The libRoadRunner SBML JIT Compiler and Simulation Library

Endre Somogyi, Maciej Swat, James A. Glazier, Herbert M. Sauro

August 21, 2014

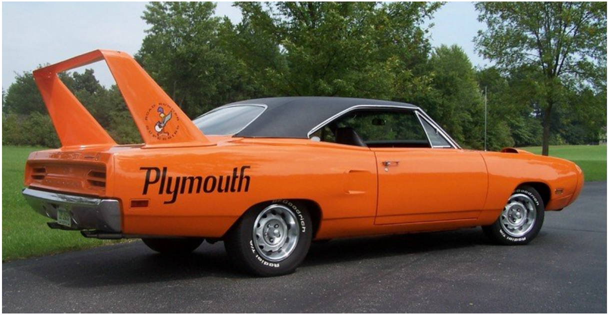
UNIVERSITY of WASHINGTON

INDIANA UNIVERSITY BLOOMINGTON

What Is It?

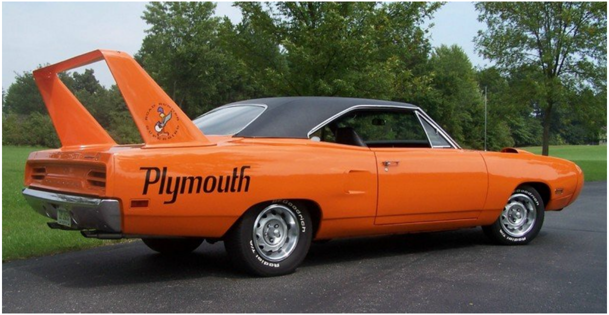
What Is It?

- A Plymouth RoadRunner?



What Is It?

- A Plymouth RoadRunner?



No, we can actually take a corner!

What Is It?

- A modular library for SBML JIT compilation, simulation and analysis.

Features

- JIT Compiler
- Analysis Features (MCA, Steady State,...)
- Cross Platform: OSX (10.6 and up), Linux (RHEL 5 and up), Win32.
- C++ with extensive native Python binding

History

- Original C# library written by Herbert Sauro and Frank Bergmann.
- Line by line transliteration to C++ by Totte Karlsson
- Current version: Ground up new design
- Kept name, re-used libStruct and other analysis functionality in original RoadRunner
- Python API wraps C++ library with Pythonic sugar.
- C-API

Modular Design

- Component based design, everything is pure virtual interfaces.
- Strict separation of model state and propagator.

$$\Gamma(t) = e^{i\mathcal{L}t}\Gamma(0)$$

- Dynamically pluggable models (system state) and integrators (propagator)
  - LLVM JIT is primary,
  - LLVM MCJIT prototype for ARM (developed by Kyle Medley)
  - GPU based on OpenCL (Kyle Medley)
  - CVODE based integrator is primary
  - Gillespie Direct Method integrator.
  - RK45 integrator (or is it just RK4?)

SBML as a Declarative Language

- Computation is specified via rules and reactions
- Fully compliant SBML event system
- Assignment rules
- Initial assignment rules
- Functions
  - most implentations treat them as macros
  - not always the most efficient form.
  - trade off between function call overhead and increase in code segment size.
  - macros increase JIT compilation time
  - developing heuristics of when more efficient to treat as macro or function.
  - function as macros yield dynamic scoping
  - most simulators expand functions inline, in order to be compatible, we implement dynamic scoping.

