The Bulirsch-Stoer Method

Numerical Integration of Smooth Functions

This is some notes to help decipher the discussion of the Bulirsch-Stoer O.D.E. integration method discussed in *Numerical Recipies* section §15.4.

Strategy

The Bulirsch-Stoer method provides a numerical approximation for f(x0+H) given f(x0) and a way to calculate f'(x,y) for any x and y (also known as the right-hand-side, or rhs(x,y).)

The strategy is to first integrate from x to x+H using a smaller stepsize, h=H/n, and an inexpensive, second order method. This gives an estimate for f(x+H).

Then we iteratively increase n and obtain more estimates, using smaller sub-step sizes. The goal is to extrapolate to an estimate using a sub-step size of zero. This is done by fitting the (h,f(x+H)) pairs to a rational function and evaluating at h equal to zero.

Modified Midpoint Method

The second order method used in Bulirsch-Stoer integration is the modified midpoint method.

The modified midpoint method estimates z[i] = f(x0 + i h) for i=0...n, where h=H/n, to produce an estimate of f(x0 + H). The given value of f(x0) is used as z[0], to prime the pump.

```
        H
{h == -, z  == y[x0]}
        n   0
```

The *modified* in modified midpoint comes from the fact that the first and last guesses are calculated differently from the others. The first is a simple first order guess to x+h.

$$z_1 == h\ rhs[x0, z_0] + z_0$$

To step accross an interval of 2h, we use the estimate at the left end plus 2h times the slope at the midpoint.

```
z[i] == z[i-2] + 2 h rhs[x+(i-1)h,z[i-1]]
```

$$z_i == 2\ h\ rhs[x + h\ (-1 + i),\ z_{-1 + i}] + z_{-2 + i}$$

And the final estimate is an average of z[n] and z[n-1] + h rhs(x+H,z[n]).

$$y[H + x0] == \frac{h\ rhs[H + x0, z_n] + z_n + z_{-1 + n}}{2}$$

■ Code for mmid

This code implements the above algorithm for any number of ODEs in parallel. It's taken from *Numerical Recipies* §15.3, but it returns all of the (x+ih,z[i]) pairs plus (x+H,f(x+H)) in a list.

```
(* Modified Midpoint ODE integrator *)
mmid[y_, (* List of Initial Conditions *)
     dydx_, (* List of Initial Slopes *)
     xs_, (* Left end of X interval *)
     htot_, (* size of X interval *)
     nstep_, (* number of steps *)
     derivs_ (* derviative routine *)
     ] :=
 Block[{n,x,swap,h2,h,ym,yn,yout,
        nvar=Length[y], result},
  h=htot/nstep;
  ym=y; (* ym and yn play leapfrog across the interval *)
  yn=y + h*dydx; (* yn starts as a 1st order guess *)
  result={{xs,First[ym]}}; (* save the first point *)
  x=xs+h;
  yout = derivs[x,yn];
  h2=2*h;
  Do[AppendTo[result,{x,First[yn]}]; (* save the point *)
     swap=ym + h2*yout; (* go 2h using the midpoint's slope *)
     ym=yn;
     yn=swap;
     x += h;
     yout = derivs[x,yn];
     ,{n,2,nstep}];
  AppendTo[result, {x,First[yn]}];
  yout = (ym + yn + h*yout) / 2; (* average for the last one *)
  AppendTo[result,{x,First[yout]}];
  result // N (* return a numerical approximation *)
 ];
```

■ Case Study: x' = A1 Cos[wt] + A2 Sin[wt]

Suppose we know that the velocity given time and displacement is described by...

```
rhs=Function[{t,x},3 Cos[3 t] + 4 Sin[3 t]];
```

Let's look at an interval from t=0 to 2 using 1 large step and 8 substeps, and say x(0)=0.

