Project 1

Omkar Mulekar
AERO 4630-002
Aerospace Structural Dynamics

8 February 2019

Problem 1: Method of Manufactured Solutions

Part 1a: Horizontal Change

For $u(x) = x_1^3$ and $f(x) = -6x_1$, the relationship

$$-\nabla^2 u(x) = f(x), x \in \omega \tag{1}$$

can be shown to be true by taking the Laplacian of $u(x)$.

$$\frac{\partial^2 u(x)}{\partial x_1^2} = \frac{\partial}{\partial x_1}\frac{\partial}{\partial x_1}(x_1^3)$$

$$= \frac{\partial}{\partial x_1}(3x_1^2) = 6x = -f(x)$$

The Python script used to discretize and solve the relationship is shown **Appendix 1**. The Paraview output of is shown in Figure 1.

The error reported by the script output is shown below:

```
error_L2   = 0.004931859322266561
error_max = 3.33066907388e-16
```

Part 1b: Vertical Change

This relationship can be shown for $u(x) = x_2^3$ and $f(x) = -6x_2$ using the same steps as were used for $u(x) = x_1^3$ and $f(x) = -6x_1$. The error from the python script used was the same as well. The Paraview output is shown in Figure 2, and the Python script used is given in **Appendix 1b**.

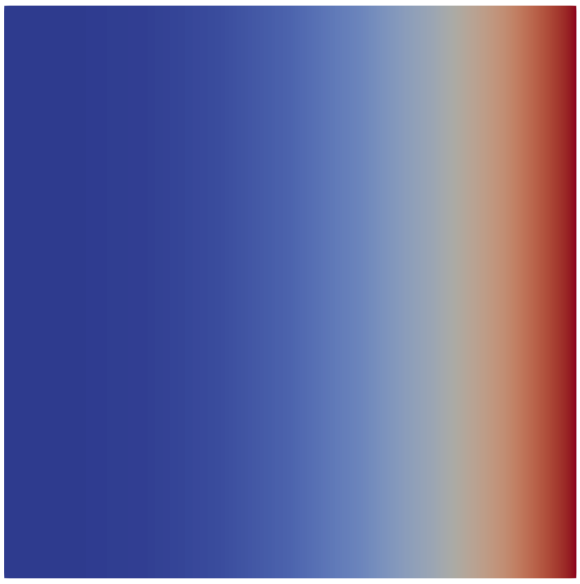


Figure 1: Paraview output for problem 1a

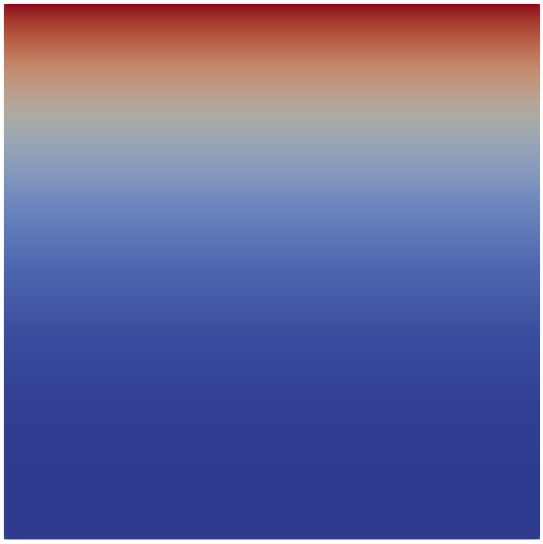


Figure 2: Paraview output for problem 1b

Problem 2: Improving Accuracy

Part 2a: Improving Through Mesh Resolution

The error was reduced by increasing the mesh resolution in the python script. Mesh resolutions of 4, 8, 16, 32, and 64 produced different errors. These are shown in Figure 3.

