18.330 Problem set 8 (spring 2020)

Submission deadline: 11:59pm on Tuesday, April 21

Exercise 1 : Eigenvalue solvers for a special matrix

In this problem we will consider a very special symmetric matrix. Recall that the second-order finite difference scheme for a function $f(x)$ is given by

$$f_n' = \frac{f_{n+1} - 2f_n + f_{n-1}}{h^2}$$

where $f_n = f(x_0 + nh)$ and $h = \frac{x_N - x_0}{N}$.

Define the vector **f** such that $\mathbf{f}_n = f_n$. Then the corresponding derivative vector using finite differences can be written as $\mathbf{f}' = D\mathbf{f}$. Some care must be taken about the boundary conditions; for simplicity we will assume that $f_0 = f_N = 0$. Under these assumptions the matrix D is a tridiagonal matrix:

$$\begin{bmatrix} f_1' \\ f_2' \\ f_3' \\ \vdots \\ f_{N-2}' \\ f_{N-1}' \end{bmatrix} = \underbrace{\begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}}_{D_N} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{bmatrix}$$

You can construct the matrix D in Julia using the `diagm` function. For simplicity keep everything dense in this problem, although there clearly a lot of structure that could be exploited.

1. Consider the matrix D. Using the ansatz (i.e. hypothesis) $\mathbf{v}_n = e^{i\pi nk/N}$, where $1 \leq k \leq N-1$, show that **v** is an eigenvector of D_N. What is the corresponding eigenvalue? Remember that the eigenvalues should be real!

2. In class we discussed several ways to find eigenvalues of matrices. The simplest algorithm for finding eigenvalues is the **power method**.

   Suppse that a symmetric matrix A has the eigendecomposition $A = X\Lambda X^{-1}$. Recall that for a symmetric matrix the eigenvectors associated with distinct eigenvalues are orthogonal. Starting with a random intitial vector **x** which can be written as a sum over the eigenvectors,

$$\mathbf{x} = \sum_n c_n \mathbf{v}_n$$

show that

$$A^k \mathbf{x} = \sum_n \lambda_n^k c_n \mathbf{v}_n$$

where we order the eigenvalues by their magnitude, $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_N|$ and hence

$$A^k \mathbf{x} = c_1 \lambda_1^k \left(\mathbf{v}_1 + \mathcal{O}\left(\frac{\lambda_2^k}{\lambda_1^k} \mathbf{v}_2 \right) \right).$$

This shows why the power method converges to the leading eigenvector.

3. The power method gives an *approximation* for the leading eigenvector $\tilde{\mathbf{v}}_1$. Since it is only an approximation it is not necessarily true that $A\tilde{\mathbf{v}}_1 = \lambda_1 \tilde{\mathbf{v}}_1$ exactly.

   Instead we will say that the best approximation to λ_1 is given by the λ that satifies

$$\lambda_1 = \min_\alpha ||A\tilde{\mathbf{v}}_1 - \alpha \tilde{\mathbf{v}}_1||^2 = \sum_i \left(\sum_k A_{ik} (\tilde{\mathbf{v}}_1)_k - \alpha (\tilde{\mathbf{v}}_1)_i \right)^2 .$$

   By differentiating this expression with respect to α, show that

$$\lambda_1 \approx \frac{\tilde{\mathbf{v}}_1 \cdot (A\tilde{\mathbf{v}}_1)}{\tilde{\mathbf{v}}_1 \cdot \tilde{\mathbf{v}}_1} = \frac{\tilde{\mathbf{v}}_1^\top A \tilde{\mathbf{v}}_1}{\tilde{\mathbf{v}}_1^\top \tilde{\mathbf{v}}_1}.$$

   This is called the **Rayleigh quotient**.

4. Implement the power method `power_method(A, N, x)` for a symmetric matrix A which iterates for a fixed number of iterations N on the intial vector `x` and returns $\tilde{\mathbf{v}}_1$ and λ_1 approximated by the Rayleigh quotient above.

5. Run this on the matrix D_{10} which is of size (9×9). Use the true largest *magnitude* vector from part 1. Plot the relative error between your result and the true value as a function of N to get a smooth plot; use the same initial vector each time.

   Remember to normalize the vector at each iteration to avoid overflow! This initial random vector should be complex. (Extra credit: show that the relationship is what you would expect analytically!)

6. A more advanced method that we discussed to find all the eigenvalues was the QR algorithm. We define $A^{(0)} = A$ and then

$$\begin{aligned} Q^{(k)}R^{(k)} &= A^{(k)} \\ A^{(k+1)} &= R^{(k)}Q^{(k)} \end{aligned}$$

   This will normally converge to an upper-triangular matrix. How does this work? We call two matrices A and B **similar** if $B = QAQ^\top$ where Q is orthogonal. Show that if A has eigenvalue pairs $(\lambda_i, \mathbf{v}_i)$ then B has eigenpairs $(\lambda_i, Q\mathbf{v}_i)$.

7. Show that in the QR algorithm $A^{(k+1)}$ is similar to $A^{(k)}$ and hence A. Therefore if the algorithm converges to an upper-triangular matrix we can read off the eigenvalues from the diagonal and the eigenvectors will be given by the columns of $Q^{(1)}Q^{(2)}Q^{(3)}\cdots Q^{(N)}$.

8. Implement the QR algorithm in a function `qr_algorithm(A, N)` for a matrix A with N iterations. It should return the resulting matrix $A^{(l)}$ and the eigenvector matrix. Run the algorithm on a random symmetric matrix of size 10×10 for 100 iterations. How close to upper-triangular is the resulting matrix?

9. Run the QR algorithm on the matrix D_{10} of size (9×9) for a suitable number of iterations to get convergence. Compare the results to the theoretical one. Can you find all the eigenvalues and eigenvectors?

10. Using the fact that the eigendecomposition of a symmetric matrix gives orthogonal matrices (which are easy to invert) propose a method to solve the linear system

$$D\mathbf{x} = \mathbf{b}$$

11. Solve the system for `b = 0.01^2*sin.(2π*(0.01:0.01:0.99))` and D as a 99×99 matrix. Plot the resulting vector **x**. Plot on the same axis `sin.(2π*(0.01:0.01:0.99))/4π^2`. Is it similar to **x**? We have just solved the boundary value problem $f'' = \sin(2\pi x)$ with $f(0) = f(2\pi) = 0$. In the next problem set we will see how to do this even quicker using Fourier analysis.

Exercise 2: Low-rank approximation

In this problem we will use the SVD to produce low-rank approximations for matrices. We will then use this to compress images and write fast multiplication algorithms.

In the last problem set we saw that we could exploit structural zeros to speed up algorithms. A matrix is of **rank** r if we can write it in the form

$$A = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

where the $\mathbf{u}_i$ and $\mathbf{v}_i$ are vectors of length N, So $\mathbf{u}_i \mathbf{v}_i^\top$ is a *matrix* of size $(N \times N)$, of rank 1.

The SVD of a square $N \times N$ matrix A exactly writes A in this form:

$$A = U \Sigma V^\top = \begin{bmatrix} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_N \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_N \end{bmatrix} \begin{bmatrix} -- & \mathbf{v}_1^\top & -- \\ -- & \mathbf{v}_2^\top & -- \\ & \vdots & \\ -- & \mathbf{v}_N^\top & -- \end{bmatrix} = \sum_{i=1}^{N} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

Truncating the summation after the largest r singular values results in a rank-r approximation of the matrix A. In fact the Eckhart–Mirsky–Young theorem shows this is the *best* rank r approximation to A.

Define the rank r approximation of A

$$A_r := \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

where $\mathbf{u}_r$ and $\mathbf{v}_r$ are the singular vectors of A and σ_r are the singular values of the matrix.

1. Write a function that takes in a matrix `A` and uses the SVD to construct its rank-r approximation, `lowrank_approx(A, r)`. You may use the `svd` function from the `LinearAlgebra` standard library package as `U, Σ, V = svd(A)`.
2. Make a square matrix `M` of size (20×20) that is all zero, except for an axis-aligned cross centred in the centre. Each arm should be of total width 2 and extend a distance 5 out from the centre in each direction parallel to the axes. The cross should consist of 1s.
3. Remembering that the rank is the column rank, i.e. the number of linearly-independent columns, how much should the rank be? Use the SVD of `M` to confirm this. What does the rank-1 approximation of `M` look like?
4. Now add small random gaussian noise using the `randn` function, of intensity 0.1. Plot the new matrix (using `heatmap`). How should this affect the (column) rank of the matrix?

5. Plot the singular values σ_n as a function of n. What do they tell us? What is a suitable rank-n approximation to take? What does it look like?

6. Now let's apply these ideas to a real image. Use the `Images.jl` package to load a test image using the following code. (Remember that you may need to install the relevant packages):

   ```
   using Images, TestImages

   img = testimage("mandrill")
   imgmat = Array(channelview(Float64.(Gray.(img))))
   heatmap(imgmat, c=ColorGradient(:grays), showaxis=false, clims=(0,1), yflip=true)
   ```

   Here `imgmat` is a standard Julia matrix that contains greyscale pixel values.

7. Plot the rank-50 approximation to `imgmat` (using `heatmap`). How does it compare to the original?

8. Plot the singular values as a function of n. You should see an "elbow" shape. Approximately where does it occur?

9. Create an interactive visualization that shows the low-rank approximation for different values of r. What do you observe? After which r are you happy with the quality of the image. Is it related to where the elbow is?

Exercise 3: Dynamic mode decomposition

In this problem we will use the SVD to predict the future!

Suppose that we have some data that satisfies an ODE,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$$

We can solve the ODE and take time snapshots of the result which we can stack into a matrix as follows:

$$X^{n,m} := \begin{bmatrix} | & | & & | \\ \mathbf{x}(t_n) & \mathbf{x}(t_{n+1}) & \cdots & \mathbf{x}(t_m) \\ | & | & & | \end{bmatrix}$$

We will assume that the dynamics is linear. This means that we expect that two consecutive snapshots should satisfy

$$\mathbf{x}_{n+1} = A\mathbf{x}_n$$

for some matrix A.

If we have N snapshots then we can write this as a matrix equation,

$$X^{2,N} = AX^{1,(N-1)}$$

1. Suppose that we have an eigen-decomposition for A, i.e. $A = W\Lambda W^{-1}$, and that we can write the initial time snapshot as a sum over the eigenbasis of A,

   $$\mathbf{x}_0 = \sum_j c_j \mathbf{w}_j$$

   where $\mathbf{w}_n$ is the nth eigenvector of A. Show that future time snapshots can be written as

   $$\mathbf{x}_n = \sum_j c_j \lambda_j^n \mathbf{w}_j$$

   where λ_n is the nth eigenvalue of A.

2. We are now going to try and find the eigendecomposition of A *without* ever constructing it! Start by calculating the SVD of $X^{1,(N-1)} = U\Sigma V^\top$. Find an expression for $U^\top AU$ in terms of $X^{2,N}, \Sigma, V, U$. We can then calculate the eigenspectrum of A by taking the eigenvalue decomposition of $U^\top AU$ since they are similar and using use the result in [1.5].

3. Write a function that calculates the eigenspectrum of A given $X^{1,N}$. Instead of using the full SVD, use a truncated SVD keeping only the first r singular values and singular vectors; i.e. use the matrices `Vr = V[:, 1:r]`, `Ur = U[:, 1:r]` and `Σr = Σ[1:r, 1:r]` in the expression above for V, U, Σ. Your function should be of the form `aspectrum(X, r)`. The reason for truncating is in case A is not full rank, in which case some terms in Σ might be nearly 0 and dividing by them would be a bad idea.

4. Test your function on a random 10×10 matrix A generating some data for X of size (10×11) from it starting from a random x_0. Compare the results of `eigen(A)` with `aspectrum(X, 10)`. Remember that eigenvectors are the same upto a multiplicative constant.

5. We are now going to apply this to some dynamical data. Use the ODE integrator code below to generate some data for N coupled oscillators with $x_1(0) = 0.5$ and all the others $x_i(0) = 0$ for N = 10 from $t = 0$ to $t = 10$. The coupled system can be written as

   $$\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \\ \ddot{x}_3 \\ \vdots \\ \ddot{x}_N \end{bmatrix} = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & \ddots & \ddots & \ddots \\ & & & 1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}$$

or as a system of first order equations,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{y}_1 \\ \dot{x}_2 \\ \dot{y}_2 \\ \dot{x}_3 \\ \dot{y}_3 \\ \vdots \\ \dot{x}_N \\ \dot{y}_N \end{bmatrix} = \begin{bmatrix} 0 & 1 & & & & & & & \\ -2 & 0 & 1 & & & & & & \\ 0 & 0 & 0 & 1 & & & & & \\ 1 & 0 & -2 & 0 & 1 & & & & \\ & 0 & 0 & 0 & 0 & 1 & & & \\ & & 1 & 0 & -2 & 0 & 1 & & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & 0 & 0 & 0 & 0 & 1 \\ & & & & & 1 & 0 & -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ \vdots \\ x_N \\ y_N \end{bmatrix}$$

The output is a data matrix with rows $x_1, \dot{x}_1, x_2, \dot{x}_2, \cdots$

Generate a plot of $x_1(t)$ to check that everything went according to plan.

6. Split the data into two parts X_1 and X_2, the first half from $t = 0$ to $t = 5$ and the second half $t = 5$ to $t = 10$. Calculate the spefctrum of A with $r = 10$ using X_1.

7. Use the first column of X_2 as the initial condition. Use the spectrum you found to predict the future dynamics. [Hint: use the initial condition to find the c_js, which is a matrix solve. Then use the equations in part 1.1 to calculate the prediction.]

8. Plot the prediction for the 10 springs on the same axis as the true solution. What happens?

9. Repeat [3.6–3.7] for $r = 15$ and $r = 20$. What do you observe?

ODE Code:

```
struct RKMethod{T}
    c::Vector{T}
    b::Vector{T}
    a::Matrix{T}
    s::Int

    # Make sure that the matrices and vectors have the correct size
    function RKMethod(c::Vector{T}, b::Vector{T}, a::Matrix{T}, s::Int) where T
        lc = length(c); lb = length(b); sa1, sa2 = size(a)
        if lc == sa1 == sa2 == s-1 && lb == s
            new{T}(c, b, a, s)
        else
            error("Sizes should be (s = $s) :  \n length(c) = s-1 ($lc) \n length(b) = s
        end
    end
end
```

```julia
function (method::RKMethod)(f, x, t, h)
    # Extract the parameters
    a, b, c, s = method.a, method.b, method.c, method.s

    # Vector to hold the k terms
    k = [f(t, x)]

    for i in 1:s-1
        tn = t + c[i]*h
        xn = x + h*sum(a[i, j] * k[j] for j in 1:i)
        push!(k, f(tn, xn))
    end

    return x + h * sum(b.*k)
end

function integrate(method, f, x0, t0, tf, h)
    # Calculate the number of time steps
    N = ceil(Int, (tf - t0) / h)
    hf = tf - (N - 2)*h

    #initiate tracking vectors
    xs = [copy(x0)]
    ts = [t0]

    #step
    for i in 1:N-1
        push!(xs, method(f, xs[i], ts[i], h))
        push!(ts, ts[i] + h)
    end

    # Special last step
    push!(xs, method(f, xs[N-1], ts[N-1], hf))
    push!(ts, ts[N-1] + hf)

    return ts, xs
end

c = [1/2, 1/2, 1]
b = (1/6) .* [1, 2, 2, 1]
a = [1/2 0 0; 0 1/2 0; 0 0 1]
rk4 = RKMethod(c, b, a, 4)

function build_springmat(N)
    springmat = zeros(2N,2N)
    for i = 1:2N
```

```
        (i < 2N) && (springmat[i, i+1] = 1.0)
        if iseven(i)
            springmat[i, i-1] = -2.0
            (i > 3) && (springmat[i, i-3] = 1.0)
        end
    end
    springmat
end

N = 10
const spmat =  build_springmat(N)
spf(t, x) = spmat*x
x0 = zeros(2N); x0[1] = 0.5
ts, xs = integrate(rk4, spf, x0, 0.0, 20.0, 0.005);
X = hcat(xs...)
plot(ts, X[1:2:2N, :]', leg=:outertopright, box=:on)
```

Exercise 4: Fourier integrals

In lectures we saw that we could write periodic functions in the form

$$f(x) = \sum_{n=-N}^{N} \hat{f}_n e^{inx}$$

where $\hat{f}_n = \frac{1}{2\pi}\int_0^{2\pi} f(x)\, e^{-inx}$.

1. Consider the **saw-tooth** function,

$$f(x) = \begin{cases} x & 0 \le x < \pi \\ 2\pi - x & \pi \le x < 2\pi \\ f(\text{mod}(x, 2\pi)) & \text{else} \end{cases}$$

Calculate the Fourier series coefficients analytically.

2. Write a function `fourier_coefficients(f::Vector,  n::Int)` that takes in a vector of samples of f uniformly distributed over $[0, 2\pi]$ and returns an approximation to $\hat{f}_n$ by calculating the integral using the trapezoidal rule.

3. Now calculate $\hat{f}_n$ using your trapezoidal code using 100 points and $n = -3, -2, \dots, 3$. How do they compare to the theoretical results?

4. Fix n to be 1. Plot the relative error between the theoretical result and the result from using `fourier_coefficients` for calculating $\hat{f}_1$ using a number of points between 10 and 1000. What does the convergence look like? Does it agree with what we discussed in class?

5. Now consider the smooth periodic function exp$(\cos(x))$. Repeat [4.3–4.4] for this function. The analytical result is given by $\widehat{f}_n = I_{|n|}(1)$. You can calculate this using the `besseli(abs(n), 1)` in `SpecialFunctions.jl`.

6. Plot the magnitude of $\widehat{f}_n$ as a function of n for the two functions. How do the coefficients decay? Is this what you expected?