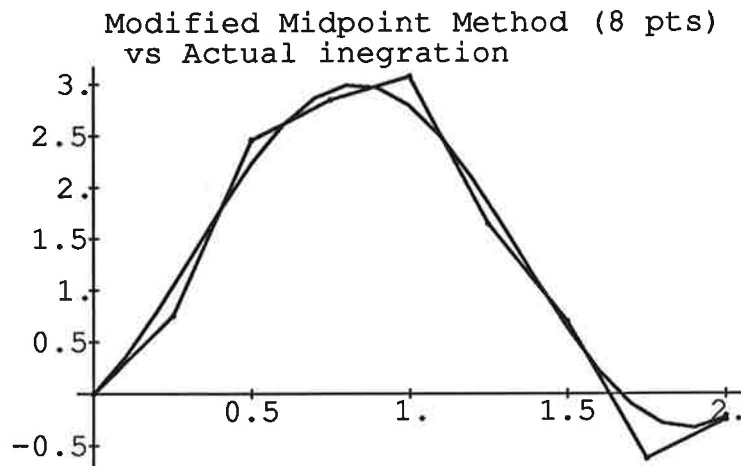
```
zPoints =
mmid[{0}, (* List of Initial Conditions *)
     {rhs[0,0]}, (* Initial Slopes *)
     0, (* Left end of interval *)
     2, (* size of interval *)
     8, (* number of steps *)
     rhs (* derviative routine *)
     ]
```

```
{{0., 0.}, {0.25, 0.75}, {0.5, 2.46081}, {0.75, 2.8511},

   {1., 3.0747}, {1.25, 1.64835}, {1.5, 0.700735},

   {1.75, -0.622907}, {2., -0.249006}, {2., -0.2156}}
```

Remember that the y value at t=2 is the result of the mmid calculation. The intermediate points are irrelevant.



■ Modified Midpoint at work

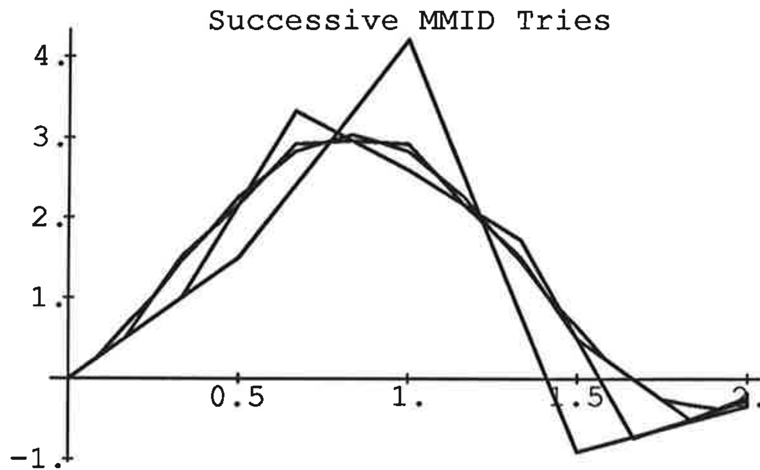
In practice, mmid is called with increasing values of n, and the results are analyzed. This is a typical sequence for n:

```
nseq={2,4,6,8,12,16,24,32,48,64,96};
```

Now we'll generate some data using the first seven sizes.

```
mmdata=Table[
 mmid[{0},           (* Initial Conditions *)
     {rhs[0,0]}, (* Initial Slopes *)
     0,              (* Left end of X interval*)
     2,               (* size of X interval *)
     nseq[[i]],       (* number of steps *)
     rhs            (* derviative routine *)
     ], {i,1,7}];
```

Let's see what it's done...



▩ Rational Extrapolation

Each time mmid produces an estimate using a given stepsize, we then extrapolate to the estimate corresponding to a stepsize of zero.

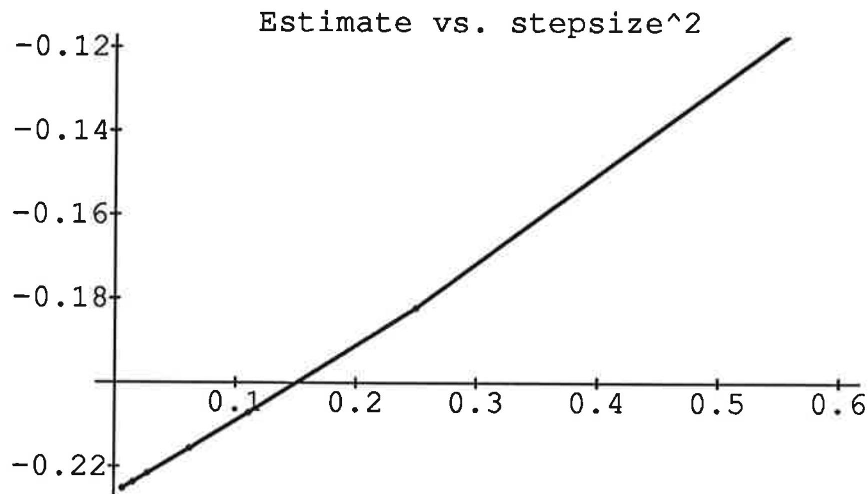
The extrapolation algorithm also produces an error estimate. We stop when the error estimate is within a given tolerance.

The Bulirsch-Stoer rational function extrapolation is based on a recurrence, similar to Neville's for polynomial extrapolation.

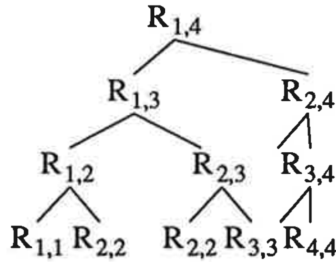
It's well known that given a set of n (xi,yi) pairs, we can find the n-1 degree polynomial that passes through them. Neville's recurrence computes the n-1 degree polynomial in terms of the n-2 degree polynomial and the nth point.

Computationally, all we care about is the *value* of the polynomial at some x value (zero, in the case of Bulirsch-Stoer). The value of the zero-degree polynomial passing through (x1,y1), is just y1 everywhere.

There is a recurrence, derived from Neville's, which gives the value of the n-1 degree polynomial at x in terms of the nth point and the value of the n-2 degree polynomial at x. In other words, each time we get a new point, we just add some offset to the previously computed value.



Let R[i,j] be the rational function passing through points i through j evaluated at x. For example R[1,2] is the value at x of the line passing through (x1,y1) and (x2,y2). The difference between R[i,i+m] and R[i,i+m-1] is called C[m,i]. For example C[2,1] is the difference between R[1,2] and R[1,3]. D[m,i] is the up-left connection from R[i+1,i+m] to R[i,i+m].



$$C_{m,i} == R_{i,i+m} - R_{i,-1+i+m}$$

$$D_{m,i} == R_{i,i+m} - R_{1+i,i+m}$$

The first generation (C[0,i], D[0,i]) are set to the y(i) values. Successive generations are given by the following reverse-derivation. It looks obtuse, but it models the *Numerical Recipies* algorithm.

```
d1 = d[m+1,i] == c[m,i+1]*dd
```

$$D_{1+m,i} == dd\ C_{m,1+i}$$

```
(* now substitute *)
d2 = d1 /. dd -> w/ddd
```

$$D_{1+m,i} == \frac{w\ C_{m,1+i}}{ddd}$$

```
d3 = d2 /. ddd -> t-c[m,i+1]
```

$$D_{1 + m,i} == \frac{w\, C_{m,1 + i}}{t - C_{m,1 + i}}$$

```
d4 = d3 /. t -> (x[i]-x)*d[m,i]/h
```

$$D_{1 + m,i} == \frac{w\, C_{m,1 + i}}{-(C_{m,1 + i}) + \frac{(-x + x_{i})\, D_{m,i}}{h}}$$

```
d5 = d4 /. h->x[i+m+1]-x
```

$$D_{1 + m,i} == \frac{w\, C_{m,1 + i}}{-(C_{m,1 + i}) + \frac{(-x + x_{i})\, D_{m,i}}{-x + x_{1 + i + m}}}$$

$$D_{1 + m,i} == \frac{(C_{m,1 + i} - D_{m,i})\, C_{m,1 + i}}{-(C_{m,1 + i}) + \frac{(-x + x_{i})\, D_{m,i}}{-x + x_{1 + i + m}}}$$

The above equation follows directly from the recurrence equation (see *Numerical Recipies*) and the definition of C and D. And similarly for C...

$$C_{1 + m,i} == \frac{(-x + x_i)\,(C_{m,1 + i} - D_{m,i})\,D_{m,i}}{(-x + x_{1 + i + m})\,\left(-(C_{m,1 + i}) + \frac{(-x + x_i)\,D_{m,i}}{-x + x_{1 + i + m}}\right)}$$

■ Code for ratint

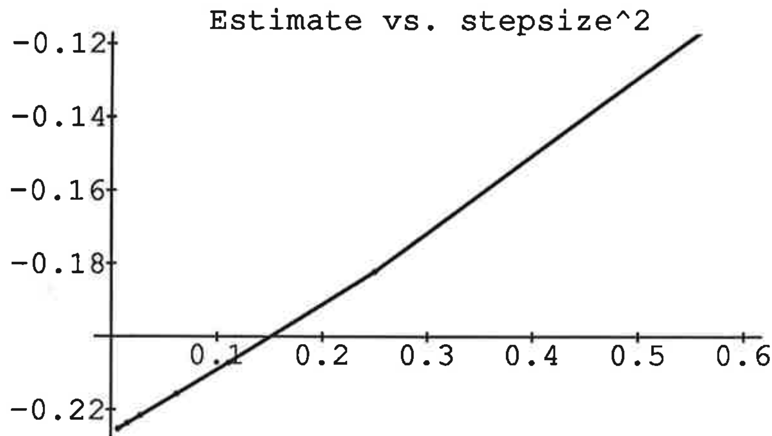
The following code copies the ya to the c and d, then sets y to the y value corresponding to the closest x. Then it computes the successive adjustments (c's and d's) and updates y accordingly.

The error estimate returned is the last y adjustment.

```
ratint[xa_, ya_] :=
 Block[{m,i,ns=1, TINY=1.0 10^-17, y, yerr,
        w,t,h,dd,c,d, n=Length[xa]},
       c=ya;
       d=c + TINY;
       y=ya[[ns--]];
       Do[Do[w=c[[i+1]]-d[[i]];
             h=xa[[i+m]];
             t=xa[[i]]*d[[i]]/h;
             dd=t-c[[i+1]];
             If[dd==0,
                Print[Message[ratint::pole]]];
             dd=w/dd;
             d[[i]]=c[[i+1]]*dd;
             c[[i]]=t*dd;
             ,{i,n-m}];
          y += (yerr =
                 If[2 ns < (n-m), c[[ns+1]],
                          d[[ns--]]]);
          ,{m,n-1}];
       {y, yerr}]
```

■ Rational Extrapolation Example

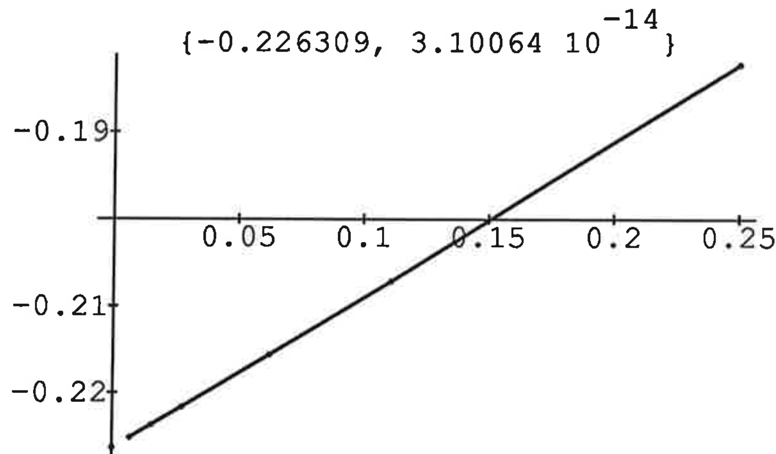
Here is a plot of (stepsize^2,estimate) from the modified midpoint data:



Now extrapolate to stepsize zero. This gives the final y and the error estimate. The estimate is the size of the last correction made in the interpolation algorithm.

```
                           -14
{-0.226309, 3.10064 10        }
```

Let's see it on the graph...



Putting it All Together

The actual BSSTEP routine just tries successively higher values of n until the extrapolation error is within tolerance.

After 10 or 12 tries, if there is still too much error, it will decrease the total stepsize, H, and try again.

It also suggests the stepsize to use next.

Here is some Fortran code that makes the whole thing work...

```
      SUBROUTINE BSSTEP(Y,DYDX,NV,X,HTRY,
     +                  EPS,YSCAL,HDID,HNEXT,DERIVS)
C     DESCRIPTION: Bulirsch-Stoer step with monitoring of local
C     truncation error to ensure accuracy and adjust stepsize.
C     BEFORE: Y is an array of the values of the NV functions
C     at X. DYDX is the values of the derivatives.
C     HTRY is the stepsize to try
C     EPS is the desired accuracy, relative to YSCAL
C     DERIVS(x,y,dydx) computes dydx at x and y
C     AFTER: Y is the estimated solution at X+HTRY
C     HNEXT is the next stepsize to use, HDID is the one used.
C     EXTERNAL: RZEXTR, MMID

      PARAMETER (NMAX = 10, ! maximum number of equasions
     +           IMAX=11,   ! maximum mmid-ratint iterations
     +           NUSE=7,    ! max qty of points to for ratint
     +           ONE=1.E0,
     +           GROW=1.2E0) ! amount to increase the stepsize
      DIMENSION Y(NV),DYDX(NV),YSCAL(NV),
     + YERR(NMAX),YSAV(NMAX),DYSAV(NMAX),YSEQ(NMAX),NSEQ(IMAX)
      DATA NSEQ /2,4,6,8,12,16,24,32,48,64,96/

C     Save input parameters
      H=HTRY
      XSAV=X
      DO I=1,NV
        YSAV(I)=Y(I)
        DYSAV(I)=DYDX(I)
      END DO

C     Iterate over stepsize-breakdowns
1     DO I=1,IMAX
C            Estimate y(x+H)
        CALL MMID(YSAV,DYSAV,NV,XSAV,H,NSEQ(I),YSEQ,DERIVS)
        XEST=(H/NSEQ(I))**2
```

```
C             Fit (stepsize^2, y-est) and extrapolate
          CALL RZEXTR(I,XEST,YSEQ,Y,YERR,NV,NUSE)
C             Compute the biggest resulting error
          ERRMAX=0.
          DO J=1,NV
            ERRMAX=MAX(ERRMAX,ABS(YERR(J)/YSCAL(J)))
          END DO
          ERRMAX = ERRMAX/EPS
C           If error is small enough...
          IF(ERRMAX.LT.ONE) THEN
            X=X+H
            HDID=H
C              Guess at next stepsize
            IF(I.EQ.NUSE)THEN
              HNEXT=H*SHRINK
            ELSE IF (I.EQ.NUSE-1) THEN
              HNEXT=H*GROW
            ELSE
              HNEXT=(H*NSEQ(NUSE-1))/NSEQ(I)
            END IF
            RETURN
          END IF
        END DO
C           Went through all substep sizes, have to decrease H.
        H=0.25*H/2**((IMAX-NUSE)/2)
        IF(X+H.EQ.X) PAUSE 'Step size underflow.'
        GOTO 1
        END

        SUBROUTINE RZEXTR(IEST,XEST,YEST,YZ,DY,NV,NUSE)
C       DESCRIPTION: Rational extrapolation to zero.
C       This is a modification of the normal RatInt routine
C       from Numerical Recipies. It's different because it
C       gets its points one at a time in stead of all at once.
C       Plus, it fits several equations in parallel, for use
C       with multiple O.D.E.s.
C       BEFORE: IEST tells how many points we now have.
C        XEST is the x coordinate of the new point.
C        YEST is a vector of the new Y coords.
C        NV is the size of YEST
C        NUSE is the maximum number of generations of rational
C             functions to compute. See equations above.
C       AFTER: YEST is an array of the extrapolation of the
C        equasions to x=0.
C        YZ is an error estimate, the last correction added to
C           get the YZs.

C            max data points, max equations, max generations
        PARAMETER (IMAX=11,NMAX=10,NCOL=7)
        DIMENSION X(IMAX),YEST(NV),YZ(NV),
```

```
     +   DY(NV),D(NMAX,NCOL),FX(NCOL)

      X(IEST)=XEST       !save latest data point
      IF(IEST.EQ.1)THEN ! fitting to just one point
        DO J=1,NV
          YZ(J)=YEST(J)
          D(J,1)=YEST(J)
          DY(J)=YEST(J)
        END DO
      ELSE
        M1=MIN(IEST,NUSE) ! actual number of points to use
C            save ???
        DO K=1,M1-1
          FX(K+1)=X(IEST-K)/XEST
        END DO
C         Iterate over equasions.
        DO J=1,NV
          YY=YEST(J)   ! first guess taken from data
C     I find the rest of this code completely inscrutible.
C     I trust it implements the equations described above.
          V=D(J,1)
          C=YY
          D(J,1)=YY
          DO K=2,M1
            B1=FX(K)*V
            B=B1-C
            IF(B.NE.0)THEN
              B=(C-V)/B
              DDY=C*B
              C=B1*B
            ELSE
              DDY=V
            END IF
            IF (K.NE.M1) V=D(J,K)
            F(J,K)=DDY
            YY=YY+DDY
          END DO
          DY(J)=DDY
          YZ(J)=YY
        END DO
      END IF
      RETURN
      END
```

```
      SUBROUTINE MMID(Y,DYDX,NVAR,XS,HTOT,NSTEP,YOUT,DERIVS)
C     DESCRIPTION: Modified Midpoint O.D.E. integrator
C     BEFORE: Y is the values of the equations at XS
C      DYDX is the values of the derivatives
C      NVAR is the size of the above
C      HTOT is the size of the interval to integrate over
C      NSTEP is the number of substeps to break it into
C      DERIVS(X,Y,DXDY) computes DXDY from X and Y
C     AFTER: YOUT is the estimated value of Y(XS+HTOT)

C     This code exactly parallels the notes above.

      PARAMETER (NMAX=10)
      DIMENSION Y(NVAR),DYDX(NVAR),YOUT(NVAR),YM(NMAX),YN(NMAX)
      H=HTOT/NSTEP
      DO I=1,NVAR
        YM(I)=Y(I)
        YN(I)=Y(I)+H*DYDX(I)
      END DO
      X=XS+H
      CALL DERIVS(X,YN,YOUT)
      H2=2.*H
      DO N=2,NSTEP
        DO I=1,NVAR
          SWAP=YM(I)+H2*YOUT(I)
          YM(I)=YN(I)
          YN(I)=SWAP
        END
        X=X+H
        CALL DERIVS(X,YN,YOUT)
      END DO
      DO I=1,NVAR
        YOUT(I)=.5*(YM(I)+YN(I)+H*YOUT(I))
      END DO
      RETURN
      END
```