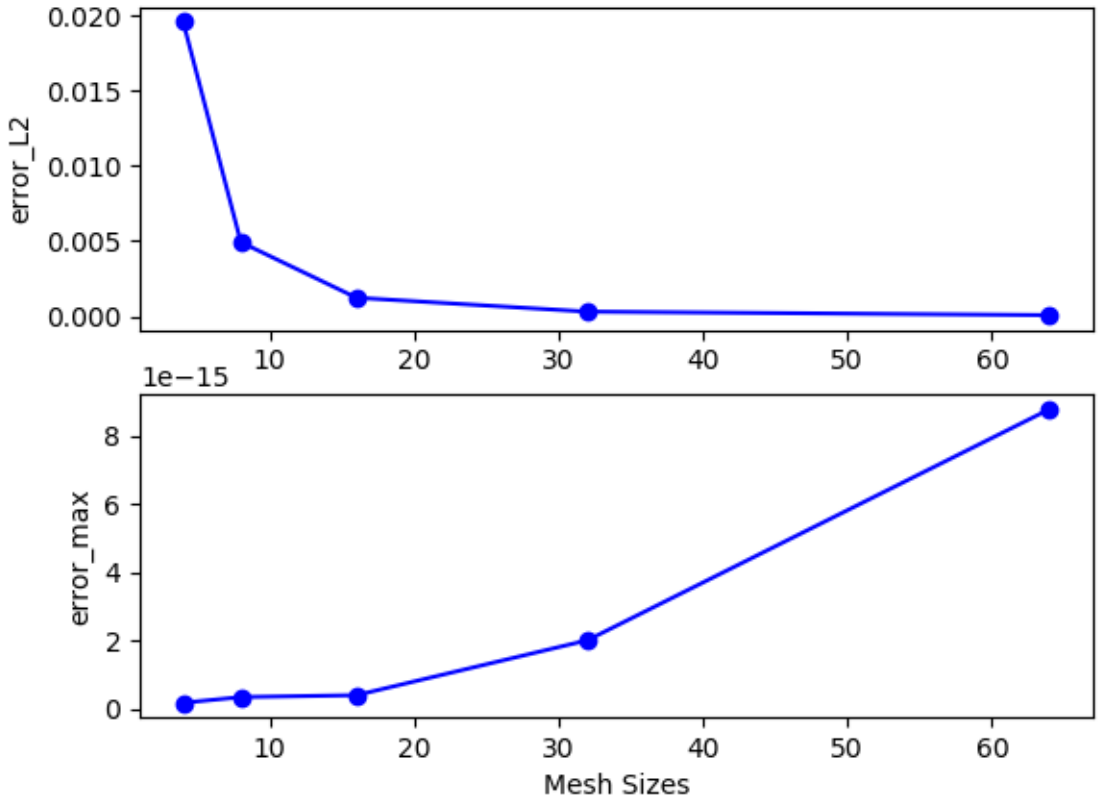


Figure 3: Error vs Mesh Sizing for Problem 2a

The code used to assess mesh refinement is provided in **Appendix 2a**.

Part 2b: Improving Through the Shape Function

The error was reduced by increasing the order of or the shape functions. Orders of 1, 2, 3, 4, and 5 were used. A plot of the error produced over the shape function order is given in Figure 4

The code used to assess shape function order is provided in **Appendix 2b**.

