Exercise 1. Download the package chebfun from www.chebfun.org. This package works with functions that are represented (to machine precision) as sums of Chebyshev polynomials. It can solve 2-point boundary value problems using spectral methods. Use chebfun to solve the same problem that you solved in HW3+; i.e.,

$$\begin{aligned} -\frac{d}{dx}\left((1+x^2)\frac{du}{dx}\right) &= f(x), 0 \leq x \leq 1, \\ u(0) &= 0, u(1) = 0 \end{aligned}$$

where $f(x) = 2(3x^2 - x + 1)$, so that the exact solution is $u(x) = x(1-x)$. Print out the L2-norm and the ∞-norm of the error.

Solution 1. This is answered in the code appendix.

Exercise 2. Write a code to solve Poisson's equation on the unit square with Dirichlet boundary conditions:

$$u_{xx} + u_{yy} = f(x, y), 0 < x, y < 1$$
$$u(x, 0) = u(x, 1) = u(0, y) = u(1, y) = 1.$$

Take $f(x, y) = x^2 + y^2$, and demonstrate numerically that your code achieves second order accuracy. [Note: If you do not know an analytic solution to a problem, one way to check the code is to solve the problem on a fine grid and pretend that the result is the exact solution, then solve on coarser grids and compare your answers to the fine grid solution. However, you must be sure to compare solution values corresponding to the same points in the domain.]

Solution 2. This is answered in the code appendix.

Exercise 3. Now use the 9-point formula with the correction term described in Sec. 3.5 to solve the same problem as in the previous exercise. Again take $f(x, y) = x^2 + y^2$, and numerically test the order of accuracy of your code by solving on a fine grid, pretending that is the exact solution, and comparing coarser grid approximations to the corresponding values of the fine grid solution. [Note: You probably will not see the 4th-order accuracy described in the text. Can you explain why?]

Solution 3. This is answered in the code appendix. I believe that fourth order accuracy isn't achieved exactly when we are comparing fine grid approximations to coarse grid approximations due to (possible) ill-conditioning of the matrix $\mathbf{A}$. Using the fact that $\Delta f = 4$ is constant, we see that the error correction is just slightly shifting our solutions u_{ij} by a factor depending on $\mathbf{A}$ since our equation becomes

$$\begin{aligned}\mathbf{AU} &= \mathbf{F} + \frac{h^2}{12}\Delta\mathbf{F} \\ \mathbf{U} &= \mathbf{A}^{-1}\left(\mathbf{F} + \frac{h^2}{3}\mathbf{1}\right).\end{aligned}$$

since $\Delta\mathbf{F} = (4, 4, \ldots, 4)$ and $\mathbf{F}$ has elements $\mathbf{F}_n = f(x_n, y_n)$ assuming the grid points are ordered. This correction term $\Delta\mathbf{F}$ could provide computational difficulties due to its small size $O(h^2)$ and ill-conditioning of $\mathbf{A}$ based on underlying solution method implemented for this system, leaving the global error order between 2 and 4.

Exercise 4. We have discussed using finite element methods to solve elliptic PDE's such as

$$\Delta u = f \text{ in } \Omega, u = 0 \text{ on } \partial\Omega$$

with homogeneous Dirichlet boundary conditions. How could you modify the procedure to solve the inhomogeneous Dirichlet problem:

$$\Delta u = f \text{ in } \Omega, u = g \text{ on } \partial\Omega$$

where g is some given function? Derive the equations that you would need to solve to compute, say, a continuous piecewise bilinear approximation for this problem when Ω is the unit square $(0,1) \times (0,1)$.

Solution 4. Adding this method will come down to adding basic functions which we can use to approximate the boundary. Assuming that we write the approximate solution to the inhomogenous problem as $\hat{u}(x,y) = \sum_k c_k \varphi_k(x,y)$, where φ_k are bilinear basis functions, our goal is to choose $c_1 \ldots, c_n$ for which

$$(\Delta \hat{u}, \varphi_l) = \sum_{k=1}^{N} c_k (\Delta \varphi_k, \varphi_l) = (g, \varphi_l).$$

In this case, we can rewrite the operator using Green's theorem so that the operator $\Delta\varphi_k, \varphi_l$ is represented in terms of the first partials of φ_k and φ_l. In short, we have that

$$(\Delta \hat{u}, \varphi_l) = \sum_{k=1}^{N} c_k \left(\int_{\partial\Omega} (\varphi_k)_{\mathbf{n}} \varphi_l d\gamma - \iint_{\Omega} (\varphi_k)_x (\varphi_l)_x + (\varphi_k)_y (\varphi_l)_y dx dy \right) = (g, \varphi_l).$$

Since we are answering the inhomogenous problems, some of the basis functions φ_l must be non-zero on the boundary and we'll have that several of terms have the integral over the boundary to be non-zero. These basis functions will be the same basis functions given in the 2d FEM notes, so the partials are simple to compute directly as we're working with a piecewise bilinear function. I really don't want to have to type out all the partials because that's a lot of equations, but I promise I know how to take partial derivatives. We'll additionally want to add bilinear basis functions along the boundary which are 1 at a specific point on the boundary and linearly decay to 0 to adjacent points on the grid. I'm neglecting to write these our precisely, but this essentially amounts to having a basis function of bilinear form φ_l for both every interior grid point and each boundary point (truncating so the functions so they are only non-zero in Ω). With this choice of basis functions, we can write the desired system of equations as $\mathbf{Ac} = \mathbf{f}$, where

$$\begin{aligned}
\mathbf{A}_{lk} &= \left(\int_{\partial\Omega} (\varphi_k)_{\mathbf{n}} \varphi_l d\gamma - \iint_{\Omega} (\varphi_k)_x (\varphi_l)_x + (\varphi_k)_y (\varphi_l)_y dx dy \right) \\
\mathbf{c} &= (c_1, c_2, \ldots, c_N)^T, \\
\mathbf{f} &= ((f, \varphi_1), (f, \varphi_2), \ldots, (f, \varphi_N))^T.
\end{aligned}$$

HW-4-Code-Figgins

February 19, 2021

[65]:
```
using Plots, LinearAlgebra, SparseArrays
```

0.1 Exercise 1. Spectral Methods.

I'll be using ApproxFun which is very similar to `chebfun` but is in julia.

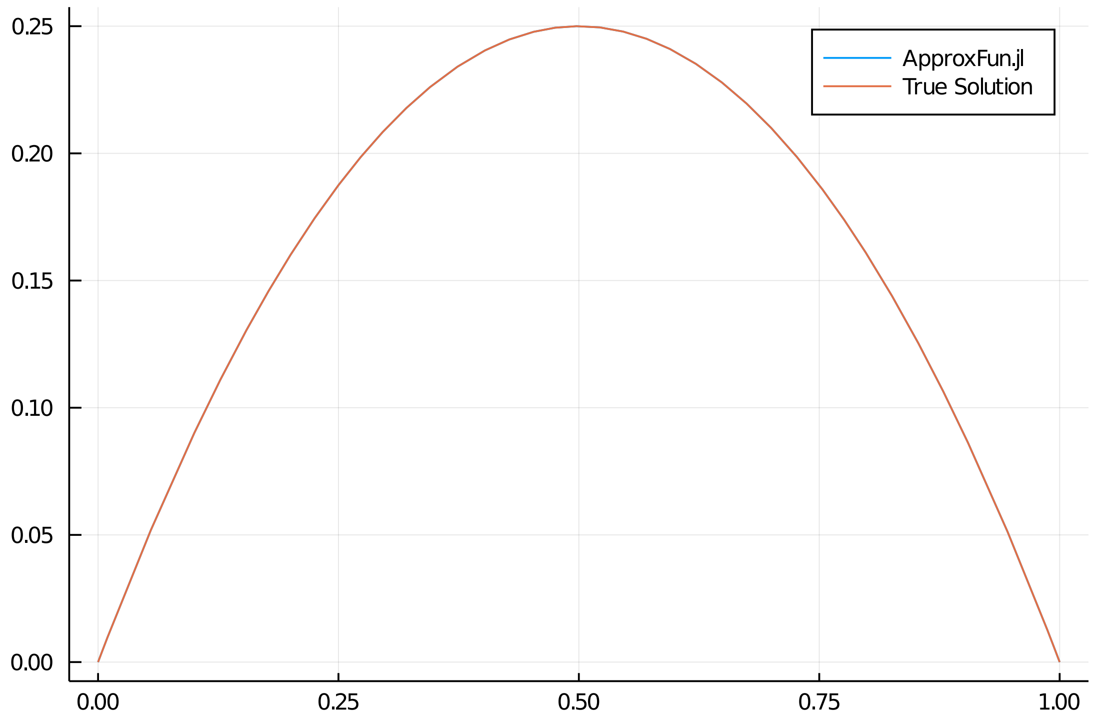
[66]:
```
using  ApproxFun
```

[215]:
```
# Defining Domain and Other ApproxFun objects
x = Fun(identity, Chebyshev(Interval(0.0,1.0)))
d = domain(x)
D = Derivative()
B = Dirichlet()

# Defining LHS
A = [B;
    - D*((1+x^2)*D)] # Defining the operator

# Defining RHS
b_boundary = [0.0 , 0.0] # u(0) = 0, u(1) = 0
b_forcing = [ 6*x^2 - 2*x + 2] # Forcing f(x)

# First equation will satisfy Dirichlet BC == b_boundary
# Second equation will satisfy Operater == Forcing
# We then solve using Backslash
u = A \ [b_boundary, b_forcing ]
```

[215]:
```
Fun(Chebyshev(0.0..1.0),[0.125, -4.337646907243555e-18, -0.125,
9.219382189774801e-19, -1.2254981311325304e-19, -8.390883395346276e-21,
5.098224702976462e-21, -5.76816300010391e-22, 9.248701851716379e-23])
```

[216]:
```
plot(t -> u(t), 0, 1,
    label = "ApproxFun.jl")
plot!(x -> x*(1-x), 0, 1,
    label = "True Solution")
```

[216]:



[217]:
```julia
true_u = Fun(x*(1-x), d)
```

[217]:
```
Fun(Chebyshev(0.0..1.0),[0.125, 0.0, -0.125])
```

[218]:
```julia
# 2-norm is:
norm(u - true_u, 2)
```

[218]:
```
2.8794101375170395e-18
```

[219]:
```julia
# Infinity-norm is:
norm(u - true_u, Inf)
```

[219]:
```
3.89052005832565e-18
```

0.2 Exercise 2. Poisson's equation.

There are a lot strange matrix flips and rotations here so that the heatmaps look right. Sorry if things end up transpoosed!

[220]:
```julia
function makePoissonA(m)
    h = 1/(m+1)
    T = Tridiagonal(ones(m-1), -4*ones(m), ones(m-1))
    S = Tridiagonal(ones(m-1), -zeros(m), ones(m-1))
    A = (kron(I(m), T) + kron(S, I(m))) / h^2
    return A
end
```

[220]: makePoissonA (generic function with 1 method)

[221]:
```
function makePoissonf(m)
    F = zeros(m,m)
    h = 1/(m+1)
    #n = 0
    for i in 1:m
        for j in 1:m
            F[i,j] = (i*h)^2 + (j*h)^2 #x^2 + y^2
        end
    end

    # Add in Dirichlet Boundary Condititions
    for i in 1:m
        F[i, 1] -= 1/h^2
        F[i, end] -= 1/h^2
        F[1, i] -= 1/h^2
        F[end,i] -= 1/h^2
    end

    return reverse(F, dims = 1)
end

function makePoissonf_flat(m)
   return reshape(rotr90(makePoissonf(m)), m^2)
end

function make_U(m)
    A = sparse(makePoissonA(m))
    F = makePoissonf_flat(m)
    return reverse(reshape(A \ F, m, m), dims = 1);
end
```

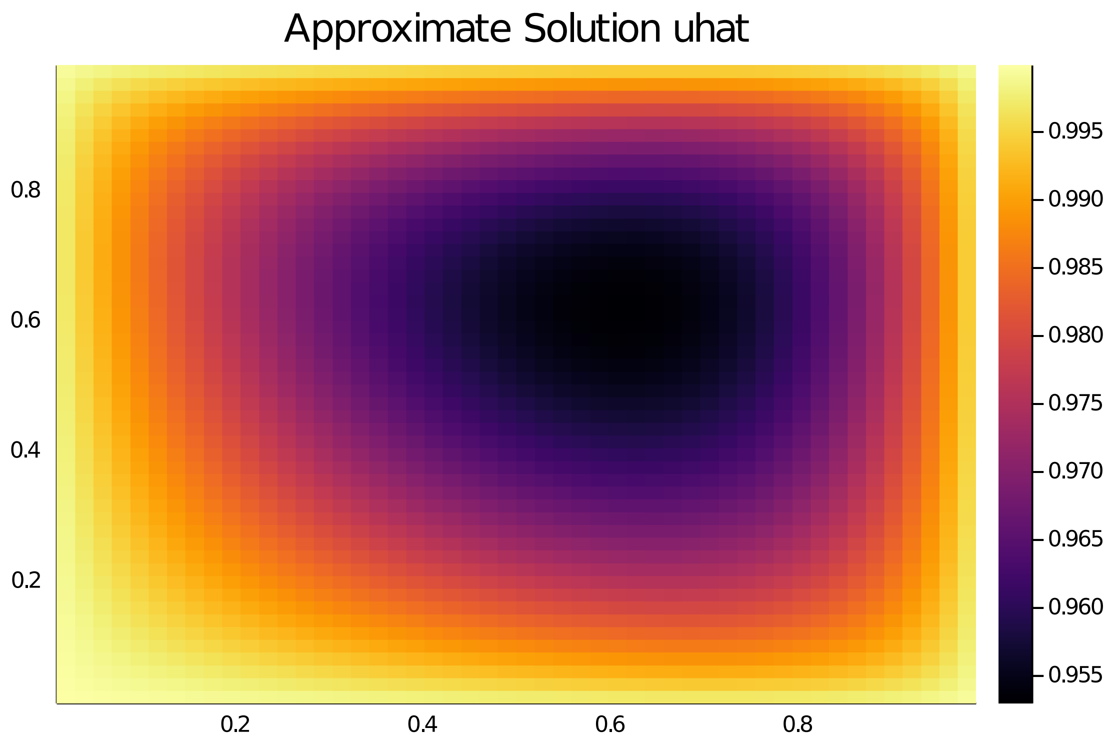
[221]: make_U (generic function with 1 method)

[222]:
```
m = 50
h = 1/(m+1)


A = makePoissonA(m)
F = makePoissonf_flat(m);

U = A \ F
```

```
U_square = reverse(reshape(U, m, m), dims = 1);
```

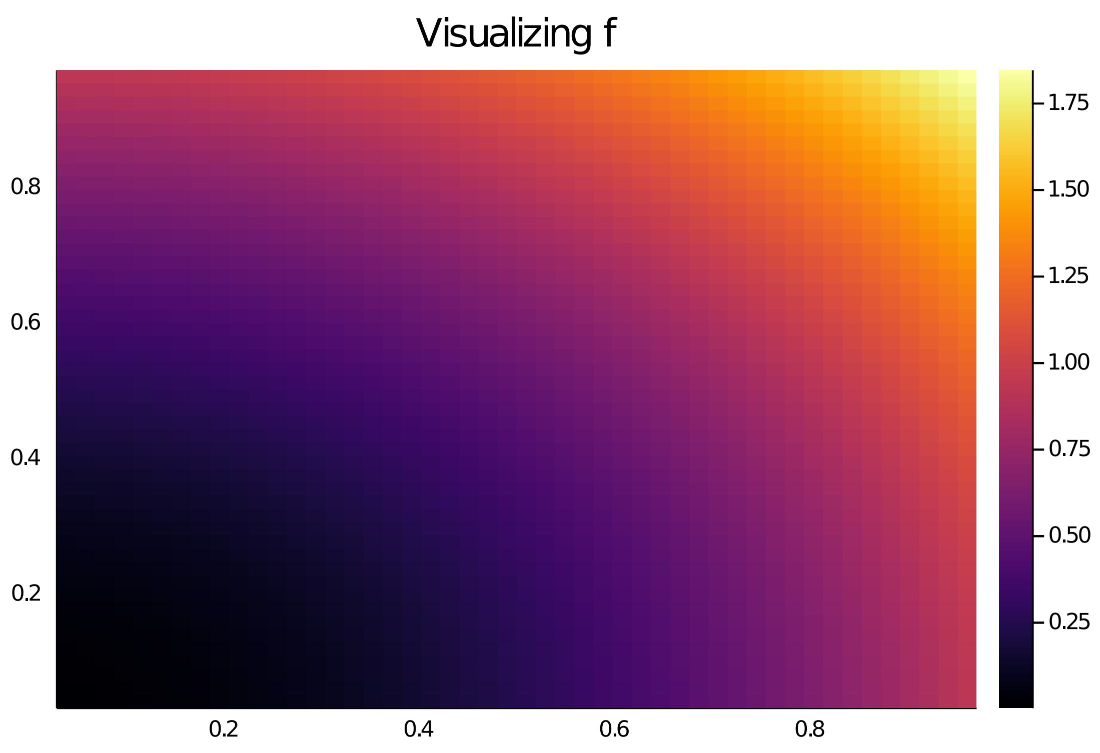
[223]:
```
heatmap([i*h for i in 1:m], [j*h for j in 1:m],
    reverse(U_square, dims = 1),
    title = "Approximate Solution uhat")
```

[223]:



[224]:
```
# Plot excludes Boundary Conditions so the gradient is clear.
heatmap([i*h for i in 2:(m-1)], [j*h for j in 2:(m-1)],
    reverse(makePoissonf(m)[2:(m-1),2:(m-1)], dims = 1),
    title = "Visualizing f")
```

[224]:



[225]:
```
function compare_sol(A_fine, A_coarse)

    m_fine = size(A_fine)[1]
    m_coarse= size(A_coarse)[1]

    mag_diff = Int((m_fine + 1)/ (m_coarse + 1))

    curr_norm = 0
    for i in 1:(m_coarse - 1)
        for j in 1:(m_coarse -1)
            #curr_norm = maximum( [abs(A_fine[i*mag_diff,j*mag_diff] -␣
↪A_coarse[i,j]), curr_norm])
            curr_norm += (A_fine[i*mag_diff,j*mag_diff] - A_coarse[i,j])^2
        end
    end

    return  sqrt(curr_norm / (m_coarse + 1) )
end
```

[225]:
```
compare_sol (generic function with 1 method)
```

[226]:
```
# Defining fine grid
m_fine = 2^7 - 1
```

```
U_fine = make_U(m_fine);
```

[227]:
```
# Comparing Fine and Coarse Grids
norm_vec = []
m_coarse_vec = []
for lgm in 3:6
    push!(m_coarse_vec, 2^lgm - 1)
    push!(norm_vec, compare_sol(U_fine, make_U(2^lgm - 1)))
end
```

[228]:
```
norm_vec
```

[228]:
```
4-element Array{Any,1}:
 0.0014111240628668187
 0.0005366698708931256
 0.00018456150605105314
 5.2483929455128475e-5
```

[229]:
```
# n-fold change between errors
for i in 2:length(norm_vec)
   println(norm_vec[i] / norm_vec[i-1] )
end
```

```
0.38031374066631324
0.34390137412392097
0.2843709426634742
```

It seems that error decreases by a factor of 1/4 as the size of h halves. Running this method for bigger than 2^8 gives my computer trouble. The accuracy appears to be increasing as h increasing and I'd say this method appears to be second order accurate as h increases.

1 Exerise 3. Adding 9-point with correction

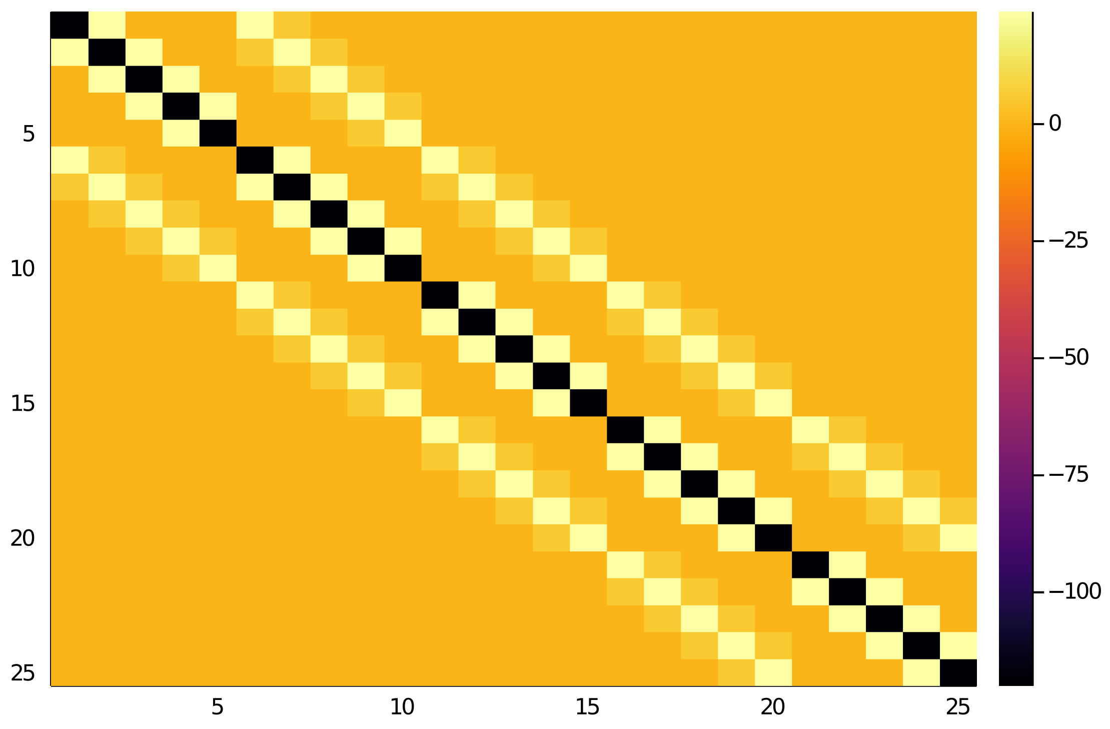
[230]:
```
# Here, we need to change the the 9-point approximation
```

[231]:
```
function makePoissonA9(m)
    h = 1/(m+1)

    T = Tridiagonal(ones(m-1), 10*ones(m), ones(m-1))
    S = Tridiagonal(-(1/2)*ones(m-1), ones(m), -(1/2)*ones(m-1))
    A = -(kron(S, T) + kron(T, S)) / 6h^2
    return A
end
```

[231]:
```
makePoissonA9 (generic function with 1 method)
```

[232]:
```
heatmap(makePoissonA9(5), yflip = true)
```

[232]:



[233]:

```julia
function makePoissonf9(m)
    F = zeros(m,m)
    h = 1/(m+1)

    for i in 1:m
        for j in 1:m
            F[i,j] = (i*h)^2 + (j*h)^2 #x^2 + y^2
            F[i,j] += h^2/3 #Laplacian of x^2 + y^2 is 4
        end
    end

    # Add in Dirichlet Boundary Condititions
    # Left and right grid sides
    # Top and Bottom of Grid
    for i in 1:m
        F[i, 1] -= 1/(h^2)
        F[i, end] -= 1/(h^2)
        F[1, i] -= 1/(h^2)
        F[end,i] -= 1/(h^2)
    end

    # Corners have additional term
    F[1,1] += 1/(6h^2)
    F[1,end] += 1/(6h^2)
    F[end,1] += 1/(6h^2)
    F[end,end] += 1/(6h^2)

    return reverse(F, dims = 1)
end
```

[233]:

```
makePoissonf9 (generic function with 1 method)
```

[234]:

```
function make_U9(m)
    A = sparse(makePoissonA9(m))
    F = reshape(rotr90(makePoissonf9(m)), m^2)
    U_square = reverse(reshape(A \ F, m, m), dims = 1)
    return U_square
end
```

[234]:

```
make_U9 (generic function with 1 method)
```

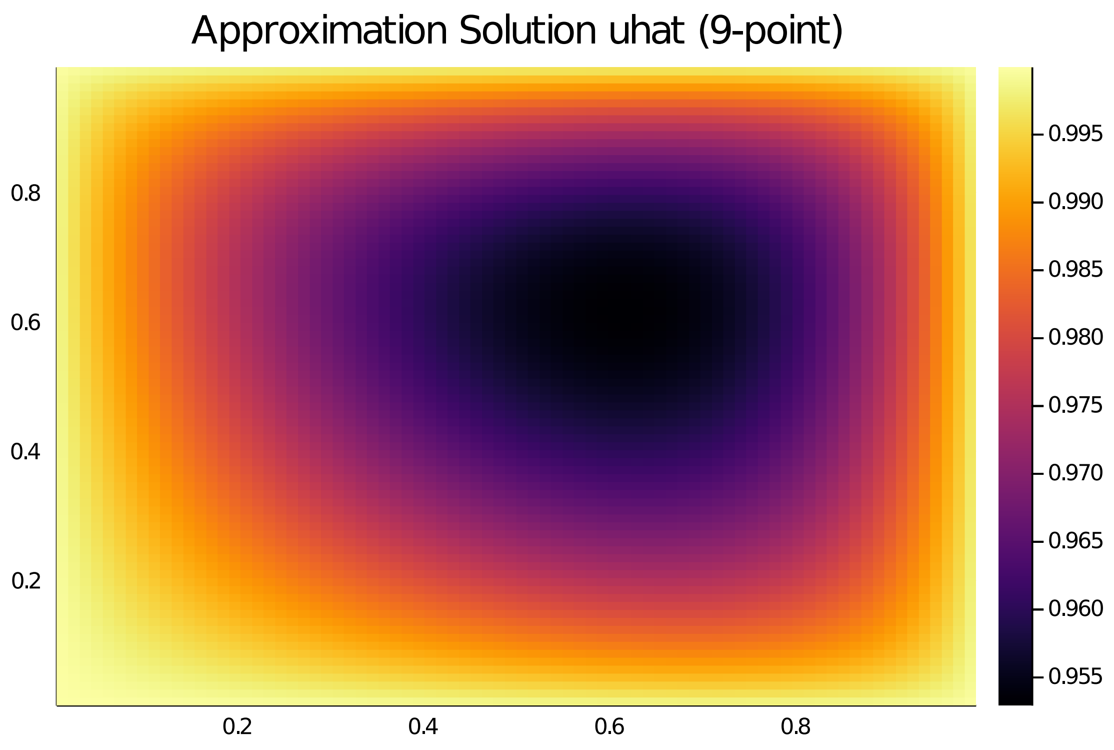
[235]:

```
m = 80
h = 1/(m+1)
U_square = make_U9(m);
```

[236]:

```
heatmap([i*h for i in 1:m], [j*h for j in 1:m],
    reverse(U_square, dims = 1),
    title = "Approximation Solution uhat (9-point)")
```

[236]:



[237]:
```
# Defining fine grid
U_fine = make_U9(2^7 - 1);
```

[238]:
```
# Comparing Fine and Coarse Grids
norm_vec = []
m_coarse_vec = []
for lgm in 3:6
    push!(m_coarse_vec, 2^lgm - 1)
   push!(norm_vec, compare_sol(U_fine, make_U9(2^lgm - 1)))
end
```

[239]:
```
norm_vec
```

[239]:
```
4-element Array{Any,1}:
 1.4044198888024176e-5
 2.359901278882616e-6
 4.0792053720953293e-7
 6.760555279892312e-8
```

[240]:
```
# n-fold change between errors
for i in 2:length(norm_vec)
   println(norm_vec[i] / norm_vec[i-1] )
end
```

```
0.16803388343460163
0.17285491594914437
0.16573216259566942
```

It seems that error decreases by a factor of approximately 1/8 as the size of h halves. Running this method for bigger than 2^8 gives my computer trouble but based on what I'm observing. The accuracy appears to be increasing as h increasing and I'd say this method appears to be a certainly better than second order accurate as h increases but not quite 4th order as expected.