NUMERICAL RECIPES

Webnote No. 24, Rev. 1

StepperSie Implementation

steppersie.h

```
template <class D>
struct StepperSie : StepperBase {
```

Semi-implicit extrapolation step for integrating stiff ODEs, with monitoring of local truncation error to adjust stepsize.

```
    typedef D Dtype;
    static const Int KMAXX=12,IMAXX=KMAXX+1;
```

`KMAXX` is the maximum number of rows used in the extrapolation.

```
    Int k_targ;                 Optimal row number for convergence.
    VecInt nseq;                Stepsize sequence.
    VecDoub cost;               A_k.
    MatDoub table;              Extrapolation tableau.
    MatDoub dfdy;               f'
    VecDoub dfdx;               ∂f/∂x (for compatibility with StepperRoss; not used.)
    Doub jac_redo;              Criterion for recomputing Jacobian.
    bool calcjac;               True if Jacobian is current.
    Doub theta;                 Recompute Jacobian if theta > jac_redo.
    MatDoub a;
    Int kright;                 Used in dense output.
    MatDoub coeff;              Coefficients in extrapolation tableau.
    MatDoub fsave;              Stores right-hand sides for dense output.
    VecDoub dens;               Stores quantities for dense interpolating polynomial.
    VecDoub factrl;             Factorials.
    StepperSie(VecDoub_IO &yy, VecDoub_IO &dydxx, Doub &xx, const Doub atol,
        const Doub rtol, bool dens);
    void step(const Doub htry,D &derivs);
    bool dy(VecDoub_I &y, const Doub htot, const Int k, VecDoub_O &yend,
        Int &ipt,VecDoub_I &scale,D &derivs);
    void polyextr(const Int k, MatDoub_IO &table, VecDoub_IO &last);
    void prepare_dense(const Doub h,VecDoub_I &ysav,VecDoub_I &scale,
        const Int k, Doub &error);
    Doub dense_out(const Int i,const Doub x,const Doub h);
    void dense_interp(const Int n, VecDoub_IO &y, const Int imit);
};
template <class D>
StepperSie<D>::StepperSie(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),dfdy(n,n),dfdx(n),calcjac(false),
    a(n,n),coeff(IMAXX,IMAXX),
    fsave((IMAXX-1)*(IMAXX+1)/2+2,n),dens((IMAXX+2)*n),factrl(IMAXX) {
```

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.

```
    static const Doub costfunc=1.0,costjac=5.0,costlu=1.0,costsolve=1.0;
```

