pdist2

Pairwise distance between two sets of observations

Syntax

```
D = pdist2(X,Y,Distance)

D = pdist2(X,Y,Distance,DistParameter)

D = pdist2( __ ,Name,Value)
[D,I] = pdist2( __ ,Name,Value)
```

Description

`D = pdist2(X,Y,Distance)` returns the distance between each pair of observations in `X` and `Y` using the metric specified by `Distance`. example

`D = pdist2(X,Y,Distance,DistParameter)` returns the distance using the metric specified by `Distance` and `DistParameter`. You can specify `DistParameter` only when `Distance` is `'seuclidean'`, `'minkowski'`, or `'mahalanobis'`. example

`D = pdist2( __ ,Name,Value)` specifies an additional option using one of the name-value pair arguments `'Smallest'` or `'Largest'` in addition to any of the arguments in the previous syntaxes.

For example,

- `D = pdist2(X,Y,Distance,'Smallest',K)` computes the distance using the metric specified by `Distance` and returns the `K` smallest pairwise distances to observations in `X` for each observation in `Y` in ascending order.
- `D = pdist2(X,Y,Distance,DistParameter,'Largest',K)` computes the distance using the metric specified by `Distance` and `DistParameter` and returns the `K` largest pairwise distances in descending order.

`[D,I] = pdist2( __ ,Name,Value)` also returns the matrix `I`. The matrix `I` contains the indices of the observations in `X` corresponding to the distances in `D`. example

Examples

collapse all

Compute Euclidean Distance

Create two matrices with three observations and two variables.

Try This Example

Copy Command

```
rng('default') % For reproducibility
X = rand(3,2);
Y = rand(3,2);
```

Compute the Euclidean distance. The default value of the input argument `Distance` is `'euclidean'`. When computing the Euclidean distance without using a name-value pair argument, you do not need to specify `Distance`.

```
D = pdist2(X,Y)
```

```
D = 3×3

    0.5387    0.8018    0.1538
    0.7100    0.5951    0.3422
    0.8805    0.4242    1.2050
```

`D(i,j)` corresponds to the pairwise distance between observation `i` in `X` and observation `j` in `Y`.

Compute Minkowski Distance

Create two matrices with three observations and two variables.

Try This Example

Copy Command

```
rng('default') % For reproducibility
X = rand(3,2);
Y = rand(3,2);
```

Compute the Minkowski distance with the default exponent 2.

```
D1 = pdist2(X,Y,'minkowski')
```

```
D1 = 3×3

    0.5387    0.8018    0.1538
    0.7100    0.5951    0.3422
    0.8805    0.4242    1.2050
```

Compute the Minkowski distance with an exponent of 1, which is equal to the city block distance.

```
D2 = pdist2(X,Y,'minkowski',1)
```

```
D2 = 3×3

    0.5877    1.0236    0.2000
    0.9598    0.8337    0.3899
    1.0189    0.4800    1.7036
```

```
D3 = pdist2(X,Y,'cityblock')
```

```
D3 = 3×3

    0.5877    1.0236    0.2000
    0.9598    0.8337    0.3899
    1.0189    0.4800    1.7036
```

Find the Two Smallest Pairwise Distances

Create two matrices with three observations and two variables.

Try This Example

Copy Command

```
rng('default') % For reproducibility
X = rand(3,2);
Y = rand(3,2);
```

Find the two smallest pairwise Euclidean distances to observations in `X` for each observation in `Y`.

```
[D,I] = pdist2(X,Y,'euclidean','Smallest',2)
```

```
D = 2×3

    0.5387    0.4242    0.1538
    0.7100    0.5951    0.3422

I = 2×3

     1     3     1
     2     2     2
```

For each observation in `Y`, `pdist2` finds the two smallest distances by computing and comparing the distance values to all the observations in `X`. The function then sorts the distances in each column of `D` in ascending order. `I` contains the indices of the observations in `X` corresponding to the distances in `D`.

Compute Pairwise Distance with Missing Elements Using a Custom Distance Function

Define a custom distance function that ignores coordinates with `NaN` values, and compute pairwise distance by using the custom distance function.

