Heat Transfer Problem

Consider the heat transfer problem:

$$u_t - u_{xx} = f(x,t), (x,t) \in (0,1) \times (0,1)$$

with the following initial and boundary conditions:

- Initial condition:

$$u(x,0) = \sin(\pi x)$$

- Dirichlet boundary conditions:

$$u(0,t) = u(1,t) = 0$$

and the source function:

$$f(x,t) = (\pi^2 - 1) e^{-t} \sin(\pi x)$$

Analytic Solution

The analytic solution to this problem is given by:

$$u(x,t) = e^{-t} \sin(\pi x)$$

Forward Euler Code

```
import numpy as np
from matplotlib import pyplot as plt
from math import sqrt

# forcing function
def f(x, t):
    return  (np.pi**2 - 1) * np.exp(-t) * np.sin(np.pi * x)

# stiffness and mass matrix assembly
def K_M(N, h):
    Ne = N - 1
    K = np.zeros((N, N))
    M = np.zeros((N, N))

    for k in range(Ne):
        mlocal = np.zeros((2, 2))
        klocal = np.zeros((2, 2))
```

```python
        for l in range(2):
            for m in range(2):
                klocal[l, m] = -1 / h if l != m else 1 / h
                mlocal[l, m] = h / 6 if l != m else h / 3

        # global stiffness/mass matrix assembly
        for l in range(2):
            globalnode1 = k + l
            for m in range(2):
                globalnode2 = k + m
                K[globalnode1, globalnode2] += klocal[l, m]
                M[globalnode1, globalnode2] += mlocal[l, m]

    return K, M

# applying dirichlet boundary conditions
def apply_dirichlet_bc(K, F, u_db, dirichlet_nodes):
    for i in dirichlet_nodes:
        for j in range(K.shape[0]):
            if i != j:
                F[j] -= K[j, i] * u_db[i]
                K[j, i] = 0  # Set column to zero
                K[i, j] = 0  # Set row to zero
        K[i, i] = 1
        F[i] = u_db[i]
    return K, F

# boundary conditions
def u_boundary(x):
    return np.sin(np.pi * x)

# analytic solution to given problem -- used for rechecking
def u_solution(x,t):
    return np.sin(np.pi * x)*np.exp(-t)

def FEM(method = "FE", N = 11, dt = 1/551, plot = True):
    if N <2:
        print("Need at least 2 nodes to perform FEM")
        return "ERROR"

    xi = np.linspace(0, 1, N)
    h = xi[1] - xi[0]

    tf = 1
    t0 = 0
    nt = int((tf - t0) / dt)
    ctime = t0

    K, M = K_M(N, h)
    M_inv = np.linalg.inv(M)
    dx_de = h / 2

    phi = lambda eta: [(1 - eta) / 2, (1 + eta) / 2]
    quad_points = [-1/sqrt(3), 1/sqrt(3)]
    quad_weights = [1,1]

    u = np.zeros((N, nt + 1))  # Time-dependent solution array
    u[:, 0] = u_boundary(xi)

    # Set Dirichlet boundary conditions
    u_db = np.zeros(N)  # Prescribed values (e.g., zero at boundaries)
    dirichlet_nodes = [0, N - 1]  # Dirichlet nodes at boundaries

    # constructing F vector
    for n in range(nt):
        ctime = t0 + n * dt
        F = np.zeros(N)

        for k in range(N - 1):
            flocal = np.zeros(2)

            for l in range(2):
                for p in range(len(quad_points)):
                    if method =="FE":
                        flocal[l] += f((h / 2) * (quad_points[p] + 1)
+ xi[k], ctime) *phi(quad_points[p])[l] * quad_weights[p]  * dx_de

                    else: # backward euler, evaluate at next/future
time step
                        flocal[l] += f((h / 2) * (quad_points[p] + 1)
+ xi[k], ctime+dt) *phi(quad_points[p])[l] * quad_weights[p]  * dx_de

            for l in range(2):
                globalnode = k + l
                F[globalnode] += flocal[l]

        if method == 'FE':
            # Apply Dirichlet boundary conditions to K and F
            K_bc, F_bc = apply_dirichlet_bc(K.copy(), F.copy(), u_db,
dirichlet_nodes)
            u[:,n + 1] = u[:,n] - dt * M_inv @ K_bc @ u[:,n] + dt *
M_inv @ F_bc
        else:
            B = M + dt * K  # System matrix
            b = M @ u[:, n] + F * dt  # Right-hand side vector

            # Apply Dirichlet boundary conditions to B and b (Bu = b
system)
```

```
            B_bc, b_bc = apply_dirichlet_bc(B.copy(), b.copy(), u_db,
dirichlet_nodes)

            # Solve for the next timestep
            u[:, n + 1] = np.linalg.solve(B_bc, b_bc)

    if plot:
        plt.scatter(xi,u[:,-1],label = f"{method} FEM solution", color
= "darkgreen")
        plt.plot(xi,u[:,-1], label = f"linear interpolation of
{method} FEM", color = "red", linewidth = 3)
        plt.plot(np.linspace(0,1,100),
u_solution(np.linspace(0,1,100),tf), label = "analytic solution",color
= "black")
        plt.legend()
        plt.show()

    mae =  np.sum(np.abs(u_solution(xi,tf)- u[:,-1]))
    return mae
```

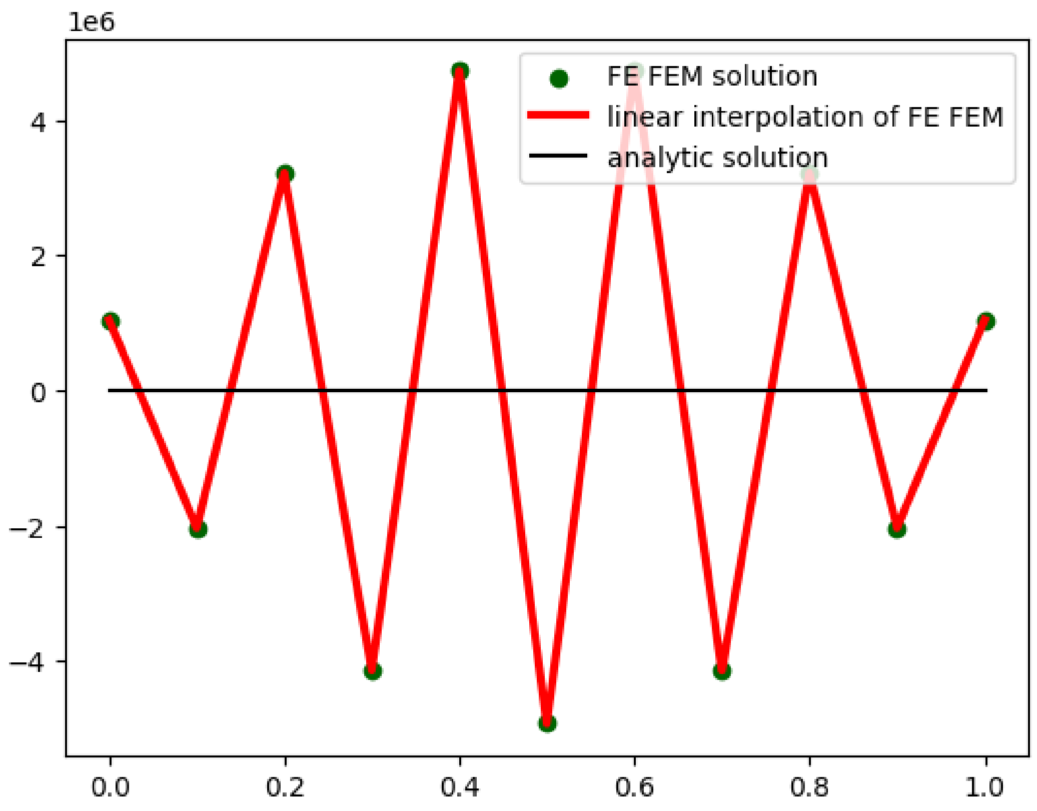
Question 2

Solve first by using a forward Euler time derivative discretization with a time-step of $\Delta t = 1/551$. Plot the results at the final time. Increase the time-step until you find the instability. What dt does this occur at? How does the solution change as N decreases?

