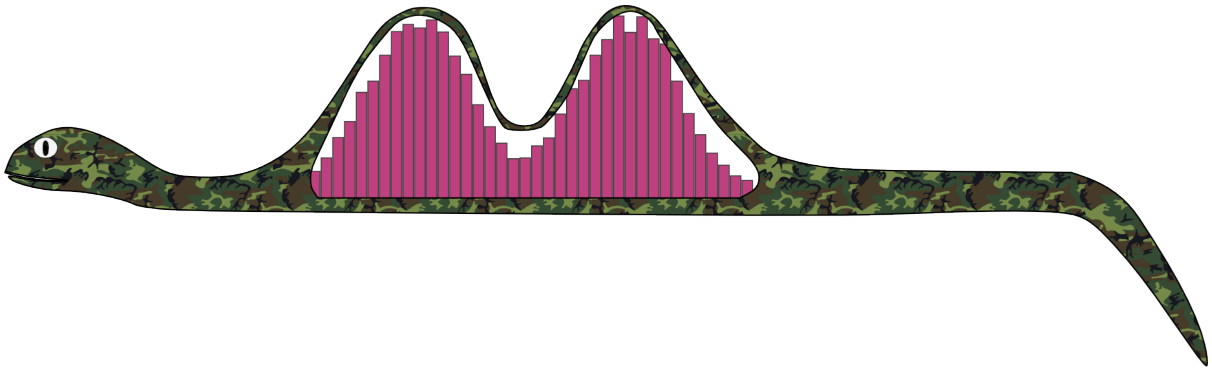

BIP - Bayesian Inference with Python Documentation

Release 0.5.13

Flávio Codeço Coelho

2014-05-27

CONTENTS

This documentation corresponds to version 0.5.13.

CHAPTER

ONE

CONTENTS

1.1 Overview

The Bip Package is a collection of useful classes for basic Bayesian inference. Currently, its main goal is to be a tool for learning and exploration of Bayesian probabilistic calculations.

Currently it also includes subpackages for stochastic simulation tools which are not strictly related to Bayesian inference, but are currently being developed within BIP. One such package is the BIP.SDE which contains a parallelized solver for stochastic differential equations, an implementation of the Gillespie direct algorithm.

The Subpackage Bayes also offers a tool for parameter estimation of Deterministic and Stochastic Dynamical Models. This tool will be fully described briefly in a scientific paper currently submitted for publication.

1.2 Parameter Estimation in Dynamic Models

A growing theme in mathematical modeling is uncertainty analysis. The Melding Module provides a Bayesian framework to analyze uncertainty in mathematical models. It includes tools that allow modellers to integrate Prior information about the model's parameters and variables into the model, in order to explore the full uncertainty associated with a model.

This framework is inspired on the original Bayesian Melding paper by Poole and Raftery [1], but extended to handle dynamical systems and different posterior sampling mechanisms, i.e., the user has the choice to use Sampling Importance resampling, Approximate Bayesian computations or MCMC. A deeper description of the methodology implemented in this package is available as published research paper [2]. This paper also contains a more extensive example of parameter estimation. If you intend to use this package for a scientific publication, you should cite this paper [1].

Once a model is thus parameterized, we can simulate the model, with full uncertainty representation and also fit the model to available data to reduce that uncertaity. Markov chain Monte Carlo algorithms are at the core of the framework, which requires a large number of simulations of the models in order to explore parameter space.

1.2.1 Single Session Retrospective estimation

Frequently, we have a complete time series corresponding to one or more state variables of our dynamic model. In such cases it may be interesting to use this information, to estimate the parameter values which maximize the fit of our model to the data. Below are examples of such inference situations.

[1] Poole, D., & Raftery, A. E. (2000). Inference for Deterministic Simulation Models: The Bayesian Melding Approach. Journal of the American Statistical Association, 95(452), 1244-1255. doi:10.2307/2669764

[2] Coelho FC, Codeço CT, Gomes MGM (2011) A Bayesian Framework for Parameter Estimation in Dynamical Models. PLoS ONE 6(5): e19616. doi:10.1371/journal.pone.0019616

Example Usage

This first example includes a simple ODE (an SIR epidemic model) model which is fitted against simulated data to which noise is added:

```python
#-*- coding: utf-8 -*-
"""
Parameter estimation and series forcasting based on simulated data with moving window.
Deterministic model
"""
#
# Copyright 2009- by Flávio Codeço Coelho
# License gpl v3
#
from BIP.Bayes.Melding import FitModel
from scipy.integrate import odeint
import scipy.stats as st
import numpy as np

beta = 1 #Transmission coefficient
tau = .2 #infectious period. FIXED
tf = 36
y0 = [.999,0.001,0.0]
def model(theta):
    beta = theta[0]
    def sir(y,t):
        '''ODE model'''
        S,I,R = y
        return  [-beta*I*S, #dS/dt
                beta*I*S - tau*I, #dI/dt
                tau*I] #dR/dt
    y = odeint(sir,inits,np.arange(0,tf,1))
    return y

F = FitModel(500, model,y0,tf,['beta'],['S','I','R'],
            wl=36,nw=1,verbose=0,burnin=100)
F.set_priors(tdists=[st.norm],tpars=[(1.1,.2)],tlims=[(0.5,1.5)],
    pdists=[st.uniform]*3,ppars=[(0,.1),(0,.1),(.8,.2)],plims=[(0,1)]*3)
d = model([1.0]) #simulate some data
noise = st.norm(0,0.01).rvs(36)
dt = {'I':d[:,1]+noise} # add noise
F.run(dt,'MCMC',likvar=1e-5,pool=True,monitor=[])
#==Uncomment the line below to see plots of the results
F.plot_results()
```

The code above starts by defining the models parameters and initial conditions, and a function which takes in the parameters runs the model and returns the output.

