This semester’s term project consists of a friendly competition to write the most efficient Adaptive Quadrature code modeled along the same lines as the (much more sophisticated) widely available Variable Step Variable Order solvers for IVPs of ODEs. Model your program after the method we discussed on Tuesday, January 9, which is described in the notes of that date.

Thus, while it might be possible to obtain a more efficient adaptive quadrature routine by some other approach, one of the main purposes of the assignment is to gain some insight into the workings of modern VSVO solvers.

Your code must be in Matlab. Use the following template:

```matlab
% Matlab template.m
function R = adaptq(f,a,b,tau)
    % insert your code here:
end
```

Your function should return an approximation $R$ of the definite integral $I = \int_a^b f(x) \, dx$ such that

$$|I - R| \leq \tau$$  \hspace{1cm} (1)

Following is an example that shows how to use Matlab file to compute the trapezoidal approximation of

$$E = \int_0^1 e^x \, dx$$

The Matlab file `trapezoidal.m` containing the (very simple and non-adaptive) Quadrature code is

```matlab
% Matlab implementation of the trapezoidal rule, trapezoidal.m
function R = trapezoidal(f,a,b,n)
    h = (b-a)/n;
    y = h*(f(a)+f(b))/2;
    for i = 1:n-1
        y = y + h*f(a+i*h);
    end
    R = y;
end
```

The function $f$ is defined in another Matlab file as

```matlab
% Matlab function definition, f.m
function R = f(x)
```

1

1

1

1

1

1

1

1
Both of those files can be used in Matlab as follows:

```matlab
% Log of Matlab session, with blank lines removed
sunbird 57> matlab
< M A T L A B >
Copyright 1984-2006 The MathWorks, Inc.
Version 7.3.0.298 (R2006b)
August 03, 2006
To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.
>> global n
>> n = 0;
>> trapezoidal(@f,0,1,20)
ans =
1.7186
>> n
n =
21
```

Notice the ‘@’ character in front of the function name in the call to trapezoidal.

**Code requirements**

- Your code should give an approximation within the specified tolerance of $\tau$, and otherwise it should use as few function values as possible.

- It should not produce any output other than the approximation $R$ of the integral $I$. (Use semicolons to suppress output. Of course, you can omit semicolons during development for debugging purposes.)

- Your code should of course run without syntax or runtime errors. You can use similar files as those above for your testing, and you will spend a lot of time thinking about your best approach, and testing your quadrature program.
Specifically, do the following:

0. Decide whether to work by yourself or with another student in our class. If you work with a partner hand in just one set of answers. You and your partner will receive the same score for this assignment.

1. On **Wednesday, March 6, 2019** give a presentation to the whole class. Describe what you have done so far, what you are still planning to do, what your experiences have been, and what you have learned so far. This presentation is part of the assignment, and you should make sure to attend class on that particular day.

2. Email me your Matlab code by **Wednesday, April 3, 2019**. I will test your code on several examples, including those described below, and some others. I will also compare it with commercial or public quadrature and ODE software.

3. In class on April 3 hand in a written description of your code, including a mathematical description of how it works.

4. Prepare another presentation and give it during our discussion of the term project which will take place after I have tested all codes. This presentation should include distribution of a written report that may be the same as the one you handed in to me.

**Test Examples**

I will test your code on some examples that I have not yet decided on. This is realistic, if you develop code professionally you don’t beforehand what somebody will use it for. However, I will also use the following, or very similar, examples:

\[[a, b] = [0, 1], \quad f(x) = e^x, \quad \tau = 10^{-3}, 10^{-4}, \ldots, 10^{-12}. \quad (2)\]

\[[a, b] = [0, 2N \pi], \quad f(x) = \sin x, \quad N = 1, 20, 100, \quad \tau = 10^{-3}, 10^{-4}, \ldots, 10^{-12}. \quad (3)\]

\[[a, b] = [-1, \pi], \quad f(x) = \arctan(Nx), \quad N = 1, 20, 100, \quad \tau = 10^{-3}, 10^{-4}, \ldots, 10^{-12}. \quad (4)\]

The first example, the exponential, has all derivatives the same, which makes it easier to compare methods of different orders. The second example has an answer 0 which causes difficulties for any method that attempts to control the relative error. Of course, in this project we use absolute error control, see (1). The last example has an increasingly pronounced step. Presumably your code will have to work harder and harder to get past the step.

