NUMERICAL RECIPES

Webnote No. 21, Rev. 1

StepperBS Implementations

As usual, the constructor simply invokes the base class instructor and initializes variables:

stepperbs.h

```
template <class D>
StepperBS<D>::StepperBS(VecDoub_IO &yy,VecDoub_IO &dydxx,Doub &xx,
    const Doub atoll,const Doub rtoll, bool dens) :
    StepperBase(yy,dydxx,xx,atoll,rtoll,dens),nseq(IMAXX),cost(IMAXX),
    table(KMAXX,n),dydxnew(n),coeff(IMAXX,IMAXX),errfac(2*IMAXX+2),ysave(IMAXX,n),
    fsave(IMAXX*(2*IMAXX+1),n),ipoint(IMAXX+1),dens((2*IMAXX+5)*n) {
```

Input to the constructor are the dependent variable `y[0..n-1]` and its derivative `dydx[0..n-1]` at the starting value of the independent variable `x`. Also input are the absolute and relative tolerances, `atol` and `rtol`, and the boolean `dense`, which is `true` if dense output is required.

```
    EPS=numeric_limits<Doub>::epsilon();
    if (dense)                                  Choose the sequence (17.3.23) ...
        for (Int i=0;i<IMAXX;i++)
            nseq[i]=4*i+2;
    else                                        ... or (17.3.6).
        for (Int i=0;i<IMAXX;i++)
            nseq[i]=2*(i+1);
    cost[0]=nseq[0]+1;                          Equation (17.3.12).
    for (Int k=0;k<KMAXX;k++) cost[k+1]=cost[k]+nseq[k+1];
    hnext=-1.0e99;                              Impossible value.
    Doub logfact=-log10(MAX(1.0e-12,rtol))*0.6+0.5;
    k_targ=MAX(1,MIN(KMAXX-1,Int(logfact)));    Initial estimate of optimal k.
    for (Int k = 0; k<IMAXX; k++) {             Coefficients in equation (17.3.8).
        for (Int l=0; l<k; l++) {
            Doub ratio=Doub(nseq[k])/nseq[l];
            coeff[k][l]=1.0/(ratio*ratio-1.0);
        }
    }
    for (Int i=0; i<2*IMAXX+1; i++) {
        Int ip5=i+5;
        errfac[i]=1.0/(ip5*ip5);
        Doub e = 0.5*sqrt(Doub(i+1)/ip5);
        for (Int j=0; j<=i; j++) {
            errfac[i] *= e/(j+1);
        }
    }
    ipoint[0]=0;
    for (Int i=1; i<=IMAXX; i++) {
        Int njadd=4*i-2;
        if (nseq[i-1] > njadd) njadd++;
        ipoint[i]=ipoint[i-1]+njadd;
    }
}
```

The `step` method attempts a step, goes through the complicated logic of controlling the stepsize and order window, and sets up the coefficients in case dense

output is needed between x and $x + h$.

stepperbs.h

```
template <class D>
void StepperBS<D>::step(const Doub htry,D &derivs) {
```

