`Inference` — A Python Package for Astrostatistics

Tom Loredo (Dept. of Astronomy, Cornell University), Alanna Connors (Eureka Scientific),
and Travis Oliphant (Dept. of Electrical & Computer Engineering, Brigham Young University)

Project Overview

Motivation

Many advanced astrostatistics methods are *conceptually* simple despite being *computationally* complex.

Competing methods of very different levels of sophistication are often similar from an end-user's perspective.

The principle obstacle to the use and understanding of advanced methods is the *art of statistical computing*—the computational tricks needed to implement advanced methods.

Goal: Eliminate this obstacle!

Example—Fitting binned spectral data from data contaminated with measured background:

- Minimize χ^2 using background-subtracted data
- Maximize a Poisson counting process likelihood marginalized over a bin-by-bin Poisson background model

These are quite similar from a user's perspective: One must (1) Define a parameterized signal model; and (2) Optimize a scalar function of the model's parameters. Analysts should not be prevented from trying the (more exact) likelihood approach simply because efficient computation of the likelihood requires unconventional computational "tricks."

Main Features

The `Inference` project is making advanced astrostatistics methods accessible to astronomers via the following project components:

- The `Inference` package—Two software components
  - *Library:* A deep and broad collection of self-contained functions and objects implementing methods tailored to astronomers' needs. Where possible, it includes multiple methods in each problem class, esp. frequentist/Bayesian
  - *Parametric Inference Engine:* A framework for analyzing parametric models allowing use of multiple methodologies (χ^2, likelihood, Bayes) with a unified interface
- Use of a modern "very high level" (VHL) computer language: Python
  - Single implementation facilitates depth/breadth (vs. spreading resources across implementations in several languages)
  - Python's VHL features speed development, facilitate testing
  - Python's simplicity allows easy access both to new users and to astronomers using PyRAF
- Outreach
  - This project organizes and sponsors astrostatistics speakers and sessions at astronomy and astroparticle physics conferences (like HEAD!)
  - Selected methods described in project-sponsored talks will be included in the Python package

A Bit About Python

Language Characteristics

- A general purpose language with a rich standard library
- Very simple syntax—resembles "pseudo code"
- Use interactively, or via scripts/modules
- Object oriented, with a very simple object model—facilitates high level interfaces, modularity
- Practical rather than "pure"—Selected capabilities of various paradigms (e.g., functional programming, list comprehensions, metaclasses)
- Sophisticated and fast scientific computing capability
- Easily extendible/embeddable with C/C++/Fortran
- Open source, cross-platform, active & growing user community
- Named for the British comedy show, not the snake!

Scientific Computing With Python

- Array computations
  - Syntax inspired by Matlab/IDL/Fortran90
  - Performance near that of C/Fortran for array calculations
  - `Numeric`: Developed by LLNL/MIT scientists & programmers
  - `numarray`: `Numeric`'s successor developed by NASA/STScI; allows larger (memory-mapped) arrays, inhomogeneous arrays (for FITS files)
- PyRAF — The IRAF command line in Python (STScI)
- SciPy (partly supported by NASA AISR via `Inference`)
  - High level interfaces to large, well-established libraries: special functions, linear algebra, FFTs, DSP, quadrature, ODE solvers, optimizers, basic stats
  - Special functions are *universal functions* (ufuncs); can be "broadcast" onto arrays at speed near compiled C; users can create ufuncs
  - Inline C via weave package
- Plotting (`matplotlib`, `Chaco` partly supported by STScI)
  - `matplotlib`: Cross-platform 2-d plotting a la Matlab (mature)
  - `Chaco`: Object-oriented, modular, cross-platform plotting (beta-level)
  - Interfaces to very many popular libraries (gnuplot, pgplot, DISLIN, etc.)

Simple Example

Rayleigh statistic for period searching in arrival time data:

$R(\omega) = \frac{1}{N}\left[\left(\sum_i \sin\omega t_i\right)^2 + \left(\sum_i \cos\omega t_i\right)^2\right]$

Python source code

```
from Numeric import *

def Rayleigh (data, w):
    wd = w*data
    return (sum(sin(wd))**2 +
         sum(cos(wd))**2)/len(data)
```

C source code

```
#include <math.h>

double Rayleigh (int n, double *data,
                 double w) {
    double S, C, wt;
    int i;
    S = 0.;
    C = 0.;
    for (i=0; i<n; i++) {
        wt = w*data[i];
        S += sin(wt);
        C += cos(wt);
    }
    return (S*S + C*C)/n;
}
```

Components of the Package

Library

Tools for Continuous Data

Methods for data from sampled functions with additive noise, $d_i = f(t_i) + e_i$:

- Linear & nonlinear regression
  - Interfaces to Bershady/Isobe packages (regression with measurement error and intrinsic scatter)
  - Bayesian errors-in-variables modeling (EVM)
  - Fitting with correlated errors
- Detection/measurement of periodic signals
  - Standard approaches: Power spectrum, Schuster periodogram, Lomb-Scargle
  - Fractional fast Fourier transform (fFFT)
  - Bretthorst algorithm (Bayesian periodograms)
  - Bayesian piecewise-constant modeling (Gregory method)
  - Kepler periodogram (Kepler reflex motion modeling for binaries, exoplanets)
- Nonperiodic time series analysis (QPOs, $1/f$ noise): ARMA models, long-memory processes
- Robust estimation/outlier detection (M-estimators, Bayesian outlier detection)

Tools for Discrete Data

Methods for data from counting processes and point processes:

- Intervals and limits for rates and ratios using counting process data
  - Likelihood & Bayesian intervals for simple processes
  - Methods with known background rate: Feldman-Cousins likelihood ordering, Bayes, ABC (bootstrap)
  - Methods with uncertain background: Profile likelihood, Bayes, ABC
- Periodic point processes (period searching in arrival time data):
  - Frequentist: Rayleigh statistic, Z_N^2
  - Bayesian: log-Fourier models, Gregory-Loredo method
  - Accelerated $(P,\dot{P})$ searching with incoherent spectra and fractional transforms
- Inhomogeneous point process models for local event detection: Bayes blocks, Poisson "Haar" wavelets
- Survey analyses: Survival analysis (ASURV), point process $+$ noise
- Nonparametric methods: Adaptive splines, neural nets (interfaces to Max Planck PPI methods), mixture models

Parametric Inference Engine (PIE)

Capabilities

- Three inference methodologies, each for various data types:
  - χ^2: point samples, binned samples, "folded" (response functions)
  - Maximum likelihood: Gaussian (matching χ^2 cases), Poisson counting processes, Point processes (surveys w/ efficiency functions)
  - Bayesian: Matching ML cases
- Automate standard parameter exploration tasks
  - Exploration on equispaced & logarithmic grids (adaptive refinement in 1-d, e.g., for period searching)
  - Optimization (unconstrained and with boundary constraints)
  - Exploration of subsets of parameter space (profiling/projection)
  - Hessian/information matrix calculation
- Bayesian computation
  - Marginalization and Bayes factors via adaptive quadrature & Laplace approximation
  - Calculation of 1-d, 2-d, 3-d credible region boundaries
  - Basic Markov chain Monte Carlo (MCMC) support
- Simulate data (calibrate confidence regions; experimental design)

Interface

Build a parametric model by creating a class with the `ParametricModel` base class, containing parameters and a `signal` method:

```
class PowerLawModel(ParametricModel):
    A = RealParameter(1., 'Amplitude')
    alpha = RealParameter(0.5, 'Power law index')

    def signal(self,E):
        return self.A*E**(self.alpha)
```

For simple inferences, create an `Inference` object using the model and one or more data sets:

```
inf = BinnedChisqrInference(PowerLawModel, data1, data2, ...)
```

The *Inference* object gives you all the methods you need to make the specified type of inference; e.g., for projected χ^2:

```
inf.A.logStep(0., 10., 51) # 51 log-spaced steps for A
inf.alpha.vary() # Let alpha vary
grid = inf.opt() # Returns a grid object w/ projected chi**2(A)
```

For more complicated inferences, e.g., combining information from different types of data, you need to use just one other set of classes: `Predictor` classes for each type of data. These specify how to compare a particular type of data to a signal, and how to simulate that type of data.

```
p1 = SampledChisqrPred(data1);  p2 = BinnedChisqrPred(data2)
inf = ChisqrInference(PowerLawModel, p1, p2)
```

Models have support for vector output, setup calculations and array broadcasting. `Predictor` classes are "tunable" (e.g., set quadrature for integrating over a bin). You can easily create your own to add new data types.

First release is expected around the New Year. Please sign up below if you'd like to be notified!

Contact information for Tom Loredo: Email — *loredo@astro.cornell.edu*; Web — *http://www.astro.cornell.edu/staff/loredo/*
The `Inference` project is sponsored by NASA's Applied Information Resources Program. We are very grateful for their support!