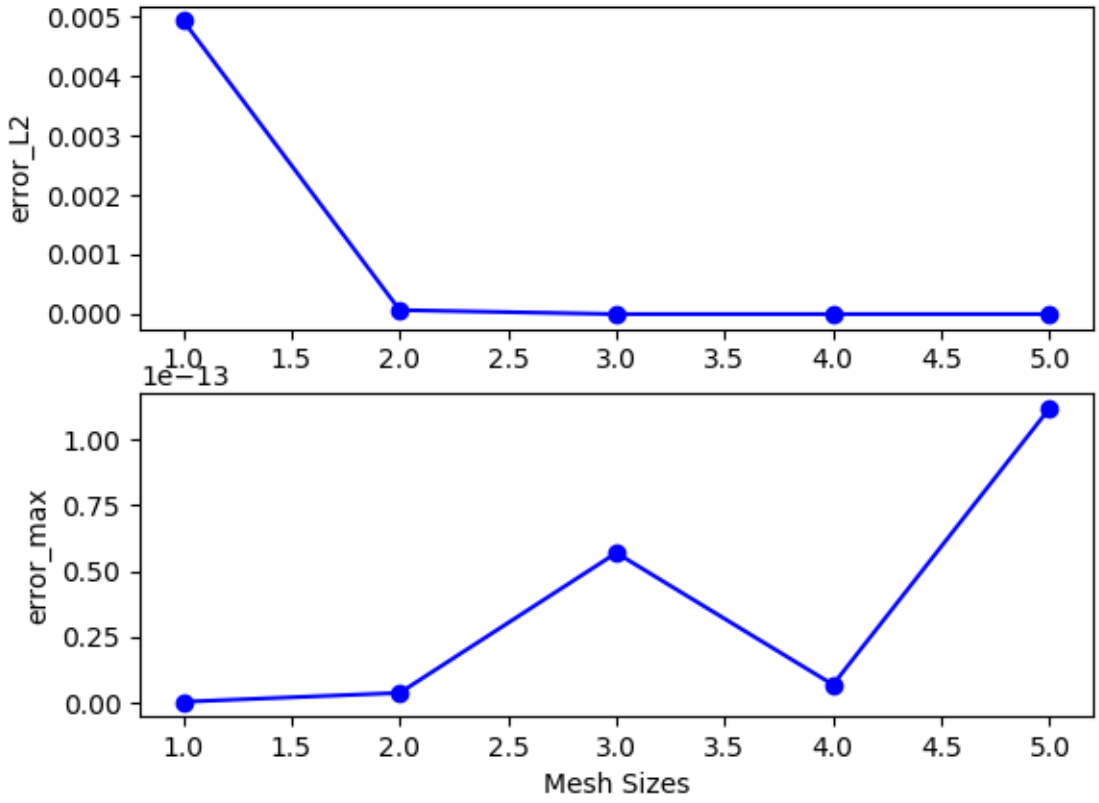


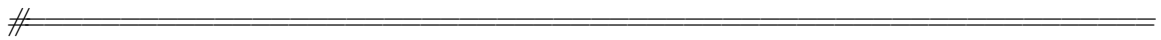
Figure 4: Error vs Mesh Sizing for Problem 2a

