CPSC 542g Assignment 1 *Interpolation and Dense Linear Algebra: 10%* **Due October 4, 2007**

(1) One of the simplest ways to approximate scattered data is the "Franke" formula. Using a non-negative weighting kernel $w(r)$, data points $\{\vec{x}_i\}$ and data values $\{f_i\}$, the function is:

$$g(\vec{x}) = \frac{\sum_i^n f_i w(\|\vec{x} - \vec{x}_i\|)}{\sum_i^n w(\|\vec{x} - \vec{x}_i\|)}$$

This can be understood as a weighted average of the data values. If $w(r)$ is largest near $r = 0$, f_i gets the largest weight when $\vec{x}$ is closest to $\vec{x}_i$.

How could you pick $w(r)$ to make sure $g(\vec{x}_i) = f_i$ for each i, no matter how the data points are scattered, and at the same time recover the constant function (where every $f_i = 1$ for example) exactly? How could you further force $g(\vec{x})$ to be differentiable everywhere?

(2) Write prototype RBF code in MATLAB for the 2D thin-plate spline case ($\phi(r) = r^2 \log r$), following the template provided in `solveRBF.m` and `evalRBF.m` (in the directory `matlabRBF`). The file `viewRBF.m` allows you to view your solution to check that it is reasonable.

(3) Compare the accuracy of your favourite Franke interpolant from question (1) that satisfies all the requested conditions to the thin-plate spline RBF in question (2). That is:

- Pick a test function, say $f(x, y) = \sin(x)\cos(y)$
- Generate a set of random sample points $\vec{x}_1, \vec{x}_2, \ldots$ in $[0, 2\pi] \times [0, 2\pi]$
- For each k, consider the Franke interpolant $g_k(\vec{x})$ and the RBF interpolant $h_k(\vec{x})$ from the first k sample points.
- Measure the error of g_k on a regular grid, as $e_g(k) = \max_{i,j} |g_k(i\Delta x, j\Delta x) - f(i\Delta x, j\Delta x)|$ for $0 \le i, j \le N$ and $\Delta x = 2\pi/N$. (Choose $N = 20$ say). Similarly measure the error $e_h(k)$ of h_k.
- Which is more accurate? Estimate the cost (in $O()$ notation) of the two methods to achieve a given accuray ϵ.

(4)

Many linear algebra operations can be written so that the bulk of computation is calculating the product of two matrices. This operation tends to be the most heavily optimized of all as a result. Using the standard formula for $C = AB$:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

there are many options for computing this. Assume that the matrices are all stored in column-major order: i.e. if A is $m \times n$ and indices are zero based, then A_{ij} is stored at offset $i + mj$. This computation can be done with three nested loops, over i, j and k. Implement the different orders (e.g. outermost loop j, then i, then innermost loop k) in a compiled language. Time the algorithms for a variety of matrix sizes (you may assume A, B, and C are all square $n \times n$ matrices for simplicity, and only test in double precision floating-point). Comment on the results.

(5)

Compare your fastest code from (4) against an optimized BLAS implementation on your platform, using routine `dgemm`.

To achieve higher performance, essentially all BLAS implementations run matrix-matrix multiplication in block form. However, the BLAS API requires that matrices still be stored column by column. An alternative is to store the matrix in a "tiled" or block form: for a fixed block size M, the matrix is split into $M \times M$ submatrices, and each of these submatrices is stored as a regular column-major matrix, one after the other. Assuming for simplicity that each big matrix is $n \times n$ where n is a multiple of M, write a tiled version of the matrix-matrix multiply. M should be a compile-time constant, not a variable. Time the result and compare to the non-tiled methods. (Also double-check that the code is in fact generating the correct answer!)