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

Boundary conditions:

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

Initial condition:

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

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

Test function: $v(x)$

$v \cdot u_t - v \cdot u_{xx} = v \cdot f$

Domain $= [0,1]$

$$\int_0^1 v u_t \, dx - \int_0^1 v u_{xx} \, dx = \int_0^1 v f \, dx$$

Using Integration by Parts formula:

$$\int_0^1 v u_{xx} \, dx = \left[v u_x \right]_0^1 - \int_0^1 v_x u_x \, dx$$

boundary conditions $\quad u(0,t) = 0 \quad u(1,t) = 0$

$v(0) = v(1) = 0$

$$\left[v u_x \right]_0^1 = 0$$

Final weak form:

$$\int_0^1 v u_t \, dx + \int_0^1 v_x u_x \, dx = \int_0^1 v f \, dx$$

$$u(x,t) = \sum_{j=1}^{N} u_j(t)$$

Galerkin time !!!!

$$U(x,t) = \sum_{j=1}^{N} U_j(t)\,\phi_j(x)$$

From weak form

$$\int_0^1 v\,u_t\,dx + \int_0^1 v_x\,u_x\,dx = \int_0^1 v f\,dx$$

$v(x) = \phi_i(x)$ $\qquad \int_0^1 \phi_i u_t\,dx + \int_0^1 \phi_i' u_x\,dx = \int_0^1 \phi_i f\,dx$

$$u(x,t) = \sum_{j=1}^{N} U_j(t)\,\phi_j(x)$$

$$u_t(x,t) = \sum_{j=1}^{N} U_j'(t)\,\phi_j(x)$$

$$u_x(x,t) = \sum_{j=1}^{N} U_j(t)\,\phi_j'(x)$$

$$\int_0^1 \phi_i \left(\sum_{j=1}^{N} U_j'(t)\,\phi_j\right) dx \Rightarrow \underbrace{\sum_{j=1}^{N} U_j'(t) \int_0^1 \phi_i \phi_j\,dx}_{\text{Mass matrix} \cdot U'}$$

$$\int_0^1 \phi_i' \left(\sum_{j=1}^{N} U_j(t)\,\phi_j'\right) dx \Rightarrow \underbrace{\sum_{j=1}^{N} U_j(t) \int_0^1 \phi_i' \phi_j'\,dx}_{\text{Stiffness matrix} \cdot U}$$

$$\underbrace{\int_0^1 \phi_i f(x,t)\,dx = F_i(t)}_{\text{load vector}}$$

The System ODE is:

$$MU'(t) + KU(t) = F(t)$$

Forward Euler:

$$\dot{U}_{t_n} = \frac{U^{n+1} - U^n}{\Delta t}, \quad KU(t_n) \approx KU^n, \quad F(t_n) = F^n$$

Plug into ode:

$$M\frac{U^{n+1} - U^n}{\Delta t} + KU^n = F^n$$

$$\frac{1}{\Delta t} M U^{n+1} - \frac{1}{\Delta t} MU^n + KU^n - F^n = 0$$

Isolate U^{n+1}

$$\frac{1}{\Delta t} M U^{n+1} = \frac{1}{\Delta t} MU^n - KU^n + F^n$$

$$MU^{n+1} = MU^n - KU^n \cdot \Delta t + F^n \cdot \Delta t$$

$$U^{n+1} = MU^n \cdot M^{-1} - KU^n \cdot \Delta t \cdot M^{-1} + F^n \cdot \Delta t \cdot M^{-1}$$

$$U^{n+1} = U^n + \Delta t M^{-1}(F^n - KU^n)$$

Forward Euler update equation

Backward Euler:

$\dot{U}(t_{n+1}) = \frac{U^{n+1} - U^n}{\Delta t}$ $\quad KU(t_{n+1}) \approx KU^{n+1}$, $\quad F(t_{n+1}) \approx F^{n+1}$

Plug into ODE:

$M\dot{U} + KU = F$:

$$M\frac{U^{n+1} - U^n}{\Delta t} + KU^{n+1} = F^{n+1}$$

$$\frac{1}{\Delta t} M U^{n+1} - M\frac{1}{\Delta t} U^n + KU^{n+1} = F^{n+1}$$

Isolate U^{n+1} terms

$$\frac{1}{\Delta t} MU^{n+1} + KU^{n+1} = F^{n+1} + \frac{1}{\Delta t} MU^n$$

$$\left(\frac{1}{\Delta t} M + K\right) U^{n+1} = F^{n+1} + \frac{1}{\Delta t} MU^n$$

$$\Delta t \left(\frac{1}{\Delta t} M + K\right) U^{n+1} = \left(F^{n+1} + \frac{1}{\Delta t} MU^n\right) \Delta t$$

$$(M + \Delta t K) U^{n+1} = F^{n+1} \cdot \Delta t + M U^n$$

Or

$$U^{n+1} = (F^{n+1} \cdot \Delta t + M U^n) \cdot (M + \Delta t K)^{-1}$$

Update equation for Backward Euler

Mapping, quadrature, and assembly which will be used in code:

Mapping: $x(\xi) = \frac{x_L + x_R}{2} + \frac{x_R - x_L}{2}\xi$, $\frac{dx}{d\xi} = J = \frac{x_R - x_L}{2}$ Where $\in [-1, 1]$

$\xi(x) = \alpha(x - x_L) + \beta$, $\xi(x_L) = -1$, $\xi(x_R) = 1$ $\Rightarrow$ $\alpha = \frac{2}{x_R - x_L}$

