IEEE Aerospace Conference, Big Sky, Montana, 12 March 2020

Julia Programming Language Benchmark Using a Flight Simulation

Ray Sells
EV42/Guidance, Navigation, and Mission Analysis
DESE Research, Inc./ESSCA
NASA-MSFC 4600-5119
256-961-4619
harold.r.sells@nasa.gov

Overview

- **Julia is a relatively new computer language that aims to reduce the challenge for math-modelers to develop fast computer tools and simulations. It potentially combines the ease-of-coding feature of scripting languages (like Python) with the performance of compiled languages (like C++).**
- **A key question for Julia application to the simulation domain is, “Can Julia, with its obvious coding simplicity, provide runtime speeds comparable to conventional compiled languages for flight simulation?”**
- **A unique combination of existing elements can be employed to address the previous question**
  - **Extensively documented object-oriented simulation architecture**
  - **Industry standard rocket flight simulation**
  - **Separate versions (C++, Java, and Python) already benchmarked**

Julia Language Overview

official logo

Key Features:

- "Pythonic" syntax
- Dynamically-typed
- Built-in REPL (read-evaluate-print-loop)
- JIT (just-in-time) compilation
- Built-in library support for multi-threading, multi-core, and distributed processing
- Direct calling of C and Fortran code without "glue" code
- Automatic garbage collection (i.e. memory leak control)
- Easy extension to multi-processing

Practical Considerations:

- Designed for technical computing (like MATLAB & Fortran)
- Free and open-source
- High-level and easy-to-learn (like MATLAB & Python)
- High-performance (like C, C++, and Fortran)

Mini-Rocket Description

DESCRIPTION:

- **A missile trajectory generator**
- **Desktop PC tool**
- **Object-oriented for maximum versatility**
- **Easy to use**
- **Modern coding structure**
- **Useful as component in larger system models**
- **Developed and used for over 30 years for missiles ranging from tactical to strategic**

Very easy to configure five degree-of-freedom missile flyout model that accurately generates trajectories in three-dimensional space, including maneuver characteristics without 6DOF overhead

FEATURES:

- Very fast running with unique *osculating-plane* formulation without the overhead of rigid body equations-of-motion (runtime speed of a 3-DOF with additional two degrees-of-freedom)
- Motion in three-dimensional space
- Two independent channels (pitch and yaw) for steering and guidance
- 1-d, 2-d, or 3-d table lookups to model aerodynamics and propulsion characteristics
- Capability to model angle-of-attack variations in lift and drag
- Constraints on lateral acceleration based on angle-of-attack and closed-loop airframe response time
- Detailed models for control and guidance subsystems *not* required

PAST APPLICATIONS:

- Generate trajectory given missile aerodynamic, mass, and propulsion parameters
- Generate family of trajectories (off-line) for higher-level system simulations or system time-line studies
- In-line missile (offensive or defensive) model for engagement simulations
- In-line missile dynamics model for HWIL
- Tool for quickly examining different guidance laws
- Tool for mapping battlespace engagement constraints
- Trajectory reconstruction/flight characteristics estimation

EXTENSIVE LEGACY OF APPLICATION *(partial list)*

Strategic

- Endo-interceptors (HEDI, ARROW, THAAD)
- Exo-interceptors (ERIS, GBI, E2I)
- Anti-Satellite boosters (KE-ASAT)
- Strategic Target Vehicles (STARS, ARES)
- Numerous booster survey studies

Tactical

- NLOS-PAM
- EAPS
- Future Missile
- HAPS
- Modular Missile Technology
- IDEEAS constructive simulation (embedded interceptor)