After that, we Instantiate our fitting Object:

```python
F = FitModel(300,model,y0,tf,['beta'],['S','I','R'],
            wl=36,nw=1,verbose=False,burnin=100)
```

Here we have to pass a few arguments: the first (`K=300`) is the number of samples we will take from the joint prior distribution of the parameters to run the inference. The second one (`model`) is the callable(function) which corresponds to the model you want to fit to data. Then you have the initial condition vector(`inits=y0`), the list of parameter names (`thetanames = ['beta']`), the list of variable names (`phinames=['S','I','R']`), inference window length (`wl=36`), number of juxtaposed windows (`nw=1`), verbosity flag (`verbose=False`) and finally the number of burnin samples (`burnin=1000`), which is only needed for if the inference method chosen is `MCMC`.

One should always have `verbose=True` on a first fitting run of a model or if the simulations seems to be taking longer than expected. When `verbose` is true, printed and graphical is generated regarding the behavior of fitting, which can be useful to fine tune its parameters.

The next line of code also carries a lot of relevant information about the inference: the specification of the prior distributions. By now you must have noticed that not all parameters included in the model need to be included in the analysis. any number of them except for one can be set constant, which is what happens with the parameter `tau` in this example:

```
F.set_priors(tdists=[st.norm],tpars=[(1.1,.2)],tlims=[(0.5,1.5)],
    pdists=[st.uniform]*3,ppars=[(0,.1),(0,.1),(.8,.2)],plims=[(0,1)]*3)
```

here we set the prior distributions for the theta (the model's parameters) and phi (the model's variables). `tdists`, `tpars` and `tlims` are theta's distributions, parameters, and ranges. For example here we use a Normal distribution (`st.norm`) for `beta`, with mean and standard deviation equal to 1.1 and .2, respectively. we also set the range of `beta` to be from 0.5 to 1.5. We do the same for phi.

The remaining lines just generate some simulated data to fit the model with, run the inference and plot the results which should include plots like this:

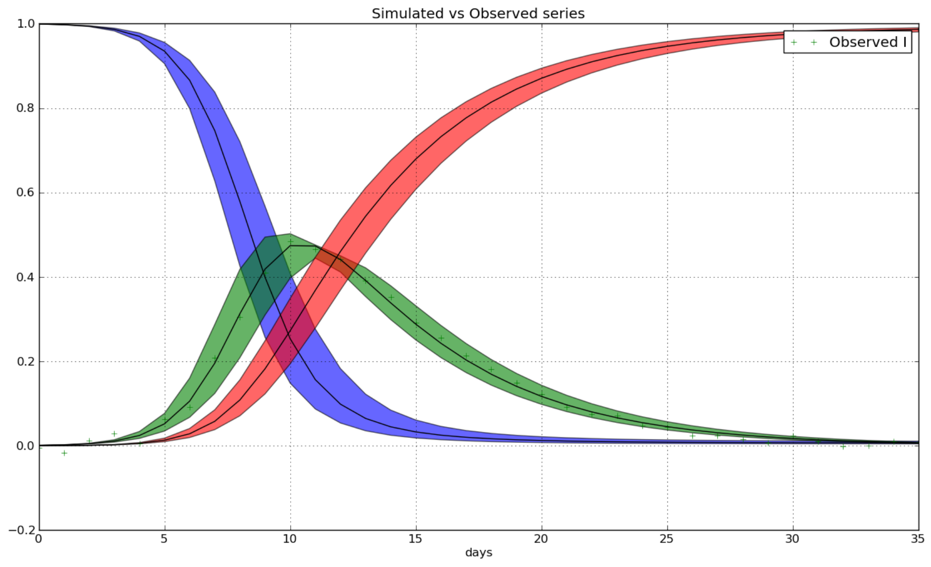

Figure 1.1: Series posterior distributions. Colored areas represent 95% credible intervals.

One important argument in the run call, is the likvar, Which is the initial value for the likelihood variance. Try to increase its value if the acceptance ratio of the markov chain is too llow. Ideal levels for the acceptance ratio should be between 0.3 and 0.5.