Copy Command

Create two matrices with three observations and three variables.

```
rng('default') % For reproducibility
X = rand(3,3)
Y = [X(:,1:2) rand(3,1)]
```

```
X =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575


Y =

    0.8147    0.9134    0.9649
    0.9058    0.6324    0.1576
```

```
    0.1270    0.0975    0.9706
```

The first two columns of X and Y are identical. Assume that `X(1,1)` is missing.

```
X(1,1) = NaN
```

```
X =

       NaN    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

Compute the Hamming distance.

```
D1 = pdist2(X,Y,'hamming')
```

```
D1 =

       NaN       NaN       NaN
    1.0000    0.3333    1.0000
    1.0000    1.0000    0.3333
```

If observation `i` in `X` or observation `j` in `Y` contains `NaN` values, the function `pdist2` returns `NaN` for the pairwise distance between `i` and `j`. Therefore, D1(1,1), D1(1,2), and D1(1,3) are `NaN` values.

Define a custom distance function `nanhamdist` that ignores coordinates with `NaN` values and computes the Hamming distance. When working with a large number of observations, you can compute the distance more quickly by looping over coordinates of the data.

```
function D2 = nanhamdist(XI,XJ)
%NANHAMDIST Hamming distance ignoring coordinates with NaNs
[m,p] = size(XJ);
nesum = zeros(m,1);
pstar = zeros(m,1);
for q = 1:p
    notnan = ~(isnan(XI(q)) | isnan(XJ(:,q)));
    nesum = nesum + ((XI(q) ~= XJ(:,q)) & notnan);
    pstar = pstar + notnan;
end
D2 = nesum./pstar;
```

Compute the distance with `nanhamdist` by passing the function handle as an input argument of `pdist2`.

```
D2 = pdist2(X,Y,@nanhamdist)
```

```
D2 =

    0.5000    1.0000    1.0000
    1.0000    0.3333    1.0000
    1.0000    1.0000    0.3333
```

Assign New Data to Existing Clusters and Generate C/C++ Code

`kmeans` performs *k*-means clustering to partition data into *k* clusters. When you have a new data set to cluster, you can create new clusters that include the existing data and the new data by using `kmeans`. The `kmeans` function supports C/C++ code generation, so you can generate code that accepts training data and returns clustering results, and then deploy the code to a device. In this workflow, you must pass training data, which can be of considerable size. To save memory on the device, you can separate training and prediction by using `kmeans` and `pdist2`, respectively.

Use `kmeans` to create clusters in MATLAB® and use `pdist2` in the generated code to assign new data to existing clusters. For code generation, define an entry-point function that accepts the cluster centroid positions and the new data set, and returns the index of the nearest cluster. Then, generate code for the entry-point function.

Generating C/C++ code requires MATLAB® Coder™.

Try This Example

This example uses:

GPU Coder

MATLAB Coder

Statistics and Machine Learning Toolbox

Copy Command

Perform *k*-Means Clustering

Generate a training data set using three distributions.

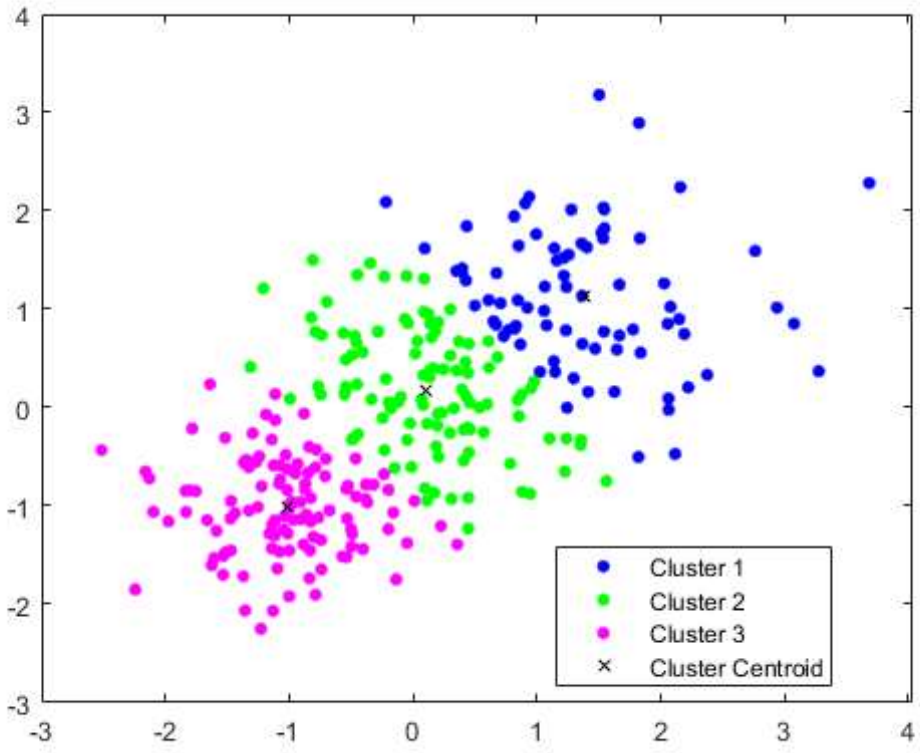
```
rng('default') % For reproducibility
X = [randn(100,2)*0.75+ones(100,2);
    randn(100,2)*0.5-ones(100,2);
    randn(100,2)*0.75];
```

Partition the training data into three clusters by using `kmeans`.

