Comparison of some different Poisson solvers for a simple model problem on a square domain

Solver based on the FST - Fast Sine Transform

Main routine

```matlab
function u = ffps(f) % Fourier fast Poisson solver
[m,n] = size(f); mx = 2:m-1; nx = 2:n-1; h = 1/(m-1); dh = 1/h^2;
u = f; f(mx,nx) = f(mx,nx)*h^2; % Copy b. values to u; scale rest
f(2,nx) = f(2,nx) - f(1,nx); % Adjust RHS to make up for the non-zero
f(m-1,nx) = f(m-1,nx) - f(m,nx); % boundaries all the way around
f(mx, 2) = f(mx, 2) - f(mx,1);
f(mx,n-1) = f(mx,n-1) - f(mx,n);
v = fst(f(mx,nx)); % Apply FST to all columns of f
e = ones(n-2,1); % Form and solve tridiagonal systems
for k=2:m-1 % for each sine mode
    a = spdiags([e 2*(cos(h*pi*(k-1)))-2*e e; e e], -1:1,n-2,n-2);
v(k-1,nx-1) = (a\v(k-1,nx-1)');
end
u(nx,mx) = fst(v)/((m-1)/2); % Return to physical space; scale it.
```

Fast sine transform

```matlab
function y = fst(x) % FST - Fast Sine Transform. Input and output in form of matrix
[m,n] = size(x); % on which FST is applied to the separate columns (leading and
xe = [zeros(1,n);x;zeros(m+1,n)]; % trailing zeros not present in input or output. After
y = imag(fft(xe)); % two successive calls, any matrix emerges scaled up by a factor
y = y(2:m+1,:); % (m+1)/2 where m is the column length in x (and y).
```

Test program; uses random numbers for RHS and boundary values

```matlab
% Test Fourier Fast Poisson solver ffps for solving Poisson's
% equation on unit square
tic;
% Start timer
k = 7; % Size of test problem (2^k+1)x(2^k+1)
tk = 2^k; h = 1/tk; k2 = 2:tk;
uorig = rand(tk+1); % Generate random data in uorig,
f = uorig; % Calculate corresponding RHS f
f(k2,k2) = (uorig(k2-1,k2)+uorig(k2+1,k2)+uorig(k2,k2-1)+...uorig(k2,k2+1)-4*uorig(k2,k2))/h^2;
u = ffps(f); % Solve to recover uorig from RHS and boundary data in f.
max(max(abs(uorig-u))) % Compare interior solution against original
toc % Print out elapsed time
```

Times on a 2.0 GHz i7-8650U (4 core) CPU.

<table>
<thead>
<tr>
<th>k</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 2^k</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>time (seconds)</td>
<td>0.008</td>
<td>0.026</td>
<td>0.08</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Note: The fast Poisson solver above can be made much more efficient if the linear systems are not solved one-by-one using Matlab’s sparse matrix solver but instead solved using the Thomas algorithm simultaneously applied to all the linear systems. Also, the FST routine relies on a complex FFT of twice the size of the sin-data; it could have been done using an FFT of half the size of the sin-data.
Solver based on Multigrid

Main routine

```matlab
define function u = lmg(u,f,k);
% Linear Multi-Grid routine

% Main routine

% Initial smoothing
u = sm(u,f,k);
if k==1
    return
else
    r = zeros(tk+1);
    % Calculate residual
    r(k2,k2) = f(k2,k2) - (u(k2-1,k2)+u(k2+1,k2)+u(k2,k2-1)+u(k2,k2+1)-4*u(k2,k2))/h^2;
    ks = 3:2:tk-1;
    % Restrict residual in order to
    f1 = zeros(tk1+1);
    % create RHS for coarser grid
    f1(k21,k21) = (r(ks-1,ks)+r(ks+1,ks)+r(ks,ks-1)+r(ks,ks+1)+4*r(ks,ks))/8;
    u1 = lmg(zeros(tk1+1),f1,k-1);
    % Call lmg on coarser grid with zero initial guess
    ut(1:2:tk+1,1:2:tk+1) = u1;
    % Prolongate correction to finer grid
    ut(2:2:tk,1:2:tk+1) = 0.5*(ut(1:tk1,:,:)+ut(2:tk1+1,:,:));
    ut(:,2:2:tk+1) = 0.5*(ut(:,1:2:tk-1)+ut(:,3:2:tk+1));
    u = sm(u+ut,f,k);
% Update solution by correction and smooth it again.
end

Smoothing routine

```matlab
define function u = sm(u,f,k)
% Smoothing (relaxation) routine
omega = 4/5;
% Choose relaxation parameter
h = 1/(2^k); k2 = 2:2^k;
for j=1:4
    % Smooth 4 times with under-relaxed Jacobi
    u(k2,k2) = omega/4*(u(k2-1,k2)+u(k2+1,k2)+u(k2,k2-1)+...
                  u(k2,k2+1)-h^2*f(k2,k2))+(1-omega)*u(k2,k2);
end
```

Test program; uses random numbers for RHS and boundary values

```matlab
% Test Linear Multi-Grid solver lmg for solving Poisson's equation on unit square

% Start timer
 tic;
% Size of test problem (2^k+1)x(2^k+1)
k = 6; tk = 2^k; h = 1=tk; k2 = 2:tk;
% Generate random data in uorig,
uorig = rand(tk+1);
% Corresponding RHS f
f = zeros(tk+1,
f(k2,k2) = (uorig(k2-1,k2)+uorig(k2+1,k2)+uorig(k2,k2-1)+uorig(k2,k2+1)-4*uorig(k2,k2))/h^2;
% Zero out interior values of u; leave boundary values in place
u(k2,k2) = 0;
for j=1:10
    u = lmg(u,f,k);
% Each call to lmg runs one V-cycle
end
max(max(abs(uorig-u)))
% Compare interior solution against original
toc
% Print out elapsed time
```

Times on a 2.0 GHz i7-8650U (4 core) CPU.

<table>
<thead>
<tr>
<th>k</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2^k</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>time (seconds)</td>
<td>0.03</td>
<td>0.08</td>
<td>0.67</td>
<td>3.3</td>
</tr>
</tbody>
</table>
Matlab’s sparse direct solver

% Test using Matlab's sparse solver for solving Poisson's
% equation on unit square
k = 9; % Size of test problem (2^k+1)x(2^k+1)
tk = 2^k; h = 1/tk; k2 = 2:tk; tk1 = tk-1;
uorig = rand(tk+1); % Generate random data in uorig,
f = zeros(tk+1); % Calculate corresponding RHS f
f(k2,k2) = (uorig(k2-1,k2)+uorig(k2+1,k2)+uorig(k2,k2-1)+...+uorig(k2,k2+1)-4*uorig(k2,k2))/h^2;
e = ones(tk1,1); % Build up the coefficient matrix
I = spdiags(e,0,tk1,tk1);
A = spdiags([-2*e e],[-1 0 1],tk1,tk1);
A = kron(A,I)+kron(I,A);
b = h^2*reshape(f(k2,k2),((tk-1)^2,1)); % Form the RHS
b(1:tk1) = b(1:tk1) - uorig(k2,1);
b(end-tk+2:end) = b(end-tk+2:end) - uorig(k2,tk1);
b(1:tk1:end) = b(1:tk1:end) - uorig(1,k2);
b(tk1:tk1:end) = b(tk1:tk1:end) - uorig(tk1,k2);
tic; % Start timer
u = A\b; % Solve by Matlab's sparse direct solver
toc % Print out elapsed time
max(max(abs(uorig(k2,k2) - reshape(u,tk1,tk1))))
% Compare interior solution against original

Times on a 2.0 GHz i7-8650U (4 core) CPU.

<table>
<thead>
<tr>
<th>k</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2^k</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>time (seconds)</td>
<td>0.02</td>
<td>0.09</td>
<td>0.45</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Matlab’s sparse iterative solver bicgstab

The only change that is needed from the previous code is to swap the line

u = A\b; % Solve by Matlab's sparse direct solver

for

u = bicgstab(A,b,1e-10,1500);

Times on a 2.0 GHz i7-8650U (4 core) CPU.

<table>
<thead>
<tr>
<th>k</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2^k</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>time (seconds)</td>
<td>0.10</td>
<td>0.61</td>
<td>7.7</td>
<td>50.</td>
</tr>
</tbody>
</table>

This last test is misleadingly slow, as it makes no use of any preconditioner – imperative for good speed with this type of iterative solvers.