1. Grab a copy of the GSL source tarball (version 2.6). For convenience, you can copy a pre-downloaded version from:

   `/work/00161/karl/stampede2/public/gsl-2.6.tar.gz`

   Untar this distribution and prepare for compiling 4 different configurations of GSL.

   1. GNU gcc with no optimization
   2. GNU gcc with -O3 optimization
   3. Intel icc with no optimization
   4. Intel icc with -O3 optimization

   We recommend taking advantage of VPATH builds to support this effort. Consequently, consider making 4 directories corresponding to the configurations above. To complete this exercise, you need to fill in the Table below documenting the time it takes to compile each case serially ("make") versus the time it takes to compile in parallel ("make -j 48"). Note that since you are compiling on a compute node, using all the processors available is reasonable.

Configuration	Compilation times		Regression tests	
	Serial (secs)	Parallel (secs)	Passes	Failures
gcc/noOpt	143.157	33.356	58	0
gcc/O3	242.400	57.072	58	0
icc/noOpt	198.798	38.087	57	1
icc/O3	331.804	70.882	54	4

   → *VPATH builds were used for this purpose. Also, we then use the following sequence of commands to find the number of regression tests passed and failed.*

```
$ make check -k -j 48 > check.log 2>&1
$ grep "# PASS:" check.log | awk '{sum = sum + $3;}END{print sum;}'
$ grep "# FAIL:" check.log | awk '{sum = sum + $3;}END{print sum;}'
```

   In addition to tracking the compile times for each case, run the regression test suite provided by GSL using the "make check" target. For each case, document how many regression tests pass or fail in your table. Note that make will normally stop if it encounters an error. However, you can ask it to continue (so that you can discover all the regression failures, if you encounter one) using "make -k". In addition to filling in values for the table above, document the exact configure line you used to setup each build here:

   - gcc/noOpt: `mkdir -p gcc/noOpt; cd gcc/noOpt; ../../configure CC=gcc CFLAGS="-g -O0"; time make -k -j 48`

- gcc/O3: `mkdir -p gcc/O3; cd gcc/O3; ../../configure CC=gcc CFLAGS="-g -O3"; time make -k -j 48`
- icc/noOpt: `mkdir -p icc/noOpt; cd icc/noOpt; ../../configure CC=icc CFLAGS="-g -O0"; time make -k -j 48`
- icc/O3: `mkdir -p icc/O3; cd icc/O3; ../../configure CC=icc CFLAGS="-g -O3"; time make -k -j 48`

Based on the results you obtained, answer the following additional questions:

- Which is faster to build - optimized Intel or GNU? How much faster is it?

  ⟶ Optimized gcc is faster to build. It takes about 89.404 seconds lesser for serial build and 13.810 seconds lesser for parallel build. If we speak about ratios, serial build for gcc is 1.39 times faster than serial build for icc, while the parallel build for gcc is 1.14 times faster than serial build for icc.

- How much faster was a parallel build of non-optimized GNU versus its serial counterpart?

  ⟶ It is about 4.3 times faster (ratios), taking about 109.8 seconds lesser (time difference).

- If you encountered any regression/build failures, document specifically which directories the failures occurred in (and the associated compiler configuration).

  ⟶ We only have failures in icc builds. All regression tests passed for GCC builds. We use the following sequence of commands to get these directory names.

```
$ make check -k -j 48 > check.log 2>&1
$ grep -C 1 "Error" check.log
```

  Basically, both STDOUT and STDERR of make check are written to `check.log`. We then search for lines around the word "Error" in `check.log`. `grep -C 1` means search with context i.e. return one line above, the line where the word is found and one line below.

  - **icc/noOpt** - Note that the number of passes and fails should add to 58, but that is not the case with icc/noOpt. This is because one of the tests itself is not built (for directory specified below). Hence only 57 tests were run and all of them passed, but we consider the one that did not even build as failed. The above command gives us the following directory names.
    1. /home1/07665/shrey911/cse380-2020-student-Shreyas911/hw03/gsl-2.6/icc/noOpt/multifit
  - **icc/O3** - The above command gives us the following directory names.
    1. /home1/07665/shrey911/cse380-2020-student-Shreyas911/hw03/gsl-2.6/icc/O3/linalg
    2. /home1/07665/shrey911/cse380-2020-student-Shreyas911/hw03/gsl-2.6/icc/O3/specfunc
    3. /home1/07665/shrey911/cse380-2020-student-Shreyas911/hw03/gsl-2.6/icc/O3/multifit