```
[idx,C] = kmeans(X,3);
```

Plot the clusters and the cluster centroids.

```
figure
gscatter(X(:,1),X(:,2),idx,'bgm')
hold on
plot(C(:,1),C(:,2),'kx')
legend('Cluster 1','Cluster 2','Cluster 3','Cluster Centroid')
```



Assign New Data to Existing Clusters

Generate a test data set.

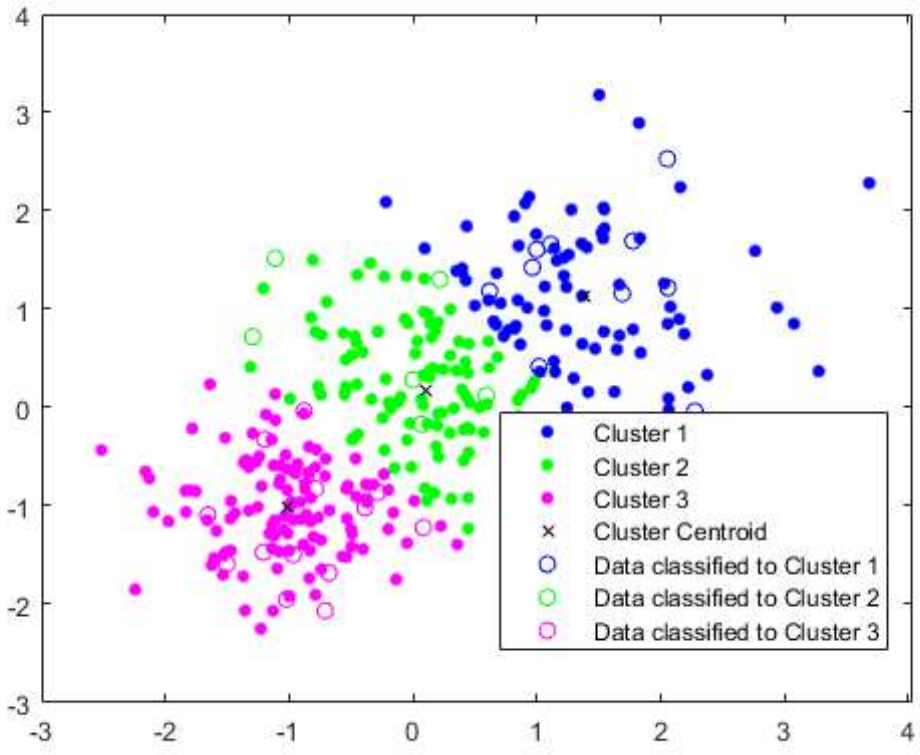
```
Xtest = [randn(10,2)*0.75+ones(10,2);
    randn(10,2)*0.5-ones(10,2);
    randn(10,2)*0.75];
```

Classify the test data set using the existing clusters. Find the nearest centroid from each test data point by using `pdist2`.