The code for the above example can be found in the examples directory of the BIP distribution as `deterministic.py`

Stochastic Model Example

This example fits a stochastic model to simulated data. It uses the *SDE* package of BIP:

```
#-*- coding:utf-8 -*-
"""
Parameter estimation and series forcasting based on simulated data with moving window.
Stochastic model
"""
#
# Copyright 2009- by Flávio Codeço Coelho
# License gpl v3
#
from BIP.SDE.gillespie import Model
```

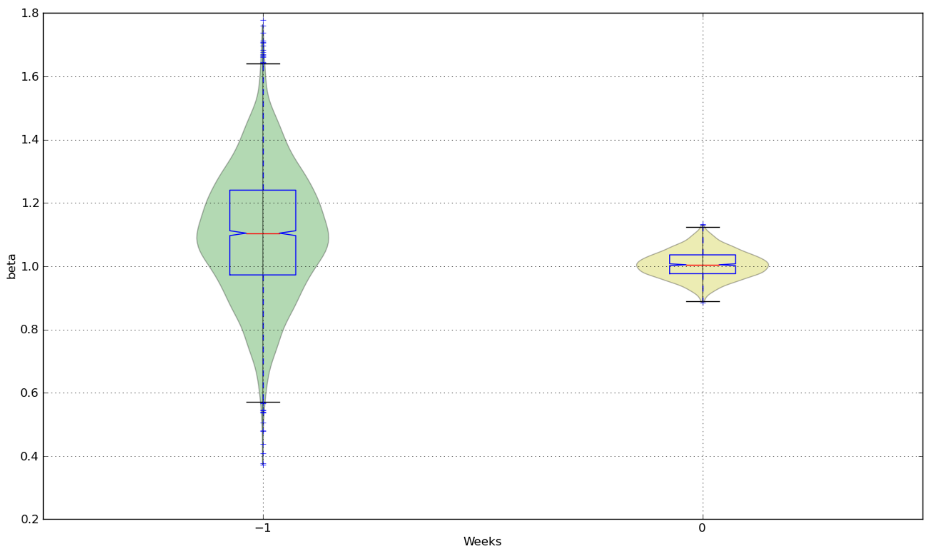

Figure 1.2: Parameters prior and posterior distributions.

```
from BIP.Bayes.Melding import FitModel
import numpy as np
from scipy import stats as st


mu = 0.0 #birth and death rate.FIXED
beta = 0.00058 #Transmission rate
eta = .5 #infectivity of asymptomatic infections relative to clinical ones. FIXED
epsilon = .1 #latency period
alpha = .2 #Probability of developing clinical influenza symptoms
sigma = .5 #reduced risk of re-infection after recovery
tau = .01 #infectious period. FIXED
# Initial conditions
global inits,tf
tf= 140
inits = [490,0,10,0,0]
pars = [beta,alpha,sigma]


# propensity functions
def f1(r,inits):return r[0]*inits[0]*(inits[2]+inits[3])#S->E
def f2(r,inits):return r[1]*inits[1]#E->I
def f3(r,inits):return r[3]*inits[2]#I->R
def f4(r,inits):return r[2]*inits[1]#E->A
def f5(r,inits):return r[4]*inits[3]#A->R

def runModel(theta):
    global tf,inits
    step = 1
    #setting parameters
    beta,alpha,sigma = theta[:3]
    vnames = ['S','E','I','A','R']
    #rates: b,ki,ka,ri,ra
    #r = (0.001, 0.1, 0.1, 0.01, 0.01)
    r = (beta, alpha*epsilon, (1-alpha)*epsilon, tau, tau)
    #print r,inits
    # propensity functions
    propf = (f1,f2,f3,f4,f5)

    tmat = np.array([[-1, 0, 0, 0, 0],
                    [ 1,-1, 0,-1, 0],
                    [ 0, 1,-1, 0, 0],
                    [ 0, 0, 0, 1,-1],
                    [ 0, 0, 1, 0, 1]
                ])
    M=Model(vnames=vnames,rates = r,inits=inits,tmat=tmat,propensity=propf)
    #t0 = time.time()
    M.run(tmax=tf,reps=1,viz=0,serial=True)
    t,series,steps,events = M.getStats()
    ser = st.nanmean(series,axis=0)
    #print series.shape, ser.shape
    return ser

d = runModel([beta,alpha,sigma])
#~ import pylab as P
#~ P.plot(d)
#~ P.show()

dt = {'S':d[:,0],'E':d[:,1],'I':d[:,2],'A':d[:,3],'R':d[:,4]}
F = FitModel(900, runModel,inits,tf,['beta','alpha','sigma'],['S','E','I','A','R'],
        wl=140,nw=1,verbose=0,burnin=100)
F.set_priors(tdists=[st.uniform]*3,tpars=[(0.00001,.0006),(.1,.5),(0.0006,1)],tlims=[(0,.001),(.0
    pdists=[st.uniform]*5,ppars=[(0,500)]*5,plims=[(0,500)]*5)

F.run(dt,'MCMC',likvar=1e1,pool=0,monitor=[])
#~ print F.optimize(data=dt,p0=[0.1,.5,.1], optimizer='oo',tol=1e-55, verbose=1, plot=1)
#==Uncomment the line below to see plots of the results
F.plot_results()
```

This example can be found in the examples folder of BIP under the name of `flu_stochastic.py`.

1.2.2 Iterative Estimation and Forecast

In some other types of application, one's data accrue gradually and it may be interesting to use newly available data to improve previously obtained parameter estimations.

Here we envision two types of scenarios: one assuming constant parameters and another where parameter values can actually vary with time. These two scenarios lead to the two fitting strategies depicted on figure

1.2.3 References

1.3 Stochastic Differential Equations

The SDE package in BIP, was born out of the need to simulate stochastic model to test the Parameters estimation routines in the Bayes Package. However, it is useful in many general-purpose application since it provides a pure Python implementation of an SDE solver.

