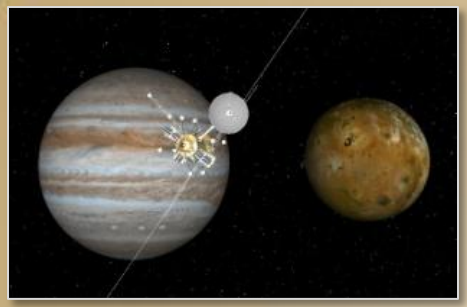
Using Python to Create Robotic Simulations for Planetary Exploration

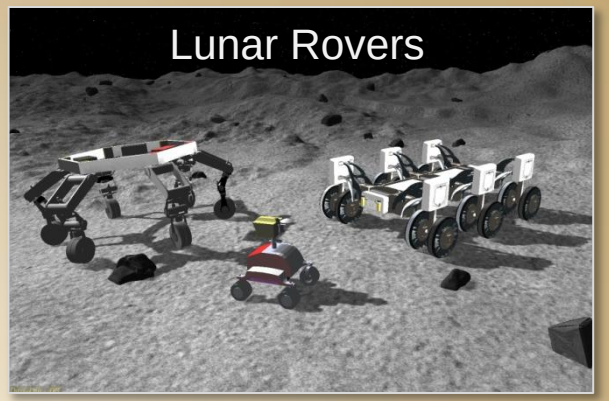
Pycon 2010, Atlanta, Georgia

Jonathan M. Cameron, Ph.D.
Jet Propulsion Laboratory / NASA
California Institute of Technology
Pasadena, California

http://jpl.nasa.gov



Background



- Jet Propulsion Laboratory
  - NASA Center focusing on robotic planetary missions
  - Mobility and Robotic Systems (http://robotics.jpl.nasa.gov)
  - Dartslab creates simulations for robotic vehicles
    - Rovers, Airships, spacecraft in space (eg, Cassini), spacecraft entering a planet's atmosphere

http://dartslab.jpl.nasa.gov

A Roams Simulation of Opportunity's Descent into the Victoria Crater

DARTS Lab
Jet Propulsion Laboratory
August 2007

Background

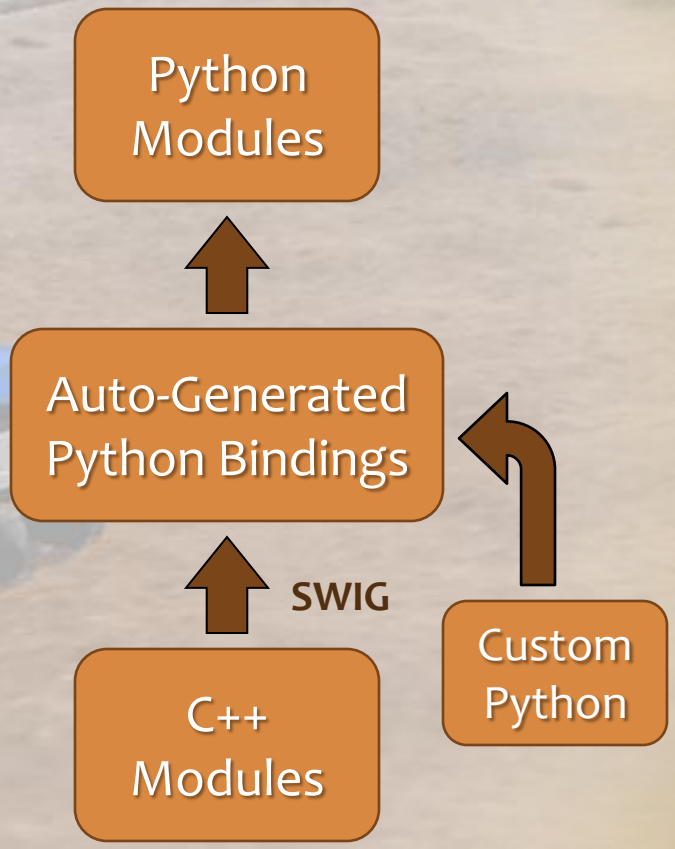
- Purposes of vehicle simulations
  - Develop and test vehicle control algorithms
    - Accurate physics, camera imagery
  - Mission planning (or analysis of mission planning)
  - Vehicle stand-in simulations for developing/testing flight software
    - High fidelity physics
    - Faster than real-time
  - Analysis (plotting, multiple simulations for performance changes to parameter variations)
  - Create and test new vehicles simulations
    - Develop and test vehicle concepts