```
[~,idx_test] = pdist2(C,Xtest,'euclidean','Smallest',1);
```

Plot the test data and label the test data using `idx_test` by using `gscatter`.

```
gscatter(Xtest(:,1),Xtest(:,2),idx_test,'bgm','ooo')
legend('Cluster 1','Cluster 2','Cluster 3','Cluster Centroid', ...
    'Data classified to Cluster 1','Data classified to Cluster 2', ...
    'Data classified to Cluster 3')
```



Generate Code

Generate C code that assigns new data to the existing clusters. Note that generating C/C++ code requires MATLAB® Coder™.

Define an entry-point function named `findNearestCentroid` that accepts centroid positions and new data, and then find the nearest cluster by using `pdist2`.

Add the `%#codegen` compiler directive (or pragma) to the entry-point function after the function signature to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would cause errors during code generation.

```
type findNearestCentroid % Display contents of findNearestCentroid.m
```

```
function idx = findNearestCentroid(C,X) %#codegen
[~,idx] = pdist2(C,X,'euclidean','Smallest',1); % Find the nearest centroid
```

Note: If you click the button located in the upper-right section of this page and open this example in MATLAB®, then MATLAB® opens the example folder. This folder includes the entry-point function file.

Generate code by using `codegen` (MATLAB Coder). Because C and C++ are statically typed languages, you must determine the properties of all variables in the entry-point function at compile time. To specify the data type and array size of the inputs of `findNearestCentroid`, pass a MATLAB expression that represents the set of values with a certain data type and array size by using the `-args` option. For details, see Specify Variable-Size Arguments for Code Generation.

```
codegen findNearestCentroid -args {C,Xtest}
```

```
Code generation successful.
```

`codegen` generates the MEX function `findNearestCentroid_mex` with a platform-dependent extension.

Verify the generated code.

```
myIndx = findNearestCentroid(C,Xtest);
myIndex_mex = findNearestCentroid_mex(C,Xtest);
```

```
verifyMEX = isequal(idx_test,myIndx,myIndex_mex)
```

```
verifyMEX = logical
   1
```

`isequal` returns logical 1 (`true`), which means all the inputs are equal. The comparison confirms that the `pdist2` function, the `findNearestCentroid` function, and the MEX function return the same index.

You can also generate optimized CUDA® code using GPU Coder™.

```
cfg = coder.gpuConfig('mex');
codegen -config cfg findNearestCentroid -args {C,Xtest}
```

For more information on code generation, see General Code Generation Workflow. For more information on GPU coder, see Get Started with GPU Coder (GPU Coder) and Supported Functions (GPU Coder).

Input Arguments

collapse all

`X,Y` — Input data

numeric matrix

Input data, specified as a numeric matrix. `X` is an mx-by-n matrix and `Y` is an my-by-n matrix. Rows correspond to individual observations, and columns correspond to individual variables.

Data Types: `single` | `double`

`Distance` — Distance metric

character vector | string scalar | function handle

Distance metric, specified as a character vector, string scalar, or function handle, as described in the following table.

Value	Description
`'euclidean'`	Euclidean distance (default).
`'squaredeuclidean'`	Squared Euclidean distance. (This option is provided for efficiency only. It does not satisfy the triangle inequality.)
`'seuclidean'`	Standardized Euclidean distance. Each coordinate difference between observations is scaled by dividing by the corresponding element of the standard deviation, `S = std(X,'omitnan')`. Use `DistParameter` to specify another value for `S`.
`'mahalanobis'`	Mahalanobis distance using the sample covariance of `X`, `C = cov(X,'omitrows')`. Use `DistParameter` to specify another value for `C`, where the matrix `C` is symmetric and positive definite.
`'cityblock'`	City block distance.
`'minkowski'`	Minkowski distance. The default exponent is 2. Use `DistParameter` to specify a different exponent `P`, where `P` is a positive scalar value of the exponent.
`'chebychev'`	Chebychev distance (maximum coordinate difference).

Value	Description
`'cosine'`	One minus the cosine of the included angle between points (treated as vectors).
`'correlation'`	One minus the sample correlation between points (treated as sequences of values).
`'hamming'`	Hamming distance, which is the percentage of coordinates that differ.
`'jaccard'`	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
`'spearman'`	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

`@distfun`	Custom distance function handle. A distance function has the form
	`function D2 = distfun(ZI,ZJ)`
	`% calculation of distance`
	`...`
	where
	• `ZI` is a 1-by-`n` vector containing a single observation.
	• `ZJ` is an `m2`-by-`n` matrix containing multiple observations. `distfun` must accept a matrix `ZJ` with an arbitrary number of observations.
	• `D2` is an `m2`-by-1 vector of distances, and `D2(k)` is the distance between observations `ZI` and `ZJ(k,:)`.
	If your data is not sparse, you can generally compute distance more quickly by using a built-in distance instead of a function handle.

For definitions, see Distance Metrics.

When you use `'seuclidean'`, `'minkowski'`, or `'mahalanobis'`, you can specify an additional input argument `DistParameter` to control these metrics. You can also use these metrics in the same way as the other metrics with a default value of `DistParameter`.

Example: `'minkowski'`

`DistParameter` — Distance metric parameter values

positive scalar | numeric vector | numeric matrix

Distance metric parameter values, specified as a positive scalar, numeric vector, or numeric matrix. This argument is valid only when you specify `Distance` as `'seuclidean'`, `'minkowski'`, or `'mahalanobis'`.

- If `Distance` is `'seuclidean'`, `DistParameter` is a vector of scaling factors for each dimension, specified as a positive vector. The default value is `std(X,'omitnan')`.
- If `Distance` is `'minkowski'`, `DistParameter` is the exponent of Minkowski distance, specified as a positive scalar. The default value is 2.
- If `Distance` is `'mahalanobis'`, `DistParameter` is a covariance matrix, specified as a numeric matrix. The default value is `cov(X,'omitrows')`. `DistParameter` must be symmetric and positive definite.

Example: `'minkowski',3`

Data Types: `single` | `double`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: Either `'Smallest',K` or `'Largest',K`. You cannot use both `'Smallest'` and `'Largest'`.

`Smallest` — Number of smallest distances to find

positive integer

Number of smallest distances to find, specified as the comma-separated pair consisting of `'Smallest'` and a positive integer. If you specify `'Smallest'`, then `pdist2` sorts the distances in each column of `D` in ascending order.

Example: `'Smallest',3`

Data Types: `single` | `double`

`Largest` — Number of largest distances to find

positive integer

Number of largest distances to find, specified as the comma-separated pair consisting of `'Largest'` and a positive integer. If you specify `'Largest'`, then `pdist2` sorts the distances in each column of `D` in descending order.

Example: `'Largest',3`

Data Types: `single` | `double`

Output Arguments

collapse all

`D` — Pairwise distances

numeric matrix

Pairwise distances, returned as a numeric matrix.

If you do not specify either `'Smallest'` or `'Largest'`, then `D` is an mx-by-my matrix, where mx and my are the number of observations in `X` and `Y`, respectively. `D(i,j)` is the distance between observation `i` in `X` and observation `j` in `Y`. If observation i in `X` or observation j in `Y` contains `NaN`, then `D(i,j)` is `NaN` for the built-in distance functions.

If you specify either `'Smallest'` or `'Largest'` as `K`, then `D` is a `K`-by-my matrix. `D` contains either the `K` smallest or `K` largest pairwise distances to observations in `X` for each observation in `Y`. For each observation in `Y`, `pdist2` finds the `K` smallest or largest distances by computing and comparing the distance values to all the observations in `X`. If `K` is greater than mx, `pdist2` returns an mx-by-my matrix.

`I` — Sort index
positive integer matrix

Sort index, returned as a positive integer matrix. `I` is the same size as `D`. `I` contains the indices of the observations in `X` corresponding to the distances in `D`.

More About

collapse all

Distance Metrics

A distance metric is a function that defines a distance between two observations. `pdist2` supports various distance metrics: Euclidean distance, standardized Euclidean distance, Mahalanobis distance, city block distance, Minkowski distance, Chebychev distance, cosine distance, correlation distance, Hamming distance, Jaccard distance, and Spearman distance.

Given an mx-by-n data matrix X, which is treated as mx (1-by-n) row vectors x_1, x_2, ..., x_{mx}, and an my-by-n data matrix Y, which is treated as my (1-by-n) row vectors y_1, y_2, ...,y_{my}, the various distances between the vector x_s and y_t are defined as follows:

- Euclidean distance

  $$d_{st}^2 = (x_s - y_t)(x_s - y_t)'.$$

  The Euclidean distance is a special case of the Minkowski distance, where $p = 2$.

- Standardized Euclidean distance

  $$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)',$$

  where V is the n-by-n diagonal matrix whose jth diagonal element is $(S(j))^2$, where S is a vector of scaling factors for each dimension.

- Mahalanobis distance

  $$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)',$$

  where C is the covariance matrix.

- City block distance

  $$d_{st} = \sum_{j=1}^{n} |x_{sj} - y_{tj}|.$$

  The city block distance is a special case of the Minkowski distance, where $p = 1$.

- Minkowski distance

  $$d_{st} = \sqrt[p]{\sum_{j=1}^{n} |x_{sj} - y_{tj}|^p}.$$

  For the special case of $p = 1$, the Minkowski distance gives the city block distance. For the special case of $p = 2$, the Minkowski distance gives the Euclidean distance. For the special case of $p = \infty$, the Minkowski distance gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j\{|x_{sj} - y_{tj}|\}.$$

The Chebychev distance is a special case of the Minkowski distance, where $p = \infty$.

- Cosine distance

$$d_{st} = \left(1 - \frac{x_s y'_t}{\sqrt{(x_s x'_s)(y_t y'_t)}}\right).$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \overline{x}_s)(y_t - \overline{y}_t)'}{\sqrt{(x_s - \overline{x}_s)(x_s - \overline{x}_s)'}\sqrt{(y_t - \overline{y}_t)(y_t - \overline{y}_t)'}},$$

where

$$\overline{x}_s = \frac{1}{n}\sum_j x_{sj}$$

and

$$\overline{y}_t = \frac{1}{n}\sum_j y_{tj}.$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj})/n).$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq y_{tj}) \cap ((x_{sj} \neq 0) \cup (y_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (y_{tj} \neq 0)]}.$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \overline{r}_s)(r_t - \overline{r}_t)'}{\sqrt{(r_s - \overline{r}_s)(r_s - \overline{r}_s)'}\sqrt{(r_t - \overline{r}_t)(r_t - \overline{r}_t)'}},$$

where

  - r_{sj} is the rank of x_{sj} taken over $x_{1j}, x_{2j}, \ldots x_{mx,j}$, as computed by `tiedrank`.
  - r_{tj} is the rank of y_{tj} taken over $y_{1j}, y_{2j}, \ldots y_{my,j}$, as computed by `tiedrank`.
  - r_s and r_t are the coordinate-wise rank vectors of x_s and y_t, that is, $r_s = (r_{s1}, r_{s2}, \ldots r_{sn})$ and $r_t = (r_{t1}, r_{t2}, \ldots r_{tn})$.
  - $\overline{r}_s = \frac{1}{n}\sum_j r_{sj} = \frac{(n+1)}{2}.$
  - $\overline{r}_t = \frac{1}{n}\sum_j r_{tj} = \frac{(n+1)}{2}.$

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- **GPU Code Generation**
  Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

- **GPU Arrays**
  Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

Version History

Introduced in R2010a

See Also

`pdist` | `createns` | `knnsearch` | `ExhaustiveSearcher` | `KDTreeSearcher`