Currently it provides a single solving algorithm, the Gillespie SSA. but other algorithms are planned for future releases

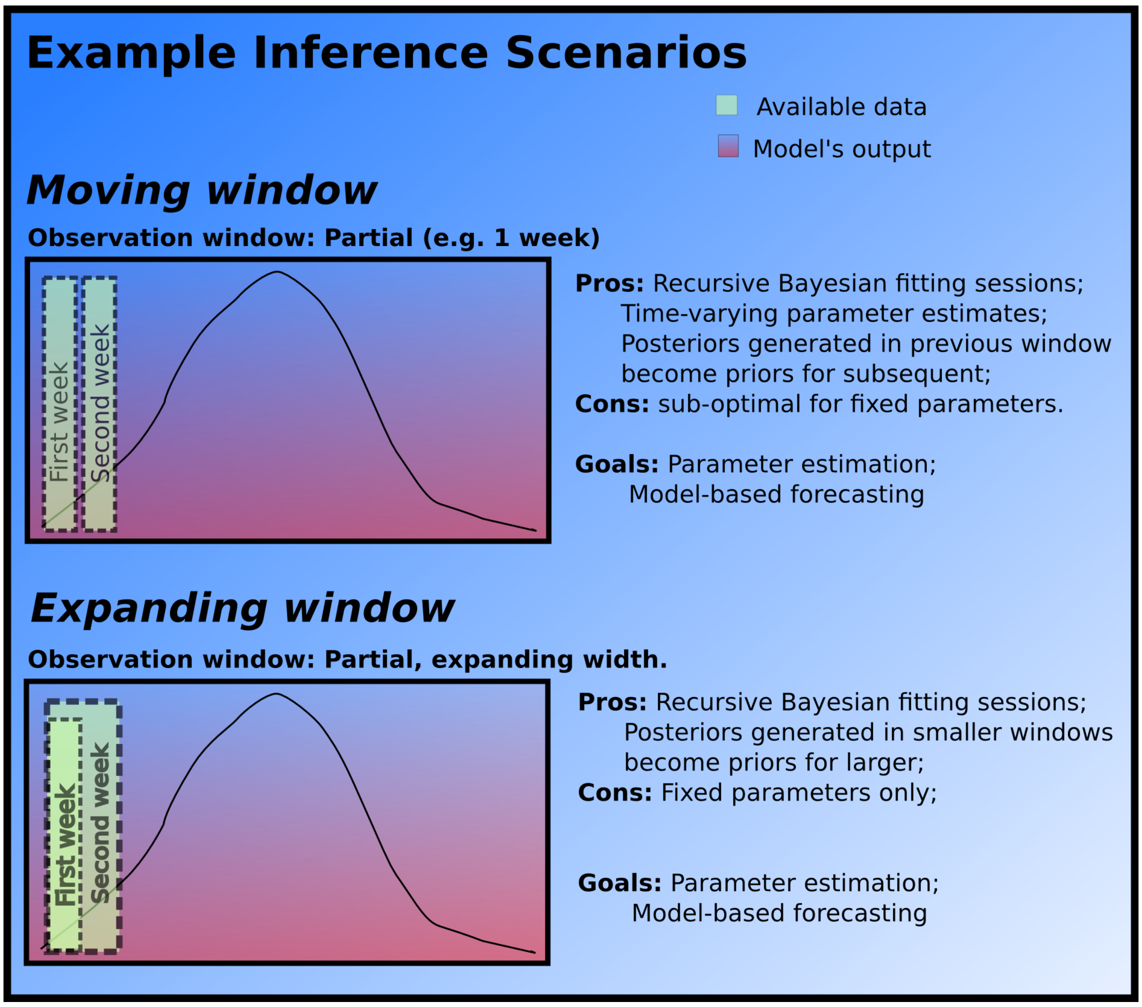
Example Inference Scenarios
🟩 Available data
🟪 Model's output
Moving window
Observation window: Partial (e.g. 1 week)
First week
Second week
Pros: Recursive Bayesian fitting sessions; Time-varying parameter estimates; Posteriors generated in previous window become priors for subsequent;
Cons: sub-optimal for fixed parameters.
Goals: Parameter estimation; Model-based forecasting
Expanding window
Observation window: Partial, expanding width.
First week
Second week
Pros: Recursive Bayesian fitting sessions; Posteriors generated in smaller windows become priors for larger;
Cons: Fixed parameters only;
Goals: Parameter estimation; Model-based forecasting

Figure 1.3: Fitting scenarios: Moving windows and expanding windows.

CHAPTER

TWO

MODULES

BIP	Bayesian Inference Package containing usefull classes and functions for doing inference in various a
BIP.Bayes	Basic Likelihood tools such as functions for computing likelihoods, Latin Hypercube sampling (effi
BIP.Bayes.Samplers	

2.1 BIP

Bayesian Inference Package containing usefull classes and functions for doing inference in various applications.

This is not a "Bayesian Statistics" package, i.e., it was not conceived to provide data analysis methods such as the one you can find on a statistical package such as R (for example).

Here, you will find some the basic building blocks those sophisticated Bayesian regression methods are built from, such as likelihood functions, MCMC samplers, SMC samplers, etc..

This package exists because such basic tools are not readily accessible in Task-oriented statistical software. From these tools, as this package matures, you will be able to easily build a solution for your own inferential inquiries, a solution which may not be available on standard statistical packages.

