AMath 483/583 — Lecture 17 — May 4, 2011

Today:

- Adaptive quadrature, recursive functions
- Load balancing with OpenMP
- nested forking

Friday:

- MPI

Read: Class notes and references

`$CLASSHG/codes/adaptive_quadrature`

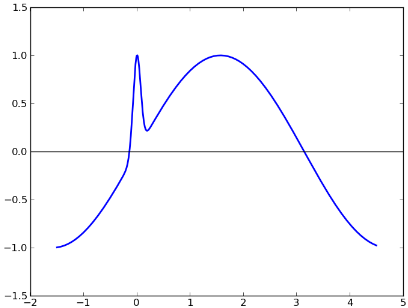
Adaptive quadrature

Problem: Approximate

$$\int_{-1}^{4} e^{-\beta^2 x^2} + \sin(x)\,dx = \left[\frac{\sqrt{\pi}}{2\beta}\mathsf{erf}(\beta x) - \cos(x)\right]_{-1}^{4}$$

where erf is the error function.

$\beta = 10$:



Adaptive Quadrature

The basic ideas will be described on the board...

See codes in `$CLASSHG/codes/adaptive_quadrature`

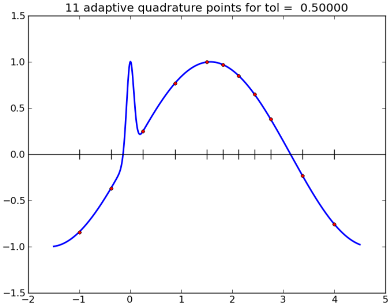
- `../serial`: Serial code with recursive subroutine
- `../openmp1`: OpenMP splitting into two pieces
- `../openmp2`: OpenMP with nested forks

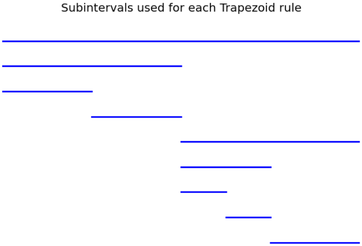
Adaptive quadrature — recursion

Selected lines from

```
! $CLASSHG/codes/adaptive_quadrature/serial/adapquad_mod.f90

recursive subroutine adapquad(f,a,b,tol,intest,errest,level,fa,fb)
! Note that level, fa, fb are optional arguments

trapezoid = 0.5d0*(b-a)*(f_a + f_b)
simpson = (b-a)*(f_a + 4.d0*fmid + f_b) / 6.d0
errest = trapezoid - simpson

if ((abs(errest) > tol) .and. (thislevel < maxlevel)) then
    tol2 = tol / 2.d0
    nextlevel = thislevel + 1
    call adapquad(f,a,xmid,tol2,intest1,errest1,nextlevel,f_a,fmid)
    call adapquad(f,xmid,b,tol2,intest2,errest2,nextlevel,fmid,f_b)
    intest = intest1 + intest2
    errest = errest1 + errest2
else
    intest = trapezoid
endif

!================
! in main program:

    call adapquad(g, a, b, tol, int_approx, errest)
```

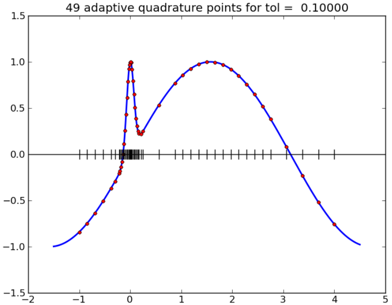
Adaptive quadrature with `tol = 0.5`

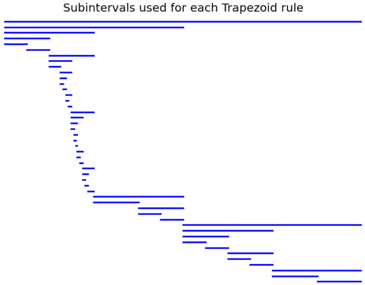





```
approx =  0.113715569029E+01
true =    0.137119131182E+01
error =  -0.234E+00
errest = -0.578E-01
g was evaluated        11 times
```

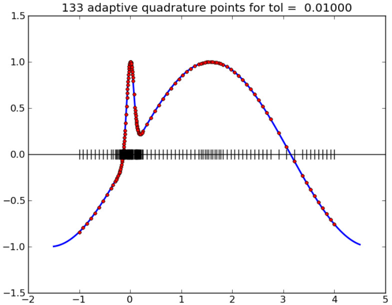
Adaptive quadrature with `tol = 0.1`

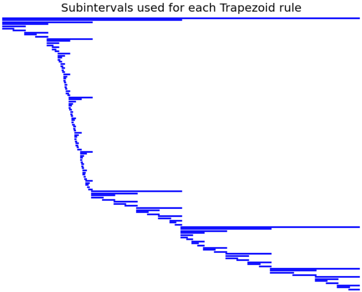





```
approx =  0.136213758404E+01
true =    0.137119131182E+01
error =  -0.905E-02
errest = -0.929E-02
g was evaluated        49 times
```

Adaptive quadrature with `tol = 0.01`






```
approx =  0.136949799545 0E+01
true =    0.137119131182 2E+01
error =  -0.169E-02
errest = -0.171E-02
g was evaluated       133 times
```

Adaptive quadrature — OpenMP

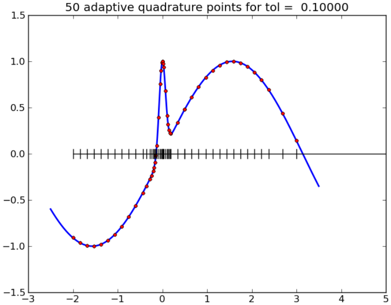
First attempt: split up original interval into 2 pieces in main program...

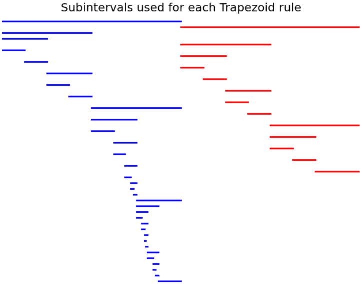
```
! $CLASSHG/codes/adaptive_quadrature/openmp1/testquad.f9

    xmid = 0.5d0*(a+b)
    tol2 = tol / 2.d0

    !$omp parallel sections
    !$omp section
        call adapquad(g,a,xmid,tol2,intest1,errest1)
    !$omp section
        call adapquad(g,xmid,b,tol2,intest2,errest2)
    !$omp end parallel sections

    int_approx = intest1 + intest2
    errest = errest1 + errest2
```

May exhibit poor load balancing if much more work has to be done in one half than the other.

Adaptive quadrature with `tol = 0.1`

Two threads, with OpenMP applied at top level only.



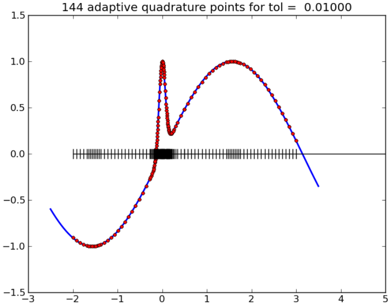


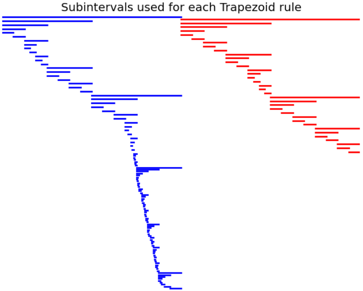
Thread 0 works only on left half,
Thread 1 works only on right half

Blue: Thread 0
Red: Thread 1

Adaptive quadrature with `tol = 0.01`

Two threads, with OpenMP applied at top level only.





Note that Thread 1 is done before Thread 0

Blue: Thread 0
Red: Thread 1

Poor load balancing if function is much smoother on one half of interval than the other!

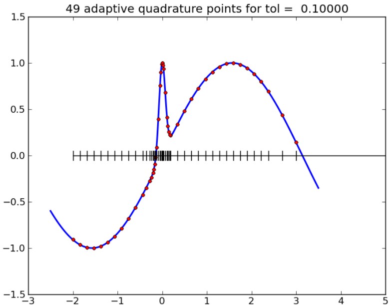
Adaptive quadrature — OpenMP

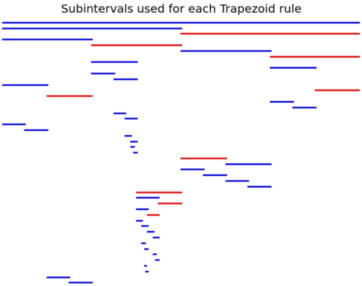
Better approach: Allow nested calls to OpenMP.

```
! $CLASSHG/codes/adaptive_quadrature/openmp2/testquad.f90

! Allow nested OpenMP threading:
!$ call omp_set_nested(.true.)

call adapquad(g, a, b, tol, int_approx, errest)

!===========

! $CLASSHG/codes/adaptive_quadrature/openmp2/adapquad_mod.f90

if ((abs(errest) > tol) .and. (thislevel < maxlevel)) then
    ! recursively apply this subroutine to each half, with
    ! tolerance tol/2 for each, and nextlevel = thislevel+1:
    tol2 = tol / 2.d0
    nextlevel = thislevel + 1

    !$omp parallel sections
    !$omp section
        call adapquad(f,a,xmid,tol2,intest1,errest1,nextlevel,f_a,fmid
    !$omp section
        call adapquad(f,xmid,b,tol2,intest2,errest2,nextlevel,fmid,f_b
    !$omp end parallel sections
```

Adaptive quadrature with `tol = 0.1`

Two threads, with nested OpenMP calls



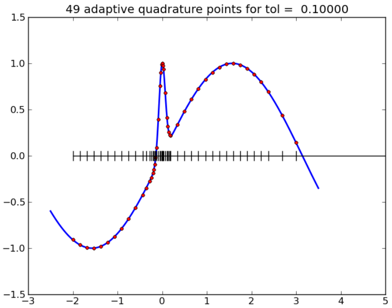


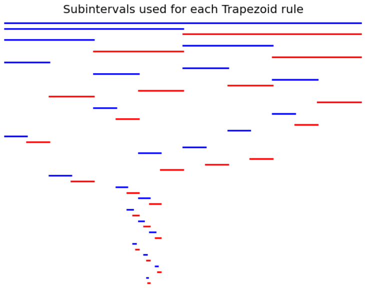
Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1

Adaptive quadrature with `tol = 0.1`

Running same thing a second time gives different pattern:



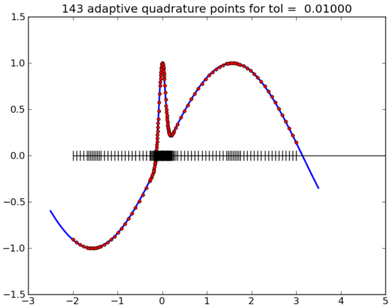


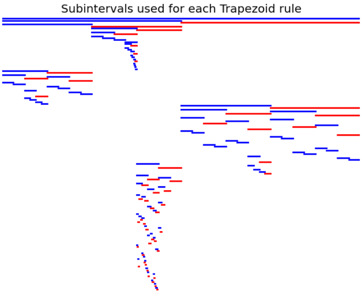
Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1

Adaptive quadrature with `tol = 0.01`

Two threads, with nested OpenMP calls





Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1