Software Architecture Issues

- Modular code for wide range of applications
  - Many modules for multibody dynamics, vehicle physics, hardware device models, terrain modeling, camera modeling, vehicle control, ephemeris, domain specific models, etc
  - Mix and match modules for specific problems
- Agile environment to build simulation models quickly
  - Script-based language really helps speed customization
- Rapidly evolving code
  - New problem areas
  - Improving common code modules
  - Need to mix and match modules without recompiling everything

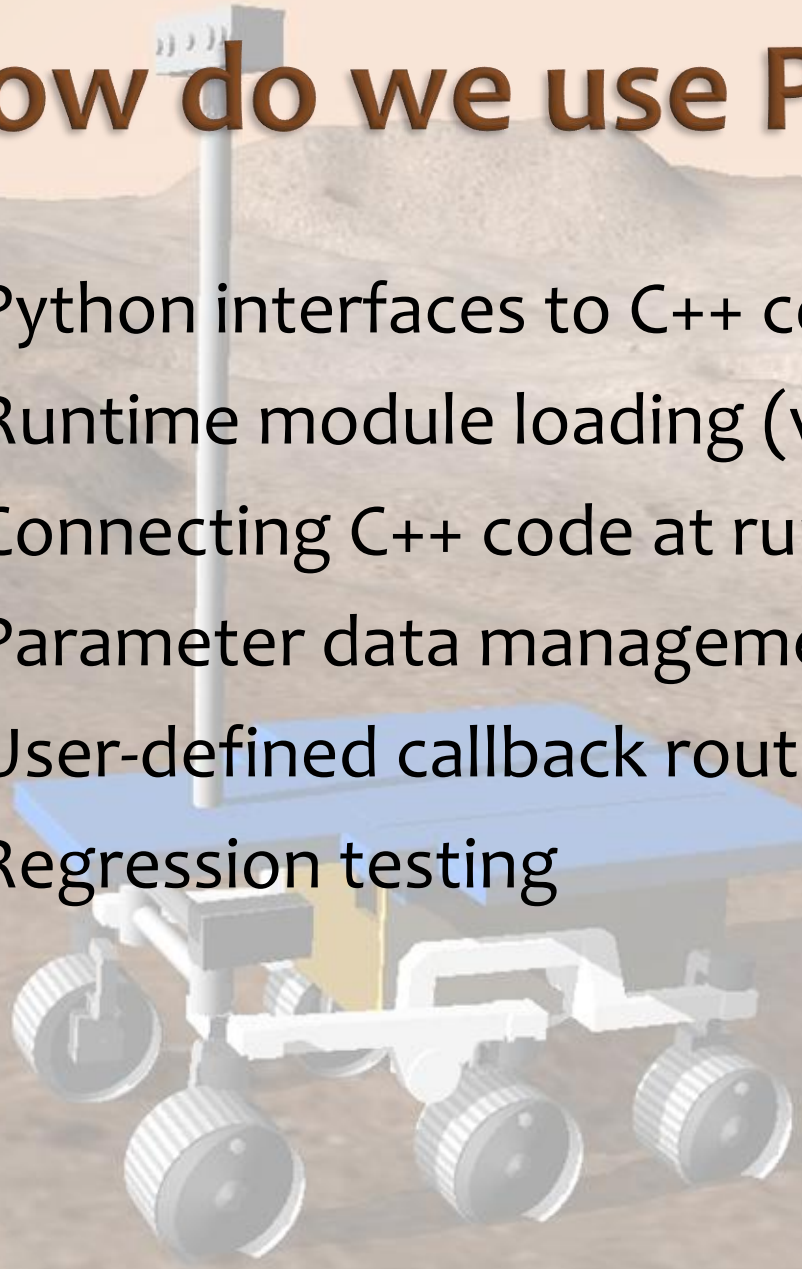
Software Architecture Highlights

- All Dartslab software is **C++** & **Python** and is organized as independent modules & libraries.
- Use **Python** as glue to configure different applications at run-time
- Use C++ library backend for "heavy-lifting', and Python for rapid prototyping – combines performance with flexibility.
- Includes support modules for visualization, large-scale Monte Carlo parametric simulations, data logging, introspection, checkpointing etc.



How do we use Python?

- Python interfaces to C++ code using SWIG
- Runtime module loading (via SWIG)
- Connecting C++ code at runtime
- Parameter data management
- User-defined callback routines
- Regression testing



C++ & Python Roles within a module

C++

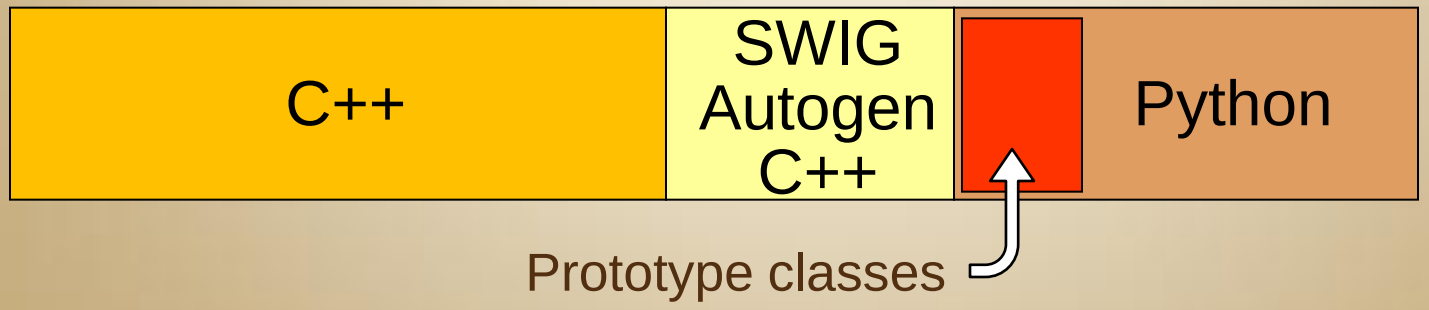
- Computationally expensive algorithms (eg. multibody dynamics)
- Time critical software