*Note: all plots are done at the final time, $t = 1$.

2.1: Forward Euler time discretization with time spacing $\Delta t = 1/551$ and number of nodes $N = 11$

```
FEM("FE",N =11, dt = 1/551)
```

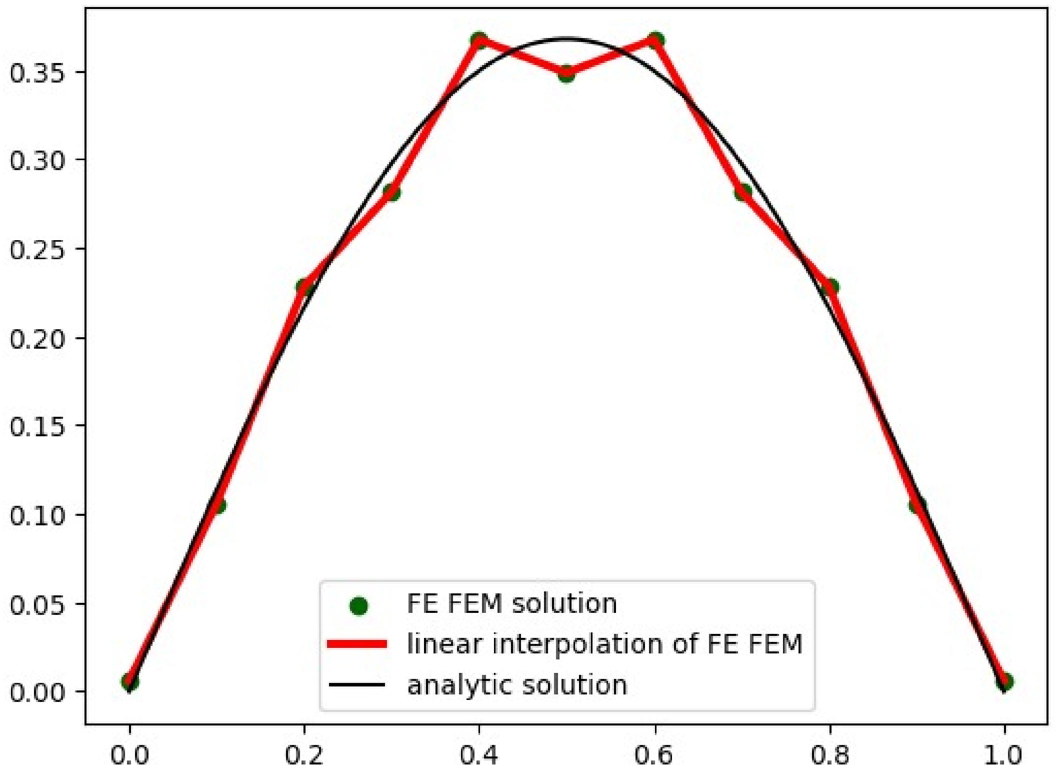


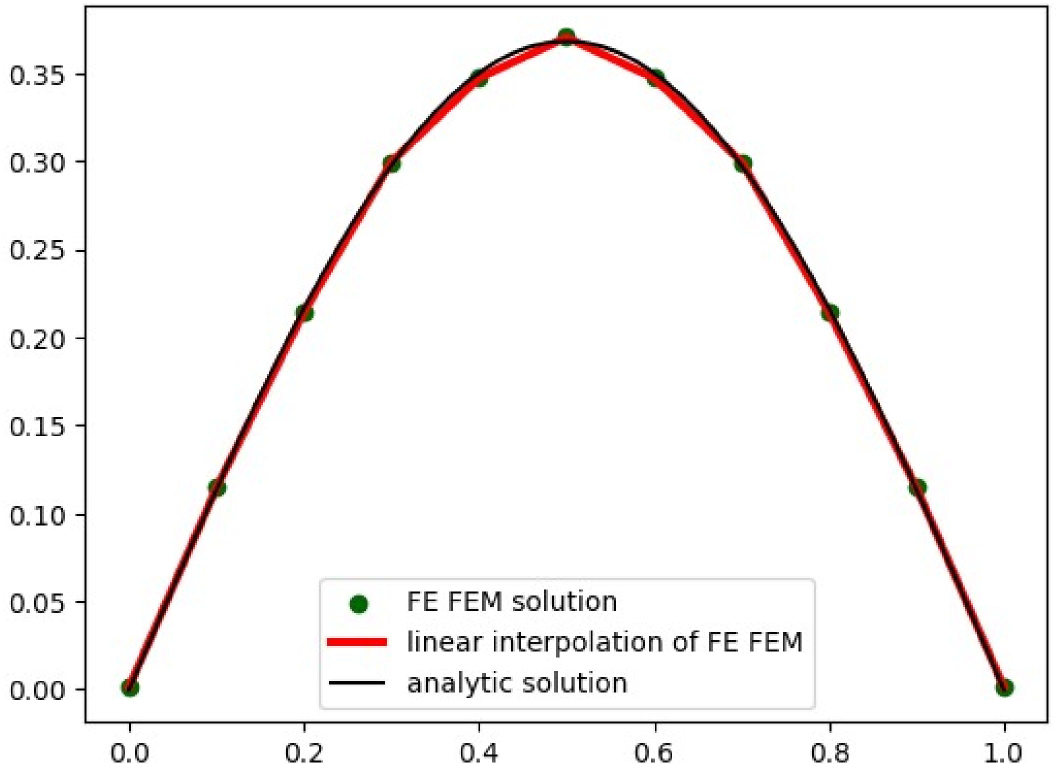
```
35246303.03182401
```

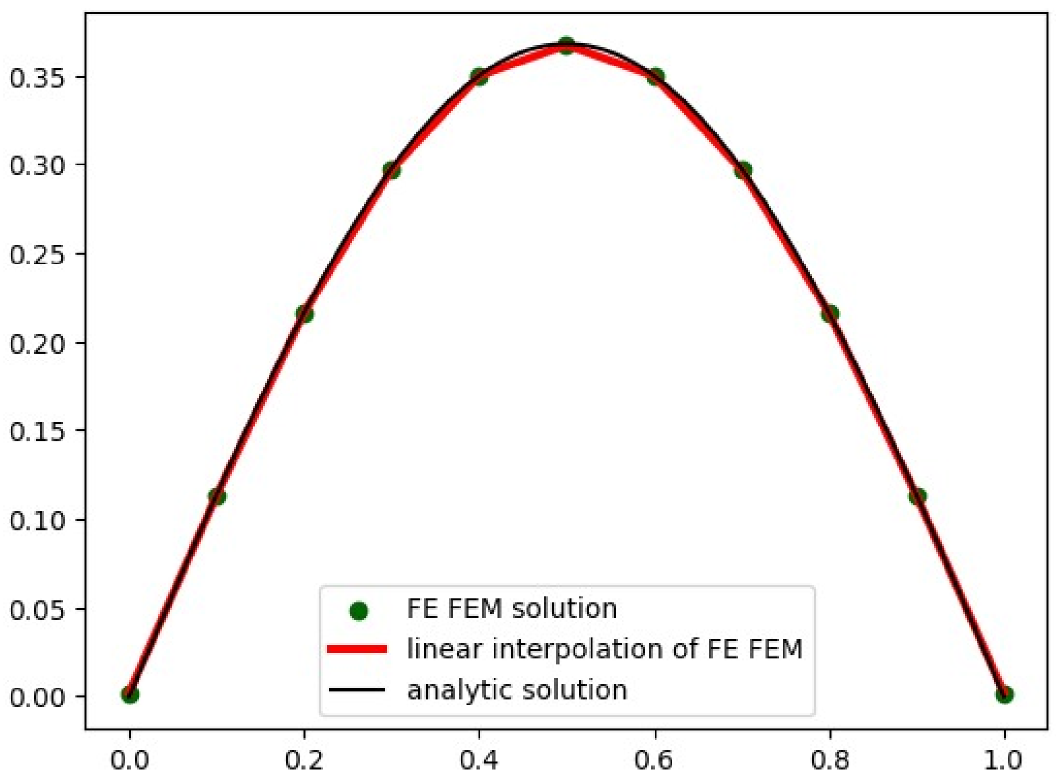
A time step of $1/551$ leads to an unstable FEM solution. We decrease the time step to find stability:

2.2: Forward Euler FEM solution with time discretization variation

```
FEM("FE",N =11, dt = 1/561)
FEM("FE",N =11, dt = 1/562)
FEM("FE",N =11, dt = 1/563)
```


FE FEM solution
linear interpolation of FE FEM
analytic solution

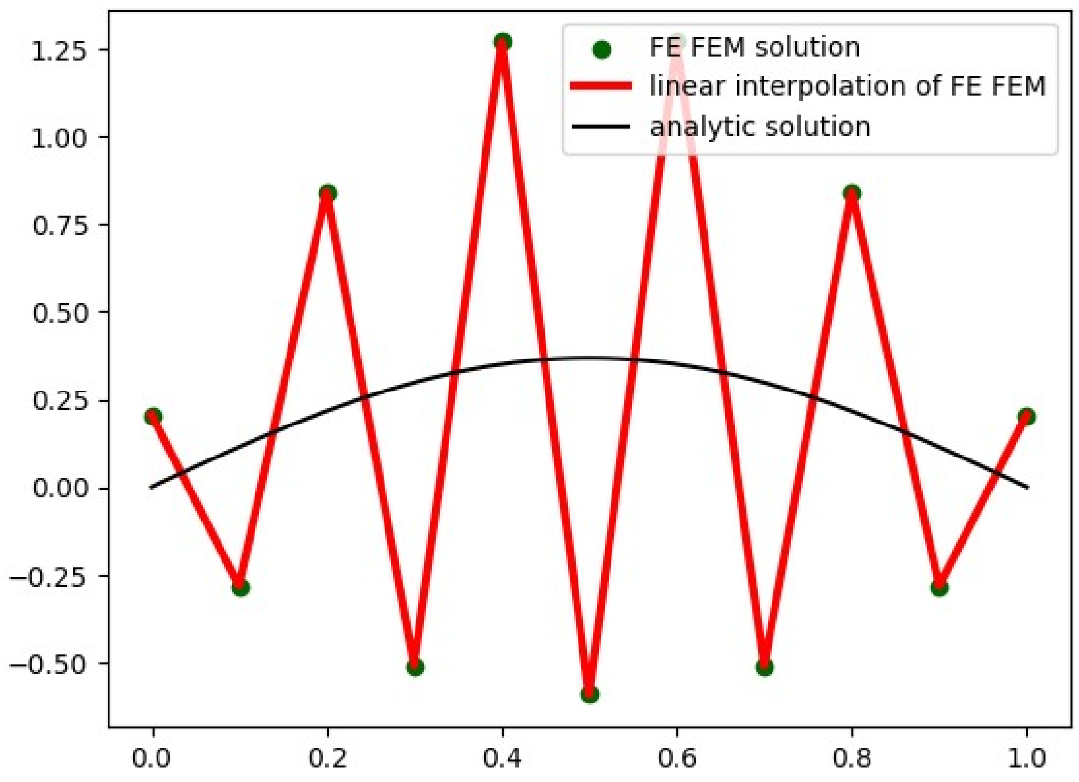

FE FEM solution
linear interpolation of FE FEM
analytic solution

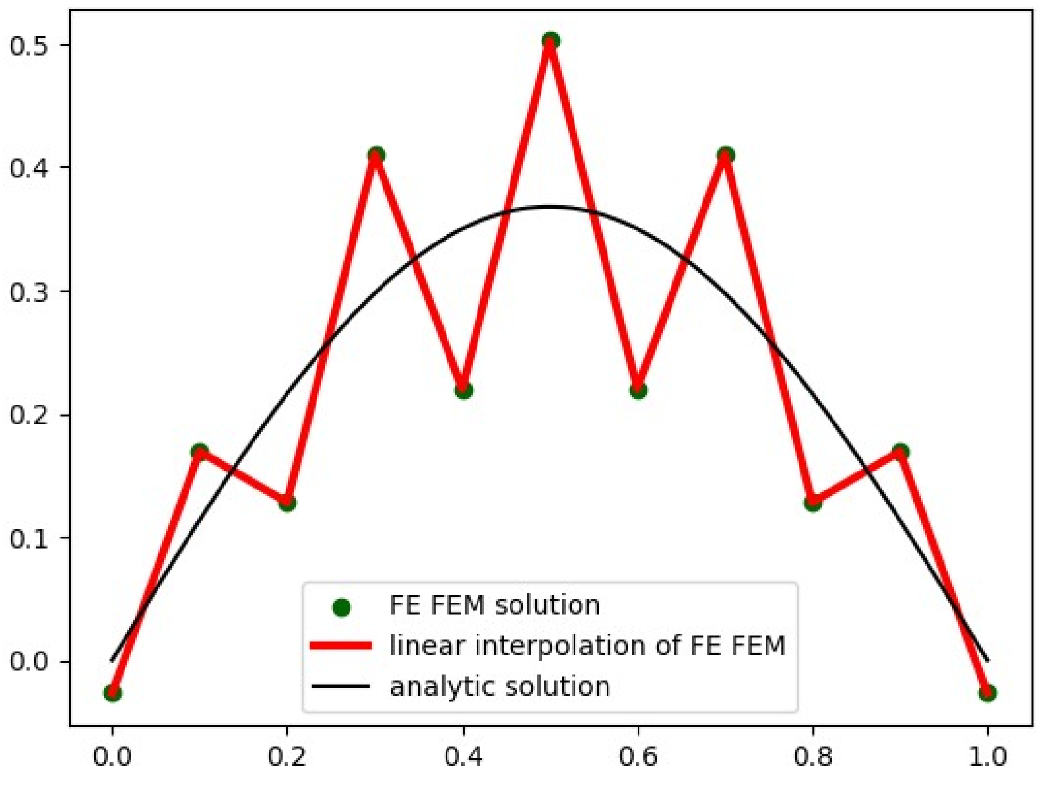


```
0.006561129354341543
```

We notice stability of the FEM solution for a timestep $\Delta t > 1/561$.

```
FEM("FE",N =11, dt = 1/559)
FEM("FE",N =11, dt = 1/560)
```





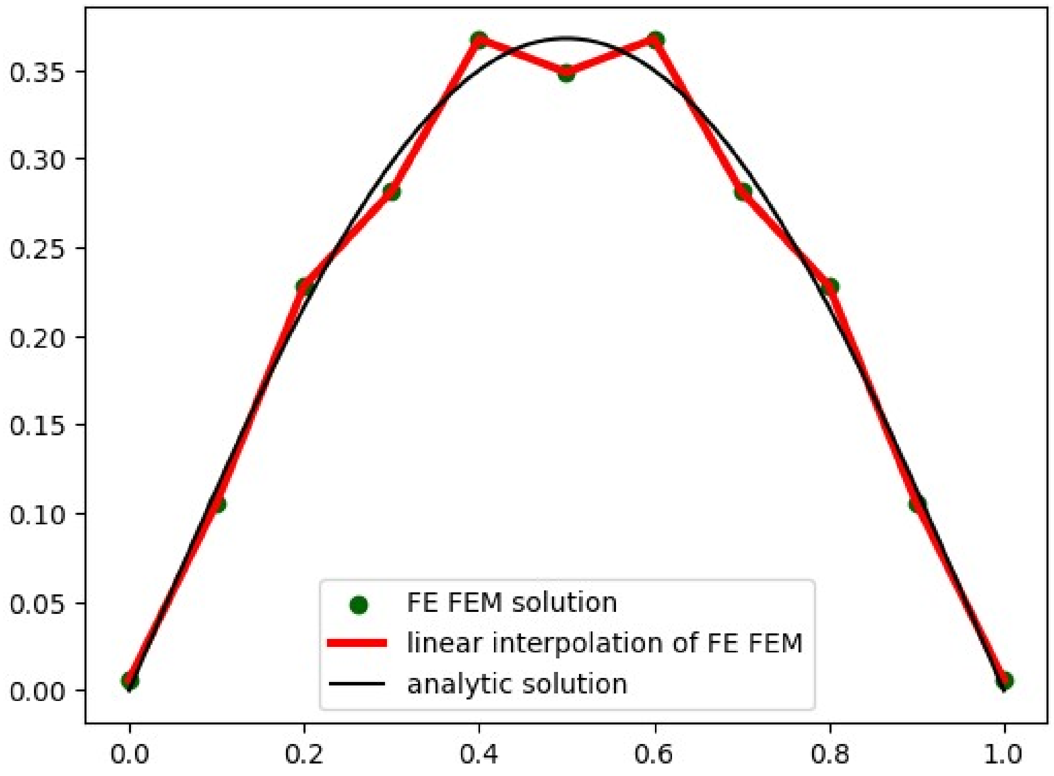
```
0.9607466929307952
```

Additionally, we notice instability of the FEM solution for a timestep $\Delta t < 1/561$. Thus, the instibility cut-off roughly occurs at a step size of $\Delta t = 1/561$, with any larger time steps leading to instability.

