def taylor_method():
    # Vector (0.01, ... , 1) * 0.01
    dt_list = np.arange(0, 101) * dt

    # Vector (0.01, ... , 1) Wn
    deltaw_list = np.random.normal(size=101) * np.sqrt(dt)

    # Vector (0.01, ... , 1) Wn1
    deltaw_list1 = np.random.normal(size=101) * np.sqrt(dt)

    # Initial value x_0
    xn_list = [x0]

    for t, dt_element in enumerate(dt_list):

        # Compute Xn+1
        xn_1 = xn_list[-1] + f(t, xn_list[-1]) * dt + g(t, xn_list[-1]) *␣
↪deltaw_list[t] + 1/2 * g(t, xn_list[-1]) * dgdx(t, xn_list[-1]) *␣
↪((deltaw_list[t])**2 - dt) + 1/2 * df(t, xn_list[-1]) * g(t, xn_list[-1]) *␣
↪dt * (deltaw_list[t] + (deltaw_list1[t])/np.sqrt(3)) + 1/2 * (df(t,␣
↪xn_list[-1]) * f(t, xn_list[-1]) + (1/2) * df2(t, xn_list[-1]) * g(t,␣
↪xn_list[-1]) ** 2) * (dt) ** 2 + 1/2 * (dgdx(t, xn_list[-1]) * f(t,␣
↪xn_list[-1]) + (1/2) * dgdx2(t, xn_list[-1]) * g(t, xn_list[-1]) ** 2) * dt␣
↪* (deltaw_list[t] - deltaw_list1[t]/np.sqrt(3)) + 1/6 * (dgdx(t,␣
↪xn_list[-1]) ** 2 * g(t, xn_list[-1]) + dgdx2(t, xn_list[-1])*g(t,␣
↪xn_list[-1]) ** 2) * dt * ((deltaw_list[t]) ** 3 - 3 * dt * deltaw_list[t])

        # Add it to initial list
        xn_list.append(xn_1)

    # Return final list
    return np.array(xn_list[:-1], dtype='f')
```

[223]:
```
fig3, ax3 = plt.subplots()
ax3.plot(dt_list, modified_milstein(), label=r"$R_k$")
ax3.plot(dt_list, taylor_method(), label=r"$T_{1.5}$")
```

```
ax3.set_title(r'Runge-Kutta $R_k$ and Taylor $1.5$ order $T_{1.5}$␣
 ↪approximations')
ax3.legend()
```

```
[223]: <matplotlib.legend.Legend at 0x12504d7c0>
```



Again, we see the approximation does seem to coincide in both Euler-Maruyama E_m, Milstein M, Runge-Kutta R_k and Taylor 1.5 order $T_{1.5}$. This is really interesting since despite the extra computational effort the approximation does not really improve that much. This actually makes us wonder if there might be computationally less expensive methods to achieve similar approximations.