- Foundational classes
- Mature and stable code
- Math libraries

Goal:

Migrate most low-level code / libraries to C++

Python

- User interface
- Command line options
- Configuration scripts
- Templates
- Regression test scripts
- GUIs
- Loading parameters
- IPC
- Matlab, Mathematics I/F



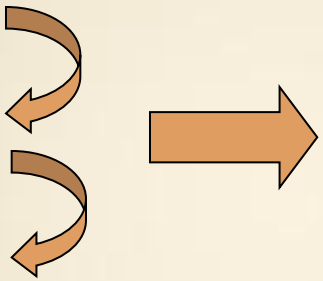
Why Python?

- Scripting language (like Tcl, Matlab), but with rich object oriented language and namespace support
- Python OO implementation schema can be made close to C++ OO software design for deployment
- Very expressive – software constructs as well as meeting computational/algorithmic needs
- Easy to import functionality from third party C/C++/Fortran libraries as Python extensions
- Ideal for prototyping – no compilation needed
- Scales well with software complexity (unlike Tcl or Perl)
- Supports named arguments
- Facilitates "live" implementation of requirements as well as validation experiments
- Comes with "batteries included" – many extension modules
- Supports command line help, inspection & introspection
- Doxygen can process Python code to generate documentation
- Supports doctests – combines documentation with testing

Developing Software…

Donald Knuth's strategy for software development:

1. Get it working
2. **Get it right**
3. Then optimize



Python has been a very effective language for making these transitions

Tony Hoare/Donald Knuth:
"Premature optimization is the root of all evil."

