Software Testing Walkthrough

David M. Rogers
Oak Ridge National Laboratory

Software Productivity and Sustainability track, ATPESC 2021

License, Citation and Acknowledgements

License and Citation

- **The requested citation the overall tutorial is: David E. Bernholdt, Anshu Dubey, Rinku K. Gupta, and David M. Rogers, Software Productivity and Sustainability track, in Argonne Training Program on Extreme-Scale Computing (ATPESC), online, 2021. DOI: 10.6084/m9.figshare.15130590**
- Individual modules may be cited as *Speaker, Module Title*, in Better Scientific Software tutorial…

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This work was performed in part at the Lawrence Livermore National Laboratory, which is managed by Lawrence Livermore National Security, LLC for the U.S. Department of Energy under Contract No. DE-AC52-07NA27344.
- This work was performed in part at the Los Alamos National Laboratory, which is managed by Triad National Security, LLC for the U.S. Department of Energy under Contract No.89233218CNA000001
- This work was performed in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Hello Numerical World Example (heat equation)

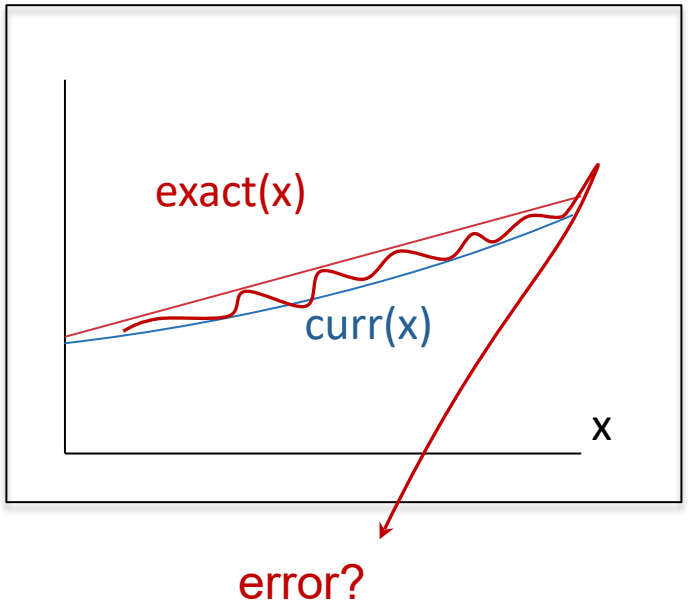
github.com/bssw-tutorials/hello-numerical-world

```
$ wc *.C
   125    494   4161 args.C     # parse arguments
   220    718   5667 heat.C     # main() – stores all vars
   151    498   3888 utils.C    # l2_norm, write, copy, init
    26    119    820 ftcs.C     # standard, centered stencil
    27    123    833 upwind15.C # alternate integration schemes
    94    344   2134 crankn.C
    43    190   1299 exact.C    # comparison solution
```

- Lots of setup code – prepares problem for kernel calls
- Isolated, swappable kernel calls
  - Imagine adding kernels to larger, multi-physics application.
- How can we support testing all these kernel configurations?

What to Test?

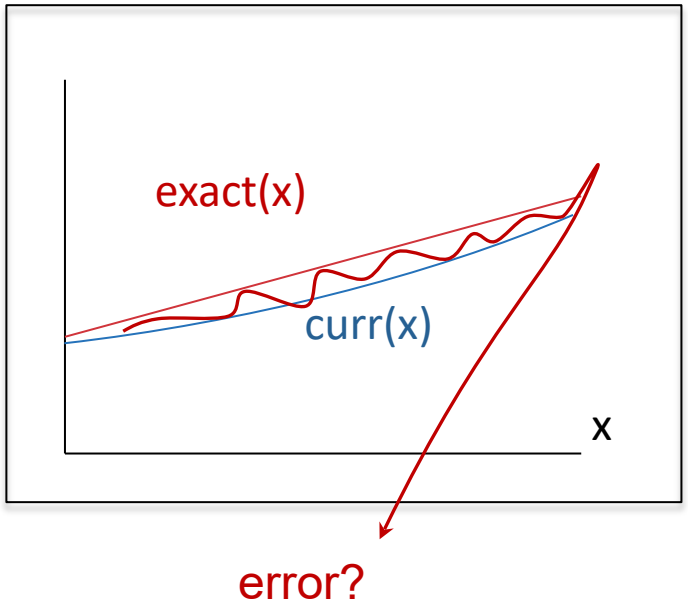
- Types of Tests:
  - code coverage – ensure options parse, bad cases detected, utilities fuction, etc.
  - steady-state (should be straight line)
    - external script can test file write() as well
  - solution time-dependence vs. reference
    - $(d/dx)^2 \sin(ax) = -a^2 \sin(ax)$
  - integration between codes?
  - test compile/run in multiple precisions?
    - combinatorial problems – listing tests in for() or matrix...



Running Tests via makefile

```
$ make check_all
      c++ -c  -Iinclude -DHEAT_VERSION_MAJOR=0 -
      DHEAT_VERSION_MINOR=5  args.C -o args.o
      c++ -o heat heat.o utils.o args.o exact.o ftcs.o upwind15.o
      crankn.o  -lm
      ./heat runame=check outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"
          runame="check"
       ...
      Stopped after 001490 iterations for threshold 2.46636e-15
      cat check/check_soln_final.curve
      # Temperature
      ...
      ./check.sh check/check_soln_final.curve 0

 make completes: commands succeeded
```



steady-state test
(should be straight line)

TODO – try out new build tools and add tests to them

- Replace makefile with *CMakeLists.txt*
  - replaces rules with *targets* (tied to a list of source files)
  - targets have *attributes*
    - target_link_libraries (e.g. MPI::MPI_CXX)
    - target_include_directories (many already inferred from link libraries)
    - target_compile_features (e.g. cxx_std11)
  - provides *find_package* command
  - targets can be installed
- Replace "make check_all" with *ctest*
  - reduces glue code
  - different interface for adding tests
- End Result: contrast two methods of testing.

existing makefile

makefile

```
...

# Implicit rule for object files
%.o : %.C
	$(CXX) -c $(CXXFLAGS) $(CPPFLAGS) $< -o $@

# Linking the final heat app
heat: $(OBJ)
	$(CXX) -o heat $(OBJ) $(LDFLAGS) –lm
```

Standard makefile – user selects compile flags.
- but flags and features are compiler and system-specific
- enter automake and cmake -> generate makefiles

Conversion to cmake (entire file)

https://cmake.org/cmake/help/latest/guide/tutorial/index.html

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(heat VERSION 0.5 LANGUAGES CXX)
# can change boolean variable with "-DCMAKE_BUILD_TESTS=OFF"
option(BUILD_TESTS "Build the tests accompanying this program." ON)
# pass cmake options (e.g. version) into a header
configure_file(include/version.H.in include/version.H)
add_executable(heat args.C crankn.C ...) # list sources
# feature – lets cmake adjust flags for compiler --std=c++11 vs –c11
target_compile_features(heat cxx_std_11)
# include directories for all files in this target:
target_include_directories(heat ${PROJECT_BINARY_DIR}/include)
if(BUILD_TESTS) add_subdirectory(tests) endif() # subdir for tests
install(TARGETS heat DESTINATION bin) # "make install" target
```

existing tests

makefile include (tests.mk)

```
...
check_crankn/check_crankn_soln_final.curve:
        ./heat alg=crankn runame=check_crankn outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"
check_crankn: heat check_crankn/check_crankn_soln_final.curve
        cat check_crankn/check_crankn_soln_final.curve
        ./check.sh check_crankn/check_crankn_soln_final.curve

check_upwind15/check_upwind15_soln_final.curve:
        ./heat alg=upwind15 ...
```

Create a test driver to:

1. run executable
2. check result
3. clean up outputs

Addition to CMakeLists.txt

cmake.org/cmake/help/latest/command/add_test.html

tests/CMakeLists.txt

```
enable_testing()

add_test(NAME heat_help
      COMMAND $<TARGET_FILE:heat> help)

add_test(NAME crankn
      COMMAND testDriver.sh $<TARGET_FILE:heat> crankn)

# functions/for/if/adding tests
```

Lots of potential for programmatically creating tests!

Try and keep it simple – complex cmake code is bad form.

Bonus: swap out test driver (perl -> awk)

tests/testDriver.sh

```
#!/bin/bash
set –e              # exit immediately on error
errbnd=1e-7
alg="$2"
$1 alg=$alg runame=check_$alg outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"

# absolute error check (deviation from straight line)
err=$(awk 'function abs(x){return ((x < 0.0) ? -x : x)}; BEGIN {err=1e10;} ! /#/ {err1=abs($2-$1); if(err1
< err) err = err1;} END {print err;}' check_$alg/check_${alg}_soln_final.curve)

echo "Error = $err"
rm -fr check_$alg    # delete directory to test is re-runnable

awk "BEGIN {exit($err >= $errbnd);}"  # final return code
```

Running

```
cmake ..
make –j
cd tests && ctest
```

```
Test project hello-numerical-world/build/tests
    Start 1: ftcs
1/3 Test #1: ftcs ..........................   Passed    0.02 sec
    Start 2: crankn
2/3 Test #2: crankn ........................   Passed    0.02 sec
    Start 3: upwind15
3/3 Test #3: upwind15 ......................   Passed    0.03 sec


100% tests passed, 0 tests failed out of 3


Total Test time (real) =   0.08 sec
```

Going Further

- Reproduce these testing strategies on another repository
  - github.com/frobnitzem/simple-heateq (same problem, different design)
- Brainstorm some simple tests you could add to your own project
  - checks you've run manually
  - difficult-to-setup and reproduce cases that could be automated
- Add some "blank tests" to your project
  - reduces the barrier to increased testing
  - What would make reporting on your build / run status better/simpler/more accessible?

Conclusion – C, kernels, makefiles, CMakeLists, coverage, etc.

- Start your projects small, stay organized
  - makefiles provide fast development path
  - add tests before complexity grows!
  - simple to do with a "make check" target
- cmake (like autoconf) helps make portable builds
  - find_package
  - programmatic build options
  - set target properties -> cmake looks up compiler flags for you
- good testing strategies exist for both
  - directly run the executable with all options
  - create shell-script "test driver"
  - build stand-alone executables loading a library