

# Manual of Frovedis C++ API



# Contents

1	Introduction	1
2	frovedis::dvector<T>	3
3	frovedis::node_local<T>	27
4	frovedis::dunordered_map<K,V>	43
5	frovedis::rowmajor_matrix_local<T>	55
6	frovedis::rowmajor_matrix<T>	63
7	frovedis::colmajor_matrix_local<T>	73
8	frovedis::colmajor_matrix<T>	79
9	frovedis::sliced_colmajor_matrix_local<T>	85
10	frovedis::sliced_colmajor_vector_local<T>	89
11	blas_wrapper	95
12	lapack_wrapper	101
13	frovedis::blockcyclic_matrix<T>	115
14	frovedis::sliced_blockcyclic_matrix<T>	123
15	frovedis::sliced_blockcyclic_vector<T>	127
16	pblas_wrapper	133
17	scalapack_wrapper	141
18	frovedis::crs_matrix_local<T,I,O>	149
19	frovedis::crs_matrix<T,I,O>	157
20	frovedis::ccs_matrix_local<T,I,O>	165
21	frovedis::ccs_matrix<T,I,O>	171
22	frovedis::ell_matrix_local<T,I>	175
23	frovedis::ell_matrix<T,I>	181

24	<code>frovedis::jds_matrix_local&lt;T,I,O,P&gt;</code>	185
25	<code>frovedis::jds_matrix&lt;T,I,O,P&gt;</code>	191
26	<code>frovedis::linear_regression_model&lt;T&gt;</code>	195
27	Linear Regression	199
28	Lasso Regression	203
29	Ridge Regression	207
30	<code>frovedis::logistic_regression_model&lt;T&gt;</code>	211
31	Logistic Regression	217
32	<code>frovedis::svm_model&lt;T&gt;</code>	223
33	Linear SVM	229
34	<code>frovedis::matrix_factorization_model&lt;T&gt;</code>	235
35	Matrix Factorization using ALS	241
36	kmeans	243
37	<code>spectral_clustering_model</code>	245
38	spectral clustering	247
39	<code>spectral_embedding_model</code>	249
40	spectral embedding	251
41	Timeseries analysis using ARIMA	253

# Chapter 1

## Introduction

This manual contains C++ API documentation. If you are new to Frovedis, please read the tutorial first.

Frovedis is composed of following layers: core, matrix library, machine learning library, and data frame. Currently we only provide part of core, matrix, and machine learning API documentation. We are still updating the contents.

- Core
  - `frovedis::dvector<T>`
  - `frovedis::node_local<T>`
  - `frovedis::dunordered_map<K,V>`
- Matrix
  - `frovedis::rowmajor_matrix_local<T>`
  - `frovedis::rowmajor_matrix<T>`
  - `frovedis::colmajor_matrix_local<T>`
  - `frovedis::colmajor_matrix<T>`
  - `frovedis::sliced_colmajor_matrix_local<T>`
  - `frovedis::sliced_colmajor_vector_local<T>`
  - `blas_wrapper`
  - `lapack_wrapper`
  - `frovedis::blockcyclic_matrix<T>`
  - `frovedis::sliced_blockcyclic_matrix<T>`
  - `frovedis::sliced_blockcyclic_vector<T>`
  - `pblas_wrapper`
  - `scalapack_wrapper`
  - `frovedis::crs_matrix_local<T,I,O>`
  - `frovedis::crs_matrix<T,I,O>`
  - `frovedis::ccs_matrix_local<T,I,O>`
  - `frovedis::ccs_matrix<T,I,O>`
  - `frovedis::ell_matrix_local<T,I>`
  - `frovedis::ell_matrix<T,I>`
  - `frovedis::jds_matrix_local<T,I,O,P>`
  - `frovedis::jds_matrix<T,I,O,P>`
- Machine Learning
  - `frovedis::linear_regression_model<T>`
  - Linear Regression
  - Lasso Regression
  - Ridge Regression
  - `frovedis::logistic_regression_model<T>`
  - Logistic Regression

- `frovedis::svm_model<T>`
- Linear SVM
- `frovedis::matrix_factorization_model<T>`
- Matrix Factorization using ALS
- `kmeans`
- `spectral_clustering_model`
- spectral clustering
- `spectral_embedding_model`
- spectral embedding
- Timeseries analysis using ARIMA

## Chapter 2

# frovedis::dvector<T>

## 2.1 NAME

`frovedis::dvector<T>` - a distributed vector of type ‘T’ supported by frovedis

## 2.2 SYNOPSIS

```
#include <frovedis.hpp>
```

### 2.2.1 Constructors

```
dvector ()  
dvector (const dvector<T>& src)  
dvector (dvector<T>&& src)
```

### 2.2.2 Overloaded Operators

```
dvector<T>& operator= (const dvector<T>& src)  
dvector<T>& operator= (dvector<T>&& src)
```

### 2.2.3 Public Member Functions

```
template <class R, class F>  
dvector<R> map(const F& f);  
  
template <class R, class U, class F>  
dvector<R> map(const F& f, const node_local<U>& l1);  
  
template <class R, class U, class V, class F>  
dvector<R> map(const F& f, const node_local<U>& l1,  
              const node_local<V>& l2);  
  
template <class R, class U, class V, class W, class F>  
dvector<R> map(const F& f, const node_local<U>& l1,  
              const node_local<V>& l2, const node_local<W>& l3);
```

```

template <class R, class U, class V, class W, class X, class F>
dvector<R> map(const F& f, const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y, class F>
dvector<R> map(const F& f, const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4, const node_local<Y>& l5);

template <class R, class TT>
dvector<R> map(R(*f)(TT));

template <class R, class U, class TT, class UU>
dvector<R> map(R(*f)(TT, UU), const node_local<U>& l);

template <class R, class U, class V, class TT, class UU, class VV>
dvector<R> map(R(*f)(TT, UU, VV), const node_local<U>& l1,
              const node_local<V>& l2);

template <class R, class U, class V, class W,
          class TT, class UU, class VV, class WW>
dvector<R> map(R(*f)(TT, UU, VV, WW), const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3);

template <class R, class U, class V, class W, class X,
          class TT, class UU, class VV, class WW, class XX>
dvector<R> map(R(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y,
          class TT, class UU, class VV, class WW, class XX, class YY>
dvector<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4, const node_local<Y>& l5);

template <class F>
dvector<T>& mapv(const F& f);

template <class U, class F>
dvector<T>& mapv(const F& f, const node_local<U>& l);

template <class U, class V, class F>
dvector<T>& mapv(const F& f, const node_local<U>& l1,
              const node_local<V>& l2);

template <class U, class V, class W, class F>
dvector<T>& mapv(const F& f, const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X, class F>
dvector<T>& mapv(const F& f, const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4);

template <class U, class V, class W, class X, class Y, class F>
dvector<T>& mapv(const F& f, const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4, const node_local<Y>& l5);

```



```

template <class TT>
dvector<T>& mapv(void(*f)(TT));

template <class U, class TT, class UU>
dvector<T>& mapv(void(*f)(TT,UU), const node_local<U>& l1);

template <class U, class V, class TT, class UU, class VV>
dvector<T>& mapv(void(*f)(TT, UU, VV), const node_local<U>& l1,
               const node_local<V>& l2);

template <class U, class V, class W,
          class TT, class UU, class VV, class WW>
dvector<T>& mapv(void(*f)(TT, UU, VV, WW), const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X,
          class TT, class UU, class VV, class WW, class XX>
dvector<T>& mapv(void(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3,
               const node_local<X>& l4);

template <class U, class V, class W, class X, class Y,
          class TT, class UU, class VV, class WW, class XX, class YY>
dvector<T>& mapv(void(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3,
               const node_local<X>& l4, const node_local<Y>& l5);

template <class R, class F>
dvector<R> map_partitions(const F& f);

template <class R, class U, class F>
dvector<R> map_partitions(const F& f, const node_local<U>& l1);

template <class R, class U, class V, class F>
dvector<R> map_partitions(const F& f, const node_local<U>& l1,
               const node_local<V>& l2);

template <class R, class U, class V, class W, class F>
dvector<R> map_partitions(const F& f, const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3);

template <class R, class U, class V, class W, class X, class F>
dvector<R> map_partitions(const F& f, const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3,
               const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y, class F>
dvector<R> map_partitions(const F& f, const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3,
               const node_local<X>& l4, const node_local<Y>& l5);

template <class R, class TT>
dvector<R> map_partitions(std::vector<R>(*f)(TT));

template <class R, class U, class TT, class UU>
dvector<R> map_partitions(std::vector<R>(*f)(TT, UU),
               const node_local<U>& l1);

template <class R, class U, class V, class TT, class UU, class VV>
dvector<R> map_partitions(std::vector<R>(*f)(TT, UU, VV),
               const node_local<U>& l1, const node_local<V>& l2);

```

```

template <class R, class U, class V, class W,
         class TT, class UU, class VV, class WW>
dvector<R> mapv_partitions(std::vector<R>(*f)(TT, UU, VV, WW),
                        const node_local<U>& l1, const node_local<V>& l2,
                        const node_local<W>& l3);

template <class R, class U, class V, class W, class X,
         class TT, class UU, class VV, class WW, class XX>
dvector<R> mapv_partitions(std::vector<R>(*f)(TT, UU, VV, WW, XX),
                        const node_local<U>& l1, const node_local<V>& l2,
                        const node_local<W>& l3, const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y,
         class TT, class UU, class VV, class WW, class XX, class YY>
dvector<R> mapv_partitions(std::vector<R>(*f)(TT, UU, VV, WW, XX, YY),
                        const node_local<U>& l1, const node_local<V>& l2,
                        const node_local<W>& l3, const node_local<X>& l4,
                        const node_local<Y>& l5);

template <class F>
dvector<T>& mapv_partitions(const F& f);

template <class U, class F>
dvector<T>& mapv_partitions(const F& f, const node_local<U>& l);

template <class U, class V, class F>
dvector<T>& mapv_partitions(const F& f, const node_local<U>& l1,
                        const node_local<V>& l2);

template <class U, class V, class W, class F>
dvector<T>& mapv_partitions(const F& f, const node_local<U>& l1,
                        const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X, class F>
dvector<T>& mapv_partitions(const F& f, const node_local<U>& l1,
                        const node_local<V>& l2, const node_local<W>& l3,
                        const node_local<X>& l4);

template <class U, class V, class W, class X, class Y, class F>
dvector<T>& mapv_partitions(const F& f, const node_local<U>& l1,
                        const node_local<V>& l2, const node_local<W>& l3,
                        const node_local<X>& l4, const node_local<Y>& l5);

template <class TT>
dvector<T>& mapv_partitions(void(*f)(TT));

template <class U, class TT, class UU>
dvector<T>& mapv_partitions(void(*f)(TT, UU), const node_local<U>& l);

template <class U, class V, class TT, class UU, class VV>
dvector<T>& mapv_partitions(void(*f)(TT, UU, VV), const node_local<U>& l1,
                        const node_local<V>& l2);

template <class U, class V, class W,
         class TT, class UU, class VV, class WW>
dvector<T>& mapv_partitions(void(*f)(TT, UU, VV, WW),
                        const node_local<U>& l1, const node_local<V>& l2,
                        const node_local<W>& l3);

template <class U, class V, class W, class X,

```

```

    class TT, class UU, class VV, class WW, class XX>
dvector<T>& mapv_partitions(void(*f)(TT, UU, VV, WW, XX),
    const node_local<U>& l1, const node_local<V>& l2,
    const node_local<W>& l3, const node_local<X>& l4);

template <class U, class V, class W, class X, class Y,
    class TT, class UU, class VV, class WW, class XX, class YY>
dvector<T>& mapv_partitions(void(*f)(TT, UU, VV, WW, XX, YY),
    const node_local<U>& l1, const node_local<V>& l2,
    const node_local<W>& l3, const node_local<X>& l4,
    const node_local<Y>& l5);

template <class F> T reduce(const F& f);
template <class TT, class UU> T reduce(T(*f)(TT,UU));

template <class F> dvector<T> filter(const F& f);
template <class TT> dvector<T> filter(bool(*f)(TT));

template <class F> dvector<T>& inplace_filter(const F& f);
template <class TT> dvector<T>& inplace_filter(bool(*f)(TT));

template <class R, class F> dvector<R> flat_map(const F& f);
template <class R, class TT> dvector<R> flat_map(std::vector<R>(*f)(TT));

void clear();
std::vector<T> gather();
const std::vector<size_t>& sizes();
size_t size() const;
dvector<T>& align_as(const std::vector<size_t>& sz);
template <class U> dvector<T>& align_to(dvector<U>& target);
dvector<T>& align_block();

void save(const std::string& path, const std::string& delim);
void saveline(const std::string& path);
void savebinary(const std::string& path);
void put(size_t pos, const T& val);
T get(size_t pos);

template <class K, class V>
dunordered_map<K, std::vector<V>> group_by_key();
template <class K, class V, class F>
dunordered_map<K, V> reduce_by_key(const F& f);
template <class K, class V, class VV, class WW>
dunordered_map<K, V> reduce_by_key(V(*f)(VV, WW));

node_local<std::vector<T>> as_node_local() const;
node_local<std::vector<T>> moveto_node_local();
node_local<std::vector<T>> viewas_node_local();

dvector<T>& sort(double rate = 1.1);
template <class F> dvector<T>& sort(F f, double rate = 1.1);

```

## 2.3 DESCRIPTION

`frovedis::dvector<T>` can be considered as the distributed version of `std::vector<T>`. Memory management is similar to vector (RAII): when a `dvector` is destructed, the related distributed data is deleted at the

time. It is possible to copy or construct it from an existing dvector. In this case, distributed data is also copied (if the source variable is an rvalue, the system tries to avoid copy).

The dvector can also be created while loading data from file. When a vector of size 4, e.g., {1,2,3,4} is distributed among two worker nodes, worker0 will have {1,2} and worker1 will have {3, 4}. Frovedis supports various member functions of a dvector to develop interesting programs.

The dvector provides a global view of the distributed vector to the user. When operating on a dvector, user can simply specify the intended operation on each element of the dvector (not on each local partition of the worker data). Thus it is simpler to handle a dvector like an `std::vector`, even though it is distributed among multiple workers. Every dvector has a “size vector” attribute, containing the size of each local vectors at worker nodes. The next section explains its functionalities in details.

## 2.3.1 Constructor Documentation

### 2.3.1.1 `dvector ()`

This is the default constructor which creates an empty dvector. But it does not allocate data, like normal container. See `make_dvector_allocate()`.

### 2.3.1.2 `dvector (const dvector<T>& src)`

This is the copy constructor which creates a new dvector of type T by copying the distributed data from the input dvector.

### 2.3.1.3 `dvector (dvector<T>&& src)`

This is the move constructor. Instead of copying the input rvalue dvector, it attempts to move the contents to the newly constructed dvector. It is faster and recommended when input dvector will no longer be needed.

## 2.3.2 Overloaded Operator Documentation

### 2.3.2.1 `dvector<T>& operator= (const dvector<T>& src)`

It copies the source dvector object into the left-hand side target dvector object of the assignment operator “=”. After successful copying, it returns the reference of the target dvector object.

### 2.3.2.2 `dvector<T>& operator= (dvector<T>&& src)`

Instead of copying, it moves the contents of the source rvalue dvector object into the left-hand side target dvector object of the assignment operator “=”. It is faster and recommended when source dvector object will no longer be needed. It returns the reference of the target dvector object after the successful assignment operation.

## 2.3.3 Public Member Function Documentation

### 2.3.3.1 `map()`

The `map()` function is used to specify the target operation to be mapped on each element of a dvector. It accepts a function or a function object (functor) and applies the same to each element of the dvector in

parallel at the workers. Then a new dvector is created from the return value of the function.

Along with the function argument, `map()` can accept maximum of five distributed data of `node_local` type. This section will explain them in details.

```
dvector<R> map(R(*f)(TT));
```

Below are the points to be noted while using the above `map()` interface.

- it accepts only the function to be mapped as an argument.
- thus the input function must also not accept more than one argument.
- the type of the function argument must be same or compatible with the type of the dvector.
- the return type, R can be anything. The resultant dvector will be of the same type.

For example,

```
float func1 (float x) { return 2*x; }
float func2 (double x) { return 2*x; }
float func3 (other_type x) { return 2*x.val; }
double func4 (float x) { return 2*x; }

// let's consider "dv" is a dvector of type float
// dv is dvector<float>, func1() accepts float.
auto r1 = dv.map(func1); // Ok, r1 would be dvector<float>.

// dv is dvector<float>, func2() accepts double.
// but float is compatible with double.
auto r2 = dv.map(func2); // Ok, r2 would be dvector<float>.

// dv is dvector<float>, but func3() accepts some user type (other_type).
// even if the member "val" of "other_type" is of float type,
// it will be an error.
auto r3 = dv.map(func3); // error

// func4() accepts float (ok) and returns double,
// but no problem with return type.
auto r4 = dv.map(func4); // Ok, r4 would be dvector<double>.

// it is possible to chain the map calls
auto r5 = dv.map(func1).map(func4); // Ok, r5 would be dvector<double>.
```

In the above case, functions accepting only one argument would be allowed to pass. If more than one arguments are to be passed, different version of `map()` interface needs to be used. Frovedis supports `map()` interface which can accept a function with maximum of five arguments as follows.

```
dvector<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
              const node_local<V>& l2, const node_local<W>& l3,
              const node_local<X>& l4, const node_local<Y>& l5);
```

When using the `map()` interface accepting function to be mapped with more than one arguments, the below points are to be noted.

- the first argument of the map interface must be the function pointer to be mapped on the target dvector.
- the type of the dvector and the type of the first function argument must be of the same or of compatible type.
- the other arguments of the map (apart from the function pointer) must be of distributed `node_local<T>` type, where "T" can be of any type and the corresponding function arguments should be of the same type.
- the return type, R can be anything. The resultant dvector will be of the same type.

The mapping of the argument types of the `map()` call and the argument types of the function to be mapped on a dvector, “dv” will be as follows:

<code>func(d,x1,x2,x3,x4,x5);</code>	<code>dv.map(func,l1,l2,l3,l4,l5);</code>
-----	-----
d: T	dv: dvector<T>
x1: U	l1: node_local<U>
x2: V	l2: node_local<V>
x3: W	l3: node_local<W>
x4: X	l4: node_local<X>
x5: Y	l5: node_local<Y>

For example,

```
int func1(int x, int y) { return x+y; }
double func2(int x, float y, double z) { return x*y+z; }

// let's consider "dv" is a dvector of type "int"
// dv is dvector<int> and func1() accepts "int" as first argument. (Ok)
// But second argument of the map() is simply "int" type,
// thus it will lead to an error.
auto r1 = dv.map(func1, 2); // error

// broadcasting integer "y" to all workers to obtain node_local<int>.
int y = 2;
auto dy = broadcast(y);
auto r2 = dv.map(func1, dy); // Ok, r2 would be dvector<int>

float y = 2.0;
double z = 3.1;
auto dy = broadcast(y); // dy is node_local<float>
auto dz = broadcast(z); // dz is node_local<double>
auto r3 = dv.map(func2, dy, dz); // Ok, r3 would be dvector<double>
```

Thus there are limitations on `map()` interface. It can not accept more than five distributed parameters. And also all of the parameters (except function pointer) have to be distributed before calling `map` (can not pass non-distributed parameter).

These limitations of `map()` can be addressed with the `map()` interfaces accepting functor (function object), instead of function pointer. This section will explain them in details.

Below are the points to be noted when passing a functor (function object) in calling the `map()` function.

