Due Friday, October 23, 17:30

IMPORTANT: Guidelines for solution sets:

- Same as before, your folder name should change to `phys414_ps02_surname_name` as you could predict.

Problem 5 (6 points)

A curious relationship occurs between optimization and linear systems in the case of symmetric positive definite matrices. Recall that a matrix is symmetric if $A_{ij} = A_{ji}$, and it is positive definite if $\mathbf{v}^T A\mathbf{v} > 0$ for any $\mathbf{v} \neq \mathbf{0}$.

(a) Show that if A is symmetric positive definite, then the solution to $A\mathbf{x} = b$ is also the value of $\mathbf{x}$ that minimizes the following function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}\cdot\mathbf{x}\ .$$

This means solving the linear system is equivalent to a minimization problem.

(b) The most basic algorithm for finding the minimum of a function is called *gradient (or steepest) descent* (see Section 2.3 of the textbook). It relies on the fact that at any point, a multivariable function increases fastest in the direction that is given by its gradient, which in this case is

$$\nabla f(\mathbf{x}) = A\mathbf{x} - \mathbf{b}\ .$$

Then if we move in the opposite direction, we go to lower values of f, and hopefully eventually reach the lowest value, which also solves $A\mathbf{x} = \mathbf{b}$. So, we start with some initial guess $\mathbf{x}_0$, and then our iteration is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \tau\ \nabla f(\mathbf{x}_n)$$

How should we choose τ? A small constant value such as 0.1 works, but we can also optimize this in each step. Simply, we want to minimize the value of $f(\mathbf{x}_{n+1})$ w.r.t. τ. Show that this happens when

$$\nabla f(\mathbf{x}_{n+1})\cdot\nabla f(\mathbf{x}_n) = 0 \Rightarrow \tau = \frac{(A\mathbf{x}_n - \mathbf{b})\cdot(A\mathbf{x}_n - \mathbf{b})}{(A\mathbf{x}_n - \mathbf{b})\cdot A(A\mathbf{x}_n - \mathbf{b})}\ ,$$

that is the next point in the iteration should be such that the gradient there (the next direction we will move) should be orthogonal to the gradient now (the current direction we are moving).

With some modifications the above procedure becomes the *conjugate gradient method*, which forms the starting point for many state-of-the-art linear solvers for symmetric and non-symmetric matrices alike.

(c) Implement the gradient descent method as

```
function x = linSolveGradientDescentLocal(A,b)
% SOLVEGRADIENTDESCENTLOCAL Solves for x satisfying Ax=b for a symmetric
% matrix A using the conjugate gradient method
```

Make sure you use matrix multiplication, not loops, for expressions like $A\mathbf{x}$ (remember the speed difference in MATLAB). Consider $n \times n$ tridiagonal matrices with entries $A_{ii,} = 1$, $A_{i,i+1} = A_{i-1,i} = -1/2$. Measure how much time is needed on average to solve $A\mathbf{x} = \mathbf{b}$ for some random vector $\mathbf{b}$ as n changes. You can measure the runtime of functions using `timeit`, or `tic/toc`, or you can devise some way of your own (but remember our advise about builtin functions). You need a criterion to stop, a

natural candidate is requiring $|A\mathbf{x} - \mathbf{b}|$ reaching a low enough value. You can check your function by comparison to MATLAB's standard `x=A\b`.

The tridiagonal matrices we considered are of utmost importance in physics. They show up in systems with short range interactions (nearest neighbor coupling) and numerical solutions to differential equations.

(d) Now repeat the previous part, but use Gauss-Seidel iteration. Compare the speeds. This matrix is not diagonally dominant, but the iterations will work nevertheless. Read about Gauss-Seidel a bit if you wonder why.

There are actually more specialized and much faster methods for our symmetric matrix (e.g. *divide-and-conquer*), have a look at them if you like solving linear systems.

(e) Note that most of the elements of A are zero, such a matrix is called a *sparse matrix*. (as opposed to a *dense* one). MATLAB, and many other platforms, have special algorithms adapted to such matrices that can speed computation by orders of magnitude. Let MATLAB know that A is a sparse matrix by building A using `spdiags`, and repeat the previous two parts. Report any change in computation speed.[1]

To sum up, your submission should contain the following MATLAB functions

- `linSolveGradientDescent(A,b)`: Solves $A\mathbf{x} = \mathbf{b}$ using the gradient descent method for a symmetric tridiagonal A.
- `linSolveGaussSeidel(A,b)`: Similar, but uses Gauss-Seidel iteration.
- `ps0205(n_vec)`: For each entry n in `n_vec`, finds the average time to solve $A\mathbf{x} = \mathbf{b}$ for an $n \times n$ tridiagonal A as described, and plots this time vs n.[2] This should be repeated for both methods, and for both cases where A was constructed in the default way and as a sparse matrix. All graphs should be on the same plot with a legend showing which is which. Do not forget to name the axes and have a title.

You can write and submit other auxiliary functions that help you to achieve the above goals, but `ps0205(n_vec)` should generate the specified plot when we run it on our own.

Problem 6

The Lambert W function is the inverse function of $f(x) = xe^x$, that is

$$W(x) = y \iff x = ye^y \ . \tag{1}$$

There are two branches of the function when $x < 0$, we will only deal with the "principal" branch that satisfies $W(x) > -1$.

(a) Calculating W is a basic root finding problem

```
function y = lambertwCustom(x)
%LAMBERTWCUSTOM Calculates the Lambert W function defined as W(xe^x)=y if
%y= xe^x , using the Newton-Raphson method.
funShifted = @(z)(z*exp(z)-x);
funDeriv = @(z)((z+1).*exp(z));
x_init=1;
y = newton(funShifted, funDeriv, x_init);
end
```

[1] You can see the dramatic effect of letting MATLAB know a matrix is sparse in a simple example here. Try it yourself.
[2] As before, this can be done solving the equation for many randomized $\mathbf{b}$ vectors and averaging over.

Here, `newton` is a function analogous to `fsolve` of MATLAB, it solves nonlinear algebraic equations. Unlike `fsolve`'s adaptive nature, `newton` purely uses the Newton-Raphson method to find the root of an expression. Implement it as it can be used in `lambertwCustom` above:

```
function root = newton(fun, dfun, x_init)
% NEWTON Solves fun(x)=0 using the Newton-Raphson method. It requires an
% initial guess and function handles to fun(x) and its derivative, fun
% and dfun respectively.
```

Plot `lambertwCustom` in the interval $(-1/e\ 10)$. W is already a builtin function in MATLAB, `lambertw`. Plot the difference between the two implementations in the same interval. Is your version as accurate as the builtin one (have an appropriate condition to terminate the iteration)? How about a speed comparison? Do you see any simple modifications to the template we provided for `lambertwCustom` that makes it faster?

(b) Newton's method is based on a linear approximation to a function and uses its derivative. We can use other approximation methods and higher derivatives to obtain more complex iteration schemes. One example is Halley's method:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)} = x_n - \frac{f(x_n)}{f'(x_n)}\left[1 - \frac{f(x_n)}{f'(x_n)}\frac{f''(x_n)}{2f'(x_n)}\right]^{-1} \tag{2}$$

Repeat the previous part with this method. Achieve this by adding a second input argument to `lambertwCustom`, and modifying it as required:

```
function y = lambertwCustom(x, method)
% LAMBERTWCUSTOM Calculates the Lambert W function defined as W(xe^x)=x
% using the nonlinear root finding method in method. method= 'n' and
% method='h' use Newton's and Halley's methods. If method is left empty,
% Newton's method is used.
```

Return this final version of the function in your solution set. Use the MATLAB variable `nargin` to implement the feature that `lambertwCustom` can still work with a single variable in which case it defaults to the previous version.

(c) Halley's method has cubic convergence, that is

$$|r - x_{n+1}| < K|r - x_n|^3$$

for some constant K when x_k is close enough to the true value of the root r. This is better than Newton-Raphson which has quadratic convergence. However, Halley's method also requires more function evaluations per iteration. By timing many root finding procedures, investigate which method is faster. For general functions, Newton's and Halley's methods give similar convergence per function evaluations, but sometimes improvements are possible for specific cases where one method can have an edge.

All your investigations should be performed in a function named `ps0206()` which has no arguments, so that we should get your plots and any other piece of information you use in your solution set when we run it.

Problem 7

The motion of a system of coupled harmonic oscillators indexed by x_n are given by

$$\ddot{x}_n = -\sum_m K_{nm} x_m$$

for certain constants K_{nm}. Recall that in a normal mode, all masses move with the same frequency and phase $x_n = c_n e^{i\omega t+\phi}$ (or, to be more precise, the real part of this expression), and one can easily show that finding the normal modes and their frequencies is equivalent to finding the eigenvalues and eigenvectors of the matrix K_{nm}.

Similarly, time evolution in quantum mechanics is governed by the Hamiltonian of the system. Finding the eigenstates and energy levels of a discrete system is also equivalent to an eigenvalue problem.

When the system under investigation has many parts and many interactions, or evolves in time in an unknown manner, we cannot solve the eigenvector problem easily. Even worse, we cannot even know the matrix whose eigenvalues we are supposed to find. However, we may know the statistical properties of the system which can give us enough information to deduce its behavior.

As a foray into such systems, we will investigate the behavior of the eigenvalues of matrices with random elements. Due to their prominence in physics (and to make things easier), *we will restrict ourselves to real symmetric matrices*. Such $N \times N$ random matrices can easily be generated by

```
A = randn(N);
A = A-tril(A)+(triu(A))'; % make sure you understand what is going on here
```

We will use the power iteration we discussed in class to find the eigenvalues of A, but we should note that there are more suitable methods for this specific class of matrices, .

(a) Write the function with the following description

```
function [lambda, v] = eigLargest(A)
% EIGLARGEST Calculates the largest eigenvalue in absolute value (lambda)
% and the corresponding normalized eigenvector (v) of the square matrix A
% using the power iteration method.
```

You can test it by comparing to the builtin `eig`. As a stopping criterion for the iteration, you can use $|\lambda^{(k+1)} - \lambda^{(k)}| < \epsilon$ for some predetermined tolerance ϵ. Here, $\lambda^{(k)} \equiv \frac{\mathbf{v}^{(k)} \cdot A\mathbf{v}^{(k)}}{\mathbf{v}^{(k)} \cdot \mathbf{v}^{(k)}}$ and $\mathbf{v}^{(k)} \equiv A^k \mathbf{v}^{(0)}$ are the estimates for the largest eigenvalue and the corresponding eigenvector at the k^{th} iteration step. [3]

(b) Once we find the largest eigenvalue in absolute value, λ_1, and the corresponding normalized eigenvector v_1, consider the matrix

$$A^{(1)} = A - \lambda_1 v_1 v_1^T \ .$$

[3] This criterion checks the change from one iteration to the next, and when it is small enough *assumes* that we are already close to the true value. This strategy works in this and many other cases, but it is not a foolproof method to see whether we are close to the true solution. In more complex iteration schemes, iteration can slow down in certain regions so much that the criterion can mistakenly decide we are close to the solution well before we reach it.

Ideally, a criterion should measure our proximity to the true solution, but how can we do that when we do not know the true solution (we are trying to find it after all)? Sometimes, analytical results are helpful, for example, in power iteration we have the following criterion

$$|\lambda - \lambda^{(k)}| = \left(\frac{\mathbf{v}^{(k+1)} \cdot \mathbf{v}^{(k+1)})}{\mathbf{v}^{(k)} \cdot \mathbf{v}^{(k)}} - \left(\lambda^{(k)}\right)^2 \right)^{1/2}$$

where λ is the exact largest eigenvalue. If you have such a helpful piece of information, use it instead of the less reliable method we followed.

Show that the largest eigenvalue of $A^{(1)}$ is the second largest eigenvalue of A, and all the eigenvalues are the same (assume that all eigenvalues of A are distinct, i.e. no degeneracy). This update on the matrix to find the next eigenvalue is called *deflation*. There are other possible ways of doing this, and the above choice is called Hotelling's deflation. Note that Hotelling's deflation would not work as smoothly for a non-symmetric matrix (can you say why?).

Write the following function

```
function [lambda_vec, eig_matrix] = eigCustom(A)
% EIGCUSTOM Calculates all the eigenvalues and normalized eigenvectors of
% the symmetric matrix A. Returns the eigenvalues as a column vector with
% ascending elements (lambda_vec), and the respective normalized
% eigenvectors as columns of a square matrix (eig_matrix). It uses
% eigLargest and finds the eigenvalues successively using Hotelling's
% deflation.
```

Test its output by comparison to `eig` for various sizes of matrices, How does its speed compare? Do the eigenvalues always agree, if not, can you identify the reason? The code that investigates these should be in a function called `ps0207b()` that has no arguments.

(c) Consider an $N \times N$ real symmetric matrix whose entries are identically and independently distributed (i.i.d) random Gaussian variables with mean 0 and standard deviation 1 (you do not really need to know all those technical terms, `randn` gives us exactly these kind of numbers). The following statement is a restricted version of Wigner's semicircle law: As $N \to \infty$ the probability that an eigenvalue λ is in the interval $(\sqrt{N}x_1\ \sqrt{N}x_2)$ is

$$P(x_1 < \frac{\lambda}{\sqrt{N}} < x_2) = \frac{1}{2\pi}\int_{x_1}^{x_2} \sqrt{4-x^2}\ dx$$

Show this by plotting histograms of the eigenvalues of random symmetric matrices for $N = 512, 1024, 2048$ by using `eig`, not `eigCustom`, to obtain the eigenvalues. You can use the MATLAB function `histogram(...,'Normalization','pdf')` to check the probability distributions.

Do you realize how wonderful this theorem is? For large enough systems for which our randomness assumptions hold, you know the energy spectrum in terms of density of states! For example, you can do statistical mechanics by calculating the partition function, even though you don't really know the details of the coupling between different parts of the system.

Optional for those who enjoy coding (no points, bonus or otherwise)

Can you get the same distribution in problem 7c using `eigCustom`? What goes wrong? Do you see why we insist on using the builtin functions on a platform whenever possible?[4]

There are methods more suitable than power iteration to calculate the eigenvalues of symmetric matrices, and you can implement some of them and obtain results close to that of `eig`. Feel free to review the literature yourself, one of my suggestions is the QR method. See if you can get similar histograms, and how fast your custom implementation is compared to `eig`.

[4] At least until you are proficient enough to become the person who writes the builtin functions.