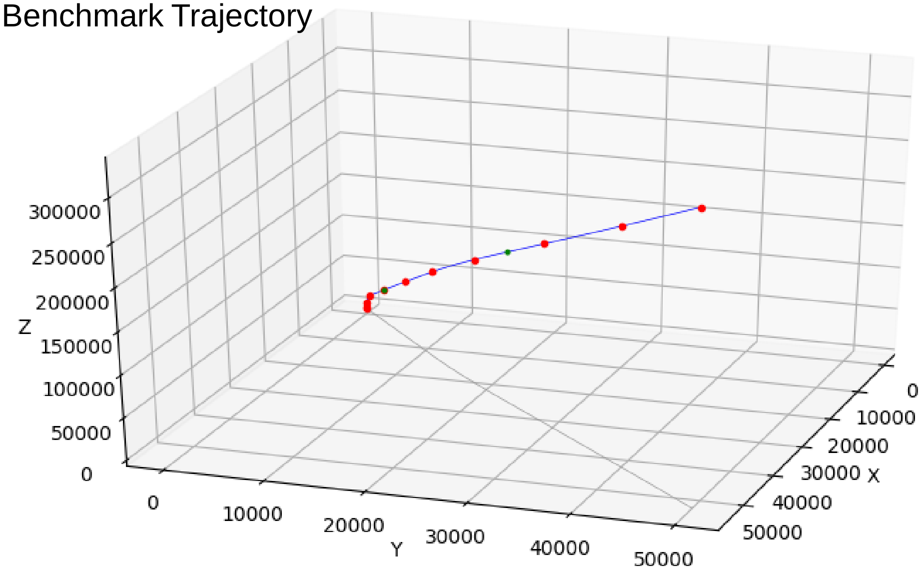
Mini-Rocket Case for Benchmark

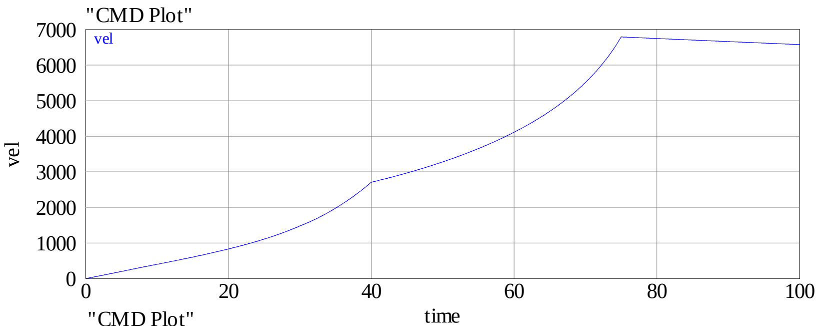
Benchmark Rocket

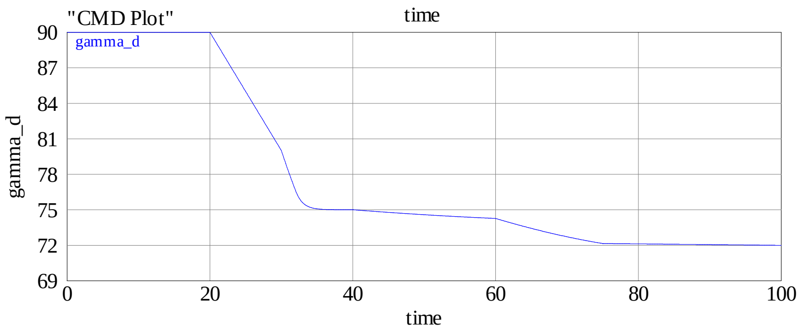
- 3-stage hypothetical rocket (to avoid SBU concerns)
- vertical launch with pitch-over
- rotating earth
- pre-programmed maneuver in pitch and yaw channels
- lift-off weight = 2500 kg
- flight time = 100 sec
- stage splits = 40, 75 at 2708, 6790 m/sec
- final velocity = 6574 m/sec at t = 100 sec

