

```

1  include("hw6_helpers.jl");
2
3  function jacobi(h::Real; tol::Real=1e-7, max_iterations::Int=10000)
4      grid = Grid(h);
5      enforce_bcs!(grid);
6      next_grid = Grid(grid);
7      const start_time = time();
8      for iteration = 1:max_iterations
9          for j=2:grid.ny-1, i=2:grid.nx-1
10             next_grid.us[i, j] = (grid.us[i-1, j] + grid.us[i+1, j] +
11                                   grid.us[i, j-1] + grid.us[i, j+1] -
12                                   h*h*force(grid.xs[i, j], grid.ys[i, j])) / 4.0;
13          end
14          if norm(next_grid.us - grid.us, 2) < tol
15              return next_grid, iteration, time() - start_time, true;
16          end
17          copy_grid_values!(grid, next_grid);
18      end
19      return grid, max_iterations, time() - start_time, false;
20 end
21
22 function gauss_seidel(h::Real; tol::Real=1e-7, max_iterations::Int=10000)
23     grid = Grid(h);
24     enforce_bcs!(grid);
25     next_grid = Grid(grid);
26     const start_time = time();
27     for iteration = 1:max_iterations
28         for j=2:grid.ny-1, i=2:grid.nx-1
29             # use updated neighboring values
30             next_grid.us[i, j] = (next_grid.us[i-1, j] + next_grid.us[i+1, j] +
31                                   next_grid.us[i, j-1] + next_grid.us[i, j+1] -
32                                   h*h*force(grid.xs[i, j], grid.ys[i, j])) / 4.0;
33         end
34         if norm(next_grid.us - grid.us, 2) < tol
35             return next_grid, iteration, time() - start_time, true;
36         end
37         copy_grid_values!(grid, next_grid);
38     end
39     return grid, max_iterations, time() - start_time, false;
40 end
41
42 function SOR(h::Real; tol::Real=1e-7, max_iterations::Int=10000)
43     const omega = 2.0 / (1.0 + pi * h);
44     grid = Grid(h);
45     enforce_bcs!(grid);
46     next_grid = Grid(grid);
47     const start_time = time();
48     for iteration = 1:max_iterations
49         for j=2:grid.ny-1, i=2:grid.nx-1
50             # use updated neighboring values
51             next_grid.us[i, j] += omega * (
52                 (next_grid.us[i-1, j] + next_grid.us[i+1, j] +
53                 next_grid.us[i, j-1] + next_grid.us[i, j+1] -
54                 h*h*force(grid.xs[i, j], grid.ys[i, j])) / 4.0 -
55                 next_grid.us[i, j]);
56         end
57         if norm(next_grid.us - grid.us, 2) < tol
58             return next_grid, iteration, time() - start_time, true;
59         end
60         copy_grid_values!(grid, next_grid);
61     end
62     return grid, max_iterations, time() - start_time, false;
63 end
64

```

```

65 for (method_str, approx_method) in [("Jacobi iterative solution", jacobi);
66                                     ("Gauss-Seidel iterative solution",
gauss_seidel);
67                                     ("SOR iterative solution", SOR)]
68     for h in [0.1; 0.05; 0.025]
69         println(method_str, ", h = ", h);
70         grid_approx_soln, iterations, time_elapsed, did_converge = approx_method(h);
71         println("    time elapsed:      ", time_elapsed);
72         println("    iterations:          ", iterations);
73         println("    sec/iteration:        ", time_elapsed / iterations);
74         println("    converged:           ", (did_converge) ? "true" : "false");
75         grid_analytical_soln = Grid(h, analytical_soln);
76         println("    rel. L2 error:      ",
77               norm(grid_analytical_soln.us - grid_approx_soln.us, 2) /
78               norm(grid_analytical_soln.us, 2));
79         println();
80     end
81 end

```