# Package 'AdapDiscom'

May 6, 2026

**Type** Package

**Title** Adaptive Sparse Regression for Block Missing Multimodal Data

**Version** 1.0.0

**Date** 2025-08-18

**Description**

Provides adaptive direct sparse regression for high-dimensional multimodal data with heterogeneous missing patterns and measurement errors. 'AdapDISCOM' extends the 'DISCOM' framework with modality-specific adaptive weighting to handle varying data structures and error magnitudes across blocks. The method supports flexible block configurations (any K blocks) and includes robust variants for heavy-tailed distributions ('AdapDISCOM'-Huber) and fast implementations for large-scale applications (Fast-'AdapDISCOM'). Designed for realistic multimodal scenarios where different data sources exhibit distinct missing data patterns and contamination levels. Diakité et al. (2025) <doi:10.48550/arXiv.2508.00120>.

**License** GPL-3

**URL** https://doi.org/10.48550/arXiv.2508.00120

**BugReports** https://github.com/AODiakite/AdapDiscom/issues

**Depends** R (>= 3.5.0)

**Imports** softImpute, Matrix, scout, robustbase

**RoxygenNote** 7.3.2

**Encoding** UTF-8

**Suggests** knitr, rmarkdown, MASS

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Diakite Abdoul Oudouss [aut, cre, cph],
Barry Amadou [aut]

**Maintainer** Diakite Abdoul Oudouss <abdouloudoussdiakite@gmail.com>

**Repository** CRAN

**Date/Publication** 2025-08-27 16:30:27 UTC

# Contents



---

`adapdiscom` *AdapDiscom: An Adaptive Sparse Regression Method for High-Dimensional Multimodal Data With Block-Wise Missingness and Measurement Errors*

---

## Description

AdapDiscom: An Adaptive Sparse Regression Method for High-Dimensional Multimodal Data With Block-Wise Missingness and Measurement Errors

## Usage

```
adapdiscom(
  beta,
  x,
  y,
  x.tuning,
  y.tuning,
  x.test,
  y.test,
  nlambda,
  nalpha,
  pp,
  robust = 0,
  standardize = TRUE,
  itcp = TRUE,
  lambda.min.ratio = NULL,
  k.value = 1.5
)
```

## Arguments

| | |
|---|---|
| `beta` | Vector, true beta coefficients (optional) |
| `x` | Matrix, training data |

| | |
|---|---|
| y | Vector, training response |
| x.tuning | Matrix, tuning data |
| y.tuning | Vector, tuning response |
| x.test | Matrix, test data |
| y.test | Vector, test response |
| nlambda | Integer, number of lambda values |
| nalpha | Integer, number of alpha values |
| pp | Vector, block sizes |
| robust | Integer, 0 for classical, 1 for robust estimation of covariance |
| standardize | Logical, whether to standardize covariates. When TRUE, uses training data mean and standard deviation to standardize tuning and test sets. When robust=1, uses Huber-robust standard deviation estimates |
| itcp | Logical, whether to include intercept |
| lambda.min.ratio | Numeric, 'lambda.min.ratio' sets the smallest lambda value in the grid, expressed as a fraction of 'lambda.max'—the smallest lambda for which all coefficients are zero. By default, it is '0.0001' when the number of observations ('nobs') exceeds the number of variables ('nvars'), and '0.01' when 'nobs < nvars'. Using a very small value in the latter case can lead to overfitting. |
| k.value | Numeric, tuning parameter for robust estimation |

## Value

List with estimation results

## Value

The function returns a list containing the following components:

**err** A multi-dimensional array storing the mean squared error (MSE) for all combinations of tuning parameters alpha and lambda.

**est.error** The estimation error, calculated as the Euclidean distance between the estimated beta coefficients and the true beta (if provided).

**lambda** The optimal lambda value chosen via cross-validation on the tuning set.

**alpha** A vector of the optimal alpha values, also selected on the tuning set.

**train.error** The mean squared error on the tuning set for the optimal parameter combination.

**test.error** The mean squared error on the test set for the final, optimal model.

**y.pred** The predicted values for the observations in the test set.

**R2** The R-squared value, which measures the proportion of variance explained by the model on the test set.

**a0** The intercept of the final model.

**a1** The vector of estimated beta coefficients for the final model.

**select** The number of non-zero coefficients, representing the number of selected variables.

**xtx** The final regularized covariance matrix used to fit the optimal model.

**fpr** The False Positive Rate (FPR) if the true beta is provided. It measures the proportion of irrelevant variables incorrectly selected.

**fnr** The False Negative Rate (FNR) if the true beta is provided. It measures the proportion of relevant variables incorrectly excluded.

**lambda.all** The complete vector of all lambda values tested during cross-validation.

**beta.cov.lambda.max** The estimated beta coefficients using the maximum lambda value.

**time** The total execution time of the function in seconds.

## Examples

```
# Simple example with synthetic data
n <- 100
p <- 20

# Generate synthetic data with 2 blocks
set.seed(123)
x_train <- matrix(rnorm(n * p), n, p)
x_tuning <- matrix(rnorm(50 * p), 50, p)
x_test <- matrix(rnorm(30 * p), 30, p)

# True coefficients
beta_true <- c(rep(2, 5), rep(0, 15))

# Response variables
y_train <- x_train %*% beta_true + rnorm(n)
y_tuning <- x_tuning %*% beta_true + rnorm(50)
y_test <- x_test %*% beta_true + rnorm(30)

# Block sizes (2 blocks of 10 variables each)
pp <- c(10, 10)

# Run AdapDiscom
result <- adapdiscom(beta = beta_true,
                     x = x_train, y = y_train,
                     x.tuning = x_tuning, y.tuning = y_tuning,
                     x.test = x_test, y.test = y_test,
                     nlambda = 20, nalpha = 10, pp = pp)

# View results
print(paste("Test R-squared:", round(result$R2, 3)))
print(paste("Selected variables:", result$select))
```

compute.xtx *Compute X'X Matrix*

## Description

Compute X'X Matrix

## Usage

```
compute.xtx(x, robust = 0, k_value = 1.5)
```

## Arguments

| | |
|---|---|
| x | Matrix, input data matrix |
| robust | Integer, 0 for classical estimate, 1 for Huber robust estimate |
| k_value | Numeric, tuning parameter for Huber function |

## Value

Covariance matrix

## Examples

```
# Create sample data with missing values
set.seed(123)
x <- matrix(rnorm(100), 20, 5)
x[sample(100, 10)] <- NA  # Introduce missing values

# Classical covariance estimation
xtx_classical <- compute.xtx(x, robust = 0)
print(round(xtx_classical, 3))

# Robust covariance estimation
xtx_robust <- compute.xtx(x, robust = 1, k_value = 1.5)
print(round(xtx_robust, 3))
```

compute.xty *Compute X'y Vector*

## Description

Compute X'y Vector

## Usage

```
compute.xty(x, y, robust = 0, k_value = 1.5)
```

### Arguments

| | |
|---|---|
| x | Matrix, input data matrix |
| y | Vector, response vector |
| robust | Integer, 0 for classical estimate, 1 for Huber robust estimate |
| k_value | Numeric, tuning parameter for Huber function |

### Value

Covariance vector

### Examples

```
# Create sample data
set.seed(123)
x <- matrix(rnorm(100), 20, 5)
y <- rnorm(20)
x[sample(100, 8)] <- NA  # Missing values in x

# Classical cross-covariance
xty_classical <- compute.xty(x, y, robust = 0)
print(round(xty_classical, 3))

# Robust cross-covariance
xty_robust <- compute.xty(x, y, robust = 1)
print(round(xty_robust, 3))
```

---

## discom *DISCOM: Optimal Sparse Linear Prediction for Block-missing Multi-modality Data Without Imputation*

---

### Description

DISCOM: Optimal Sparse Linear Prediction for Block-missing Multi-modality Data Without Imputation

### Usage

```
discom(
  beta,
  x,
  y,
  x.tuning,
  y.tuning,
  x.test,
  y.test,
  nlambda,
  nalpha,
```

```
  pp,
  robust = 0,
  standardize = TRUE,
  itcp = TRUE,
  lambda.min.ratio = NULL,
  k.value = 1.5
)
```

## Arguments

| | |
|---|---|
| beta | Vector, true beta coefficients (optional) |
| x | Matrix, training data |
| y | Vector, training response |
| x.tuning | Matrix, tuning data |
| y.tuning | Vector, tuning response |
| x.test | Matrix, test data |
| y.test | Vector, test response |
| nlambda | Integer, number of lambda values |
| nalpha | Integer, number of alpha values |
| pp | Vector, block sizes. Discom supports 2, 3, or 4 blocks. |
| robust | Integer, 0 for classical, 1 for robust estimation |
| standardize | Logical, whether to standardize covariates. When TRUE, uses training data mean and standard deviation to standardize tuning and test sets. When robust=1, uses Huber-robust standard deviation estimates |
| itcp | Logical, whether to include intercept |
| lambda.min.ratio | Numeric, ‘lambda.min.ratio‘ sets the smallest lambda value in the grid, expressed as a fraction of ‘lambda.max‘—the smallest lambda for which all coefficients are zero. By default, it is ‘0.0001‘ when the number of observations (‘nobs‘) exceeds the number of variables (‘nvars‘), and ‘0.01‘ when ‘nobs < nvars‘. Using a very small value in the latter case can lead to overfitting. |
| k.value | Numeric, tuning parameter for robust estimation |

## Value

List with estimation results

## Value

The function returns a list containing the following components:

**err** A multi-dimensional array storing the mean squared error (MSE) for all combinations of tuning parameters alpha and lambda.

**est.error** The estimation error, calculated as the Euclidean distance between the estimated beta coefficients and the true beta (if provided).

**lambda** The optimal lambda value chosen via cross-validation on the tuning set.

**alpha** A vector of the optimal alpha values, also selected on the tuning set.

**train.error** The mean squared error on the tuning set for the optimal parameter combination.

**test.error** The mean squared error on the test set for the final, optimal model.

**y.pred** The predicted values for the observations in the test set.

**R2** The R-squared value, which measures the proportion of variance explained by the model on the test set.

**a0** The intercept of the final model.

**a1** The vector of estimated beta coefficients for the final model.

**select** The number of non-zero coefficients, representing the number of selected variables.

**xtx** The final regularized covariance matrix used to fit the optimal model.

**fpr** The False Positive Rate (FPR) if the true beta is provided. It measures the proportion of irrelevant variables incorrectly selected.

**fnr** The False Negative Rate (FNR) if the true beta is provided. It measures the proportion of relevant variables incorrectly excluded.

**lambda.all** The complete vector of all lambda values tested during cross-validation.

**beta.cov.lambda.max** The estimated beta coefficients using the maximum lambda value.

**time** The total execution time of the function in seconds.

## Examples

```
# Simple example with synthetic multimodal data
n <- 100
p <- 24

# Generate synthetic data with 3 blocks
set.seed(456)
x_train <- matrix(rnorm(n * p), n, p)
x_tuning <- matrix(rnorm(50 * p), 50, p)
x_test <- matrix(rnorm(30 * p), 30, p)

# True coefficients with sparse structure
beta_true <- c(rep(1.5, 4), rep(0, 4), rep(-1, 4), rep(0, 12))

# Response variables
y_train <- x_train %*% beta_true + rnorm(n, sd = 0.5)
y_tuning <- x_tuning %*% beta_true + rnorm(50, sd = 0.5)
y_test <- x_test %*% beta_true + rnorm(30, sd = 0.5)

# Block sizes (3 blocks of 8 variables each)
pp <- c(8, 8, 8)

# Run DISCOM
result <- discom(beta = beta_true,
                 x = x_train, y = y_train,
                 x.tuning = x_tuning, y.tuning = y_tuning,
```

```
                x.test = x_test, y.test = y_test,
                nlambda = 25, nalpha = 15, pp = pp)

# View results
print(paste("Test error:", round(result$test.error, 4)))
print(paste("R-squared:", round(result$R2, 3)))
print(paste("Variables selected:", result$select))
```

---

`fast_adapdiscom` *Fast AdapDiscom*

---

## Description

Fast AdapDiscom

## Usage

```
fast_adapdiscom(
  beta,
  x,
  y,
  x.tuning,
  y.tuning,
  x.test,
  y.test,
  nlambda,
  pp,
  robust = 0,
  n.l = 30,
  standardize = TRUE,
  itcp = TRUE,
  lambda.min.ratio = NULL,
  k.value = 1.5
)
```

## Arguments

| | |
|---|---|
| `beta` | Vector, true beta coefficients (optional) |
| `x` | Matrix, training data |
| `y` | Vector, training response |
| `x.tuning` | Matrix, tuning data |
| `y.tuning` | Vector, tuning response |
| `x.test` | Matrix, test data |
| `y.test` | Vector, test response |
| `nlambda` | Integer, number of lambda values |

| | |
|---|---|
| pp | Vector, block sizes |
| robust | Integer, 0 for classical, 1 for robust estimation |
| n.l | Integer, number of tuning parameter (‘l‘) values for fast variants (number of alpha values) |
| standardize | Logical, whether to standardize covariates. When TRUE, uses training data mean and standard deviation to standardize tuning and test sets. When robust=1, uses Huber-robust standard deviation estimates |
| itcp | Logical, whether to include intercept |
| lambda.min.ratio | Numeric, ‘lambda.min.ratio‘ sets the smallest lambda value in the grid, expressed as a fraction of ‘lambda.max‘—the smallest lambda for which all coefficients are zero. By default, it is ‘0.0001‘ when the number of observations (‘nobs‘) exceeds the number of variables (‘nvars‘), and ‘0.01‘ when ‘nobs < nvars‘. Using a very small value in the latter case can lead to overfitting. |
| k.value | Numeric, tuning parameter for robust estimation |

## Value

List with estimation results

## Value

The function returns a list containing the following components:

**err** A multi-dimensional array storing the mean squared error (MSE) for all combinations of tuning parameters alpha and lambda.

**est.error** The estimation error, calculated as the Euclidean distance between the estimated beta coefficients and the true beta (if provided).

**lambda** The optimal lambda value chosen via cross-validation on the tuning set.

**alpha** A vector of the optimal alpha values, also selected on the tuning set.

**train.error** The mean squared error on the tuning set for the optimal parameter combination.

**test.error** The mean squared error on the test set for the final, optimal model.

**y.pred** The predicted values for the observations in the test set.

**R2** The R-squared value, which measures the proportion of variance explained by the model on the test set.

**a0** The intercept of the final model.

**a1** The vector of estimated beta coefficients for the final model.

**select** The number of non-zero coefficients, representing the number of selected variables.

**xtx** The final regularized covariance matrix used to fit the optimal model.

**fpr** The False Positive Rate (FPR) if the true beta is provided. It measures the proportion of irrelevant variables incorrectly selected.

**fnr** The False Negative Rate (FNR) if the true beta is provided. It measures the proportion of relevant variables incorrectly excluded.

**lambda.all** The complete vector of all lambda values tested during cross-validation.

**beta.cov.lambda.max** The estimated beta coefficients using the maximum lambda value.

**time** The total execution time of the function in seconds.

## Examples

```
# Fast computation example with synthetic data
n <- 80
p <- 16

# Generate synthetic data with 2 blocks
set.seed(789)
x_train <- matrix(rnorm(n * p), n, p)
x_tuning <- matrix(rnorm(40 * p), 40, p)
x_test <- matrix(rnorm(25 * p), 25, p)

# True coefficients
beta_true <- c(rep(1.2, 3), rep(0, 5), rep(-0.8, 2), rep(0, 6))

# Response variables
y_train <- x_train %*% beta_true + rnorm(n, sd = 0.3)
y_tuning <- x_tuning %*% beta_true + rnorm(40, sd = 0.3)
y_test <- x_test %*% beta_true + rnorm(25, sd = 0.3)

# Block sizes (2 blocks of 8 variables each)
pp <- c(8, 8)

# Run fast AdapDiscom (faster with fewer tuning parameters)
result <- fast_adapdiscom(beta = beta_true,
                          x = x_train, y = y_train,
                          x.tuning = x_tuning, y.tuning = y_tuning,
                          x.test = x_test, y.test = y_test,
                          nlambda = 15, pp = pp, n.l = 20)

# View results
print(paste("Test R-squared:", round(result$R2, 3)))
print(paste("Computation time:", round(result$time[3], 2), "seconds"))
```

---

`fast_discom` *Fast DISCOM*

---

## Description

Fast DISCOM

## Usage

```
fast_discom(
  beta,
  x,
  y,
  x.tuning,
  y.tuning,
```

```
  x.test,
  y.test,
  nlambda,
  pp,
  robust = 0,
  n.l = 30,
  standardize = TRUE,
  itcp = TRUE,
  lambda.min.ratio = NULL,
  k.value = 1.5
)
```

## Arguments

| | |
|---|---|
| beta | Vector, true beta coefficients (optional) |
| x | Matrix, training data |
| y | Vector, training response |
| x.tuning | Matrix, tuning data |
| y.tuning | Vector, tuning response |
| x.test | Matrix, test data |
| y.test | Vector, test response |
| nlambda | Integer, number of lambda values |
| pp | Vector, block sizes |
| robust | Integer, 0 for classical, 1 for robust estimation |
| n.l | Integer, number of tuning parameter (‘l‘) values for fast variants |
| standardize | Logical, whether to standardize covariates. When TRUE, uses training data mean and standard deviation to standardize tuning and test sets. When robust=1, uses Huber-robust standard deviation estimates |
| itcp | Logical, whether to include intercept |
| lambda.min.ratio | Numeric, ‘lambda.min.ratio‘ sets the smallest lambda value in the grid, expressed as a fraction of ‘lambda.max‘—the smallest lambda for which all coefficients are zero. By default, it is ‘0.0001‘ when the number of observations (‘nobs‘) exceeds the number of variables (‘nvars‘), and ‘0.01‘ when ‘nobs < nvars‘. Using a very small value in the latter case can lead to overfitting. |
| k.value | Numeric, tuning parameter for robust estimation |

## Value

List with estimation results

## Value

The function returns a list containing the following components:

**err** A multi-dimensional array storing the mean squared error (MSE) for all combinations of tuning parameters alpha and lambda.

**est.error** The estimation error, calculated as the Euclidean distance between the estimated beta coefficients and the true beta (if provided).

**lambda** The optimal lambda value chosen via cross-validation on the tuning set.

**alpha** A vector of the optimal alpha values, also selected on the tuning set.

**train.error** The mean squared error on the tuning set for the optimal parameter combination.

**test.error** The mean squared error on the test set for the final, optimal model.

**y.pred** The predicted values for the observations in the test set.

**R2** The R-squared value, which measures the proportion of variance explained by the model on the test set.

**a0** The intercept of the final model.

**a1** The vector of estimated beta coefficients for the final model.

**select** The number of non-zero coefficients, representing the number of selected variables.

**xtx** The final regularized covariance matrix used to fit the optimal model.

**fpr** The False Positive Rate (FPR) if the true beta is provided. It measures the proportion of irrelevant variables incorrectly selected.

**fnr** The False Negative Rate (FNR) if the true beta is provided. It measures the proportion of relevant variables incorrectly excluded.

**lambda.all** The complete vector of all lambda values tested during cross-validation.

**beta.cov.lambda.max** The estimated beta coefficients using the maximum lambda value.

**time** The total execution time of the function in seconds.

## Examples

```
# Fast DISCOM example with synthetic multimodal data
n <- 70
p <- 18

# Generate synthetic data with 3 blocks
set.seed(321)
x_train <- matrix(rnorm(n * p), n, p)
x_tuning <- matrix(rnorm(35 * p), 35, p)
x_test <- matrix(rnorm(20 * p), 20, p)

# True coefficients with block structure
beta_true <- c(rep(1.0, 3), rep(0, 3), rep(-1.2, 3), rep(0, 3), rep(0.8, 3), rep(0, 3))

# Response variables
y_train <- x_train %*% beta_true + rnorm(n, sd = 0.4)
y_tuning <- x_tuning %*% beta_true + rnorm(35, sd = 0.4)
y_test <- x_test %*% beta_true + rnorm(20, sd = 0.4)
```

```
# Block sizes (3 blocks of 6 variables each)
pp <- c(6, 6, 6)

# Run fast DISCOM (efficient for large datasets)
result <- fast_discom(beta = beta_true,
                      x = x_train, y = y_train,
                      x.tuning = x_tuning, y.tuning = y_tuning,
                      x.test = x_test, y.test = y_test,
                      nlambda = 20, pp = pp, n.l = 25)

# View results
print(paste("Test error:", round(result$test.error, 4)))
print(paste("R-squared:", round(result$R2, 3)))
print(paste("Runtime:", round(result$time, 2), "seconds"))
```

---

`generate.cov` *Generate Covariance Matrix*

---

## Description

Generate Covariance Matrix

## Usage

```
generate.cov(p, example)
```

## Arguments

| | |
|---|---|
| `p` | Integer, dimension of the covariance matrix |
| `example` | Integer, type of covariance structure (1=AR(1), 2=Block diagonal, 3=Kronecker product) |

## Value

A p x p covariance matrix

## Examples

```
# AR(1) covariance structure
Sigma1 <- generate.cov(p = 20, example = 1)
print(Sigma1[1:3, 1:3])

# Block diagonal structure (p must be multiple of 5)
Sigma2 <- generate.cov(p = 25, example = 2)
print(Sigma2[1:5, 1:5])

# Kronecker product structure (p must be multiple of 10)
Sigma3 <- generate.cov(p = 100, example = 3)
print(dim(Sigma3))
```

## get_block_indices *Get Block Indices*

### Description

Get Block Indices

### Usage

```
get_block_indices(pp)
```

### Arguments

| | |
|---|---|
| pp | Vector, block sizes |

### Value

List with start and end indices for each block

### Examples

```
# Define block sizes
pp <- c(10, 15, 20)
indices <- get_block_indices(pp)
print(indices)
# Shows: $starts = c(1, 11, 26) and $ends = c(10, 25, 45)

# For two blocks
pp2 <- c(25, 25)
indices2 <- get_block_indices(pp2)
print(indices2)
```

## lambda_max *Compute Lambda Max for L1 Regularization using KKT Conditions*

### Description

Compute Lambda Max for L1 Regularization using KKT Conditions

### Usage

```
lambda_max(X, y, Methode = "lasso", robust = 0)
```

## Arguments

| | |
|---|---|
| X | Matrix, design matrix |
| y | Vector, response vector |
| Methode | Character, method for computation |
| robust | Integer, 0 for classical, 1 for robust |

## Value

Maximum lambda value

## Examples

```
# Generate sample data
set.seed(123)
n <- 50; p <- 20
X <- matrix(rnorm(n*p), n, p)
y <- rnorm(n)

# Different methods for lambda_max computation
lmax_lasso <- lambda_max(X, y, Methode = "lasso")
lmax_discom <- lambda_max(X, y, Methode = "discom")

print(paste("Lambda max (lasso):", round(lmax_lasso, 4)))
print(paste("Lambda max (discom):", round(lmax_discom, 4)))
```

# Index