- Benchmark case replicated for all three languages: C++, Python, Java

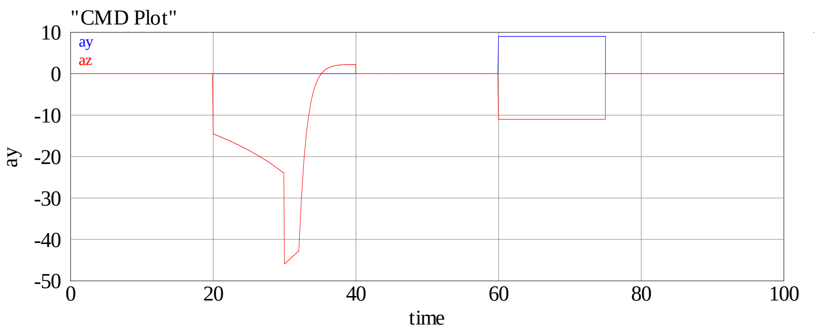
Benchmark Trajectory







- a case was chosen to illustrate 5DOF maneuver model (i.e. 6DOF minus roll)
- pitch-over (red)
- out-of-plane yaw (blue)
- start pitch over at 20 sec to 75 deg, then execute dual pitch/yaw ~1g maneuvers for 15 sec



Julia Mini-Rocket Coding Highlights – Object Oriented

DIFFERENTIAL EQUATION (DE) ENGINE (the "main" program)

```
#### SIMULATION
mr = build_rocket( "rocket.dat", "**** rocket ****")
integrators = [ mr.myclock, mr.mass, mr.vmx, mr.vmy, mr.vmz,
                mr.pmx, mr.pmy, mr.pmz]

for ii in 1:1  # MC LOOP
  clock = Clock1( 0.0, 0.01)
  init( mr)
  println( "Launch!!!")
  while clock.t <= 100.0 + 1e-6
    #### UPDATE
    update( mr, clock)
    #### REPORT
    report( mr, clock)
    #### PROPAGATE
    [ propagate( state, clock) for state in integrators]
    update( clock)
  end
end
```

OBJECT-ORIENTED

- all simulation entities are a hierarchy of objects
  - clock
  - rocket
  - rocket stages
  - atmosphere model
  - etc. down-the-hierarchy
- Differential Equation Engine orchestrates execution

```
# CREATE ATMOSPHERE OBJECT
mr.atm = Atm62()

# CREATE STAGE OBJECTS
stage1 = stage_read( fname, "**** stage 1 ****")
stage2 = stage_read( fname, "**** stage 2 ****")
stage3 = stage_read( fname, "**** stage 3 ****")
mr.stages = [ stage1, stage2, stage3]
```

TABLE OBJECT CREATION AND USAGE

```
# CREATE TABLE OBJECT WITHIN STAGE OBJECT
txv_table = table1_read( fname, "txv_table", line0)
ca_off_table = table2_read( fname, "ca_off", line0)
ca_on_table = table2_read( fname, "ca_on", line0)
cna_table = table1_read( fname, "cna_table", line0)
guide_mode_table = table1_read( fname, "guide_mode_table", line0)
ay_table = table1_read( fname, "ay_table", line0)
az_table = table1_read( fname, "az_table", line0)
gammad_table = table1_read( fname, "gammad_table", line0)
gamma_table = table1_read( fname, "gamma_table", line0)
...
...
# ACCESS TABLE OBJECT TO INTERPOLATE
txv = interp( txv_table, clock.t)
```

MINI-ROCKET JULIA BUILD PROCESS

- Simulation components built incrementally starting with DE engine and support elements (tables, atm. model, stages, integrator (clock), …)
- Extensive experimentation with coding techniques and structures to leverage Julia features with emphasis on code readability and then timing
- Benchmark timing studies conducted with DE engine and table elements to understand optimum Julia coding practice for speed
  - Multiple, independent 2^{nd} order transfer functions ran in parallel to prototype different DE engine representations and numerical integrators
  - Large-scale table lookup benchmarks conducted to optimize speed (without sacrificing readability)
  - After much experimentation, Julia was ~1/2 as fast as equivalent C++ code for DE and table look-up benchmarks

Lines-of-Code Comparison

	C++	Java	Python	Julia[4]
DE Engine[1]	360	310	227	98[2]
rocket model	830	753	596	613
vector utilities	533	524	351	0[3]
table utilities	650	384	252	165
misc. utilities	104	153	64	82
TOTAL	2477	2124	1490	958

NOTES:

1. All models coded to same O-O simulation kernel architecture
   Reference
   Sells, H.R., A Code Architecture to Streamline the
   Missile Simulation Life Cycle
   AIAA Modeling and Simulation Technologies Conference, 12 January 2017
2. Julia DE engine does not encompass full functionality for OSK train-of-objects built into other DEs. This lack of functionality was not relevant for this study.
3. Julia built-in linear algebra functionality used
4. Julia characters-of-code would have been GREATLY reduced without specifying types (at cost of great speed penalty and code readability)

Benchmark Results – Mini-Rocket[1]

	1 run		10 runs		100 runs	
C++ (CMD)[2] Borland C++ 5.5	0.125	1.00	0.126	1.01	0.127	1.02
Java 12.0.1	0.169	1.35	0.139	1.11	0.099	0.79
Python 2.7.16	3.765	30.12	3.752	30.02	3.762	30.10
Julia 1.1.1	1.470	11.76	0.378	3.02	0.269	2.15

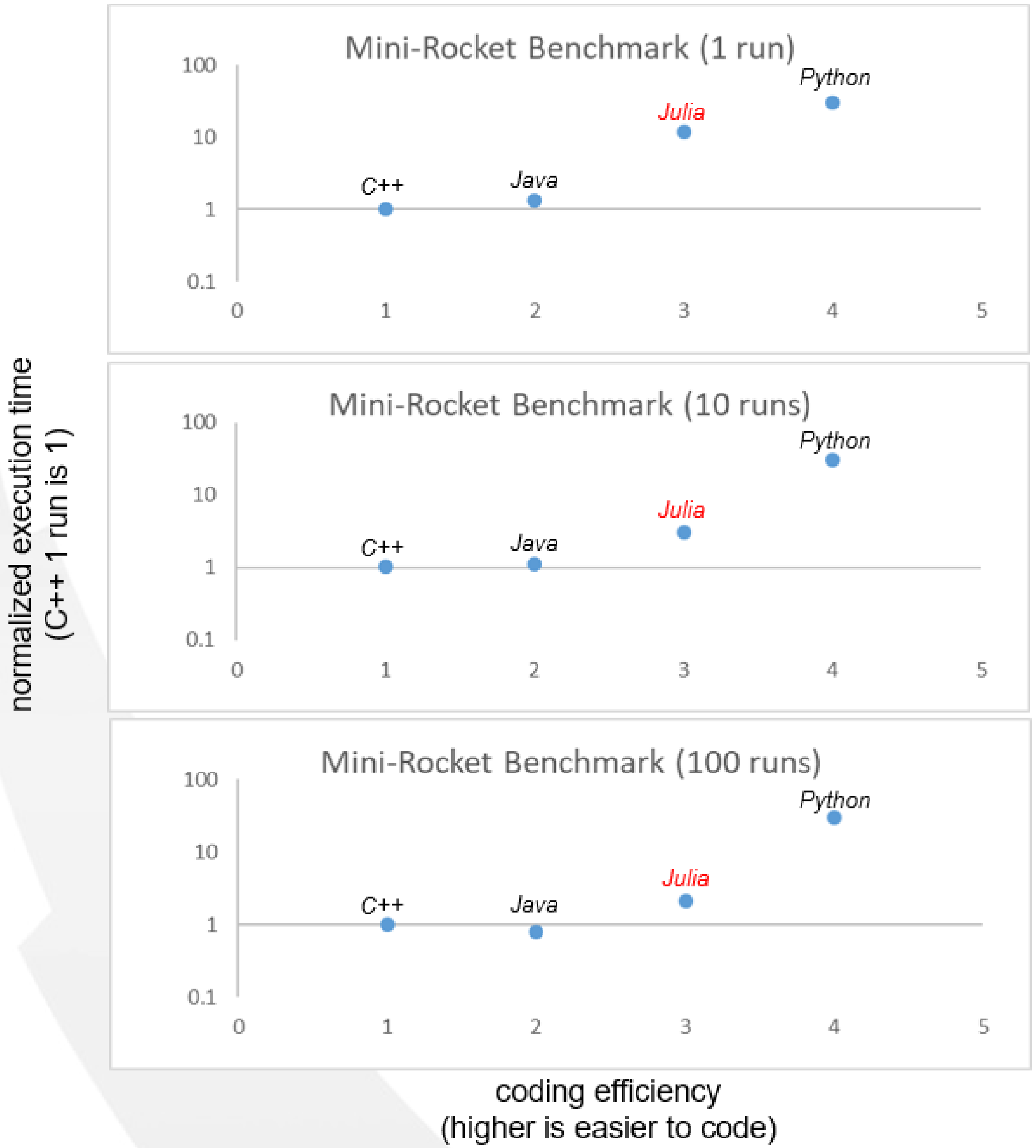
colored numbers normalized to C++ single run

- initial data file read not included in timing loop
- small amount of screen output redirected to buffered file (SSD)
- no explicit compiler optimization flags for C++ and Java
- all cases run on Windows 10, Dell Precision 7820, Wintel High Performance Dual Socket Engineering Workstation, Intel Xeon Gold 6130 CPU @ 2.10GHz (2 processors, 32 cores each) , normal single-thread processing used for these runs

References:

1. “Mini-Rocket User Guide”, U.S. Army Technical Report AMR-SS-07-27, online link https://apps.dtic.mil/docs/citations/ADA472173
2. “C++ Model Developer”, U.S. Army Technical Report, U.S. Army Technical Report AMR-SG-05-12, online link https://apps.dtic.mil/docs/citations/ADA433836

Benchmark Plots – Mini-Rocket



- Nature of Just-In-Time (JIT) is evident for Java and Julia
- JIT compile on-the-fly is more efficient for large # runs (JIT compilation consumes less amount of total run time)
- Surprise #1: Java as efficient (or better than C++) – previous experience was Java ~ 4x slower*
- Surprise #2: Python much better than previous experience ~ 100x slower*

*my personal experience as well as generally-accepted independent benchmarks

Multi-Core Processing Benchmarks

- Julia has very easy-to-extend functionality for utilizing multiple cores.
- The Julia `Distributed` package adds functionality to extend the single execution thread used-to-now to multiple cores.
- All cases run on Dell Precision 7820, Wintel High Performance Dual Socket Engineering Workstation, Intel Xeon Gold 6130 CPU @ 2.10GHz (2 processors, 32 cores each)

configuration	# concurrent runs	time (sec)	time/run
1 run/core	65[1]	3	0.046
10 runs/core	650	7.627	0.0117
100 runs/core[3]	6500	53	0.00815
1000 runs/core[3]	65000	525	0.00808

NOTES:

1. 65 runs executed concurrently including the master process and its 64 worker processes.
2. All output turned off, except final report
3. More runs per core appears to reduce overhead of distributing runs

Wrap-Up

- Challenges still exist in providing the tools and environment for quickly and efficiently constructing dynamical system simulations that address every step in the missile & rocket simulation life cycle
- The potential contribution of Julia is to give “non-expert” coders (scripters) the ability to build high performance simulations
- Julia was well suited to coding the object-oriented structure in MR with an exceptional economy-of-code
- Julia execution speed was much faster than Python but still slower than C++ and Java

Path Forward

- The economy of Julia to express complex programming constructs makes it attractive as a simulation experiment “testbed” for prototyping any future simulation applications
- Although only touched upon in these results, parallel computing capability and its application to time-domain dynamic system simulations is especially compelling for further flight simulation research