$x(\xi) = x_L + \frac{x_R - x_L}{2}(\xi + 1)$

$\frac{d\xi}{dx} = \frac{2}{x_R - x_L}$, $\frac{dx}{d\xi} = \frac{x_R - x_L}{2}$

Shaping in parent space: $\hat{\phi}_1(\xi) = \frac{1-\xi}{2}$, $\hat{\phi}_2(\xi) = \frac{1+\xi}{2}$

$(Me)_{ij} = \int_{x_L}^{x_R} \phi_i \phi_j dx = \int_{-1}^{1} \hat{\phi}_i(\xi)\hat{\phi}_j(\xi)\, J\, d\xi$

$(ke)_{ij} = \int_{x_L}^{x_R} \phi_i'(x)\phi_j'(x)\, dx$

$\frac{d\phi}{dx} = \frac{d\hat{\phi}}{d\xi}\frac{d\xi}{dx} = \frac{d\hat{\phi}/d\xi}{J}$

$(ke)_{ij} = \int_{-1}^{1} \left(\frac{d\hat{\phi}_i}{dx}\right)\left(\frac{d\hat{\phi}_j}{dx}\right) J$

$(Fe)_i(t) = \int_{x_L}^{x_R} \phi_i(x) f(x,t)\, dx$

$(F_e)_i(t) = \int_{-1}^{1} \hat{\phi}_i(\xi) f(x(\xi), t)\, J\, d\xi$

COE_Project

December 4, 2025

[2]:
```
print("Hello, this is my Final Project, this project can be run on standard␣
↪terminal but seeing as how this project focuses on graphs, it is highly␣
↪reccomended if you want to see the graphs that this is run on jupyterhub")
```

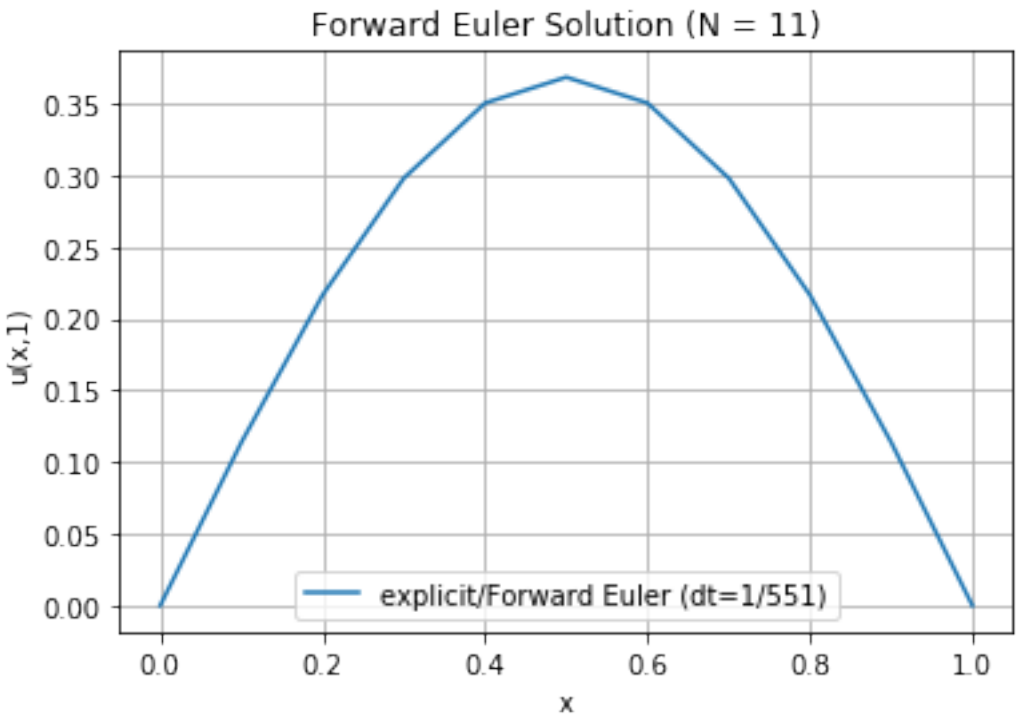
```
Hello, this is my Final Project, this project can be run on standard terminal
but seeing as how this project focuses on graphs, it is highly reccomended if
you want to see the graphs that this is run on jupyterhub
```