Classes

`RotatingFileHandler`(filename[, mode, ...])	Handler for logging to a set of files, which switches from one file to the next

2.2 BIP.Bayes

Basic Likelihood tools such as functions for computing likelihoods, Latin Hypercube sampling (efficient random sampling) and other tools which don't belong on other packages, or apply to multiple packages.

2.2.1 Melding Module

class `BIP.Bayes.Melding.`**`FitModel`**(*K*, *model*, *inits*, *tf*, *thetanames*, *phinames*, *wl=None*, *nw=1*, *verbose=False*, *burnin=1000*, *constraints=[ ]*)

Fit a model to data generating Bayesian posterior distributions of input and outputs of the model.

`AIC_from_RSS`()

Calculates the Akaike information criterion from the residual sum of squares of the best fitting run.

`do_inference`(*prior*, *data*, *predlen*, *method*, *likvar*)

Parameters

- **prior** –
- **data** –

- **predlen** –
- **method** –
- **likvar** –

optimize (*data*, *p0*, *optimizer='scipy'*, *tol=0.0001*, *verbose=0*, *plot=0*)
Finds best parameters using an optimization approach

Parameters

- *data*: Dictionary of observed series
- *p0*: Sequence (list or tuple) of initial values for the parameters
- *optimizer*: Optimization library to use: 'scipy': fmin (Nelder-Mead) or 'oo':OpenOpt.NLP
- *tol*: Tolerance of the error
- *verbose*: If true show stats of the optimization run at the end
- *plot*: If true plots a run based on the optimized parameters.

plot_results (*names=[ ]*, *dbname='results'*, *savefigs=0*)
Plot the final results of the inference

prior_sample ()
Generates a set of samples from the starting theta prior distributions for reporting purposes.

Returns Dictionary with (name,sample) pairs

run (*data*, *method*, *likvar*, *pool=False*, *adjinits=True*, *ew=0*, *dbname='results'*, *monitor=False*, *initheta=[ ]*)
Fit the model against data

Parameters

- *data*: dictionary with variable names and observed series, as Key and value respectively.
- *method*: Inference method: "ABC", "SIR", "MCMC" or "DREAM"
- *likvar*: Variance of the likelihood function in the SIR and MCMC method
- *pool*: Pool priors on model's outputs.
- *adjinits*: whether to adjust inits to data
- *ew*: Whether to use expanding windows instead of moving ones.
- *dbname*: name of the sqlite3 database
- *monitor*: Whether to monitor certains variables during the inference. If not False, should be a list of valid phi variable names.
- *initheta*: starting position in parameter space for the sampling to start. (only used by MCMC and DREAM)

set_priors (*tdists*, *tpars*, *tlims*, *pdists*, *ppars*, *plims*)
Set the prior distributions for Phi and Theta

Parameters

- *pdists*: distributions for the output variables. For example: [scipy.stats.uniform,scipy.stats.norm]
- *ppars*: paramenters for the distributions in pdists. For example: [(0,1),(0,1)]
- *plims*: Limits of the range of each phi. List of (min,max) tuples.
- *tdists*: same as pdists, but for input parameters (Theta).
- *tpars*: same as ppars, but for tdists.

- *tlims*: Limits of the range of each theta. List of (min,max) tuples.

class `BIP.Bayes.Melding.`**`Meld`**(*K*, *L*, *model*, *ntheta*, *nphi*, *alpha=0.5*, *verbose=0*, *viz=False*)
Bayesian Melding class

`abcRun`(*fitfun=None*, *data={}*, *t=1*, *pool=False*, *savetemp=False*)
Runs the model for inference through Approximate Bayes Computation techniques. This method should be used as an alternative to the sir.

Parameters

- *fitfun*: Callable which will return the goodness of fit of the model to data as a number between 0-1, with 1 meaning perfect fit
- *t*: number of time steps to retain at the end of the of the model run for fitting purposes.
- *data*: dict containing observed time series (lists of length t) of the state variables. This dict must have as many items the number of state variables, with labels matching variables names. Unorbserved variables must have an empty list as value.
- *pool*: if True, Pools the user provided priors on the model's outputs, with the model induced priors.
- *savetemp*: Should temp results be saved. Useful for long runs. Alows for resuming the simulation from last sa

`add_salt`(*dataset*, *band*)
Adds a few extra uniformly distributed data points beyond the dataset range. This is done by adding from a uniform dist.

Parameters

- *dataset*: vector of data
- *band*: Fraction of range to extend [0,1[

Returns Salted dataset.

`current_plot`(*series*, *data*, *idx*, *vars=*[ ], *step=0*)
Plots the last simulated series

Parameters

- *series*: Record array with the simulated series.
- *idx*: Integer index of the curve to plot .
- *data*: Dictionary with the full dataset.
- *vars*: List with variable names to be plotted.
- *step*: Step of the chain

`filtM`(*cond*, *x*, *limits*)
Multiple condition filtering. Remove values in x[i], if corresponding values in cond[i] are less than limits[i][0] or greater than limits[i][1].

Parameters

- *cond*: is an array of conditions.
- *limits*: is a list of tuples (ll,ul) with length equal to number of lines in *cond* and *x*.
- *x*: array to be filtered.

`getPosteriors`(*t*)
Updates the posteriors of the model's output for the last t time steps. Returns two record arrays: - The posteriors of the Theta - the posterior of Phi last t values of time-series. self.L by *t* arrays.