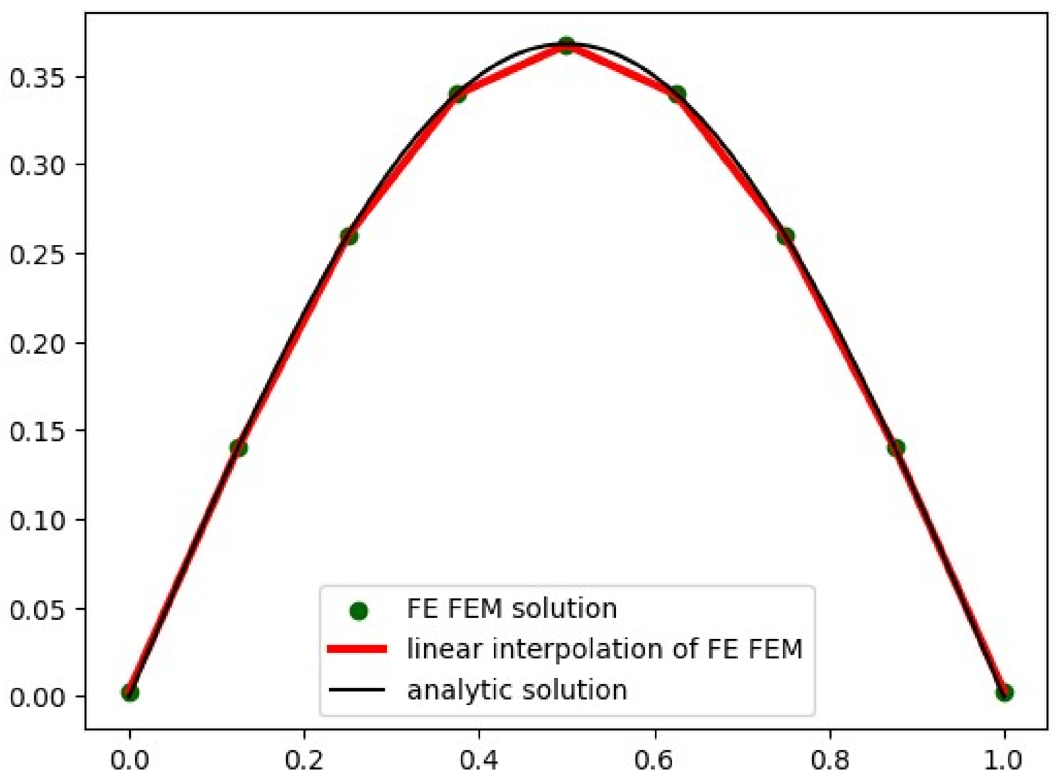
2.3: Forward Euler FEM solution with spatial discretization variation

```
for N in list(range(11, 5, -2)) + [3, 2, 1]:
    print(f"spatial elements, N = {N}")
    FEM("FE", N, dt=1/561)
```

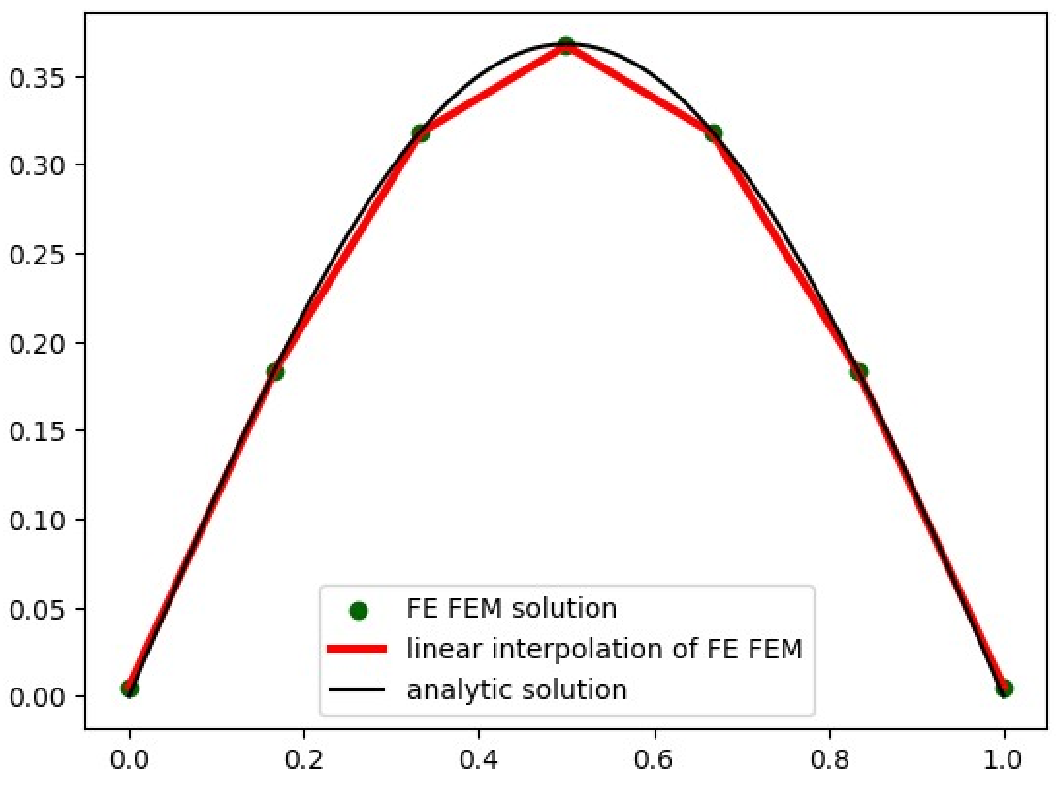
```
spatial elements, N = 11
```



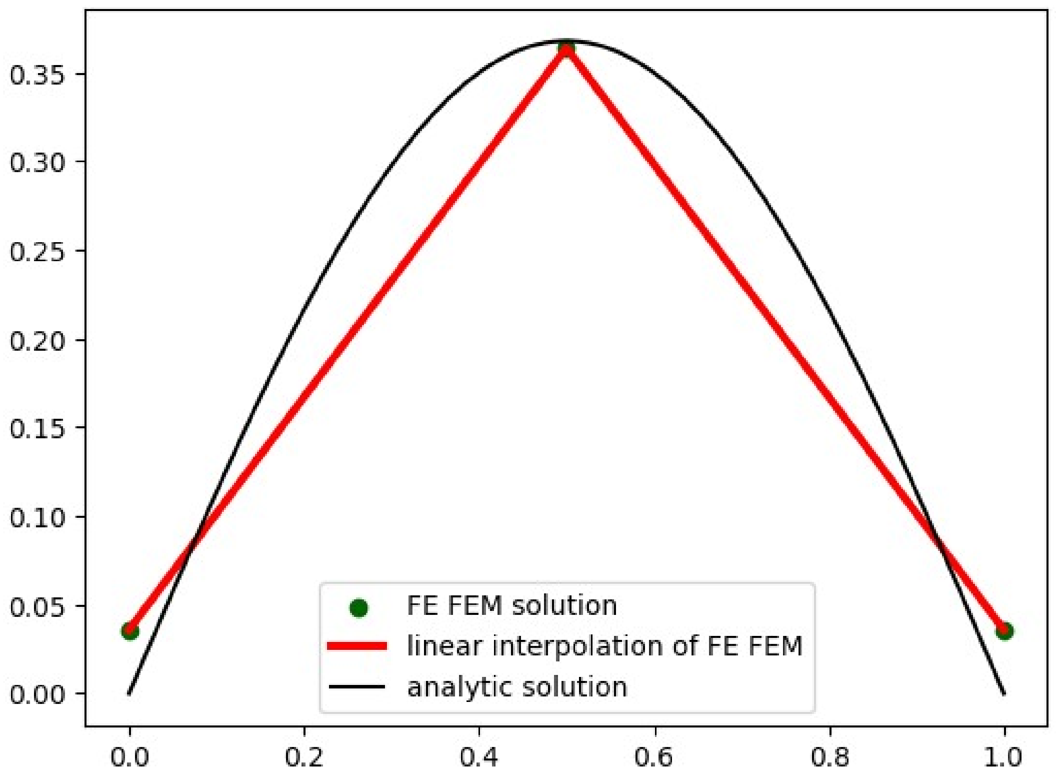
```
spatial elements, N = 9
```



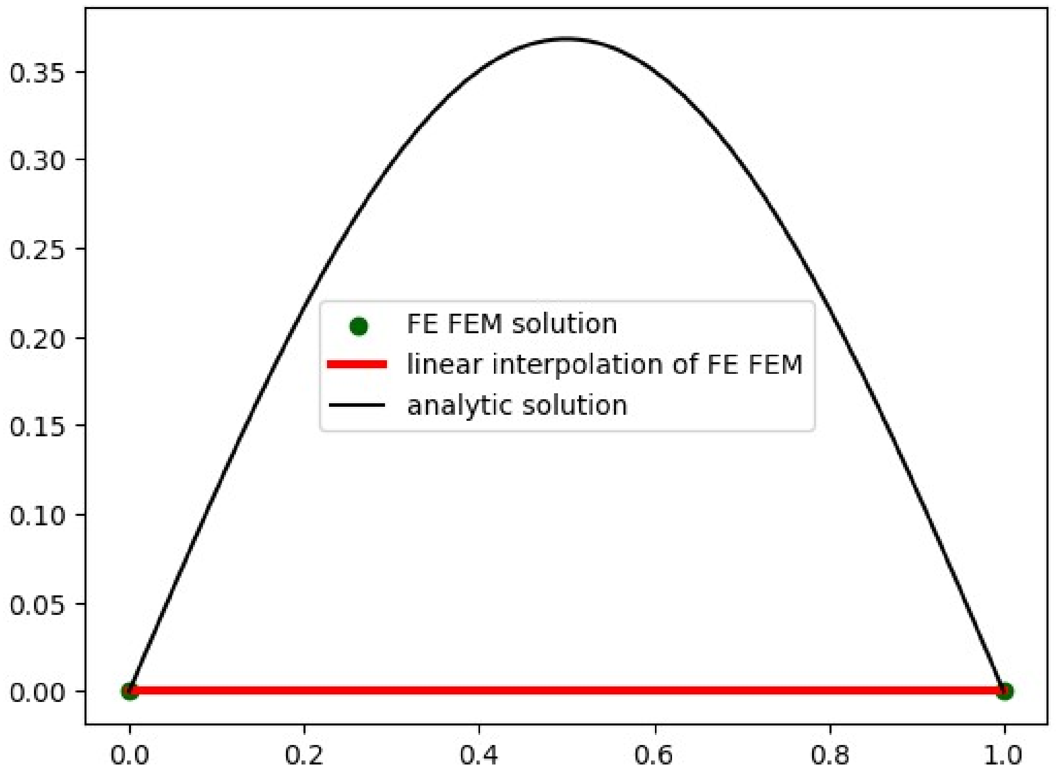
```
spatial elements, N = 7
```



```
spatial elements, N = 3
```



```
spatial elements, N = 2
```

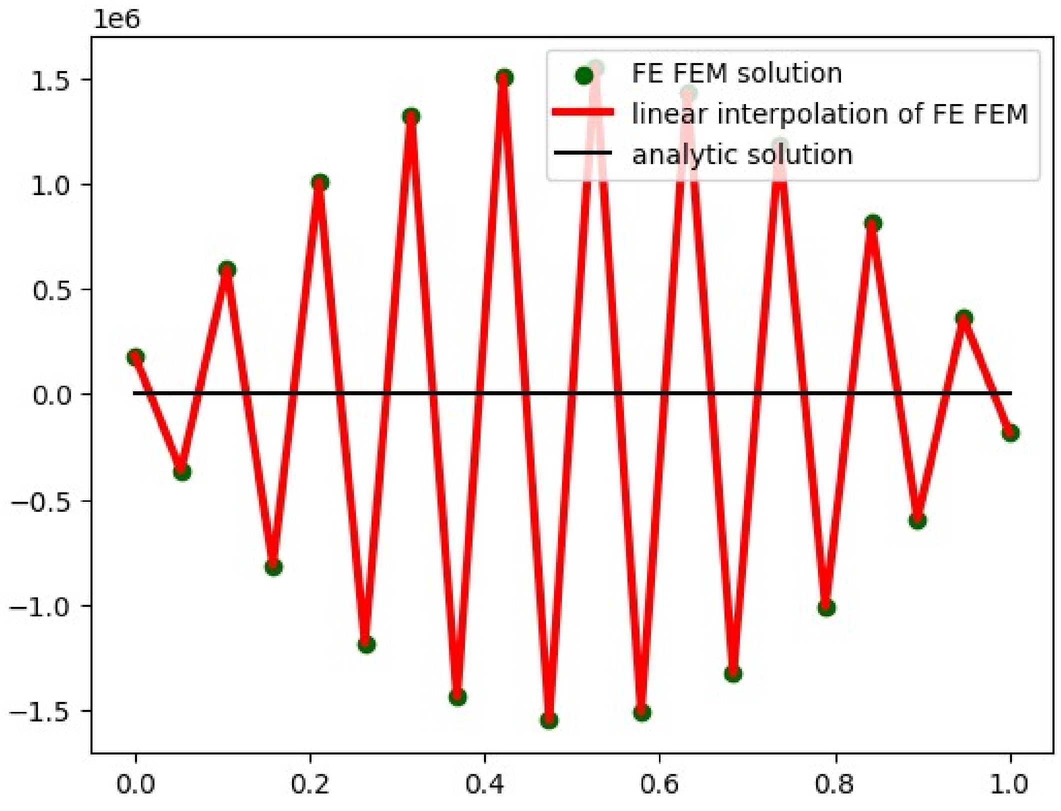


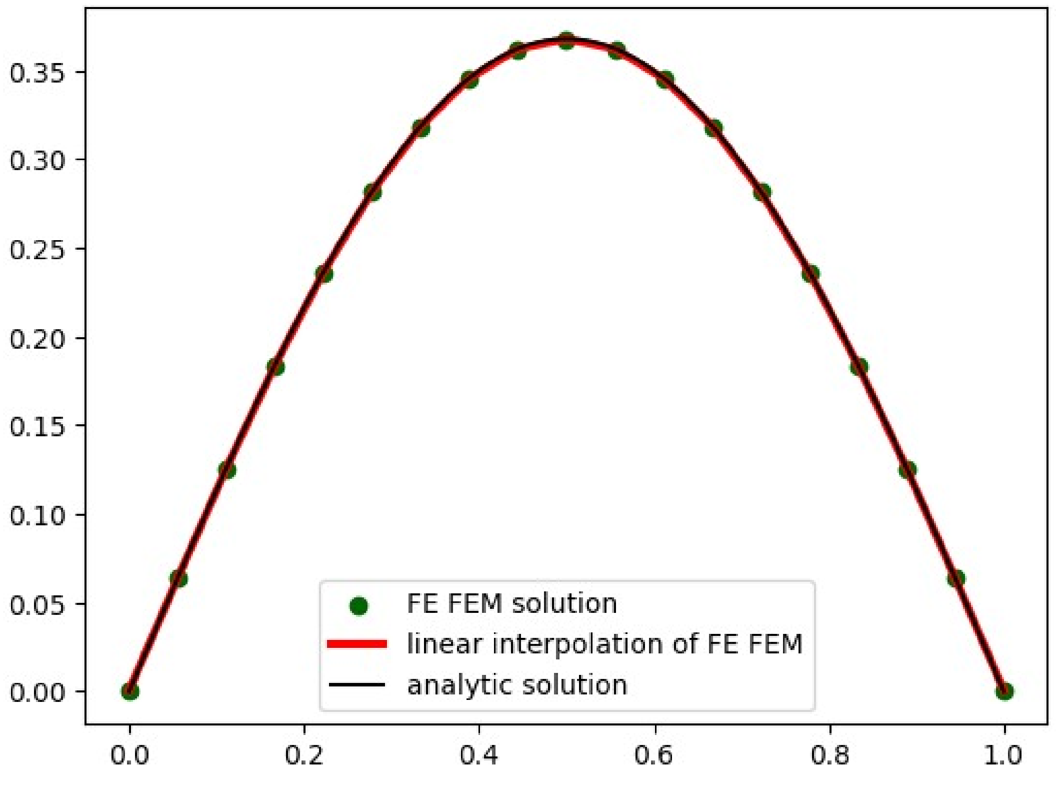
```
spatial elements, N = 1
Need at least 2 nodes to perform FEM
```

There are 2 effects of decreasing the number of spatial discretizations:

1. As the the spatial discretization coarsens, the solution increases in stability -- a larger time discritization can be used. This is demonstrated further below. For $ N = 20, \Delta t =1/2100$ leads to an unstable solution, but for $N = 19, \Delta t = 1/2100$ leads to a stable solution. Notice that a finer spatial discrization requires a finer time discritization.
2. Additionally, the prediction accuracy decreases--while nodal accuracy (solution evaluation at nodes) remains high, a linear solution interpolation between nodes becomes more erraneous. This is further shown when calling the FEM with 2 or 1 spatial nodes. For $N = 2 $, we notice that only the two boundary conditions are satisfied, and for $ N =1$, the FEM cannot be applied.

```
FEM("FE",N =20, dt = 1/2100)
FEM("FE",N =19, dt = 1/2100)
```


1e6
FE FEM solution
linear interpolation of FE FEM
analytic solution
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
FE FEM solution
linear interpolation of FE FEM
analytic solution
0.0
0.2
0.4
0.6
0.8
1.0

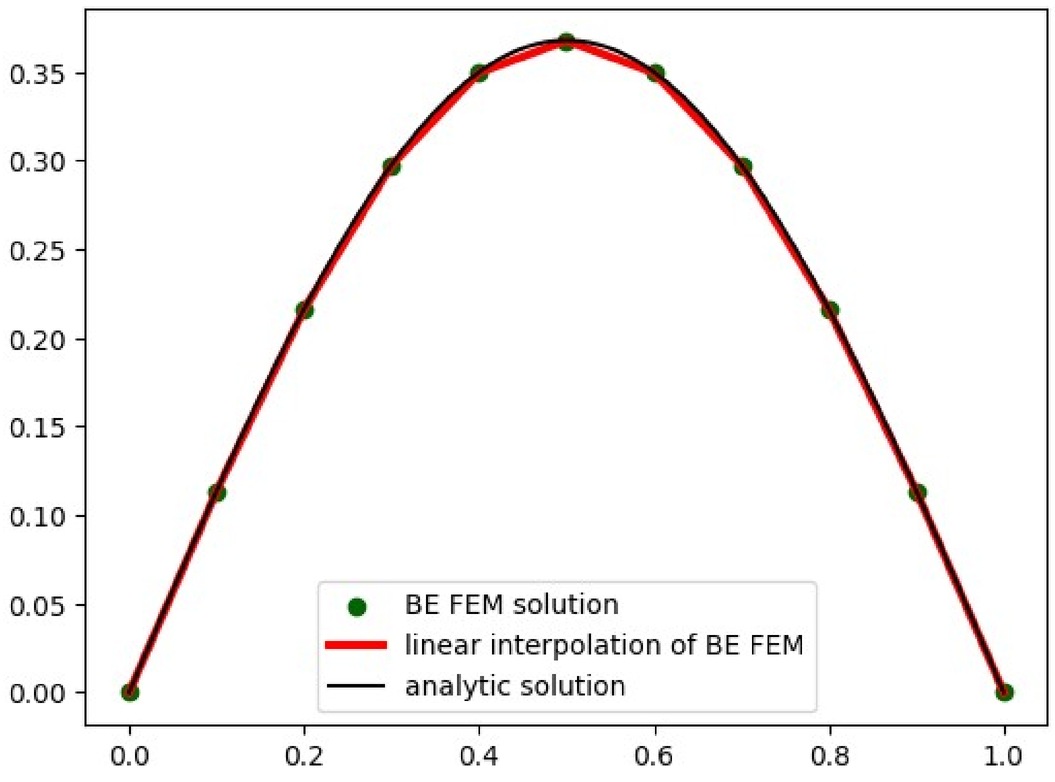
```
0.002509755029599544
```

Question 3

Solve the same problem with the same time-steps using an implicit backward Euler. What happens as the time-step is equal to or greater than the spatial step size? Explain why.

3.1: Backward Euler time discritization with $\Delta t = 1/551$ and $N = 11$

```
FEM("BE",N = 11, dt = 1/551)
```

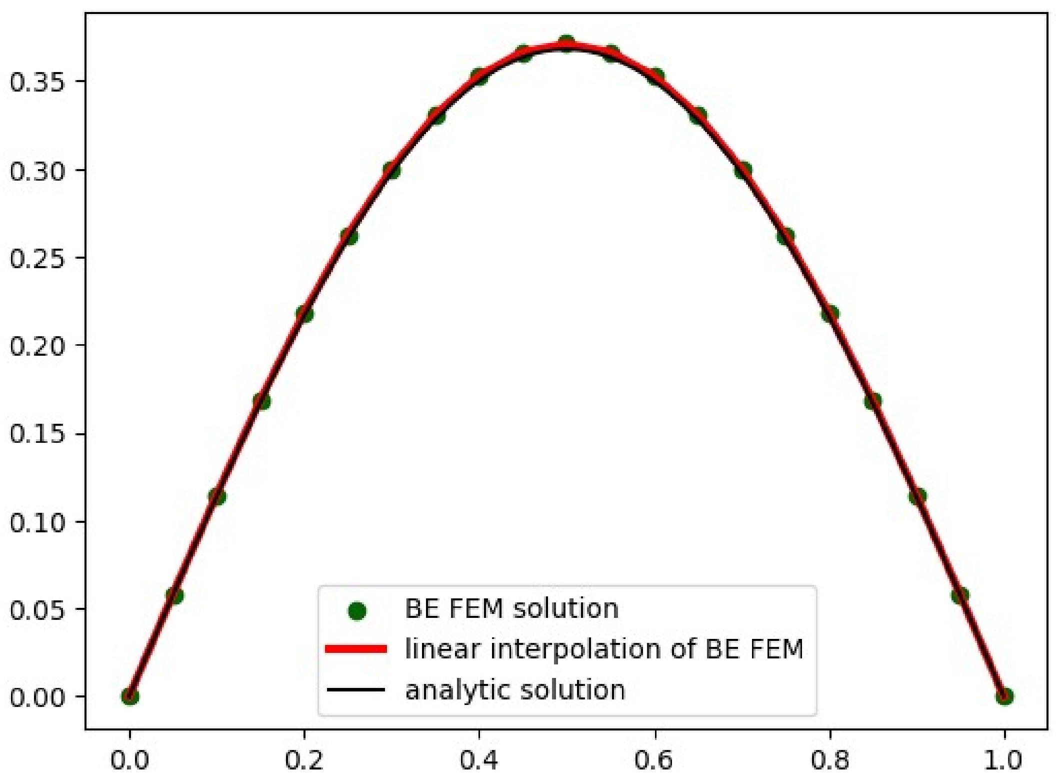


```
0.001889693889565714
```

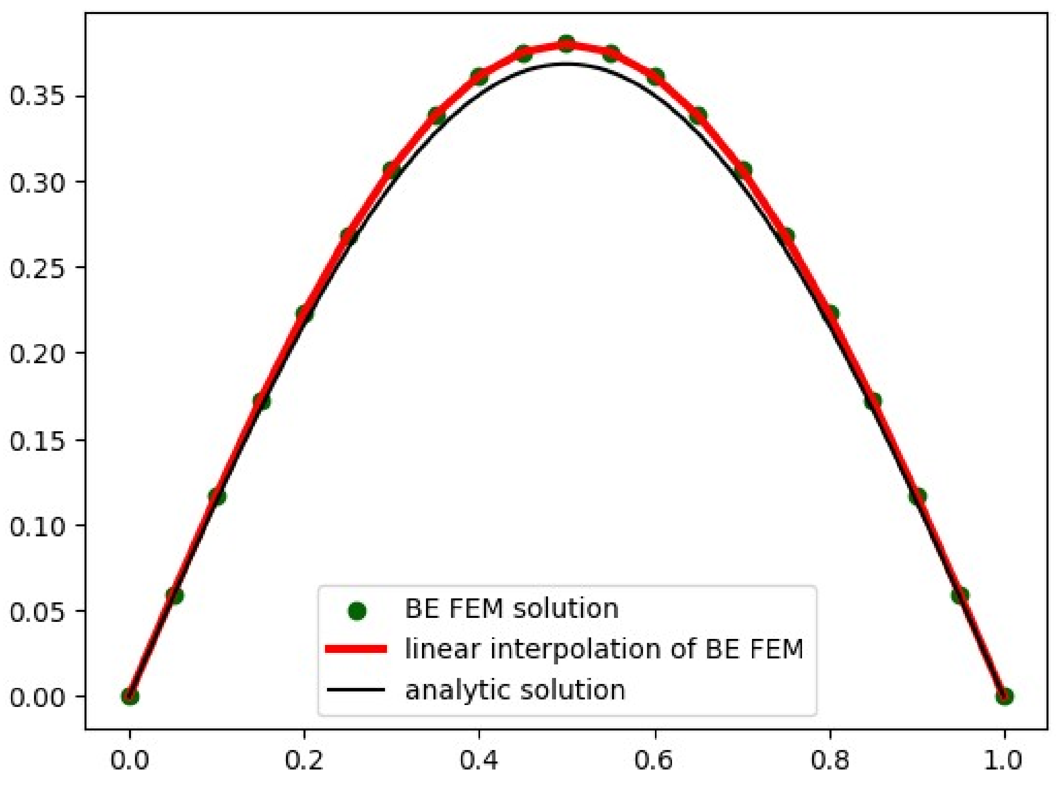
3.2: Backward Euler with time step size greater than spatial step size

As the time step increases to be equal to or greater than the spatial step size, the error increases; specifically, the FEM solution has an increased curvature (higher peak) than the analytic solution. This is shown with the below 3 plots.

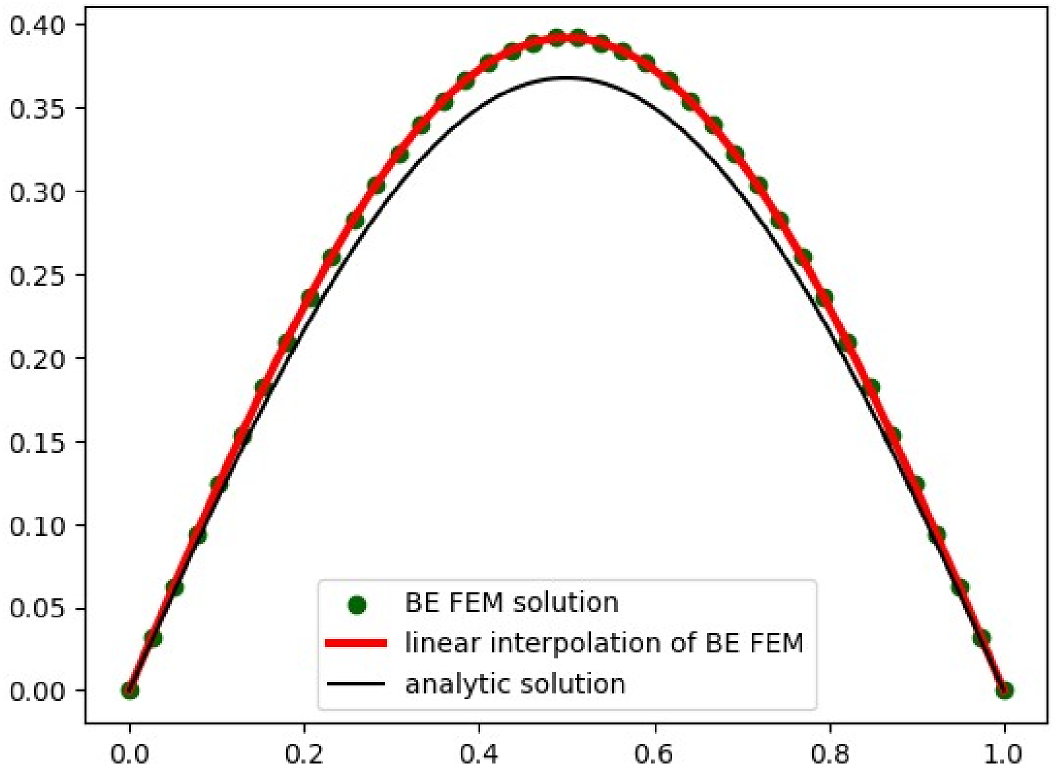
```
print("MAE:", FEM("BE",N = 21, dt =1/7))
print("MAE:", FEM("BE",N = 21, dt =1/2))
print("MAE:", FEM("BE",N = 40, dt =1))
```



```
MAE: 0.0384367897445019
```



```
MAE: 0.14833362523486437
```