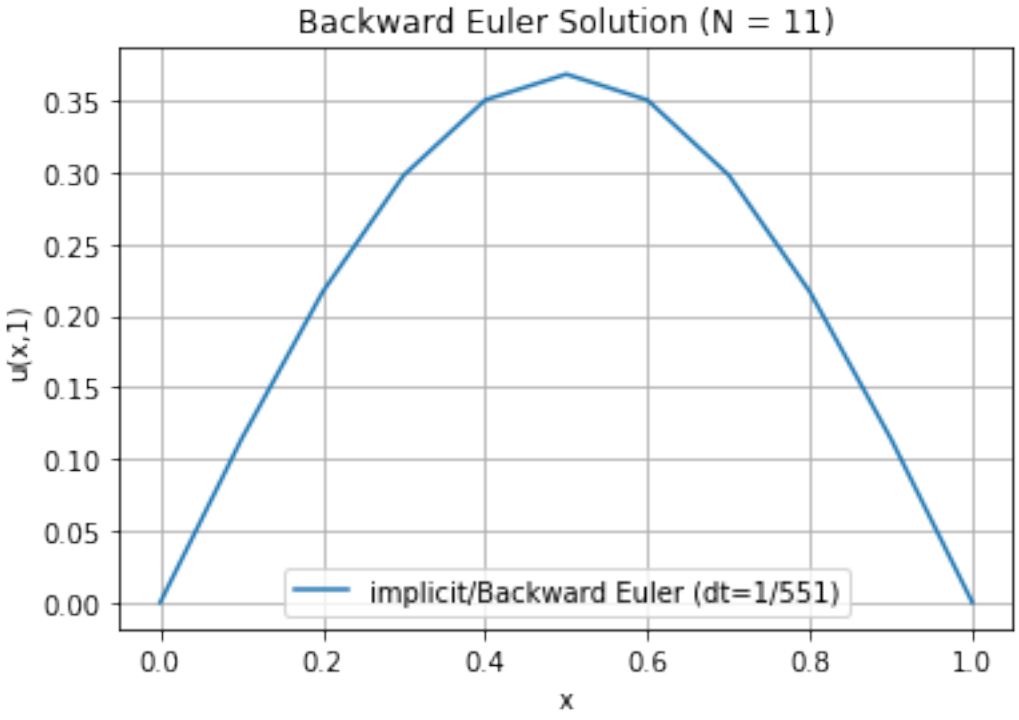
[39]:
```
#Main code, juypter persists data so I don't need to rewrite the code for␣
↪different cells

import numpy as np
import matplotlib.pyplot as plt

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

# gaussian points for 2 point rule
gp = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
gw = np.array([1, 1])

# number of nodes
N = 11

# mesh on [0,1]
x_nodes = np.linspace(0, 1, N)
num_elems = N - 1
h = x_nodes[1] - x_nodes[0]

# global matrices
M = np.zeros((N, N))
K = np.zeros((N, N))

# assemble M and K
for e in range(num_elems):
    xL = x_nodes[e]
    xR = x_nodes[e+1]
```

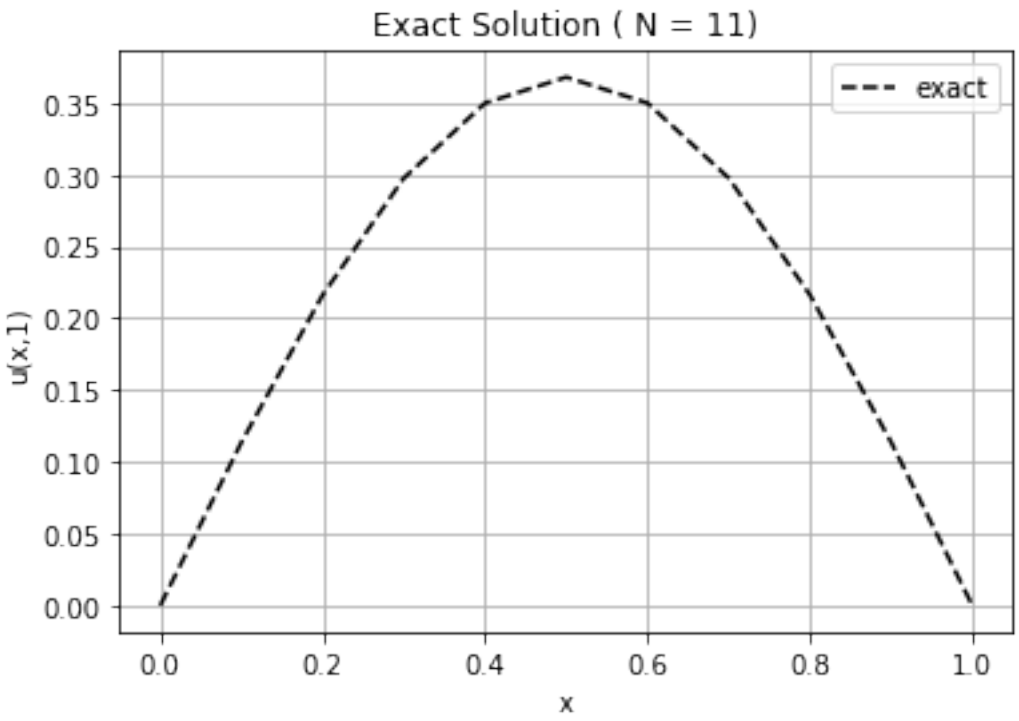
```
    detJ = (xR - xL) / 2.0

    Me = np.zeros((2,2))
    Ke = np.zeros((2,2))

    for q in range(2):
        xi = gp[q]
        w = gw[q]

        phi = np.array([(1 - xi)/2, (1 + xi)/2])
        dphi_dxi = np.array([-0.5, 0.5])
        dphi_dx = dphi_dxi / detJ

        Me += w * detJ * np.outer(phi, phi)
        Ke += w * detJ * np.outer(dphi_dx, dphi_dx)

    inds = [e, e+1]
    for i in range(2):
        for j in range(2):
            M[inds[i], inds[j]] += Me[i,j]
            K[inds[i], inds[j]] += Ke[i,j]

# build F vector
def build_F(t):
    F = np.zeros(N)
    for e in range(num_elems):
        xL = x_nodes[e]
        xR = x_nodes[e+1]
        detJ = (xR - xL) / 2.0

        Fe = np.zeros(2)

        for q in range(2):
            xi = gp[q]
            w = gw[q]
            phi = np.array([(1 - xi)/2, (1 + xi)/2])
            x_q = detJ * xi + (xL + xR) / 2.0
            Fe += w * detJ * phi * f_func(x_q, t)

        inds = [e, e+1]
        for i in range(2):
            F[inds[i]] += Fe[i]

    return F

# initial condition
```

```
u0 = np.sin(np.pi * x_nodes)

# time settings
dt = 1.0 / 551.0
t_final = 1.0
steps = int(t_final / dt)


# explicit forward euler


# lumped mass matrix
M_lumped = np.sum(M, axis=1)
M_inv = 1.0 / M_lumped

u_exp = u0.copy()

for step in range(steps):
    t = step * dt
    F = build_F(t)
    rhs = F - K @ u_exp
    u_exp = u_exp + dt * (rhs * M_inv)

    # enforce BC
    u_exp[0] = 0
    u_exp[-1] = 0


# implicit backward euler


u_imp = u0.copy()

A = M + dt * K
A[0,:] = 0
A[-1,:] = 0
A[:,0] = 0
A[:,-1] = 0
A[0,0] = 1
A[-1,-1] = 1

A_inv = np.linalg.inv(A)

for step in range(steps):
    t = step * dt
    F = build_F(t)
    rhs = M @ u_imp + dt * F
```

```
    rhs[0] = 0
    rhs[-1] = 0
    u_imp = A_inv @ rhs

# exact solution
u_exact = np.exp(-1.0) * np.sin(np.pi * x_nodes)

# plot main comparison
# 1) Explicit Forward Euler only
plt.figure()
plt.plot(x_nodes, u_exp, label="explicit/Forward Euler (dt=1/551)")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.title("Forward Euler Solution (N = 11)")
plt.grid()
plt.legend()
plt.show()

# 2) Implicit Backward Euler only
plt.figure()
plt.plot(x_nodes, u_imp, label="implicit/Backward Euler (dt=1/551)")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.title("Backward Euler Solution (N = 11)")
plt.grid()
plt.legend()
plt.show()

# 3) Exact only
plt.figure()
plt.plot(x_nodes, u_exact, "--k", label="exact")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.title("Exact Solution ( N = 11)")
plt.grid()
plt.legend()
plt.show()
```


