

## TIM'S 1-D MULTIGRID CODES

**1. Introduction.** This is the description of how I did problem 13 on page 71 and then created an FMG code. I'll start by doing a hack job that does not use the correct data structures and then add the data structures at the end. At each stage I'll test the parts and explain what I'm looking for. The equation is the Helmholtz equation with zero boundary data.

$$-u'' + \sigma u = f; u(0) = u(1) = 0.$$

I made  $\sigma$  a MATLAB global variable.

There are several MATLAB codes for this. All live on the course web page.

**2. Direct Solver.** `helmholtz.m` is a Helmholtz solver. I use MATLAB sparse matrix commands to construct the tridiagonal matrix and right side (which do not include the boundary conditions!). The I use the backslash command to solve the system.

```
function sol=helmholtz(f);
% HELMHOLTZ Solve discrete Helmholtz equation with zero boundary data
%
global sigma
nt=length(f); nl=length(f)-1; nv=nl-1;
rhs=f(2:nt-1);
sol=zeros(nt,1);
%
% a is the coefficient matrix of the differential equation
% this lets us compute the exact solution for any right side
% and make sure that everything else is ok
%
% I build a with the matlab sparse matrix commands.
%
hm2=nl*nl;
da=(sigma+2*hm2)*ones(nv,1); od=hm2*ones(nv,1);
a=spdiags([-od, da, -od], [-1,0,1], nv,nv);
air=a\rhs;
sol(2:nt-1)=air;
```

To test that the discretization is correct and that the solver does its job in  $O(N)$  work, I use the `time_test.m` program to solve a fairly nasty test problem (a bit harder than the one in problem 13).

$$-u'' + 10u = f, u(0) = u(1) = 0.$$

where the exact solution is  $u(x) = e^x \sin(\pi x)$  and

$$f(x) = -2\pi e^x \cos(\pi x) + (\pi^2 + \sigma - 1)e^x \sin(\pi x).$$

In `time_test.m` I used various values of  $h$  and tabulate  $e(h)$ , the error with stepsize  $h$ ,  $w(h)$ , the compute time for the solve as computed by the matlab `cputime` command, and the ratios  $e(h)/e(2h)$  and  $w(h)/w(2h)$ . The ration of errors should be  $\approx 1/4$ , since our scheme is second

order and the ratios of times should be  $\approx 2$ , since a tridiagonal solver takes  $O(N)$  work. The source to **time\_test.m** is at the end of this document.

In this example, I'm measuring progress toward truncation error and the error is the difference between my computed solution and the exact solution of the differential equation evaluated at the mesh points.

Here's the table. You should compare it to the table for fmg that I'll show you later. The range for  $h$  was  $1/32$  to  $1/1024$ . The results are exactly what they're supposed to be. I had to do 1000 calls to the solver to obtain decent timings.

TABLE 2.1  
*Helmholtz solver error/work data*

$h$	$e(h)$	$e(h)/e(2h)$	$1000 \times w(h)$	$w(h)/w(2h)$
0.031	5.7678e-04		1.00e-02	
0.016	1.4446e-04	2.50e-01	1.00e-02	1.00e+00
0.008	3.6119e-05	2.50e-01	1.00e-02	1.00e+00
0.004	9.0303e-06	2.50e-01	2.00e-02	2.00e+00
0.002	2.2576e-06	2.50e-01	3.00e-02	1.50e+00
0.001	5.6441e-07	2.50e-01	6.00e-02	2.00e+00

**3. Damped Jacobi Smoother.** I used  $\omega = 2/3$  as recommended and attempted to duplicate figures (see `jacobi_image.m` 2.3(a) on page 14, and 2.4 – 2.5 on page 15. The MATLAB code uses vector instructions. This time, and this time only, I've left in the loop that is equivalent to the vector instructions. Note the pain I undergo because MATLAB does not support zero-based arrays.

```
function vnew=dampj(vold,f)
% DAMPJ Damped Jacobi code for Helmholtz
%
% Tim's damped Jacobi code
% Check out the fancy matlab!
%
global sigma
%
% Stay awake! The range of unknowns is 2 to nl-1. MATLAB does not
% support zero-based arrays.
%
omega=2/3;
nl=length(vold);
h=1/(nl-1); h2=h*h;
d=omega/(2 + h*h*sigma);
vnew=vold;
%
% the matlab vector instructions do what the loop does, only faster
%
%for j=2:nl-1
```

```

%      vnew(j)=vold(j-1)+vold(j+1)+h2*f(j);
%end
vnew(2:nl-1)=vold(1:nl-2)+vold(3:nl)+h2*f(2:nl-1);
vnew=(1-omega)*vold + d*vnew;

