Software Testing Made Easy

Common Sense Tips and Suggestions

Robert McLay and Doug James

April 8, 2014

Overview

A Bit of Context

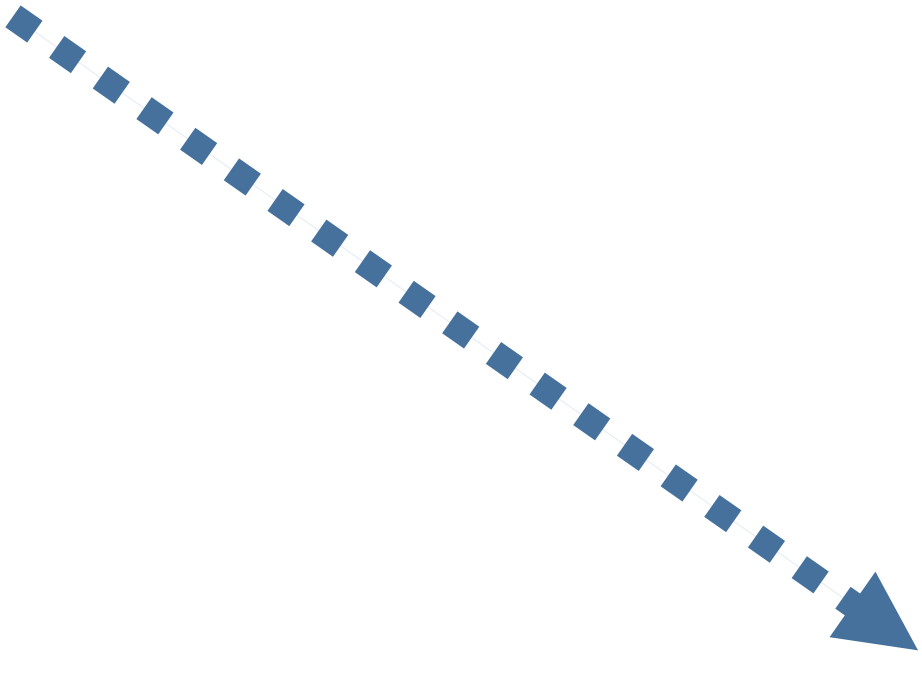
Tips and Suggestions

References and Final Thoughts

You're busy. You might wonder if you have time to do this.
My answer: testing will save you gobs of time. You don't have time not to do this!

The Vision

Real World



Conclusions

TACC

The Path

Real World

Model

PDE

Algorithm

Numerical method

Code

Computer program

Experiment

Batch job

Results

Data Files,
Visualization

Conclusions

TACC

The Prime Directive

Real World

Model

Algorithm

Code

Experiment

Results

Conclusions

Get the right answer.

~~confidence!~~

"…testing means having faith that our answers mean something."

TACC

Validation

Real World

Model

Algorithm

Code

Experiment

Results

Conclusions

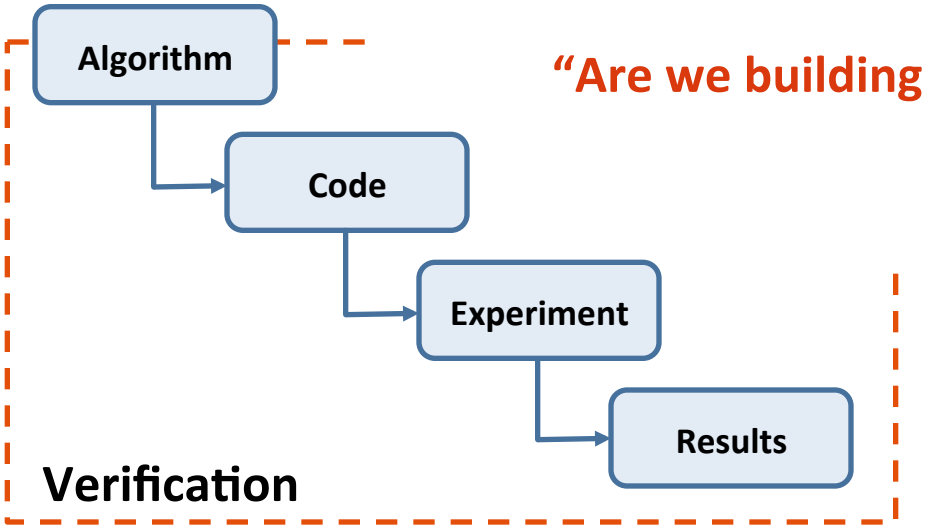
Appropriateness of design:
model, algorithm, specs, requirements

"Are we building the right thing?"