Forward Euler Solution (N = 11)
u(x,1)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.2
0.4
0.6
0.8
1.0
x
explicit/Forward Euler (dt=1/551)


Backward Euler Solution (N = 11)
u(x,1)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.2
0.4
0.6
0.8
1.0
x
implicit/Backward Euler (dt=1/551)



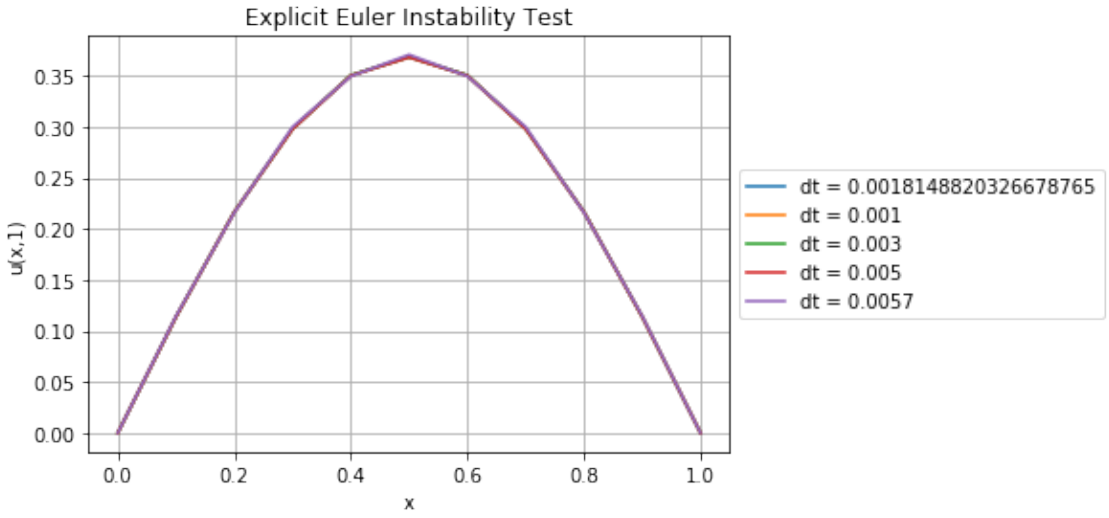
[31]:
```
print("As we can see the Forward Euler and Backward Euler all work and match␣
 ↪the exact solution very well. I am very proud of myself!.")
print("Now onto analyzing.")
```

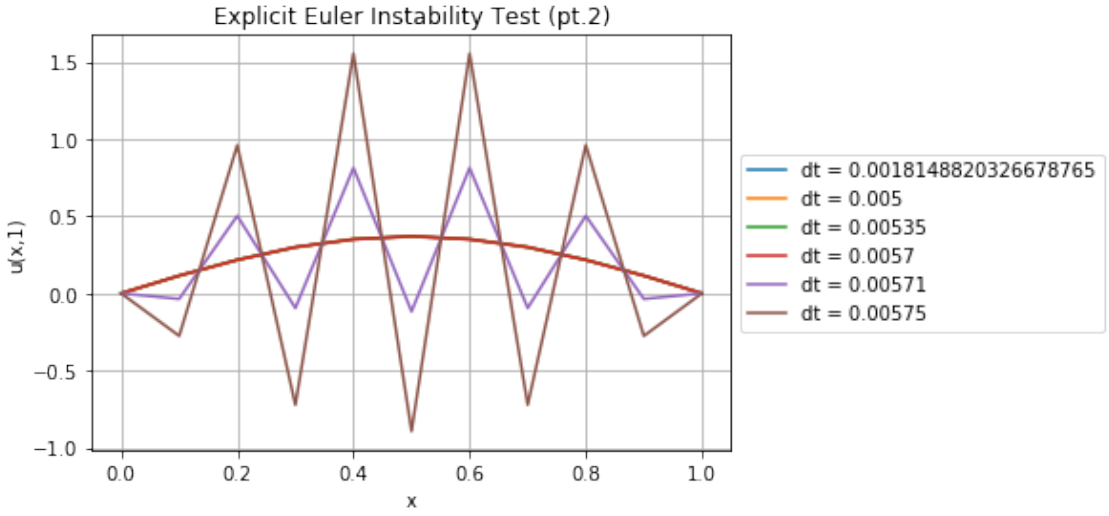
```
As we can see the Forward Euler and Backward Euler all work and match the exact
solution very well. I am very proud of myself!.
Now onto analyzing.
```