Parameters

- *t*: length of the posterior time-series to return.

imp_sample (*n*, *data*, *w*)
Importance sampling

Returns returns a sample of size n

logPooling (*phi*)
Returns the probability associated with each phi[i] on the pooled pdf of phi and q2phi.

Parameters

- *phi*: prior of Phi induced by the model and q1theta.

mcmc_run (*data*, *t=1*, *likvariance=10*, *burnin=1000*, *nopool=False*, *method='MH'*, *constraints=*[ ])
MCMC based fitting

Parameters

- *data*: observed time series on the model's output
- *t*: length of the observed time series
- *likvariance*: variance of the Normal likelihood function
- *nopool*: True if no priors on the outputs are available. Leads to faster calculations
- *method*: Step method. defaults to Metropolis hastings

run (**args*)
Runs the model through the Melding inference.model model is a callable which return the output of the deterministic model, i.e. the model itself. The model is run self.K times to obtain phi = M(theta).

runModel (*savetemp*, *t=1*, *k=None*)
Handles running the model k times keeping a temporary savefile for resuming calculation in case of interruption.

Parameters

- *savetemp*: Boolean. create a temp file?
- *t*: number of time steps

Returns

- self.phi: a record array of shape (k,t) with the results.

setPhi (*names*, *dists=[<scipy.stats._continuous_distns.norm_gen object at 0x411ebd0>]*, *pars=[(0, 1)]*, *limits=[(-5, 5)]*)
Setup the models Outputs, or Phi, and generate the samples from prior distributions needed for the melding replicates.

Parameters

- *names*: list of string with the names of the variables.
- *dists*: is a list of RNG from scipy.stats
- *pars*: is a list of tuples of variables for each prior distribution, respectively.
- *limits*: lower and upper limits on the support of variables.

setPhiFromData (*names*, *data*, *limits*)
Setup the model outputs and set their prior distributions from the vectors in data. This method is to be used when the prior distributions are available in the form of a sample from an empirical distribution such as a bayesian posterior. In order to expand the samples provided, K samples are generated from a kernel density estimate of the original sample.

Parameters

- *names*: list of string with the names of the variables.
- *data*: list of vectors. Samples of the proposed distribution.

- *limits*: list of tuples (ll,ul),lower and upper limits on the support of variables.

setTheta(*names*, *dists=[<scipy.stats._continuous_distns.norm_gen object at 0x411ebd0>]*, *pars=[(0, 1)]*, *lims=[(0, 1)]*)

Setup the models inputs and generate the samples from prior distributions needed for the dists the melding replicates.

Parameters

- *names*: list of string with the names of the parameters.
- *dists*: is a list of RNG from scipy.stats
- *pars*: is a list of tuples of parameters for each prior distribution, respectivelydists

setThetaFromData(*names*, *data*, *limits*)

Setup the model inputs and set the prior distributions from the vectors in data. This method is to be used when the prior distributions are available in the form of a sample from an empirical distribution such as a bayesian posterior. In order to expand the samples provided, K samples are generated from a kernel density estimate of the original sample.

Parameters

- *names*: list of string with the names of the parameters.
- *data*: list of vectors. Samples of a proposed distribution
- *limits*: List of (min,max) tuples for each theta to make sure samples are not generated outside these limits.

sir(*data={}*, *t=1*, *variance=0.1*, *pool=False*, *savetemp=False*)

Run the model output through the Sampling-Importance-Resampling algorithm. Returns 1 if successful or 0 if not.

Parameters

- *data*: observed time series on the model's output
- *t*: length of the observed time series
- *variance*: variance of the Normal likelihood function
- *pool*: False if no priors on the outputs are available. Leads to faster calculations
- *savetemp*: Boolean. create a temp file?

BIP.Bayes.Melding.**basicfit**(*s1*, *s2*)

Calculates a basic fitness calculation between a model- generated time series and a observed time series. it uses a Mean square error.

Parameters

- *s1*: model-generated time series. record array.
- *s2*: observed time series. dictionary with keys matching names of s1

Return Root mean square deviation between ´s1´ and ´s2´.

BIP.Bayes.Melding.**clearNaN**(*obs*)

Loops through an array with data series as columns, and Replaces NaNs with the mean of the other series.

Parameters

- *obs*: 2-dimensional numpy array

Returns array of the same shape as obs

BIP.Bayes.Melding.**enumRun**(*model*, *theta*, *k*)

Returns model results plus run number.

Parameters

- *model*: model callable

- *theta*: model input list
- *k*: run number

Return

- res: result list
- *k*: run number

`BIP.Bayes.Melding.`**`model`**`(`*theta*, *n=1*`)`

Model (r,p0, n=1) Simulates the Population dynamic Model (PDM) Pt = rP0 for n time steps. P0 is the initial population size. Example model for testing purposes.

`BIP.Bayes.Melding.`**`model_as_ra`**`(`*theta*, *model*, *phinames*`)`

Does a single run of self.model and returns the results as a record array

`BIP.Bayes.Melding.`**`plotRaHist`**`(`*arr*`)`

Plots a record array as a panel of histograms

`BIP.Bayes.Melding.`**`randint`**`(`*low*, *high=None*, *size=None*`)`

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

low [int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

high [int, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

size [int or tuple of ints, optional] Output shape. Default is None, in which case a single int is returned.

out [int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

random.random_integers [similar to *randint*, only for the closed] interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```

`BIP.Bayes.Melding.`**`random`**`()`

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`BIP.Bayes.Melding.`**`seed`**(*seed=None*)

Seed the generator.

This method is called when *RandomState* is initialized. It can be called again to re-seed the generator. For details, see *RandomState*.

seed [int or array_like, optional] Seed for *RandomState*.

RandomState

2.2.2 Log-Likelihood Functions

`BIP.Bayes.like.`**`Bernoulli`**(*x*, *p*)

Log-Like Bernoulli >>> Bernoulli([0,1,1,1,0,0,1,1],0.5) -5.54517744448

`BIP.Bayes.like.`**`Beta`**(*x*, *a*, *b*)

Log-Like Beta >>> Beta([.2,.3,.7,.6,.4],2,5) -0.434845728904

`BIP.Bayes.like.`**`Binomial`**(*x*, *n*, *p*)

Binomial Log-Likelihood >>> Binomial([2,3],6,0.3) -2.81280615454

`BIP.Bayes.like.`**`Categor`**(*x*, *hist*)

Categorical Log-likelihood generalization of a Bernoulli process for variables with any constant number of discrete values.

Parameters

- *x*: data vector (list)
- *hist*: tuple (prob,classes) classes contain the superior limit of the histogram classes

```
>>> Categor([1],([.3,.7],[0,1]))
-0.356674943939
```

`BIP.Bayes.like.`**`Gamma`**(*x*, *alpha*, *beta*)

Log-Like Gamma >>> Gamma([2,3,7,6,4],2,2) -11.015748357

`BIP.Bayes.like.`**`Lognormal`**(*x*, *mu*, *tau*)

Lognormal Log-likelihood

Parameters

- *mu*: mean
- *tau*: precision (1/sd)

```
>>> Lognormal([0.5,1,1.2],0,0.5)
-3.15728720569
```

`BIP.Bayes.like.`**`Negbin`**`(`*x*, *r*, *p*`)`
Negative Binomial Log-Likelihood >>> Negbin([2,3],6,0.3) -9.16117424315

`BIP.Bayes.like.`**`Normal`**`(`*x*, *mu*, *tau*`)`
Normal Log-like

Parameters

- *mu*: mean
- *tau*: precision (1/variance)

```
>>> Normal([0],0,1)
-0.918938533205
```

`BIP.Bayes.like.`**`Poisson`**`(`*x*, *mu*`)`
Poisson Log-Likelihood function >>> Poisson([2],2) -1.30685281944

`BIP.Bayes.like.`**`Simple`**`(`*x*, *w*, *a*, *start=0*`)`
find out what it is.

`BIP.Bayes.like.`**`Uniform`**`(`*x*, *min*, *max*`)`
Uniform Log-likelihood

Parameters

- *x*: data vector(list)
- *min*: lower limit of the distribution
- *max*: upper limit of the distribution

```
>>> Uniform([1.1,2.3,3.4,4],0,5)
-6.4377516497364011
```

`BIP.Bayes.like.`**`Weibull`**`(`*x*, *alpha*, *beta*`)`
Log-Like Weibull >>> Weibull([2,1,0.3,.5,1.7],1.5,3) -7.811955373

`BIP.Bayes.like.`**`find_best_tau`**`(`*x*, *mu*`)`
returns the value of tau which maximizes normal loglik for a fixed (x,mu)

2.2.3 Plotting tools

Module with specialized plotting functions for the Melding results

`BIP.Bayes.PlotMeld.`**`peakdet`**`(`*v*, *delta*, *x=None*`)`
Converted from MATLAB script at http://billauer.co.il/peakdet.html Currently returns two lists of tuples, but maybe arrays would be better function [maxtab, mintab]=peakdet(v, delta, x) %PEAKDET Detect peaks in a vector % [MAXTAB, MINTAB] = PEAKDET(V, DELTA) finds the local % maxima and minima ("peaks") in the vector V. % MAXTAB and MINTAB consists of two columns. Column 1 % contains indices in V, and column 2 the found values. % % With [MAXTAB, MINTAB] = PEAKDET(V, DELTA, X) the indices % in MAXTAB and MINTAB are replaced with the corresponding % X-values. % % A point is considered a maximum peak if it has the maximal % value, and was preceded (to the left) by a value lower by % DELTA. % Eli Billauer, 3.4.05 (Explicitly not copyrighted). % This function is released to the public domain; Any use is allowed.

`BIP.Bayes.PlotMeld.`**`pred_new_cases`**`(`*obs*, *series*, *weeks*, *names=[ ]*, *title='Total new cases per window: predicted vs observed'*, *ws=7*`)`
Predicted total new cases in a window vs oserved.

`BIP.Bayes.PlotMeld.`**`violin_plot`**`(`*ax*, *data*, *positions*, *bp=False*, *prior=False*`)`
Create violin plots on an axis

Parameters

- *ax*: A subplot object

- *data*: A list of data sets to plot
- *positions*: x values to position the violins. Can be datetime.date objects.
- *bp*: Whether to plot the boxplot on top.
- *prior*: whether the first element of data is a Prior distribution.