Why Prototype with Python?

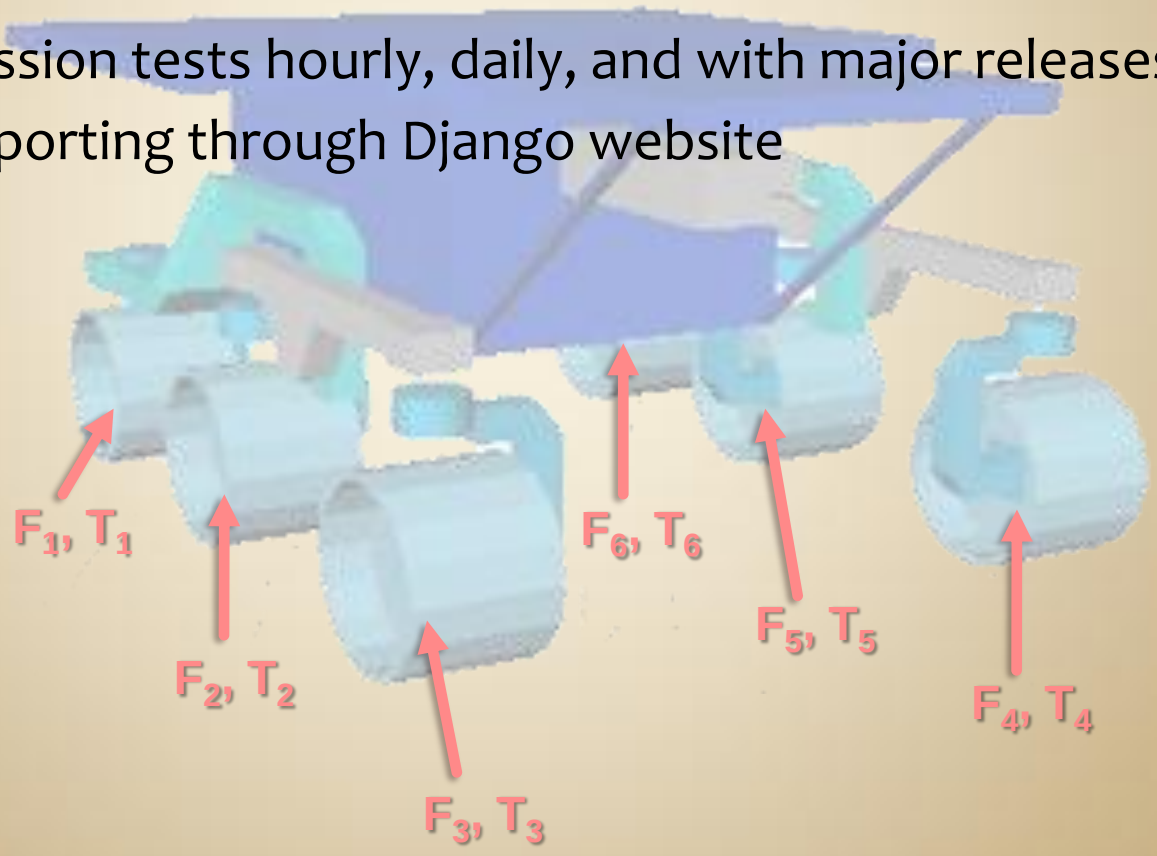
- To flesh out and mature issues related to
  - Architecture
  - Design
  - Requirements
  - Usage
- Refactoring existing code
- Working directly in C++ is not a problem for incremental or modest extensions (if design is clear)
- Large scale extensions, significant refactoring, or new module development can be more expensive and time consuming to do directly in C++
  - C++ complexity
  - Makefiles
  - Declarations
  - Compiling & linking
  - Limited access to mid-level TPS modules, eg. Sockets, XMLRPC, XML etc. that straddle multiple domains

Python connects C++ code

- Using Python to glue together C++ code at runtime
  - Although low-level classes are C++, we use Python classes to interact with C++ objects. Once connected all interactions are at full C++ speeds.
  - Use Python data (dictionaries) to configure and set up low-level C++ objects. Once set up, the objects interact with C++ mechanisms; therefore they are faster.
  - Reduces need for monolithic complied programs
  - Lazy loading of C++ modules/libraries

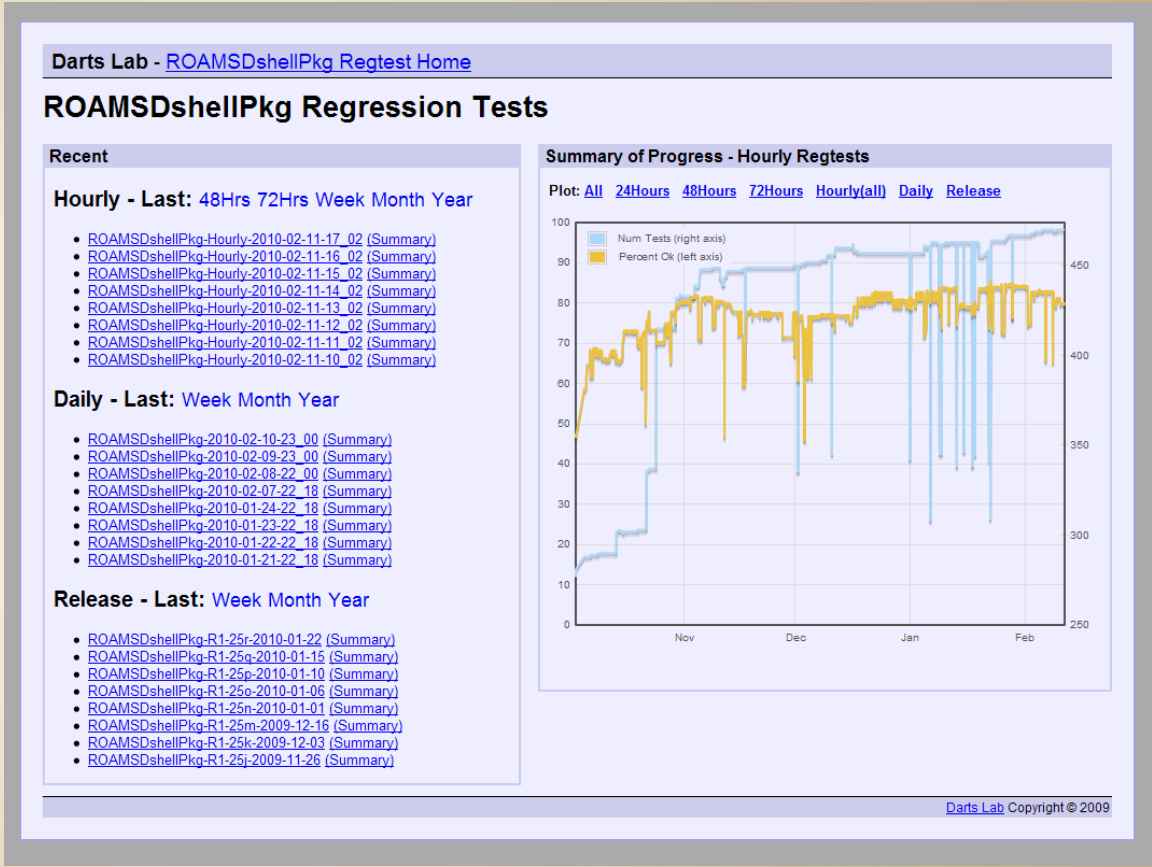
Python for regression testing

- Using Python for regression testing
  - Use doctests for testing (not PyUnit yet ☹)
  - Large regression test suite
  - Run regression tests hourly, daily, and with major releases
  - Results reporting through Django website



Python for regression testing

- Reporting through internal Django website
  - Virtualenv
  - “flot” Javascript library for plotting
- Identifies when regtests fail
- Helps maintain large codebase despite rapid code evolution



Conclusions

- Good experience with Python
- An important tool in the Dartslab