[38]:
```
# explicit euler instability test
dt_list = [1/551, 0.001, 0.003, 0.005,0.0057]

plt.figure()

for dt_test in dt_list:
    u_test = u0.copy()
    steps_test = int(t_final / dt_test)
    M_lumped = np.sum(M, axis=1)
    M_inv = 1.0 / M_lumped

    for step in range(steps_test):
        t = step * dt_test
        F = build_F(t)
        rhs = F - K @ u_test
```

```
        u_test = u_test + dt_test * (rhs * M_inv)
        u_test[0] = 0
        u_test[-1] = 0

        if np.isnan(np.max(u_test)):
            break

    plt.plot(x_nodes, u_test, label="dt = " + str(dt_test))

plt.title("Explicit Euler Instability Test")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.grid()
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()

# explicit euler instability test
dt_list = [1/551, 0.005,0.00535,0.0057,0.00571, 0.00575]

plt.figure()

for dt_test in dt_list:
    u_test = u0.copy()
    steps_test = int(t_final / dt_test)
    M_lumped = np.sum(M, axis=1)
    M_inv = 1.0 / M_lumped

    for step in range(steps_test):
        t = step * dt_test
        F = build_F(t)
        rhs = F - K @ u_test
        u_test = u_test + dt_test * (rhs * M_inv)
        u_test[0] = 0
        u_test[-1] = 0

        if np.isnan(np.max(u_test)):
            break

    plt.plot(x_nodes, u_test, label="dt = " + str(dt_test))

plt.title("Explicit Euler Instability Test (pt.2)")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.grid()
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
```

```
plt.show()
```





[21]:
```
print("CONTEXT: These are the graphs for the explicit Euler instability Test. ␣
↪The first figure is the Explicit Euler instability test while everything␣
↪does not blow up, the second graph shows everything blowing up. I had to␣
↪make different graphs because even having a difference of .0001, caused the␣
↪graph to explode such that you can barely see the heat equation anymore.")
```

CONTEXT: These are the graphs for the explicit Euler instability Test. The first figure is the Explicity Euler instability test while everything does not blow up, the second graph shows everything blowing up. I had to make different graphs because even having a difference of .0003, caused the graph to explode such that

you can barely see the heat equation anymore.

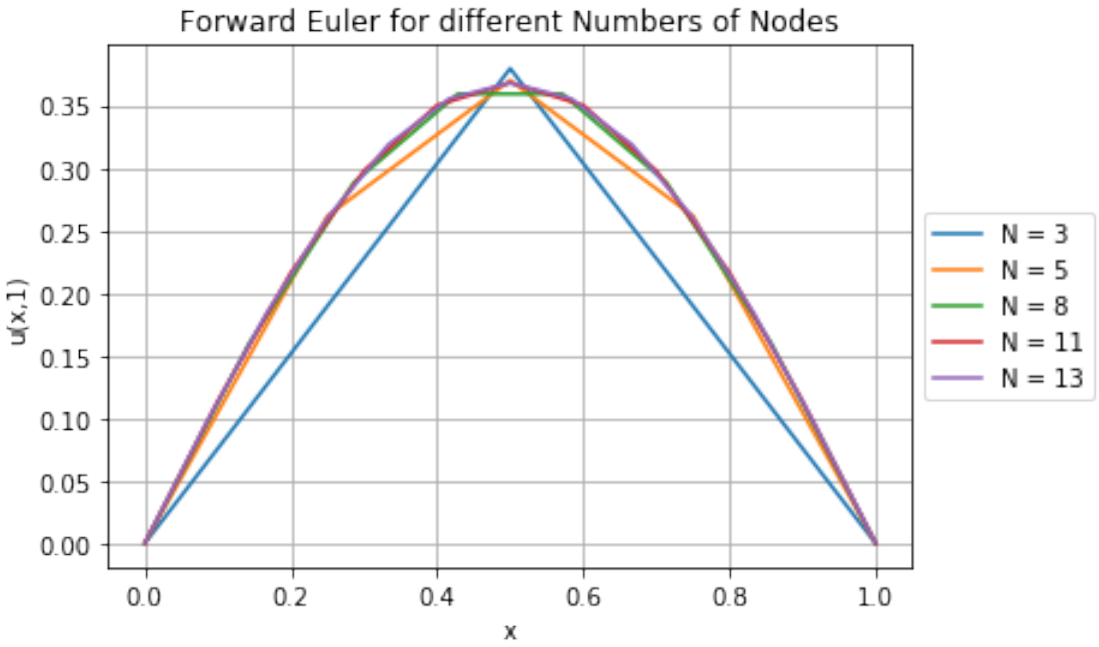
[13]:
```
print("As we can see from the first graph, we have a smooth continous stable␣
↪graph from dt's up until 0.0057, so much so that the lines overlap on␣
↪eachother. From there we can look at the second graph for values above 0.
↪0057, in these lines the start and end is the same ('0') which is enforced␣
↪by our boundary conditions but line of the dt's greater than 0.0057, start␣
↪oscillatiing indicating instability. From this, we can logically conclude␣
↪that the critical dt value for this heat diffusion function is 0.0057, and␣
↪for all dt's greater than 0.0057 there graphs will be unstable and thus␣
↪unreliable")
```

