Package 'dynemu'

March 28, 2025

Title Emulation of Dynamic Simulators via One-Step-Ahead Approach

Version 1.0.0

Maintainer Junoh Heo <heojunoh@msu.edu>

Description Performs emulation of dynamic simulators using Gaussian process via one-step ahead approach. The package implements a flexible framework for approximating time-dependent outputs from computationally expensive dynamic systems. It is specifically designed for nonlinear dynamic systems where full simulations may be costly. The underlying Gaussian process model accounts for temporal dependency through the one-step-ahead formulation, allowing for accurate emulation of complex dynamics. Hyperparameters are estimated via maximum likelihood. See Heo (2025, <doi:10.48550/arXiv.2503.20250>) for exact method, and Mohammadi, Challenor, and Goodfellow (2019, <doi:10.1016/j.csda.2019.05.006>) for methodological details.

License MIT + file LICENSE

Encoding UTF-8

Imports deSolve, plgp, stats, utils, MASS

Suggests lhs

RoxygenNote 7.3.2

NeedsCompilation no

Author Junoh Heo [aut, cre]

Repository CRAN

Date/Publication 2025-03-28 18:50:01 UTC

Contents

`dynemu_exact` *Predictive posterior computation via exact closed-form expression for one-steap-ahead Gaussian process (GP) emulators*

Description

The function computes the predictive posterior distribution (mean and variance) for GP emulators using closed-form expression, given uncertain inputs.

Usage

```
dynemu_exact(mean, var, dynGP)
```

Arguments

`mean`	numeric vector or matrix specifying the mean of uncertain inputs. The number of columns must match the input dimension of 'dynGP'.
`var`	numeric vector or matrix specifying the variance of uncertain inputs. The number of columns must match the input dimension of 'dynGP'.
`dynGP`	list of trained GP emulators fitted by `dynemu_GP`, each corresponding to a state variable.

Details

Given a trained set of GP emulators 'dynGP' fitted using `dynemu_GP`, this function: 1. Computes the closed-form predictive posterior mean and variance for each state variable. 2. Incorporates input uncertainty by integrating over the input distribution via exact computation.

The computation follows a closed-form approach, leveraging the posterior distributions of Linked GP.

Value

A list containing:

- `predy`: A single-row matrix of predictive mean values, with each column for corresponding state variable.
- `predsig2`: A single-row matrix of predictive variance values, with each column for corresponding state variable.

Examples

```
library(lhs)
### Lotka-Volterra equations ###
LVmod0D <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    IngestC <- rI * P * C
    GrowthP <- rG * P * (1 - P/K)
```

```
    MortC <- rM * C

    dP <- GrowthP - IngestC
    dC <- IngestC * AE - MortC
    return(list(c(dP, dC)))
  })
}

### Define parameters ###
pars <- c(rI = 1.5, rG = 1.5, rM = 2, AE = 1, K = 10)

### Define time sequence ###
times <- seq(0, 30, by = 0.01)

### Initial conditions ###
set.seed(1)
d <- 2
n <- 12*d
Design <- maximinLHS(n, d) * 5 # Generate LHS samples in range [0,5]
colnames(Design) <- c("P", "C")

### Fit GP emulators ###
fit.dyn <- dynemu_GP(LVmod0D, times, pars, Design)

### Define uncertain inputs ###
xmean <- c(P = 1, C = 2)
xvar <- c(P = 1e-5, C = 1e-5)

### Compute the next point ###
dynemu_exact(xmean, xvar, fit.dyn)
```

---

dynemu_GP *Fit Gaussian process (GP) emulators for One-step-ahead approach*

Description

The function fits GP emulators for each state variable by solving an ODE model one step ahead and using the generated outputs as training data.

Usage

```
dynemu_GP(odefun, times, param, X)
```

Arguments

odefun	function defining the ODE system. It should return a list where the first element is a numeric vector of derivatives.
times	numeric vector of time points where the solution is computed. Only the first two time points are used for one-step-ahead prediction.

param	numeric scalar or vector of parameters to be passed to odefun.
X	A matrix where each row represents an initial condition for the ODE system. Column names must be provided to match the state variables in odefun.

Details

Given an initial set of input conditions, this function: 1. Solves the ODE system for a short time interval using dyn_solve. 2. Fits GP emulators to model the next-step evolution of each state variable.

The ODE system is defined by the user through odefun, and the GP emulators are trained using the generated one-step-ahead outputs.

Value

A list of GP emulators, each corresponding to a state variable.

Examples

```
library(lhs)
### Lotka-Volterra equations ###
LVmod0D <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    IngestC <- rI * P * C
    GrowthP <- rG * P * (1 - P/K)
    MortC <- rM * C

    dP <- GrowthP - IngestC
    dC <- IngestC * AE - MortC
    return(list(c(dP, dC)))
  })
}

### Define parameters ###
pars <- c(rI = 1.5, rG = 1.5, rM = 2, AE = 1, K = 10)

### Define time sequence ###
times <- seq(0, 30, by = 0.01)

### Initial conditions ###
set.seed(1)
d <- 2
n <- 12*d
Design <- maximinLHS(n, d) * 5 # Generate LHS samples in range [0,5]
colnames(Design) <- c("P", "C")

### Fit GP emulators ###
fit.dyn <- dynemu_GP(LVmod0D, times, pars, Design)
print(fit.dyn)
```

dynemu_mc *Predictive posterior computation via Monte Carlo (MC) approximation for one-steap-ahead Gaussian process (GP) emulators*

Description

The function computes the predictive posterior distribution (mean and variance) for GP emulators using MC method, given uncertain inputs.

Usage

```
dynemu_mc(mean, var, dynGP, mc.sample)
```

Arguments

mean	numeric vector or matrix specifying the mean of uncertain inputs. The number of columns must match the input dimension of 'dynGP'.
var	numeric vector or matrix specifying the variance of uncertain inputs. The number of columns must match the input dimension of 'dynGP'.
dynGP	list of trained GP emulators fitted by dynemu_GP, each corresponding to a state variable.
mc.sample	a number of mc samples generated for MC approximation. Default is 1000.

Details

Given a trained set of GP emulators 'dynGP' fitted using dynemu_GP, this function: 1. Computes the MC-approximated predictive posterior mean and variance for each state variable. 2. Incorporates input uncertainty by integrating over the input distribution via MC sampling.

Unlike dynemu_exact, which provides a closed-form solution, this function uses MC sampling to approximate the posterior.

Value

A list containing:

- predy: A single-row matrix of predictive mean values, with each column for corresponding state variable.
- predsig2: A single-row matrix of predictive variance values, with each column for corresponding state variable.

References

H. Mohammadi, P. Challenor, and M. Goodfellow (2019). Emulating dynamic non-linear simulators using Gaussian processes. *Computational Statistics & Data Analysis*, 139, 178-196; doi:10.1016/j.csda.2019.05.006

Examples

```
library(lhs)
### Lotka-Volterra equations ###
LVmod0D <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    IngestC <- rI * P * C
    GrowthP <- rG * P * (1 - P/K)
    MortC <- rM * C

    dP <- GrowthP - IngestC
    dC <- IngestC * AE - MortC
    return(list(c(dP, dC)))
  })
}

### Define parameters ###
pars <- c(rI = 1.5, rG = 1.5, rM = 2, AE = 1, K = 10)

### Define time sequence ###
times <- seq(0, 30, by = 0.01)

### Initial conditions ###
set.seed(1)
d <- 2
n <- 12*d
Design <- maximinLHS(n, d) * 5 # Generate LHS samples in range [0,5]
colnames(Design) <- c("P", "C")

### Fit GP emulators ###
fit.dyn <- dynemu_GP(LVmod0D, times, pars, Design)

### Define uncertain inputs ###
xmean <- c(P = 1, C = 2)
xvar <- c(P = 1e-5, C = 1e-5)

### Define number of MC samples ###
mc.sample <- 1000

### Compute the next point ###
dynemu_mc(xmean, xvar, fit.dyn, mc.sample)
```

---

dynemu_pred *Predictive posterior computation for dynamic systems using one-steap-ahead approach by Gaussian process (GP) emulators*

Description

The function computes the predictive posterior distribution (mean and variance) at all time steps for dynamic systems modeled by GP emulators. It can use either the exact computation or Monte Carlo (MC) approximation to compute the predictive posterior.

Usage

```
dynemu_pred(dynGP, times, xnew, method="Exact", mc.sample=NULL, trace=TRUE)
```

Arguments

dynGP	list of trained GP emulators fitted by dynemu_GP, each corresponding to a state variable.
times	numeric vector specifying the time sequence at which predictions are to be made.
xnew	numeric vector or single-row matrix specifying the initial state values at time 0. The number of columns must match the input dimension of the GP emulators.
method	character specifying the method for prediction, to be chosen between "Exact"(exact closed-form solution), or "MC"(MC approximation). Default is "Exact".
mc.sample	a number of mc samples generated for MC approximation. Required only if method="MC".
trace	logical indicating whether to print the progress bar. If trace=TRUE, the progress bar is printed. Default is TRUE.

Details

Given a trained set of GP emulators 'dynGP' fitted using dynemu_GP, this function: 1. Computes the predictive posterior mean and variance for each state variable at all time steps. 2. The function can either: - Use the exact computation for a closed-form solution by method="Exact". - Use the MC approximation method to generate samples and estimate the posterior by method="MC".

Value

A list containing:

- times: The time sequence vector.
- mu: A matrix of predictive mean values for each state variable. Each row corresponds to a time step, and each column corresponds to a state variable.
- sig2: A matrix of predictive variance values for each state variable. Each row corresponds to a time step, and each column corresponds to a state variable.

Examples

```
library(lhs)
### Lotka-Volterra equations ###
LVmod0D <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)), {
    IngestC <- rI * P * C
    GrowthP <- rG * P * (1 - P/K)
    MortC <- rM * C

    dP <- GrowthP - IngestC
    dC <- IngestC * AE - MortC
    return(list(c(dP, dC)))
```

```
  })
}

### Define parameters ###
pars <- c(rI = 1.5, rG = 1.5, rM = 2, AE = 1, K = 10)

### Define time sequence ###
times <- seq(0, 30, by = 0.01)

### Initial conditions ###
set.seed(1)
d <- 2
n <- 12*d
Design <- maximinLHS(n, d) * 5 # Generate LHS samples in range [0,5]
colnames(Design) <- c("P", "C")

### Fit GP emulators ###
fit.dyn <- dynemu_GP(LVmod0D, times, pars, Design)

### Define initial test values ###
state <- c(P = 1, C = 2)

### Generate test set ###
test <- dyn_solve(LVmod0D, times, pars, state)

### Define number of MC samples ###
mc.sample <- 1000

### Prediction ###
pred.exact.dyn <- dynemu_pred(fit.dyn, times, state, method="Exact")
pred.exact.mu <- pred.exact.dyn$mu
pred.exact.sig2 <- pred.exact.dyn$sig2

pred.mc.dyn <- dynemu_pred(fit.dyn, times, state, method="MC", trace=TRUE)
pred.mc.mu <- pred.mc.dyn$mu
pred.mc.sig2 <- pred.mc.dyn$sig2

### Compute RMSE ###
sqrt(mean((pred.exact.mu[,1]-test$P)^2)) # 0.03002421
sqrt(mean((pred.exact.mu[,2]-test$C)^2)) # 0.02291742

sqrt(mean((pred.mc.mu[,1]-test$P)^2)) # 0.03050849
sqrt(mean((pred.mc.mu[,2]-test$C)^2)) # 0.02330542
```

---

dyn_solve *Solving ordinary (ODEs) differntial equations with a specified time squence.*

Description

The function computes numerical solutions for a system of ordinary differential equations given an initial state, parameters, and a sequence of time points $t_1, ..., t_T$.

Usage

```
dyn_solve(odefun, times, param, init, method)
```

Arguments

`odefun`	function defining the ODE system. It should return a list where the first element is a numeric vector of derivatives.
`times`	numeric vector of time points where the solution is computed.
`param`	numeric scalar or vector of parameters to be passed to `odefun`.
`init`	numeric vector or a single-row matrix specifying the initial state values. Each element corresponds to a state variable.
`method`	character specifying the numerical solver. Default is `"lsoda"`.

Details

This function numerically integrates a system of ODEs using solvers available in the `deSolve` package. The ODE system is defined by the user through `odefun`, which specifies the rate of change for each state variable.

The solver computes the values of the state variables at each time step, producing a trajectory of the system's evolution over time.

The ODE system must be written in first-order form:

$$\frac{dy}{dt} = f(t, y, p)$$

where: - y is the vector of state variables, - t is time, - p is a set of model parameters, - f is a function returning the derivatives.

This function simplifies the process of solving ODEs by managing input formatting and output structure, making it easier to extract and analyze results.

For details, see "`ode`".

Value

A list containing:

- `times`: copy of `times`
- `<state variable>`: Numeric vectors of computed values for each state variable.

Examples

```
### Lotka-Volterra equations ###
LVmod0D <- function(Time, State, Pars) {
  with(as.list(c(State, Pars)),{
    IngestC <- rI * P * C
    GrowthP <- rG * P * (1 - P/K)
    MortC <- rM * C

    dP <- GrowthP - IngestC
    dC <- IngestC * AE - MortC
    return(list(c(dP, dC)))
  })
}

### Define parameters ###
pars <- c(rI = 1.5, rG = 1.5, rM = 2, AE = 1, K = 10)

### Define time sequence ###
times <- seq(0, 30, by = 0.01)

### Initial conditions ###
state <- c(P = 1, C = 2)

### Solve ODE ###
dyn_solve(LVmod0D, times, pars, state)
```

Index