- the first argument of the `map()` interface must be a functor definition.  
`dvector map(const F& f);`
- the type of the dvector must be same or compatible with the type of the first argument of the overloaded “operator()” of the functor.
- apart from the functor, the `map()` interface can accept a maximum of five distributed `node_local` objects of any type as follows.  
`dvector map(const F& f, const node_local& l1,  
const node_local& l2, const node_local& l3,  
const node_local& l4, const node_local& l5);`

Where U, V, W, X, Y can be of any type and the corresponding arguments of the overloaded “operator()” must be of the same or compatible type.

- the functor itself can have any number of data members of any type and they need not to be of the distributed type and they must be specified with “SERIALIZE” macro. If the functor does not have any data members, then the “struct” definition must be ended with “SERIALIZE\_NONE” macro.
- the return type, R of the overloaded “operator()”, can be anything. The resultant dvector would be of the same type. But the type needs to be explicitly defined while calling the map() interface.

For example,

```
struct foo {
    foo() {}
    foo(float a, float b): al(a), be(b) {}
    double operator() (int x) { // 1st definition
        return al*x+be;
    }
    double operator() (int x, int y) { // 2nd definition
        return al*x+be*y;
    }
    float al, be;
    SERIALIZE(al,be)
};
```

```
// let's consider "dv" is a dvector of "int" type.
auto r1 = dv.map<double>(foo(2.0,3.0)); // ok, r1 would be dvector<double>
```

In the above call of map(), it is taking a function object with “al” and “be” values as 2.0 and 3.0 respectively. Since these are the values for initializing the members of the function object, they can be passed like a simple constructor call.

“dv” is `dvector<int>` and map() is called with only functor definition. Thus it will hit the first definition of the overloaded “operator()”. The return type is “double” which can be of any type and needs to be explicitly mentioned while calling the map() function like `map<double>()` (otherwise some compiler errors might be encountered).

Like map() with function pointer, map with function object can also accept up to five distributed node\_local objects of any type.

For example, in order to hit the 2nd definition of the overloaded “operator()” in previous foo structure, the map() function can be called as follows:

```
int be = 2;
// "be" needs to be broadcasted to all workers before calling the below
// map() function in order to get node_local<int> object. r2 would be
// dvector<double>.
auto r2 = dv.map<double>(foo(2.0,3.0),broadcast(be));
```

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

### 2.3.3.2 mapv()

The mapv() function is also used to specify the target operation to be mapped on each element of the dvector. It accepts a void returning function or a function object (functor) and applies the same to each element of the dvector in parallel at the workers. Since the applied function does not return anything, the mapv() function simply returns the reference of the source dvector itself in order to support method chaining while calling mapv().

Like `map()`, `mapv()` has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. It can not be mapped on a function which returns something other than “void”.

For example,

```
void func1(int x) { x = 2*x; // updates on temporary x local to func1() }
void func2(int& x) { x = 2*x; // in-place update }
int func3(int x) { return 2*x; }

// let's consider "dv" is a dvector of integer type.
dv.mapv(func1); // Ok, but "dv" would remain unchanged.
dv.mapv(func2); // Ok, all the elements of "dv" would be doubled.

// "dv" is dvector<int>, func3() accepts int, but it also returns int.
// thus it can not be passed to a mapv() call.
dv.mapv(func3); // error, func3() is a non-void function

// method chaining is allowed (since mapv returns reference to
// the source dvector)
auto r = dv.mapv(func2).map(func3);
```

Here the resultant dvector “r” will be of integer type and it will contain 4 times the values stored in “dv”. While mapping `func2()` on the elements of “dv”, it will double all its elements in-place and the `mapv()` will return the reference of the updated “dv” on which the `map()` function will apply the function `func3()` to double all its elements once again (not in-place) and will return a new `dvector<int>`.

### 2.3.3.3 map\_partitions()

Unlike `map()` function, `map_partitions()` accept the function or function object to be mapped on each partition of the dvector (not on each element of the dvector). Thus the input function (or functor) must accept an `std::vector` of type T (as the first argument) and must return an `std::vector` of type R. Where “T” must be the same or compatible with the type of the dvector and “R” can be of any type, the resultant dvector would be of same type.

For example,

```
std::vector<double> func(std::vector<int> part) {
    std::vector<double> ret(part.size());
    for(size_t i=0; i<part.size(); ++i) ret[i] = 2.0 * part[i];
    return ret;
}

// let's consider "dv" is a dvector of type integer.
// mapping the rule defined by func() in each partition of dv.
auto r = dv.map_partitions(func); // "r" would be of type dvector<double>
```

Like `map()` function, it also has similar rules and limitations.

- The first argument of the `map_partitions()` must be a function or function object to be mapped on each partition of the dvector.
- Apart from the function (or function object), it can accept a maximum of five distributed `node_local` objects of any type which must be same or compatible with the corresponding types of the input function arguments.
- In case more than five arguments are required to be passed, a function object can be defined by setting all the required values and can be passed to the `map_partitions()` call, as explained in earlier in the `map` section.



- Usually function object version is a bit faster, but special treatment like explicit specification of the return type etc. (as explained in map section) needs to be considered.

Usually `map_partitions()` can work faster than `map()` on SX, since in case of `map_partitions()` user needs to pass a function defining rules to be mapped on each element of a specific partition. The do-loop (to iterate over a partition) inside such functions can be vectorized, whereas `map()` allows user to simply define the rule to be mapped on each element of the dvector without any do-loop to iterate over the partition.

#### 2.3.3.4 `mapv_partitions()`

Like `map_partitions()`, it also accepts the function or function object to be mapped on each partition of the dvector (not on each element of the dvector). But in this case, the input function (or functor) must accept an `std::vector` of type T (as the first argument) where “T” must be the same or compatible with the type of the dvector and it must not return anything (void returning function).

Like `map_partitions()`, `mapv_partitions()` has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. Although it can not be mapped on a function which returns something other than “void”, the `mapv_partitions()` itself returns the reference of the source dvector in order to support method chaining.

For example,

```
void func1(std::vector<int> part) {
    // update on a temporary "part" local to the func1()
    for(size_t i=0; i<part.size(); ++i) part[i] *= 2;
}

void func2(std::vector<int>& part) {
    // in-place update on the "part" itself
    for(size_t i=0; i<part.size(); ++i) part[i] *= 2;
}

std::vector<double> func3(std::vector<int> part) {
    std::vector<double> ret(part.size());
    for(size_t i=0; i<part.size(); ++i) ret[i] = 2.0 * part[i];
    return ret;
}

// let's consider "dv" is a dvector of type integer.
// mapping the rule defined by func1() in each partition of dv.
dv.mapv_partitions(func1); // Ok, but "dv" will remain unchanged

// mapping the rule defined by func2() in each partition of dv.
dv.mapv_partitions(func2); // Ok, "dv" will get doubled in-place

// mapping the rule defined by func3() in each partition of dv.
dv.mapv_partitions(func3); // error, func3() is not a void function

// since mapv_partitions() returns the reference of the source
// dvector itself, the method chaining is possible.
auto r = dv.mapv_partitions(func2).map_partitions(func3);
```

Here the resultant dvector “r” will be of double type and it will contain 4 times the values stored in “dv”. While mapping `func2()` on the elements of “dv”, it will double all its partitions in-place and the `mapv_partitions()` will return the reference of the updated “dv” on which the `map_partitions()` function will apply the function `func3()` to double all its partitions once again (not in-place) and will return a new dvector.

### 2.3.3.5 flat\_map()

Like `map()`, `flat_map()` can also be used to map a user function on each elements of a dvector. But in this case, the user function must return more than one values in a vector form while mapping the function against each elements. It flattens the output vector returned by the user function while constructing the resultant dvector. For this reason, the size of the resultant vector is larger than the size of the source dvector.

Unlike `map()`, `flat_map()` can accept only the user function or function object to be mapped on each dvector elements. The type of the argument of the user function and the type of the dvector must be same or compatible. The return type, “R” can be anything. The resultant dvector will be of the same type.

```
dvector<R> flat_map(const F& f);
dvector<R> flat_map(std::vector<R>(*f)(TT));
```

For example,

```
// function (returning a vector) to be mapped on a dvector
std::vector<int> duplicate (int i) {
    std::vector<int> ret;
    ret.push_back(i); ret.push_back(2*i);
    return ret;
}

// let's consider "dv" is a dvector of type "int".
auto r1 = dv.map(duplicate); // r1 will be dvector<std::vector<int>>
auto r2 = dv.flat_map(duplicate); // r2 will be dvector<int>
```

Let’s consider a vector of integers {1,2,3,4} is scattered over two workers to create the dvector “dv”. Then worker0 will have {1,2} and worker1 will have {3,4}. Now in case of the resultant dvector, “r1” from “map(duplicate)” operation, worker0 will have {{1,2},{2,4}} and worker1 will have {{3,6},{4,8}} and it would be of the type `std::vector` and both “dv” and “r1” will have the same size (4 in this case). But in case of the resultant dvector, “r2” from `flat_map(duplicate)` operation, worker0 will have {1,2,2,4} and worker1 will have {3,6,4,8} and it would be of the type “int” with double the size of “dv”.

### 2.3.3.6 reduce()

It reduces all the elements in the dvector to a single scalar value, by specifying some rule to be used for reduction. The rule can be any function or function object that satisfies associative law, like min, max, sum etc. with the below signatures.

```
T reduce(const F& f);
T reduce(T(*f)(TT,UU));
```

The type of the input/output of the input function defining the rule must be same or compatible with the type of the dvector.

On success, it returns the reduced scalar value of the same type of the dvector.

For example,

```
int sum (int x, int y) { return x + y; }

// let's consider "dv" is a dvector of type "int"
// "r" would be an integer value containing the summed-up of all the elements
// in the dvector, "dv".
auto r = dv.reduce(sum);
```

**2.3.3.7 filter()**

Some specific elements from a dvector can be filtered out with the help of filter() function. It accepts a function or a function object specifying the condition on which the element is to be filtered out from the dvector. The type of the function argument must be same or compatible with the type of the dvector and the function must return a boolean value (true/false).

```
dvector<T> filter(const F& f);
dvector<T> filter(bool(*f)(TT));
```

On success, it will return a new dvector of same type containing the filtered out elements.

For example,

```
bool is_even(int n) { return n%2 == 0; }

// let's consider "dv" is a dvector of type "int"
// r will be the resultant dvector<int> containing only the even numbers
// from the target dvector "dv".
auto r = dv.filter(is_even);
```

**2.3.3.8 inplace\_filter()**

Like filter(), this function can also be used to filter out some specific elements from a dvector. But in this case the filtration happens in-place, i.e., instead of returning a new dvector, this function aims to update the source dvector by keeping only the filtered out elements in it.

Like filter(), it also accepts a function or a function object specifying the condition on which the element is to be filtered out from the dvector. The type of the function argument must be same or compatible with the type of the dvector and the function must return a boolean value (true/false).

```
dvector<T>& inplace_filter(const F& f);
dvector<T>& inplace_filter(bool(*f)(TT));
```

On success, the source dvector will be updated with only the filtered out elements in-place and this function will return a reference to itself.

For example,

```
bool is_even(int n) { return n%2 == 0; }

// let's consider "dv" is a dvector of type "int" containing both even
// and odd numbers. it will contain only the even numbers after the below
// in-place operation.
dv.inplace_filter(is_even);
```

**2.3.3.9 clear()**

In order to remove the existing elements and clear the memory space occupied by a dvector, clear() function can be used. It returns void.

**2.3.3.10 gather()**

In order to gather the distributed vector data from all the workers to master process, gather() function can be used. It returns an std::vector of type T, where “T” is the type of the dvector.

```
std::vector<T> gather();
```

Data gathering happens worker-by-worker. For example if there are two worker nodes and worker0 has {1,2} and worker2 has {3,4}, then performing `gather()` on that dvector will result in a vector containing {1,2,3,4}.

### 2.3.3.11 `sizes()`

This function returns the size vector of an existing dvector containing the size of each local vectors at worker nodes. It has the below signature:

```
const std::vector<size_t>& sizes();
```

For example if “dv” is a `dvector<int>` distributed over two worker nodes and worker0 has {1,2,3,4} and worker1 has {5,6}, then calling `sizes()` on “dv” will result in a a vector containing {4,2}, i.e., the sizes of each local vectors. The size of the output vector will be same as the number of participating worker nodes. For example if the above “dv” is distributed over four worker nodes and worker0 has {1,2,3,4}, worker1 has {5,6} and worker2 and worker3 doesn’t contain any part of this distributed vector, then calling `size()` on that dvector will result in a vector containing {4,2,0,0}.

### 2.3.3.12 `size()`

This function returns the size of the distributed vector. The signature of this function is as follows:

```
size_t size() const;
```

For example, if `std::vector<int> {1,1,0,1}` is distributed among workers to create a `dvector<int>` “dv”, then `dv.size()` will return 4.

### 2.3.3.13 `align_as()`

This function can be used to re-align the distribution of an existing dvector. It accepts an `std::vector<size_t>` containing the sizes of the local vectors as per the desired re-alignment.

The function will work well, only when below conditions are true:

- the size of the input vector must match with the number of worker nodes.
- the size of the source dvector must also match with the size of the desired re-aligned dvector.

On success, it will return a reference to the re-aligned dvector. The signature of the function is as follows:

```
dvector<T>& align_as(const std::vector<size_t>& sz);
```

For example, if `std::vector<int> {1,1,0,1}` is distributed among 2 worker nodes to create a `dvector<int>` “dv”, then

```
auto r1 = dv.sizes(); // it will return size vector of "dv" -> {2,2}
std::vector<size_t> t1 = {3,1};
auto r2 = dv.align_as(t1).sizes(); // Ok, it will return {3,1}
std::vector<size_t> t2 = {2,1,1};
// it will throw an exception, since size of t2 is 3,
// but number of workers is 2
r2 = dv.align_as(t2).sizes(); // error
std::vector<size_t> t3 = {3,2};
// it will throw an exception, since size of input dvector was 4,
// but provided size after re-alignment is 3+2 = 5
r2 = dv.align_as(t3).sizes(); // error
```

**2.3.3.14 align\_to()**

This function is used to re-align an existing dvector, “v1” according to the alignment of another existing dvector, “v2”. The type of “v1” and “v2” can differ, but their size must be same in order to perform a re-alignment.

On success, it will return the reference to the re-aligned dvector “v1”. The signature of this function is as follows:

```
template <class U> dvector<T>& align_to(dvector<U>& target);
```

For example, if size vector of “v1” is {2,2} and the size vector of “v2” is {3,1}, then

```
v1.align_to(v2);
// "v1" will get re-aligned according to "v2"
auto r = v1.sizes(); // it will return {3,1}
```

**2.3.3.15 align\_block()**

This function is used to re-align an existing dvector according to the frovedis default alignment.

If the target dvector is of the size 10 and the number of worker nodes is 4, then frovedis computes the chunk size per worker node according to the formula “ $\text{ceil}(\text{size\_of\_dvector}/\text{num\_of\_worker})$ ”, which would be evaluated as 3 in this case  $\lceil 10/4 \rceil$ .

So worker0 will contain the first 3 elements, worker1 will contain next 3 elements, worker2 will contain next 3 elements and worker3 will contain the remaining last element. Therefore, the size vector after this re-alignment will be {3,3,3,1}.

On success, it will return the reference to the re-aligned dvector. If the size vector of the target dvector already is in frovedis default alignment, then no operation will be performed. Simply the reference to the target dvector would be returned. The signature of this function is as follows:

```
dvector<T>& align_block();
```

For example, let’s consider “dv” is a `dvector<int>` of size 4 distributed among four worker nodes. Then,

```
std::vector<size_t> tmp = {2,2,0,0};
auto r1 = dv.align_as(tmp).sizes(); // it will return {2,2,0,0}
auto r2 = dv.align_block().sizes(); // it will return {1,1,1,1}
```

**2.3.3.16 save()**

In order to dump the contents of a dvector in a file in readable text form, this function can be used. The signature of this function is as follows:

```
void save(const std::string& path,
          const std::string& delim);
```

It accepts the absolute/relative path of the filename where to write the contents, along with the delimiter by which two consecutive elements of the dvector is to be separated while writing into the specified file.

For example, if “dv” is a `dvector<int>` created from a vector {1,2,3,4}, then `dv.save("./sample", "\n");` will write the content of the dvector in the specified file “sample” with a new line character after each element as follows:

```
1
2
3
4
```

**2.3.3.17 saveline()**

saveline() is a short-cut version of the function save(). While in case of save(), the delimiter value is required to be provided, saveline() writes the contents of the dvector with a new line character after each element.

The signature of this function is as follows:

```
void saveline(const std::string& path);
```

It only accepts the absolute/relative path of the file where to write the contents of the dvector. For example, saveline("./sample") is same as save("./sample", "\n").

**2.3.3.18 savebinary()**

Unlike save(), savebinary() writes the contents of the dvector in the specified file in non-readable binary (little-endian) form.

The signature of the function is as follows:

```
void savebinary(const std::string& path);
```

It only accepts the absolute/relative path of the file where to write the contents of the dvector in binary form.

**2.3.3.19 put()**

This function can be used to modify or replace any existing element of the dvector at a given position. It accepts the position (zero-based) of the type "size\_t" and the element to be inserted at that position. It has the below signature:

```
void put(size_t pos, const T& val);
```

It allows user to perform a simple assignment like operation "dv[pos] = val", where "dv" is a distributed vector. But such an operation should not be performed within a loop in order to avoid poor loop performance.

Here "pos" is the position where the element is to be put. It's value must be within 0 to size-1 of the dvector. And "val" must be of the same or compatible type with the dvector.

For example, if "dv" is a dvector<int> created from {1,2,3,4}, then

```
dv.put(2,2); // this will modify the dvector as -> {1,2,2,4}
dv.put(4,4); // error, "pos" value must be within 0 to 3
```

**2.3.3.20 get()**

This function can be used to get an existing element of the dvector from a given position (zero-based). It has the below signature:

```
T get(size_t pos);
```

It is equivalent to an indexing operation "dv[pos]", performed on a distributed vector, "dv". But such an operation should not be used within a loop in order to avoid poor loop performance.

Here "pos" is the position (0 to size-1) from which the element is to be obtained. On success, it returns the element of the given position.

For example, if "dv" is a dvector<int> created from {1,2,3,4}, then

```
auto r = dv.get(2); // "r" will contain the 3rd element of the dvector, 3
auto x = dv.get(4); // error, "pos" value must be within 0 to 3
```

**2.3.3.21 group\_by\_key()**

When a vector containing key-value pairs with key-type K and value-type V is distributed among participating worker nodes to create a `dvector<std::pair<K,V>>`, it is possible to group the values based on the unique keys in that dvector using `group_by_key()` member function.

On success, the output will be a distributed `unordered_map` like structure containing the group of values corresponding to each unique key. In `frovedis`, such an `unordered_map` is represented as `dunordered_map<K,vector<V>>` (see manual of `dunordered_map`). The signature of the function is as follows:

```
dunordered_map<K,std::vector<V>> group_by_key();
```

For example,

```
std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));
auto m = make_dvector_scatter(v).group_by_key<int,int>;
```

Here “m” is an object of `dunordered_map<int,vector<int>>` type with the below contents:

```
1: {100, 300}
2: {200, 400}
```

Note that, it will be required to explicitly specify the key-value types when calling the function, as shown in the example above `group_by_key<int,int>` (else some compilation error can be experienced).

There is also a non-member function (global function in `frovedis` namespace) which accepts the source dvector as input argument as follows:

```
dunordered_map<K,std::vector<V>> group_by_key(dvector<std::pair<K,V>& dv);
```

It can be called without explicitly specifying the key-value types of the resultant `dunordered_map`, as follows:

```
auto dv = make_dvector_scatter(v);
auto m2 = frovedis::group_by_key(dv); // no need for explicit key-value type
```

**2.3.3.22 reduce\_by\_key()**

As explained above, the values corresponding to a unique key in a `dvector<std::pair<K,V>>` can be grouped together using `group_by_key()`. Similarly the values associated with a unique key can be reduced using `reduce_by_key()` function passing a user defined reduction function (or function object) satisfying the associative laws like sum, min, max etc.

On success, it will return an object of the type `dunordered_map<K,V>` containing the distributed `unordered_map` where every key of K type will have an associated reduced value of V type.

Note that, in case of `group_by_key()`, the key-value types of the resultant `dunordered_map` are in `<K,vector<V>>` form. Whereas, in case of `reduce_by_key()`, these are in `<K,V>` form since it reduces all the values in a group associated with a unique key. The signature of the function is as follows:

```
dunordered_map<K,V> reduce_by_key(const F& f);
dunordered_map<K,V> reduce_by_key(V(*f)(VV,WW));
```

The input/output type of the user specified reduction function must be same or compatible with the value-type in the target dvector containing key-value pairs. And the resultant `dunordered_map` key-value type must be explicitly specified when calling the `reduce_by_key()` function (else some compilation error might be experienced).

For example,

```
int sum(int x, int y) { return x + y; }

int vec_sum(int k, std::vector<int>& v) {
    int res = 0; for(auto& i: v) res += i; return res;
}

std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));
auto m = make_dvector_scatter(v).reduce_by_key<int,int>(sum);
```

Here “m” is an object of `dunordered_map<int,int>` type with the below contents:

```
1: 400 -> reduced sum value of the group {100,300}
2: 600 -> reduced sum value of the group {200,400}
```

Calling `reduce_by_key()` is equivalent to the below operation:

```
auto m2 = make_dvector_scatter(v).group_by_key<int,int>()
        .map_values(vec_sum);
```

But from the implementation point of view, `reduce_by_key()` is better than calling `group_by_key()` and then reducing the vector, because the system tries to reduce the data locally at first, which reduces the communication cost.

Note that, it will be required to explicitly specify the key-value types when calling the function, as shown in the example above `reduce_by_key<int,int>`.

There is also a non-member function (global function in `frovedis` namespace) which accepts the source dvector and the reduction rule as functor (function object) as follows:

```
dunordered_map<K,V> reduce_by_key(dvector<std::pair<K,V>& dv,
                                const F& f);
```

It can be called without explicitly specifying the key-value types of the resultant `dunordered_map`, as follows:

```
struct sumtor {
    sumtor() {}
    int operator()(int x, int y) { return x + y; }
    SERIALIZE_NONE
};
auto dv = make_dvector_scatter(v);
auto m3 = frovedis::reduce_by_key(dv,sumtor()); // no need for explicit key-value type
```

### 2.3.3.23 as\_node\_local()

This function can be used to convert a `dvector<T>` to a `node_local<std::vector<T>>`, where “T” can be of any type. In this case, while converting to the `node_local` (see manual entry for `node_local`) object it copies the entire elements of the source dvector. Thus after the conversion, source dvector will remain unchanged.

The signature of the function is as follows:

```
node_local<std::vector<T>> as_node_local() const;
```

For example, if “dv” is a `dvector<int>` created from {1,2,3,4}, then



```

void display_local(const std::vector<int>& v) {
    for (auto& e: v) std::cout << e << " ";
    std::cout << std::endl;
}

void display_global(int x) {
    std::cout << x << " ";
}

void two_times_in_place(int& x) { x = 2*x; }

dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.as_node_local(); // conversion to node_local -> copy
dv.mapv(two_times_in_place); // dvector elements will get doubled
dv.mapv(display_global); // dvector elements will be printed as 2 4 6 8

// node_local elements will be printed as in original dvector 1 2 3 4
nloc.mapv(display_local);

```

### 2.3.3.24 moveto\_node\_local()

This function can be used to convert a `dvector<T>` to a `node_local<std::vector<T>>`, where “T” can be of any type. In this case, while converting to the `node_local` (see manual entry for `node_local`) object it avoids copying the data in the source dvector. Thus after the conversion, source dvector will become invalid. This is useful and faster when input `node_local` object will no longer be needed in a user program.

The signature of the function is as follows:

```
node_local<std::vector<T>> moveto_node_local();
```

For example, if “dv” is a `dvector<int>` created from {1,2,3,4}, then

```

void display_local(const std::vector<int>& v) {
    for (auto& e: v) std::cout << e << " ";
    std::cout << std::endl;
}

void display_global(int x) {
    std::cout << x << " ";
}

void two_times_in_place(int& x) { x = 2*x; }
void two_times_in_place_part(std::vector<int>& v) {
    for(auto i=0; i<v.size(); ++i) v[i] *= 2;
}

dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.moveto_node_local(); // conversion to node_local -> no copy
dv.mapv(two_times_in_place); // error, "dv" will no longer be a valid dvector

nloc.mapv(display_local); // node_local elements will be printed as 1 2 3 4
nloc.mapv(two_times_in_place_part); // node_local elements will get doubled
nloc.mapv(display_local); // node_local elements will be printed as 2 4 6 8

```

### 2.3.3.25 `viewas_node_local()`

This function can be used to create a view of a `dvector<T>` as a `node_local<std::vector<T>>`, where “T” can be of any type. Since it is about just creation of a view, the data in source dvector is neither copied nor moved. Thus it will remain unchanged after the view creation and any changes made in the source dvector will be reflected in its `node_local` view as well and the reverse is also true.

The signature of the function is as follows:

```
node_local<std::vector<T>> viewas_node_local();
```

For example, if “dv” is a `dvector<int>` created from {1,2,3,4}, then

```
void display_local(const std::vector<int>& v) {
    for (auto& e: v) std::cout << e << " ";
    std::cout << std::endl;
}

void display_global(int x) {
    std::cout << x << " ";
}

void two_times_in_place(int& x) { x = 2*x; }

dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.viewas_node_local(); // will create a node_local view
// "dv" and "nloc" both are referring to the same worker memory
// thus any changes in view "nloc" will also be reflected in source "dv"
dv.mapv(two_times_in_place); // dvector elements (in view) will get doubled
dv.mapv(display_global); // dvector elements will be printed as 2 4 6 8
nloc.mapv(display_local); // node_local elements will be printed as 2 4 6 8
```

There might be a situation when some user function expects to receive `node_local<vector<T>>` data just for reading, but input data is in `dvector<T>` form. In that case, this function will be useful just to create a `node_local` view and send to that user function for reading.

### 2.3.3.26 `sort()`

This function can be used to sort the elements of the dvector. It has the below signature:

```
dvector<T>& sort(double rate = 1.1);
```

“rate” is a double type parameter used internally while sorting of the dvector.

Frovedis also supports another version of `sort()` which accepts a function object defining the user given sort function, as follows:

```
dvector<T>& sort(F f, double rate = 1.1);
```

On success, this will sort the dvector in-place and will return a reference to the sorted dvector. Note, currently `sort()` is not supported on a view.

## 2.3.4 Public Global Function Documentation

### 2.3.4.1 `dvector<T> make_dvector_allocate()`

#### Purpose

This function is used to allocate empty vector instances of type “T” at the worker nodes to create a valid empty `dvector<T>` at master node.

The default constructor of `dvector`, does not allocate any memory at the worker nodes. Whereas, this function can be used to create a valid empty `dvector` with allocated zero-sized vector memory at worker nodes.

Note that, the intended type of the `dvector` needs to be explicitly specified while calling this function.

For example,

```
void asign_data(std::vector<int>& v) {
    // get_selfid() returns rank of the worker node
    // which will execute this function
    auto myrank = frovedis::get_selfid(); // (0 to nproc-1)
    std::vector<int> temp;
    for(int i=1; i<=2; ++i) temp.push_back(i*myrank);
    v.swap(temp);
}

void display(int e) { std::cout << e << " "; }

dvector<int> dv1; // empty dvector without any allocated memory
auto dv2 = make_dvector_allocate<int>(); // empty dvector with allocated memory
dv1.mapv(display); // error, can't display "dv1" (it is not valid).
dv2.mapv(display); // okay, an empty view
// asigning data at each allocated empty partition and display contents
// if there are two worker nodes, it will display -> 0 0 1 2
dv2.mapv_partitions(asign_data).mapv(display);
```

#### Return Value

On success, it returns the allocated `dvector<T>`.

### 2.3.4.2 `dvector<T> make_dvector_scatter(vec)`

#### Parameters

*vec*: An `std::vector<T>` containing the elements to be scattered.

#### Purpose

This function accepts a normal vector of elements of type T and scatter them to the participating worker nodes to create a `dvector<T>`. Before scattering the data it partitions the data in blocks, as explained in `align_block()` function above. The input vector will remain unchanged.

For example,

```
void display(const std::vector<int>& v) {
    auto myrank = frovedis::get_selfid();
    for (auto& e: v) std::cout << "[" << myrank << "]: " << e << "\n";
}

std::vector<int> v = {1,2,3,4};
auto dv = make_dvector_scatter(v);
dv.mapv_partitions(display);
```

If there are two worker nodes, it will output (order of the display can be different):

```
[0]: 1
[0]: 2
[1]: 3
[1]: 4
```

master	worker0	worker1
-----	-----	-----
v: vector<int> ({1,2,3,4})		
dv: dvector<int>	vector<int>: ({1,2})	vector<int>: ({3,4})

### Return Value

On success, it returns the created dvector<T>.

#### 2.3.4.3 dvector<T> make\_dvector\_loadline(path)

##### Parameters

*path*: A string object containing the path of the file to be loaded

##### Purpose

This function accepts a filename (with relative/absolute path) and loads the data from the text file to create a dvector<T> object. It expects the vector elements in the given file is separated by new-lines. If “T” is not explicitly provided, then it creates a dvector<std::string> type object. Otherwise, it creates a dvector<T>.

For example, if the text file “sample” contains below data

```
1
2
3
4
```

Then,

```
auto dv1 = make_dvector_loadline("./sample"); // dv1: dvector<std::string>
auto dv2 = make_dvector_loadline<int>("./sample"); // dv2: dvector<int>
```

### Return Value

On success, it returns the created dvector<T>.

#### 2.3.4.4 dvector<T> make\_dvector\_load(path,delim)

##### Parameters

*path*: A string object containing the path of the file to be loaded

*delim*: A string object containing the delimiter used in the file for vector elements

##### Purpose

This function accepts a filename (with relative/absolute path), along with a delimiter of string type and loads the data from the text file based on the delimiter to create a dvector<T> object. If “T” is not explicitly provided, then it creates a dvector<std::string> type object. Otherwise, it creates a dvector<T>.

For example, if the text file “sample” contains below data

```
1, 2, 3, 4
```

Then,

```
auto dv1 = make_dvector_load("./sample", ","); // dv1: dvector<std::string>
auto dv2 = make_dvector_load<int>("./sample", ","); // dv2: dvector<int>
```

Note than `make_dvector_loadline<T>(filename)`, internally calls `make_dvector_load<T>(filename, "\n")`.

**Return Value**

On success, it returns the created `dvector<T>`.

## 2.4 SEE ALSO

`node_local`, `dunordered_map`



## Chapter 3

# frovedis::node\_local<T>

### 3.1 NAME

frovedis::node\_local<T> - a distributed object of type ‘T’ stored locally at each worker nodes

### 3.2 SYNOPSIS

```
#include <frovedis.hpp>
```

#### 3.2.1 Constructors

```
node_local ()  
node_local (const node_local<T>& src)  
node_local (node_local<T>&& src)
```

#### 3.2.2 Overloaded Operators

```
node_local<T>& operator= (const node_local<T>& src)  
node_local<T>& operator= (node_local<T>&& src)
```

#### 3.2.3 Public Member Functions

```
template <class R, class F>  
node_local<R> map(const F& f);  
  
template <class R, class U, class F>  
node_local<R> map(const F& f, const node_local<U>& l1);  
  
template <class R, class U, class V, class F>  
node_local<R> map(const F& f, const node_local<U>& l1,  
                 const node_local<V>& l2);  
  
template <class R, class U, class V, class W, class F>  
node_local<R> map(const F& f, const node_local<U>& l1,  
                 const node_local<V>& l2, const node_local<W>& l3);
```

```

template <class R, class U, class V, class W, class X, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4, const node_local<Y>& l5);

template <class R, class TT>
node_local<R> map(R(*f)(TT));

template <class R, class U, class TT, class UU>
node_local<R> map(R(*f)(TT, UU), const node_local<U>& l);

template <class R, class U, class V, class TT, class UU, class VV>
node_local<R> map(R(*f)(TT, UU, VV), const node_local<U>& l1,
                 const node_local<V>& l2);

template <class R, class U, class V, class W,
          class TT, class UU, class VV, class WW>
node_local<R> map(R(*f)(TT, UU, VV, WW), const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3);

template <class R, class U, class V, class W, class X,
          class TT, class UU, class VV, class WW, class XX>
node_local<R> map(R(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y,
          class TT, class UU, class VV, class WW, class XX, class YY>
node_local<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4, const node_local<Y>& l5);

template <class F>
node_local<T>& mapv(const F& f);

template <class U, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l);

template <class U, class V, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2);

template <class U, class V, class W, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4);

template <class U, class V, class W, class X, class Y, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
                 const node_local<V>& l2, const node_local<W>& l3,
                 const node_local<X>& l4, const node_local<Y>& l5);

```



```

template <class TT>
node_local<T>& mapv(void(*f)(TT));

template <class U, class TT, class UU>
node_local<T>& mapv(void(*f)(TT,UU), const node_local<U>& l1);

template <class U, class V, class TT, class UU, class VV>
node_local<T>& mapv(void(*f)(TT, UU, VV), const node_local<U>& l1,
    const node_local<V>& l2);

template <class U, class V, class W,
    class TT, class UU, class VV, class WW>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW), const node_local<U>& l1,
    const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X,
    class TT, class UU, class VV, class WW, class XX>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
    const node_local<V>& l2, const node_local<W>& l3,
    const node_local<X>& l4);

template <class U, class V, class W, class X, class Y,
    class TT, class UU, class VV, class WW, class XX, class YY>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
    const node_local<V>& l2, const node_local<W>& l3,
    const node_local<X>& l4, const node_local<Y>& l5);

template <class F> T reduce(const F& f);
template <class TT, class UU> T reduce(T(*f)(TT,UU));

template <class F> node_local<T> allreduce(const F& f);
template <class TT, class UU> node_local<T> allreduce(T(*f)(TT,UU));

std::vector<T> gather();
T vector_sum();
void put(int n_id, const T& val);
T get(int n_id);

template <class U> dvector<U> as_dvector() const;
template <class U> dvector<U> moveto_dvector();
template <class U> dvector<U> viewas_dvector();

```

### 3.3 DESCRIPTION

Frovedis provides an efficient data structure to perform an operation locally on a distributed data either broadcasted or scattered. When a data of type “T” is broadcasted or a vector containing elements of type “vector” is scattered among worker nodes, a node local view of those data can be represented by a `node_local<T>` or a `node_local<std::vector<T>>` object respectively.

Let’s consider there are two worker nodes and an integer object containing “5” is broadcasted to them and a vector containing `{{1,2},{3,4}}` is scattered to the participating worker nodes. Then a node local view of these data can be picturized as below:

```

iData(5) -> broadcast
iVector({{1,2},{3,4}}) -> scatter

```

master	worker0	worker1
-----	-----	-----

```
d_iData      (5)      (5)
d_iVector    ({1,2})  ({3,4})
```

The `d_iData` and `d_iVector` in the above case can be considered as `node_local<int>` and `node_local<std::vector<int>>` respectively. These will provide the local view of the distributed data allowing user to perform the operations locally on each worker node in a faster and efficient way.

Such kind of data structure is useful in many machine learning algorithms, where the training process can be performed on the training data stored locally at the worker nodes in parallel and then reducing the local model to update the global model at master node etc.

Since the `node_local` provides a local view of the distributed object, a user is supposed to define the operation to be performed on each worker data (in case of a scattered vector, operation needs to be defined on each local vectors, instead of each elements like in `dvector`) in a `map()` like call. The next section explains functionalities of `node_local` in details.

### 3.3.1 Constructor Documentation

#### 3.3.1.1 `node_local ()`

This is the default constructor which creates an empty `node_local` object. But it does not allocate any memory for the container. See `make_node_local_allocate()`.

#### 3.3.1.2 `node_local (const node_local<T>& src)`

This is the copy constructor which creates a new `node_local` of type `T` by copying the distributed data from the input `node_local` object.

#### 3.3.1.3 `node_local (node_local<T>&& src)`

This is the move constructor. Instead of copying the input rvalue `node_local`, it attempts to move the contents to the newly constructed `node_local` object. It is faster and recommended when input `node_local` object will no longer be needed.

### 3.3.2 Overloaded Operator Documentation

#### 3.3.2.1 `node_local<T>& operator= (const node_local<T>& src)`

It copies the source `node_local` object into the left-hand side target `node_local` object of the assignment operator “`=`”. After successful copying, it returns the reference of the target `node_local` object.

#### 3.3.2.2 `node_local<T>& operator= (node_local<T>&& src)`

Instead of copying, it moves the contents of the source rvalue `node_local` object into the left-hand side target `node_local` object of the assignment operator “`=`”. It is faster and recommended when source `node_local` object will no longer be needed. It returns the reference of the target `node_local` object after the successful assignment operation.

### 3.3.3 Public Member Function Documentation

#### 3.3.3.1 map()

The `map()` function is used to specify the target operation to be mapped on each worker data (each node local partition) of the distributed object. It accepts a function or a function object (functor) and applies the same to each worker data in parallel. Then a new `node_local` object is created from the return value of the function.

Along with the function argument, `map()` can accept maximum of five distributed data of `node_local` type. This section will explain them in details.

```
node_local<R> map(R(*f)(TT));
```

Below are the points to be noted while using the above `map()` interface.

- it accepts only the function to be mapped as an argument.
- thus the input function must also not accept more than one arguments.
- the type of the function argument must be same or compatible with the type of the `node_local` object.
- the return type, `R` can be anything. The resultant `node_local` object will be of the same type.

For example,

```
float func1 (float x) { return 2*x; }
float func2 (double x) { return 2*x; }
float func3 (other_type x) { return 2*x.val; }
double func4 (float x) { return 2*x; }

// let's consider "nloc" is a node_local of type float
// nloc is node_local<float>, func1() accepts float.
auto r1 = nloc.map(func1); // Ok, r1 would be node_local<float>.

// nloc is node_local<float>, func2() accepts double.
// but float is compatible with double.
auto r2 = nloc.map(func2); // Ok, r2 would be node_local<float>.

// nloc is node_local<float>, but func3() accepts some user type (other_type).
// even if the member "val" of "other_type" is of float type,
// it will be an error.
auto r3 = nloc.map(func3); // error

// func4() accepts float (ok) and returns double,
// but no problem with return type.
auto r4 = nloc.map(func4); // Ok, r4 would be node_local<double>.

// it is possible to chain the map calls
auto r5 = nloc.map(func1).map(func4); // Ok, r5 would be node_local<double>.
```

In the above case, functions accepting only one argument would be allowed to pass. If more than one arguments are to be passed, different version of `map()` interface needs to be used. Frovedis supports `map()` interface which can accept a function with maximum of five arguments as follows.

```
node_local<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
               const node_local<V>& l2, const node_local<W>& l3,
               const node_local<X>& l4, const node_local<Y>& l5);
```

When using the `map()` interface accepting function to be mapped with more than one arguments, the below points are to be noted.

- the first argument of the map interface must be the function pointer to be mapped on the target `node_local`.
- the type of the `node_local` and the type of the first function argument must be of the same or of compatible type.
- the other arguments of the map (apart from the function pointer) must be of distributed `node_local<T>` type, where “T” can be of any type and the corresponding function arguments should be of the same type.
- the return type, R can be anything. The resultant `node_local` object will be of the same type.

The mapping of the argument types of the `map()` call and the argument types of the function to be mapped on a `node_local`, “nloc” will be as follows:

<code>func(d,x1,x2,x3,x4,x5);</code>	<code>nloc.map(func,l1,l2,l3,l4,l5);</code>
-----	-----
<code>d: T</code>	<code>nloc: node_local&lt;T&gt;</code>
<code>x1: U</code>	<code>l1: node_local&lt;U&gt;</code>
<code>x2: V</code>	<code>l2: node_local&lt;V&gt;</code>
<code>x3: W</code>	<code>l3: node_local&lt;W&gt;</code>
<code>x4: X</code>	<code>l4: node_local&lt;X&gt;</code>
<code>x5: Y</code>	<code>l5: node_local&lt;Y&gt;</code>

For example,

```
std::vector<int> func1(const std::vector<int>& x, int y) {
    std::vector<int> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y;
    return ret;
}

std::vector<double> func2(const std::vector<int>& x,
                        float y, double z) {
    std::vector<double> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = x[i] * y + z;
    return ret;
}

// let's consider "nloc" is a node_local of type "std::vector<int>"
// nloc is node_local<vector<int>> and func1() accepts
// "vector<int>" as first argument. (Ok)
// But second argument of the map() is simply "int" type in the below call,
// thus it will lead to an error.
auto r1 = nloc.map(func1, 2); // error

// broadcasting integer "y" to all workers to obtain node_local<int>.
int y = 2;
auto dy = broadcast(y);
auto r2 = nloc.map(func1, dy); // Ok, r2 would be node_local<vector<int>>

float y = 2.0;
double z = 3.1;
auto dy = broadcast(y); // dy is node_local<float>
auto dz = broadcast(z); // dz is node_local<double>
auto r3 = nloc.map(func2, dy, dz); // Ok, r3 would be node_local<vector<double>>
```

Thus there are limitations on `map()` interface. It can not accept more than five distributed parameters. And also all of the parameters (except function pointer) have to be distributed before calling map (can not pass non-distributed parameter).

These limitations of `map()` can be addressed with the `map()` interfaces accepting functor (function object), instead of function pointer. This section will explain them in details.

Below are the points to be noted when passing a functor (function object) in calling the `map()` function.

- the first argument of the `map()` interface must be a functor definition.

```
node_local map(const F& f);
```

- the type of the `node_local` must be same or compatible with the type of the first argument of the overloaded “operator()” of the functor.
- apart from the functor, the `map()` interface can accept a maximum of five distributed `node_local` objects of any type as follows.

```
node_local map(const F& f, const node_local& l1,
               const node_local& l2, const node_local& l3,
               const node_local& l4, const node_local& l5);
```

Where U, V, W, X, Y can be of any type and the corresponding arguments of the overloaded “operator()” must be of the same or compatible type.

- the functor itself can have any number of data members of any type and they need not to be of the distributed type and they must be specified with “SERIALIZE” macro. If the functor does not have any data members, then the “struct” definition must be ended with “SERIALIZE\_NONE” macro.
- the return type, R of the overloaded “operator()”, can be anything. The resultant `node_local` would be of the same type. But the type needs to be explicitly defined while calling the `map()` interface.

For example,

```
struct foo {
    foo() {}
    foo(float a, float b): al(a), be(b) {}
    std::vector<double> operator() (std::vector<int>& x) { // 1st definition
        std::vector<double> ret(x.size());
        for(auto i=0; i<x.size(); ++i) ret[i] = al*x[i]+be;
        return ret;
    }
    std::vector<double> operator() (std::vector<int>& x, int y) { // 2nd definition
        std::vector<double> ret(x.size());
        for(auto i=0; i<x.size(); ++i) ret[i] = al*x[i]+be*y;
        return ret;
    }
    float al, be;
    SERIALIZE(al,be)
};
```

```
// let's consider "nloc" is a node_local of "std::vector<int>" type.
// the below call will be ok, r1 would be node_local<vector<double>>
auto r1 = nloc.map<vector<double>>(foo(2.0,3.0));
```

In the above call of `map()`, it is taking a function object with “al” and “be” values as 2.0 and 3.0 respectively. Since these are the values for initializing the members of the function object, they can be passed like a simple constructor call.

“nloc” is `node_local<vector<int>>` and `map()` is called with only functor definition. Thus it will hit the first definition of the overloaded “operator()”. The return type is `std::vector<double>` which can be of any type and needs to be explicitly mentioned while calling the `map()` function like `map<vector<double>>()` (otherwise some compiler errors might be encountered).

Like `map()` with function pointer, `map` with function object can also accept up to five distributed `node_local` objects of any type.

For example, in order to hit the 2nd definition of the overloaded “`operator()`” in previous `foo` structure, the `map()` function can be called as follows:

```
int be = 2;
// "be" needs to be broadcasted to all workers before calling the below
// map() function in order to get node_local<int> object. r2 would be
// node_local<vector<double>>.
auto r2 = nloc.map<vector<double>>>(foo(2.0,3.0),broadcast(be));
```

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

Note mapping a function on a `node_local<vector<T>>` is equivalent to perform `map_partitions()` on a `dvector<T>`.

### 3.3.3.2 `mapv()`

The `mapv()` function is also used to specify the target operation to be mapped on each element of the `node_local`. It accepts a void returning function or a function object (functor) and applies the same to each worker data in parallel.

Since the applied function does not return anything, the `mapv()` function simply returns the reference of the source `node_local` itself in order to support method chaining while calling `mapv()`.

Like `map()`, `mapv()` has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. It can not be mapped on a function which returns something other than “void”.

For example,

```
void func1(int x) { x = 2*x; // updates on temporary x local to func1() }
void func2(int& x) { x = 2*x; // in-place update }
int func3(int x) { return 2*x; }

// let's consider "nloc" is a node_local of integer type.
nloc.mapv(func1); // Ok, but "nloc" would remain unchanged.
nloc.mapv(func2); // Ok, all the worker data would get doubled.

// "nloc" is node_local<int>, func3() accepts int, but it also returns int.
// thus it can not be passed to a mapv() call.
nloc.mapv(func3); // error, func3() is a non-void function

// method chaining is allowed (since mapv returns reference to
// the source node_local)
auto r = nloc.mapv(func2).map(func3);
```

Here the resultant `node_local` “`r`” will be of integer type and it will contain 4 times the values stored in “`nloc`”. While mapping `func2()` on the worker data of “`nloc`”, it will get doubled in-place and the `mapv()` will return the reference of the updated “`nloc`” on which the `map()` function will apply the function `func3()` to double all the worker data once again (not in-place) and will return a new `node_local<int>`.

### 3.3.3.3 reduce()

It reduces all the worker data of the `node_local` object, by specifying some rule to be used for reduction. The rule can be any function or function object that satisfies associative law, like min, max, sum etc. with the below signatures.

```
T reduce(const F& f);
T reduce(T(*f)(TT,UU));
```

The type of the input/output of the input function defining the rule must be same or compatible with the type of the `node_local` object.

On success, it returns the reduced value of the same type of the `node_local` object.

For example,

```
int sum (int x, int y) { return x + y; }

std::vector<int> v_sum(const std::vector<int>& x,
                     const std::vector<int>& y) {
    std::vector<int> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
    return ret;
}
```

```
// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.reduce(sum);
```

```
// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.reduce(v_sum);
```

“r1” will be the reduced integer value of all the worker data as in “nloc1”. Whereas “r2” will be the reduced integer vector of all the worker vector data as in “nloc2” as depicted below with two workers and with sample values (considering 5 is broadcasted to create “nloc1” and {{1,2},{3,4}} is scattered to create “nloc2”):

master	worker0	worker1
-----	-----	-----
nloc1: node_local<int>	int: (5)	int: (5)
nloc2: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({3,4})
r1: int -> (10)		
r2: vector<int> -> ({4,6})		

Note, reducing a `dvector<int>` will result an integer value (e.g., 10 as in above case). Whereas, reducing a `node_local<vector<int>>` will result an integer vector (e.g., {4,6} as in above case) containing sum of each elements of the worker vector data.

### 3.3.3.4 vector\_sum()

This is a short-cut function which can be used to reduce a `node_local<vector<T>>` using the associative rule of “sum”. It can not be used on a `node_local` object of type other than `vector<T>`.

For example,

```
std::vector<int> v_sum(const std::vector<int>& x,
                     const std::vector<int>& y) {
    std::vector<int> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
    return ret;
}
```

```

}

// let's consider "nloc1" is a node_local<int> and
// "nloc2" is a node_local<vector<int>>
auto l1 = nloc1.vector_sum(); // error
auto l2 = nloc2.vector_sum(); // Ok
auto l3 = nloc2.reduce(v_sum); // Ok, same as "l2"

```

### 3.3.3.5 allreduce()

allreduce() can be considered as reducing the worker data of a node\_local object and then broadcasting the reduced data to all the worker nodes to create a new node\_local object.

Like reduce(), it also aims to reduce worker data with a reduction function or function object satisfying associative law, like min, max, sum etc. The reduction happens locally in this case. It has the below signature:

```

node_local<T> reduce(const F& f);
node_local<T> reduce(T(*f)(TT,UU));

```

The type of the input/output of the input function defining the rule must be same or compatible with the type of the node\_local object.

On success, it returns a node\_local object of the same type as in the source node\_localobject, containing the reduced values at each worker nodes.

For example,

```

int sum (int x, int y) { return x + y; }

std::vector<int> v_sum(const std::vector<int>& x,
                      const std::vector<int>& y) {
    std::vector<int> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
    return ret;
}

// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.allreduce(sum);

```

```

// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.allreduce(v_sum);

```

“r1” will be a node\_local<int> object containing the reduced values at each worker node for source node\_local object “nloc1”.

Whereas “r2” will be a node\_local<vector<int>> object containing the reduced vectors at each worker node for the source node\_local object “nloc2”, as depicted below with two workers and with sample values (considering 5 is broadcasted to create “nloc1” and {{1,2},{3,4}} is scattered to create “nloc2”):

master	worker0	worker1
-----	-----	-----
nloc1: node_local<int>	int: (5)	int: (5)
nloc2: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({3,4})
r1: node_local<int>	int: (10)	int: (10)
r2: node_local<vector<int>>	vector<int>: ({4,6})	vector<int>: ({4,6})

Note that “broadcast(nloc2.reduce(v\_sum))” is same as “nloc2.allreduce(v\_sum)”. But allreduce() attempts to reduce the elements of the worker data locally, thus it is more efficient and faster.



### 3.3.3.6 gather()

In order to gather the worker data of a `node_local` object one-by-one to the master node, `gather()` function can be used. It returns an `std::vector` of type `T`, where “`T`” is the type of the `node_local` object.

```
std::vector<T> gather();
```

For example,

```
// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.gather();

// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.gather();
```

“`r1`” will be a `vector<int>` containing the gathered integers from “`nloc1`” Whereas “`r2`” will be a `vector<vector<int>>` containing the gathered integer vectors from “`nloc2`” as depicted below with two workers and with sample values (considering 5 is broadcasted to create “`nloc1`” and  $\{\{1,2\},\{3,4\}\}$  is scattered to create “`nloc2`”):

master	worker0	worker1
-----	-----	-----
nloc1: node_local<int>	int: (5)	int: (5)
nloc2: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({3,4})
r1: vector<int> -> ({5,5})		
r2: vector<vector<int>> -> ({1,2}, {3,4})		

Note, gathering a `dvector<int>` will result a `vector<int>` (e.g.,  $\{1,2,3,4\}$  as in above case). Whereas, gathering a `node_local<vector<int>>` will result a `vector<vector<int>>` (e.g.,  $\{\{1,2\},\{3,4\}\}$  as in above case) containing the vector chunk of each worker scattered data.

### 3.3.3.7 put()

This function can be used to modify or replace any existing worker data of a `node_local` object at a given position. It accepts the worker node id (zero-based) of the type “`int`” and the intended data to be inserted at that worker node for the source `node_local` object. It has the below signature:

```
void put(int nid, const T& val);
```

It allows user to perform a simple assignment like operation “`nloc[nid] = val`”, where “`nloc`” is a `node_local` object. But such an operation should not be performed within a loop in order to avoid poor loop performance.

Here “`nid`” is the worker node id associated with the source `node_local` object. It’s value must be within 0 to `nproc-1`, where “`nproc`” is the total number of participating nodes which can be obtained from “`frovedis::get_nodesize()`” call.

And “`val`” must be of the same or compatible type with the source `node_local`.

For example, if “`nloc`” is a `node_local<int>` created by broadcasting “5” among two worker nodes, then

```
// error, "nid" must be within 0 to nproc-1
nloc.put(frovedis::get_nodesize(),4);
nloc.put(0,2); // this will modify the node_local object as shown below
```

master	worker0	worker1
-----	-----	-----
nloc: node_local<int>	int: (5)	int: (5)
(modified) nloc: node_local<int>	int: (2)	int: (5)

### 3.3.3.8 get()

This function can be used to get an existing worker data from a requested worker node associated with a `node_local` object. It has the below signature:

```
T get(int nid);
```

It is equivalent to an indexing operation “`nloc[nid]`”, performed on a `node_local` object, “`nloc`”. But such an operation should not be used within a loop in order to avoid poor loop performance.

Here “`nid`” is the target node id (0 to `nproc-1`) from which the node data is to be obtained. On success, it returns the data of the given position.

For example, if “`nloc`” is a `node_local<int>` created from broadcasting “5” among two worker nodes, then

```
auto r = nloc.get(1); // "r" will contain the 2nd worker data, "5"
auto x = nloc.get(2); // error, "nid" value must be within 0 to 1
```

### 3.3.3.9 as\_dvector()

This function can be used to convert a `node_local<vector<T>>` to a `dvector<U>`, where type `T` and `U` must be same or compatible type. In this case, while converting to the `dvector` (see manual entry for `dvector`) object it copies the entire elements of the source `node_local<vector<T>>`. Thus after the conversion, source `node_local` will remain unchanged.

Note that, `dvector` conversion is possible only when the source `node_local` has vector chunk at associated worker nodes. And the type of the output `dvector` (`U`) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> as_dvector() const;
```

Let’s consider “`l1`” is a `node_local<int>` and “`l2`” is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.as_dvector<int>(); // error
auto dv2 = l2.as_dvector<int>(); // Okay
```

Now let’s consider “`nloc`” is a `node_local<vector<int>>` created from scattering `{{1,2},{3,4}}` among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }
```

```
auto dv = nloc.as_dvector<int>(); // conversion to dvector<int> -> copy
// converted dvector elements will get doubled,
// but source node_local worker data will remain unchanged
dv.mapv(two_times_in_place);
```

master	worker0	worker1
-----	-----	-----
<code>nloc: node_local&lt;vector&lt;int&gt;&gt;</code>	<code>vector&lt;int&gt;: ({1,2})</code>	<code>vector&lt;int&gt;: ({3,4})</code>
<code>(converted) dv: dvector&lt;int&gt;</code>	<code>vector&lt;int&gt;: ({1,2})</code>	<code>vector&lt;int&gt;: ({3,4})</code>
<code>(doubled) dv: dvector&lt;int&gt;</code>	<code>vector&lt;int&gt;: ({2,4})</code>	<code>vector&lt;int&gt;: ({6,8})</code>

### 3.3.3.10 moveto\_dvector()

This function can be used to convert a `node_local<vector<T>>` to a `dvector<U>`, where type `T` and `U` must be same or compatible type. In this case, while converting to the `dvector` object it avoids copying the data in the source `node_local`. Thus after the conversion, source `node_local` object will become invalid. This is useful and faster when input `node_local` object will no longer be needed in a user program.

Note that, Like as `_dvector()` in this case also, `dvector` conversion is possible only when the source `node_local` has vector chunk at associated worker nodes. And the type of the output `dvector` (U) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> moveto_dvector();
```

Let's consider "l1" is a `node_local<int>` and "l2" is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.moveto_dvector<int>(); // error
auto dv2 = l2.moveto_dvector<int>(); // Okay
```

Now let's consider "nloc" is a `node_local<vector<int>>` created from scattering `{{1,2},{3,4}}` among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }

auto dv = nloc.moveto_dvector<int>(); // conversion to dvector<int> -> move
// converted dvector elements will get doubled,
dv.mapv(two_times_in_place);
// but source node_local will become invalid
auto temp = nloc.gather(); // error (node_local data is moved, thus invalid)
```

master	worker0	worker1
-----	-----	-----
nloc: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({3,4})
(converted) dv: dvector<int>	vector<int>: ({1,2})	vector<int>: ({3,4})
nloc: node_local<vector<int>>	---	---
(doubled) dv: dvector<int>	vector<int>: ({2,4})	vector<int>: ({6,8})

### 3.3.3.11 viewas\_dvector()

This function can be used to create a view of a `node_local<vector<T>>` as a `dvector<U>`, where T and U must be of same or compatible type. Since it is about just creation of a view, the data in source `node_local` is neither copied nor moved. Thus it will remain unchanged after the view creation and any changes made in the source `node_local` will be reflected in its `dvector` view as well and the reverse is also true.

Note that, Like as `_dvector()` in this case also, `dvector` conversion is possible only when the source `node_local` has vector chunk at associated worker nodes. And the type of the output `dvector` (U) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> viewas_dvector();
```

Let's consider "l1" is a `node_local<int>` and "l2" is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.moveto_dvector<int>(); // error
auto dv2 = l2.moveto_dvector<int>(); // Okay
```

Now let's consider "nloc" is a `node_local<vector<int>>` created from scattering `{{1,2},{3,4}}` among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }

void display_local(const std::vector<int>& v) {
    for (auto& e: v) std::cout << e << " ";
    std::cout << std::endl;
}

void display_global(int x) {
    std::cout << x << " ";
}
```

```

}

nloc.mapv(display_local); // node_local elements will be printed as 1 2 3 4
auto dv = nloc.viewas_dvector<int>(); // creation of a dvector<int> view
// "dv" and "nloc" both are referring to the same worker memory
// thus any changes in view "dv" will also be reflected in source "nloc"
dv.mapv(two_times_in_place);
dv.mapv(display_global); // dvector elements will be printed as 2 4 6 8
nloc.mapv(display_local); // node_local elements will be printed as 2 4 6 8

```

There might be a situation when some user function expects to receive `dvector<T>` data just for reading, but input data is in `node_local<vector<T>>` form. In that case, this function will be useful just to create a `dvector` view and send to that user function for reading.

### 3.3.4 Public Global Function Documentation

#### 3.3.4.1 `node_local<T> make_node_local_allocate()`

##### Purpose

This function is used to allocate empty `T` type instances at the worker nodes to create a valid empty `node_local<T>` at master node.

The default constructor of `node_local`, does not allocate any memory at the worker nodes. Whereas, this function can be used to create a valid empty `node_local` with allocated memory at worker nodes.

Note that, the intended type of the `node_local` object needs to be explicitly specified while calling this function.

For example,

```

void assign_data(std::vector<int>& v) {
    // get_selfid() returns rank of the worker node
    // which will execute this function
    auto myrank = frovedis::get_selfid(); // (0 to nproc-1)
    std::vector<int> temp;
    for(int i=1; i<=2; ++i) temp.push_back(i*myrank);
    v.swap(temp);
}

void display(const std::vector<int>& v) {
    for (auto& e: v) std::cout << e << " ";
    std::cout << std::endl;
}

node_local<vector<int>> nc1; // empty node_local without any allocated memory
// empty node_local with allocated memory
auto nc2 = make_node_local_allocate<vector<int>>();
nc1.mapv(display); // error, can't display "nc1" (it is not valid).
nc2.mapv(display); // okay, an empty view
// assigning data at each allocated empty partition and display contents
// if there are two worker nodes, it will display -> 0 0 1 2
nc2.mapv(assign_data).mapv(display);

```

##### Return Value

On success, it returns the allocated `node_local<T>`.

**3.3.4.2 node\_local<T> make\_node\_local\_scatter(vec)****Parameters**

*vec*: An `std::vector<T>` containing the elements to be scattered.

**Purpose**

This function accepts a normal vector of elements of type `T` and scatter them one-by-one to each participating worker node to create a `node_local<T>`. The size of the input vector must be same with the number of participating worker nodes, else an exception will be thrown. After the scattering, The input vector will remain unchanged.

Note that, the block size of each worker partition is auto decided by the `frovedis` when scattering a `vector<T>` to create a `dvector<T>`. But when a `node_local` object is to be created by scattering a vector data, user needs to specify the same in chunk-per-worker, thus in that case the input argument has to be a `vector<vector<T>>` (instead of `vector<T>`).

For example, if there are two worker nodes, then

```
std::vector<int> v1 = {2,4};
auto nc1 = make_node_local_scatter(v1); // nc1 will be a node_local<int>
std::vector<std::vector<int>> v2 = {{1,2},{3,4}};
auto nc2 = make_node_local_scatter(v2); // nc2 will be a node_local<vector<int>>
std::vector<int> v3 = {2,4,6};
auto nc3 = make_node_local_scatter(v3); // error, vector size != worker size
```

master	worker0	worker1
-----	-----	-----
v1: vector<int> ({2,4})		
v2: vector<vector<int>> ({1,2},{3,4})		
nc1: node_local<int>	int: (2)	int: (4)
nc2: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({3,4})

**Return Value**

On success, it returns the created `node_local<T>`.

**3.3.4.3 node\_local<T> make\_node\_local\_broadcast(data)****Parameters**

*data*: A `const&` of a “`T`” type data to be broadcasted.

**Purpose**

This function accepts a `T` type data and broadcasts it to each participating worker node to create a `node_local<T>`.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2};
auto nc1 = make_node_local_broadcast(2); // nc1 will be a node_local<int>
auto nc2 = make_node_local_broadcast(v); // nc2 will be a node_local<vector<int>>
```

master	worker0	worker1
-----	-----	-----
v: vector<int> ({1,2})		
nc1: node_local<int>	int: (2)	int: (2)
nc2: node_local<vector<int>>	vector<int>: ({1,2})	vector<int>: ({1,2})

Note that, there is a short-cut method, called “`broadcast()`” to perform the same thing. For example, `make_node_local_broadcast(t)` and `broadcast(t)` both are equivalent.

**Return Value**

On success, it returns the created `node_local<T>`.

**3.4 SEE ALSO**

`dvector`, `dunordered_map`

## Chapter 4

# frovedis::dunordered\_map<K,V>

### 4.1 NAME

`frovedis::dunordered_map<K,V>` - a distributed `unordered_map` with key-type ‘K’ and value-type ‘V’ supported by `frovedis`

### 4.2 SYNOPSIS

```
#include <frovedis.hpp>
```

#### 4.2.1 Constructors

```
dunordered_map ()  
dunordered_map (const dunordered_map<K,V>& src)  
dunordered_map (dunordered_map<K,V>&& src)
```

#### 4.2.2 Overloaded Operators

```
dunordered_map<K,V>& operator= (const dunordered_map<K,V>& src)  
dunordered_map<K,V>& operator= (dunordered_map<K,V>&& src)
```

#### 4.2.3 Public Member Functions

```
template <class R, class F>  
dunordered_map<K,R> map_values(const F& f);  
  
template <class R, class U, class F>  
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1);  
  
template <class R, class U, class W, class F>  
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,  
                             const node_local<W>& l2);
```

```

template <class R, class U, class W, class X, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3);

template <class R, class U, class W, class X, class Y, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4);

template <class R, class U, class W, class X, class Y, class Z, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4, const node_local<Z>& l5);

template <class R, class KK, class VV>
dunordered_map<K,R> map_values(R(*f)(KK, VV));

template <class R, class U, class KK, class VV, class UU>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU), const node_local<U>& l);

template <class R, class U, class W, class KK, class VV, class UU, class WW>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW), const node_local<U>& l1,
    const node_local<W>& l2);

template <class R, class U, class W, class X,
    class KK, class VV, class UU, class WW, class XX>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3);

template <class R, class U, class W, class X, class Y,
    class KK, class VV, class UU, class WW, class XX, class YY>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4);

template <class R, class U, class W, class X, class Y, class Z,
    class KK, class VV, class UU, class WW, class XX, class YY, class ZZ>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY,ZZ), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4, const node_local<Z>& l5);

template <class F>
dunordered_map<K,V>& mapv(const F& f);

template <class U, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l);

template <class U, class W, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
    const node_local<W>& l2);

template <class U, class W, class X, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3);

template <class U, class W, class X, class Y, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4);

```



```

template <class U, class W, class X, class Y, class Z, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4, const node_local<Z>& l5);

template <class KK, class VV>
dunordered_map<K,V>& mapv(void(*f)(KK,VV));

template <class U, class KK, class VV, class UU>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU), const node_local<U>& l);

template <class U, class W, class KK, class VV, class UU, class WW>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW), const node_local<U>& l1,
    const node_local<W>& l2);

template <class U, class W, class X,
    class KK, class VV, class UU, class WW, class XX>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3);

template <class U, class W, class X, class Y,
    class KK, class VV, class UU, class WW, class XX, class YY>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX,YY), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4);

template <class U, class W, class X, class Y, class Z,
    class KK, class VV, class UU, class WW, class XX, class YY, class ZZ>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX,YY,ZZ), const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4, const node_local<Z>& l5);

template <class F> dunordered_map<K,V> filter(const F& f);
template <class KK, class VV> dunordered_map<K,V> filter(bool(*f)(KK,VV));

template <class F> dunordered_map<K,V>& inplace_filter(const F& f);
template <class KK, class VV> dunordered_map<K,V>& inplace_filter(bool(*f)(KK,VV));

void clear();
size_t size();

void put(const K& key, const V& val);
V get(const K& key);
V get(const K& key, bool& found);

dvector<std::pair<K,V>> as_dvector();
node_local<MAP<K,V>> as_node_local();
node_local<MAP<K,V>> moveto_node_local();
node_local<MAP<K,V>> viewas_node_local();

```

## 4.3 DESCRIPTION

`frovedis::dunordered_map<K,V>` can be considered as the distributed version of `std::unordered_map<K,V>`. Memory management is similar to `unordered_map` (RAII): when a `dunordered_map` is destructed, the related distributed data is deleted at the time. It is possible to copy or construct it from an existing `dunordered_map`. In this case, distributed data is also copied (if the source variable is an rvalue, the system tries to avoid copy).

In `dunordered_map`, each item (Key-Value pair) is distributed according to the hash value of the Key. In addition, the Key should be unique just like `unordered_map` (not `multimap`).

Usually, `dunordered_map` is created from a `dvector` (see manual of `dvector`), whose actual type should be `dvector<std::pair<K,V>>` by performing `group_by_key()` or `reduce_by_key()` like operations.

Like `dvector`, `dunordered_map` provides a global view of the distributed unordered map to the user. When operating on a `dunordered_map`, user can simply specify the intended operation to be performed on each Key of the `dunordered_map` (not on each local partition of the worker data). Thus it is simpler to handle a `dunordered_map` like an `std::unordered_map`, even though it is distributed among multiple workers. The next section explains functionalities on a `dunordered_map` in details.

## 4.3.1 Constructor Documentation

### 4.3.1.1 `dunordered_map ()`

This is the default constructor which creates an empty `dunordered_map`. But it does not allocate data, like normal container. See `make_dunordered_map_allocate()`.

### 4.3.1.2 `dunordered_map (const dunordered_map<K,V>& src)`

This is the copy constructor which creates a new `dunordered_map` with key-type `K` and value-type `V` by copying the distributed data from the input `dunordered_map`.

### 4.3.1.3 `dunordered_map (dunordered_map<K,V>&& src)`

This is the move constructor. Instead of copying the input rvalue `dunordered_map`, it attempts to move the contents to the newly constructed `dunordered_map`. It is faster and recommended when input `dunordered_map` will no longer be needed.

## 4.3.2 Overloaded Operator Documentation

### 4.3.2.1 `dunordered_map<K,V>& operator= (const dunordered_map<K,V>& src)`

It copies the source `dunordered_map` object into the left-hand side target `dunordered_map` object of the assignment operator “`=`”. After successful copying, it returns the reference of the target `dunordered_map` object.

### 4.3.2.2 `dunordered_map<K,V>& operator= (dunordered_map<K,V>&& src)`

Instead of copying, it moves the contents of the source rvalue `dunordered_map` object into the left-hand side target `dunordered_map` object of the assignment operator “`=`”. It is faster and recommended when source `dunordered_map` object will no longer be needed. It returns the reference of the target `dunordered_map` object after the successful assignment operation.

### 4.3.3 Public Member Function Documentation

#### 4.3.3.1 map\_values()

The `map_values()` function is used to specify the target operation to be mapped on each Key of a `dunordered_map`. It accepts a function or a function object (functor) and applies the same to each Key of the `dunordered_map` in parallel at the workers. Then a new `dunordered_map` is created from the return value of the function.

Along with the function argument, `map_values()` can accept maximum of five distributed data of `node_local` type. This section will explain them in details.

```
dunordered_map<K,R> map_values(R(*f)(KK,VV));
```

Below are the points to be noted while using the above `map_values()` interface.

- it accepts only the function to be mapped on each key as an argument.
- the input function must accept a key parameter of type `KK` and a value parameter of type `VV`, where `KK` and `VV` must be same or compatible with `K` and `V` (the key and value type of the target `dunordered_map`).
- the return type, `R` can be anything. The value type of the resultant `dunordered_map` will be of the same type. The key type will remain same.

For example,

```
std::vector<int> func1(int k, std::vector<int>& v) {
    std::vector<int> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}
std::vector<float> func2(int k, std::vector<int>& v) {
    std::vector<float> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}
std::vector<float> func3(int k, std::vector<float>& v) {
    std::vector<float> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}

std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));

// m would be a unordered_map<int,std::vector<int>>
auto m = make_dvector_scatter(v).group_by_key<int,int>();
auto m2 = m.map_values(func1); // ok, m2: unordered_map<int,vector<int>>
auto m3 = m.map_values(func2); // ok, m3: unordered_map<int,vector<float>>
auto m4 = m.map_values(func3); // error

// it is possible to chain the map_values calls
// ok, m5: unordered_map<int,vector<float>>
auto m5 = m.map_values(func2).map_values(func3);
```

“m” is `unordered_map<int,vector<int>>`,

`func1()` expects `(int,vector<int>)` -> OK and returns `vector<int>` -> OK. Resultant `unordered_map`, “m2” becomes `unordered_map<int,vector<int>>`.

`func2()` expects `(int,vector<int>)` -> OK and returns `vector<float>` -> OK (return value-type can differ). Resultant `unordered_map`, “m3” becomes `unordered_map<int,vector<float>>`.

func3() expects (int,vector<float>) -> vector<int> and vector<float> are incompatible, thus it will lead to a compilation error.

Result of “m.map\_values(func2)” is `dunordered_map<int,vector<float>>` and func3() expects (int,vector<float>) -> thus no issues. func3() returns vector<float>, thus “m5” becomes `dunordered_map<int,vector<float>>`.

Note that, the key parameter “k” was not used in any of the above input functions for map\_values(). But this is required to map the functions on each key of the source unordered\_map objects.

In the above case, functions accepting only two arguments (key and value) would be allowed to pass. If any other arguments are to be passed, different version of map\_values() interface needs to be used. Frovedis supports map\_values() interface which can accept a function with maximum of five arguments as follows.

```
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY,ZZ),
    const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4, const node_local<Z>& l5);
```

When using the map\_values() interface accepting function to be mapped with more than two arguments (arguments other than key and values), the below points are to be noted.

- the first argument of the map\_values interface must be the function pointer to be mapped on the target unordered\_map.
- the key and value type of the unordered\_map and the type of the first two function arguments must be of the same or of compatible type.
- the other arguments of the map\_values (apart from the function pointer) must be of distributed `node_local<T>` type, where “T” can be of any type and the corresponding function arguments should be of the same type.
- the return type, R can be anything. The value type of the resultant unordered\_map will be of the same type. The key type will remain same.

The mapping of the argument types of the map\_values() call and the argument types of the function to be mapped on a unordered\_map, “um” will be as follows:

<pre>func(key,val,x1,x2,x3,x4,x5); ----- key: K, val: V x1: U x2: W x3: X x4: Y x5: Z</pre>	<pre>um.map_values(func,l1,l2,l3,l4,l5); ----- dv: unordered_map&lt;K,V&gt; l1: node_local&lt;U&gt; l2: node_local&lt;W&gt; l3: node_local&lt;X&gt; l4: node_local&lt;Y&gt; l5: node_local&lt;Z&gt;</pre>
---	---

For example,

```
std::vector<int> func1(int k, std::vector<int>& v, int n) {
    std::vector<int> tmp; for(auto& i: v) tmp.push_back(n*i); return tmp;
}

// let's consider "m" is a unordered_map<int,vector<int>>
// key-value type of "m" and type of the first two arguments of func1() -> Ok
// But third argument of the map_values() is simply "int" type,
// thus it will lead to an error.
auto m1 = m.map_values(func1, 2); // error

// broadcasting "2" to all workers to obtain node_local<int>.
// m2: unordered_map<int,vector<int>>
auto m2 = m.map_values(func1, broadcast(2)); // Ok
```

Thus there are limitations on `map_values()` interface. It can not accept more than five distributed parameters. And also all of the parameters (except function pointer) have to be distributed before calling `map` (can not pass non-distributed parameter).

These limitations of `map_values()` can be addressed with the `map_values()` interfaces accepting functor (function object), instead of function pointer. This section will explain them in details.

Below are the points to be noted when passing a functor (function object) in calling the `map_values()` function.

- the first argument of the `map_values()` interface must be a functor definition.
- the key-value type of the `dunordered_map` must be same or compatible with the type of the first two arguments of the overloaded “operator()” of the functor.
- apart from the functor, the `map_values()` interface can accept a maximum of five distributed `node_local` objects of any type as follows.

```
dunordered_map<K,R> map_values(const F& f, const node_local& l1,
const node_local& l2, const node_local& l3,
const node_local& l4, const node_local& l5);
```

Where U, W, X, Y, Z can be of any type and the corresponding arguments of the overloaded “operator()” must be of the same or compatible type.

- the functor itself can have any number of data members of any type and they need not to be of the distributed type and they must be specified with “SERIALIZE” macro. If the functor does not have any data members, then the “struct” definition must be ended with “SERIALIZE\_NONE” macro.
- the return type, R of the overloaded “operator()”, can be anything. The value-type of resultant `dunordered_map` would be of the same type. The key-type will remain same. But the value-type needs to be explicitly defined while calling the `map_values()` interface.

For example,

```
struct foo {
    foo() {}
    foo(int n_): n(n_) {}
    std::vector<int> operator() (int k, std::vector<int>& v) {
        std::vector<int> tmp; for(auto& i: v) tmp.push_back(n*i); return tmp;
    }
    int n;
    SERIALIZE(n)
};
```

```
// let's consider "m" is a dunordered_map<int,vector<int>>
auto m1 = m.map_values(foo(2)); // error in type deduction
auto m2 = m.map_values<vector<int>>>(foo(2)); // ok
```

In the above call of `map_values()`, it is taking a function object with “n” value as 2. Since it is the value for initializing the member of the function object, it can be passed like a simple constructor call.

“m” is `dunordered_map<int,vector<int>>` and `map_values()` is called with only functor definition (operator() accepting int and `vector<int>`). Thus it will be fine. Return type is of operator() is `vector<int>` which can be of any type and needs to be explicitly mentioned while calling the `map_values()` function like `map<vector<int>>>()` (otherwise some compiler errors might be encountered).

Like `map_values()` with function pointer, `map` with function object can also accept up to five distributed `node_local` objects of any type.

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

#### 4.3.3.2 mapv()

The `mapv()` function is also used to specify the target operation to be mapped on each key of the `dunordered_map`. It accepts a void returning function or a function object (functor) and applies the same to each key of the `dunordered_map` in parallel at the workers. Since the applied function does not return anything, the `mapv()` function simply returns the reference of the source `dunordered_map` itself in order to support method chaining while calling `mapv()`.

Like `map_values()`, `mapv()` has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. It can not be mapped on a function which returns something other than “void”.

For example,

```
void func1(int k, std::vector<int> v) {
    for(auto i=0; i<v.size(); ++i) v[i] *= 2; // updates on temporary v local to func1()
}
void func2(int k, std::vector<int>& v) {
    for(auto i=0; i<v.size(); ++i) v[i] *= 2; // in-place update
}
std::vector<int> func3(int k, std::vector<int> v) {
    std::vector<int> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}

// let's consider "m" is a unordered_map<int,vector<int>>
m.mapv(func1); // Ok, but "m" would remain unchanged.
m.mapv(func2); // Ok, all the values of "m" associated with a key would be doubled.
m.mapv(func3); // error, func3() is a non-void function

// method chaining is allowed (since mapv returns reference to
// the source unordered_map)
auto r = dv.mapv(func2).map_values(func3); // Ok
```

Here the resultant `dunordered_map` “r” will be of `<int,vector<int>>` type and all its values associated with a particular key will contain 4 times of the initial values. While mapping `func2()` on the keys of “m”, its associated values will be doubled in-place and the `mapv()` will return the reference of the updated “m” on which the next `map_values()` function will apply the function `func3()` to double values associated with each key once again (not in-place) and will return a new `dunordered_map<int,vector<int>>`.

#### 4.3.3.3 filter()

Some specific values from a `dunordered_map` can be filtered out with the help of `filter()` function. It accepts a function or a function object specifying the condition on which the value is to be filtered out from the `dunordered_map`. The type of the function arguments must be same or compatible with the key-value type of the `dunordered_map` and the function must return a boolean value (true/false).

```
dunordered_map<K,V> filter(const F& f);
dunordered_map<K,V> filter(bool(*f)(KK,VV));
```

On success, it will return a new `dunordered_map` of same key-value type containing the filtered out elements.

For example,

```
bool is_even(int k, std::vector<int>& v) { return k%2 == 0; }

// let's consider "m" is a dunordered_map<int,vector<int>>
// r will be the resultant dunordered_map<int,vector<int>> containing only
// the values for the keys with even numbers in "m".
auto r = m.filter(is_even);
```

#### 4.3.3.4 inplace\_filter()

Like filter(), this function can also be used to filter out some specific values from a dunordered\_map. But in this case the filtration happens in-place, i.e., instead of returning a new dunordered\_map, this function aims to update the source dunordered\_map by keeping only the filtered out values in it.

Like filter(), it also accepts a function or a function object specifying the condition on which the value is to be filtered out from the dunordered\_map. The type of the function arguments must be same or compatible with the key-value type of the dunordered\_map and the function must return a boolean value (true/false).

```
dunordered_map<K,V>& inplace_filter(const F& f);
dunordered_map<K,V>& inplace_filter(bool(*f)(KK,VV));
```

On success, the source dunordered\_map will be updated with only the filtered out values in-place and this function will return a reference to the updated dunordered\_map.

For example,

```
bool is_even(int k, std::vector<int>& v) { return k%2 == 0; }

// let's consider "m" is a dunordered_map<int,vector<int>> containing both
// even and odd keys. it will contain only the values associated with even
// keys after the below in-place filtration.
m.inplace_filter(is_even);
```

#### 4.3.3.5 clear()

In order to remove the existing elements and clear the memory space occupied by a dunordered\_map, clear() function can be used. It returns void.

#### 4.3.3.6 size()

This function returns the size of the distributed unordered\_map, i.e., the number of unique keys present in the source dunordered\_map as “size\_t” parameter.

For example,

```
std::vector<std::pair<int,int>> v1;
v1.push_back(make_pair(1,100));
v1.push_back(make_pair(2,200));
v1.push_back(make_pair(3,300));
v1.push_back(make_pair(4,400));

std::vector<std::pair<int,int>> v2;
v2.push_back(make_pair(1,100));
v2.push_back(make_pair(2,200));
v2.push_back(make_pair(1,300));
v2.push_back(make_pair(2,400));
```

```
std::cout << make_dvector_scatter(v1).group_by_key<int,int>.size(); // -> 4
std::cout << make_dvector_scatter(v2).group_by_key<int,int>.size(); // -> 2
```

#### 4.3.3.7 put()

This function can be used to modify a value associated with an existing key or insert a value with a new key in the source `dunordered_map`. It has the below signature:

```
void put(const K& key, const V& val);
```

It allows user to perform simple map assignment like operation “`m[key] = val`”, where “`m`” is a distributed `unordered_map`. But such an operation should not be performed within a loop in order to avoid poor loop performance.

Here “key” can be either ‘an existing key’ or ‘a new key’ and “val” is the intended value ‘to be modified with’ or ‘to be inserted in’ the map. Types of the given key and value must be same or compatible with the key-value types of the source `dunordered_map`.

For example, if “`m`” is a `dunordered_map<int,int>`, then “`m.put(2,5)`” will either modify the value associated with key “2” as “5” or insert a new key “2” with associated value “5”.

#### 4.3.3.8 get()

This function can be used to get the value associated with a given key in the source `dunordered_map`.

On success, if the given key exists, it returns the associated value of type “V”. But if the key does not exist, it returns the default value of type “V” (i.e., `V()`). It has the below signature:

```
V get(const K& key);
```

It is equivalent to an indexing operation “`m[key]`”, performed on a distributed `unordered_map`, “`m`”. But such an operation should not be used within a loop in order to avoid poor loop performance.

For example, if “`m`” is a `dunordered_map<int,int>` and its associated value with key “2” is “5” and there is no entry for the key “3”, then

```
auto r = m.get(2); // "r" will contain 5
auto x = m.get(3); // "x" will contain 0 (considering default integer value)
```

But it might happen that for a key “4” the associated value itself is “0”. Then,

```
// y will contain 0, but it would be unknown whether the key "4" exists.
auto y = m.get(4);
```

In that case, a special interface of “`get()`” with the below signature is provided:

```
V get(const K& key, bool& found);
```

The second boolean parameter will be reflected (passed-by-reference) based on the existence of the given key. For example,

```
bool flag = false;
auto y = m.get(4,flag); // y: 0, flag: true -> key "4" exists with value "0"
auto z = m.get(3,flag); // z: 0, flag: false -> key "3" does not exist
```



#### 4.3.3.9 as\_dvector()

A `dunordered_map` can be considered as the distributed version of the `std::unordered_map` containing the key-value pairs. Now in order to convert a `dunordered_map<K,V>` to a `dvector<std::pair<K,V>>`, member function `as_dvector()` can be used on the source `dunordered_map`. The source `dunordered_map` will remain unchanged after the `dvector` conversion. The signature of the function is as follows:

```
dvector<std::pair<K,V>> as_dvector();
```

For example, if there is a `dunordered_map<int,int>` “m” containing the below elements:

```
1: 100
2: 200
3: 300
4: 400
```

Then,

```
auto dv = m.as_dvector(); // dv: dvector<std::pair<int,int>> (copy)
auto v = dv.gather(); // v: vector<int> -> {(1,100),(2,200),(3,300),(4,400)}
```

Note that, there is no `gather()` method provided on a `dunordered_map`. When gathering of the data will be required, it needs to be converted to a `dvector` object first and then `gather()` on the converted `dvector` object can be called.

#### 4.3.3.10 as\_node\_local()

This function can be used to convert a `dunordered_map<K,V>` to a `node_local<MAP<K,V>>`, where `MAP` can be either a ‘`std::map`’ or a ‘`std::unordered_map`’ depending upon the user configuration (`USE_ORDERED_MAP` macro is defined or not) in `config.hpp` file. While converting to the `node_local` (see manual entry for `node_local`) object it copies the entire elements of the source `dunordered_map`. Thus after the conversion, source `dunordered_map` will remain unchanged. The signature of the function is as follows:

```
node_local<MAP<K,V>> as_node_local();
```

#### 4.3.3.11 moveto\_node\_local()

This function can be used to convert a `dunordered_map<K,V>` to a `node_local<MAP<K,V>>`, where `MAP` can be either a ‘`std::map`’ or a ‘`std::unordered_map`’ depending upon the user configuration (`USE_ORDERED_MAP` macro is defined or not) in `config.hpp` file. While converting to the `node_local` object, it avoids copying the data. Thus the source `dunordered_map` will become invalid after the conversion. This is faster and recommended to use when source `dunordered_map` will no longer be used in a user program. The signature of the function is as follows:

```
node_local<MAP<K,V>> moveto_node_local();
```

#### 4.3.3.12 views\_node\_local()

This function can be used to create a view of a `dunordered_map<K,V>` as a `node_local<MAP<K,V>>`, where `MAP` can be either a ‘`std::map`’ or a ‘`std::unordered_map`’ depending upon the user configuration (`USE_ORDERED_MAP` macro is defined or not) in `config.hpp` file. Since it is about just creation of a view, the data in source `dunordered_map` is neither copied nor moved. Thus it will remain unchanged after the view creation and any changes made in the source `dunordered_map` will be reflected in its `node_local` view as well and the reverse is also true. The signature of the function is as follows:

```
node_local<MAP<K,V>> views_node_local();
```

### 4.3.4 Public Global Function Documentation

#### 4.3.4.1 `dunordered_map<K,V> make_dunordered_map_allocate()`

**Purpose**

This function is used to allocate empty `unordered_map` instances with key-type “K” and value-type “V” at the worker nodes to create a valid empty `dunordered_map<K,V>` at master node.

The default constructor of `dunordered_map`, does not allocate any memory at the worker nodes. Whereas, this function can be used to create a valid empty `dunordered_map` with allocated zero-sized map memory at worker nodes.

Note that, the intended key-value types needs to be explicitly mentioned while calling this function.

For example,

```
dunordered_map<int,int> m1; // empty dunordered_map without any allocated memory
```

```
// empty dunordered_map with allocated memory  
auto m2 = make_dunordered_map_allocate<int,int>();
```

```
m1.put(1,5); // error, can't insert key-value pair in map (it is not valid)  
m2.put(1,5); // Ok, a key "1" with associated value "5" will be inserted
```

**Return Value**

On success, it returns the allocated `dunordered_map<K,V>`.

## 4.4 SEE ALSO

`dvector`, `node_local`

## Chapter 5

# frovedis::rowmajor\_matrix\_local<T>

### 5.1 NAME

`frovedis::rowmajor_matrix_local<T>` - A two-dimensional dense matrix with elements stored in row-wise order supported by `frovedis`

### 5.2 SYNOPSIS

```
#include <frovedis/matrix/rowmajor_matrix.hpp>
```

#### 5.2.1 Constructors

```
rowmajor_matrix_local ();  
rowmajor_matrix_local (size_t nrow, size_t ncol);  
rowmajor_matrix_local (const rowmajor_matrix_local<T>& m);  
rowmajor_matrix_local (rowmajor_matrix_local<T>&& m);  
rowmajor_matrix_local (const std::vector<T>& v);  
rowmajor_matrix_local (std::vector<T>&& v);
```

#### 5.2.2 Overloaded Operators

```
rowmajor_matrix_local<T>& operator= (const rowmajor_matrix_local<T>& m);  
rowmajor_matrix_local<T>& operator= (rowmajor_matrix_local<T>&& m);
```

#### 5.2.3 Public Member Functions

```
void set_local_num (size_t nrow, size_t ncol);  
void save (const std::string& file);  
void savebinary (const std::string& dir);  
void debug_print ();  
rowmajor_matrix_local<T> transpose () const;  
node_local<rowmajor_matrix_local<T>> broadcast();
```

### 5.2.4 Public Data Members

```
std::vector<T> val;
size_t local_num_row;
size_t local_num_col;
```

## 5.3 DESCRIPTION

`rowmajor_matrix_local<T>` is a template based non-distributed row-major data storage supported by `frovedis`.

Although it provides a 2D row-major storage view to the user, internally the matrix elements are stored in 1D vector form with additional row and column number information stored separately. The structure of this class is as follows:

```
template <class T>
struct rowmajor_matrix_local {
    std::vector<T> val;      // to contain matrix elements in 1D rowmajor form
    size_t local_num_row;    // number of rows in 2D matrix view
    size_t local_num_col;    // number of columns in 2D matrix view
};
```

### 5.3.1 Constructor Documentation

#### 5.3.1.1 `rowmajor_matrix_local ()`

This is the default constructor which creates an empty rowmajor matrix with `local_num_row = local_num_col = 0`.

#### 5.3.1.2 `rowmajor_matrix_local (size_t nrow, size_t ncol)`

This is the parameterized constructor which creates an empty rowmajor matrix of the given dimension (memory allocation takes place).

#### 5.3.1.3 `rowmajor_matrix_local (const rowmajor_matrix_local<T>& m)`

This is the copy constructor which creates a new rowmajor matrix by deep-copying the contents of the input rowmajor matrix.

#### 5.3.1.4 `rowmajor_matrix_local (rowmajor_matrix_local<T>&& m)`

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

#### 5.3.1.5 `rowmajor_matrix_local (const std::vector<T>& v)`

This is a special constructor for implicit conversion. It converts an input lvalue `std::vector<T>` to `rowmajor_matrix_local<T>` with dimensions  $N \times 1$ , where  $N$  = size of the input vector. It attempts to copy the input vector during the conversion. Thus input vector remains unchanged.

**5.3.1.6 rowmajor\_matrix\_local (std::vector<T>&& v)**

This is a special constructor for implicit conversion. It converts an input rvalue `std::vector<T>` to `rowmajor_matrix_local<T>` with dimensions  $N \times 1$ , where  $N = \text{size of the input vector}$ . It attempts to move the elements from the input vector during the conversion. Thus input vector will contain unknown values after the conversion.

**5.3.2 Overloaded Operator Documentation****5.3.2.1 rowmajor\_matrix\_local<T>& operator= (const rowmajor\_matrix\_local<T>& m)**

It deep-copies the input rowmajor matrix into the left-hand side matrix of the assignment operator “=”.

**5.3.2.2 rowmajor\_matrix\_local<T>& operator= (rowmajor\_matrix\_local<T>&& m)**

Instead of copying, it moves the contents of the input rvalue rowmajor matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**5.3.3 Public Member Function Documentation****5.3.3.1 void set\_local\_num (size\_t nrow, size\_t ncol)**

It sets the matrix information related to number of rows and number of columns as specified by the user. It assumes the user will provide the valid matrix dimension according to the number of elements in it. Thus no validity check is performed on the provided dimension values.

**5.3.3.2 void debug\_print ()**

It prints the contents and other information related to the matrix on the user terminal. It is mainly useful for debugging purpose.

For example,

```
std::vector<int> v = {1,2,3,4};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(2,2); // nrow: 2, ncol:2
m.debug_print();
```

The above program will output:

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 2 3 4
```

**5.3.3.3 rowmajor\_matrix\_local<T> transpose ()**

It returns the transposed `rowmajor_matrix_local` of the source matrix object.

For example,

```
std::vector<int> v = {1,2,3,4};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(2,2); // nrow: 2, ncol:2
std::cout << m.transpose(); // a rowmajor matrix can be printed on user terminal
```

It will output like:

```
1 3
2 4
```

#### 5.3.3.4 void save (const std::string& file)

It writes the elements of a rowmajor matrix to the specified file in rowmajor format with text data.

#### 5.3.3.5 void savebinary (const std::string& dir)

It writes the elements of a rowmajor matrix to the specified directory in rowmajor format with binary data.

The output directory will contain two files, named “nums” and “val” respectively. “nums” is a text file containing the number of rows and number of columns information in first two lines of the file. And “val” is a binary file containing the matrix elements stored in little-endian form.

#### 5.3.3.6 node\_local<rowmajor\_matrix\_local<T>> broadcast();

It broadcasts the source `rowmajor_matrix_local<T>` to all the participating worker nodes. After successful broadcasting, it returns a `node_local<rowmajor_matrix_local<T>>` object representing the broadcasted matrices at each worker nodes.

It is equivalent to broadcasting the matrix using `frovedis::broadcast()` (explained in `node_local` manual). But from performance point of view this is efficient as it avoids the internal serialization overhead of the vector elements.

For example,

```
std::vector<int> v = {1,2,3,4};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(2,2); // nrow: 2, ncol:2
auto bm1 = m.broadcast(); // faster
auto bm2 = frovedis::broadcast(m); // slower (serialization overhead)
```

master	worker0	worker1
-----	-----	-----
m: rowmajor_matrix_local<int>		
1 2		
3 4		
bm1: node_local<		
rowmajor_matrix_local<int>>	rowmajor_matrix_local<int>	rowmajor_matrix_local<int>
	1 2	1 2
	3 4	3 4
bm2: node_local<		
rowmajor_matrix_local<int>>	rowmajor_matrix_local<int>	rowmajor_matrix_local<int>
	1 2	1 2

3 4

3 4

### 5.3.4 Public Data Member Documentation

#### 5.3.4.1 `val`

An instance of `std::vector<T>` type to contain the elements of the matrix in 1D row-major form.

#### 5.3.4.2 `local_num_row`

A `size_t` attribute to contain the number of rows in the 2D matrix view.

#### 5.3.4.3 `local_num_col`

A `size_t` attribute to contain the number of columns in the 2D matrix view.

### 5.3.5 Public Global Function Documentation

#### 5.3.5.1 `rowmajor_matrix_local<T> make_rowmajor_matrix_local_load(filename)`

##### Parameters

*filename*: A string object containing the name of the text file having the data to be loaded.

##### Purpose

This function loads the text data from the specified file and creates a `rowmajor_matrix_local<T>` object filling the data loaded.

It assumes that there is no empty lines in the input file. The desired type of the matrix (e.g., int, float, double etc.) is to be explicitly specified when loading the matrix data from reading a file.

For example, considering “./data” is a text file having the data to be loaded,

```
auto m1 = make_rowmajor_matrix_local_load<int>("./data");
auto m2 = make_rowmajor_matrix_local_load<float>("./data");
```

“m1” will be a `rowmajor_matrix_local<int>`, whereas “m2” will be a `rowmajor_matrix_local<float>`.

##### Return Value

On success, it returns the created matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.

#### 5.3.5.2 `rowmajor_matrix_local<T> make_rowmajor_matrix_local_loadbinary(dirname)`

##### Parameters

*dirname*: A string object containing the name of the directory having the data to be loaded. It expects two files “nums” and “val” to be presented in the input directory, where “nums” is the text file containing number of rows and number of columns information (new line separated) and “val” is the little-endian binary data to be loaded.

##### Purpose

This function loads the binary data from the specified directory and creates a `rowmajor_matrix_local<T>` object filling the data loaded. The desired type of the matrix (e.g., int, float, double etc.) is to be explicitly specified when loading the matrix data from reading a file.

For example, considering “./bin” is a binary file having the data to be loaded,

```
auto m1 = make_rowmajor_matrix_local_loadbinary<int>("./bin");
auto m2 = make_rowmajor_matrix_local_loadbinary<float>("./bin");
```

“m1” will be a `rowmajor_matrix_local<int>`, whereas “m2” will be a `rowmajor_matrix_local<float>`.

### Return Value

On success, it returns the created matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.

#### 5.3.5.3 `std::ostream& operator<<(str, mat)`

##### Parameters

*str*: A `std::ostream&` object representing the output stream buffer.

*mat*: A `const&` object of the type `rowmajor_matrix_local<T>` containing the matrix to be handled.

##### Purpose

This function writes the contents of the matrix in 2D row-major matrix form in the given output stream. Thus a rowmajor matrix can simply be printed on the user terminal as “`std::cout « mat`”, where “mat” is the input matrix.

### Return Value

On success, it returns a reference to the output stream.

#### 5.3.5.4 `std::istream& operator>>(str, mat)`

##### Parameters

*str*: A `std::istream&` object representing the input stream buffer.

*mat*: A `const&` object of the type `rowmajor_matrix_local<T>` to be filled.

##### Purpose

This function reads the data from the input stream and writes the same in the given matrix. Each new-line character in the given stream is considered as a new row. The number of columns is automatically calculated based on the read elements count in each line of the input stream (it assumes that all the lines contain same number of elements).

Here the matrix “mat” is overwritten with the data read from the input stream. Thus any prior data in the matrix “mat” would be lost. Thus a rowmajor matrix can simply be read from standard input terminal as “`std::cin » mat`”, where “mat” is the matrix to be filled with data read from “`std::cin`”.

### Return Value

On success, it returns a reference to the input stream.

#### 5.3.5.5 `rowmajor_matrix_local<T> operator*(m1,m2)`

##### Parameters

*m1*: A `const&` object of the type `rowmajor_matrix_local<T>`.

*m2*: Another `const&` object of the type `rowmajor_matrix_local<T>`.

##### Purpose

This function performs matrix multiplication between two input `rowmajor_matrix_local` objects of the same type.

### Return Value

If the input matrix conforms matrix multiplication rule (number of columns in m1 matches with the number of rows in m2), then it returns the resultant rowmajor matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.



**5.3.5.6 rowmajor\_matrix\_local<T> operator\*(m1,m2)****Parameters**

*m1*: A const& object of the type `rowmajor_matrix_local<T>`.

*m2*: A const& object of the type `diag_matrix_local<T>`.

**Purpose**

When multiplying a rowmajor matrix with a diagonal matrix (e.g., unit matrix etc.), actually every column of the input rowmajor matrix is multiplied by every diagonal element of the input diagonal matrix, as depicted below.

```

-----
                2  1  5
                *  *  *
1 2 3      2 0 0      1 2 3      2 2 15
4 5 6      * 0 1 0 =>  4 5 6 =>  8 5 30
7 8 9      0 0 5      7 8 9      14 8 45
-----

```

Thus `frovedis` provides an efficient overloaded `operator*()` to handle such situation. In case of diagonal matrix, it only stores the diagonal elements (e.g., 2, 3, 5) in a data structure called `diag_matrix_local<T>` (see `diag_matrix_local` manual) and the overloaded `operator*()` simply multiplies each column of the input rowmajor matrix with each diagonal element.

**Return Value**

If number of columns in the input rowmajor matrix equals to the number of diagonal elements in the input diagonal matrix, it returns the resultant rowmajor matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.

**5.4 SEE ALSO**

`diag_matrix_local`, `colmajor_matrix_local`, `rowmajor_matrix`



## Chapter 6

# frovedis::rowmajor\_matrix<T>

### 6.1 NAME

`frovedis::rowmajor_matrix<T>` - A distributed two-dimensional dense matrix with elements stored in row-wise order supported by frovedis

### 6.2 SYNOPSIS

```
#include <frovedis/matrix/rowmajor_matrix.hpp>
```

#### 6.2.1 Constructors

```
rowmajor_matrix ();  
rowmajor_matrix (frovedis::node_local<rowmajor_matrix_local<T>>&& data);
```

#### 6.2.2 Public Member Functions

```
void set_num (size_t nrow, size_t ncol);  
void save (const std::string& file);  
void savebinary (const std::string& dir);  
void debug_print ();  
rowmajor_matrix<T> transpose () const;  
rowmajor_matrix_local<T> gather();  
rowmajor_matrix<T>& align_as(const std::vector<size_t>& sz);  
template <class U> rowmajor_matrix<T>& align_to(rowmajor_matrix<U>& m);  
rowmajor_matrix<T>& align_block();
```

#### 6.2.3 Public Data Members

```
frovedis::node_local<rowmajor_matrix_local<T>> data  
size_t num_row  
size_t num_col
```

## 6.3 DESCRIPTION

`rowmajor_matrix<T>` is a template based two-dimensional dense matrix with elements stored in row-major order and distributed among the participating worker nodes in row-wise.

A `rowmajor_matrix<T>` contains public member “data” of the type `node_local<rowmajor_matrix_local<T>>`. The actual distributed matrices are contained in all the worker nodes locally, thus named as `rowmajor_matrix_local<T>` (see manual of `rowmajor_matrix_local`) and “data” is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original matrix.

```
template <class T>
struct rowmajor_matrix {
    frovedis::node_local<rowmajor_matrix_local<T>> data; // local matrix information
    size_t num_row; // number of rows in global matrix
    size_t num_col; // number of columns in global matrix
};
```

For example, if the below row-major matrix with 4 rows and 4 columns is distributed over two worker nodes, then the distribution can be shown as:

```
1 2 3 4
5 6 7 8
8 7 6 5
4 3 2 1
```

master	worker0	worker1
-----	-----	-----
<code>rowmajor_matrix&lt;int&gt;</code>	<code>-&gt; rowmajor_matrix</code>	<code>-&gt; rowmajor_matrix</code>
	<code>    _local&lt;int&gt;</code>	<code>    _local&lt;int&gt;</code>
<code>*data: node_local&lt;</code>	<code>val: vector&lt;int&gt;</code>	<code>val: vector&lt;int&gt;</code>
<code>    rowmajor_matrix</code>	<code>    ({1,2,3,4,</code>	<code>    ({8,7,6,5,</code>
<code>    _local&lt;int&gt;&gt;&gt;</code>	<code>        5,6,7,8})</code>	<code>        4,3,2,1})</code>
<code>num_row: size_t (4)</code>	<code>local_num_row: size_t (2)</code>	<code>local_num_row: size_t (2)</code>
<code>num_col: size_t (4)</code>	<code>local_num_col: size_t (2)</code>	<code>local_num_col: size_t (2)</code>

The `node_local<rowmajor_matrix_local<int>>` object “data” is simply a (\*)handle of the (->)local matrices at worker nodes.

### 6.3.1 Constructor Documentation

#### 6.3.1.1 `rowmajor_matrix ()`

This is the default constructor which creates an empty distributed rowmajor matrix without any memory allocation at worker nodes.

#### 6.3.1.2 `rowmajor_matrix (frovedis::node_local<rowmajor_matrix_local<T>>&& data)`

This is the parameterized constructor which accepts an rvalue of the type `node_local<rowmajor_matrix_local<T>>` and *moves* the contents to the created rowmajor matrix.

In general, this constructor is used internally by some other functions. But user may need this constructor while constructing their own rowmajor matrix using the return value of some function (returning a `rowmajor_matrix_local<T>`) called using “`frovedis::node_local::map`” (thus returned value would be an object of type `node_local<rowmajor_matrix_local<T>`).

For example,

```
// --- a sample functor definition ---
struct foo {
    foo() {}
    foo(int r, int c): nrow(r), ncol(c) {}
    rowmajor_matrix_local<int> operator()(std::vector<int>& v) {
        rowmajor_matrix_local<int> ret;
        ret.val.swap(v);
        ret.set_local_num(nrow,ncol);
        return ret;
    }
    size_t nrow, ncol;
    SERIALIZE(nrow, ncol)
};

size_t sum(size_t x, size_t y) { return x + y; }
size_t get_nrows(rowmajor_matrix_local<int>& m) { return m.local_num_row; }
size_t get_ncols(rowmajor_matrix_local<int>& m) { return m.local_num_col; }

std::vector<int> v = {1,2,3,4,5,6,7,8}; // 4x2 rowmajor storage
auto bv = broadcast(v);
// demo of such a constructor call
rowmajor_matrix<int> m(bv.map<rowmajor_matrix_local<int>>(foo(4,2))); //m: 8x2
// getting total number of rows in the global matrix
m.num_row = m.data.map(get_nrows).reduce(sum); // 4+4 = 8
m.num_col = m.data.map(get_ncols).get(0);      // 2
```

The above program will perform the below tasks in order

- broadcast a vector containing sample elements of a 4x2 rowmajor\_matrix\_local.
- local rowmajor matrices will be created in worker nodes when the functor would be called.
- “bv.map<rowmajor\_matrix\_local<int>>(foo(4,2))” will return a node\_local<rowmajor\_matrix\_local<int>> object.
- the constructor call will be made for rowmajor\_matrix passing the above rvalue node\_local object.
- total number of rows will be set by summing local\_num\_row of all worker matrices.
- total number of columns will be set as per the number of columns in the worker0 matrix (from any worker will be fine).

## 6.3.2 Public Member Function Documentation

### 6.3.2.1 void set\_num (size\_t nrow, size\_t ncol)

It sets the global matrix information related to number of rows and number of columns as specified by the user. It assumes the user will provide the valid matrix dimension according to the number of elements in it. Thus no validity check is performed on the provided dimension values.

### 6.3.2.2 void debug\_print ()

It prints the contents and other information of the local matrices node-by-node on the user terminal. It is mainly useful for debugging purpose.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2,3,4,5,6,7,8};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(4,2); // m: 4x2 rowmajor matrix
// it scatters a dense rowmajor matrix
// in order to create the distributed rowmajor matrix
auto gm = make_rowmajor_matrix_scatter(m);
gm.debug_print();
```

The above program will output (order of display might differ):

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 2 3 4
node = 1, local_num_row = 2, local_num_col = 2, val = 5 6 7 8
```

### 6.3.2.3 rowmajor\_matrix<T> transpose ()

It constructs the transposed matrix of the source distributed rowmajor\_matrix object and returns the same.

### 6.3.2.4 rowmajor\_matrix\_local<T> gather ()

It gathers the local matrices from the worker nodes and constructs the original dense matrix at master node.

On success, it returns the constructed local matrix of the type rowmajor\_matrix\_local<T>, where T is the type of the distributed matrix.

### 6.3.2.5 void save (const std::string& file)

It writes the elements of the global rowmajor matrix to the specified file in rowmajor format with text data.

### 6.3.2.6 void savebinary (const std::string& dir)

It writes the elements of the global rowmajor matrix to the specified directory in rowmajor format with binary data.

The output directory will contain three files, named “nums”, “val”, and “type” respectively. “nums” is a text file containing the number of rows and number of columns information in first two lines of the file. And “val” is a binary file containing the matrix elements stored in little-endian form. The file “type” is a text file containing the type information of val.

### 6.3.2.7 rowmajor\_matrix<T>& align\_as(const std::vector<size\_t>& sz)

This function can be used to re-align the distribution of an existing rowmajor matrix. It accepts an std::vector<size\_t> containing the desired distribution, i.e., number of rows to be distributed per worker node.

The function will work well, only when below conditions are true:

- the size of the input vector must match with the number of worker nodes.
- the total number of rows in the source rowmajor matrix (to be re-aligned) must match with the sum-total value provided in the input vector.

On success, it will return a reference to the re-aligned rowmajor\_matrix.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2,3,4,5,6,7,8};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(4,2); // m: 4x2 matrix
auto gm = make_rowmajor_matrix_scatter(m);
gm.debug_print();
std::vector<size_t> new_sizes = {3,1};
gm.align_as(new_sizes); // Ok
gm.debug_print();
```

The above program will output (display order might differ):

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 2 3 4
node = 1, local_num_row = 2, local_num_col = 2, val = 5 6 7 8
node = 0, local_num_row = 3, local_num_col = 2, val = 1 2 3 4 5 6
node = 1, local_num_row = 1, local_num_col = 2, val = 7 8
```

But the below cases will lead to a runtime error:

```
new_sizes = {2,1};
gm.align_as(new_sizes); // error, sumtotal (2+1=3) != num_row (4)
new_sizes = {2,1,1};
gm.align_as(new_sizes); // error, input vector size (3) != worker size (2)
```

### 6.3.2.8 rowmajor\_matrix<T>& align\_to(rowmajor\_matrix<U>& m)

This function is used to re-align an existing rowmajor matrix, “m1” according to the distribution alignment of another existing rowmajor\_matrix, “m2”. The type of “m1” and “m2” can differ, but their total number of row count must be same in order to perform the re-alignment.

On success, it will return a reference to the re-aligned matrix “m1”.

For example,

```
std::vector<int> v1 = {1,2,3,4};
std::vector<int> v2 = {1,2,3,4,5,6,7,8};
std::vector<double> v3 = {1,2,3,4,5,6,7,8};

rowmajor_matrix_local<int> m1, m2;
rowmajor_matrix_local<double> m3

m1.val.swap(v1);
m1.set_local_num(2,2); // m1: 2x2 matrix (type: int)
m2.val.swap(v2);
m2.set_local_num(4,2); // m2: 4x2 matrix (type: int)
m3.val.swap(v3);
m3.set_local_num(4,2); // m3: 4x2 matrix (type: double)

auto gm1 = make_rowmajor_matrix_scatter(m1);
auto gm2 = make_rowmajor_matrix_scatter(m2);
auto gm3 = make_rowmajor_matrix_scatter(m3);

gm2.align_to(gm3); // ok, type differs, but total num of rows matches
gm2.align_to(gm1); // error, type matches, but total num of rows differs
```

### 6.3.2.9 rowmajor\_matrix<T>& align\_block()

This function is used to re-align an existing rowmajor matrix according to the frovedis default distribution block alignment.

If total number of rows in the target matrix is 5 and the number of worker nodes is 2, then frovedis computes the number of rows to be distributed per worker node according to the formula “ $\text{ceil}(\text{total\_num\_rows}/\text{num\_of\_worker})$ ”, which would be evaluated as 3 in this case [ $\text{ceil}(5/2)$ ]. So worker0 will contain the first 3 rows and worker1 will contain next 2 rows.

On success, it will return the reference to the re-aligned rowmajor matrix. If the source matrix is already distributed according to frovedis default block alignment, then no operation will be performed. Simply the reference to the target rowmajor matrix would be returned.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(5,2); // m: 5x2 rowmajor matrix
auto gm = make_rowmajor_matrix_scatter(m);
gm.debug_print(); // original distribution
std::vector<int> new_sizes = {4,1};
gm.align_as(new_sizes);
gm.debug_print(); // 4,1 distribution
gm.align_block();
gm.debug_print(); // default block distribution (as in original -> 3,2)
```

The above program will output (display order might differ):

```
node = 0, local_num_row = 3, local_num_col = 2, val = 1 2 3 4 5 6
node = 1, local_num_row = 2, local_num_col = 2, val = 7 8 9 10
node = 0, local_num_row = 4, local_num_col = 2, val = 1 2 3 4 5 6 7 8
node = 1, local_num_row = 1, local_num_col = 2, val = 9 10
node = 0, local_num_row = 3, local_num_col = 2, val = 1 2 3 4 5 6
node = 1, local_num_row = 2, local_num_col = 2, val = 7 8 9 10
```

## 6.3.3 Public Data Member Documentation

### 6.3.3.1 data

An instance of `node_local<rowmajor_matrix_local<T>>` type to contain the reference information related to local matrices at worker nodes.

### 6.3.3.2 num\_row

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

### 6.3.3.3 num\_col

A `size_t` attribute to contain the total number of columns in the 2D matrix view.



## 6.3.4 Public Global Function Documentation

### 6.3.4.1 `rowmajor_matrix<T> make_rowmajor_matrix_load(filename)`

#### Parameters

*filename*: A string object containing the name of the text file having the data to be loaded.

#### Purpose

This function loads the text data from the specified file and creates the distributed `rowmajor_matrix<T>` object filling the data loaded. It assumes that there is no empty lines in the input file. The desired type of the matrix (e.g., int, float, double etc.) is to be explicitly specified when loading the matrix data from reading a file.

For example, considering “./data” is a text file having the data to be loaded,

```
auto m1 = make_rowmajor_matrix_load<int>("./data");
auto m2 = make_rowmajor_matrix_load<float>("./data");
```

“m1” will be a `rowmajor_matrix<int>`, whereas “m2” will be a `rowmajor_matrix<float>`.

#### Return Value

On success, it returns the created matrix of the type `rowmajor_matrix<T>`. Otherwise, it throws an exception.

### 6.3.4.2 `rowmajor_matrix<T> make_rowmajor_matrix_loadbinary(dirname)`

#### Parameters

*dirname*: A string object containing the name of the directory having the data to be loaded. It expects two files “nums” and “val” to be presented in the input directory, where “nums” is the text file containing number of rows and number of columns information (new line separated) and “val” is the little-endian binary data to be loaded.

#### Purpose

This function loads the binary data from the specified directory and creates the distributed `rowmajor_matrix<T>` object filling the data loaded. The desired type of the matrix (e.g., int, float, double etc.) is to be explicitly specified when loading the matrix data from reading a file.

For example, considering “./bin” is a binary file having the data to be loaded,

```
auto m1 = make_rowmajor_matrix_loadbinary<int>("./bin");
auto m2 = make_rowmajor_matrix_loadbinary<float>("./bin");
```

“m1” will be a `rowmajor_matrix<int>`, whereas “m2” will be a `rowmajor_matrix<float>`.

#### Return Value

On success, it returns the created matrix of the type `rowmajor_matrix<T>`. Otherwise, it throws an exception.

### 6.3.4.3 `rowmajor_matrix<T> make_rowmajor_matrix_scatter(mat)`

#### Parameters

*mat*: A `const&` of a `rowmajor_matrix_local<T>` object containing the data to be scattered among worker nodes.

#### Purpose

This function accepts a `rowmajor_matrix_local<T>` object and row-wise scatters the elements to the participating worker nodes to create a distributed `rowmajor_matrix<T>` object. During the scatter operation, it follows frovedis default distribution block alignment (see `rowmajor_matrix::as_block()` for details).

#### Return Value

On success, it returns the created matrix of the type `rowmajor_matrix<T>`. Otherwise, it throws an exception.

#### 6.3.4.4 rowmajor\_matrix<T> make\_rowmajor\_matrix\_scatter(mat,dst)

##### Parameters

*mat*: A const& of a rowmajor\_matrix\_local<T> object containing the data to be scattered among worker nodes.

*dst*: A vector of “size\_t” elements containing the number of rows to be scattered per worker node.

##### Purpose

This function accepts a rowmajor\_matrix\_local<T> object and row-wise scatters the elements to the participating worker nodes according to the specified number of rows per worker in the input “dst” vector to create a distributed rowmajor\_matrix<T> object.

This function will work well, only when below conditions are true:

- the size of the input vector must match with the number of worker nodes.
- the total number of rows in the source local matrix, “mat” (to be scattered) must match with the sum-total value provided in the input vector, “dst”.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2,3,4,5,6,7,8};
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(4,2); // m: 4x2 matrix
auto gm1 = make_rowmajor_matrix_scatter(m); //ok, an usual scatter operation
gm1.debug_print();
std::vector<size_t> new_sizes = {3,1};
auto gm2 = make_rowmajor_matrix_scatter(m,new_sizes); //ok, nrow == sumtotal
gm2.debug_print();
```

The above program will output (display order might differ):

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 2 3 4
node = 1, local_num_row = 2, local_num_col = 2, val = 5 6 7 8
node = 0, local_num_row = 3, local_num_col = 2, val = 1 2 3 4 5 6
node = 1, local_num_row = 1, local_num_col = 2, val = 7 8
```

But the below cases will lead to a runtime error:

```
new_sizes = {2,1};
auto gm3 = make_rowmajor_matrix_scatter(m,
    new_sizes); //error, nrow (4) != sumtotal (2+1=3)
new_sizes = {2,1,1};
auto gm4 = make_rowmajor_matrix_scatter(m,
    new_sizes); //error, input vector size (3) != worker size (2)
```

##### Return Value

On success, it returns the created matrix of the type rowmajor\_matrix<T>. Otherwise, it throws an exception.

#### 6.3.4.5 std::ostream& operator<<(str, mat)

##### Parameters

*str*: A std::ostream& object representing the output stream buffer.

*mat*: A const& object of the type rowmajor\_matrix<T> containing the matrix to be handled.

##### Purpose

This function writes the contents of the matrix in 2D row-major matrix form in the given output stream. Thus a distributed rowmajor matrix can simply be printed on the user terminal as “std::cout « mat”, where

“mat” is the input matrix. In this case, it first gathers the local matrices from the worker nodes and then writes them one-by-one on the output stream.

**Return Value**

On success, it returns a reference to the output stream.

## 6.4 SEE ALSO

rowmajor\_matrix\_local, colmajor\_matrix, blockcyclic\_matrix



## Chapter 7

# frovedis::colmajor\_matrix\_local<T>

### 7.1 NAME

`frovedis::colmajor_matrix_local<T>` - A two-dimensional dense matrix with elements stored in column-wise order supported by `frovedis`

### 7.2 SYNOPSIS

```
#include <frovedis/matrix/colmajor_matrix.hpp>
```

#### 7.2.1 Constructors

```
colmajor_matrix_local ();  
colmajor_matrix_local (size_t nrow, size_t ncol);  
colmajor_matrix_local (const colmajor_matrix_local<T>& m);  
colmajor_matrix_local (colmajor_matrix_local<T>&& m);  
colmajor_matrix_local (const rowmajor_matrix_local<T>& m);
```

#### 7.2.2 Overloaded Operators

```
colmajor_matrix_local<T>& operator= (const colmajor_matrix_local<T>& m);  
colmajor_matrix_local<T>& operator= (colmajor_matrix_local<T>&& m);
```

#### 7.2.3 Public Member Functions

```
rowmajor_matrix_local<T> to_rowmajor();  
rowmajor_matrix_local<T> moveto_rowmajor();  
colmajor_matrix_local<T> transpose () const;  
node_local<colmajor_matrix_local<T>> broadcast();  
void debug_print ();
```

### 7.2.4 Public Data Members

```
std::vector<T> val;
size_t local_num_row;
size_t local_num_col;
```

## 7.3 DESCRIPTION

`colmajor_matrix_local<T>` is a template based non-distributed column-major data storage supported by `frovedis`.

Although it provides a 2D column-major storage view to the user, internally the matrix elements are stored in 1D vector form with additional row and column number information stored separately. The structure of this class is as follows:

```
template <class T>
struct colmajor_matrix_local {
    std::vector<T> val;        // to contain matrix elements in 1D colmajor form
    size_t local_num_row;     // number of rows in 2D matrix view
    size_t local_num_col;     // number of columns in 2D matrix view
};
```

A `colmajor_matrix_local` can be created from a `rowmajor_matrix_local` object and it can be converted back to the `rowmajor_matrix_local` object. Thus loading from file, saving into file etc. interfaces are not provided for `colmajor_matrix_local` structure. User may like to perform the conversion from/to `rowmajor_matrix_local` structure for the same.

### 7.3.1 Constructor Documentation

#### 7.3.1.1 `colmajor_matrix_local ()`

This is the default constructor which creates an empty `colmajor` matrix with `local_num_row = local_num_col = 0`.

#### 7.3.1.2 `colmajor_matrix_local (size_t nrow, size_t ncol)`

This is the parameterized constructor which creates an empty `colmajor` matrix of the given dimension (memory allocation takes place).

#### 7.3.1.3 `colmajor_matrix_local (const colmajor_matrix_local<T>& m)`

This is the copy constructor which creates a new `colmajor` matrix by deep-copying the contents of the input `colmajor` matrix.

#### 7.3.1.4 `colmajor_matrix_local (colmajor_matrix_local<T>&& m)`

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**7.3.1.5 colmajor\_matrix\_local (const rowmajor\_matrix\_local<T>& m)**

It accepts a rowmajor\_matrix\_local object and constructs an equivalent colmajor\_matrix\_local object by simply changing the storage order of the elements in input matrix. Number of rows and number of columns will be same in both the input matrix and constructed colmajor matrix.