As we can see from the first graph, we have a smooth continous stable graph from dt's up until 0.0057. From there we can look at the second graph for values above 0.0057, in these lines the start and end is the same ('0') which is enforced by our boundary conditions but line of the dt's greater than 0.0057, start oscillatiing indicating instability. From this, we can logically conclude that the critical dt value for this heat diffusion function is 0.0057, and for all dt's greater than 0.0057 there lines graphs will be unstable and thus unreliable

[48]:
```
# forward euler solution for different N values
node_list = [3, 5, 8, 11,13]

plt.figure()

for Node in node_list:
    # build mesh
    x_nodes2 = np.linspace(0, 1, Node)
    num_elems2 = Node - 1
    h2 = x_nodes2[1] - x_nodes2[0]

    # new matrices for this N
    M2 = np.zeros((Node, Node))
    K2 = np.zeros((Node, Node))

    # assemble for this N
    for e in range(num_elems2):
        xL = x_nodes2[e]
        xR = x_nodes2[e+1]
        detJ = (xR - xL) / 2.0

        Me2 = np.zeros((2,2))
        Ke2 = np.zeros((2,2))

        for q in range(2):
            xi = gp[q]
```

```
            w = gw[q]
            phi = np.array([(1 - xi)/2, (1 + xi)/2])
            dphi_dxi = np.array([-0.5, 0.5])
            dphi_dx = dphi_dxi / detJ

            Me2 += w * detJ * np.outer(phi, phi)
            Ke2 += w * detJ * np.outer(dphi_dx, dphi_dx)

        inds = [e, e+1]
        for i in range(2):
            for j in range(2):
                M2[inds[i], inds[j]] += Me2[i,j]
                K2[inds[i], inds[j]] += Ke2[i,j]

    # initial condition for this N
    u2 = np.sin(np.pi * x_nodes2)

    # lumped mass for forward euler
    M2_lumped = np.sum(M2, axis=1)
    M2_inv = 1.0 / M2_lumped

    # explicit time stepping
    steps2 = int(t_final / dt)
    for step in range(steps2):
        t = step * dt

        # build F2
        F2 = np.zeros(Node)
        for e in range(num_elems2):
            xL = x_nodes2[e]
            xR = x_nodes2[e+1]
            detJ = (xR - xL) / 2.0

            Fe2 = np.zeros(2)
            for q in range(2):
                xi = gp[q]
                w = gw[q]
                phi = np.array([(1 - xi)/2, (1 + xi)/2])
                x_q = detJ * xi + (xL + xR) / 2.0
                Fe2 += w * detJ * phi * f_func(x_q, t)

            inds = [e, e+1]
            for i in range(2):
                F2[inds[i]] += Fe2[i]

        rhs2 = F2 - K2 @ u2
        u2 = u2 + dt * (rhs2 * M2_inv)
```

```
        # enforce BCs
        u2[0] = 0
        u2[-1] = 0

    plt.plot(x_nodes2, u2, label="N = " + str(Node))

plt.title("Forward Euler for different Numbers of Nodes")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.grid()
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```



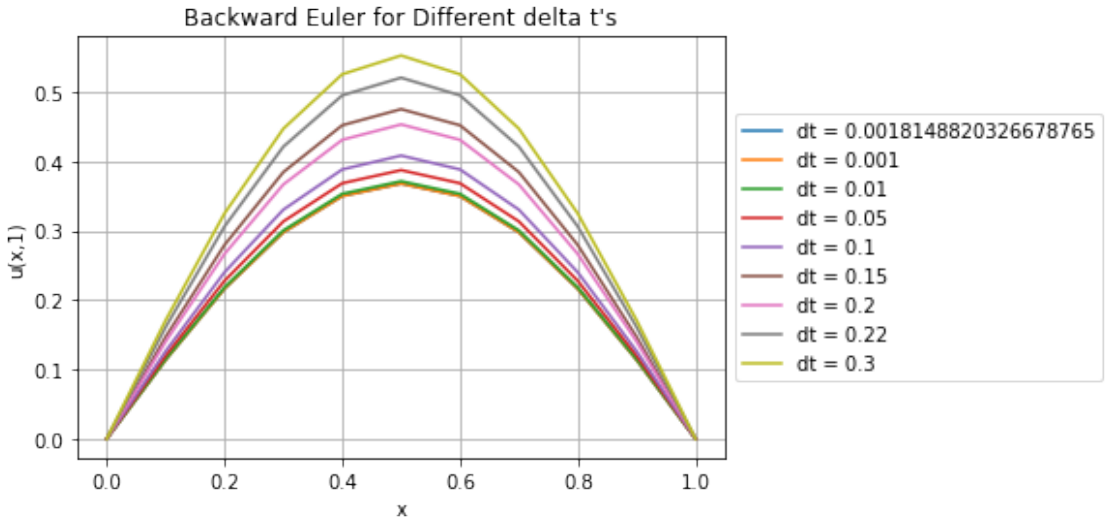
[50]:

```
print("In the plot above, I compared different values of N (nodes) to see how␣
 ↪the graph of my heat equation changes with regards to different nodes. When␣
 ↪I decrease N (so the mesh is coarser and each element is bigger), the␣
 ↪solution becomes more jagged or 'stiff' with steeper and more abrupt turns.
 ↪The peak near x=0.5 is also not captured well. Both the peak and the curve␣
 ↪look too triangular, or just not match the smooth sinusoidal exact shape for␣
 ↪small number of Nodes. As the amount of Nodes increase however, we see a␣
 ↪'smoothening' effect where the peaks are no longer as point and it follows␣
 ↪the smooth heat diffusion we expect. From this we can see that The Number of␣
 ↪nodes is sort of like the resolution of a camera, where each increase of␣
 ↪nodes exponentially increases the resolution of our 'image' or graph of the␣
 ↪function that we want to capture.")
```

In the plot above, I compared different values of N (nodes) to see how the graph of my heat equation changes with regards to different nodes. When I decrease N (so the mesh is coarser and each element is bigger), the solution becomes more jagged or 'stiff' with steeper and more abrupt turns.The peak near x=0.5 is also not captured well. Both the peak and the curve look too triangular, or just not match the smooth sinusoidal exact shape for small number of Nodes. As the amount of Nodes increase however, we see a 'smoothening' effect where the peaks are no longer as point and it follows the smooth heat diffusion we expect. From this we can see that The Number of nodes is sort of like the resolution of a camera, where each increase of nodes exponentially increases the resolution of our 'image' or graph of the function that we want to capture.

[56]:
```
dt_list_backward = [1/551, 0.001, 0.01, 0.05, 0.1,0.15, 0.2, 0.22, 0.3]

plt.figure()

for dt_tester in dt_list_backward:
    u_be = u0.copy()

    A = M + dt_tester * K
    A[0,:] = 0
    A[-1,:] = 0
    A[:,0] = 0
    A[:,-1] = 0
    A[0,0] = 1
    A[-1,-1] = 1

    A_inv = np.linalg.inv(A)

    steps_test = int(t_final / dt_tester)

    for step in range(steps_test):
        t = step * dt_tester
        F = build_F(t)
```

```
        rhs = M @ u_be + dt_tester * F
        rhs[0] = 0
        rhs[-1] = 0
        u_be = A_inv @ rhs

    plt.plot(x_nodes, u_be, label="dt = " + str(dt_tester))

plt.title("Backward Euler for Different delta t's")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.grid()
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



[51]:
```
print("In this graph, I graphed the behavior of Backward Euler at different␣
↪dt's. From the graph that regardless of the dt I provided, the graph stays␣
↪smooth, continous, and very much so models, at least in apperance, the␣
↪correct graph. However, we can also see that as the dt increases, even␣
↪though the curves are smooth, the larger dt's begin to overshoot the correct␣
↪curve. In other words, the dt stays stable but the error gets larger the␣
↪greater the dt. ")
```

```
In this graph, I graphed the behavior of Backward Euler at different dt's. From
the graph that regardless of the dt I provided, the graph stays smooth,
continous, and very much so models, at least in apperance, the correct graph.
However, we can also see that as the dt increases, even though the curves are
smooth, the larger dt's begin to overshoot the correct curve. In other words,
the dt stays stable but the error gets larger the greater the dt.
```

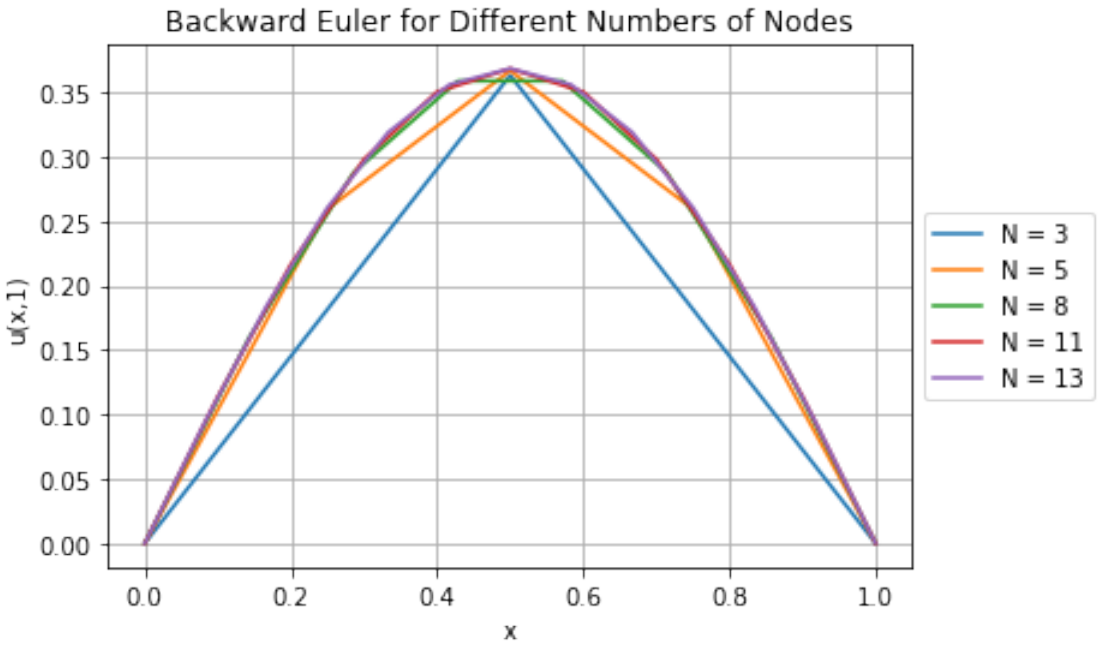
[57]:
```
print("This result appears due to a combination of things, The graph stays␣
 ↪stable because of the dampening effect of backward Euler. Because it is an␣
 ↪energy sink, the graph will never blow up but this can be dangerous because␣
 ↪the graph could return a stable but incorrect curve. Now onto the error. The␣
 ↪error comes from the fact that as explained in notes earlier in class,␣
 ↪backward euler is first order accurate in time which means that the bigger␣
 ↪the jumps in dt between the less reliable data the backward euler function␣
 ↪will use thus meaning the error will also increase. This can be seen␣
 ↪especially when the dt is greater than the special step size. The spatial␣
 ↪step size for the backward Euler heat equation is 0.1, derived from the 1/
 ↪(N-1) equation where N is 11. After the critical dt of 0.1, derived from␣
 ↪dt>=h, the curve remains stable but the error starts to increase␣
 ↪exponentially.")
```

```
This result appears due to a combination of things, The graph stays stable
because of the dampening effect of backward Euler. Because it is an energy sink,
the graph will never blow up but this can be dangerous because the graph could
return a stable but incorrect curve. Now onto the error. The error comes from
the fact that as explained in notes earlier in class, backward euler is first
order accurate in time which means that the bigger the jumps in dt between the
less reliable data the backward euler function will use thus meaning the error
will also increase. This can be seen especially when the dt is greater than the
special step size. The spatial step size for the backward Euler heat equation is
0.1, derived from the 1/(N-1) equation where N is 11. After the critical dt of
0.1, derived from dt>=h the error starts to increase exponentially.
```

[49]:
```
# test how the solution changes as the number of nodes changes for backward␣
 ↪euler
node_list = [3, 5, 8, 11,13]

plt.figure()

for Node in node_list:

    # remake the mesh
    x_nodes2 = np.linspace(0, 1, Node)
    num_elems2 = Node - 1
    h2 = x_nodes2[1] - x_nodes2[0]

    # make new matrices
    M2 = np.zeros((Node, Node))
    K2 = np.zeros((Node, Node))

    # assemble for this N
    for e in range(num_elems2):
        xL = x_nodes2[e]
        xR = x_nodes2[e+1]
```

```
        detJ = (xR - xL) / 2.0

        Me2 = np.zeros((2,2))
        Ke2 = np.zeros((2,2))

        for q in range(2):
            xi = gp[q]
            w = gw[q]
            phi = np.array([(1 - xi)/2, (1 + xi)/2])
            dphi_dxi = np.array([-0.5, 0.5])
            dphi_dx = dphi_dxi / detJ
            Me2 += w * detJ * np.outer(phi, phi)
            Ke2 += w * detJ * np.outer(dphi_dx, dphi_dx)

        inds = [e, e+1]
        for i in range(2):
            for j in range(2):
                M2[inds[i], inds[j]] += Me2[i,j]
                K2[inds[i], inds[j]] += Ke2[i,j]

    # initial condition for this N
    u2 = np.sin(np.pi * x_nodes2)

    # implicit since it's always stable
    A2 = M2 + dt * K2
    A2[0,:] = 0
    A2[-1,:] = 0
    A2[:,0] = 0
    A2[:,-1] = 0
    A2[0,0] = 1
    A2[-1,-1] = 1

    A2_inv = np.linalg.inv(A2)

    # time stepping for this N
    steps2 = int(t_final / dt)
    for step in range(steps2):
        t = step * dt
        F2 = np.zeros(Node)
        # build F for this N
        for e in range(num_elems2):
            xL = x_nodes2[e]
            xR = x_nodes2[e+1]
            detJ = (xR - xL) / 2.0
            Fe2 = np.zeros(2)
            for q in range(2):
                xi = gp[q]
```

```
                w = gw[q]
                phi = np.array([(1 - xi)/2, (1 + xi)/2])
                x_q = detJ * xi + (xL + xR) / 2.0
                Fe2 += w * detJ * phi * f_func(x_q, t)
            inds = [e, e+1]
            for i in range(2):
                F2[inds[i]] += Fe2[i]

        rhs2 = M2 @ u2 + dt * F2
        rhs2[0] = 0
        rhs2[-1] = 0
        u2 = A2_inv @ rhs2

    plt.plot(x_nodes2, u2, label="N = " + str(Node))

plt.title("Backward Euler for Different Numbers of Nodes")
plt.xlabel("x")
plt.ylabel("u(x,1)")
plt.grid()
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



[45]:

```
print("In the plot above, very similalry to the Forward Euler Node test, we see␣
 ↪that as the amount of nodes increase, the 'resolution' of the graph becomes␣
 ↪smoother, less jagged, and overall more correct. That being said, I would␣
 ↪like to highlight an interesting difference I have seen in the graphs, for␣
 ↪Forward Euler, at peaks of less 'resolution' nodes i.e lower nodes, we would␣
 ↪see an overshoot at the peaks but in this graph we see an undershoot. This␣
 ↪further demonstrates how the Forward Euler is an energy adder whereas the␣
 ↪Backward Euler is an energy sinker and dampner. This was a very cool thing␣
 ↪that I just happened to see.")
```

```
In the plot above, very similalry to the Forward Euler Node test, we see that as
the amount of nodes increase, the 'resolution' of the graph becomes smoother,
less jagged, and overall more correct. That being said, I would like to
highlight an interesting difference I have seen in the graphs, for Forward
Euler, at peaks of less 'resolution' nodes i.e lower nodes, we would see an
overshoot at the peaks but in this graph we see an undershoot. This further
demonstrates how the Forward Euler is an energy adder whereas the Backward Euler
is an energy sinker and dampner. This was a very cool thing that I just happened
to see.
```

[ ]: