HOMEWORK 5

CEE 361-513: Introduction to Finite Element Methods

Due: Friday Nov. 10 @ Midnight

NB: Students taking CEE 513 must complete all problems. All other students will not be graded for problems marked with $\star$, but are encourage to attempt them anyhow.

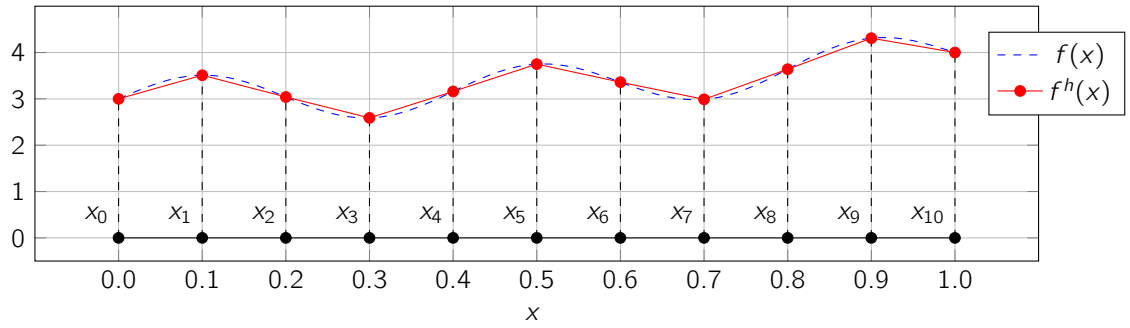
PROBLEM 1: Lagrange polynomials and approximation error

Preamble

A very important family of functions for finite element methods are called the *Lagrange* polynomials. This polynomials are very useful for *interpolating* functions.

As we saw in class, an interpolation $f^h(x)$ of a function $f(x)$ is a function that at certain points (sometimes called nodes or knots), let's denote them $x_i, i = 0 \dots n$, satisfies the condition $f^h(x_i) = f(x_i)$.

An example is the plot below where the function $f(x)$ is interpolated by $f^h(x)$. The function $f^h(x)$ consists piecewise linear polynomial. Namely, between the nodes x_i and x_{i+1} the function $f^h(x)$ is a linear (in reality, an affine) function.



As we saw in class the function $f^h(x)$ can be written as the linear combination of *basis functions* $\phi_i(x)$ and nodal values $f_i = f(x_i)$ such that

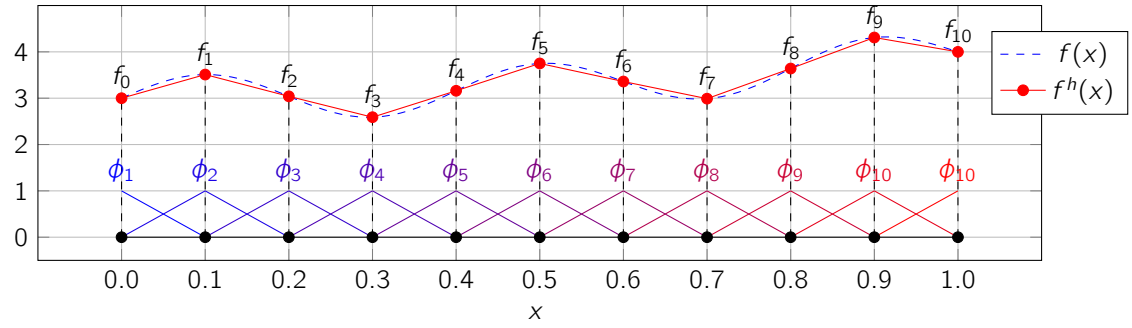
$$f^h(x) = \sum_{i=1}^{n} f_i \, \phi_i(x)$$

where the interpolation properties stems from the really nice property that $\phi_i(x_j) = \delta_{ij}$; namely, at the i^{th} node all basis functions $\phi_j(x)$ are zero with the only exception of $\phi_{j=i}(x)$. See the plot below for a graphical interpretation.

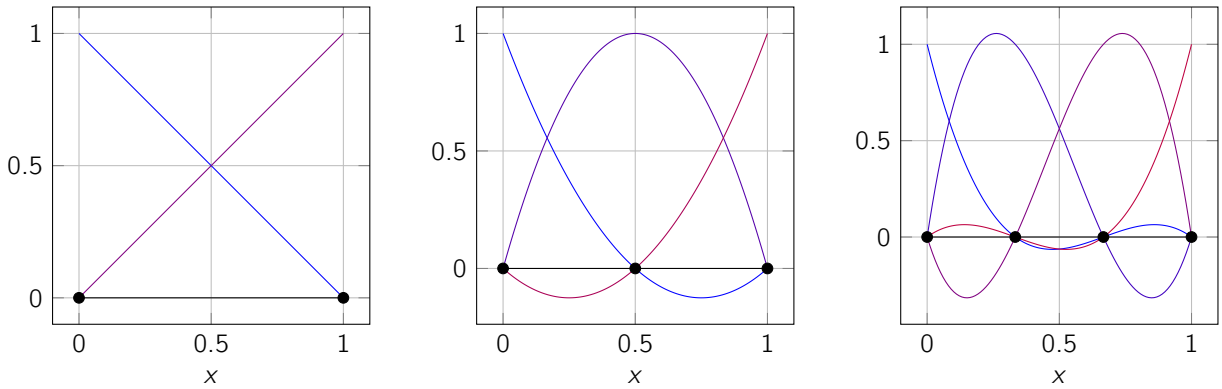
We remark again that the superscript h here refers to the average element size

$$h \approx 1/(n-1) \sum_{i=1}^{n-1} |x_{i+1} - x_i|.$$

If we *refine the mesh*, namely we increase the number of elements, the average mesh size will decrease and the number of basis functions will increase, hence we will have a better approximation to f (effectively as $h \downarrow$, $|f - f^h| \downarrow$).



Lagrange polynomials basis $\ell_i^p(x), i = 0, \ldots, p$ constructed on a set of $p+1$ nodes $(x_i, i = 0, \ldots, p)$ represent a set of $p+1$ polynomials of order p (eg. if $p = 1$ we have 2 linear polynomials constructed onto 2 nodes, if $p = 2$ we have 3 quadratic polynomials constructed onto 3 nodes, etc) which satisfy $\ell_i(x_j) = \delta_{ij}$. The plots below show the Lagrange polynomial basis for $p = 1, 2, 3$. As you can see the basis are polynomials of order p and further they satisfy $\ell_i(x_j) = \delta_{ij}$.



(a) $p = 1$ (b) $p = 2$ (c) $p = 3$

Figure 1: Showcasing the Lagrange polynomial basis

With the above Lagrange polynomial basis one can construct an approximation to any function as

$$f^p(x) = \sum_{i=0}^{p} f_i \, \ell_i^p(x).$$

Problem

With the above preamble we move on to the actual problem:

1. Implement a function that, given a set of *nodes* $x_i, i = 0 \ldots p$, returns the *Lagrange* basis function for knot i. The expression for the i^{th} basis function for the p^{th} order polynomials is given by[1]

$$\ell_i^p(x) = \prod_{\substack{0 \le j \le p \\ i \neq j}} \frac{x - x_j}{x_i - x_j}. \tag{1}$$

   Use the following code as a starter code (you will have to fill in minor additions). Plot all the Lagrange basis for $p = 1, 2, 3, 4$ similarly to Figure 1 for the interval $[-1, 1]$ where the nodes are equally spaced as well as for unevenly (you can choose how unevenly) spaced nodes.