SBML as a Declarative Language

- Reactions

$$\frac{d}{dt}\mathbf{S}(t) = \mathbf{N}(t) \cdot \nu(\mathbf{S}(t), \mathbf{p})$$

- Rate Rules
- System Dynamics with Events

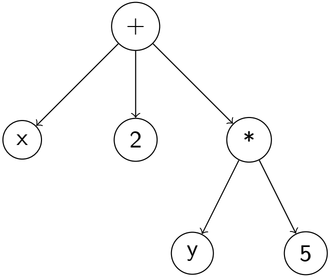
$$\mathbf{S}(t) = \sum_{E} \int_{t_i}^{ti+1} \dot{\mathbf{S}}(\mathbf{p}_i, t)dt.$$

JIT Compilation

- Compiler design overview: (1) lexical (2) syntactic, (3) semantic (4) intermediate code generation, (5), code optimizer, and (6) native code generator.
- Phases 1 through 4 are the analysis phase.
  - Source code is separated into parts and then arranged into a meaningful structure (or grammar of the language).
- Stages 5 through 6 are the synthesis phases.
  - Executable machine code is generated.
- Initial and final stages are generalizable.
- First phases are handled by libSBML.
- Final phase is handled by LLVM.
- We perform the semantic, intermediate and partially code optimization phases.

JIT Compilation

- libSBML effectively yields an AST

Infix	MathML	AST
`x + 2 + (y * 5)`	<math> <apply> <plus/> <ci>x</ci> <cn>2<cn> <apply> <times/> <ci>y</ci> <cn>5</cn> </apply> </apply> </math>	

JIT Compilation

- We generate intermediate language representation.
- Processed by LLVM to generate machine native code
- LLVM is re-targetable, x86 is most common,
- Also supports ARM, SPARC, PPC, etc code generators
- GPU code generation in development.
- LLVM IR is SSA or Single Static Assignment:
    - each var may be assigned exactly once
    - contract to Java byte code or MSIL which are stack based ILs.
- Simple low level form suitable for further analysis, optimization and native machine code generation.
- Fairly easy to look at LLVM IR and immediately tell what x86 instructions will be generated.

JIT Compilation

- We have very simple expression generator.
- Does produce large number of redundant operations.
- LLVM optimization pass performs constant folding, instruction combining, dead code elimination, etc...
- Mixed Mode Arithmetic

```
(+ 1.123 1.1 5), (* 1.0 2.34 (> A 5)).
```

  - Results of AST subtree may be logical, integer or double.
  - Sign extend logicals to double for arithmetic.
  - Integers and doubles used in logical operations are checked for zero to yield logical (fcmp, icmp)

JIT Compilation

- Data layout is tuned specifically for each model.
- All state variables are grouped in a contiguous block.
- State variables are accessed directly by the integrator, (single pointer assignment, no copying is involved).
- Lazy Evaluation
  - No redundant state variables are ever created.
  - Only store amounts
  - Everything is accessed via JITed accessors functions.
  - Rules are compiled into accessors functions.
  - Accessor functions are effectively a indirect branch (`jmp *%eax`)
  - Address is computed at compile time (jump table).

Symbol Resolution

- Each symbol in procedural languages typically correspond to a single location.
- Symbol resolution is handled with a symbol table
  - Maps symbols to memory locations
  - May be chained - local global scope.
- In SBML, symbols mean different things at different times
  - Initial assignment, assignment and functions apply at different times.
  - Symbols are not assigned memory locations if they are assignment rules.
- Sometimes a symbol table is not enough
- We created a symbol forest
- Replacement rules are resolved in the symbol forest.
- Can be chained
- Function to model to initial state scope.

Benchmarking: Brusselator and Piecewise Functions

# Brusselators	libRoadRunner - std	libRoadRunner - stiff	COPASI	libSBMLSim	SBSCL
50	0.5	0.8	0.9	12.2	13.0
100	0.9	2.3	3.8	50.5	46.1
150	1.1	4.4	7.2	N/A	1:52
200	1.9	7.2	13.2	N/A	3:51
250	2.6	11.1	21.8	N/A	8:37
300	3.3	15.6	30.1	N/A	14:09
350	3.9	21.5	46.3	N/A	21:11
400	4.7	28.6	55.7	N/A	33:35
450	5.6	36.3	1:14	N/A	48:12
500	6.6	44.5	1:35	N/A	1:14:21

# Rate Rules	libRoadRunner - std	libRoadRunner - stiff	COPASI	libSBMLSim	SBSCL
1	1.60	1.84	12.3	N/A	2.4
2	2.14	2.61	24.1	N/A	13:20
3	2.71	3.68	35.2	N/A	39:52
4	3.39	4.83	46.5	N/A	1:32:32
5	4.20	6.48	58.4	N/A	3:04:21

Benchmarking: Standard vs. Stiff Solver

model	9	14	22	33	repressilator
simulation time	150	300	2000	60	10e4
# state variables	22	86	28	10	6
absolute	1e-15	1.0e-4	1.0e-4	1.0e-4	1.0e-4
relative	1e-9	1.0e-9	1.0e-9	1.0e-9	1.0e-9
dynamics	T / S	T / S	T / S	oscil	oscil / stiff
libRoadRunner - stiff	95	510	230	125	1,040
libRoadRunner - std	760	920	235	180	3,320
COPASI	200	1,980	510	250	1,600
SBSCL	1,700	6,950	25,300	2,200	98,000
2005 SOSLib Data *Not comparable to above data					
absolute	1.0e-4	1.0e-4	1.0e-4	1.0e-4	1.0e-14
relative	1.0e-9	1.0e-9	1.0e-9	1.0e-9	1.0e-9
Dizzy 1.11.1 ODEtoJava-dopr54-adaptive	15,499	12,711	2,634	19,350	6,369
Jarnac 2.16n	344	14,531	1,157	5,843	4,516
SBMLToolbox MatlabR14SP3ode15s	188	920	302	5,554	6,681
COPASI 4.0 Build 15	156	4,062	109	1,437	500
SOSlib 1.6.0pre from CVS, Nov. 17th 2005	234	515	171	562	1,062

Sparse Multiply

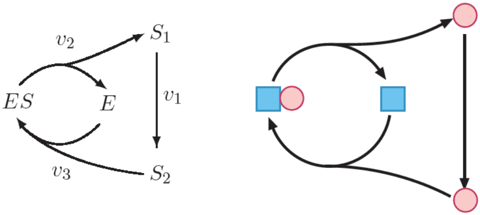
- Linear vs Quadratic scaling.
- Faster to perform CSR than JIT multiply
- Need to consider size of L1 and L2 caches
- CSR code segment fits entirely in L1 cache
- JITed code requires continuous fetch for code and data segments.
- Is a bad choice when stoichiometric coefficients are state variables.
- Rarely encountered - optimize for most common situations.

Integrators

- New integrators only require implementing the Integrator interface and registering with the IntegratorFactory.
- We plan on investigating many new integrators.
- LSODA not very suitable: FORTRAN with global state vars can not be used in parallel.
- GPU based integrator currently being developed by Kyle Medley.
- New multi-scale integrators Sundials ARKode suite will be implemented.

Conserved Moieties

- What are they and why should you care.



$$\mathbf{S}_d(t) = \mathbf{L}_0\mathbf{S}_i(t) + \mathbf{T}.$$

- Conserved Moiety Converter
- libSBML extension and converter.
- Mutable Conserved Moieties and JIT

Native Python API

- Designed from the ground up as native Python module
- SWIG'ed C++ with heavy customization
- Used 100% native Python types - lists, numpy array, etc...
- Completely self contained
- Designed to feel like part of SciPy

Native Python API

- Designed to be embedded in existing simulators
- Full access to everything via numpy arrays
- No copying

Python API Interactive Use Features

- Dynamic Python properties.
- Native Python documentation.
- All variables accessible dictionaries.
- Selection syntax.

```
r = RoadRunner('test.xml') # create a RoadRunner obj
r.k1 = 0.1                 # set a param
res = r.simulate()         # default time series sim
res = r.simulate(0,5,100,['time', '[S1]', 'S2', "S1'"])
r['init(S3)'] = 5
r.simulate(reset=True)
r.simulate(integrator='gillespie')
r.simulate(integrator='rk45')
```

Event Hooks

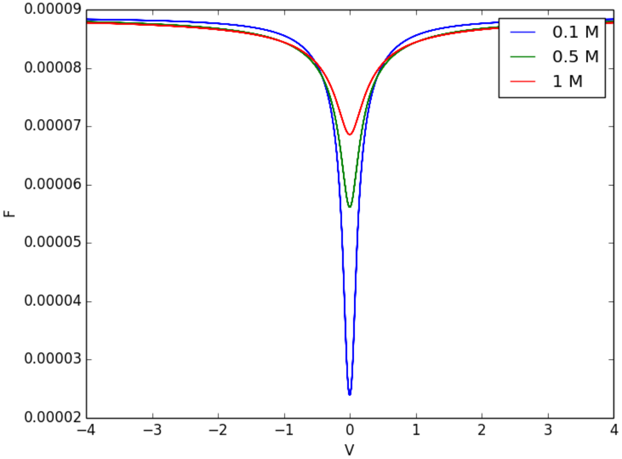
- Tie into SBML event system
- Hook models into existing simulations.
- Register user callbacks.
- Any code in callbacks can modify any model parametere.

```
def onEvent(integrator, model, eventId, time):
    print("onEvent, time: {}".format(time))
    model.SomeParameter = getSomeData(eventId)

r = RoadRunner('test.xml')

listener = PyIntegratorListener()
listener.setOnEvent(onEvent)
r.getIntegrator().setListener(listener)
res = r.simulate(0, 10, 100)
```

Real Work: Membrane Protein Mediated RedOx Reactions



capacitance vs. voltage for membrane electrical double layer.

Real Work: Membrane Protein Mediated RedOx Reactions

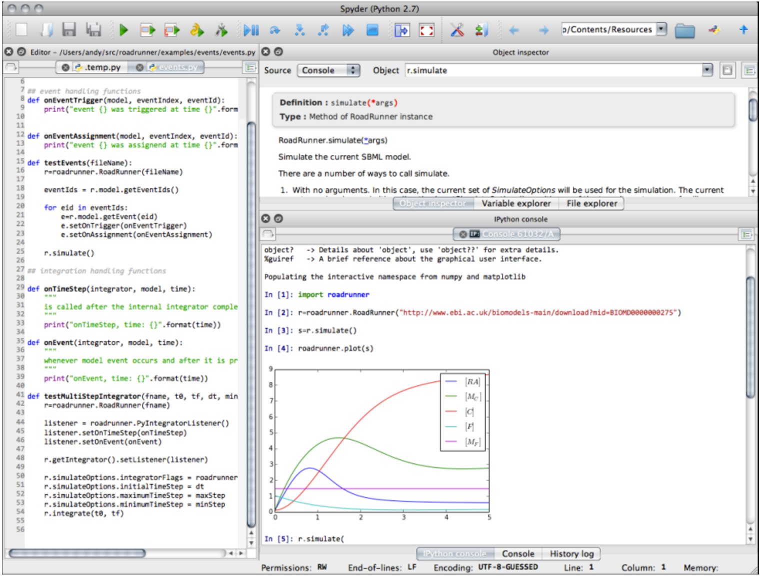
```python
r = RoadRunner('cap.xml') # create a RoadRunner obj
r.M = 0.1                 # set the molarity
def c(v):                 # cap as fuction of volt
    r.V = v               # set voltage
    r.steadyState()       # perform steady state
    return r.C            # return capacitance

cap = [c(v) for v in arange(-4, 4, 0.1)]
```

Current Applications

- University of Washington: Tellurium
- Indiana University: CompuCell3D
- University of Southern California: Bouteiller Lab
- Charité - Universitätsmedizin Berlin

Shameless Tellurium Plug



What Is It?

Definitly Not Plymouth RoadRunner.

Small, Light and Fast: More like a Lotus Elise.

Acknowledgements

- For More Information, `http://libroadrunner.org`

- The libRoadRunner project is support by NIH/NIGMS (GM081070).

National Institute of General Medical Sciences