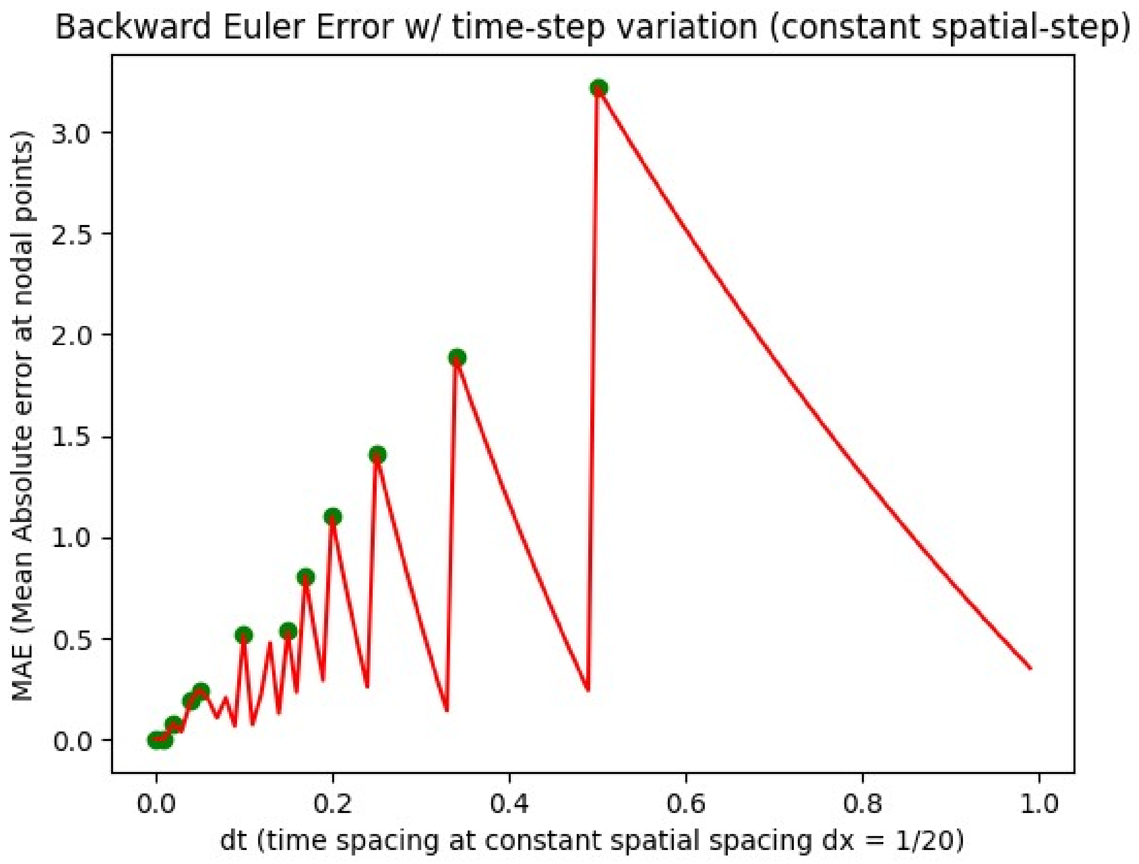


```
MAE: 0.6024988822107951
```

For a more extensive analysis on the error change, we plot the Mean Absolute Error of the FEM solution (calculated at the nodes, vs. the analytic solution) vs. the time step. We use MAE rather than a standard Mean Squared Error (MSE) so outliers (specifically at the peak of the solution) are not extensively penalized; regardless both penalty functions are strictly increasing, so a larger MAE corresponds with a larger MAE.

```
maes=[]
dts = np.arange(.0001,1,.01)
for i in dts:
    mae = FEM("BE", N=21, dt = i, plot = False)
    maes +=[mae]

upper_envelope = []
unique_dts = []
max_so_far = -np.inf

for dt, mae in zip(dts, maes):
    if mae > max_so_far:
        upper_envelope.append(mae)
        unique_dts.append(dt)
        max_so_far = mae

# Plot
plt.scatter(unique_dts, upper_envelope, color="green", label="Top Line
(No Repeats)")
plt.plot(dts,maes, color = "red")
plt.xlabel("dt (time spacing at constant spatial spacing dx = 1/20)")
plt.ylabel("MAE (Mean Absolute error at nodal points)")
plt.title("Backward Euler Error w/ time-step variation (constant
spatial-step)")
```

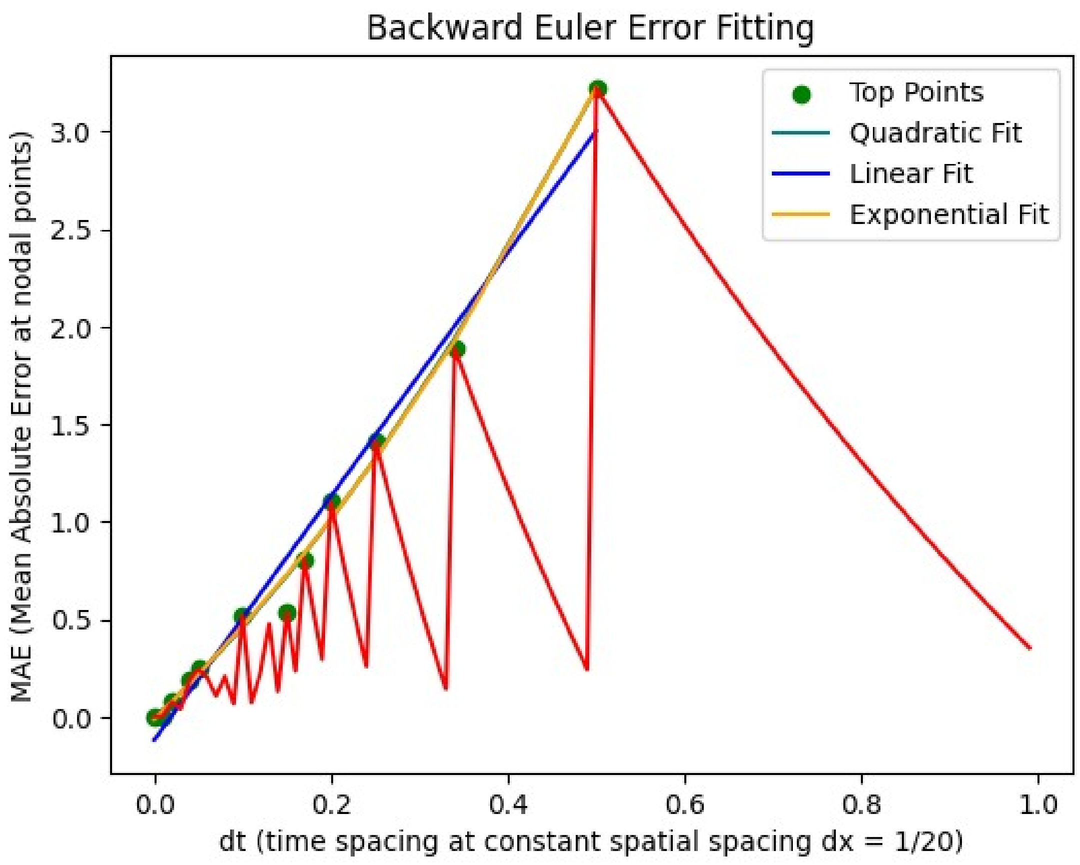
```
Text(0.5, 1.0, 'Backward Euler Error w/ time-step variation (constant
spatial-step)')
```



We notice a "spiky" increasing pattern of the error, as the time spacing is increased. The downward drops are uncharacterestic--likely due to a numerical cancelation of higher order error terms--as we expect an increasing function for the error. Practically, only the values of the peaks should be considered. We plot the peaks of the error (shown in green dots) vs the time step below, and fit 3 different possible error functions: a linear $(error \approx O(dt))$, a quadratic $(error \approx O(dt^2))$, and an exponential function $(error \approx O(e^{c*dt}), c $ is a constant).

```python
from scipy.optimize import curve_fit
from sklearn.metrics import mean_squared_error

upper_envelope = np.array(upper_envelope)
unique_dts = np.array(unique_dts)

# Fitting functions
def quadratic(x, a, b, c):
    return a * x**2 + b * x + c

def linear(x, m, c):
    return m * x + c

def exponential(x, a, b, c):
    return a * np.exp(b * x) + c

# Fit models
quad_params, _ = curve_fit(quadratic, unique_dts, upper_envelope)
lin_params, _ = curve_fit(linear, unique_dts, upper_envelope)
exp_params, _ = curve_fit(exponential, unique_dts, upper_envelope)

# Predict
quad_fit = quadratic(unique_dts, *quad_params)
lin_fit = linear(unique_dts, *lin_params)
exp_fit = exponential(unique_dts, *exp_params)

# Calculate errors
quad_error = mean_squared_error(upper_envelope, quad_fit)
lin_error = mean_squared_error(upper_envelope, lin_fit)
exp_error = mean_squared_error(upper_envelope, exp_fit)

# Print errors
print(f"Quadratic Fit MSE: {quad_error}")
print(f"Linear Fit MSE: {lin_error}")
print(f"Exponential Fit MSE: {exp_error}")

# Plot
plt.scatter(unique_dts, upper_envelope, color="green", label="Top
Points")
plt.plot(unique_dts, quad_fit, color="teal", label="Quadratic Fit")
plt.plot(unique_dts, lin_fit, color="blue", label="Linear Fit")
plt.plot(unique_dts, exp_fit, color="orange", label="Exponential Fit")
plt.plot(dts,maes, color = "red")
plt.xlabel("dt (time spacing at constant spatial spacing dx = 1/20)")
plt.ylabel("MAE (Mean Absolute Error at nodal points)")
plt.title("Backward Euler Error Fitting")
plt.legend()
plt.show()
```

```
Quadratic Fit MSE: 0.005122798694291241
Linear Fit MSE: 0.015989999987420716
Exponential Fit MSE: 0.005169969048497315
```



We notice that a quadratic fit has the least MSE (used since large errors are penalized more significantly, allowing approximation to minimize overall deviation from true data), followed closely by the exponential fit. Thus, we can conclude that the error is $O\left(dt^2\right)$ when dx is held constant.

Note that for more extreme cases ($dt \geq \Delta t$, where Δt is the total time interval, 1 in this case), only one time step is taken, such that the FEM system solves the $Ku = F$ system, the poisson equation. Thus, for any $dt \geq \Delta t$, the error will be constant, but for such a large dt, the FEM solution is found for a different PDE.