TACC

Verification

Correctness of Implementation:
compliance with specs/requirements



"Are we building the thing right?"

Correctness: The Prime Directive

- **Testing**: Is the answer right?
  - Focus is on…
    - correctness of the implementation
    - computation rather than the choice of model
    - verification rather than validation
- **Regression Testing**: Is the answer still right?
  - Making sure your changes don't break anything
  - On-going – nearly continuous
  - Arguably the most important thing you can do

Constructing Test Problems

- Build a small suite of simple test problems
  - Easy to run, easy to understand
  - Probe and challenge your algorithm's properties
    - e.g. $O(h^2)$, exact for quadratics, etc.
- Build problems with known solutions
  - Exploit the literature
  - Solve simple problems by hand
  - Use another code/language/algorithm (R, Octave, etc)
  - Or use the best trick of all…

Working Backwards

- Start by choosing your favorite exact solution, then reverse engineer the test problem.
- For example…

  …if testing a solver for matrix equation Ax=b, start with x, then generate b.

  …if solving an ODE or PDE, start with a solution function, then compute the RHS and auxiliary conditions.

Measuring Error: General

- When comparing computed and 'exact' solutions, do not expect perfect match
  - Roundoff error – order of ops, optimizations, etc
  - Truncation or discretization error – the result of approximating a continuous process
- Need to determine whether computed and 'exact' are 'close enough'

Measuring Error: Scalars

- Absolute Error (signed or unsigned)

```
error = computed – exact;
```

- Relative error (signed or unsigned)

```
relError = error / exact;
```

- Test either/both against appropriate tolerances

```
if ( abs(error) < TOLERANCE )…
if ( abs(relError) < REL_TOLERANCE )…
```

Measuring Error: Vectors (pseudocode)

```
initialize L1Error, L2Error, LinfError to 0.0
initialize L1SolnSize et al to 0.0 (here we show L1)

for each i
        localError = abs( computed[i] - exact[i] )
        L1Error    = L1Error + localError
        L2Error    = L2Error + localError**2
        LinfError  = max( LinfError, localError )
        L1SolnSize = L1SolnSize + abs( exact[i] )
end for each i

L2Error = sqrt( L2Error )  (or compare to TOL**2)
relL1Error = L1Error / L1SolnSize (as one example)
```

Measuring Error: Acceptance Criteria

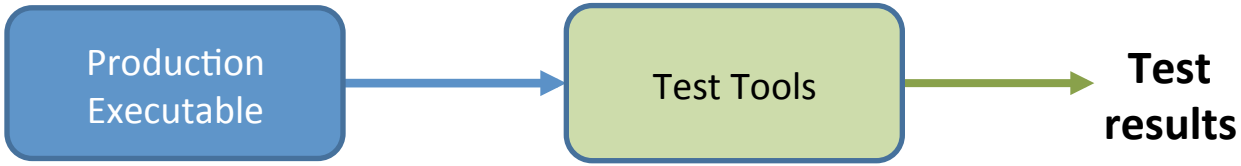
- Choosing appropriate tolerances can be tricky
  - 100 vs 1,000,000 components? time steps?
  - global (cumulative) vs local (current time step)?
- No magic bullet, but theorems and heuristics often suggest plausible approaches, e.g. …

```
relTolerance = coef * totalSteps * deltaT**4
if (  abs( relError ) < relTolerance  )…
```

Executing Tests

Option 1: Testing as post-processing

…Separate the execution from the testing
…Run your existing executable 'as is'
…Use other tools (scripts, binaries, spreadsheets) to do post-process analysis

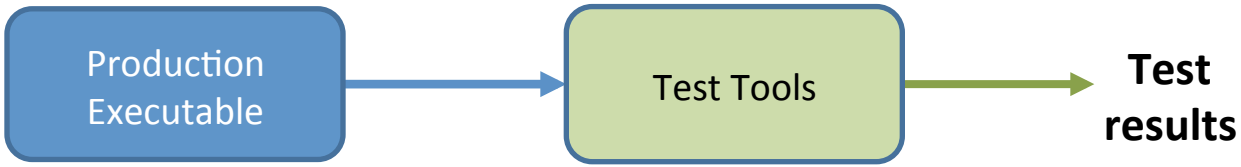


Executing Tests

Option 1: Testing as post-processing

…Separate the execution from the testing
…Run your existing executable 'as is'
…Use other tools (scripts, binaries, spreadsheets) to do post-process analysis

Verdict: A simple way to get started (move on as your needs evolve)

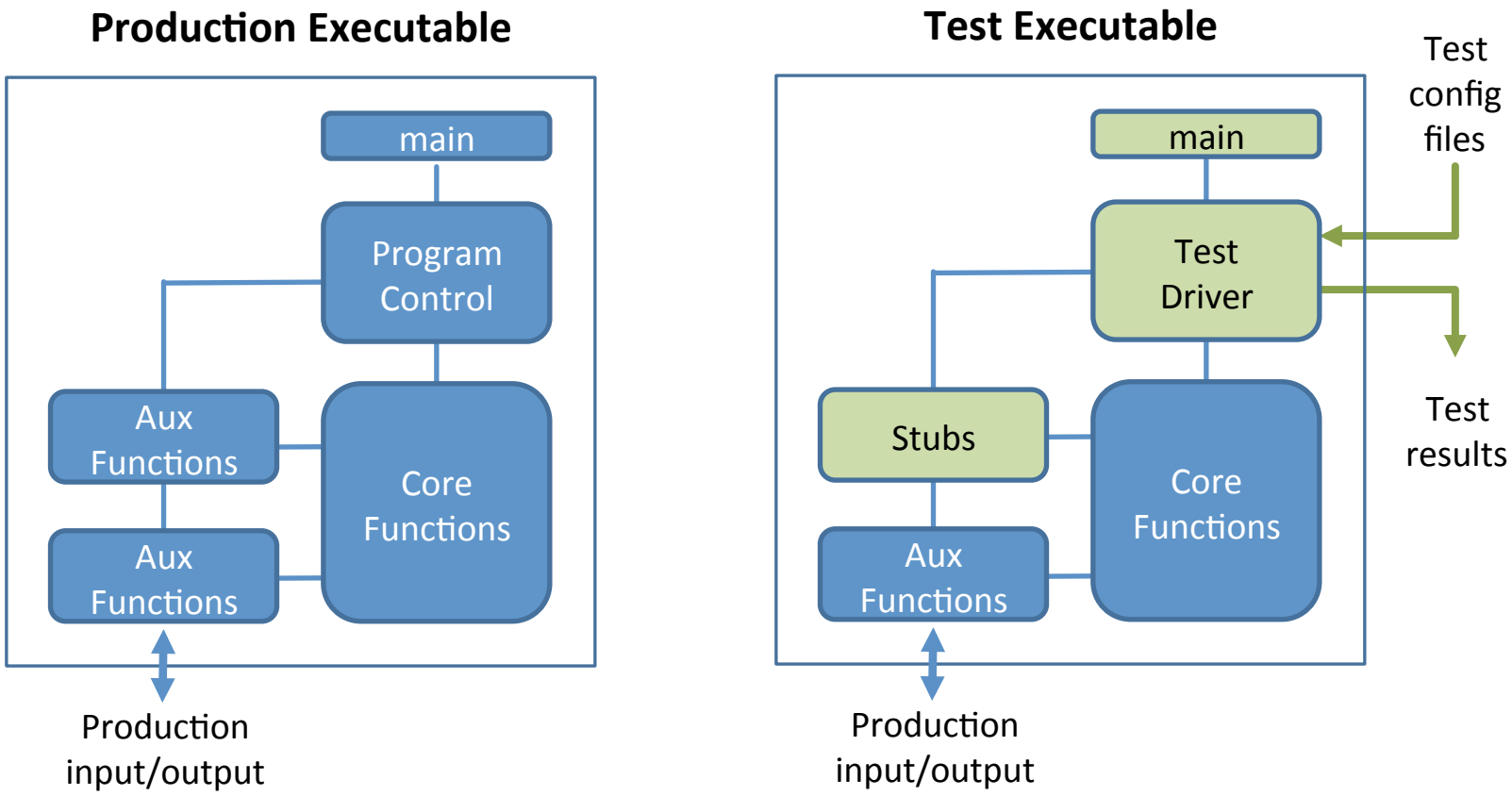


Executing Tests

Option 2: Build a special test executable

Replace production functionality with special test modules

May require stubs that simulate production activities



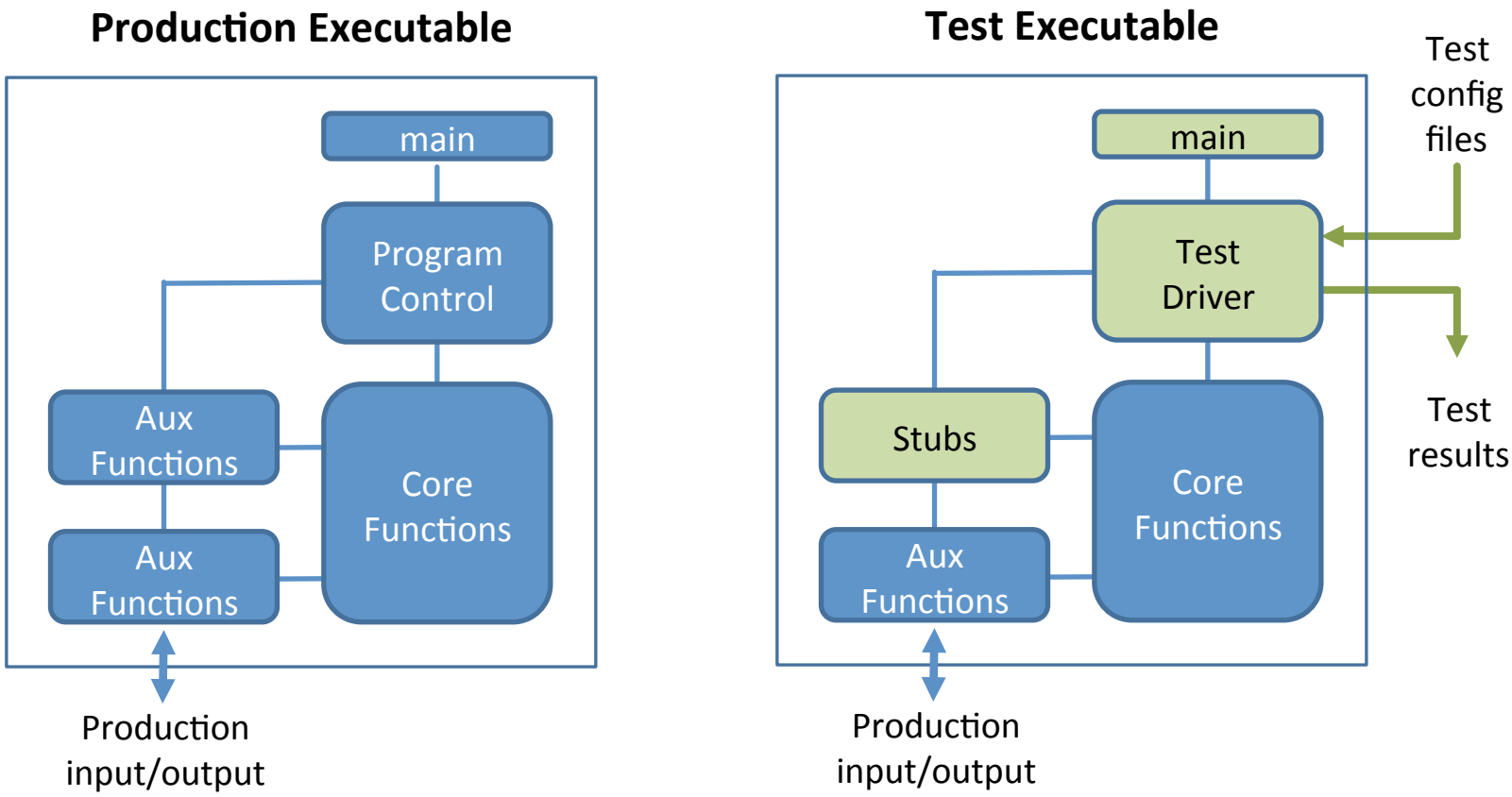
Executing Tests

Option 2: Build a special test executable

Replace production functionality with special test modules

May require stubs that simulate production activities

Verdict: Can be clunky – hard to maintain – especially for small teams

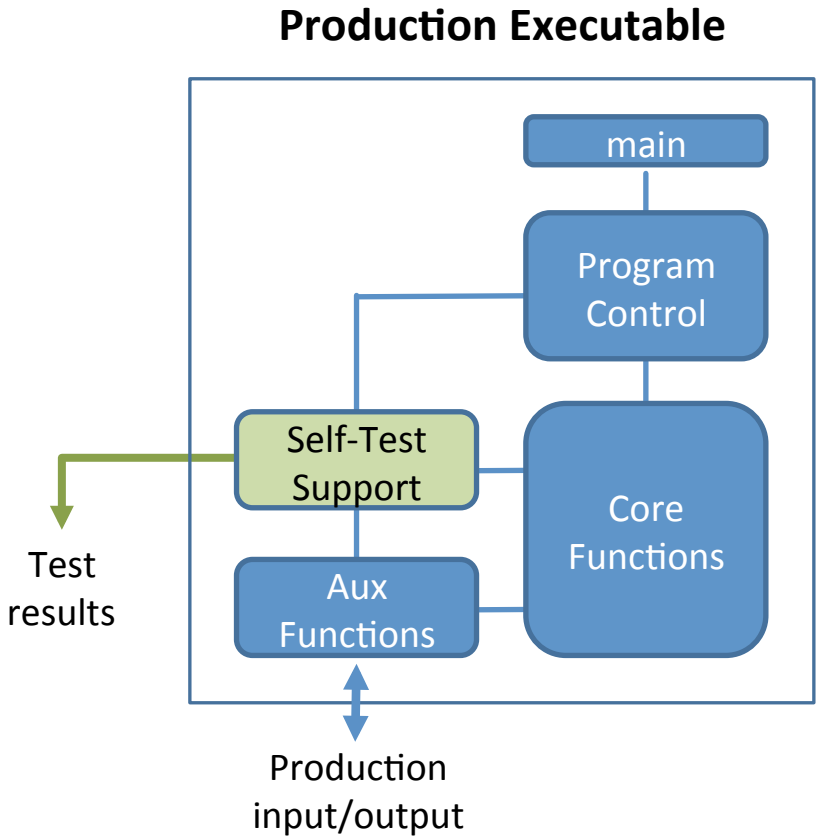


Executing Tests

Option 3: Build self-test mode(s) into your executable

Input files and/or flags indicate whether true solution is available

Other possibilities:
additional metrics/logging, deterministic seeds, etc.



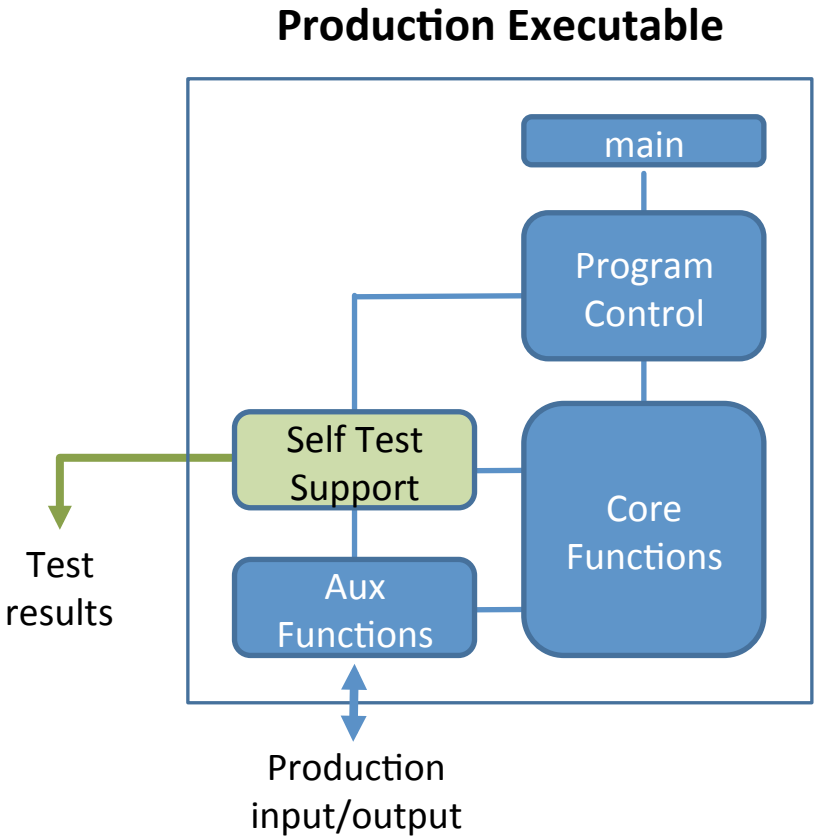
Executing Tests

Option 3: Build self-test mode(s) into your executable

Input files and/or flags indicate whether true solution is available

Other possibilities:
additional metrics/logging, deterministic seeds, etc.

Verdict: Very robust – and not as hard as it might sound



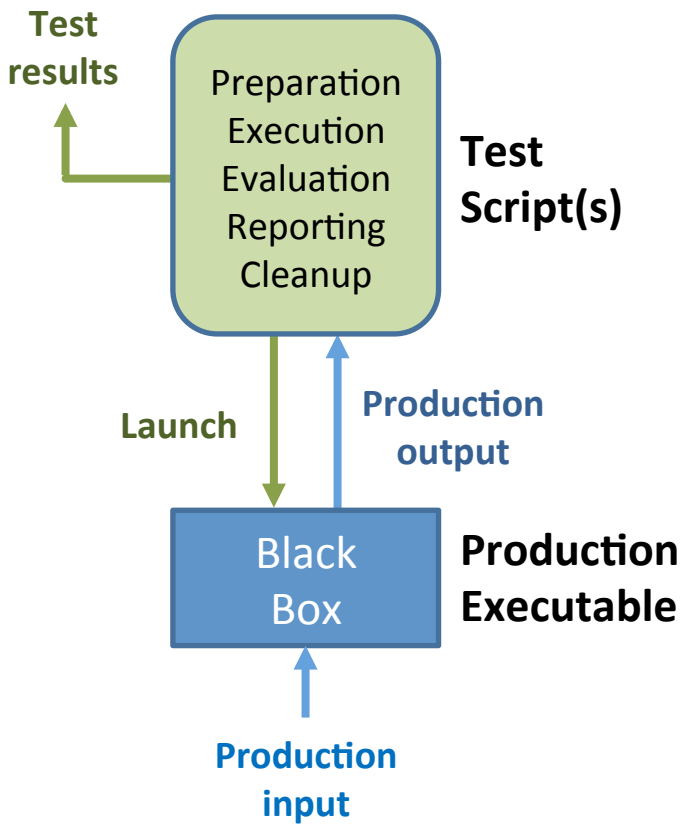
Executing Tests

Option 4: Build external test harness (test driver)

Write script(s) that launch
your app against test problem(s)

Can automate the entire process,
including assessment and reporting

TACC Products (tm and rtm by Robert McLay)
provides such a turnkey infrastructure
for scientific (floating point) codes



Executing Tests

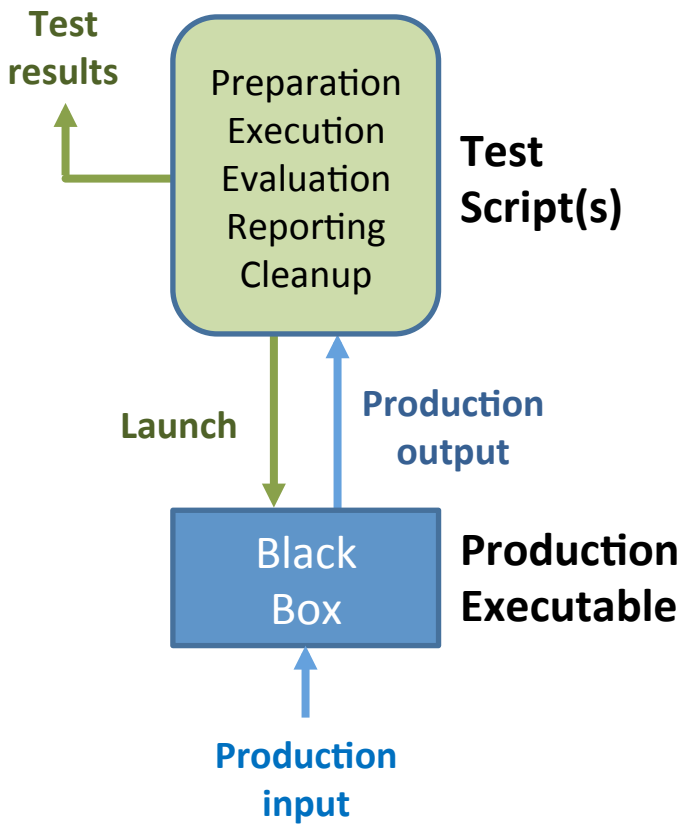
Option 4: Build external test harness (test driver)

Write script(s) that launch
your app against test problem(s)

Can automate the entire process,
including assessment and reporting

TACC Products (tm and rtm by Robert McLay)
provides such a turnkey infrastructure
for scientific (floating point) codes

**Verdict: Very flexible –
arguably the 'suite spot' [sic]
for research codes**



Repeatability and Traceability

- Simple documentation (e.g. spreadsheets) goes a long way
  - Feature list – current/planned
  - Test logs (even as simple as a diary)
- Automate as your needs evolve
  - Start with checklists, move to scripts
  - Exploit capabilities of modern development tools
  - Consider off-the-shelf test support products

Workflow -- test early and often!

- Fix-and-test continuously while writing code
  - You're always testing
  - Results are like red/green status lights
  - When light turns red you know WHEN and WHERE things went wrong
- Let bug fixes lead to new regression tests
- Make your life easy with yes/no reports

```
 ints     frac   time      computed integral     rel err   correct?
*************************************************************************
 5000     0.000  1.22    1.4699531128395666     4.9e-09     true
 5000     0.025  1.06    1.4699531128395642     4.9e-09     true
 5000     0.050  1.02    1.4699531128395680     4.9e-09     true
 5000     0.075  0.99    1.4699531128395604     4.9e-09     true
 5000     0.100  0.97    1.4699531128395558     4.9e-09     true

                          << etc etc >>
```

Testing and Refactoring

- Regression tests as canaries
  - Your frequent, simple, and immediate regression tests become like a thumbs up (or early warning) every time you make a simple code change
- Regression tests as spotters
  - You'll no longer be afraid to gut your code
- Regression tests as guides
  - You'll know what to change and when to change

Mistakes and Misconceptions

- Testing only at major milestones
- Forgetting to check if the answer is still right
- Blaming the environment (e.g. MPI stack)
- Equating normal termination with correctness
- Equating roundoff and truncation error
- Perfection paralysis and empire building

The small test suite you actually use does you more good than the comprehensive one that you don't.

Recommended Resources

- Web resources
  - OneStopSoftwareTesting.com
  - TestingExcellence.com
  - SearchSoftwareQuality.com
- Common sense testing
  - Glenford J.Myers, The Art of Software Testing (2nd ed)
- Common sense development
  - Steve McConnell: especially Code Complete (1993), Code Complete 2 (2004) (see also Construx.com and SteveMcConnell.com)
  - Other TACC courses (e.g. Defensive Programming)

Final Thoughts

- Take a step – do something!
  - Pick one or two potentially high impact ideas
  - Adapt them to meet your needs and style
- Start simple
  - Let your approach evolve as your needs do
  - sticky note to checklist to script to metrics

"The perfect is the enemy of the good."

(Voltaire)

Another Set of Final Thoughts

- Not all apps involve floating point ops
  - I write and maintain to Lmod* (environment modules)
  - My regression tests depend heavily on string comparisons

*Shameless plug: Lmod tutorial tomorrow

XXXXXXX

- XXXXXXX
- XXXXX
  - …XXXXXX
  - …XXXXXX