The cost of a Jacobian is taken to be 5 function evaluations. Performance is not too sensitive to the value used.

```
    EPS=numeric_limits<Doub>::epsilon();
    jac_redo=MIN(1.0e-4,rtol);
```

```
    theta=2.0*jac_redo;                    Make sure Jacobian is computed on first step.
    nseq[0]=2;                             Sequence is different from StepperBS.
    nseq[1]=3;
    for (Int i=2;i<IMAXX;i++)
        nseq[i]=2*nseq[i-2];
    cost[0]=costjac+costlu+nseq[0]*(costfunc+costsolve);
    for (Int k=0;k<KMAXX;k++)
        cost[k+1]=cost[k]+(nseq[k+1]-1)*(costfunc+costsolve)+costlu;
    hnext=-1.0e99;                         Impossible value.
    Doub logfact=-log10(rtol+atol)*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), but ra-
        for (Int l=0; l<k; l++) {              tio not squared.
            Doub ratio=Doub(nseq[k])/nseq[l];
            coeff[k][l]=1.0/(ratio-1.0);
        }
    }
    factrl[0]=1.0;
    for (Int k=0; k<IMAXX-1; k++)
        factrl[k+1]=(k+1)*factrl[k];
}
template <class D>
void StepperSie<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=0.6,STEPFAC2=0.93,STEPFAC3=0.1,STEPFAC4=4.0,
        STEPFAC5=0.5,KFAC1=0.7,KFAC2=0.9;
    Stepsize and order control parameters are different from StepperBS.
    static bool first_step=true,last_step=false;
    static bool forward,reject=false,prev_reject=false;
    static Doub errold;
    Int i,k;
    Doub fac,h,hnew,err;
    bool firstk;
    VecDoub hopt(IMAXX),work(IMAXX);
    VecDoub ysav(n),yseq(n);
    VecDoub ymid(n),scale(n);
    work[0]=1.e30;
    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;
        theta=2.0*jac_redo;                Make sure Jacobian gets recomputed.
    }
    for (i=0;i<n;i++)                      Initial scaling.
        scale[i]=atol+rtol*abs(y[i]);
    reject=false;
    firstk=true;
    hnew=abs(h);
    compute_jac:                           Restart here if Jacobian error too big.
    if (theta > jac_redo && !calcjac) {    Evaluate Jacobian.
        derivs.jacobian(x,y,dfdx,dfdy);
        calcjac=true;
    }
    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 StepperSie");
Int ipt=-1;                          Initialize counter for saving stuff.
for (k=0; k<=k_targ+1;k++) {         The sequence of semi-implicit Euler steps.
    bool success=dy(ysav,h,k,yseq,ipt,scale,derivs);
    if (!success) {                  Stability problems, reduce stepsize.
        reject=true;
        hnew=abs(h)*STEPFAC5;
        break;
    }
    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*abs(ysav[i]);
            err+=SQR((y[i]-table[0][i])/scale[i]);
        }
        err=sqrt(err/n);
        if (err > 1.0/EPS || (k > 1 && err >= errold)) {
            reject=true;             Stability problems, reduce stepsize.
            hnew=abs(h)*STEPFAC5;
            break;
        }
        errold=max(4.0*err,1.0);
        Doub expo=1.0/(k+1);
        Compute optimal stepsize for this order. Note k instead of 2k in exponent.
        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>nseq[k_targ]*nseq[k_targ+1]*4.0) {
                reject=true;         No convergence expected by k_targ+1.
                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)          Converged within order window.
                break;
            else if (err>nseq[k+1]*2.0) {
                reject=true;         No convergence expected by k_targ+1.
                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;
            if (!calcjac) {
                theta=2.0*jac_redo;
                goto compute_jac;
            }
        }
    }                                       Go back if step was rejected.
    calcjac=false;                          Successful step. Allow Jacobian to be re-
    if (dense)                                  computed if theta too big.
        prepare_dense(h,ysav,scale,k,err);
    xold=x;                                 Used by 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 (forward)
        hnext=hnew;
    else
        hnext=-hnew;
}
template <class D>
```

```
bool StepperSie<D>::dy(VecDoub_I &y,const Doub htot,const Int k,VecDoub_O &yend,
    Int &ipt,VecDoub_I &scale,D &derivs) {
```

Semi-implicit Euler step. Inputs are y, H, k and `scale[0..n-1]`. 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 del(n),ytemp(n),dytemp(n);
    Int nstep=nseq[k];
    Doub h=htot/nstep;                          Stepsize this trip.
    for (Int i=0;i<n;i++) {                     Set up the matrix 1/h − f′.
        for (Int j=0;j<n;j++) a[i][j] = -dfdy[i][j];
        a[i][i] += 1.0/h;
    }
    LUdcmp alu(a);                              LU decomposition of the matrix.
    Doub xnew=x+h;                              Special step for nonautonomous system.
    derivs(xnew,y,del);
    for (Int i=0;i<n;i++)
        ytemp[i]=y[i];
    alu.solve(del,del);
    if (dense && nstep==k+1) {
        ipt++;
        for (Int i=0;i<n;i++)
            fsave[ipt][i]=del[i];
    }
    for (Int nn=1;nn<nstep;nn++) {              General step.
        for (Int i=0;i<n;i++)
            ytemp[i] += del[i];
        xnew += h;
        derivs(xnew,ytemp,yend);
        if (nn ==1 && k<=1) {                   Stability test and test for recomputing Jacobian.
            Doub del1=0.0;
            for (Int i=0;i<n;i++)
                del1 += SQR(del[i]/scale[i]);
            del1=sqrt(del1);
            derivs(x+h,ytemp,dytemp);
            for (Int i=0;i<n;i++)
                del[i]=dytemp[i]-del[i]/h;
            alu.solve(del,del);
            Doub del2=0.0;
            for (Int i=0;i<n;i++)
                del2 += SQR(del[i]/scale[i]);
            del2=sqrt(del2);
            theta=del2/MAX(1.0,del1);
            if (theta > 1.0)
                return false;
        }
        alu.solve(yend,del);
        if (dense && nn >= nstep-k-1) {
            ipt++;
            for (Int i=0;i<n;i++)
                fsave[ipt][i]=del[i];
        }
    }
    for (Int i=0;i<n;i++)                       Last step.
        yend[i]=ytemp[i]+del[i];
    return true;
}
template <class D>
void StepperSie<D>::polyextr(const Int k,MatDoub_IO &table,VecDoub_IO &last) {
```

Use polynomial extrapolation to evaluate `l` functions at $h = 0$. This routine is identical to the routine in `StepperBS`.

```
    Int l=last.size();
    for (Int j=k-1; j>0; j--)
        for (Int i=0; i<l; i++)
            table[j-1][i]=table[j][i]+coeff[k][j]*(table[j][i]-table[j-1][i]);
```

```
    for (Int i=0; i<l; i++)
        last[i]=table[0][i]+coeff[k][0]*(table[0][i]-last[i]);
}
template <class D>
void StepperSie<D>::prepare_dense(const Doub h,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, 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.

```
    kright=k;
    for (Int i=0; i<n; i++) {
        dens[i]=ysav[i];
        dens[n+i]=y[i];
    }
    for (Int klr=0; klr < kright; klr++) {      Compute differences.
        if (klr >= 1) {
            for (Int kk=klr; kk<=k; kk++) {
                Int lbeg=((kk+3)*kk)/2;
                Int lend=lbeg-kk+1;
                for (Int l=lbeg; l>=lend; l--)
                    for (Int i=0; i<n; i++)
                        fsave[l][i]=fsave[l][i]-fsave[l-1][i];
            }
        }
        for (Int kk=klr; kk<=k; kk++) {      Compute derivatives at right end.
            Doub facnj=nseq[kk];
            facnj=pow(facnj,klr+1)/factrl[klr+1];
            Int ipt=((kk+3)*kk)/2;
                Int krn=(kk+2)*n;
            for (Int i=0; i<n; i++) {
                dens[krn+i]=fsave[ipt][i]*facnj;
            }
        }
        for (Int j=klr+1; j<=k; j++) {
            Doub dblenj=nseq[j];
            for (Int l=j; l>=klr+1; l--) {
                Doub factor=dblenj/nseq[l-1]-1.0;
                for (Int i=0; i<n; i++) {
                    Int krn=(l+2)*n+i;
                    dens[krn-n]=dens[krn]+(dens[krn]-dens[krn-n])/factor;
                }
            }
        }
    }
    for (Int in=0; in<n; in++) {          Compute coefficients of the interpolation poly-
        for (Int j=1; j<=kright+1; j++) {     nomial.
            Int ii=n*j+in;
            dens[ii]=dens[ii]-dens[ii-n];
        }
    }
}
template <class D>
Doub StepperSie<D>::dense_out(const Int i,const Doub x,const Doub h) {
```

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

```
    Doub theta=(x-xold)/h;
    Int k=kright;
    Doub yinterp=dens[(k+1)*n+i];
    for (Int j=1; j<=k; j++)
        yinterp=dens[(k+1-j)*n+i]+yinterp*(theta-1.0);
    return dens[i]+yinterp*theta;
}
```