Appendix 1a: Code for Problem 1

```
"""
Python script for 1a of Project 1

Original Author:          Vinamra Agrawal
Date:                     January 25, 2019

Edited By:                Omkar Mulekar
Date:                     January 30, 2019

        -Laplace(u) = f     in the unit square
        u = u_D   on the boundary

 In this problem,
        u_D = x^3
    f = -6x

"""

#=============================================================
# This statement needs to be added for most (if not all)
# scripts for Fenics
from __future__ import print_function
from fenics import *
import matplotlib.pyplot as plt
#=============================================================

#=============================================================
# This is to create a mesh.
# In the class we saw how our rectangular domain was split
# into triangles. This function, by defualt, creates a box
# of dimensions 1x1, with 8 intervals on each edge.
# These intervals are connected through triangles.
mesh = UnitSquareMesh(8, 8)
#=============================================================

#=============================================================
# This is to choose what kind of function space we are dealing
# with. Essentially these are shape functions.
# Don't worry, we will do shape functions in class later.
V = FunctionSpace(mesh, 'P', 1)
#=============================================================
```

```
#=======================================================================
# This statement defines the boundary condition u_D
# "Expression" is a convenient way to declare complicated
# expressions in Fenics. On the back-end, a program reads this
# and converts it in a format, the code can work with.
# This is a feature to make your life easier.
# degree = 2 tells Fenics, the degree of the polynomial.
u_D = Expression('x[0]*x[0]*x[0]', degree=3)
#=======================================================================

#=======================================================================
# This is a "function" called boundary, that takes in two inputs
#               x                       - location of a point
#               on_boundary - location of the boundary
# Right now, this function is not doing much. It is simply
# returning the location of the boundary. In the future, we will
# mess with this function as well. We will see that it is
# extremely useful.
def boundary(x, on_boundary):
    return on_boundary
#=======================================================================

#=======================================================================
# This is where we actually define the boundary condition
# Fenics only needs you to define Dirichlet (or displacement)
# boundary condition. You define the Neumann (or force)
# boundary condition in a different way.
# This function takes in three inputs
#       V                       - the function space (shape functions)
#       u_D             - the expression of the dirichlet boundary
#       boundary        - a function (declared above) that tells you
#                                 exactly where the boundary is
bc = DirichletBC(V, u_D, boundary)
#=======================================================================

#=======================================================================
# This is where you define the weak form of the equation
# First step:   tell Fenics to start with u as a trial function
#                               in the function space V
u = TrialFunction(V)
# Second step:  tell Fenics to choose v(x) as a test function
#                               in the function space V.
v = TestFunction(V)
# Third step:   Define f(x)
f = Expression('-6*x[0]', degree=1)
```

```
# Fourth step:  Define the left hand side of the weak form
a = dot(grad(u), grad(v))*dx
# Fifth step:   Define the right hand side of the weak form
L = f*v*dx
#=============================================================

#=============================================================
# This chunk of code computes the solution.
# First, move away from trial function u and make it a function.
# Basically it tells Fenics to make u as a combination of
# shape functions. We'll do this later too.
u = Function(V)
# Finally... let it run. Solve LHS = RHS, for u given BC.
solve(a == L, u, bc)
#=============================================================

#=============================================================
# Save solution to file in VTK format.
# In the class, we visualized the file in Paraview.
vtkfile = File('poisson/solution.pvd')
vtkfile << u
#=============================================================

#=============================================================
# Now we test if our code is working. This is where we use the
# method of manufactured solutions. We know that
#       u = 1 + x^2 + 2y^2
# satisfies the PDE. That's why we chose this as the boundary
# condition. u = u_D is automatically satisfied.

# This line computes the L2 error between the computed solution
# u and the the solution we expect u_D
error_L2 = errornorm(u_D, u, 'L2')

# Next, we find the value of the solution (expected and computed)
# at each vertex of the mesh. Then we compare the two. Our goal here
# is to find the maximum error we get.
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))

# Print errors
print('error_L2  =', error_L2)
print('error_max =', error_max)
```

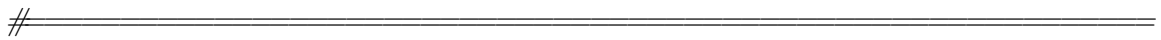


Appendix 1b: Code for Problem 1

```
"""
Python script for 1b of Project 1

Original Author:            Vinamra Agrawal
Date:                       January 25, 2019

Edited By:                  Omkar Mulekar
Date:                       January 30, 2019

        -Laplace(u) = f     in the unit square
        u = u_D   on the boundary

 In this problem,
        u_D = x^3
    f = -6x

"""

#=============================================================
# This statement needs to be added for most (if not all)
# scripts for Fenics
from __future__ import print_function
from fenics import *
import matplotlib.pyplot as plt
#=============================================================

#=============================================================
# This is to create a mesh.
# In the class we saw how our rectangular domain was split
# into triangles. This function, by defualt, creates a box
# of dimensions 1x1, with 8 intervals on each edge.
# These intervals are connected through triangles.
mesh = UnitSquareMesh(8, 8)
#=============================================================

#=============================================================
# This is to choose what kind of function space we are dealing
# with. Essentially these are shape functions.
# Don't worry, we will do shape functions in class later.
V = FunctionSpace(mesh, 'P', 1)
#=============================================================
```

```
#=============================================================
# This statement defines the boundary condition u_D
# "Expression" is a convenient way to declare complicated
# expressions in Fenics. On the back-end, a program reads this
# and converts it in a format, the code can work with.
# This is a feature to make your life easier.
# degree = 2 tells Fenics, the degree of the polynomial.
u_D = Expression('x[1]*x[1]*x[1]', degree=3)
#=============================================================

#=============================================================
# This is a "function" called boundary, that takes in two inputs
#               x                       - location of a point
#               on_boundary - location of the boundary
# Right now, this function is not doing much. It is simply
# returning the location of the boundary. In the future, we will
# mess with this function as well. We will see that it is
# extremely useful.
def boundary(x, on_boundary):
    return on_boundary
#=============================================================

#=============================================================
# This is where we actually define the boundary condition
# Fenics only needs you to define Dirichlet (or displacement)
# boundary condition. You define the Neumann (or force)
# boundary condition in a different way.
# This function takes in three inputs
#       V                       - the function space (shape functions)
#       u_D             - the expression of the dirichlet boundary
#       boundary        - a function (declared above) that tells you
#                                 exactly where the boundary is
bc = DirichletBC(V, u_D, boundary)
#=============================================================

#=============================================================
# This is where you define the weak form of the equation
# First step:   tell Fenics to start with u as a trial function
#                               in the function space V
u = TrialFunction(V)
# Second step:  tell Fenics to choose v(x) as a test function
#                               in the function space V.
v = TestFunction(V)
# Third step:   Define f(x)
f = Expression('-6*x[1]', degree=1)
```

```
# Fourth step:  Define the left hand side of the weak form
a = dot(grad(u), grad(v))*dx
# Fifth step:   Define the right hand side of the weak form
L = f*v*dx
#=============================================================

#=============================================================
# This chunk of code computes the solution.
# First, move away from trial function u and make it a function.
# Basically it tells Fenics to make u as a combination of
# shape functions. We'll do this later too.
u = Function(V)
# Finally... let it run. Solve LHS = RHS, for u given BC.
solve(a == L, u, bc)
#=============================================================

#=============================================================
# Save solution to file in VTK format.
# In the class, we visualized the file in Paraview.
vtkfile = File('poisson/solution.pvd')
vtkfile << u
#=============================================================

#=============================================================
# Now we test if our code is working. This is where we use the
# method of manufactured solutions. We know that
#       u = 1 + x^2 + 2y^2
# satisfies the PDE. That's why we chose this as the boundary
# condition. u = u_D is automatically satisfied.

# This line computes the L2 error between the computed solution
# u and the the solution we expect u_D
error_L2 = errornorm(u_D, u, 'L2')

# Next, we find the value of the solution (expected and computed)
# at each vertex of the mesh. Then we compare the two. Our goal here
# is to find the maximum error we get.
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))

# Print errors
print('error_L2  =', error_L2)
print('error_max =', error_max)
```



Appendix 2a: Code for Problem 2a

```
"""
Python script for 2a of Project 1

Original Author:            Vinamra Agrawal
Date:                       January 25, 2019

Edited By:                  Omkar Mulekar
Date:                       January 30, 2019

        -Laplace(u) = f     in the unit square
        u = u_D   on the boundary

 In this problem,
        u_D = x^3
    f = -6x

"""

#=============================================================
# This statement needs to be added for most (if not all)
# scripts for Fenics
from __future__ import print_function
from fenics import *
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
#=============================================================

#=============================================================
# This is to create a mesh.
# In the class we saw how our rectangular domain was split
# into triangles. This function, by defualt, creates a box
# of dimensions 1x1, with 8 intervals on each edge.
# These intervals are connected through triangles.

# Define Mesh Sizes and initialize error variables
meshSizes = [4,8,16,32,64]
error_L2  = [0] * len(meshSizes)
error_max = [0] * len(meshSizes)

for i in range(len(meshSizes)):
        mesh = UnitSquareMesh(meshSizes[i], meshSizes[i])
```

```
#=============================================================

#=============================================================
# This is to choose what kind of function space we are dealing
# with. Essentially these are shape functions.
# Don't worry, we will do shape functions in class later.
V = FunctionSpace(mesh, 'P', 1)
#=============================================================

#=============================================================
# This statement defines the boundary condition u_D
# "Expression" is a convenient way to declare complicated
# expressions in Fenics. On the back-end, a program reads this
# and converts it in a format, the code can work with.
# This is a feature to make your life easier.
# degree = 2 tells Fenics, the degree of the polynomial.
u_D = Expression('x[0]*x[0]*x[0]', degree=3)
#=============================================================

#=============================================================
# This is a "function" called boundary, that takes in two inputs
#                x                          - location of a point
#                on_boundary - location of the boundary
# Right now, this function is not doing much. It is simply
# returning the location of the boundary. In the future, we will
# mess with this function as well. We will see that it is
# extremely useful.
def boundary(x, on_boundary):
    return on_boundary
#=============================================================

#=============================================================
# This is where we actually define the boundary condition
# Fenics only needs you to define Dirichlet (or displacement)
# boundary condition. You define the Neumann (or force)
# boundary condition in a different way.
# This function takes in three inputs
#       V                          - the function space (shape functions
#       u_D              - the expression of the dirichlet boundary
#       boundary         - a function (declared above) that tells you
#                                   exactly where the boundary is
bc = DirichletBC(V, u_D, boundary)
#=============================================================

#=============================================================
```

```
# This is where you define the weak form of the equation
# First step:   tell Fenics to start with u as a trial function
#                                 in the function space V
u = TrialFunction(V)
# Second step:  tell Fenics to choose v(x) as a test function
#                                 in the function space V.
v = TestFunction(V)
# Third step:   Define f(x)
f = Expression('-6*x[0]', degree=1)
# Fourth step:  Define the left hand side of the weak form
a = dot(grad(u), grad(v))*dx
# Fifth step:   Define the right hand side of the weak form
L = f*v*dx
#=============================================================

#=============================================================
# This chunk of code computes the solution.
# First, move away from trial function u and make it a function.
# Basically it tells Fenics to make u as a combination of
# shape functions. We'll do this later too.
u = Function(V)
# Finally... let it run. Solve LHS = RHS, for u given BC.
solve(a == L, u, bc)
#=============================================================

#=============================================================
# Save solution to file in VTK format.
# In the class, we visualized the file in Paraview.
vtkfile = File('poisson/solution.pvd')
vtkfile << u
#=============================================================

#=============================================================
# Now we test if our code is working. This is where we use the
# method of manufactured solutions. We know that
#       u = 1 + x^2 + 2y^2
# satisfies the PDE. That's why we chose this as the boundary
# condition. u = u_D is automatically satisfied.

# This line computes the L2 error between the computed solution
# u and the the solution we expect u_D
error_L2[i] = errornorm(u_D, u, 'L2')

# Next, we find the value of the solution (expected and computed)
# at each vertex of the mesh. Then we compare the two. Our goal here
```

```
        # is to find the maximum error we get.
        vertex_values_u_D = u_D.compute_vertex_values(mesh)
        vertex_values_u = u.compute_vertex_values(mesh)
        import numpy as np
        error_max[i] = np.max(np.abs(vertex_values_u_D - vertex_values_u))

        # Print errors
        print('meshSizes =', meshSizes)
        print('error_L2  =', error_L2)
        print('error_max =', error_max)
        #=============================================================

plt.figure(1)
plt.subplot(211)
plt.plot(meshSizes,error_L2,'bo-')
plt.ylabel('error_L2')
plt.subplot(212)
plt.plot(meshSizes,error_max,'bo-')
plt.xlabel('Mesh Sizes')
plt.ylabel('error_max')
plt.savefig('2afig1.png')
```

Appendix 2a: Code for Problem 2b

```
"""
Python script for 2b of Project 1

Original Author:            Vinamra Agrawal
Date:                       January 25, 2019

Edited By:                  Omkar Mulekar
Date:                       January 30, 2019

        -Laplace(u) = f     in the unit square
        u = u_D   on the boundary

 In this problem,
        u_D = x^3
    f = -6x

"""

#=============================================================
# This statement needs to be added for most (if not all)
# scripts for Fenics
from __future__ import print_function
from fenics import *
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
#=============================================================

#=============================================================
# This is to create a mesh.
# In the class we saw how our rectangular domain was split
# into triangles. This function, by defualt, creates a box
# of dimensions 1x1, with 8 intervals on each edge.
# These intervals are connected through triangles.

# Define Mesh Sizes and initialize error variables
order = [1,2,3,4,5]
error_L2  = [0] * len(order)
error_max = [0] * len(order)

for i in range(len(order)):
        mesh = UnitSquareMesh(8, 8)
```

```
#=============================================================

#=============================================================
# This is to choose what kind of function space we are dealing
# with. Essentially these are shape functions.
# Don't worry, we will do shape functions in class later.
V = FunctionSpace(mesh, 'P', order[i])
#=============================================================

#=============================================================
# This statement defines the boundary condition u_D
# "Expression" is a convenient way to declare complicated
# expressions in Fenics. On the back-end, a program reads this
# and converts it in a format, the code can work with.
# This is a feature to make your life easier.
# degree = 2 tells Fenics, the degree of the polynomial.
u_D = Expression('x[0]*x[0]*x[0]', degree=3)
#=============================================================

#=============================================================
# This is a "function" called boundary, that takes in two inputs
#               x                       - location of a point
#               on_boundary - location of the boundary
# Right now, this function is not doing much. It is simply
# returning the location of the boundary. In the future, we will
# mess with this function as well. We will see that it is
# extremely useful.
def boundary(x, on_boundary):
    return on_boundary
#=============================================================

#=============================================================
# This is where we actually define the boundary condition
# Fenics only needs you to define Dirichlet (or displacement)
# boundary condition. You define the Neumann (or force)
# boundary condition in a different way.
# This function takes in three inputs
#       V                       - the function space (shape functions
#       u_D             - the expression of the dirichlet boundary
#       boundary        - a function (declared above) that tells you
#                                 exactly where the boundary is
bc = DirichletBC(V, u_D, boundary)
#=============================================================

#=============================================================
```

```
# This is where you define the weak form of the equation
# First step:   tell Fenics to start with u as a trial function
#                                 in the function space V
u = TrialFunction(V)
# Second step:  tell Fenics to choose v(x) as a test function
#                                 in the function space V.
v = TestFunction(V)
# Third step:   Define f(x)
f = Expression('-6*x[0]', degree=1)
# Fourth step:  Define the left hand side of the weak form
a = dot(grad(u), grad(v))*dx
# Fifth step:   Define the right hand side of the weak form
L = f*v*dx
#=============================================================

#=============================================================
# This chunk of code computes the solution.
# First, move away from trial function u and make it a function.
# Basically it tells Fenics to make u as a combination of
# shape functions. We'll do this later too.
u = Function(V)
# Finally... let it run. Solve LHS = RHS, for u given BC.
solve(a == L, u, bc)
#=============================================================

#=============================================================
# Save solution to file in VTK format.
# In the class, we visualized the file in Paraview.
vtkfile = File('poisson/solution.pvd')
vtkfile << u
#=============================================================

#=============================================================
# Now we test if our code is working. This is where we use the
# method of manufactured solutions. We know that
#       u = 1 + x^2 + 2y^2
# satisfies the PDE. That's why we chose this as the boundary
# condition. u = u_D is automatically satisfied.

# This line computes the L2 error between the computed solution
# u and the the solution we expect u_D
error_L2[i] = errornorm(u_D, u, 'L2')

# Next, we find the value of the solution (expected and computed)
# at each vertex of the mesh. Then we compare the two. Our goal here
```

```
        # is to find the maximum error we get.
        vertex_values_u_D = u_D.compute_vertex_values(mesh)
        vertex_values_u = u.compute_vertex_values(mesh)
        import numpy as np
        error_max[i] = np.max(np.abs(vertex_values_u_D - vertex_values_u))

        # Print errors
        print('order =', order)
        print('error_L2  =', error_L2)
        print('error_max =', error_max)
        #=============================================================

plt.figure(1)
plt.subplot(211)
plt.plot(order,error_L2,'bo-')
plt.ylabel('error_L2')
plt.subplot(212)
plt.plot(order,error_max,'bo-')
plt.xlabel('Mesh Sizes')
plt.ylabel('error_max')
plt.savefig('2bfig1.png')
```