2.2.4 Latin Hypercube Sampling

Module with specialized plotting functions for the Melding results

`BIP.Bayes.PlotMeld.`**`peakdet`**(*v*, *delta*, *x=None*)

Converted from MATLAB script at http://billauer.co.il/peakdet.html Currently returns two lists of tuples, but maybe arrays would be better function [maxtab, mintab]=peakdet(v, delta, x) %PEAKDET Detect peaks in a vector % [MAXTAB, MINTAB] = PEAKDET(V, DELTA) finds the local % maxima and minima ("peaks") in the vector V. % MAXTAB and MINTAB consists of two columns. Column 1 % contains indices in V, and column 2 the found values. % % With [MAXTAB, MINTAB] = PEAKDET(V, DELTA, X) the indices % in MAXTAB and MINTAB are replaced with the corresponding % X-values. % % A point is considered a maximum peak if it has the maximal % value, and was preceded (to the left) by a value lower by % DELTA. % Eli Billauer, 3.4.05 (Explicitly not copyrighted). % This function is released to the public domain; Any use is allowed.

`BIP.Bayes.PlotMeld.`**`pred_new_cases`**(*obs*, *series*, *weeks*, *names=[ ]*, *title='Total new cases per window: predicted vs observed'*, *ws=7*)

Predicted total new cases in a window vs oserved.

`BIP.Bayes.PlotMeld.`**`violin_plot`**(*ax*, *data*, *positions*, *bp=False*, *prior=False*)

Create violin plots on an axis

Parameters

- *ax*: A subplot object
- *data*: A list of data sets to plot
- *positions*: x values to position the violins. Can be datetime.date objects.
- *bp*: Whether to plot the boxplot on top.
- *prior*: whether the first element of data is a Prior distribution.

2.3 BIP.Bayes.Samplers

2.3.1 MCMC

Module implementing MCMC samplers

- Metropolis: Adaptive Metropolis Hastings sampler
- Dream: DiffeRential Evolution Adaptive Markov chain sampler

class `BIP.Bayes.Samplers.MCMC.`**`Dream`**(*meldobj*, *samples*, *sampmax*, *data*, *t*, *parpriors*, *parnames*, *parlimits*, *likfun*, *likvariance*, *burnin*, *thin=5*, *convergenceCriteria=1.1*, *nCR=3*, *DEpairs=1*, *adaptationRate=0.65*, *eps=5e-06*, *mConvergence=False*, *mAccept=False*, ***kwargs*)

DiffeRential Evolution Adaptive Markov chain sampler

`delayed_rejection`(*xi*, *zi*, *pxi*, *zprob*)

Generates a second proposal based on rejected proposal xi

`step`()

Does the actual sampling loop.

class `BIP.Bayes.Samplers.MCMC.`**`Metropolis`**(*meldobj*, *samples*, *sampmax*, *data*, *t*, *parpriors*, *parnames*, *parlimits*, *likfun*, *likvariance*, *burnin*, ***kwargs*)

Standard random-walk Metropolis Hastings sampler class

`step`(*nchains=1*)

Does the actual sampling loop.

`BIP.Bayes.Samplers.MCMC.`**`model_as_ra`**(*theta*, *model*, *phinames*)

Does a single run of self.model and returns the results as a record array

`BIP.Bayes.Samplers.MCMC.`**`multinomial`**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalisation of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

n [int] Number of experiments.

pvals [sequence of floats, length p] Probabilities of each of the `p` different outcomes. These should sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

size [tuple of ints] Given a *size* of `(M, N, K)`, then `M*N*K` samples are drawn, and the output shape becomes `(M, N, K, p)`, since each sample has shape `(p,)`.

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded dice is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5)
array([13, 16, 13, 16, 42])
```

`BIP.Bayes.Samplers.MCMC.`**`multivariate_normal`**(*mean*, *cov*[, *size*])

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. Must be symmetric and positive semi-definite for "physically meaningful" results.

size [int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0,0]
>>> cov = [[1,0],[0,100]] # diagonal covariance, points lie on x or y-axis

>>> import matplotlib.pyplot as plt
>>> x,y = np.random.multivariate_normal(mean,cov,5000).T
>>> plt.plot(x,y,'x'); plt.axis('equal'); plt.show()
```

Note that the covariance matrix must be non-negative definite.

Papoulis, A., *Probability, Random Variables, and Stochastic Processes*, 3rd ed., New York: McGraw-Hill, 1991.

Duda, R. O., Hart, P. E., and Stork, D. G., *Pattern Classification*, 2nd ed., New York: Wiley, 2001.

```
>>> mean = (1,2)
>>> cov = [[1,0],[1,0]]
>>> x = np.random.multivariate_normal(mean,cov,(3,3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> print list( (x[0,0,:] - mean) < 0.6 )
[True, True]
```

`BIP.Bayes.Samplers.MCMC.`**`rand`**`(`*d0, d1, ..., dn*`)`

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, ..., dn [int, optional] The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to np.random.random_sample .

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

`BIP.Bayes.Samplers.MCMC.`**`random`**`()`

random_sample(size=None)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

size [int or tuple of ints, optional] Defines the shape of the returned array of random floats. If None (the default), returns a single float.

out [float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

```
>>> np.random.random_sample()
0.47108547995356098
>>> type(np.random.random_sample())
<type 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428])
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984],
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`BIP.Bayes.Samplers.MCMC.`**`timeit`**`(`*method*`)`

Decorator to time methods

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX