# SDE Documentation

***Release 0.3.0***

**M. Schauer**

**Jul 05, 2017**

# Contents

CHAPTER 1

# Layout

The package contains the following modules:

***Diffusion*** Generate Ito processes and diffusions

***NonparBayes*** Nonparametrically estimate the drift of a diffusion

***Schauder*** Provides rudimentary finite element methods and Schauder basis for NonparBayes

***Lyap*** Computes the solution of the continuous Lyapunov equation, useful for the generation of linear processes

***Randm*** Random symmetric, positive definite, stable matrix for testing purposes.

***LinProc*** Homogeneous vector linear processes with additive noise

CHAPTER 2

# Method

The producure in `NonparBayes` is a Julia implementation of nonparametric Bayesian inference for "continuously" observed one dimensional diffusion processes with unit diffusion coefficient. The drift is modeled as linear combination of hierarchical Faber–Schauder basis functions with a Gaussian prior on the coefficients. This incorporates a Brownian motion like prior on the drift function. The posterior is then computed using Gaussian conjugacy.

This is work in progress.

CHAPTER 3

# Data structures

I did not introduce type definitions for stochastic processes and use vectors/arrays, so it should be easy do wrap Dataframes around everything. For the meanwhile, I like the natural notation obtained by having just vectors/arrays for dW and dt

```
N = 100 t = linspace(0., 1., N) dt = diff(t) X = ito(2dt + 2dW1(dt))
```

CHAPTER 4

# Location of the documentation

https://sdejl.readthedocs.org

Contents:

## Module Diffusion

### Introduction

The functions in this module operate on three conceptual different objects, (although they are currently just represented as vectors and arrays.)

Stochastic processes, denoted `x`, `y`, `w` are arrays of values which are sampled at distance `dt`, `ds`, where `dt`, `ds` are either scalar or Vectors `length(dt)=size(W)[end]`. Stochastic differentials are denoted `dx`, `dw` etc., and are first differences of stochastic processes. Finally, t can denote the total time or correspond to a vector of `size(W)[end]` sampling time poins.

Note the following convention: In analogy with the definition of the Ito integral,

> intxdw[i] = x[i]](w[i+1]-w[i]) (== x[i]dw[i])

and

> length(w) = length(dw) + 1

### Reference

`Diffusion.`**`brown1`**(*u*, *t*, *n::Integer*)
: Compute `n` equally spaced samples of 1d Brownian motion in the interval `[0,t]`, starting from point `u`

`Diffusion.`**`brown`**(*u*, *t*, *d::Integer*, *n::Integer*)
: Simulate `n` equally spaced samples of `d`-dimensional Brownian motion in the interval `[0,t]`, starting from point `u`

`Diffusion.`**`dW1`**(*t*, *n::Integer*)
`Diffusion.`**`dW`**(*t*, *d::Integer*, *n::Integer*)
Simulate a `1`-dimensional (`d`-dimensional) Wiener differential with `n` values in the the interval `[0,t]`, starting from point `u`

`Diffusion.`**`dW`**(*dt::Vector*, *d::Integer*)
Simulate a `d`-dimensional Wiener differential sampled at time points with distances given by the vector `dt`

`Diffusion.`**`ito`**(*y*, *dx*)
`Diffusion.`**`ito`**(*dx*)
`Diffusion.`**`cumsum0`**(*dx*)
Integrate a valued stochastic process with respect to a stochastic differential. R, R^2 (d rows, n columns), R^3.

`ito(dx)` is a shortcut for `ito(ones(size(dx)[end], dx)`. So `ito(dx)` is just a `cumsum0` function which is a inverse to `dx = diff([0, x1, x2, x3,...])`.

`..`(*y*, *dx*)
`Diffusion.`**`ydx`**(*y*, *dx*)
`y .. dx` returns the stochastic differential `ydx` defined by the property

ito(ydx) == ito(y, dx)

`Diffusion.`**`bb`**(*u*, *v*, *t*, *n*)
Simulates `n` equidistant samples of a Brownian bridge from point `u` to `v` in time `t`

`Diffusion.`**`dWcond1`**(*v*, *t*, *n*)
Simulates `n` equidistant samples of a "bridge noise": that is a Wiener differential `dW` conditioned on `W(t) = v`

`Diffusion.`**`aug`**(*dw*, *dt*, *n*)
`Diffusion.`**`aug`**(*dt*, *n*)
Take Wiener differential sampled at `dt` and return Wiener differential subsampled `n` times between each observation with new length `length(dw)*n`. `aug(dt,n)` computes the corresponding subsample of times.

`Diffusion.`**`quvar`**(*x*)
Computes quadratic variation of `x`.

`Diffusion.`**`bracket`**(*x*)
`Diffusion.`**`bracket`**(*x*, *y*)
Computes quadratic variation process of `x` (of `x` and `y`).

`Diffusion.`**`euler`**(*t0*, *u*, *b*, *sigma*, *dt*, *dw*)
`Diffusion.`**`euler`**(*t0*, *u*, *b*, *sigma*, *dt*)
Simulates a 1-dimensional diffusion process using the Euler-Maruyama approximation with drift `b(t,x)` and diffusion coefficient `sigma(t,x)` starting in `(t0, u)` using `dt` and given Wiener differential `dw`.

# Module Schauder

## Introduction

In the following `hat(x)` is the piecewise linear function taking values values `(0,0), (0.5,1), (1,0)` on the interval `[0,1]` and `0` elsewhere.

The Schauder basis of level `L > 0` in the interval `[a,b]` can be defined recursively from `n = 2^L-1` classical finite elements $\psi_i(x)$ on the grid

```
a + (1:n)/(n+1)*(b-a).
```

Assume that f is expressed as linear combination

$$f(x) = \sum_{i=1}^{n} c_i \psi_i(x)$$

with

$$\psi_{2j-1}(x) = hat(nx - j + 1) \text{ for } j = 1 \ldots 2^{L-1}$$

and

$$\psi_{2j}(x) = hat(nx - j + 1/2) \text{ for } j = 1 \ldots 2^{L-1} - 1$$

Note that these coefficients are easy to find for the finite element basis, just

```
function fe_transf(f, a,b, L)
    n = 2^L-1
    return map(f, a + (1:n)/(n+1)*(b-a))
end
```

Then the coefficients of the same function with respect to the Schauder basis

$$f(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

where for `L = 2`

$$\phi_2(x) = 2hat(x)$$

$$\phi_1(x) = \psi_1(x) = hat(2x)$$

$$\phi_3(x) = \psi_3(x) = hat(2x - 1)$$

can be computed directly, but also using the recursion

$$\phi_2(x) = \psi_1(x) + 2\psi_2(x) + \psi_2(x)$$

This can be implemented inplace (see `pickup()`), and is used throughout.

```
for l in 1:L-1
        b = sub(c, 2^l:2^l:n)
        b[:] *= 0.5
        a = sub(c, 2^(l-1):2^l:n-2^(l-1))
        a[:] -= b[:]
        a = sub(c, 2^l+2^(l-1):2^l:n)
        a[:] -= b[1:length(a)]
end
```

## Reference

`Schauder.`**`pickup!`**`(x)`

Inplace computation of 2^L-1 Schauder-Faber coefficients from `2^L-1` overlapping finite-element coefficients `x`.

– inverse of `Schauder.drop`

– L = level(xj)

`Schauder.`**`drop!`**`(x)`

Inplace computation of 2^L-1 finite element coefficients from 2^L-1 Faber schauder coefficients `x`.

– inverse of `Schauder.pickup`

`Schauder.`**`finger_permute`**`(x)`

Reorders vector `x` or matrix `A` according to the reordering of the elements of a Faber-Schauder-basis from left to right, from bottom to top.

`Schauder.`**`finger_pm`**(*L*, *K*)
Returns the permuation used in `finger_permute`. Memoized reordering of faber schauder elements from low level to high level. The last K elements/rows are left untouched.

`Schauder.`**`level`**(*x*)

**Gives the no. of levels of the biggest Schauder basis with less then length(x) elements.** level(x) = ilogb(size(x,1)+1)

`Schauder.`**`level`**(*x*, *K*)
Gives the no. of levels `l` of the biggest Schauder basis with less then length(x) elements and the number of additional elements `n-2^l+1`.

`Schauder.`**`vectoroflevels`**(*L*, *K*)
Gives a vector with the level of the hierarchical elements.