[1] Note that often we omit the superscript p on ℓ; namely we write $\ell_i \equiv \ell_i^p$

Lagrange Polynomials Starter Code

```
# @param[in] nodes the coordinates of the nodes
# @param[in] index the index of the basis function
# @param[in] order the order of the polynomial basis
# @param[in] x the coordinate where to evaluate the basis
# @return the value of the index-th basis function at point x
def lagrange_basis(nodes, index, order, x):

  # Check that we have the right number of nodes
  assert len(nodes) == (order + 1)

  # Create the initial value of the function
  ell = 1.

  # Loop over all nodes
  for i in range(0,order+1):

    # If this node is the same as the
    # support node of the basis function, skip it
    if i == index:
      continue

    # Otherwise perform the multiplication
    ell *= (x - ???)/(??? - ???) # <- fill here!!!

  return ell
```

2. The derivatives of the Lagrange polynomial basis are given by

$$\frac{d\ell_i^p(x)}{dx} = \ell_i^p(x) \sum_{\substack{0 \le j \le p \\ i \neq j}} \frac{1}{x - x_j}. \tag{2}$$

Use the following code as a starter code (you will have to fill in minor additions). Plot the derivatives of all the Lagrange basis for $p = 1, 2, 3, 4$ similarly to Figure 1.

Lagrange Polynomials Starter Code

```
# @param[in] nodes the coordinates of the nodes
# @param[in] index the index of the basis function
# @param[in] order the order of the polynomial basis
# @param[in] x the coordinate where to evaluate the basis
# @return the value of the derivative of the
#         index-th basis function at point x
def d_lagrange_basis(nodes, index, order, x):

  # Check that we have the right number of nodes
  assert len(nodes) == (order + 1)

  # Create the initial value of the function
  d_ell = 0

  # Loop over all nodes
```

```
  for i in range(0,order+1):

    # If this node is the same as the
    # support node of the basis function, skip it
    if i == index:
      continue

    # Otherwise perform the multiplication
    d_ell += 1./(x - ???) # <- fill here!!!

  # Multiply by the corresponding Lagrange basis
  d_ell *= lagrange_basis(nodes, index, order, x)

  return d_ell
```

3. Suppose now that you are given a function

$$f(x) = \frac{1}{2}\cos(2\pi x) + \exp(x) + x^3$$

and you want to approximate this function f with f^p where

$$f^p(x) = \sum_{i=0}^{p} f(x_i)\,\ell_i^p(x). \tag{3}$$

Save the two functions from part 1. and 2. in a file called `lagrange_polynomials.py`. Using the starter code below implement the function f^p and plot for $p = 1, \ldots, 10$ the function $f(x)$ and the function $f^p(x)$ over the interval $[-1, 1]$. Use as starter code the file `lagrange_interpolation.py`.

Lagrange Polynomials Starter Code

```
# @param[in] f the function to be interpolated
# @param[in] nodes the nodes of the interpolation
# @param[in] poly_order the polynomial order of the interp
# @param[in] x the point where to evaluate the interpolation
# @return the value of the interpolation
def lagrange_interpolation( f, nodes, poly_order, x ):

  # Initialize the value of the interpolation
  fp_val = 0

  # Sum up over all the basis
  for i in range(0,poly_order+1):

    fp_val += ??? # <-- fill here

  return fp_val
```

4. We would like to measure the error between $f^p(x)$ and $f(x)$ for a few values of p. To this extent we need to define a way to measure the error. To do so we introduce the L^2-*norm* of a function $g(x)$ over an interval $[a, b]$ defined as

$$\|g\|_{L^2([a,b])} = \left[\int_a^b g^2 dx\right]^{1/2}.$$

For our one dimensional example, if we define the error as $e(x) = |f^p(x) - f(x)|$, we have that the $L^2([-1,1])$ norm of the error is defined as

$$\|e\|_{L^2([-1,1])} = \left[\int_{-1}^{1} e^2 dx\right]^{1/2}.$$

The above integral can be very hard to perform as $f^p(x)$ is a rather complicated expression and $f(x)$ could also be very challenging to integrate analytically.

