NUMERICAL RECIPES
Webnote No. 3, Rev. 1

Implementation of Stiel

We now describe some of the C++ intricacies of the coding of `Stiel`. (If you are not an aficionado, you can skip to browsing the comments in the code.) The internal quadrature `quad` in `Stiel` invokes two functions `pp` and `ppx` that are the integrands in the numerators of (4.6.7). These are coded as functors internal to `Stiel`. By overloading `operator()` for the cases of one argument or two, we handle the finite and infinite range cases automatically. These functors need access to the various weight functions, mapping functions, and the orthogonal polynomial function that are members of the `Stiel` structure. We arrange this by giving each functor a pointer to the host `Stiel` object that gets initialized by the functor constructor. Conversely, the `Stiel` object needs to pass these functors to the constructors for `DErule` or `Trapzd`. It does so by creating instances of `pp` and `ppx` (called `ppfunc` and `ppxfunc` in the program), initializing them to point to the current `Stiel` object (`ppfunc.st=this` and similarly for `ppxfunc`), and passing them to `DErule` or `Trapzd`.

Here is the program:

stiel.h

```
struct Stiel {
```
Structure for calculating the abscissas and weights of an n-point Gaussian quadrature formula using the Stieltjes procedure.
```
    struct pp {
```
Functor returning the integrand for b_j in eq. (4.6.7).
```
        Stiel *st;
        Doub operator() (const Doub x, const Doub del)
```
Returns $W(x)\,p^2(x)$.
```
        {
            Doub pval=st->p(x);
            return pval*(st->wt1)(x,del)*pval;
```
This order lessens the chance of overflow.
```
        }
        Doub operator() (const Doub t)
```
Returns $W(x)\,p^2(x)\,dx/dt$.
```
        {
            Doub x=(st->fx)(t);
            Doub pval=st->p(x);
            return pval*(st->wt2)(x)*(st->fdxdt)(t)*pval;
```
This order lessens the chance of overflow.
```
        }
    };
    struct ppx {
```
Functor returning the integrand for a_j in eq. (4.6.7).
```
        Stiel *st;
        Doub operator() (const Doub x, const Doub del)
```
Returns $W(x)\,x\,p^2(x)$.
```
        {
            return st->ppfunc(x,del)*x;
        }
        Doub operator() (const Doub t)
```
Returns $W(x)\,x\,p^2(x)\,dx/dt$.
```
        {
            return st->ppfunc(t)*(st->fx)(t);
        }
    };
    Int j,n;
```
Degree of the current polynomial during the computation and desired number of abscissas.
```
    Doub aa,bb,hmax;
```
For the finite range case, limits of integration and limit of integration in transformed variable for the DE rule. For infinite range case, limits of integration in transformed variable (hmax not used.
```
    Doub (*wt1)(const Doub x, const Doub del);     Pointers to user-supplied functions.
    Doub (*wt2)(const Doub x);
    Doub (*fx)(const Doub t);
    Doub (*fdxdt)(const Doub t);
    VecDoub a,b;
```
Coefficents of the recurrence relation for the orthogonal polynomials.
```
    Quadrature *s1,*s2;
```
The two quadratures required in each iteration of eq. (4.6.7).
```
    Doub p(const Doub x);
    pp ppfunc;
    ppx ppxfunc;
    Stiel(Int nn, Doub aaa, Doub bbb, Doub hmaxx, Doub wwt1(Doub,Doub));
    Stiel(Int nn, Doub aaa, Doub bbb, Doub wwt2(Doub), Doub ffx(Doub),
        Doub ffdxdt(Doub));
    Doub quad(Quadrature *s);
    void get_weights(VecDoub_O &x, VecDoub_O &w);
};
Doub Stiel::p(const Doub x)
```
Returns the orthogonal polynomial $p_j(x)$.
```
{
    Doub pval,pj,pjm1;
    if (j == 0)
        return 1.0;
    else {                                  Compute p_j(x) using recurrence relation.
        pjm1=0.0;
        pj=1.0;
        for (Int i=0;i<j;i++) {
            pval=(x-a[i])*pj-b[i]*pjm1;
            pjm1=pj;
            pj=pval;
        }
    }
    return pval;
}

Stiel::Stiel(Int nn, Doub aaa, Doub bbb, Doub hmaxx, Doub wwt1(Doub,Doub)) :
    n(nn), aa(aaa), bb(bbb), hmax(hmaxx), wt1(wwt1), a(nn), b(nn) {
```
Constructor for finite-range case. Input are nn, the number of quadrature abscissas and weights desired, aaa and bbb, the lower and upper limits of integration, the parameter $h_{\max}$ to be passed to the DE rule (see §4.5), and the weight function coded as a function $W(x,\delta)$.
```
    ppfunc.st=this;
    ppxfunc.st=this;
    s1=new DErule<pp>(ppfunc,aa,bb,hmax);
    s2=new DErule<ppx>(ppxfunc,aa,bb,hmax);
}
Stiel::Stiel(Int nn, Doub aaa, Doub bbb, Doub wwt2(Doub), Doub ffx(Doub),
    Doub ffdxdt(Doub)) : n(nn), aa(aaa), bb(bbb), a(nn), b(nn), wt2(wwt2),
    fx(ffx), fdxdt(ffdxdt) {
```
Constructor for infinite-range case. Input are nn, the number of quadrature abscissas and weights desired, aaa and bbb, the lower and upper limits of integration, the weight function $W(x)$, the

mapping function $x(t)$ and its derivative dx/dt.

```
    ppfunc.st=this;
    ppxfunc.st=this;
    s1=new Trapzd<pp>(ppfunc,aa,bb);
    s2=new Trapzd<ppx>(ppxfunc,aa,bb);
}
```

```
Doub Stiel::quad(Quadrature *s)
```

Carries out the quadrature.

```
{
    const Doub EPS=3.0e-11, MACHEPS=numeric_limits<Doub>::epsilon();
```

The accuracy of the quadrature is very roughly the square of EPS. This choice ensures full double precision.

```
    const Int NMAX=11;
    Doub olds,sum;
    s->n=0;
    for (Int i=1;i<=NMAX;i++) {
        sum=s->next();
        if (i > 3)
```

Test for convergence. Modify to test absolute error if integral can be zero.

```
            if (abs(sum-olds) <= EPS*abs(olds))
                return sum;
        if (i == NMAX)
            if (abs(sum) <= MACHEPS && abs(olds) <= MACHEPS)
                return 0.0;
        olds=sum;
    }
    throw("no convergence in quad");
    return 0.0;
}
```

```
void Stiel::get_weights(VecDoub_O &x, VecDoub_O &w)
```

This function returns arrays x[0..n-1] and w[0..n-1] of length n, containing the abscissas and weights of the n-point Gaussian quadrature formula for the weight function $W(x)$.

```
{
    Doub amu0,c,oldc=1.0;
    if (n != x.size()) throw("bad array size in Stiel");
    for (Int i=0;i<n;i++) {              Compute a and b arrays via eq. (4.6.7).
        j=i;                             Keep track of j, degree of current polynomial.
        c=quad(s1);
        b[i]=c/oldc;                     Use b[0] to store μ0 = ∫ W(x) dx.
        a[i]=quad(s2)/c;
        oldc=c;
    }                                    The coefficients a and b of the recurrence rela-
    amu0=b[0];                               tion are available at this point if needed for
    gaucof(a,b,amu0,x,w);                    other purposes.
}
```

Note that Stiel is not written in the most efficient way possible. It recomputes the polynomials $p_j(x)$ from scratch each time they are needed. This choice makes it easy to incorporate the quadrature routines Trapzd and DErule without extensive recoding. Since typically Stiel will be used once to generate abscissas and weights that are then used many times, this trade-off of efficiency for ease of use is worthwhile.