Michael Rizzo
12/10/2014
Continuous Methods

Solving the 2d Poisson equation using 2d Bilinear Finite Elements

Goal

The Goal of my final Project is to solve the 2d Poisson equation $-u_{xx}-u_{yy}=f(x,y)$ using 2d Bilinear Finite Elements.

Implementation

Bilinear Elements

There are a few differences between Bilinear elements for 2d fem and linear elements for 1d fem. One of the differences is you will have 4 local to global node per element in 2d for bilinear elements vs 2 local to global nodes in 1d for linear elements. Another difference is the quadrature point for bilinears will have both x and y values vs just having an x value for the quadrature point.The derivative basis calculation also changes.In 1d we had just the x component;In 2d we now have an x and a y component,to do the derivative basis calculation we now have to introduce the concept of partial derivatives.The basis derivative caculation will now involve Φ_x and Φ_y .

$\Phi_x(node)=\Phi'(x\,array,quad\,point,node)+\Phi(y\,array,quad\,point,node)$

and

$\Phi_y(node)=\Phi(x\,array,quad\,point,node)+\Phi'(y\,array,quad\,point,node)$.

We also have to introduce the concept of the gradient. The gradient is a vector of partial derivatives.This means the derivative basis calculation will be $\nabla\Phi_{test}\cdot\nabla\Phi_{trial}$. This means each entry in the stiffness matrix will be $\Phi_{x\,test}*\Phi_{x\,trial}+\Phi_{y\,test}*\Phi_{y\,trial}$. Also the way the entries in the mass matrix are calculated change. In the 1d fem case, we just iterated over the test and the trial nodes in x, in the 2d case we have to iterate over the test and the trial nodes for x and y.Each entry in the mass matrix will be calculated by $\Phi(x\,array,quad\,point,test\,node)*\Phi(x\,array,quad\,point,trial\,node)$ + $\Phi(y\,array,quad\,point,test\,node)*\Phi(y\,array,quad\,point,trial\,node)$.

.

The implementation of my project was done using bilinear 2d Finite Elements. Most of the geometry routine had to be completely rewritten. The only function that stayed mostly the same was the basis value calculator and the derivative basis value calculator. The x_array and x_point were changed to array and point . This is done so we can calculate the basis function value and the derivative basis function value for x and y and not have to create seperate basis value calculator functions and derivative value basis function for x and y .

The next function that had to be changed was the x array and y array creator. This function has the inputs for the initial x value, final x value, number of nodes in the x array,initial y value,final y value,and the number of nodes in the y array. To calculate the x array and the y array I used the linspace function in the numpy library. This function will return the x array and the y array.

The next function that had to be changed was the function that calculated the number of unknown in the x array and the y array. The way I did this was I inputted the x array,y array,top boundary condition, bottom boundary condition, left boundary condition, and the right boundary condition. I initialized the number of unknowns for the x array using the length of the x array and the number of unknowns for the y array is the length of the y array. The next step was to determine if each of the top,bottom,left,or right boundary conditions. The next step is to subtract 1 from the number of unknowns in the x array if the left or right boundary condition is a Dirichlet boundary condition. The next step is to subtract 1 from the number of unknowns in the y array if the top or bottom boundary condition is a Dirichlet boundary condition.

The next function that had to be changed was the function that calculated the number of elements in the x and y array. The inputs into this function are the x array and the y array. This function will calculate the number of elements of x and y by taking the length of the x or y array and subtracting one.

The next function that had to be changed was the unknown node array status creator. This function will generate the status for each node in the array. The way I did this was I inputted the top boundary condition,bottom boundary condition,left boundary condition,right boundary condition, and the number of nodes in the x and the y array. Then I looped through the x and the y nodes. If the y node is on the left boundary condition or the right boundary condition and the boundary condition is a Dirichlet boundary, we set that x node coordinate value and y node coordinate value in the matrix to -1. Also if the x node is on the top boundary condition or the bottom boundary condition and the boundary condition is a Dirichlet boundary, we set that x node coordinate value and y node coordinate value in the matrix to -1. Otherwise we set the x node coordinate and y node coordinate value in the matrix to the value of the node. The final step was to reshape the matrix into a list.

The next function I had to change was the local to global node array creator.This converts all of the global nodes in my domain to the local nodes for each element. Since I am using bilinear elements , each element has 4 local nodes. The way I did this is I inputted the number of elements in x , the number of elements in y, and the number of nodes per element. Then I looped through the elements in x and the elements in y. The way I set my code up is to have the initial node per element be the node in the bottom left corner of the element. That node would be the first local node position in the element row in the local to global node array. The second node position in the element row of the local to global node array would be the initial node +1.The third

node position in the element row of the local to global node array would be the initial node +number of elements in x+1.The fourth node position in the element row of the local to global node array would be the initial node +number of elements in x+2.After each element I increased the element by 1 and the initial node value was set to element +(element/4).

The one new function I had to create was a function that had x index and y index mapping for all of the nodes. This had to be done so we could still iterate through all of the nodes and stay in the bounds of the x array and the y array and to keep the structure of the assembly routine the same.
The way I did the x_array and y_array point mapping was I iterated through the length of the y array , then I iterated through the length of x array.Inside this double for loop I appended the y point to y point list, and I appended the x point to the x point list.

The things i had to change in the assembly routine were the way the mass stiffness,and right hand side matrices were calculated,and the mass,stiffness,right hand side matrix indexing.In 1d we had just the x component;In 2d we now have an x and a y component,to do the derivative basis calculation we now have to introduce the concept of partial derivatives.The basis derivative caculation will now involve Φ_x and Φ_y .
$\Phi_x(node)=\Phi'(x\,array, quad\,point\,for\,x, node)+\Phi(y\,array, quad\,point\,for\,y, node)$
and $\Phi_y(node)=\Phi(x\,array, quad\,point, node)+\Phi'(y\,array, quad\,point, node)$.
We also have to introduce the concept of the gradient. The gradient is a vector of partial derivatives.This means the derivative basis calculation will be $\nabla\Phi_{test}\cdot\nabla\Phi_{trial}$. This means each entry in the stiffness matrix will be $\Phi_{x\,test}*\Phi_{x\,trial}+\Phi_{y\,test}*\Phi_{y\,trial}$. Also the way the entries in the mass matrix are calculated change. In the 1d fem case, we just iterated over the test and the trial nodes in x, in the 2d case we have to iterate over the test and the trial nodes for x and y.Each entry in the mass matrix will be calculated by $\Phi(x\,array, quad\,point\,for\,x, test\,node)*\Phi(x\,array, quad\,point\,for\,x, trial\,node)$ + $\Phi(y\,array, quad\,point, test\,node)*\Phi(y\,array, quad\,point, trial\,node)$. Also the indexing changed from array[test node-1] to array[(y_point_array[int(test_node)]-1)*number_of_unknowns_in_x+ (x_point_array[int(test_node)]-1)] for the rhs.

The mass and stiffness matrix indexing became array[[(y_point_array[int(test_node)]-1)*number_of_unknowns_in_x+ (x_point_array[int(test_node)]-1),(y_point_array[int(trial_node)]-1)*number_of_unknowns_in_x+(x_point_array[int(trial_node)]-1)] .

The right hand side calculation became my_functions2.rhs_function(quad_point[0],quad_point[1])*my_function s2.linear_basis_calculator(x_array,quad_point[0],x_point_array[int(test_node)]) *my_functions2.linear_basis_calculator(y_array,quad_point[1],y_point_array[int (test_node)])*area_element_array[0] .

The final routine had to change was the error routine .The first thing I had to change was the inputs. The new inputs became number_of_elements_in_x,number_of_elements_in_y,number_of_unknowns_in_x,x

_array,y_array,x_point_array,y_point_array,local_to_global_node_array,solution, unknown_node _array.The next thing I had to change was the way the quadrature points were calculated.I calculated the quadrature points by first creating two lists that hold the x and y coordinates for the quadrature points needed for the error routine.Then I set up a list comprehension structure to iterate over the x coordinate and y coordinate and put each coordinate into a list.Now I have a list of x and y coordinate pairs into a list.The next thing I had to change was the way u approximation was calcuated.i had to change the solution indexing for the u approximation from int[test_node-1] to (y_point_array[int(test_node)]-1)*number_o f_unknowns_in_x+ (x_point_array[int(test_node)]-1). I also had to change the u approximation calculation to u_approximation+=solution[(y_point_array[int(test_node)]-1)*number_o f_unknowns_in_x+(x_point_array[int(test_node)]-1)]*my_geometry.linear_basis_calculator(x_ar ray,quad_point[0],x_point_array[int(test_node)])*my_geometry.linear_basis_cal culator(y_arra y,quad_point[1],y_point_array[int(test_node)]). I also had to change the error calculation to error+=(u_approximation-sin(pi*quad_point[0])*sin(pi*quad_point[1]))**2*((x_arr ay[x_point_array[int(x_b)]]-x_array[x_point_array[int(x_a)]])/2.0)*((y_array[y_point_array[int(y_b)]]-y_array[y_point_array[int(y_a)]])/2.0).

Results

Convergence Rate Table

Number of x,y nodes	Convergence
5	0
10	1.421042030921045
15	-0.843419249985926
20	-0.2840838452358722
25	-0.12217143309212737
30	-0.06164495261295767
35	-0.03489512309513541
40	-0.0214983615463
45	-0.007000155601801498
50	-0.005191254229795831
55	-0.003953402703510129
60	-0.003078610431130078
65	-0.002443313388452188
70	-0.001971045813588266
75	-0.093124033750594
80	-0.0009468577508341345

85	-0.0008080382197562325
90	No clue where it is
95	No clue where it is
100	No clue where it is

Even though I have the wrong convergence rates , as the number of nodes increase the countours converge to a circular shape.Attached is the movie proving this.