**Resources**

There is of course a huge number of books on Matlab, and much information is built into Matlab or available online. My own main source of information on Matlab is

VSVO methods


These codes are written in Fortran. For your information and entertainment I put links to them onto our class web page.

Several ODE codes are built into Matlab. In particular, ode113 and ode15s are VSVO methods.

Of course one would expect that a special purpose method for quadrature problems performs better than a general purpose method for ODE-IVPs. On the other hand, those are professional codes that presumably adhere to more stringent standards than our codes.

We will see how all that software will stack up!

Following are the instructions for the use of lsode and shpgor. Contrast the complexity of those codes with the complexity of what we are contemplating in this semester project.

```fortran
C-----------------------------------------------------------------------
SUBROUTINE LSODE (F, NEQ, Y, T, TOUT, ITOL, RTOL, ATOL, ITASK, 1
ISTATE, IOPT, RWORK, LRW, IWORK, LIW, JAC, MF)
EXTERNAL F, JAC
INTEGER NEQ, ITOL, ITASK, ISTATE, IOPT, LRW, IWORK, LIW, MF
DOUBLE PRECISION Y, T, TOUT, RTOL, ATOL, RWORK
DIMENSION NEQ(1), Y(1), RTOL(1), ATOL(1), RWORK(LRW), IWORK(LIW)
C-----------------------------------------------------------------------
C THIS IS THE AUGUST 13, 1981 VERSION OF
C LSODE.. LIVERMORE SOLVER FOR ORDINARY DIFFERENTIAL EQUATIONS.
C THIS VERSION IS IN DOUBLE PRECISION.
C
C LSODE SOLVES THE INITIAL VALUE PROBLEM FOR STIFF OR NONSTIFF
C SYSTEMS OF FIRST ORDER ODE-S,
C DY/DT = F(T,Y) , OR, IN COMPONENT FORM,
C DY(I)/DT = F(I) = F(I,T,Y(1),Y(2),...,Y(N)) (I = 1,...,NEQ).
C LSODE IS A PACKAGE BASED ON THE GEAR AND GEARB PACKAGES, AND ON THE
C OCTOBER 23, 1978 VERSION OF THE TENTATIVE ODEPACK USER INTERFACE
C STANDARD, WITH MINOR MODIFICATIONS.
C-----------------------------------------------------------------------
C REFERENCE.. ALAN C. HINDMARSH, LSODE AND LSODI, TWO NEW INITIAL VALUE
C ORDINARY DIFFERENTIAL EQUATION SOLVERS,
C-----------------------------------------------------------------------
C AUTHOR AND CONTACT.. ALAN C. HINDMARSH,
C MATHEMATICS AND STATISTICS DIVISION, L-316
C LAWRENCE LIVERMORE NATIONAL LABORATORY
C LIVERMORE, CA 94550.
C-----------------------------------------------------------------------
C SUMMARY OF USAGE.
```
COMMUNICATION BETWEEN THE USER AND THE LSODE PACKAGE, FOR NORMAL SITUATIONS, IS SUMMARIZED HERE. THIS SUMMARY DESCRIBES ONLY A SUBSET OF THE FULL SET OF OPTIONS AVAILABLE. SEE THE FULL DESCRIPTION FOR DETAILS, INCLUDING OPTIONAL COMMUNICATION, NONSTANDARD OPTIONS, AND INSTRUCTIONS FOR SPECIAL SITUATIONS. SEE ALSO THE EXAMPLE PROBLEM (WITH PROGRAM AND OUTPUT) FOLLOWING THIS SUMMARY.

A. FIRST PROVIDE A SUBROUTINE OF THE FORM...

    SUBROUTINE F (NEQ, T, Y, YDOT)
    DIMENSION Y(NEQ), YDOT(NEQ)
    WHICH SUPPLIES THE VECTOR FUNCTION F BY LOADING YDOT(I) WITH F(I).

B. NEXT DETERMINE (OR GUESS) WHETHER OR NOT THE PROBLEM IS STIFF.


C. IF THE PROBLEM IS STIFF, YOU ARE ENCOURAGED TO SUPPLY THE JACOBIAN DIRECTLY (MF = 21 OR 24), BUT IF THIS IS NOT FEASIBLE, LSODE WILL COMPUTE IT INTERNALLY BY DIFFERENCE QUOTIENTS (MF = 22 OR 25). IF YOU ARE SUPPLYING THE JACOBIAN, PROVIDE A SUBROUTINE OF THE FORM...

    SUBROUTINE JAC (NEQ, T, Y, ML, MU, PD, NROWPD)
    DIMENSION Y(NEQ), PD(NROWPD,NEQ)
    WHICH SUPPLIES DF/DY BY LOADING PD AS FOLLOWS.
    FOR A FULL JACOBIAN (MF = 21), LOAD PD(I,J) WITH DF(I)/DY(J), I.E. LOAD THE DIAGONAL LINES OF DF/DY INTO THE ROWS OF PD FROM THE TOP DOWN.
    IN EITHER CASE, ONLY NONZERO ELEMENTS NEED BE LOADED.

D. WRITE A MAIN PROGRAM WHICH CALLS SUBROUTINE LSODE ONCE FOR EACH POINT AT WHICH ANSWERS ARE DESIRED. THIS SHOULD ALSO PROVIDE FOR POSSIBLE USE OF LOGICAL UNIT 6 FOR OUTPUT OF ERROR MESSAGES.

BY LSODE. ON THE FIRST CALL TO LSODE, SUPPLY ARGUMENTS AS FOLLOWS...

F = NAME OF SUBROUTINE FOR RIGHT-HAND SIDE VECTOR F.
NEQ = NUMBER OF FIRST ORDER ODE'S.
Y = ARRAY OF INITIAL VALUES, OF LENGTH NEQ.
T = THE INITIAL VALUE OF THE INDEPENDENT VARIABLE.
TOUT = FIRST POINT WHERE OUTPUT IS DESIRED (.NE. T).
ITOL = 1 OR 2 ACCORDING AS ATOL (BELOW) IS A SCALAR OR ARRAY.
RTOL = RELATIVE TOLERANCE PARAMETER (SCALAR).
ATOL = ABSOLUTE TOLERANCE PARAMETER (SCALAR OR ARRAY).
EWT(I) = RTOL*ABS(Y(I)) + ATOL IF ITOL = 1, OR EWT(I) = RTOL*ABS(Y(I)) + ATOL(I) IF ITOL = 2.

THUS THE LOCAL ERROR TEST PASSES IF, IN EACH COMPONENT,
either the absolute error is less than atol (or atol(i)),
or the relative error is less than rtol.
use rtol = 0.0 for pure absolute error control, and
use atol = 0.0 (or atol(i) = 0.0) for pure relative error
control. caution.. actual (global) errors may exceed these
local tolerances, so choose them conservatively.

itask = 1 for normal computation of output values of y at t = tout.
istate = integer flag (input and output). set istate = 1.
iopt = 0 to indicate no optional inputs used.
rwork = real work array of length at least..
20 + 16*neq for mf = 10,
22 + 9*neq + neq**2 for mf = 21 or 22,
22 + 10*neq + (2*ml + mu)*neq for mf = 24 or 25.
lrw = declared length of rwork (in user-s dimension).

iwork = integer work array of length at least..
20 for mf = 10,
20 + neq for mf = 21, 22, 24, or 25.
if mf = 24 or 25, input in iwork(1), iwork(2) the lower
and upper half-bandwidths ml, mu.
liw = declared length of iwork (in user-s dimension).
jac = name of subroutine for jacobian matrix (mf = 21 or 24).
if used, this name must be declared external in calling
program. if not used, pass a dummy name.
mf = method flag. standard values are..
10 for nonstiff (adams) method, no jacobian used.
21 for stiff (bdf) method, user-supplied full jacobian.
22 for stiff method, internally generated full jacobian.
24 for stiff method, user-supplied banded jacobian.
25 for stiff method, internally generated banded jacobian.

note that the main program must declare arrays y, rwork, iwork,
and possibly atol.

the output from the first call (or any call) is..
y = array of computed values of y(t) vector.
t = corresponding value of independent variable (normally tout).

istate = 2 if lsode was successful, negative otherwise.
-1 means excess work done on this call (perhaps wrong mf).
-2 means excess accuracy requested (tolerances too small).
-3 means illegal input detected (see printed message).
-4 means repeated error test failures (check all inputs).
-5 means repeated convergence failures (perhaps bad jacobian
supplied or wrong choice of mf or tolerances).
-6 means error weight became zero during problem. (solution
component i vanished, and atol or atol(i) = 0.)

f. to continue the integration after a successful return, simply
reset tout and call lsode again. no other parameters need be reset.
THE SUBROUTINE INTEGRATES FROM T TO TOUT. ON RETURN THE PARAMETERS IN THE CALL LIST ARE INITIALIZED FOR CONTINUING THE INTEGRATION. THE USER HAS ONLY TO DEFINE A NEW VALUE TOUT AND CALL DE AGAIN.

DE CALLS TWO CODES, THE INTEGRATOR STEP AND THE INTERPOLATION ROUTINE INTRP. STEP USES A MODIFIED DIVIDED DIFFERENCE FORM OF THE ADAMS PECE FORMULAS AND LOCAL EXTRAPOLATION. IT ADJUSTS THE ORDER AND STEP SIZE TO CONTROL THE LOCAL ERROR. NORMALLY EACH CALL TO STEP ADVANCES THE SOLUTION ONE STEP IN THE DIRECTION OF TOUT. FOR REASONS OF EFFICIENCY DE INTEGRATES BEYOND TOUT INTERNALLY, THOUGH NEVER BEYOND T + 10*(TOUT-T), AND CALLS INTRP TO INTERPOLATE THE SOLUTION AT TOUT. AN OPTION IS PROVIDED TO STOP THE INTEGRATION AT TOUT BUT IT SHOULD BE USED ONLY IF IT IS IMPOSSIBLE TO CONTINUE THE INTEGRATION BEYOND TOUT.

THIS CODE IS COMPLETELY EXPLAINED AND DOCUMENTED IN THE TEXT COMPUTER SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS: THE INITIAL VALUE PROBLEM BY L. F. SHAMPINE AND M. K. GORDON.

THE PARAMETERS FOR DE ARE:

F -- SUBROUTINE F(T,Y,YP) TO EVALUATE DERIVATIVES YP(I)=DY(I)/DT
NEQN -- NUMBER OF EQUATIONS TO BE INTEGRATED
Y(*) -- SOLUTION VECTOR AT T
T -- INDEPENDENT VARIABLE
TOUT -- POINT AT WHICH SOLUTION IS DESIRED
RELERR,ABSERR -- RELATIVE AND ABSOLUTE ERROR TOLERANCES FOR LOCAL ERROR TEST. AT EACH STEP THE CODE REQUIRES
   DABS(LOCAL ERROR) .LE. DABS(Y)*RELERR + ABSERR
   FOR EACH COMPONENT OF THE LOCAL ERROR AND SOLUTION VECTORS
IFLAG -- INDICATES STATUS OF INTEGRATION
   FOR FIRST CALL TO DE --
   THE USER MUST PROVIDE STORAGE IN HIS CALLING PROGRAM FOR THE ARRAY IN THE CALL LIST Y(NEQN), DECLARE F IN AN EXTERNAL STATEMENT, SUPPLY THE SUBROUTINE F(T,Y,YP) TO EVALUATE
   DY(I)/DT = YP(I) = F(T,Y(1),Y(2),...,Y(NEQN))
   AND INITIALIZE THE PARAMETERS:
   NEQN -- NUMBER OF EQUATIONS TO BE INTEGRATED
   Y(*) -- VECTOR OF INITIAL CONDITIONS
   T -- STARTING POINT OF INTEGRATION
   TOUT -- POINT AT WHICH SOLUTION IS DESIRED
   RELERR,ABSERR -- RELATIVE AND ABSOLUTE LOCAL ERROR TOLERANCES
   IFLAG -- +1,-1. INDICATOR TO INITIALIZE THE CODE. NORMAL INPUT IS +1. THE USER SHOULD SET IFLAG=-1 ONLY IF IT IS IMPOSSIBLE TO CONTINUE THE INTEGRATION BEYOND TOUT.
   ALL PARAMETERS EXCEPT F, NEQN AND TOUT MAY BE ALTERED BY THE CODE ON OUTPUT SO MUST BE VARIABLES IN THE CALLING PROGRAM.

OUTPUT FROM DE --

   NEQN -- UNCHANGED
   Y(*) -- SOLUTION AT T
   T -- LAST POINT REACHED IN INTEGRATION. NORMAL RETURN HAS T = TOUT.
   TOUT -- UNCHANGED
   RELERR,ABSERR -- NORMAL RETURN HAS TOLERANCES UNCHANGED. IFLAG=3 SIGNALS TOLERANCES INCREASED

7
IFLAG = 2 -- NORMAL RETURN. INTEGRATION REACHED TOUT
= 3 -- INTEGRATION DID NOT REACH TOUT BECAUSE ERROR
TOLERANCES TOO SMALL. RELERR, ABSERR INCREASED
APPROPRIATELY FOR CONTINUING
= 4 -- INTEGRATION DID NOT REACH TOUT BECAUSE MORE THAN
MAXIMUM STEPS NEEDED
= 5 -- INTEGRATION DID NOT REACH TOUT BECAUSE EQUATIONS
APPEAR TO BE STIFF
= 6 -- INVALID INPUT PARAMETERS (FATAL ERROR)
THE VALUE OF IFLAG IS RETURNED NEGATIVE WHEN THE INPUT
VALUE IS NEGATIVE AND THE INTEGRATION DOES NOT REACH TOUT,
I.E., -3, -4, -5.

SUBSEQUENT CALLS TO DE --
SUBROUTINE DE RETURNS WITH ALL INFORMATION NEEDED TO CONTINUE
THE INTEGRATION. IF THE INTEGRATION REACHED TOUT, THE USER NEED
ONLY DEFINE A NEW TOUT AND CALL AGAIN. IF THE INTEGRATION DID NOT
REACH TOUT AND THE USER WANTS TO CONTINUE, HE JUST CALLS AGAIN.
The output value of IFLAG IS THE APPROPRIATE INPUT VALUE FOR
SUBSEQUENT CALLS. THE ONLY SITUATION IN WHICH IT SHOULD BE ALTERED
IS TO STOP THE INTEGRATION INTERNALLY AT THE NEW TOUT, I.E.,
CHANGE OUTPUT IFLAG=2 TO INPUT IFLAG=-2. ERROR TOLERANCES MAY
BE CHANGED BY THE USER BEFORE CONTINUING. ALL OTHER PARAMETERS MUST
REMAIN UNCHANGED.