```

**4. Intergrid Transfers.** I wrote two MATLAB codes `ctof.m` and `ftoc.m` for coarse-to-fine and fine-to-coarse intergrid transfers. My fine-to-coarse transfer is full weighting. Note the use of MATLAB vector operations in these codes. I tried to do this whenever possible. It really speeds up the run times.

```

function vfine=ctof(vcoarse)
% CTOF coarse to fine intergrid transfer by interpolation
%
nc=length(vcoarse); %nc = 2^lc +1
nf=2*(nc-1)+1;
vfine=zeros(nf,1);
vfine(1:2:nf)=vcoarse;
vfine(2:2:nf-1)=.5*(vfine(1:2:nf-2)+vfine(3:2:nf));
function vcoarse=ftoc(vfine)
% FTOC fine to coarse intergrid transfer by full weighting
%
nf=length(vfine);
nc=.5*(nf-1)+1;
vcoarse=.5*vfine(1:2:nf);
vcoarse(2:nc-1)=vcoarse(2:nc-1)+.25*(vfine(2:2:nf-2)+vfine(4:2:nf));

```

To test these codes I use the fact that linear interpolation is second order accurate. Since  $v - \text{ctof}(\text{ftoc}(v))$  and  $v - \text{ftoc}(\text{ctof}(v))$  both be  $O(h^2)$ , reduction of  $h$  to  $h/2$  should reduce both errors by a factor of four. `intergrid_test.m` does this and also verifies that the two formulae on page 92 for full weighting hold. Columns 1 and 3 of Table 4.1 are the infinity norms of the differences

$$nfc(h) = \|v - \text{ftoc}(\text{ctof}(v))\|_\infty \text{ and } ncf(h) = \|v - \text{ctof}(\text{ftoc}(v))\|_\infty.$$

Columns 2 and 4 are ratios of successive errors,

$$rfc(h) = nfc(h)/nfc(2h) \text{ and } rcf(h) = ncf(h)/ncf(2h),$$

as  $h$  is decreased from  $1/8$  to  $1/512$ . As you can see, the ratio converge nicely to  $1/4$ . `intergrid_test.m` also computes the errors in the relations on page 92, finding that the sum of the differences is roughly  $10^{-13}$ .

**5. Two grid code.** `twogrid.m` is a simple two grid code. I test it with `twogrid_test.m`. As you can see, `twogrid.m` simply puts the parts together.

```

function vnew=twogrid(vold,f,nul,nu2)
global sigma
vnew=vold;
for ij=1:nul
vnew=dampj(vnew,f);

```

TABLE 4.1  
*Test of intergrid transfers*

$h$	$nfc(h)$	$rfc(h)$	$ncf(h)$	$rcf(h)$
0.1250	9.06e-01		1.85e+00	
0.0625	3.78e-01	0.417	1.75e+00	0.946
0.0312	1.03e-01	0.274	5.67e-01	0.324
0.0156	2.71e-02	0.262	1.61e-01	0.283
0.0078	6.82e-03	0.251	4.07e-02	0.254
0.0039	1.71e-03	0.251	1.03e-02	0.252
0.0020	4.28e-04	0.250	2.57e-03	0.250

```

end
nt=length(vnew); nl=nt-1; h=1/nl; hm2=nl*nl;
av=zeros(nt,1);
av(2:nt-1)=-vnew(1:nt-2)+(2+sigma*h*h)*vnew(2:nt-1)-vnew(3:nt);
%
% prepare coarse mesh data
%
r=f-hm2*av; fc=ftoc(r);
%
% call direct solver
%
vc=helmholtz(fc);
vnew=vnew+ctof(vc);
for ij=1:nu2
vnew=dampj(vnew,f);
end

```

In `twogrid_test.m` I take 5 two-grid iterations for the problem:

$$u'' = 0; u(0) = u(1) = 0$$

with initial iterate  $v(x) = \sin(6\pi x) + \sin(17\pi x)$  for six different values of  $h$  from  $1/32$  to  $1/1024$ . By comparing the errors (in the inf norm!) at each iterate, I get an estimate of the spectral radius of the two-grid iteration matrix. This should be (and is) independent of  $h$ . Each column of the table tabulates  $\|e_i(h)\|/\|e_i(h)\|$  for  $i = 1, \dots, 5$ .

```

%
% test of two grid code: twogrid_test.m
%
global sigma
sigma=0;
lmax=10;
itmax=5;
convhist=zeros(itmax,lmax-4);
for l=5:lmax
    nl=2^l; nt=nl+1; x=0:nl; x=x'/nl; v=sin(6*pi*x)+sin(17*pi*x);
    f=zeros(nt,1);

```

```

    for it = 1:itmax
        vt=twogrid(v,f,2,2);
        convhist(it,l-4)=norm(vt,inf)/norm(v,inf);
        v=vt;
    end
end
%
% print the data from the table
%
convhist

```

TABLE 5.1  
*Two-grid performance as function of h*

i/h	0.0312	0.0156	0.0078	0.0039	0.0020	0.0010
1	3.26e-02	4.22e-02	2.13e-02	6.40e-03	1.68e-03	4.24e-04
2	4.89e-02	5.30e-02	3.63e-02	1.96e-02	1.43e-02	1.28e-02
3	5.29e-02	5.70e-02	3.81e-02	1.99e-02	1.43e-02	1.28e-02
4	5.41e-02	5.92e-02	3.90e-02	2.01e-02	1.43e-02	1.28e-02
5	5.44e-02	6.03e-02	3.92e-02	2.02e-02	1.43e-02	1.28e-02

**6. V-cycle.** Once your smoother and intergrid transfers work. Building a V-cycle code should be a simple matter of translating the loops on page 46 into code. I implemented this in two ways, one `vcycle2.m` as on page 46, where the pre and post smoothings are the coarse mesh solve and one `vcycle1.m` where I solve exactly on the coarse mesh.

Here is the source of `vcycle1.m`

```

function vnew=vcycle1(vold,f,nul,nu2)
% VCYCLE1 V-cycle with a direct solve at the coarsest level.
global sigma
lmin=3;
nt=length(f); l=log2(nt-1);
if l==lmin
    vnew=helmholtz(f);
else
    vnew=vold;
    for ij=1:nul
        vnew=dampj(vnew,f);
    end
    nt=length(vnew); nl=nt-1; h=1/nl; hm2=nl*nl;
    av=zeros(nt,1);
    av(2:nt-1)=-vnew(1:nt-2)+(2+sigma*h*h)*vnew(2:nt-1)-vnew(3:nt);
    r=f-hm2*av;
    fc=ftoc(r); nc=length(fc); vc=zeros(nc,1);
    vc=vcycle1(vc,fc,nul,nu2);
    vnew=vnew+ctof(vc);
    for ij=1:nu2

```

```

        vnew=dampj(vnew,f);
    end
end

```

To make sure that the spectral radius is independent of the mesh spacing and that the two formulations give similar performance. I used the problem from § 2. The code is `vcycle_test.m`, which is just like `twogrid_test.m`.

I measure convergence by  $e_i = v_i - ue$ , where  $ue$  is the exact solution to the discrete problem (as computed with `helmholtz.m`) and  $v_i$  is the solution for the  $i$ th iteration. I use this error because I'm measuring the convergence of the solver, and the problem the V-cycle is solving is the discrete problem.

I tabulate the ratio of the infinity norms of the errors as functions of  $h$  and  $i$  to see if ratios tend to a limit as  $i$  gets large and that this limit is independent of  $h$ . As you can see, both V-cycles are working, but the one with the exact solver on the coarse mesh is much better. There are many ways to fix this, the most obvious one being adding several more smoothing steps at the coarsest mesh (try it!).

TABLE 6.1  
*V-cycle performance, exact coarse mesh solve*

i/h	0.0312	0.0156	0.0078	0.0039	0.0020	0.0010
1	0.02	0.02	0.02	0.02	0.02	0.02
2	0.06	0.05	0.05	0.05	0.05	0.05
3	0.11	0.11	0.11	0.11	0.11	0.10
4	0.13	0.15	0.14	0.13	0.13	0.13
5	0.14	0.16	0.15	0.15	0.15	0.15

TABLE 6.2  
*V-cycle performance, coarse mesh smoothing*

i/h	0.0312	0.0156	0.0078	0.0039	0.0020	0.0010
1	0.76	0.76	0.76	0.76	0.76	0.76
2	0.76	0.76	0.76	0.76	0.76	0.76
3	0.77	0.76	0.76	0.76	0.76	0.76
4	0.77	0.77	0.77	0.77	0.77	0.77
5	0.77	0.77	0.77	0.77	0.77	0.77

**7. FMG.** After the V-cycle is done, getting FMG to work is (or should be) as simple as copying the loop from page 49. Here's the MATLAB.

```

function sol=fmg1(f,nu0,nu1,nu2)
% FMG1: recursive fmg as on page 49, but still using the simple
%     data structure which allocates and deallocates memory a lot.
%
lmin=3;
nt=length(f); l=log2(nt-1);

```

```

if l==lmin
    v=zeros(nt,1);
else
    fc=ftoc(f);
    vc=fmg1(fc,nu0,nu1,nu2);
    v=ctof(vc);
end
for i=1:nu0
    v=vcycle1(v,f,nu1,nu2);
end
sol=v;

```

I ran `time_test.m` with `fmg1.m` as the solver. Table 7.1 shows the same data as Table 2.1 did for the direct solver. As you can see I'm getting truncation error accuracy and the work is scaling linearly in the number of unknowns. I used  $\nu_0 = 2$ , you might want to see if 1 is worse and/or 3 is better.

TABLE 7.1  
*FMG1(2,1,1) solver error/work data*

$h$	$e(h)$	$e(h)/e(2h)$	$1000 \times w(h)$	$w(h)/w(2h)$
0.031	8.0575e-04		2.23e+00	
0.016	2.0129e-04	2.50e-01	3.63e+00	1.63e+00
0.008	5.0185e-05	2.49e-01	5.40e+00	1.49e+00
0.004	1.2508e-05	2.49e-01	7.53e+00	1.39e+00
0.002	3.1239e-06	2.50e-01	1.01e+01	1.35e+00
0.001	7.8092e-07	2.50e-01	1.33e+01	1.31e+00

I repeated this with `fmg2.m`, which uses `vcycle2.m` as the solver. As you might expect, this needs more smoothing to get the same kind of accuracy, and the timings reflect that. Here are the results for FMG(3,3,3).

TABLE 7.2  
*FMG2(3,3,3) solver error/work data*

$h$	$e(h)$	$e(h)/e(2h)$	$1000 \times w(h)$	$w(h)/w(2h)$
0.031	3.3158e-03	0.00e+00	5.23e+00	0.00e+00
0.016	4.0902e-04	1.23e-01	8.89e+00	1.70e+00
0.008	6.3268e-05	1.55e-01	1.36e+01	1.53e+00
0.004	1.2267e-05	1.94e-01	1.94e+01	1.43e+00
0.002	2.7411e-06	2.23e-01	2.65e+01	1.37e+00
0.001	6.5510e-07	2.39e-01	3.53e+01	1.33e+00

## 8. Serious FMG.

**8.1. Data Structure.** I store the levels in a single array. Level  $l$  has  $2^l + 1$  elements, two of which are boundary data and  $2^l - 1$  are unknowns. Even if the coarsest level is not  $l = 1$ , I store

all of them to make finding things easier. I store the right side  $f$  in the same way, even though boundary data for right hand side vectors is never used. I do this so that I can use the same data structure for right sides and solutions.

We let  $v_l$  be the solution vector at level  $l$ .  $v_l$  has length  $2^l + 1$ . The column vector

$$v = (v_1^T, v_2^T, \dots, v_L^T)^T$$

has length

$$\sum_{l=1}^L 2^l + 1 = 2^{L+1} + L - 2.$$

Hence (using MATLAB notation)

$$v_l = v(2^{l-1} + l - 3 : 2^l + l - 2).$$

**8.2. Two Grid Code.** Using this data structure I wrote a two grid code `twog.m`. I used the damped Jacobi and intergrid transfers from the previous section with no modifications. The new data structure only gets used in the two grid code.

I still have to store the residual in this code. A real code would use parts of  $f_{out}$  to do this.

```
function [vout,fout]=twog(vin,fin,nul,nu2)
%
% two grid method with the full data structure
%
global sigma
%
% cheap hack to back out the level
%
ntl=length(fin); l=max(1,floor(log2(ntl))-1);
%
% lower and upper bounds for the current mesh and the one below
%
fout=fin; vout=vin;
ifu=ntl; ifl=ntl-2^l; icu=ifl-1; icl=icu-2^(l-1);
nl=2^l; h=1/nl; hm2=nl*nl; nt=nl+1;
for ij=1:nul
    vout(ifl:ifu)=dampj(vout(ifl:ifu),fin(ifl:ifu));
end
av=zeros(nt,1);
av(2:nt-1)=-vout(ifl:ifu-2) +(2+sigma*h*h)*vout(ifl+1:ifu-1)...
    -vout(ifl+2:ifu);
%
% prepare coarse mesh data
%
r=fin(ifl:ifu)-hm2*av; fout(icl:icu)=ftoc(r);
%
% call direct solver
%
vout(icl:icu)=helmholtz(fout(icl:icu));
```

```

vout(ifl:ifu)=vout(ifl:ifu)+ctof(vout(icl:icu));
for ij=1:nu2
    vout(ifl:ifu)=dampj(vout(ifl:ifu),fin(ifl:ifu));
end

```

twogrid\_testa.m is the code that tests twog.m. It should and does produce the same results as twogrid\_test.m.

```

%
% test of two grid code with full data structure; test3a.m
%
global sigma
sigma=0;
lmax=10;
itmax=5;
convhist=zeros(itmax,lmax-4);
lv=2^(lmax+1)+lmax-2;
v=zeros(lv,1); f=zeros(lv,1);
vt=v; ft=f;
for l=5:lmax
    ntl=2^(l+1)+l-2; ifu=ntl; ifl=ntl-2^l;
    nl=2^l; nt=nl+1; x=0:nl; x=x'/nl;
    v(ifl:ifu)=sin(6*pi*x)+sin(17*pi*x);
    f(ifl:ifu)=zeros(nt,1);
    for it = 1:itmax
        [vt(1:ifu),f(1:ifu)]=twog(v(1:ifu),f(1:ifu),2,2);
        convhist(it,l-4)=norm(vt(ifl:ifu),inf)/norm(v(ifl:ifu),inf);
        v(ifl:ifu)=vt(ifl:ifu);
    end
end
convhist

```

**8.3. V-cycle.** vcycle.m begins with the two grid code. We do several smoothing steps instead of an exact coarse mesh solve, but you could easily change this. As you can see, this is a minimal change to twog.m.

```

function [vout,fout]=vcycle(vin,fin,nu1,nu2)
% VCYCLE V-cycle with the full data structure
%
global sigma
%
% cheap hack to back out the level
%
ntl=length(fin); l=max(1,floor(log2(ntl))-1);
%
% lower and upper bounds for the current mesh and the one below
%
fout=fin; vout=vin;
ifu=ntl; ifl=ntl-2^l; icu=ifl-1; icl=icu-2^(l-1);

```

```

nl=2^l; h=1/nl; hm2=nl*nl; nt=nl+1;
lmin=3;
for ij=1:nu1
    vout(ifl:ifu)=dampj(vout(ifl:ifu),fin(ifl:ifu));
end
if l > lmin
    av=zeros(nt,1);
    av(2:nt-1)=-vout(ifl:ifu-2) +(2+sigma*h*h)*vout(ifl+1:ifu-1)...
        -vout(ifl+2:ifu);
%
% prepare coarse mesh data
%
    r=fin(ifl:ifu)-hm2*av; fout(icl:icu)=ftoc(r);
vout(icl:icu)=zeros(icu-icl+1,1);
%
% recursive call
%
    [vout(1:icu),fout(1:icu)]=vcycle(vout(1:icu),fout(1:icu),nu1,nu2);
    vout(ifl:ifu)=vout(ifl:ifu)+ctof(vout(icl:icu));
end
for ij=1:nu2
    vout(ifl:ifu)=dampj(vout(ifl:ifu),fin(ifl:ifu));
end

```

I test this with `vcycle_testa.m` and the results should be, and are, exactly like those in Table 6.2.

**8.4. FMG.** As I claimed in class, once the two grid code is done, the worst is over. Here's the FMG code `fmg.m`. The calling sequence is a little different, since it needs the data structure to do its job. `time_test.m` is an example of how this works.

```

function sol=fmg(f,nu0,nu1,nu2)
%
% recursive fmg as on page 49, builds and uses the real data structure
%
nt=length(f); l=log2(nt-1);
lt=2^(l+1)-2+1;
ifu=lt; ifl=lt-2^l;
ft=zeros(lt,1); vt=zeros(lt,1); ft(ifl:ifu)=f;
[vt,ft]=fmgx(vt,ft,nu0,nu1,nu2);
sol=vt(ifl:ifu);
%
% internal fmg code
%
function [vout,fout]=fmgx(vin,fin,nu0,nu1,nu2)
%
% cheap hack to back out the level
%

```

```

ntl=length(fin); l=max(1,floor(log2(ntl))-1);
%
% lower and upper bounds for the current mesh and the one below
%
vout=vin; fout=fin;
ifu=ntl; ifl=ntl-2^l; icu=ifl-1; icl=icu-2^(l-1);
fx=zeros(ifl,1);
nl=2^l; nt=nl+1;
lmin=3;
if l==lmin
    vout(ifl:ifu)=zeros(nt,1);
else
    fout(icl:icu)=ftoc(fin(ifl:ifu));
    [vout(1:icu),fx(1:icu)]=fmgx(vout(1:icu),fout(1:icu),nu0,nu1,nu2);
    vout(ifl:ifu)=ctof(vout(icl:icu));
end
for i=1:nu0
    [vout(1:ifu),fout(1:ifu)]=vcycle(vout(1:ifu),fin(1:ifu),nu1,nu2);
end

% TIME_TEST
%
% test of solvers for
%
% -u'' + sigma u = f
%
% where the solution is u(x) = exp(x) sin(pi x) and
%
% f(x) = -2 pi exp(x) cos(pi x) + (pi^2 +sigma -1)u(x)
%
% We solve the problem on a sequence of grids and (I hope!) observe
% the errors decrease by roughly a factor of 4 and the compute times
% increase by roughly a factor of 2.
%
% Why are the increases in compute times so hard to measure?
%
% You can pick direct, or three flavors of FMG by editing the
% commented out calls to the solvers.
%
global sigma
sigma=10;
lmin=5; lmax=10; ld=lmax-lmin+1;
errdata=zeros(ld,5);
for l=lmin:lmax
    nl=2^l; x=0:nl; x=x'/nl;
    h=1/nl;
    u=exp(x).*sin(pi*x); fe=-2*pi*exp(x).*cos(pi*x)+(pi^2 + sigma -1)*u;

```

```

        tin=cputime;
%
% Pick your favorite solver and compare errors and work as
% the mesh is refined
%
% Do the solve many times to make sure you get something measurable.
    repeats=1000;
%
%     for i=1:repeats; v=fmg1(fe,3,3,3); end
%     for i=1:repeats; v=fmg2(fe,3,3,3); end
%     for i=1:repeats; v=fmg(fe,3,3,3); end
%
% v=helmholtz(fe,repeats);
% tout=cputime-tin;
% errdata(l-lmin+1,1)=h;
% errdata(l-lmin+1,4)=tout;
% errdata(l-lmin+1,2)=norm(v-u,inf);
end
errdata(2:ld,3)=errdata(2:ld,2)./errdata(1:ld-1,2);
errdata(2:ld,5)=errdata(2:ld,4)./errdata(1:ld-1,4)
    fmg.m produces the same output as fmg2.m and the cputime almost identical (but a little
    faster). Here's the table as generated by time_test.

```

TABLE 8.1  
*FMG(3,3,3) solver error/work data*

$h$	$e(h)$	$e(h)/e(2h)$	$1000 \times w(h)$	$w(h)/w(2h)$
0.031	3.3158e-03	0.00e+00	6.01e+00	0.00e+00
0.016	4.0902e-04	1.23e-01	1.02e+01	1.70e+00
0.008	6.3268e-05	1.55e-01	1.56e+01	1.53e+00
0.004	1.2267e-05	1.94e-01	2.24e+01	1.43e+00
0.002	2.7411e-06	2.23e-01	3.08e+01	1.37e+00
0.001	6.5510e-07	2.39e-01	4.20e+01	1.36e+00