FUNCTION APPROXIMATION

JOHAN JANSSON

1. TO READ

- Piecewise linear interpolation (117)
- Quadrature (118)

2. GOALS

2.1. **Understanding.**

- Piecewise linear and constant interpolation
- Interpolation error
- Numerical integration/quadrature (as polynomial interpolation, as time-stepping)
- Quadrature error
- Piecewise linear "hat functions" as basis for piecewise linear functions.
- L2 projection

2.2. **Skills.**

- Symbolically show with pen and paper the error estimate for piecewise linear interpolation.
- Construct piecewise linear interpolants in 1D and 2D
- Implement Python functions for quadrature based on piecewise linear interpolation (trapezoid method), compare to time-stepping.

3. SOFTWARE INTERFACES

Follow the interface specifications in the template Python file in the "python" subdirectory. Copy the templates into a new file named "approximation.py" and continue adding your own code. When you're finished or want to check how well you're doing, submit your "approximation.py" (must be named exactly that) to the Web-CAT system at http://webcat.csc.kth.se.

Some of the templates make use of the DOLFIN Python module. To get documentation about a specific class or function, you can use the Python built-in documentation system (from the Python prompt):

Simulation Technology Module.

```
>>> import dolfin
>>> help(dolfin.Mesh)
Help on class Mesh in module dolfin.cpp:

class Mesh(Variable)
 |  A Mesh consists of a set of connected and numbered mesh entities.
...
>>> help(dolfin.Mesh.num_vertices)
Help on method Mesh_num_vertices in module _cpp:

Mesh_num_vertices(...) unbound dolfin.cpp.Mesh method
    Mesh_num_vertices(Mesh self) -> uint

    Return number of vertices.
...
```

Note that in Python a function may take a function as an argument, and also return a function as return value. For example, to define the square of a function, you can do:

```
def f(t):
  return 1.0/t + t

def square(g):
  def gsquare(t):
    return g(t)*g(t)
  return gsquare

fsquare = square(f)
print fsquare(t)
```

Thus f(2.0) gives 2.5 and fsquare(2.0) gives 6.25, as expected.

4. EXAMINATION

The examination of this module consists of the questions below. Each question gives 1 point (a question with two sub-questions gives $0.5 + 0.5 = 1$ point). To pass the module 2.5/4 points are necessary.

5. QUESTIONS

5.1. **Game/interactive simulation.** The purpose of this module is to introduce piecewise linear functions in 1D and 2D. This is a basis for the solution of partial differential equations by the finite element method, which you will do in M6: PDE. Look at some of the sample finite element solutions for PDE on the course home page to get a feel for how piecewise linear function approximation is used to solve advanced problems.

In this module, play around with the functions in `game/game.py` and try to do the following yourself with data of your choice:

- Play around with piecewise constant and linear interpolation, can you get a feel for the interpolation error?
- Construct a piecewise linear function as a sum of hat basis functions with your own choice of parameters.
- Choose a point in a 2D mesh and plot the corresponding hat basis function.

5.2. **Piecewise linear interpolation.**

(1) Derive the interpolation error for piecewise linear interpolation as in ch. 117, and be able to explain the steps.
(2) Construct a piecewise linear interpolant of the function $f(x) = e^{-10x}$ on [0, 1] with 5 nodes (4 sub-intervals) such that the maximum interpolation error on the whole interval is less than 0.15. Note: For linear interpolation the interpolation constant is $C = \frac{1}{8}$. Hint: Use the error estimate derived in the previous subquestion, and compute the second derivative in the left end point of each sub-interval. Implement the function `interpolant_exponential()` to return the points. Plot the function and its interpolant.

5.3. **Quadrature / Integration.**

(1) Implement `integrate()` (very similar to `primitive()` from module 1). Think about the error estimates and convergence rates now in terms of piecewise constant and linear interpolation (see ch. 118).
(2) The analog of trapezoid integration/quadrature in 2D is called "vertex" (corner points of a triangle) quadrature, and is defined by: $\int_K f(x)dx \approx \sum_{j=1}^{3} \frac{1}{3} f(x^j) V_K$, where K is one triangle in the domain, V_K is the area of K and x^j are the vertices of K.

Implement `integrate2D` based on piecewise linear interpolation, use the given vertex quadrature formula on the test cases: $f_1(x, y) = x$ and $f_2(x, y) = x^2 + y^2$. Use the code below for looping over the triangles and edges in the mesh:

```
mesh = dolfin.UnitSquare(4, 4)

integral = 0.0
for c in dolfin.cells(mesh):
  cell_integral = 0.0
  for v in dolfin.vertices(c):
    cell_integral += ...
  integral += ...
```

5.4. L_2**-projection.**

(1) (Hard) Compute the L_2-projection Pf of the same function $f(x) = e^{-10x}$ as for 1D interpolation with a "mesh" with the same distance between all points (use UnitInterval()), see the template for sample code. Implement the computation of the L_2-norm of the error $\|e\|_{L2} = \sqrt{\int_a^b e^2 dx}$, where the error is $e = Pf - f$ as the `l2_norm()` function in the template (Hint: in Python it's possible for a function to return another function, use this to to define a function which is the square of another function, and to define a function for the error). Plot $f(x)$, the interpolant and the projection. Compute the error of the projection and the interpolant. Can you explain the difference in error for the piecewise linear interpolant and the L_2 -projection?