M 3/4/5 JMC SC – SCIENTIFIC COMPUTING -- Dan Moore -- Due: 8pm 2 May 2014

Please have a statement that this is all you own work except where external sources are acknowledged at the start of your report

Background and discussion

The real discrete Sine transform (DST) represents the data x_i at N-1 discrete points as the sum of N-1 Sine functions of amplitude y_j :

$$x_i = \sum_{j=1}^{N-1} y_j \sin\left(\frac{i\,j\,\pi}{N}\right), \qquad 1 \le i < N. \tag{1}$$

This operation may be thought of as a matrix-vector multiply: $\underline{x} = \mathbf{S_N}\,\underline{y}$, where $\underline{x}$ and $\underline{y}$ are vectors of size N-1 and $\mathbf{S_N}$ is an N-1 by N-1 matrix whose j, k th element is $\sin(j\,k\,\pi/N)$.
In this form it will take $O(N^2)$ multiplications and $O(N^2)$ additions to find $\underline{x}$ given $\underline{y}$.

However, if the data y_j is separated into its symmetric and anti-symmetric parts, viz:

$$s_j = y_j + y_{N-j}, \quad (s_{N/2} = y_{N/2}), \quad a_j = y_j - y_{N-j}, \qquad \text{for } 1 \le j < \frac{N}{2}. \tag{2}$$

and the symmetric and anti-symmetric parts of vector $\underline{y}$ of size N-1 are written as two separate vectors, $\underline{s}$ and $\underline{a}$ respectively of sizes N/2 and N/2-1, then the N/2-1 even components of vector $\underline{x}$ (x_{2i}) are given by $\mathbf{S_{N/2}}\,\underline{a}$ and the N/2 odd components of $\underline{x}$ (x_{2i-1}) are given by $\mathbf{T_{N/2}}\,\underline{s}$ where $\mathbf{T_M}$ is an M by M matrix whose j,k th element is $\sin((2j-1)k\pi/(2M))$. Replacing the single matrix-vector multiply of size N-1 by two matrix-vector multiplies of sizes N/2 and N/2-1 reduces the total number of arithmetic operations by approximately a factor of 2. Of course, if N/2 is even, $\mathbf{S_{N/2}}$ can be factored into $\mathbf{S_{N/4}}$ and $\mathbf{T_{N/4}}$ and so on until $N/2^l$ is an odd number for some value of l.

The matrix $\mathbf{T_N}$ may be factored as well: if $\underline{x} = \mathbf{T_N}\,\underline{y}$, then if $z_{2i} = \sum_{j=1}^{N/2} [\,\mathbf{T_{N/2}}\,]_{i,j}\, y_{2j}$ and

$z_{2i-1} = \sum_{j=1}^{N/2} [\,\mathbf{U_{N/2}}\,]_{i,j}\, y_{2j-1}$ for $1 \le i \le N/2$, where $\mathbf{U_M}$ is an M by M matrix whose j,k th element is

$\sin((2j-1)(2k-1)\pi/(4M))$, then $x_i = z_{2i-1} + z_{2i}$ and $x_{N+1-i} = z_{2i-1} - z_{2i}$, for $1 \le i \le N/2$.
Again, this factorisation of the $\mathbf{T_N}$ matrix into a $\mathbf{T_{N/2}}$ and $\mathbf{U_{N/2}}$ matrix reduces the total number of arithmetic operations to calculate the product $\mathbf{T_N}\,\underline{y}$ by approximately a factor of 2!
And this factorisation also can be continued until $N/2^l$ is an odd number for some value of l.

These factorizations for $\mathbf{S_N}$ and $\mathbf{T_N}$ are well known and can be found in the literature from 1920 onwards. However, there is a factorisation for $\mathbf{U_N}$! It is less obvious, but it can be demonstrated to give the correct answer. Starting from $\underline{x} = \mathbf{U_N}\,\underline{y}$, if we write $u_i = y_{N+1-2i} - y_{N-2i}$, $v_i = y_{2i} + y_{2i+1}$ for $1 \le i < N/2$, with $u_{N/2} = y_1$ and $v_{N/2} = y_N$, then if $\underline{a} = \mathbf{T_{N/2}}\,\underline{u}$ and $\underline{b} = \mathbf{T_{N/2}}\,\underline{v}$ we can show that:

$$\begin{aligned} x_i &= \sin\left(\frac{(2i-1)\pi}{4N}\right)(-1)^{i+1} a_i + \sin\left(\frac{(2N+1-2i)\pi}{4N}\right) b_i , \\ x_{N+1-i} &= \sin\left(\frac{(2N+1-2i)\pi}{4N}\right)(-1)^{i+1} a_i - \sin\left(\frac{(2i-1)\pi}{4N}\right) b_i . \end{aligned} \tag{3}$$

Thus, the matrix $\mathbf{U_N}$ also admits a factorisation into two half size matrices, although in this case there are more operations to be performed on the data before and after half size transforms can be performed. If we can calculate $\mathbf{S_p}$, $\mathbf{T_p}$ and $\mathbf{U_p}$ for p small, then we can calculate the discrete Sine transform efficiently for the sizes N = 2p, 4p, 8p, 16p, …… Useful values of p are 2, 3 & 5. This allows the relatively dense transform size set {2,3,4,5,6,8,10,12,16,20,24,32,40,48, 64,80,96,128,160,192,256,320,384,512,640,768,1024,..}. [Although we could consider adding 7,14,28,56,112, ….. and 9, 18,36,72,144,288, etc. if we needed to fill in some of the gaps…]

For a given N there are $\log_2 N$ factorisation levels possible and each level requires O(N) operations. The total operation count is expected to be $O(N \log_2 N)$, a marked improvement over the $O(N^2)$ count of the direct matrix vector multiply.

To determine $\underline{y}$ given $\underline{x}$, we use the property of $\mathbf{S}_N$ **that it is its own inverse to within a constant! If $\underline{x} = \mathbf{S}_N \underline{y}$ then it can be proved that $\underline{y} = (2/N)\,\mathbf{S}_N \underline{x}$**. For $\mathbf{U}_N$: if $\underline{x} = \mathbf{U}_N \underline{y}$ then $\underline{y} = (2/N)\,\mathbf{U}_N \underline{x}$, likewise. The $\mathbf{T}_N$ matrix is not symmetric, unlike $\mathbf{S}_N$ and $\mathbf{U}_N$. It does satisfy $\mathbf{T}_N^t \bullet \mathbf{T}_N = (N/2)\, \mathbf{I}_N$, except that the last element on the product diagonal is N!

Summary:

I:	$[\mathbf{S}_N]_{j,k} = \sin(jk\pi/N)$.	$1 \le j,k \le N-1$		(4)
II:	$[\mathbf{T}_N]_{j,k} = \sin((2j-1)k\pi/(2N))$.	$1 \le j \le N$,	$1 \le k \le N$,	(5)
III:	$[\mathbf{U}_N]_{j,k} = \sin((2j-1)(2k-1)\pi/(4N))$.	$1 \le j,k \le N$,		(6)

The final M3SC Project

1. Write a C function to return a vector of type double of size N/2-1 (or greater) with the entries:
   $\{\sin(\pi/4), \sin(\pi/8), \sin(3\pi/8), \sin(\pi/16), \sin(3\pi/16), \sin(5\pi/16), \ldots \sin((N/2-1)\pi/N)\}$
   Its prototype should be: `double* SFactors(int).` This particular ordering should be useful for the constants required in `FastUN.` It is inefficient to calculate the same Sine values every time you need them in $\mathbf{U}_4$, $\mathbf{U}_8$, $\mathbf{U}_{16}$. etc. It is better to calculate them once and store them for future use.

2. Write 3 C functions to implement recursively the three matrix-vector products: (i) $\underline{x} = \mathbf{S}_N \underline{y}$, (ii) $\underline{x} = \mathbf{T}_N \underline{y}$ (iii) $\underline{x} = \mathbf{U}_N \underline{y}$ as defined on the previous page These functions should have the prototype (they should return an `int`: 0 for OK, -1 if called for a wrong size of N)
   ```
   int FastSN(double *x, double *y, double *w, double *S,
                  int N, int skip)
   ```
   (with the obvious changes in name for $\mathbf{T}_N$ and $\mathbf{U}_N$)
   where: `*x` is a pointer to the type double array to hold x from x[skip] to x[(N-1) * skip];
   `*y` is a pointer to the type double array to hold y from y[skip] to y[(N-1) * skip];
   `*w` is a pointer to the type double array w[skip] to w[(N-1) * skip] that can be used as vector temporary storage for intermediate work arrays such as $\underline{a}$, and $\underline{s}$, etc. The existence of the work array w **<u>should remove the need to call malloc</u>** in any of `FastSN, FastTN` or in `FastUN.`
   `*S` is a pointer to the array of useful Sine values defined in Question #1 above. [Note, this is not double ** $\mathbf{S}_N$]
   `N` is the size of the desired transform
   `skip` is the spacing between successive transform points. Skip is useful for multi-dimensional Sine transforms and in exploiting the recursive structure of the DST algorithm.

   You should assume that the data to be transformed is located at y[skip], y[2*skip], y[3*skip], …, y[(N-1)*skip] (or . . . , y[(N*skip].) and that the output array is at x[skip], x[2*skip], x[3*skip], …, x[(N-1)*skip. (or . . . , x[(N*skip].)
   `FastSN` should test N and if N is 2 or 1, calculate the results of the matrix-vector directly (taking advantage of any 1 or 0 factors!). If N is even and greater than 2 it should perform the necessary data manipulations to replace the matrix-vector product by two function calls to the appropriate half size transforms: `FastSN(N/2)` and `FastTN(N/2).`

   [Hints: 1. Write C functions to calculate the N-1 by N-1 (or N by N) matrices $\mathbf{S}_N$, $\mathbf{T}_N$, $\mathbf{T}_N^t$ and $\mathbf{U}_N$ directly to use for debugging and testing your code.
   2. Use these and a general matrix-vector multiply function to generate synthetic data $\underline{y}$ (such as $y_i = i$) to help you find any logic errors.
   3. Implementing $\mathbf{S}_1$, $\mathbf{T}_1$, and $\mathbf{U}_1$ directly makes for a complete set of transforms of the form $N = 2^m$, $m \ge 0$. Although $\mathbf{S}_2$, $\mathbf{T}_2$, and $\mathbf{U}_2$ can be factored into $\mathbf{S}_1$ and $\mathbf{T}_1$, $\mathbf{S}_2$, $\mathbf{T}_2$, and $\mathbf{U}_2$ are so trivial, but are called so often, that it saves time to calculate $\mathbf{S}_2$, $\mathbf{T}_2$, and $\mathbf{U}_2$ directly. So if N = 1 or 2 calculate the transforms directly.
   4. Debug `FastSN` for N = 2, then debug `FastTN` for N = 2, and finally debug `FastSN` for N = 4,

5. Debug **`FastUN`** for N = 2, then debug **`FastTN`** for N = 4 , and finally debug **`FastSN`** for N = 8.
6. Debug **`FastUN`** for N = 4 directly. A *testbed* program just to debug just **`FastUN`** may be useful here.
7. Putting the data for $\mathbf{S}_{N/2}$ and $\mathbf{T}_{N/2}$ into every other position in w and calling **`FastSN`** and **`FastTN`** with twice the skip can avoid any extra movement of the data, Which slows down the code. So N → N/2 and Skip → 2*Skip
8. Use the direct $\mathbf{S}_N$ C functions from Hint #1 to generate the $\underline{y}$ vector required to have $x_i = i$. Use this data to demonstrate that **`FastSN`** works for N = 16, 32, etc.

3. Determine the time taken for each transform for the permitted values of N.
For what value of N is the *fast transform* twice as fast as the direct matrix-vector multiply?
For what value of N is it 10 times faster? How much do the *overheads* of indexing addresses and of calling functions reduce the real speed of this algorithm compared to its theoretical maximum?
How many floating point arithmetic operations are required to calculate $\underline{x}$ if $\underline{x} = \mathbf{S}_2\,\underline{y}$ and this Matrix vector multiply is done explicitly to minimize operations?
Call this number W_{S2}. What are W_{T2} and W_{U2}?
If W_{SN}, W_{TN} and W_{UN} are known, what are W_{S2N}, W_{T2N} and W_{U2N}?
Hence or otherwise calculate W_{S2}, W_{S4}, W_{S8}, W_{S16}, W_{S32}, ... , $W_{S1,048,576}$
{You may use C, Maple, Excel or any program for this part!}
What is the best fit of these numbers to the function: $W_{SN} = A_2\ N \log_2 N + B_2\ N$? (3.1)

4. Use the Fast Discrete Sine transform to solve Poisson's equation on a unit line.

If $\rho_i = \rho(i\Delta) = \sum_{j=1}^{N-1} \rho^j \sin(\frac{ij\pi}{N})$ (4.1) and $\Psi_i = \Psi(i\Delta) = \sum_{j=1}^{N-1} \Psi^j \sin(\frac{ij\pi}{N})$ (4.2) are substituted into the 1-D finite difference approximation for Poisson's equation:

$\Psi_{i-1} - 2\Psi_i + \Psi_{i+1} = -\Delta^2 \rho_i$ for 0 < i < N , and the coefficients of $\sin(\frac{j\pi}{N})$ for each value of j are equated, it can be shown that

$$\left(2-2\cos\left(\frac{j\pi}{N}\right)\right)\Psi^j = \Delta^2\ \rho^j, \quad \text{for } 0<j<N. \qquad (4.3)$$

Thus, if ρ^j is calculated from ρ_i by the formula: $\rho^j = \frac{2}{N}\sum_{i=1}^{N-1} [\mathbf{S}_N]_{i,j}\,\rho_i$ (4.4)

and Ψ^j is calculated from (4.3) then $\Psi_i = \Psi(i\Delta) = \sum_{j=1}^{N-1} [\mathbf{S}_N]_{i,j}\,\Psi^j$. (4.5)

Two Fast Discrete Sine transforms are required to solve Poisson's equation using this technique: one to transform ρ_i into its Fourier components ρ^j and one to transform the Fourier Components of Ψ^j back to Ψ_i for 0 < i < N.
Redo the calculation of the solution of Poisson's equation on a uniform grid in 1-D using the 'smooth' ρ(x) distribution from previous project. It is sufficient to have a 4 column table listing:
(i) N, (ii) max(Ψ_i), (iii) x of max Ψ_i , (iv) time to solve the problem.
How large a value of N can you reach while keeping the execution time below 10 minutes with this new approach? How should the time increase as a function of N for this method.? Plot Ψ(x) vs. x.

If we write equations (4.1) and (4.2) as = $\rho(x) = \sum_{j=1}^{N-1} \rho^j \sin(j\ x\ \pi)$ and $\Psi(x) = \sum_{j=1}^{N-1} \Psi^j \sin(j\ x\ \pi)$

and substitute these directly into Poisson's equation in 1-D it can be shown that

$$j^2\pi^2\ \Psi^j = \rho^j \quad \text{for } 0<j<N. \qquad (4.6)$$

If you use this more accurate relation between Ψ^j and ρ^j in place of (4.3) how do your results change?

5. Use the Fast Discrete Sine transform to solve Poisson's equation over the 2-D square for the smooth 2-D potential from the previous project.

The relevant formulae are: $$\rho_{ij} = \rho(i\Delta,j\Delta) = \sum_{k=1}^{N-1}\sum_{l=1}^{N-1} \rho^{kl}\sin(\frac{ik\pi}{N})\sin(\frac{jl\pi}{N}) \;, \qquad (5.1)$$

$$\Psi_{ij} = \Psi(i\Delta,j\Delta) = \sum_{k=1}^{N-1}\sum_{l=1}^{N-1} \Psi^{kl}\sin(\frac{ik\pi}{N})\sin(\frac{jl\pi}{N}) \;, \qquad (5.2)$$

$$\left(4-2\cos\left(\frac{k\pi}{N}\right)-2\cos\left(\frac{l\pi}{N}\right)\right)\Psi^{kl} = \Delta^2\ \rho^{kl} \quad \text{for } 0<k,l<N. \qquad (5.3)$$

$$\rho^{kl} = \frac{4}{N^2}\sum_{i=1}^{N-1}\sum_{j=1}^{N-1}\ [\mathbf{S_N}]_{ik}[\mathbf{S_N}]_{jl}\ \ \rho_{ij} \qquad (5.4)$$

$$\Psi_{ij} = \Psi(i\Delta,j\Delta) = \sum_{k=1}^{N-1}\sum_{l=1}^{N-1}\ [\mathbf{S_N}]_{ik}[\mathbf{S_N}]_{jl}\Psi^{kl}. \qquad (5.5)$$

$$(\boldsymbol{k^2+l^2})\,\boldsymbol{\pi^2}\ \ \Psi^{kl} = \rho^{kl} \quad \text{for } 0<k,l<N. \qquad (5.6)$$

It is sufficient for this part to have a 5 column table listing:

(i) N, (ii) max(Ψ_{ij}), (iii) x of max Ψ_{ij}, (iv) y of max Ψ_{ij}, (v) time to solve the problem.

How large a value of N can you reach while keeping the execution time below 10 minutes with this new method? How should the time increase as a function of N? Draw a Contour Plot of $\Psi(x,y)$. Does execution time or computer memory limit the size of the problem you can solve with this method?

Note: Four sets of N-1 Fast Discrete Sine transforms are required to solve Poisson's equation in 2-D using this technique:

N-1 transforms to transform ρ_{ij} into its Fourier components ρ^k_j for $0<j<N$,
N-1 transforms to transform ρ^k_j into its Fourier components ρ^{kl} for $0<k<N$,
N-1 transforms to transform the Fourier Components of Ψ^{kl} back to Ψ^k_j for $0<k<N$, and
N-1 transforms to transform the Fourier Components of Ψ^k_j back to Ψ_{ij} for $0<j<N$.

7. Mastery: 3-D Poisson Equation.

Solve

$$\frac{\partial^2\Psi}{\partial x^2}+\frac{\partial^2\Psi}{\partial y^2}+\frac{\partial^2\Psi}{\partial z^2} = -\rho, \quad \text{with}$$

$$\rho(x,y,z) = \sum_{j,k,l=1}^{N-1}\rho^{jkl}\ \ \mathit{sin(j\pi x)sin(k\pi y)sin(l\pi z)}\ \ \&\ \ \Psi(x,y,z) = \sum_{j,k,l=1}^{N-1}\Psi^{jkl}\ \mathit{sin(\boldsymbol{j}\pi x)sin(\boldsymbol{k}\pi y)sin(\boldsymbol{l}\pi z)}\ ,$$

Using the smoothed 3_D potential and the unit cube domain from the previous exercise. It is sufficient for this part to have a 6 column table listing:

(i) N, (ii) max(Ψ_{ijk}), (iii) x of max Ψ_{ijk}, (iv) y of max Ψ_{ijk}, (v) z of max Ψ_{ijk},
(vi) time to solve the problem.

What limits the size of problem you can solve? Time or storage?

Using Matlab, Maple, GNUPlot or any other Graphics product, plot contours of constant Ψ in the two planes:

I. The plane passing through the points: (0,0,0), (1,1,0) and (1,1,1)
II. The plane parallel to the x-y plane passing through the point (¼, ¼,¼)