

Help

```
#include "optype.h"
#include "linsys.h"
#include "mathtools.h"
#include <math.h>
#include <stdio.h>

/*CDS (Compressed Diagonal Storage) Format*/
double norm2(unsigned long n,double sx[])
{
    unsigned long i;
    double tmp;

    tmp=0.;
    for (i=1;i<=n;i++)
        tmp+=sx[i]*sx[i];

    return sqrt(tmp);
}

double dot(unsigned long n,double sx[],double sy[
    ])
{
    unsigned long i;
    double tmp;

    tmp=0.;
    for (i=1;i<=n;i++)
        tmp+=sx[i]*sy[i];

    return tmp;
}

void cp_vector(unsigned long n,double sx[],
    double sy[])
{
    unsigned long i;

    for (i=1;i<=n;i++)
        sx[i]=sy[i];
}
```

```

    return;
}

void mv_product(int n,double **band,double b[],
    double x[])
{
    int i,j,nsr;

    nsr=(int)sqrt(n);

    for (i=1;i<=n;i++)
        b[i]=0.;

    for (i=-nsr-1;i<=-nsr+1;i++)
        for (j=MAX(1,1-i);j<=MIN(n,n-i);j++)
            b[j]+=band[i+nsr+2][j]*x[i+j];

    for (i=-1;i<=1;i++)
        for (j=MAX(1,1-i);j<=MIN(n,n-i);j++)
            b[j]+=band[i+5][j]*x[i+j];

    for (i=nsr-1;i<=nsr+1;i++)
        for (j=MAX(1,1-i);j<=MIN(n,n-i);j++)
            b[j]+=band[i-nsr+8][j]*x[i+j];

    return;
}

/*Preconditioner*/

/*Diagonal Jacobi Preconditioner*/
void Diagonal_Precond(double **band, int n,
    double *pivots)
{
    int i;

    for (i=1;i<=n;i++)
        pivots[i]=(band[5][i] !=0. ? 1./band[5][i] :
            1.);
}

```

```

    return;
}

/*D-ILU Incomplete Factorization Preconditioner*/
void ILU_Precond(double **band, int n,double *pi
    vots)
{
    int i,j,z,cz,nsr;

    nsr=(int)sqrt(n);

    for(i=1;i<=n;i++)
        pivots[i]=band[5][i];

        for(i=1;i<=n;i++)
        {
            pivots[i]=1./pivots[i];
            for (j=i+1;j<=n;j++) {

                /*b c*/
                if (j==i+1) {
z=6;
cz=4;
pivots[j]-=pivots[i]*band[z][i]*band[cz][j];
                }

                /*g f*/
                if(j==i+nsr-1) {
z=7;
cz=3;
pivots[j]-=pivots[i]*band[z][i]*band[cz][j];
                }

                /*e d*/
                if(j==i+nsr) {
z=8;
cz=2;
pivots[j]-=pivots[i]*band[z][i]*band[cz][j];
                }
            }
        }
}

```

```

        /*i j*/
        if(j==i+nsr+1) {
z=9;
cz=1;
pivots[j]-=pivots[i]*band[z][i]*band[cz][j];
        }
    }
}
return;
}

/*Solving system with diagonal preconditioner*/
void solve_diag(int n,double *pivots,double **band, double b[],double x[])
{
    int i;

    for (i=1;i<=n;i++)
        x[i]=b[i]*pivots[i];

    return;
}

/*Solving system with an incomplete factorization LU preconditioner*/
int solve_ILU(int n,double *pivots,double **band, double b[],double x[])
{
    int i,nsr;
    double sum,*y;

    y=(double *)calloc(n+1,sizeof(double));
    if (y==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    nsr=(int)sqrt(n);
    /*Forward Pass*/
    for(i=1;i<=n;i++) {
        sum=0.;
        if(i>1) {
            sum+=band[4][i]*y[i-1];

```

```

        if(i>nsr+1)
sum+=band[1][i]*y[i-nsr-1];
        if (i>nsr) {
sum+=band[2][i]*y[i-nsr];
sum+=band[3][i]*y[i+1-nsr];
        }
    }
    y[i]=pivots[i]*(b[i]-sum);
}

/*Backward Pass*/
for(i=n;i>=1;i--) {
    sum=0.;
    if(i<n){
sum+=band[6][i]*y[i+1];

        if(i<n-nsr)
sum+=band[9][i]*y[i+1+nsr];
            if (i<=n-nsr) {
sum+=band[8][i]*y[i+nsr];
sum+=band[7][i]*y[i-1+nsr];
            }
        }
    x[i]=y[i]-pivots[i]*sum;
}

/*Memory desallocation*/
free(y);

return 0;
}

/* BICGSTAB ALGORITHM*/
int bicgstab(double **band,double x[],double b[],
    int n,int max_iter,double tol,int precondition,double *
    pivots)
{
    int i,j,dummy;
    double resid,tmp,tmp1;
    double rho_1,rho_2,alpha,beta,omega;
    double normb, normr,norms;

```

```

double *p,*phat,*s,*shat,*t,*v,*r,*rtilde,*btil
    de;

/*Memory Allocation*/
p=(double *)calloc(n+1,sizeof(double));
if (p==NULL)
    return MEMORY_ALLOCATION_FAILURE;

phat=(double *)calloc(n+1,sizeof(double));
if (phat==NULL)
    return MEMORY_ALLOCATION_FAILURE;

s=(double *)calloc(n+1,sizeof(double));
if (s==NULL)
    return MEMORY_ALLOCATION_FAILURE;

shat=(double *)calloc(n+1,sizeof(double));
if (shat==NULL)
    return MEMORY_ALLOCATION_FAILURE;

t=(double *)calloc(n+1,sizeof(double));
if (t==NULL)
    return MEMORY_ALLOCATION_FAILURE;

v=(double *)calloc(n+1,sizeof(double));
if (v==NULL)
    return MEMORY_ALLOCATION_FAILURE;

r=(double *)calloc(n+1,sizeof(double));
if (r==NULL)
    return MEMORY_ALLOCATION_FAILURE;

rtilde=(double *)calloc(n+1,sizeof(double));
if (rtilde==NULL)
    return MEMORY_ALLOCATION_FAILURE;

btilde=(double *)calloc(n+1,sizeof(double));
if (btilde==NULL)
    return MEMORY_ALLOCATION_FAILURE;

mv_product(n,band,btilde,x);

```

```
for (i=1;i<=n;i++) {
    r[i]=b[i]-btilde[i];
    rtilde[i]=r[i];
}

normb=norm2(n,b);
if (normb==0.) normb=1;

normr=norm2(n,r);
resid=normr/normb;
if(resid<=tol) {
    tol=resid;
    max_iter=0;
    /*Memory desallocation*/
    free(p);
    free(phat);
    free(s);
    free(shat);
    free(t);
    free(v);
    free(r);
    free(rtilde);
    free(btilde);
    return 0;
}

for (i=1;i<=max_iter;i++) {

    rho_1=dot(n,rtilde,r);
    if(rho_1==0.) {
        normr=norm2(n,r);
        tol=normr/normb;

        /*Memory desallocation*/
        free(p);
        free(phat);
        free(s);
        free(shat);
        free(t);
        free(v);
```

```

        free(r);
        free(rtilde);
        free(btilde);
        return 2;
    }

    if(i==1) cp_vector(n,p,r);
    else {
        beta=(rho_1/rho_2)*(alpha/omega);
        for (j=1;j<=n;j++)
            p[j]=r[j]+beta*(p[j]-omega*v[j]);
    }

    if(precond==1)
        solve_diag(n,pivots,band,p,phat);
    else dummy=solve_ILU(n,pivots,band,p,phat);

    mv_product(n,band,v,phat);

    tmp=dot(n,rtilde,v);
    alpha=rho_1/tmp;

    for (j=1;j<=n;j++)
        s[j]=r[j]-alpha*v[j];

    norms=norm2(n,s);
    resid=norms/normb;
    if(resid<tol) {
        for (j=1;j<=n;j++)
            x[j]+=alpha*phat[j];
        tol=resid;
    }

    /*Memory desallocation*/
    free(p);
    free(phat);
    free(s);
    free(shat);
    free(t);
    free(v);
    free(r);
    free(rtilde);

```



```

    free(btilde);
    return 0;
}

if(precond==1)
    solve_diag(n,pivots,band,s,shat);
else
    dummy=solve_ILU(n,pivots,band,s,shat);

mv_product(n,band,t,shat);

tmp=dot(n,t,s);
tmp1=dot(n,t,t);
omega=tmp/tmp1;
for (j=1;j<=n;j++) {
    x[j]+=alpha*phat[j]+omega*shat[j];
    r[j]=s[j]-omega*t[j];
}

rho_2=rho_1;
normr=norm2(n,r);
resid=normr/normb;
if(resid<tol) {
    tol=resid;
    max_iter=i;

    /*Memory desallocation*/
    free(p);
    free(phat);
    free(s);
    free(shat);
    free(t);
    free(v);
    free(r);
    free(rtilde);
    free(btilde);
    return 0;
}

if(omega==0){
    tol=normr/normb;

```

```

        /*Memory desallocation*/
        free(p);
        free(phat);
        free(s);
        free(shat);
        free(t);
        free(v);
        free(r);
        free(rtilde);
        free(btilde);
        return 3;
    }
}
tol=resid;

/*Memory desallocation*/
free(p);
free(phat);
free(s);
free(shat);
free(t);
free(v);
free(r);
free(rtilde);
free(btilde);

return 1;
}

/*GMRES ALGORITHM*/
void Update(double *x,int k,int n,double **H,
            double *s,double **V)
{
    int i,j,z;
    double *y;

    y=(double *)calloc(k+1,sizeof(double));

    for (i=0;i<=k;i++) y[i]=s[i];

    for (i=k;i>=0;i--) {

```

```
        y[i]/=H[i][i];
        for (j=i-1;j>=0;j--)
            y[j]-=H[j][i]*y[i];
    }

    for(z=1;z<=n;z++)
        for (j=0;j<=k;j++)
            x[z]+=V[j][z]*y[j];

    free(y);

    return;
}

void GeneratePlaneRotation(double dx,double dy,
    double *cs,double *sn)
{
    double temp;

    if(dy==0.) {
        *cs=1.;
        *sn=0.;
    } else if (dabs(dy) > dabs(dx)) {
        temp=dx/dy;
        *sn=1./sqrt(1.+SQR(temp));
        *cs=temp*(*sn);
    } else {
        temp= dy/dx;
        *cs=1./sqrt(1.+SQR(temp));
        *sn=temp*(*cs);
    }
    return;
}

void ApplyPlaneRotation(double *dx,double *dy,
    double cs,double sn)
{
    double temp;

    temp=cs*(*dx)+sn*(*dy);
    *dy=-sn*(*dx)+cs*(*dy);
}
```

```

    *dx=temp;

    return;
}

int gmres(double **H,double **band,double x[],
          double b[],int m,int precondition,int n,int max_iter,
          double tol,double *pivots)
{
    int i,j,k,z,dummy;
    double resid;
    double beta;
    double normb,normr;
    double **V;
    double *w,*r,*rtilde,*btilde;
    double *s,*cs,*sn;

    /*Memory Allocation*/
    w=(double *)calloc(n+1,sizeof(double));
    if (w==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    r=(double *)calloc(n+1,sizeof(double));
    if (r==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    rtilde=(double *)calloc(n+1,sizeof(double));
    if (rtilde==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    btilde=(double *)calloc(n+1,sizeof(double));
    if (btilde==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    s=(double *)calloc(m+1,sizeof(double));
    if (s==NULL)
        return MEMORY_ALLOCATION_FAILURE;

    cs=(double *)calloc(m+1,sizeof(double));
    if (cs==NULL)
        return MEMORY_ALLOCATION_FAILURE;

```

```

sn=(double *)calloc(m+1,sizeof(double));
if (sn==NULL)
    return MEMORY_ALLOCATION_FAILURE;

if(precond==1)
    solve_diag(n,pivots,band,b,btilde);
else dummy=solve_ILU(n,pivots,band,b,btilde);

normb=norm2(n,btilde);
mv_product(n,band,btilde,x);

for (i=1;i<=n;i++)
    rtilde[i]=b[i]-btilde[i];

if(precond==1)
    solve_diag(n,pivots,band,rtilde,r);
else dummy=solve_ILU(n,pivots,band,rtilde,r);

beta=norm2(n,r);
if (normb==0.) normb=1;

normr=norm2(n,r);
resid=normr/normb;

if(resid<=tol) {
    tol=resid;
    max_iter=0;
    return 0;
}

V=(double**)calloc(m+1,sizeof(double*));
if (V==NULL)
    return MEMORY_ALLOCATION_FAILURE;
for (i=0;i<m+1;i++)
{
    V[i]=(double *)calloc(n+1,sizeof(double));
    if (V[i]==NULL)
return MEMORY_ALLOCATION_FAILURE;
}

```

```

j=1;
while(j<=max_iter) {

    for (z=1;z<=n;z++)
        V[0][z]=r[z]/beta;

    for (z=1;z<=m;z++)
        s[z]=0.;
    s[0]=beta;

    for(i=0;(i<m) && (j<=max_iter);i++,j++) {
        mv_product(n,band,btilde,V[i]);

        if(precond==1)
            solve_diag(n,pivots,band,btilde,w);
        else dummy=solve_ILU(n,pivots,band,btilde,w);

        for (k=0;k<=i;k++) {
            H[k][i]=dot(n,w,V[k]);

        }

        H[i+1][i]=norm2(n,w);

        for (z=1;z<=n;z++)
            V[i+1][z]=w[z]/H[i+1][i];

        for(k=0;k<i;k++)
            ApplyPlaneRotation(&H[k][i],&H[k+1][i],cs[k],
                               sn[k]);

        GeneratePlaneRotation(H[i][i],H[i+1][i],&cs
                               [i],&sn[i]);
        ApplyPlaneRotation(&H[i][i],&H[i+1][i],cs[
                               i],sn[i]);
        ApplyPlaneRotation(&s[i],&s[i+1],cs[i],sn[
                               i]);
    }
}

```

```

        resid=dabs(s[i+1])/normb;

        if(resid<tol) {
            Update(x,i,n,H,s,V);
        }
    tol=resid;
    max_iter=j;

/*Memory desallocation*/
for (z=0;z<m+1;z++)
    free(V[z]);
free(V);
free(w);
free(r);
free(rtilde);
free(btilde);
free(s);
free(cs);
free(sn);
return 0;
}
}
Update(x,m-1,n,H,s,V);

mv_product(n,band,btilde,x);

for (z=1;z<=n;z++) {
    rtilde[z]=b[z]-btilde[z];
}
if(precond==1)
    solve_diag(n,pivots,band,rtilde,r);
else dummy=solve_ILU(n,pivots,band,rtilde,
r);

beta=norm2(n,r);
resid=beta/normb;
if(resid<tol) {
    tol=resid;
    max_iter=j;

/*Memory desallocation*/

```

```
        for (z=0;z<m+1;z++)
            free(V[z]);
        free(V);
        free(w);
        free(r);
        free(rtilde);
        free(btilde);
        free(s);
        free(cs);
        free(sn);

        return 0;
    }
}

tol=resid;

/*Memory desallocation*/
for (z=0;z<m+1;z++)
    free(V[z]);
free(V);
free(w);
free(r);
free(rtilde);
free(btilde);
free(s);
free(cs);
free(sn);

return 1;

}

/*Cholesky Decomposition*/

double *DM;

int Chol_Allocation(int dim)
{
    if (DM==NULL)
        DM=(double*)malloc(dim*sizeof(double));
```



```
    if (DM==NULL) return MEMORY_ALLOCATION_FAILURE;

    return 0;
}

void Chol_Liberation()
{
    if (DM!=NULL){
        free(DM);
        DM=NULL;
    }
}

void Cholesky(double *M, int dim)
{
    int i,j,k, erreur=0;
    double somme;

    i=0;
    while (!(erreur)&&(i<dim)){
        for (j=i;j<dim;j++){
            for (somme=M[i*dim+j],k=i-1;k>=0;k--){
                somme-=M[i*dim+k]*M[j*dim+k];
            }
            if (i==j) {
                if (somme<=0) erreur=1;
                else M[j*dim+i]=sqrt(somme);
            }
            else M[j*dim+i]=somme/M[i*dim+i];
        }
        i++;
    }
    return;
}

void InverseCholesky(double *M, int dim)
{
    int i,j,k;
    double somme;
```

```

    for (i=0;i<dim;i++)
    DM[i]=M[i*dim+i];

    for (i=0;i<dim;i++){
    M[i*dim+i]=1./DM[i];
    for (j=i+1;j<dim;j++){
        somme=0.;
        for (k=i;k<j;k++)
            somme-=M[j*dim+k]*M[k*dim+i];
        M[i*dim+j]=somme/DM[j];
        M[j*dim+i]=0.;
    }
    }
    return;
}

void Resolution(double *AuxR, double *Res,
               double *M, int dim)
{
    int i,k;
    double somme;
    for (i=0;i<dim;i++){
    for (somme=AuxR[i],k=i-1;k>=0;k--){
        somme-=M[i*dim+k]*Res[k];
    Res[i]=somme/M[i*dim+i];
    }
    for (i=dim-1;i>=0;i--){
    for (somme=Res[i],k=i+1;k<dim;k++){
        somme-=M[k*dim+i]*Res[k];
    Res[i]=somme/M[i*dim+i];
    }
    }
    return;
}

```

References