Exercise 1. The key to efficiency in the `chebfun` package, which you used in a previous homework exercise, is the ability to rapidly translate between the values of a function at the Chebyshev points, $\cos(\pi j/n)$, $j = 0, \dots, n$, and the coefficients $a_0, \dots, a_n$, in a Chebyshev expansion of the function's nth-degree polynomial interpolant: $p(x) = \sum_{j=0}^{n} a_j T_j(x)$, where $T_j(x) = \cos(j \arccos(x))$ is the jth degree Chebyshev polynomial. Knowing the coefficients $a_0, \dots, a_n$, one can evaluate p at the Chebyshev points by evaluating the sums

$$p(\cos(k\pi/n)) = \sum_{j=0}^{n} a_j \cos(jk\pi/n), \quad k = 0, \dots, n. \tag{1}$$

These sums are much like the real part of the sums in the FFT,

$$F_k = \sum_{j=0}^{n-1} e^{2\pi ijk/n} f_j, \quad k = 0, \dots, n-1,$$

but the argument of the cosine differs by a factor of 2 from the values that would make them equal. Explain how the FFT or a closely related procedure could be used to evaluate the sums in (1). To go in the other direction, and efficiently determine the coefficients $a_0, \dots, a_n$ from the function values $f(\cos(k\pi/n))$, what method would you use?

Solution 1. Writing the input f_j as a vector $\mathbf{f}$ with elements

$$f_j = e^{-\pi ijk/n} a_j,$$

we see that

$$F_k(\mathbf{f}) = \sum_{j=0}^{n-1} a_j e^{\pi ijk/n}.$$

Taking the real part of this quantity

$$\Re[F_k(\mathbf{f})] = \sum_{j=0}^{n-1} a_j \cos(\pi jk/n) = p(\cos(k\pi/n)) - a_n \cos(k\pi)$$

We can re-arrange this as

$$p(\cos(k\pi)/n) = a_n \cos(k\pi) + F_k(\mathbf{f}),$$

with $\mathbf{f}$ as defined above. With the substitution defined as above, we can use the FFT to compute the values defined in (1). Going backwards, starting with the function values $p(\cos(k\pi)/n)$ would require taking the inverse FFT of

$$p(\cos(k\pi)/n) - a_n \cos(k\pi) = F_k(\mathbf{f}).$$

This would allows us to recover $\mathbf{f}$ and therefore the coefficients a_i as we know how they are scaled to get the f_i.

Exercise 2. On the course web page is a finite difference code (steady2d.m) to solve the boundary value problem:

$$\frac{\partial}{\partial x}\left(a(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(a(x,y)\frac{\partial u}{\partial y}\right) = f(x,y) \quad \text{in } (0,1)\times(0,1)$$

$$u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0,$$

where $a(x,y) = 1 + x^2 + y^2$ and $f(x,y) = 1$. It uses a direct solver for the linear system. Replace this direct solver first by the Jacobi method, then by the Gauss Seidel method, and then by the SOR method. For each method, make a plot of the relative residual norm, $\|b - Au^k\|/\|b\|$ versus iteration number k. (Use a logarithmic scale for the residual; i.e., you may use `semilogy` in Matlab to do the plot.) Try several different values for the parameter ω in SOR, until you find one that seems to work well.
Then try solving the linear system using the conjugate gradient method. You may write your own CG code or use the one in Matlab (called **pcg**). First try CG without a preconditioner (i.e., with preconditioner equal to the identity) and then try CG with the Incomplete Cholesky decomposition as the preconditioner. You may use `ichol` in Matlab to generate the incomplete Cholesky decomposition. Again make a plot of relative residual norm versus iteration number for the CG method.
Experiment with a few different mesh sizes and comment on how the number of iterations required to reach a fixed level of accuracy seems to vary with h for each method.

Solution 2. I implemented all solvers in julia. The code for which can be found attached in the appendix. I also present two plots. One showing the relative residual norms of different methods assuming the same starting vector $\mathbf{u}_0$ and a fixed matrix $\mathbf{A}$ of size $N^2 \times N^2$.

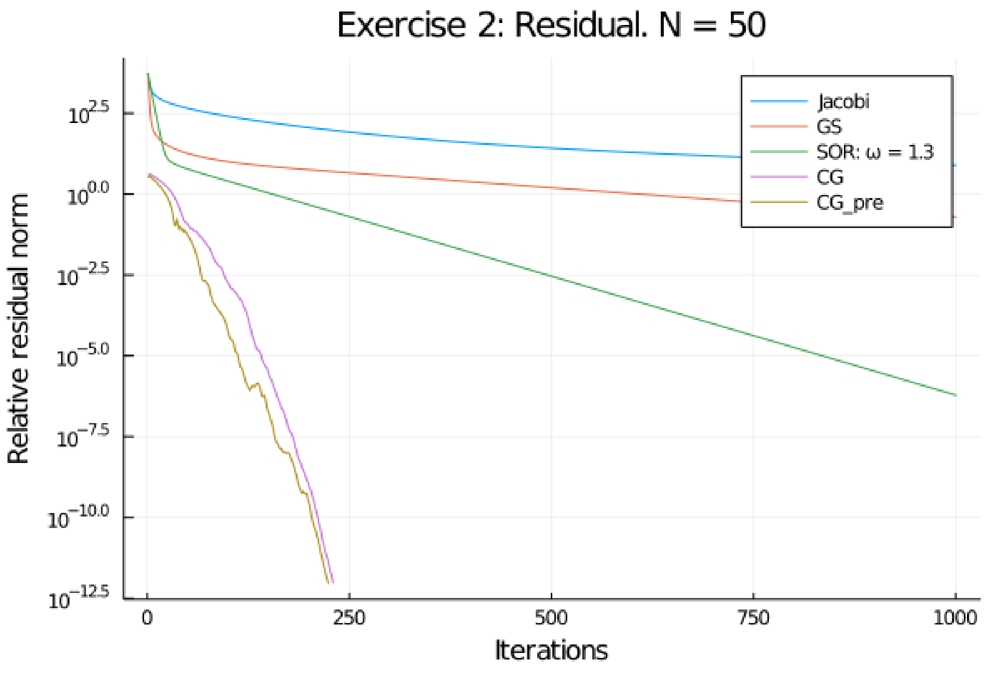

Figure 1: Plotting the relative residual norm for various iterative methods.

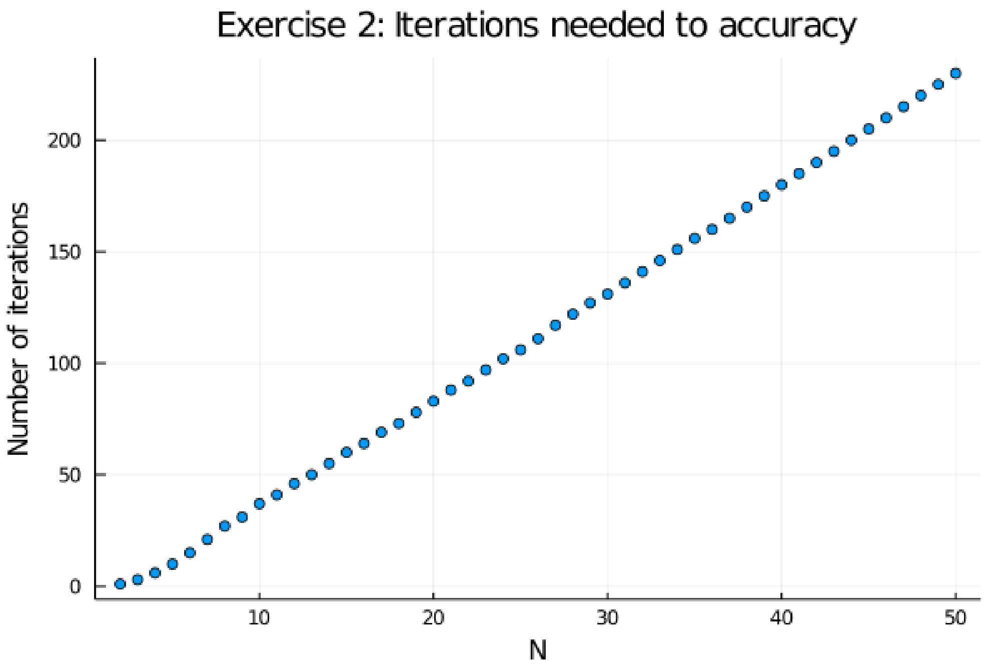

Figure 2: Plotting the relative residual norm for various iterative methods.

Exercise 3. Suppose a symmetric positive definite matrix A has one thousand eigenvalues uniformly distributed between 1 and 10, and one eigenvalue of 10^4. Suppose another symmetric positive definite matrix B has an eigenvalue of 1 and has one thousand eigenvalues uniformly distributed between 10^3 and 10^4. Since each matrix has condition number $\kappa = 10^4$, we have seen that the error at step k of the CG algorithm satisfies

$$\frac{\|e^{(k)}\|_{A,B}}{\|e^{(0)}\|_{A,B}} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k = 2\left(\frac{99}{101}\right)^k.$$

Give another bound on $\|e^{(k)}\|_A/\|e^{(0)}\|_A$ based on a polynomial that is a product of a degree $k-1$ Tchebyshev polynomial on the interval $[1, 10]$ and a linear polynomial that is 1 at the origin and 0 at 10^4. Give another bound on $\|e^{(k)}\|_B/\|e^{(0)}\|_B$ based on a polynomial that is a product of a degree $k-1$ Tchebyshev polynomial on the interval $[10^3, 10^4]$ and a linear polynomial that is 1 at the origin and 0 at 1. For which matrix would you expect CG to converge more rapidly? [If you are not sure, you may try it in Matlab.]

Solution 3. By exhibiting a polynomial p_k which is small near the cluster of eigenvalues and small near the outlier eigenvalue, we can bound $\left|\left|e^{(k)}\right|\right|_{A,B}$. Given a polynomial $p(x)$, we have that

$$\left|\left|P(A)e^{(0)}\right|\right|_A \leq \left|\left|e^0\right|\right|_A \max_{1\leq j\leq m} ||P(\lambda_j)||.$$

Let $[\lambda_a, \lambda_b]$ be the interval where the eigenvalues are clustered and λ_o is the outlier eigenvalue and $\epsilon > 0$. Starting with the example for which $\lambda_1 = 1$ and $\lambda_m = 10$ and $\lambda_o = 10000$, we have that

$$\left|\left|P(A)e^{(0)}\right|\right|_A \leq \left|\left|e^{(0)}\right|\right|_A \max_{\lambda_a<\lambda<\lambda_b} ||P_k(\lambda)||$$

if we have that $P_k(\lambda_0) = 0$. Additionally, we make sure that P_k is near 0 where our eigenvalue clusters. Suppose that we have $k-1 < N$ clusters of width ϵ where N is the number of the eigenvalues. Then, we can refine our estimate as

$$\frac{\left|\left|P_k(A)e^{(0)}\right|\right|_A}{\left|\left|e^{(0)}\right|\right|_A} \leq \max_{\lambda\in(\lambda_i-\epsilon,\lambda_i+\epsilon), 1\leq i\leq k} ||P_k(\lambda)||.$$

We can then write the k-degree polynomial as

$$P_k(\lambda) = \left(\prod_{i=1}^{k-1}\lambda_i\right)^{-1}\left(\prod_{x=1}^{k-1}(\lambda_i - \lambda)\right)\left(\frac{\lambda_o - \lambda}{\lambda_o}\right).$$

In our case, with one cluster, we have that

$$\begin{aligned}
\max_{|\lambda_1-\lambda|<\epsilon} ||P_2|| &= \max_{|\lambda_1-\lambda|<\epsilon}\left|\frac{(\lambda_1-\lambda)(\lambda_o-\lambda)}{\lambda_1\lambda_o}\right| \\
&= \left|\frac{(\lambda_1-(\lambda_1-\epsilon))(\lambda_o-(\lambda_1-\epsilon))}{\lambda_1\lambda_o}\right| \\
&= \frac{\epsilon(\lambda_o-\lambda_1+\epsilon)}{\lambda_1\lambda_0},
\end{aligned}$$

where I've assumed that $\lambda_o > \lambda_1 - \epsilon$. This is on order of ϵ so as the size of the grouping shrinks, we see that the

$$\frac{||P_2(A)e^{(0)}||_A}{||e^{(0)}||_A} \text{ is of order } O(\epsilon).$$

This gives one step bound for A like

$$\frac{||e^{(2)}||_A}{||e^{(0)}||_A} \leq 0.8.$$

Repeating, this process for higher k will produce a polynomial which is approximately flat on the region $(\lambda_i - \epsilon, \lambda_i + \epsilon)$. In the case of matrix A, we can try to tile the region $[1, 10]$ with $k - 1$ evenly spaced intervals to speed up convergence. This will give a better bound on the rate of convergence. In the case of matrix B for which the region is $[10^3, 10^4]$ we have that $\lambda_o \ll \lambda_a$, so there should be a sign flip in the derivation of $\max ||P_2||$. Regardless, we can use the formula derived to get an intuition on which matrix will be quicker to solve. My guess is that matrix A will be easier to solve since ϵ is much smaller so fewer intervals are needed to cover $[\lambda_a, \lambda_b]$ i.e. a lower value of k is needed to cover the interval with the main cluster of the eigenvalues.

HW-5-Code-Figgins

March 1, 2021

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

0.1 Exercise 2

[ ]:
```
#
#  Solves the steady-state heat equation in a square with conductivity
#  c(x,y) = 1 + x^2 + y^2:
#
#     -d/dx( (1+x^2+y^2) du/dx ) - d/dy( (1+x^2+y^2) du/dy ) = f(x),
#                                                        0 < x,y < 1
#     u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0
#
#  Uses a centered finite difference method.

#  Set up grid.
function make_A(n)
    h = 1/n;
    N = (n-1)^2;

    #  Form block tridiagonal finite difference matrix A and right-hand side
    #  vector b.

    A= sparse(zeros(N,N));
    b = ones(N);          # Use right-hand side vector of all 1's.

    #  Loop over grid points in y direction.
    for j=1:n-1,
        yj = j*h;
        yjph = yj+h/2;
        yjmh = yj-h/2;

    #  Loop over grid points in x direction.
      for i=1:n-1,
        xi = i*h;
        xiph = xi+h/2;  ximh = xi-h/2;
        aiphj = 1 + xiph^2 + yj^2;
        aimhj = 1 + ximh^2 + yj^2;
        aijph = 1 + xi^2 + yjph^2;
        aijmh = 1 + xi^2 + yjmh^2;
        k = (j-1)*(n-1) + i;

        A[k,k] = aiphj+aimhj+aijph+aijmh;
        if i > 1
            A[k,k-1] = -aimhj
        end
        if i < n-1
            A[k,k+1] = -aiphj
        end
        if j > 1
            A[k,k-(n-1)] = -aijmh
        end;
        if j < n-1
            A[k,k+(n-1)] = -aijph
        end
      end
    end
    A = (1/h^2)*A;    # Remember to multiply A by (1/h^2).

    return A, b
end
```

[ ]:
```
## Jacobi
function jacobi_iter(A, b, u0, max_iter)
    # Defining M and N
    M = diag(A)
    N = Diagonal(M) - A

    # Initializing storage for u
    u_iter = Vector{Vector{Float64}}(undef, max_iter+1)
    u_iter[1] = u0

    for iter in 2:(max_iter+1)
        u_iter[iter] = (1 ./ M) .* (N*u_iter[iter-1] + b)
    end
   return u_iter
end
```

[ ]:
```
## Gauss Seidel
function GS_iter(A, b, u0, max_iter)
    # Defining M and N
    M = LowerTriangular(A)
    N = UpperTriangular(-A)
    N[diagind(N)] .= 0.0
    # Initializing storage for u
    u_iter = Vector{Vector{Float64}}(undef, max_iter+1)
    u_iter[1] = u0

    c = M \ b

    for iter in 2:(max_iter+1)
        u_iter[iter] = M \ (N*u_iter[iter-1]) + c
    end
   return u_iter
end
```

[ ]:

```
## Successive overrelaxation
function SOR_iter(A, b, u0,  , max_iter)
    # Defining M and N
    L = LowerTriangular(-A)
    L[diagind(L)] .= 0.0
    U = UpperTriangular(-A)
    U[diagind(U)] .= 0.0
    D = Diagonal(A)

    M = (1/ )*D - L
    N = ((1 -  )/ )*D + U

    # Initializing storage for u
    u_iter = Vector{Vector{Float64}}(undef, max_iter+1)
    u_iter[1] = u0

    c = M \ b

    for iter in 2:(max_iter+1)
        u_iter[iter] = M \ (N*u_iter[iter-1]) + c
    end
   return u_iter
end
```

[ ]:

```
function make_residuals(A, b, u_iter)
    residuals = Vector{Float64}(undef, length(u_iter))
    for i in 1:length(u_iter)
        residuals[i] = norm(b - A*u_iter[i]) / norm(b)
    end
    return residuals
end
```

[ ]:

```
# Constant for methods below
max_iter = 400
  = 1.6
```

```
n = 25
u0 = rand((n-1)^2)
A, b = make_A(n);
```

[ ]:
```
u_JAC = jacobi_iter(A, b, u0, max_iter);
u_GS = GS_iter(A, b, u0, max_iter);
u_SOR = SOR_iter(A, b, u0, 1.6, max_iter);

residuals_JAC = make_residuals(A, b, u_JAC);
residuals_GS = make_residuals(A, b, u_GS);
residuals_SOR = make_residuals(A, b, u_SOR);
```

[ ]:
```
p = plot(yscale = :log10,
            xlabel = "Iterations",
            ylabel = "Relative residual norm",
            title = "Exercise 2: Residual")
p = plot!(residuals_JAC,
            label = "Jacobi")
p = plot!(residuals_GS,
            label = "GS")
p = plot!(residuals_SOR,
            label = "SOR:   = $ ")
```

[ ]:
```
using IterativeSolvers
```

[ ]:
```
p = plot(yscale = :log10,
            xlabel = "Iterations",
            ylabel = "Relative residual norm",
            title = "Exercise 2: CG")
p = plot!(residual_CG,
            label = "CG");
```

[ ]:
```
function ichol(B)
    A = copy(Matrix(B))
    N = size(A)[1]
    L = zeros(N, N)
    for k in 1:N
        L[k,k] = sqrt(A[k,k])
        for i in (k+1):N
            if A[i,k] != 0
               A[i,k] = A[i,k] / A[k,k]
            end
        end
        for j in (k+1):N
            for i in j:n
                if A[i,j] != 0
                   A[i,j] = A[i,j] - A[i, k]*A[j, k]
                end
            end
        end
    end

    for i in 1:n
       for j in (i+1):n
            A[i,j] = 0
        end
    end
    return LowerTriangular(A)
end
```

[ ]:
```
x, ch = cg(A, b, reltol = 1e-12, log=true)
residuals_CG = ch.data[:resnorm] ./ norm(b);

L = ichol(A)
x, ch = cg(A, b, reltol = 1e-12, log=true, Pl = L)
residual_CG_pre = ch.data[:resnorm] ./ norm(b);
```

[ ]:
```
p = plot(yscale = :log10,
            xlabel = "Iterations",
            ylabel = "Relative residual norm",
            title = "Exercise 2: Residual")
p = plot!(residuals_JAC,
            label = "Jacobi")
p = plot!(residuals_GS,
            label = "GS")
p = plot!(residuals_SOR,
            label = "SOR:   = $ ")
p = plot!(residuals_CG,
            label = "CG")
p = plot!(residual_CG_pre,
            label = "CG_pre");
```

[ ]:
```
function run_methods_fixed_h(N, max_iter = 400,   = 1.6)

    u0 = rand((N-1)^2)
    A, b = make_A(N)

    u_JAC = jacobi_iter(A, b, u0, max_iter);
    u_GS = GS_iter(A, b, u0, max_iter);
    u_SOR = SOR_iter(A, b, u0, 1.6, max_iter);

    residuals_JAC = make_residuals(A, b, u_JAC);
    residuals_GS = make_residuals(A, b, u_GS);
    residuals_SOR = make_residuals(A, b, u_SOR);
    x, ch = cg(A, b, reltol = 1e-12, log=true)
    residuals_CG = ch.data[:resnorm] ./ norm(b);

    L = ichol(A)
    x, ch = cg(A, b, reltol = 1e-12, log=true, Pl = L)
    residual_CG_pre = ch.data[:resnorm] ./ norm(b);

    p = plot(yscale = :log10,
            xlabel = "Iterations",
            ylabel = "Relative residual norm",
            title = "Exercise 2: Residual. N = $N")
    p = plot!(residuals_JAC,
                label = "Jacobi")
    p = plot!(residuals_GS,
                label = "GS")
    p = plot!(residuals_SOR,
                label = "SOR:   = $ ")
    p = plot!(residuals_CG,
                label = "CG")
    p = plot!(residual_CG_pre,
                label = "CG_pre")
    return p
end
```

[ ]:
```
exer_2_1 = run_methods_fixed_h(50, 1000, 1.3)
```

[ ]:
```
savefig(exer_2_1, "../hw/figs/hw-5-exer-2-residual.png")
```

[ ]:
```
function CG_method_ns(n_sequence, reltol = 1e-10)
    num_iters = []
    for n in n_sequence
        A, b = make_A(n)
        x, ch = cg(A, b, reltol = reltol, log=true)
        push!(num_iters, ch.iters)
    end
    return num_iters
end
```

[ ]:
```
n_sequence = 2:50
reltol = 1e-12
num_iters_need = CG_method_ns(n_sequence, reltol);
```

[ ]:
```
exer_2_2 = scatter(n_sequence,
    num_iters_need,
    label = false,
    xlabel = "N",
    ylabel = "Number of iterations",
    title = "Exercise 2: Iterations needed to accuracy")
```

[ ]:
```
savefig(exer_2_2, "../hw/figs/hw-5-exer-2-iterations-needed.png")
```