Attempts a step with stepsize `htry`. On output, `y` and `x` are replaced by their new values, `hdid` is the stepsize that was actually accomplished, and `hnext` is the estimated next stepsize.

```
    const Doub STEPFAC1=0.65,STEPFAC2=0.94,STEPFAC3=0.02,STEPFAC4=4.0,
        KFAC1=0.8,KFAC2=0.9;
    static bool first_step=true,last_step=false;
    static bool forward,reject=false,prev_reject=false;
    Int i,k;
    Doub fac,h,hnew,hopt_int,err;
    bool firstk;
    VecDoub hopt(IMAXX),work(IMAXX);
    VecDoub ysav(n),yseq(n);
    VecDoub ymid(n),scale(n);
    work[0]=0;
    h=htry;
    forward = h>0 ? true : false;
    for (i=0;i<n;i++) ysav[i]=y[i];          Save the starting values.
    if (h != hnext && !first_step) {         h gets reset in Odeint for the last step.
        last_step=true;
    }
    if (reject) {                            Previous step was rejected.
        prev_reject=true;
        last_step=false;
    }
    reject=false;
    firstk=true;
    hnew=abs(h);
    interp_error:                            Restart here if interpolation error too big.
    while (firstk || reject) {               Loop until step accepted.
        h = forward ? hnew : -hnew;
        firstk=false;
        reject=false;
        if (abs(h) <= abs(x)*EPS)
            throw("step size underflow in StepperBS");
        Int ipt=-1;                          Initialize counter for saving stuff.
        for (k=0; k<=k_targ+1;k++) {         Evaluate the sequence of modified midpoint
            dy(ysav,h,k,yseq,ipt,derivs);        integrations.
            if (k == 0)
                y=yseq;
            else                             Store result in tableau.
                for (i=0;i<n;i++)
                    table[k-1][i]=yseq[i];
            if (k != 0) {
                polyextr(k,table,y);         Perform extrapolation.
                err=0.0;                     Compute normalized error estimate err_k.
                for (i=0;i<n;i++) {
                    scale[i]=atol+rtol*MAX(abs(ysav[i]),abs(y[i]));
                    err+=SQR((y[i]-table[0][i])/scale[i]);
                }
                err=sqrt(err/n);
                Doub expo=1.0/(2*k+1);       Compute optimal stepsize for this order.
                Doub facmin=pow(STEPFAC3,expo);
                if (err == 0.0)
                    fac=1.0/facmin;
                else {
                    fac=STEPFAC2/pow(err/STEPFAC1,expo);
                    fac=MAX(facmin/STEPFAC4,MIN(1.0/facmin,fac));
                }
                hopt[k]=abs(h*fac);
                work[k]=cost[k]/hopt[k];     Work per unit step (17.3.13).
```

```
                if ((first_step || last_step) && err <= 1.0)
                    break;
                if (k == k_targ-1 && !prev_reject && !first_step && !last_step) {
                    if (err <= 1.0)          Converged within order window.
                        break;
                    else if (err>SQR(nseq[k_targ]*nseq[k_targ+1]/(nseq[0]*nseq[0])))
{
                        reject=true;         Criterion (17.3.17) predicts step will fail.
                        k_targ=k;
                        if (k_targ>1 && work[k-1]<KFAC1*work[k])
                            k_targ--;
                        hnew=hopt[k_targ];
                        break;
                    }
                }
                if (k == k_targ) {
                    if (err <= 1.0)
                        break;               Converged within order window.
                    else if (err>SQR(nseq[k+1]/nseq[0])) {
                        reject=true;         Criterion (17.3.20) predicts step will fail.
                        if (k_targ>1 && work[k-1]<KFAC1*work[k])
                            k_targ--;
                        hnew=hopt[k_targ];
                        break;
                    }
                }
                if (k == k_targ+1) {
                    if (err > 1.0) {
                        reject=true;
                        if (k_targ>1 && work[k_targ-1]<KFAC1*work[k_targ])
                            k_targ--;
                        hnew=hopt[k_targ];
                    }
                    break;
                }
            }
        }                                    Go back and try next k.
        if (reject)                          Arrive here from any break in for loop.
            prev_reject=true;
    }                                        Go back if step was rejected.
    derivs(x+h,y,dydxnew);                   Used for start of next step and in dense out-
    if (dense) {                                 put.
        prepare_dense(h,dydxnew,ysav,scale,k,err);
        hopt_int=h/MAX(pow(err,1.0/(2*k+3)),0.01);
        Stepsize based on interpolation error.
        if (err > 10.0) {                    Interpolation error too big, reject step.
            hnew=abs(hopt_int);
            reject=true;
            prev_reject=true;
            goto interp_error;
        }
    }
    dydx=dydxnew;                            Update for start of next step.
    xold=x;                                  For dense output.
    x+=h;
    hdid=h;
    first_step=false;
    Int kopt;                                Determine optimal order for next step.
    if (k == 1)
        kopt=2;
    else if (k <= k_targ) {
        kopt=k;
        if (work[k-1] < KFAC1*work[k])
            kopt=k-1;
```

```
        else if (work[k] < KFAC2*work[k-1])
            kopt=MIN(k+1,KMAXX-1);
    } else {
        kopt=k-1;
        if (k > 2 && work[k-2] < KFAC1*work[k-1])
            kopt=k-2;
        if (work[k] < KFAC2*work[kopt])
            kopt=MIN(k,KMAXX-1);
    }
    if (prev_reject) {                          After a rejected step neither order nor step-
        k_targ=MIN(kopt,k);                         size should increase.
        hnew=MIN(abs(h),hopt[k_targ]);
        prev_reject=false;
    }
    else {                                      Stepsize control for next step.
        if (kopt <= k)
            hnew=hopt[kopt];
        else {
            if (k<k_targ && work[k]<KFAC2*work[k-1])
                hnew=hopt[k]*cost[kopt+1]/cost[k];
            else
                hnew=hopt[k]*cost[kopt]/cost[k];
        }
        k_targ=kopt;
    }
    if (dense)                                  Keep interpolation error small enough.
        hnew=MIN(hnew,abs(hopt_int));
    if (forward)
        hnext=hnew;
    else
        hnext=-hnew;
}
```

The algorithm routine dy carries out the modified midpoint method.

stepperbs.h

```
template <class D>
void StepperBS<D>::dy(VecDoub_I &y,const Doub htot,const Int k,VecDoub_O &yend,
    Int &ipt,D &derivs) {
```

Modified midpoint step. Inputs are y, H, and k. The output is returned as yend[0..n-1]. The counter ipt keeps track of saving the right-hand sides in the correct locations for dense output.

```
    VecDoub ym(n),yn(n);
    Int nstep=nseq[k];
    Doub h=htot/nstep;                          Stepsize this trip.
    for (Int i=0;i<n;i++) {                     First step.
        ym[i]=y[i];
        yn[i]=y[i]+h*dydx[i];
    }
    Doub xnew=x+h;
    derivs(xnew,yn,yend);                       Use yend for temporary storage of deriva-
    Doub h2=2.0*h;                                  tives.
    for (Int nn=1;nn<nstep;nn++) {
        if (dense && nn == nstep/2) {
                for (Int i=0;i<n;i++)
                    ysave[k][i]=yn[i];
        }
        if (dense && abs(nn-nstep/2) <= 2*k+1) {
            ipt++;
            for (Int i=0;i<n;i++)
                fsave[ipt][i]=yend[i];
        }
        for (Int i=0;i<n;i++) {                 General step.
            Doub swap=ym[i]+h2*yend[i];
```

```
            ym[i]=yn[i];
            yn[i]=swap;
        }
        xnew += h;
        derivs(xnew,yn,yend);
    }
    if (dense && nstep/2 <= 2*k+1) {
        ipt++;
        for (Int i=0;i<n;i++)
            fsave[ipt][i]=yend[i];
    }
    for (Int i=0;i<n;i++)                    Last step.
        yend[i]=0.5*(ym[i]+yn[i]+h*yend[i]);
}
```

Next comes the polynomial extrapolation routine:

```
template <class D>                                                    stepperbs.h
void StepperBS<D>::polyextr(const Int k, MatDoub_IO &table, VecDoub_IO &last) {
```

Use polynomial extrapolation to evaluate `l` functions at $h = 0$. This is call number `k` in the sequence of calls. On input, `table[k-1][0..l-1]` contains the first (vector) element of the new row `k`, while `table[0..k-2][0..l-1]` contains the previous row in reverse order, except the last element, which is in `last[0..l-1]`. On output, `table` and `last` have been updated to contain row `k` of the tableau.

```
    Int l=last.size();
    for (Int j=k-1; j>0; j--)                Update the current row using the Neville re-
        for (Int i=0; i<l; i++)                  cursive formula.
            table[j-1][i]=table[j][i]+coeff[k][j]*(table[j][i]-table[j-1][i]);
    for (Int i=0; i<l; i++)                  Update the last element.
        last[i]=table[0][i]+coeff[k][0]*(table[0][i]-last[i]);
}
```

The routine `prepare_dense` sets up the dense output quantities. Our coding is closely based on that of the Fortran code ODEX of [4].

```
template <class D>                                                    stepperbs.h
void StepperBS<D>::prepare_dense(const Doub h,VecDoub_I &dydxnew,
    VecDoub_I &ysav,VecDoub_I &scale,const Int k,Doub &error) {
```

Store coefficients of interpolating polynomial for dense output in `dens` array. Input stepsize `h`, derivative at end of interval `dydxnew[0..n-1]`, function at beginning of interval `ysav[0..n-1]`, scale factor `atol`$+|y|$`rtol` in `scale[0..n-1]`, and column `k` in which convergence was achieved. Output interpolation error in `error`.

```
    mu=2*k-1;                                Degree of interpolating polynomial is mu+4.
    for (Int i=0; i<n; i++) {                Store y and y' at both ends of interval.
        dens[i]=ysav[i];
        dens[n+i]=h*dydx[i];
        dens[2*n+i]=y[i];
        dens[3*n+i]=dydxnew[i]*h;
    }
    for (Int j=1; j<=k; j++) {               Compute solution at midpoint.
        Doub dblenj=nseq[j];
        for (Int l=j; l>=1; l--) {
            Doub factor=SQR(dblenj/nseq[l-1])-1.0;
            for (Int i=0; i<n; i++)
                ysave[l-1][i]=ysave[l][i]+(ysave[l][i]-ysave[l-1][i])/factor;
        }
    }
    for (Int i=0; i<n; i++)
        dens[4*n+i]=ysave[0][i];
    for (Int kmi=1; kmi<=mu; kmi++) {        Compute kmi-th derivative at midpoint.
        Int kbeg=(kmi-1)/2;
        for (Int kk=kbeg; kk<=k; kk++) {
```

```
            Doub facnj=pow(nseq[kk]/2.0,kmi-1);
            Int ipt=ipoint[kk+1]-2*kk+kmi-3;
            for (Int i=0; i<n; i++)
                ysave[kk][i]=fsave[ipt][i]*facnj;
        }
        for (Int j=kbeg+1; j<=k; j++) {
            Doub dblenj=nseq[j];
            for (Int l=j; l>=kbeg+1; l--) {
                Doub factor=SQR(dblenj/nseq[l-1])-1.0;
                for (Int i=0; i<n; i++)
                    ysave[l-1][i]=ysave[l][i]+
                        (ysave[l][i]-ysave[l-1][i])/factor;
            }
        }
        for (Int i=0; i<n; i++)
            dens[(kmi+4)*n+i]=ysave[kbeg][i]*h;
        if (kmi == mu) continue;
        for (Int kk=kmi/2; kk<=k; kk++) {  Compute differences.
            Int lbeg=ipoint[kk+1]-1;
            Int lend=ipoint[kk]+kmi;
            if (kmi == 1) lend += 2;
            for (Int l=lbeg; l>=lend; l-=2)
                for (Int i=0; i<n; i++)
                    fsave[l][i]=fsave[l][i]-fsave[l-2][i];
            if (kmi == 1) {
                Int l=lend-2;
                for (Int i=0; i<n; i++)
                    fsave[l][i]=fsave[l][i]-dydx[i];
            }
        }
        for (Int kk=kmi/2; kk<=k; kk++) {
            Int lbeg=ipoint[kk+1]-2;
            Int lend=ipoint[kk]+kmi+1;
            for (Int l=lbeg; l>=lend; l-=2)
                for (Int i=0; i<n; i++)
                    fsave[l][i]=fsave[l][i]-fsave[l-2][i];
        }
    }
    dense_interp(n,dens,mu);                Compute the interpolation coefficients in dens.
    error=0.0;                              Estimate the interpolation error.
    if (mu >= 1) {
        for (Int i=0; i<n; i++)
            error += SQR(dens[(mu+4)*n+i]/scale[i]);
        error=sqrt(error/n)*errfac[mu-1];
    }
}
```

The next routine, `dense_out`, uses the coefficients stored by the previous routine to evaluate the solution at an arbitrary point.

stepperbs.h

```
template <class D>
Doub StepperBS<D>::dense_out(const Int i,const Doub x,const Doub h) {
```

Evaluate interpolating polynomial for `y[i]` at location `x`, where `xold` $\leq$ `x` $\leq$ `xold` $+$ `h`.

```
    Doub theta=(x-xold)/h;
    Doub theta1=1.0-theta;
    Doub yinterp=dens[i]+theta*(dens[n+i]+theta1*(dens[2*n+i]*theta
        +dens[3*n+i]*theta1));
    if (mu<0)
        return yinterp;
    Doub theta05=theta-0.5;
    Doub t4=SQR(theta*theta1);
    Doub c=dens[n*(mu+4)+i];
    for (Int j=mu;j>0; j--)
```

```
            c=dens[n*(j+3)+i]+c*theta05/j;
        yinterp += t4*c;
        return yinterp;
    }
```

The final routine is a utility routine used by `prepare_dense`.

stepperbs.h

```
template <class D>
void StepperBS<D>::dense_interp(const Int n, VecDoub_IO &y, const Int imit) {
```

Compute coefficients of the the dense interpolation formula. On input, `y[0..n*(imit+5)-1]` contains the `dens` array from `prepare_dense`. On output, these coefficients have been updated to the required values.

```
    Doub y0,y1,yp0,yp1,ydiff,aspl,bspl,ph0,ph1,ph2,ph3,fac1,fac2;
    VecDoub a(31);
    for (Int i=0; i<n; i++) {
        y0=y[i];
        y1=y[2*n+i];
        yp0=y[n+i];
        yp1=y[3*n+i];
        ydiff=y1-y0;
        aspl=-yp1+ydiff;
        bspl=yp0-ydiff;
        y[n+i]=ydiff;
        y[2*n+i]=aspl;
        y[3*n+i]=bspl;
        if (imit < 0) continue;
        ph0=(y0+y1)*0.5+0.125*(aspl+bspl);
        ph1=ydiff+(aspl-bspl)*0.25;
        ph2=-(yp0-yp1);
        ph3=6.0*(bspl-aspl);
        if (imit >= 1) {
            a[1]=16.0*(y[5*n+i]-ph1);
            if (imit >= 3) {
                a[3]=16.0*(y[7*n+i]-ph3+3*a[1]);
                for (Int im=5; im <=imit; im+=2) {
                    fac1=im*(im-1)/2.0;
                    fac2=fac1*(im-2)*(im-3)*2.0;
                    a[im]=16.0*(y[(im+4)*n+i]+fac1*a[im-2]-fac2*a[im-4]);
                }
            }
        }
        a[0]=(y[4*n+i]-ph0)*16.0;
        if (imit >= 2) {
            a[2]=(y[n*6+i]-ph2+a[0])*16.0;
            for (Int im=4; im <=imit; im+=2) {
                fac1=im*(im-1)/2.0;
                fac2=im*(im-1)*(im-2)*(im-3);
                a[im]=(y[n*(im+4)+i]+a[im-2]*fac1-a[im-4]*fac2)*16.0;
            }
        }
        for (Int im=0; im<=imit; im++)
            y[n*(im+4)+i]=a[im];
    }
}
```