4. /home1/07665/shrey911/cse380-2020-student-Shreyas911/hw03/gsl-2.6/icc/O3/ode-initval2

2. Create a simple Makefile to build an executable named "tardis". Using the example C src code files in the shared class repository at: `examples/buildsystems/standalone`

   Your Makefile should include the following features:

   - A "clean" target which removes all associated object files and executable
   - Convenience variables to provide system flexibility:
     - Variable that defines the desired executable name
     - Variable that defines the desired C compiler
     - Variable that defines the location of the desired GSL top-level path to link against based on an external environment variable (e.g. *TACC_GSL_DIR*)
     - Variable that defines the compiler flags to use

   Submit your working Makefile along with the src code files into a `hw03/make` subdirectory in your repository.

   The Makefile is shown here. Note that the GSL module has to be loaded (for example `module load gsl/2.6`) before the `make` command can be used. `GSL` flag points to the GSL directory, `LDFLAGS` helps link to the GSL libraries, `LDLIBS` specifies the library names. `CINC` helps the compiler to resolve a non-standard path and it points to the `include` subdirectory in GSL which contains the header files. A "clean" target is also included.

```
EXEC := tardis
SRC  := $(wildcard *.c)
OBJ  := $(patsubst %.c, %.o, $(SRC))

# Options

CC      := gcc
CFLAGS  := -O0 -g
GSL     := $(TACC_GSL_DIR)
LDFLAGS := -L$(GSL)/lib
LDLIBS  := -lgsl -lgslcblas
CINC    := -I$(GSL)/include

# Rules

$(EXEC): $(OBJ)
        $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
        $(CC) $(CFLAGS) $(CINC) -c $<
mytools.o main.o: mytools.h

# Useful phony targets
```

```
.PHONY: clean

clean:
	$(RM) $(EXEC) $(OBJ)
```

3. The error function is defined by the equation:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Write a code in (C/C++ or Fortran) that performs numerical integration to calculate erf(1) using two methods with constant mesh spacing: (1) trapezoid rule and (2) Simpson's rule. Required items:

- provide a Makefile to build your code on Stampede2
- the resulting executable should be named `integrate`
- the choice of integration rule should be controllable via command line argument
- the # of discretization points (N) should be controllable via command line argument
- compare the resulting integral evaluation error against a tabulated value of: $\text{erf}_{\text{tab}}(1) = 0.84270079295$

Once completed, perform a verification exercise by providing asymptotic convergence analysis for each integration method. Include at least $N = 10$ points for each method and construct a job script that runs the convergence tests and saves the results (include job script and results file(s) in git repo). Finally, use your favorite plotting package to create two well-labeled plots that compare asymptotic convergence rates from your code to expected rates for each method (hint: your plots should be on a log/log scale and if all is groovy with the code, the slope should roughly match the expected convergence rate).

`job.sh` executes the `make` command, runs the executable `integrate` several times with different arguments, and then plots the log files on a log-log scale by calling `plot.script` with `gnuplot`.

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
##SBATCH -o job_cpp_results.log
##SBATCH -e error.%j.out
#SBATCH -J INTEGRATION
#SBATCH -p skx-normal
#SBATCH -A cse38018
#SBATCH -t 00:01:00

TIMEFORMAT=%R ## Change time format to give only real time value, got this from↩
 stack overflow
```

```
## Create empty log files
>results_trapezoidal.log
>results_simpson.log

## Create headings in log files
echo "#N integral_computed e_rel time" >> results_trapezoidal.log
echo "#N integral_computed e_rel time" >> results_simpson.log

## Output stores stdout of integrate and stderr of time

make integrate

{ time ./integrate 1 6  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 10  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 50  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 100  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 200  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 500  >> results_trapezoidal.log;} 2>> results_trapezoidal.↩
log
{ time ./integrate 1 1000  >> results_trapezoidal.log;} 2>> results_trapezoidal↩
.log
{ time ./integrate 1 2000  >> results_trapezoidal.log;} 2>> results_trapezoidal↩
.log
{ time ./integrate 1 5000  >> results_trapezoidal.log;} 2>> results_trapezoidal↩
.log
{ time ./integrate 1 10000  >> results_trapezoidal.log;} 2>> ↩
results_trapezoidal.log

## Simpson needs even number of intervals, i.e. odd number of points
{ time ./integrate 2 5  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 7  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 9  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 11  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 21  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 31  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 41  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 51  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 71  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 91  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 111  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 131  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 161  >> results_simpson.log;} 2>> results_simpson.log
{ time ./integrate 2 191  >> results_simpson.log;} 2>> results_simpson.log

## Plot results using a script
gnuplot plot.script

## Clean the make build
make clean
```

`Makefile` creates the executable and also can clean up the object files and executable.

```
CC := g++
CPPFLAGS := -O3 -g
EXEC := integrate
SRC  := $(wildcard *.cpp)
OBJ  := $(patsubst %.cpp,%.o,$(SRC))

$(EXEC): $(OBJ)
        $(CC) -o $@ $^
%.o: %.cpp
        $(CC) $(CPPFLAGS) -c $< -o $@

integrate.o trapezoidal.o: trapezoidal.h
integrate.o simpson.o: simpson.h

# Useful phony targets
clean:
        $(RM) $(OBJ) $(EXEC)
```

`func.cpp` defines the function you wish to integrate on the interval $[0, x]$ where x is given as an argument.

```
#include <cmath>
// Define the function you want to integrate here.
double func(double t){
        return (2.0/sqrt(atan(1.0)*4.0) * exp(-pow(t,2.0)));
}
```

`simpson.cpp` integrates using the simpson method.

```
#include <cmath>
#include "func.h"

// Integrate function defined in func.h by simpson method. x is limit of ↩
integration i.e. the interval [0,x] and n is the number of points used. Simpson↩
 method requires even number of intervals, odd number of points.
double integral_simpson(double x, int n){

        double integral = 0;
        double dt = x/(n-1);

    integral += dt/3 * func(0);
        for(int i = 1; i<(n-1);i++){
        if(i%2 == 1){
                    integral += dt/3 * 4 * func(i*dt);
        }
        else{
            integral += dt/3 * 2 * func(i*dt);
        }
        }
    integral += dt/3 * func(1);
        return integral;
}
```

`trapezoidal.cpp` integrates using the trapezoidal method.

```
#include <cmath>
#include "func.h"

// Integrate function defined in func.h by trapezoidal method. x is limit of integra
// tion i.e. the interval [0,x] and n is the number of points used.
double integral_trapezoidal(double x, int n){

        double integral = 0;
        double dt = x/(n-1);

        for(int i = 0; i<(n-1);i++){
                integral += dt*(func(i*dt)+func((i+1)*dt))/2;
        }

        return integral;
}
```

`integrate.cpp` is the driver routine, actually taking in command line inputs and integrating.

```
#include <iostream>
#include <cmath>
#include "trapezoidal.h"
#include "simpson.h"
#include <iomanip>
#define erf_tab 0.84270079295

using namespace std;

int main(int argc, char *argv[]) {
    // Input verification
    // atoi returns integer part of input and 0 if not a numeric input
    if (argc != 3 || atoi(argv[2]) <= 0 || (atoi(argv[1]) != 1 & atoi(argv[1]) != 2)) {
        cerr << "ERROR: First input should be 1 or 2 for Trapezoidal and Simpson's rule respectively. The second input should be a positive integer for the number of points. Simpson's rule works properly only for odd number of points and even number of intervals." << endl;
    return 1;
    }

    int N = atoi(argv[2]);

    if (atoi(argv[1]) == 1){

        double integral = integral_trapezoidal(1, N);
        double erel = abs((integral/erf_tab)-1.0);

        cout << N << " " << setprecision(10) << integral << " " << setprecision(10) << erel << " ";
```

```
    }
    else{

        double integral = integral_simpson(1, N);
        double erel = abs((integral/erf_tab)-1.0);

        cout << N << " " << setprecision(10) << integral << " " << setprecision↩
        (10) << erel << " ";
        }
}
```

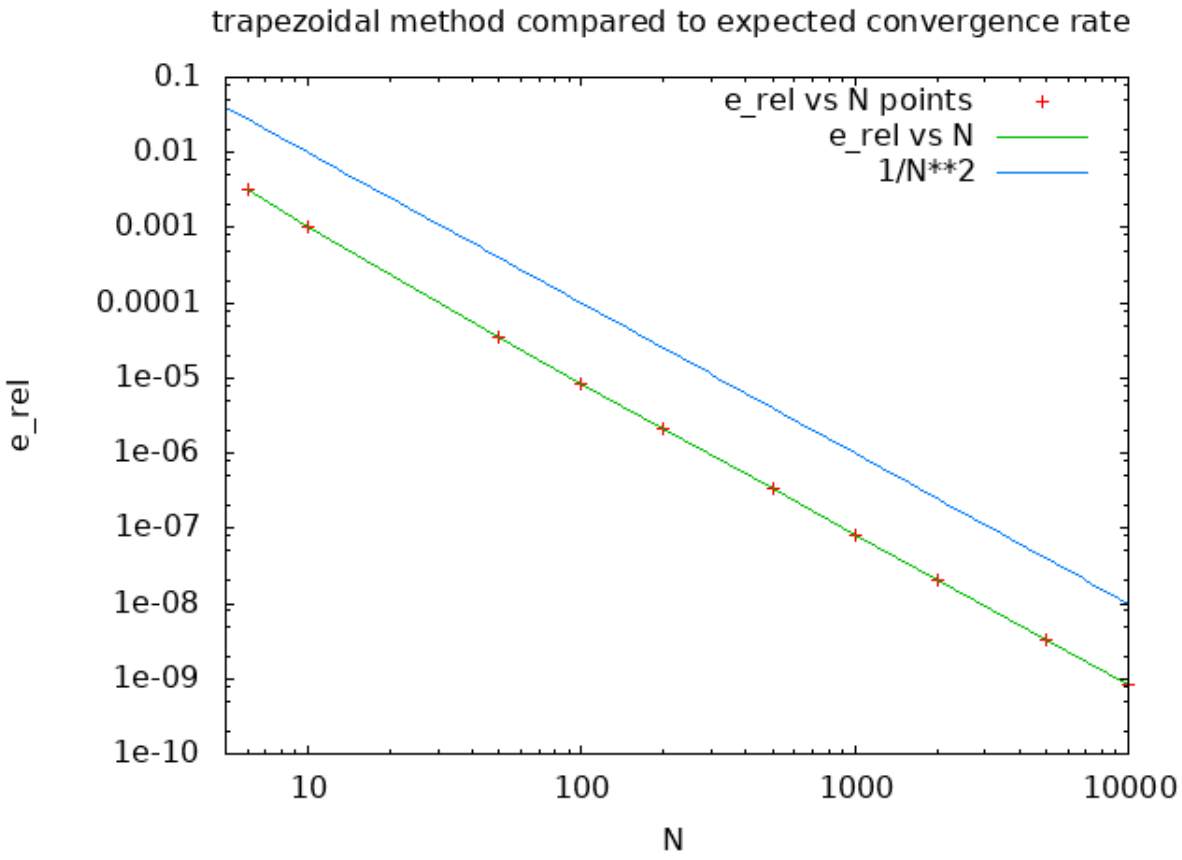
Results and plots -

Note - *N is the number of points, N-1 is the number of intervals. Simpson's rule requires even number of intervals.*

`results_trapezoidal.log`

```
#N integral_computed e_rel time
6 0.8399297272 0.003288315068 0.003
10 0.841846311 0.001013980222 0.003
50 0.8426719776 3.419399138e-05 0.003
100 0.842693734 8.37658558e-06 0.003
200 0.8426990459 2.073147528e-06 0.003
500 0.8427005151 3.297124427e-07 0.003
1000 0.8427007236 8.226341897e-08 0.003
2000 0.8427007756 2.054553705e-08 0.003
5000 0.8427007902 3.285594352e-09 0.003
10000 0.8427007923 8.214878777e-10 0.003
```

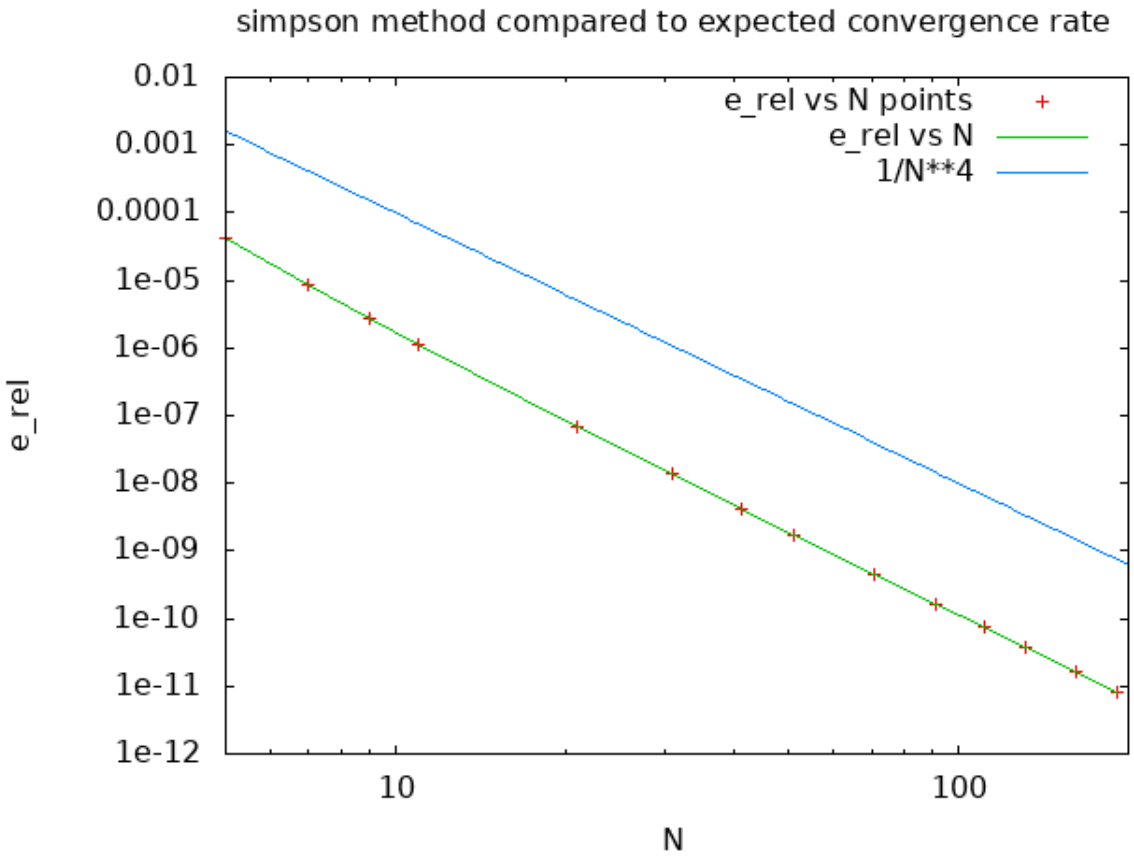
We plot the relative error ($\mathtt{e}_{rel}$) (wrt the tabulated vaulue $\mathrm{erf}_{tab}(1)$) for the trapezoidal case on a log-log scale. The expected value of slope is -2. We can that our results hold up pretty well since the slopes are nearly identical apart from the start where the number of points was too low.



results_simpson.log

```
#N integral_computed e_rel time
5 0.8427360514 4.183980797e-05 0.003
7 0.8427078551 8.38038889e-06 0.003
9 0.8427030358 2.661556717e-06 0.003
11 0.8427017131 1.091879236e-06 0.003
21 0.8427008506 6.837384148e-08 0.003
31 0.8427008043 1.351024026e-08 0.003
41 0.8427007966 4.274993248e-09 0.003
51 0.8427007944 1.750931844e-09 0.003
71 0.8427007933 4.555531508e-10 0.002
91 0.8427007931 1.664981486e-10 0.003
111 0.842700793 7.442646499e-11 0.003
131 0.842700793 3.798739101e-11 0.003
161 0.842700793 1.636424329e-11 0.003
191 0.842700793 8.061551426e-12 0.003
```

We plot the relative error ($\mathtt{e}_{rel}$) (wrt the tabulated value $\mathrm{erf}_{tab}(1)$) for the simpson case on a log-log scale. The expected value of slope is -4. We can see that our results hold up pretty well since the slopes are nearly identical. Simpson's rule converges must faster than the trapezoidal method.



The plot script `plot.script` is given below -

```
set terminal png

set title "trapezoidal method compared to expected convergence rate"
set xlabel "N"
set ylabel "e_rel"
set logscale xy
set xrange [5:10000]
set output 'trapezoidal_plot.png'; plot 'results_trapezoidal.log' using 1:3 ↩
    title 'e_rel vs N points' with points, 'results_trapezoidal.log' using 1:3 ↩
    title 'e_rel vs N' with lines, 1/x**2 title '1/N**2'with lines

set title "simpson method compared to expected convergence rate"
set xlabel "N"
set ylabel "e_rel"
set logscale xy
set xrange [5:200]
set output 'simpson_plot.png'; plot 'results_simpson.log' using 1:3 title '↩
    e_rel vs N points' with points, 'results_simpson.log' using 1:3 title 'e_rel↩
     vs N' with lines, 1/x**4 title '1/N**4' with lines
```