To this extent we are going to adopt *Gaussian quadrature* (we will further talk about this in class). Gaussian quadrature is a way to approximate an integral by a weighted sum of function values at special points. The weights and the special sampling points are called quadrature weights w_i and quadrature points $\hat{x}_i$; together $\{(w_i, \hat{x}_i)\}_{i=1}^n$ form a quadrature rule. More specifically we will perform the following approximation

$$\int_a^b g(x)dx \approx \sum_{i=1}^{n} f(\hat{x}_i)\, w_i.$$

where n is the number of quadrature points. Of course, both the quadrature weights and quadrature points depend on the domain of integration. Some quadrature rules are exact for certain order of polynomials. For now we will not get hung up on the accuracy of various quadrature rules as we will talk about it in class but I encourage you to read on the subject.

Luckily for us some of the more famous quadratures rules are already implemented in `Python`. We will be using Gauss-Legendre quadrature of order n that integrates exactly polynomials of order $p = 2 * n - 1$. For example, if we want to integrate a polynomial of order p we should choose a quadrature rule of order $n = \lceil (p+1)/2 \rceil$.

In our case we are integrating a function $f(x)$ that is rather complicated (i.e. is not just a simple polynomial) thus we will use a very large quadrature rule to ensure the quadrature error does not pollutes our interpolation error $\|e\|_{L^2([-1,1])}$. To perform the integration you simply have to add the following lines in the right places in the code of part 3. Using the following code and the previous code plot on a log-log plot (in matplotlib `loglog( x, y )`) the error vs the number of basis functions (i.e. $p+1$).

Computing the L^2 norm of the error

```
# Get the gauss quadrature rule
quadrature_order = max(poly_order + 1, 10 )
gauss_points, gauss_weights = \
  np.polynomial.legendre.leggauss( quadrature_order )

# Compute the L2 norm of the error
e = f(gauss_points) - fp(gauss_points)
l2_err = np.sqrt( sum( gauss_weights*( pow( e ,2) ) ) )
```

5. Look up what Weierstrass Approximation Theorem states and explain why the choice of (Lagrange) polynomials is a good choice to approximate functions.

PROBLEM 2: Constructing Element Arrays

Preamble

Consider the simple problem of solving for $u : [-1, 1] \to \mathbb{R}$ such that

$$\frac{d^2u}{dx^2} = f \quad \forall x \in (-1, 1)$$

and

$$u(-1) = g_l, \quad u(1) = g_r.$$

The weak form becomes: find $u \in \mathcal{S}$, with $\mathcal{S}$ being the set of trial functions, such that

$$a(u, v) = F(v) \quad \forall v \in \mathcal{V}$$

where $\mathcal{V}$ is the set of test functions and

$$a(u, v) = \int_{-1}^{1} \frac{du}{dx}\frac{dv}{dx}dx, \quad F(v) = -\int_{-1}^{1} f\, v dx.$$

We then approximate the space of trial and test functions with piecewise polynomials as to arrive at the Galerkin approximation as we have done in class.

In this part of the homework we are going through an implementation of the finite element solution for the above problem. Note that a lot of the infrastructure (the different parts of the code) are reusable. Effectively the code consists of a few parts.

i. A function for the calculation of the *local* element arrays (stiffness and source terms)

ii. A local to global map for the degrees of freedom

iii. A routine to perform the assembly as we did for matrix structural analysis

The above are implemented in the attached code named `finite_element_example.py` in three functions:

i. `element_stiffness`

ii. `local_to_global_map`

iii. `assemble_global`

Effectively the ultimate goal of any finite element code is to assemble the "stiffness matrix" K and a source vector F such that the values of the degrees of freedoms (the u_i in $u^h = \sum_i u_i\phi_i$) can be found by solving $[K]\{U\} = \{F\}$. To do so we perform the following steps:

i. We loop over all elements in the mesh

ii. For each element we compute the local element arrays. To do so we

   (a) Loop over all of the quadrature points inside the element

   (b) At each quadrature point (ξ_g, w_g) we evaluate the integrands and add them to the local element arrays

$$k^e_{ij} += \frac{d\phi_i}{dx}(\xi_g)\frac{d\phi_j}{dx}(\xi_g)\left(\frac{d\hat{x}}{d\xi}(\xi_g)\right)^{-1} w_g \approx \int \frac{d\phi_i}{dx}\frac{d\phi_j}{dx}\left(\frac{d\hat{x}}{d\xi}\right)^{-1} d\xi$$

   and

$$f^e_i += \phi_i(\xi_g) f(\hat{x}(\xi_g))\frac{d\hat{x}}{d\xi}(\xi_g)\, w_g \approx \int \phi_i(\xi) f(\hat{x}(\xi))\frac{d\hat{x}}{d\xi}\, d\xi$$

iii. We assemble k^e and f^e in the global system matrix

Problem

With the above preamble let's move forward to the actual problem.

We would like to manufacture boundary conditions g_l, g_r and source terms f such that the solution of the above boundary problem is exactly

$$u^e(x) = \frac{1}{2}\cos(2\pi x) + \exp(x) + x^3.$$

To do so we simply set

$$g_l = u^e(-1), \quad g_r = u^r(1), \quad f(x) = \frac{d^2u^e}{dx^2}.$$

The above is already implemented for you in the following lines using `simpy`

```
x = sp.symbols('x')
ue = 1./2*sp.cos( 2.*sp.pi*x ) + sp.exp(x) + x**3
f = sp.lambdify(x,sp.diff(ue,x,2))
ue = sp.lambdify(x,ue)
```

1. Run the code provided in `finite_element_example.py` and plot the solution for 2^4 elements.
2. Compute the L^2 norm of the error using the provided function. Namely you will have to write somewhere after the computation of the solution

```
l2_err = compute_l2_norm_error(coordinates_array, connectivity_array, \
      poly_order, u, ue )
```

3. Run the above code for $2^1, 2^4, \ldots, 2^{10}$ elements and for each solution compute the error. Plot the computed errors as a function of the mesh size $h = 2/2^1, \ldots, 2/2^{10}$ on a log-log plot (use `matplotlib loglog` plot). Here you will have to write a `for` loop that wraps the relevant parts of the code.
4. What is the rate of convergence of the solution? Namely, we see that for a mesh size h the L^2 norm of the error is

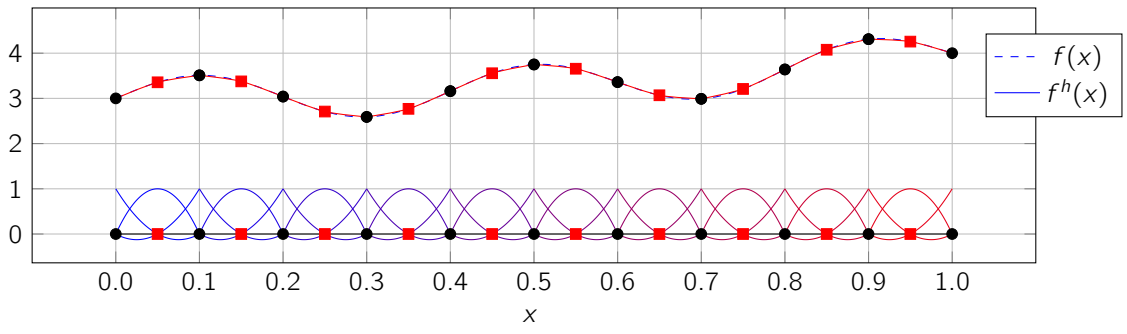
   $$\|e^h\|_{L^2} = \|u^h - u^e\| \approx \alpha h^k$$

   where α is a constant independent of h. Our goal is to determine k.

   To do so choose a fixed h (for example $h = 2/2^7$) then look at

   $$\frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}} \approx \frac{\alpha\, h^k}{\alpha\,(h/2)^k} = 2^k \Rightarrow k = \log_2 \frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}}.$$

5. So far we have used piecewise linear basis functions to approximate the solution. Since in the previous problem we also implemented higher order polynomials it would be nice to use them. To do so simply change the parameter `poly_order`. Effectively the basis functions will become piecewise p order polynomials over each element, as shown for $p = 2$ in the figure below. For $p = 2, 3$ repeat the above steps. Namely, on a single log-log plot, plot the L^2 norm of the error vs the mesh size for $p = 1, 2, 3$ and compute the rates of convergence.



6. If similarly to the mid-term exam we wanted to solve instead

$$\frac{d^2u}{dx^2} + ku = f \quad \forall x \in (0,1)$$

keeping the boundary conditions the same. How would you implement this? Namely

   (a) State which function you would modify

   (b) Write the pseudo-code to illustrate how you would implement it (you don't have to actually implement it).

7. $\star$ Suppose now that you were interested in measuring the error in your derivatives. To do so you can use another measure of error which is the H^1 norm of the error defined as

$$\|e\|_{H^1([-1,1])} = \left[\int_{-1}^{1} e^2 dx + \int_{-1}^{1} \left(\frac{de}{dx}\right)^2 dx\right]^{1/2}.$$

   Implement a function to compute the above, using `compute_l2_norm_error` as a starting point. On a single log-log plot for $p = 1, 2, 3$ show the error vs the mesh size and compute (and report) the convergence rates of the H^1 norm of the error.

PROBLEM 3: Learning FEniCS

This problem is to familiarize yourself with FEniCS, a very useful, high-level, and open-source finite element library.

1. Visit the FEniCS webpage (`https://fenicsproject.org`) and go to the Download tab at the top. If you are using Mac or Linux we suggest you use the Anaconda prebuilt distribution. If you are using Windows we suggest you use Docker distribution. If you have any issues along the installation please contact us for help.

2. We are going to solve the simplest possible problem. The problem reads: find $u : [-1, 1] \to \mathbb{R}$ such that

$$\frac{d^2u}{dx^2} = f \quad \forall x \in (-1, 1)$$

and

$$u(-1) = g, \quad \frac{du}{dx}(1) = t.$$

We would like to manufacture boundary conditions (t and g) and source terms (f) such that the solution of the above boundary problem is exactly

$$u^e(x) = \frac{1}{2}\cos(2\pi x) + \exp(x) + x^3.$$

To do so we simply set

$$g = u^e(-1), \quad t = \frac{du^e}{dx}(1), \quad f(x) = \frac{d^2u^e}{dx^2}.$$

With the above we can then derive the weak form of the problem statement to be: find $u \in \mathcal{S}$, with $\mathcal{S}$ being the set of trial functions, such that

$$a(u, v) = F(v) \quad \forall v \in \mathcal{V}$$

where $\mathcal{V}$ is the set of test functions and

$$a(u, v) = \int_{-1}^{1} \frac{du}{dx}\frac{dv}{dx}dx, \quad F(v) = -\int_{-1}^{1} f\, v dx + t\, v(1).$$

All of the above is implemented for you in the attached code named `fenics_truss.py`. Your job for this part of the problem is to go through the code and make sure it runs and it makes sense. Once it does plot the analytical solution u^e and the finite element approximation u^h.

Here all I am going to do is to break down what we are doing.

First and foremost we use `sympy` to create the analytical solution u^e and obtain the expressions for $g, t, f(x)$. We also cast the variable `solution`, which is a `sympy` function, into a `python` lambda function for later plotting.

```
x = sp.symbols('x[0]')                                                    1
solution = 1./2*sp.cos( 2.*sp.pi*x ) + sp.exp(x) + x**3                   2
ue_code = sp.ccode(solution).replace('M_PI','pi')                         3
t_code = sp.ccode( sp.diff(solution,x,1)).replace('M_PI','pi')            4
f_code = sp.ccode( sp.diff(solution,x,2)).replace('M_PI','pi')            5
solution = sp.lambdify(x,solution)                                        6
```

Next we create the mesh (aka the subdivision) of our domain by first creating a mesh of a domain $[0, 1]$ then scaling the domain by 2 and shifting it to the left by 1.

```
# Start by creating a unit interval mesh
# subdivided in ndiv elements
ndiv = 10
mesh = UnitIntervalMesh(ndiv)

# We are going to shift the above mesh
# such that it dicretizes the interval [-1,1]
mesh.coordinates()[:] *= 2
mesh.coordinates()[:] -= 1
```

We now create the subspace of piecewise linear polynomials as we saw in class (the triangle-looking functions also shown in the second plot of the problem on Lagrange polynomials).

```
# Create the function space of
poly_order = 1
V = FunctionSpace(mesh, 'Lagrange', poly_order)
```

Next we identify the parts of the boundary of our domain (in $1-D$ these are two points, $\Gamma = \{x = -1, x = 1\}$) that are Dirichlet Γ_D and the ones that are Neumann Γ_N.

```
# Define the dirichlet boundary
class dirichlet_boundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and abs( x + 1. ) < DOLFIN_EPS

# Define the neumann boundary
class neumann_boundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and abs( x - 1. ) < DOLFIN_EPS
```

We next mark the boundaries of our domain

```
boundaries = FacetFunction("size_t", mesh)
dirichlet_boundary().mark(boundaries,1)
neumann_boundary().mark(boundaries,2)
ds = ds(subdomain_data=boundaries)
```

and we define the Dirichlet boundary conditions

```
bc = DirichletBC(V, ue, dirichlet_boundary())
```

Now we are ready to define the trial function and the test function, the expression for the source term, and the Neumann boundary value

```
u = TrialFunction(V)
v = TestFunction(V)
```

```
f = Expression(f_code,pi=np.pi, degree=5)
t = Expression(t_code,pi=np.pi, degree=5)
```

We now define the bilinear form and the forcing functional

```
a = dot(u.dx(0), v.dx(0))*dx
```

```
F = -f*v*dx + t*v*ds(2)
```

and we finally solve the proble

```
uh = Function(V)
solve(a == F, uh, bc)
```

3. With the above code construct boundary conditions and source terms such that the analytical solution is of the form

$$u^e(x) = \frac{1}{2}\sin(3\pi x)\, x + \exp(x) + x^2.$$

Then compute the L^2 norm of the error for a mesh with $N = 2^1, 2^2, 2^3, 2^4, \ldots, 2^8$ elements (here you will have to write a `for` loop) and plot the the L^2 norm of the error against the average element size $h = 2/2^1, 2/2^2, \ldots, 2/2^8$ on a log-log plot (use `matplotlib loglog` plot).

4. What is the rate of convergence of the solution? Namely, we see that for a mesh size h the L^2 norm of the error is

$$\|e^h\|_{L^2} = \|u^h - u^e\| \approx \alpha h^k$$

where α is a constant independent of h. Our goal is to determine k.

To do so choose a fixed h (for example $h = 2/2^7$) then look at

$$\frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}} \approx \frac{\alpha\, h^k}{\alpha\,(h/2)^k} = 2^k \Rightarrow k = \log_2 \frac{\|e^h\|_{L^2}}{\|e^{h/2}\|_{L^2}}.$$

5. What is the rate of convergence if instead of linear polynomials we use quadratic or cubic polynomials? Namely, repeat the above steps by changing the `poly_order` variable in

```
poly_order = 1
V = FunctionSpace(mesh, 'Lagrange', poly_order)
```

Note, that when we change the polynomial order we are simply replacing the "triangle" functions with the higher order Lagrange polynomials of the previous problem constructed over the interior of each element.