`Schauder.`**`hat`**(*x*)
Hat function. Piecewise linear functions with values (-inf,0), (0,0),(0.5,1), (1,0), (inf,0). – `x` vector or number

# Module NonparBayes

## Introduction

The procedure is as follows. Consider the diffusion process $(x_t : 0 \leq t \leq T)$ given by

$$dx_t = b(x_t)dt + dw_t$$

where the drift `b` is expressed as linear combination

$$f(x) = \sum_{i=1}^{n} c_i \phi_i(x)$$

(see *Module Schauder*) and prior distribution on the coefficients

$$c_i \sim N(0, \xi_i)$$

Then the posterior distribution of $b$ given observations $x_t$ is given by

$$c_i | x_s \sim N(W^{-1}\mu, W^{-1})\ W = \Sigma + (\mathrm{diag}(\xi))^{-1},$$

with the nxn-matrix

$$\Sigma_{ij} = \int_0^T \phi_i(x_t)\phi_j(x_t)dt$$

and the n-vector

$$\mu_i = \int_0^T \phi_i(x_t)dx_t.$$

Using the recursion detailed in *Module Schauder*, one rather computes

$$\Sigma'_{ij} = \int_0^T \psi_i(x_t)\psi_j(x_t)dt$$

and the n-vector

$$\mu'_i = \int_0^T \psi_i(x_t)dx_t$$

and uses `pickup_mu!(mu)` and `pickup_Sigma!(Sigma)` to obtain $\mu$ and $\Sigma$.

## Optional additional basis functions

One can extend the basis by additional functions, implemented are variants. `B1` includes a constant, `B2` two linear functions

**B1** $\phi_1 \dots \phi_n, c$

**B2** $\phi_1 \dots \phi_n, \max(1-x,0), \max(x,0)$

To compute `mu`, use

```
mu = pickup_mu!(fe_mu(y,L, 0))
mu = fe_muB1(mu, y);
```

or

```
mu = pickup_mu!(fe_mu(y,L, 0))
mu = fe_muB2(mu, y);
```

## Reference

Functions taking `y` without parameter [a,b] expect ``y` to be shifted into the intervall `[0,1]`.

`NonparBayes.`**`pickup_mu!`**(*mu*)
computes mu from mu'

`NonparBayes.`**`drop_mu!`**(*mu*)
Computes mu' from mu.

`NonparBayes.`**`pickup_Sigma!`**(*Sigma*)
Transforms Sigma' into Sigma.

`NonparBayes.`**`drop_Sigma!`**(*Sigma*)
Transforms Sigma into Sigma'.

`NonparBayes.`**`fe_mu`**(*y*, *L*, *K*)
Computes mu' from the observations *y* using `2^L-1` basis elements and returns a vector with `K` trailing zeros (in case one ones to customize the basis.

`NonparBayes.`**`fe_muB1`**(*mu*, *y*)
Append $\mu_{n+1} = \int_0^T \phi_{n+1} dx_t$ with $\phi_{n+1} = 1$.

`NonparBayes.`**`fe_muB2`**(*mu*, *y*)
Append $\mu_{n+1} = \int_0^T \phi_{n+1} dx_t$ with $\phi_{n+1} = \max(1-x,0)$ and $\mu_{n+2} = \int_0^T \phi_{n+2} dx_t$ with $\phi_{n+2} = \max(x,0)$

`NonparBayes.`**`fe_Sigma`**(*y*, *dt*, *L*)
Computes the matrix Sigma' from the observations *y* uniformly spaced at distance `dt` using `2^L-1` basis elements.

`NonparBayes.`**`bayes_drift`**(*x*, *dt*, *a*, *b*, *L*, *xirem*, *beta*, *B*)
Performs estimation of drift on observations `x` in [a,b] spaced at distance `dt` using the Schauder basis of level `L` and level wise coefficients decaying at rate `beta`. A Brownian motion like prior is obtained for beta= 0.5. The `K` remaining optional basiselements have variance `xirem`.

The result is returned as `[designp coeff se]` where `coeff` are coefficients of finite elements with maximum at the designpoints `designp` and standard error `se`.

Observations outside [a,b] may influence the result through `phi_{n+1}, ..., phi_{n+K}`

`NonparBayes.`**`visualize_posterior`**(*post*[, *truedrift* ])
Plot 2r*se wide marginal credibility bands, where `post` is the result of bayes_drift and truedrift the true drift (if known :-) ).

# Module Lyap.jl

## Documentation

**DESCRIPTION** Solves the real matrix equation A'X + XA = C, where A and C are constant matrices of dimension n x n with C=C'. The matrix A is transformed into upper Schur form and the transformed system is solved by back substitution. The option is provided to input the Schur form directly and bypass the Schur decomposition. This equation is also know as continuous Lyapunov equation.

The method of Bartels and Stewart is used. The system is first reduced such that A is in upper real schur form. The resulting triangular system is solved via back-substitution. Has a unique solution, if A and -A have no common eigenvalues, which is guaranteed if A is stable (and the real part of each eigenvalue is negative).

**HISTORY** The classic ACM algorithm from Bartels and Stewart was implemented E. Armstrong as part of ORACLS – optimal regulator algorithms for the control of linear systems. The implementation from the nasa cosmic archive is reported to be in the public domain, under the terms of Title 17, Chapter 1, Section 105 of the US Code. This is rather direct translation of forementioned implementation into Julia, put under MIT licence as Julia.

**REFERENCES**

- Bartels, R.H.; and Stewart, G.W.: Algorithm 432 - Solution of the Matrix Equation AX + XB = C. Commun. ACM, vol. 15, no. 9, Sept. 1972, pp. 820-826.

**SEE ALSO** atxpxa in ORACLS, strsyl, dtrsyl in LAPACK, lyap in GNU Octave

## Reference

`Lyap.`**`issquare`**(*a*)
Checks if matrix `a` is square.

`Lyap.`**`lyap`**(*a*, *cc*)
Solves the real matrix equation A'X + XA = C, where A and C are constant matrices of dimension n x n with C=C'. The matrix A is transformed into upper Schur form and the transformed system is solved by back substitution. The option is provided to input the Schur form directly and bypass the Schur decomposition. This equation is also know as continuous Lyapunov equation.

The method of Bartels and Stewart is used. The system is first reduced such that A is in upper real schur form. The resulting triangular system is solved via back-substitution. Has a unique solution, if A and -A have no common eigenvalues, which is guaranteed if A is stable (and the real part of each eigenvalue is negative).

`Lyap.`**`symslv`**(*a*, *c*)
Solves `A'*x + x*A = C`, where `C` is symmetric and `A` is in upper real schur form. via back substitution

`Lyap.`**`syl`**(*a*, *b*, *c*)
Solves the Sylvester equation `AX + XB = C`, where `C` is symmetric and `A` and `−B` have no common eigenvalues using (inefficient) algebraic approach via the Kronecker product, see http://en.wikipedia.org/wiki/Sylvester_equation

# Module Randm

## Introduction

Random matrices for testing purposes. I did not figure out the actual distributions the matrices are drawn from.

### Reference

`Randm.`**`randposdef`**`(`*d*`)`
Random positive definite matrix of dimension `d`.

`Randm.`**`randstable`**`(`*d*`)`
Random stable matrix (matrix with eigenvalues with negative real part) with dimension `d`.

`Randm.`**`randunitary`**`(`*d*`)`
Random unitary matrix of dimension `d`.

`Randm.`**`randorth`**`(`*d*`)`
Orthogonal matrix drawn according to the Haar measure on the group of orthogonal matrices.

`Randm.`**`randnormal`**`(`*d*`)`
Random normal matrix.

## LinProc

### Introduction

This module covers

- The simulation of multivariate diffusion processes with a simple Euler scheme
- the simulation of (Vector) Ornstein–Uhlenbeck processes
- the simulation of Ornstein–Uhlenbeck bridges
- mean and covariance functions and transition density of Ornstein–Uhlenbeck processes
- the Monte Carlo simulation of Diffusion bridges

A multivariate *diffusion process* is the solution to the stochastic differential equation (SDE)

$$dX_t = b(t, X_t)\sigma(t, X_t)dW_t$$

where `W_t` is a `d`-dimension Wiener process, `b` is a vector valued *drift* function and `a = sigma sigma'` the *diffusion* matrix. The function

eulerv(t0, u, b(s,x), sigma(s,x), Dt, DW::Matrix)

implements the multivariate euler scheme, starting in the point `u` the same conventions as in module `Diffusion` apply.

A *vector linear process* (Ornstein–Uhlenbeck process) is of the special form

$$dX_t = BX_t + \beta + \sigma dW_t$$

where `B` is a stable matrix and `beta` a vector, `A = sigma sigma'`

A *conditional vector linear processes* (Ornstein–Uhlenbeck bridges so to say) ending at time `T` in point `v` are given by

$$dX_t^\star = BX_t + \beta + Ar(s, X_t^\star) + \sigma dW_t$$

where $r(t, x) = \mathrm{grad}_x \log p(t, x; T, v)$ and `p` is the transition density of the corresponding linear process `X`.

The parameter `lambda` is the solution to the Lyapunov equation `B lambda + lambda B' = -A`, see module `Lyap`,

```
lambda = lyap(B', -A)
```

If `B = 0`, `lambda = lyap(B',-A)` is not defined, provide `lambda = inv(a)` as argument to the functions instead.

## Reference

`LinProc.`**`mu`**(*h*, *x*, *B*, *beta*)
Expectation $E_x(X_t)$

`LinProc.`**`K`**(*h*, *B*, *lambda*)
Covariance matrix $Cov(X_t, X_{t+h})$

`LinProc.`**`r`**(*h*, *x*, *v*, *B*, *beta*, *lambda*)
Returns $r(t,x) = \mathrm{grad}_x \log p(t,x;T,v)$ where `p` is the transition density of the linear process, used by `Bstar`.

`LinProc.`**`H`**(*h*, *B*, *lambda*)
Negative Hessian of $\log p(t,x;T,v)$ as a function of `x`.

`LinProc.`**`bstar`**(*T*, *v*, *b*, *beta*, *a*, *lambda*)
Returns the drift function of a vector linear process bridge which end at time T in point v.

`LinProc.`**`bcirc`**(*T*, *v*, *b*, *beta*, *a*, *lambda*)
Drift for guided proposal derived from a vector linear process bridge which end at time T in point v.

`LinProc.`**`llikeliXcirc`**(*t*, *T*, *Xcirc*, *b*, *a*, *B*, *beta*, *lambda*)
Loglikelihood (log weights) of Xcirc with respect to Xstar.

> t, T – timespan Xcirc – bridge proposal (drift Bcirc and diffusion coefficient sigma) b, sigma – diffusion coefficient sigma target B, beta – drift b(x) = Bx + beta of Xtilde lambda – solution of the lyapunov equation for Xtilde

`LinProc.`**`lp`**(*h*, *x*, *y*, *b*, *beta*, *lambda*)
Returns $logp(t,x;T,y)$, the log transition density of the linear process, h = T - t

`LinProc.`**`sample_p`**(*h*, *x*, *b*, *beta*, *lambda*)
Samples from the transition density of the linear process, h = T - t.

`LinProc.`**`linexact`**(*u*, *B*, *beta*, *lambda*, *dt*)
Simulate linear process starting in *u* on a discrete grid *dt* from its transition probability, corresponding to drift parameters *B*, *beta* and Lyapunov matrix *lambda*.

`LinProc.`**`linll`**(*X*, *B*, *beta*, *lambda*, *dt*)
Compute log likelihood evaluated in *B*, *beta* and Lyapunov matrix *lambda* for a observed linear process on a discrete grid *dt* from its transition density.

`LinProc.`**`lp0`**(*h*, *x*, *y*, *mu*, *gamma*)
Returns $logp(t,x;T,y)$, the log transition density of a Brownian motion with drift mu and diffusion a=inv(gamma), h = T - t

`LinProc.`**`sample_p0`**(*h*, *x*, *mu*, *l*)
Samples from the transition density a affine Brownian motion. Takes the Cholesky factor as argument.

> l = chol(a)

`LinProc.`**`eulerv`**(*t0*, *u*, *v*, *b(s, x)*, *sigma(s, x)*, *Dt*, *DW::Matrix*)
`LinProc.`**`eulerv`**(*t0*, *u*, *b*, *sigma*, *dt*, *dw::Matrix)* = *eulerv(t0*, *u*, *NaN*, *b*, *sigma*, *dt*, *dw::Matrix*)
Multivariate euler scheme, starting in u, fixing X[N] = v if v!=NaN (this makes sense, if b pulls X towards v.). `dw` – Wiener differential with `n` values in the the interval `[t0,sum(dt)]` sampled at timepoints `t0+Dt[1], t0 + Dt[1] + Dt[2], ...` `b, sigma` – drift and diffusion coefficient.

Example:

Dt = diff(linspace(0., T, N)) DW = randn(2, N-1) .* sqrt(dt) dt = Dt[1] yy = euler(0.0, u, b, sigma, Dt, DW)

LinProc.**stable**(*Y*, *d*, *ep*)

Return real stable *d*-dim matrix with real eigenvalues smaller than *ep* parametrized with a vector of length *d*d*,

For maximum likelihood estimation we need to search the maximum over all stable matrices. These are matrices with eigenvalues with strictly negative real parts. We obtain a dxd stable matrix as difference of a antisymmetric matrix and a positive definite matrix.

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## H



## I



## K



## L



## M



## P



## Q



## R

## S



## V



## Y