CPSC 542g Assignment 1 *Interpolation and Dense Linear Algebra: 15%* **Due October 15, 2008, any time**

Questions 1, 4, 5, and 6 involve programming: email me an archive of your source code, with README files if necessary to explain what you did. Questions 2 and 3, and parts of the others (reporting on your results), are written: you can either give me hardcopy or email me a PDF with your write-up.

(1) Implement RBF interpolation in 2D, in particular looking at the problem of reconstructing an image from a small scattered set of samples. You may find it simplest to use MATLAB (which has functions such as `imread` to get image data); if you want you can use any other language. Try out your code on a few images of your choice and a few choices of distribution of sample points; try to draw conclusions about how effective the RBF approach can be. In particular, how well do RBFs cope with sharp edges in images versus smooth regions?

(2) How well-posed is matrix-vector multiplication? In other words, given a matrix A, how can you bound the relative error in $y = Ax$ relative to the relative error in x? You may assume A is invertible.

(3) A matrix A is diagonally dominant when every diagonal entry is bigger than the rest of the entries in its row put together:

$$a_{ii} \geq \sum_{j \neq i} |a_{ij}|$$

Show that the LU factorization exists for a nonsingular diagonally dominant matrix without need for pivoting. (Hint: use the right-looking version of LU, argue that $A_{11} \neq 0$ so the first step can proceed, and show that even after the rank one update the rest of the matrix is still diagonally dominant.)

(4) Write your own code for LU factorization of diagonally dominant matrices, i.e. without pivoting, in a compiled language such as C++. Use the right-looking version for simplicity, overwriting A with the LU factors in-place. There is one choice to make in here though, in the double loop required for the rank one update—the outer loop could be over rows and the inner loop over columns, or vice versa. Try both loop orderings, and time your code on a variety of matrix sizes to see if you can measure a difference in performance—if there is one, explain why.

You needn't spend time optimizing this code: a simple, readable implementation is fine.

(5) Compare the performance of your non-pivoting LU code against the speed of LAPACK's `dgetrf` routine, which does general LU factorization (with pivoting—but for diagonally dominant matrices it won't ever need to, of course).

(6) One of the classic strategies to improve performance when dealing with large matrices, where memory traffic is the bottleneck, is to use "blocking". Instead of storing your matrices column-by-column, break it up into small $k \times k$ submatrices and store each of these blocks one after the other. (Each submatrix block can be stored column-by-column.) Try implementing block right-looking LU with this storage scheme (at each block step of the algorithm, factor the top-left $k \times k$ submatrix, get the next k rows of U and columns of L, then update the rest of the matrix with a rank k update), and time the performance again. For simplicity, you can assume that the input matrix has to have dimensions an integer multiple of your chosen k; try out different values of k (as a compile time constant); $k = 4$ might be a good first choice.