**7.3.2 Overloaded Operator Documentation****7.3.2.1 colmajor\_matrix\_local<T>& operator= (const colmajor\_matrix\_local<T>& m)**

It deep-copies the input colmajor matrix into the left-hand side matrix of the assignment operator “=”.

**7.3.2.2 colmajor\_matrix\_local<T>& operator= (colmajor\_matrix\_local<T>&& m)**

Instead of copying, it moves the contents of the input rvalue colmajor matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**7.3.3 Public Member Function Documentation****7.3.3.1 void debug\_print ()**

It prints the contents and other information related to the matrix on the user terminal. It is mainly useful for debugging purpose.

For example,

```
std::vector<int> v = {1,3,2,4}; //desired storage
colmajor_matrix_local<int> m;
m.val.swap(v);
m.local_num_row = 2;
m.local_num_col = 2;
m.debug_print();
```

The above program will output:

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 3 2 4
```

**7.3.3.2 colmajor\_matrix\_local<T> transpose ()**

It returns the transposed colmajor\_matrix\_local of the source matrix object.

For example,

```
std::vector<int> v = {1,3,2,4};
colmajor_matrix_local<int> m;
m.val.swap(v);
m.local_num_row = 2;
m.local_num_col = 2;
m.transpose().debug_print();
```

The above program will output:

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 2 3 4
```

### 7.3.3.3 rowmajor\_matrix\_local<T> to\_rowmajor();

It converts the colmajor storage of the target matrix to a rowmajor storage and returns the output `rowmajor_matrix_local<T>` after successful conversion. The target colmajor storage remains unchanged after the conversion.

### 7.3.3.4 rowmajor\_matrix\_local<T> moveto\_rowmajor();

If the target matrix has only a single column, then rowmajor storage and column major storage both will be the same. Thus instead of any conversion overhead, elements in target matrix can simply be moved while creating the `rowmajor_matrix_local` object. It is faster and recommended, only when the target matrix is no longer be needed in a user program.

### 7.3.3.5 node\_local<colmajor\_matrix\_local<T>> broadcast();

It broadcasts the source `colmajor_matrix_local<T>` to all the participating worker nodes. After successful broadcasting, it returns a `node_local<colmajor_matrix_local<T>>` object representing the broadcasted matrices at each worker nodes.

It is equivalent to broadcasting the matrix using frovedis global function “`frovedis::broadcast()`” (explained in `node_local` manual). But from performance point of view this is efficient as it avoids the internal serialization overhead of the vector elements.

For example,

```
std::vector<int> v = {1,3,2,4};
colmajor_matrix_local<int> m;
m.val.swap(v);
m.local_num_row = 2;
m.local_num_col = 2;
auto bm1 = m.broadcast(); // faster
auto bm2 = frovedis::broadcast(m); // slower (serialization overhead)
```

master	worker0	worker1
-----	-----	-----
m: colmajor_matrix_local<int>		
1 3		
2 4		
 bm1: node_local<		
colmajor_matrix_local<int>>	colmajor_matrix_local<int>	colmajor_matrix_local<int>
	1 3	1 3
	2 4	2 4
 bm2: node_local<		
colmajor_matrix_local<int>>	colmajor_matrix_local<int>	colmajor_matrix_local<int>
	1 3	1 3
	2 4	2 4

## 7.3.4 Public Data Member Documentation

### 7.3.4.1 val

An instance of `std::vector<T>` type to contain the elements of the matrix in 1D column-major form.



**7.3.4.2 local\_num\_row**

A `size_t` attribute to contain the number of rows in the 2D matrix view.

**7.3.4.3 local\_num\_col**

A `size_t` attribute to contain the number of columns in the 2D matrix view.

**7.4 SEE ALSO**

`rowmajor_matrix_local`, `colmajor_matrix`



## Chapter 8

# frovedis::colmajor\_matrix<T>

### 8.1 NAME

`frovedis::colmajor_matrix<T>` - A distributed two-dimensional dense matrix with elements stored in column-wise order supported by `frovedis`

### 8.2 SYNOPSIS

```
#include <frovedis/matrix/colmajor_matrix.hpp>
```

#### 8.2.1 Constructors

```
colmajor_matrix ();  
colmajor_matrix (frovedis::node_local<colmajor_matrix_local<T>>&& data);  
colmajor_matrix (const rowmajor_matrix<T>& m);
```

#### 8.2.2 Public Member Functions

```
void debug_print ();  
rowmajor_matrix<T> to_rowmajor();  
rowmajor_matrix<T> moveto_rowmajor();
```

#### 8.2.3 Public Data Members

```
frovedis::node_local<colmajor_matrix_local<T>> data  
size_t num_row  
size_t num_col
```

### 8.3 DESCRIPTION

`colmajor_matrix<T>` is a template based two-dimensional dense matrix with elements stored in column-major order and distributed among the participating worker nodes in row-wise.

A `colmajor_matrix<T>` contains public member “data” of the type `node_local<colmajor_matrix_local<T>>`. The actual distributed matrices are contained in all the worker nodes locally, thus named as `colmajor_matrix_local<T>` (see manual of `colmajor_matrix_local`) and “data” is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original matrix.

```
template <class T>
struct colmajor_matrix {
    frovedis::node_local<colmajor_matrix_local<T>> data; // local matrix information
    size_t num_row; // number of rows in global matrix
    size_t num_col; // number of columns in global matrix
};
```

For example, if the below column-major matrix with 4 rows and 4 columns is distributed over two worker nodes, then the distribution can be shown as:

```
1 5 8 4
2 6 7 3
3 7 6 2
4 8 5 1
```

master	worker0	worker1
-----	-----	-----
<code>colmajor_matrix&lt;int&gt;</code>	<code>-&gt; colmajor_matrix</code>	<code>-&gt; colmajor_matrix</code>
	<code>    _local&lt;int&gt;</code>	<code>    _local&lt;int&gt;</code>
<code>  *data: node_local&lt;</code>	<code>  val: vector&lt;int&gt;</code>	<code>  val: vector&lt;int&gt;</code>
<code>    colmajor_matrix</code>	<code>    ({1,5,8,4,</code>	<code>    ({3,7,6,2,</code>
<code>    _local&lt;int&gt;&gt;&gt;</code>	<code>      2,6,7,3})</code>	<code>      4,8,5,1})</code>
<code>  num_row: size_t (4)</code>	<code>  local_num_row: size_t (2)</code>	<code>  local_num_row: size_t (2)</code>
<code>  num_col: size_t (4)</code>	<code>  local_num_col: size_t (2)</code>	<code>  local_num_col: size_t (2)</code>

The `node_local<colmajor_matrix_local<int>>` object “data” is simply a (\*)handle of the (->)local matrices at worker nodes.

A distributed `colmajor_matrix` can be created from a distributed `rowmajor_matrix` object and it can be converted back to the `rowmajor_matrix` object. Thus loading from file, saving into file etc. interfaces are not provided for `colmajor_matrix` structure. User may like to perform the conversion from/to `rowmajor_matrix` structure for the same.

### 8.3.1 Constructor Documentation

#### 8.3.1.1 `colmajor_matrix ()`

This is the default constructor which creates an empty distributed `colmajor_matrix` without any memory allocation at worker nodes.

#### 8.3.1.2 `colmajor_matrix(const rowmajor_matrix<T>& m)`

It accepts a distributed `rowmajor_matrix<T>` object with elements stored in row-major order and constructs an equivalent distributed `colmajor` storage of same number of rows and columns. Input row-major storage remains unchanged.

**8.3.1.3 colmajor\_matrix (frovedis::node\_local<colmajor\_matrix\_local<T>>&& data)**

This is the parameterized constructor which accepts an rvalue of the type `node_local<colmajor_matrix_local<T>>` and *moves* the contents to the created colmajor matrix.

In general, this constructor is used internally by some other functions. But user may need this constructor while constructing their own colmajor matrix using the return value of some function (returning a `colmajor_matrix_local<T>`) called using “`frovedis::node_local::map`” (thus returned value would be an object of type `node_local<colmajor_matrix_local<T>`).

For example,

```
// --- a sample functor definition ---
struct foo {
    foo() {}
    foo(int r, int c): nrow(r), ncol(c) {}
    colmajor_matrix_local<int> operator()(std::vector<int>& v) {
        colmajor_matrix_local<int> ret;
        ret.val.swap(v);
        ret.local_num_row = nrow;
        ret.local_num_col = ncol;
        return ret;
    }
    size_t nrow, ncol;
    SERIALIZE(nrow, ncol)
};

size_t sum(size_t x, size_t y) { return x + y; }
size_t get_nrows(colmajor_matrix_local<int>& m) { return m.local_num_row; }
size_t get_ncols(colmajor_matrix_local<int>& m) { return m.local_num_col; }

std::vector<int> v = {1,3,5,7,2,4,6,8}; // 4x2 col-major storage
auto bv = broadcast(v);
// demo of such a constructor call
colmajor_matrix<int> m(bv.map<colmajor_matrix_local<int>>(foo(4,2))); // m: 8x2 matrix
// getting total number of rows in the global matrix
m.num_row = m.data.map(get_nrows).reduce(sum); // 4+4 = 8
m.num_col = m.data.map(get_ncols).get(0);      // 2
```

The above program will perform the below tasks in order

- broadcast a vector containing sample elements of a 4x2 `colmajor_matrix_local`.
- local colmajor matrices will be created in worker nodes when the functor would be called.
- “`bv.map<colmajor_matrix_local<int>>(foo(4,2))`” will return a `node_local<colmajor_matrix_local<int>` object.
- the constructor call will be made for `colmajor_matrix` passing the above rvalue `node_local` object.
- total number of rows will be set by summing `local_num_row` of all worker matrices.
- total number of columns will be set as per the number of columns in the worker0 matrix (from any worker will be fine).

### 8.3.2 Public Member Function Documentation

#### 8.3.2.1 void debug\_print ()

It prints the contents and other information of the local matrices node-by-node on the user terminal. It is mainly useful for debugging purpose.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2,3,4,5,6,7,8}; // 4x2 col-major storage
rowmajor_matrix_local<int> m;
m.val.swap(v);
m.set_local_num(nrow,ncol);
// scattering local matrix to create the distributed rowmajor matrix
auto rm = make_rowmajor_matrix_scatter(m);
colmajor_matrix<int> cm(rm); // rowmajor_matrix => colmajor_matrix
cm.debug_print();
```

The above program will output (order of display might differ):

```
node = 0, local_num_row = 2, local_num_col = 2, val = 1 3 2 4
node = 1, local_num_row = 2, local_num_col = 2, val = 5 7 6 8
```

#### 8.3.2.2 rowmajor\_matrix<T> to\_rowmajor();

It converts the colmajor storage of the target distributed matrix to a distributed rowmajor storage and returns the output `rowmajor_matrix<T>` after successful conversion. The target colmajor storage remains unchanged after the conversion.

#### 8.3.2.3 rowmajor\_matrix<T> moveto\_rowmajor();

If the target distributed column major matrix has only a single column, then rowmajor storage and column major storage both will be the same. Thus instead of any conversion overhead, elements in target matrix can simply be moved while creating the `rowmajor_matrix` object. It is faster and recommended, only when the target matrix is no longer be needed in a user program.

### 8.3.3 Public Data Member Documentation

#### 8.3.3.1 data

An instance of `node_local<colmajor_matrix_local<T>>` type to contain the reference information related to local matrices at worker nodes.

#### 8.3.3.2 num\_row

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

#### 8.3.3.3 num\_col

A `size_t` attribute to contain the total number of columns in the 2D matrix view.

## 8.4 **SEE ALSO**

`colmajor_matrix_local`, `rowmajor_matrix`, `blockcyclic_matrix`





## Chapter 9

# frovedis::sliced\_colmajor\_matrix\_local<T>

### 9.1 NAME

`frovedis::sliced_colmajor_matrix_local<T>` - a data structure containing the slicing information of a two-dimensional `frovedis::colmajor_matrix_local<T>`

### 9.2 SYNOPSIS

```
#include <frovedis/matrix/sliced_matrix.hpp>
```

#### 9.2.1 Constructors

```
sliced_colmajor_matrix_local ()  
sliced_colmajor_matrix_local (const colmajor_matrix_local<T>& m)  
sliced_colmajor_matrix_local (const std::vector<T>& v)
```

#### 9.2.2 Public Member Functions

```
bool is_valid () const  
void debug_print () const
```

#### 9.2.3 Public Data Members

```
T* data  
size_t ldm  
size_t sliced_num_row  
size_t sliced_num_col
```

### 9.3 DESCRIPTION

In order to perform matrix operations on sub-matrices instead of entire physical matrix, `frovedis` provides some sliced data structures. `sliced_colmajor_matrix_local<T>` is one of them. It is actually not a real

matrix, rather it only contains some slicing information of a physical `colmajor_matrix_local<T>`. Thus any changes performed on the sliced matrix, would actually make changes on the physical matrix from which slice was made.

Like `colmajor_matrix_local<T>`, a `sliced_colmajor_matrix_local<T>` is also a template based structure with type “**T**”. This has the below structure:

```
template <class T>
struct sliced_colmajor_matrix_local {
    T* data;           // pointer pointing to the begining of the slice
    size_t ldm;        // leading dimension of the physical matrix
    size_t sliced_num_row; // number of rows in the sliced matrix
    size_t sliced_num_col; // number of columns in the sliced matrix
};
```

E.g., if a physical `colmajor_matrix_local<T>` M has dimensions 4x4 and slice is needed from 2nd row and 2nd column ([1,1]) till 3rd row and 3rd column ([2,2]), then “data” will hold the address of M[1][1] (data -> &M[1][1]),

“ldm” would be 4 (leading dimension of the matrix M, i.e., number of rows),

From 2nd row till 3rd row, number of rows to be sliced is 2, thus “sliced\_num\_row” would be 2.

From 2nd column till 3rd column, number of columns to be sliced is 2, thus “sliced\_num\_col” would be 2.

Such matrices are very useful in operations of external libraries like blas, lapack etc.

### 9.3.1 Constructor Documentation

#### 9.3.1.1 `sliced_colmajor_matrix_local ()`

This is the default constructor which creates an empty sliced matrix with `num_row = num_col = 0` and “data” points to NULL. In general of no use, unless it is needed to manipulate the slice information explicitly.

#### 9.3.1.2 `sliced_colmajor_matrix_local (const colmajor_matrix_local<T>& m)`

This is a special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced matrix. Thus the created `sliced_colmajor_matrix_local<T>` would have the same dimensions as with the input `colmajor_matrix_local<T>` and “data” pointing to the base address of the input `colmajor_matrix_local<T>`.

#### 9.3.1.3 `sliced_colmajor_matrix_local (const std::vector<T>& v)`

This is a special constructor for implicit conversion. This constructor treats an entire physical vector as a sliced matrix. Thus the created `sliced_colmajor_matrix_local<T>` would have “sliced\_num\_row” equals to the length of the input `std::vector<T>`, “sliced\_num\_col” equals to 1 and “data” pointing to the base address of the input vector.

### 9.3.2 Public Member Function Documentation

#### 9.3.2.1 `bool is_valid () const`

This function returns true, if the caller object is a valid sliced matrix, else it returns false.

Kindly note that an empty sliced matrix is also an invalid sliced matrix, since no valid operation can be performed on its data pointing to NULL.

**9.3.2.2 void debug\_print () const**

It prints the contents of the sliced part of the original (physical) `colmajor_matrix_local<T>` on the user standard output terminal.

**9.3.3 Public Data Member Documentation****9.3.3.1 data**

A pointer of type “T” pointing to the starting location of a physical `colmajor_matrix_local<T>` from which slice has been made.

**9.3.3.2 ldm**

A `size_t` attribute to contain the leading dimension of the physical matrix from which slice has been made (number of rows in the physical matrix).

**9.3.3.3 sliced\_num\_row**

A `size_t` attribute to contain the number of rows in the sliced matrix.

**9.3.3.4 sliced\_num\_col**

A `size_t` attribute to contain the number of columns in the sliced matrix.

**9.3.4 Public Global Function Documentation****9.3.4.1 make\_sliced\_colmajor\_matrix\_local (mat, start\_ridx, start\_cidx, num\_row, num\_col)****Parameters**

*mat*: An object of either `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` type.

*start\_ridx*: A `size_t` attribute to indicate the start row index for the slice.

*start\_cidx*: A `size_t` attribute to indicate the start column index for the slice.

*num\_row*: A `size_t` attribute to indicate the number of rows to be sliced from the starting row index.

*num\_col*: A `size_t` attribute to indicate the number of columns to be sliced from the starting column index.

**Purpose**

This function accepts a valid `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` with some slicing information like row and column index from which slicing is to be started, and the size of the output sliced matrix, i.e., number of rows and columns to be sliced from the starting location. On receiving the valid inputs, it outputs a `sliced_colmajor_matrix_local<T>` object containing the slicing information, else it throws an exception.

**Example:**

If a physical `colmajor_matrix_local<T>` “mat” has the dimensions 4x4 and slicing is required from its 2nd row and 2nd column ([1,1]) till 4th row and 4th column ([3,3]), then this function should be called like:

```
auto smat = make_sliced_colmajor_matrix_local(mat,1,1,3,3);
```

Index of the 2nd row is 1, thus `start_row_index = 1`.

Index of the 2nd column is 1, thus `start_col_index = 1`.

From 2nd row till 4th row, number of rows to be sliced is 3, thus `num_row = 3`.

From 2nd column till 4th column, number of columns to be sliced is 3, thus `num_col = 3`.

Input (mat):		Output (smat):
-----		-----
1 2 3 4		6 7 8
5 6 7 8	=>	7 6 5
8 7 6 5		3 2 1
4 3 2 1		

Now if we need to slice further this sliced matrix, “smat” from its 2nd row and 2nd column ([1,1]) till 3rd row and 3rd column ([2,2]), then we would call this function like below:

```
auto ssmat = make_sliced_colmajor_matrix_local(smat,1,1,2,2);
```

Index of the 2nd row of smat is 1, thus `start_row_index = 1`.

Index of the 2nd column of smat is 1, thus `start_col_index = 1`.

From 2nd row till 3rd row of smat, number of rows to be sliced is 2, thus `num_row = 2`.

From 2nd column till 3rd column of smat, number of columns to be sliced is 2, thus `num_col = 2`.

Kindly note that 2nd row of “smat” is actually the 3rd row of the physical matrix “mat”, but this function takes care of it internally. Thus you just need to take care of the index of the input sliced matrix, not the actual physical matrix.

Input (smat):		Output (ssmat):
-----		-----
6 7 8		6 5
7 6 5	=>	2 1
3 2 1		

### Return Value

On success, it returns an object of the type `sliced_colmajor_matrix_local<T>`. Otherwise it throws an exception.

## 9.4 SEE ALSO

`colmajor_matrix`, `sliced_colmajor_vector_local`

## Chapter 10

# frovedis::sliced\_colmajor\_vector\_local<T>

### 10.1 NAME

`frovedis::sliced_colmajor_vector_local<T>` - a data structure containing the row or column wise slicing information of a two-dimensional `frovedis::colmajor_matrix_local<T>`

### 10.2 SYNOPSIS

```
#include <frovedis/matrix/sliced_vector.hpp>
```

#### 10.2.1 Constructors

```
sliced_colmajor_vector_local ()  
sliced_colmajor_vector_local (const colmajor_matrix_local<T>& m)  
sliced_colmajor_vector_local (const std::vector<T>& v)
```

#### 10.2.2 Public Member Functions

```
bool is_valid () const  
void debug_print () const
```

#### 10.2.3 Public Data Members

```
T* data  
size_t size  
size_t stride
```

### 10.3 DESCRIPTION

In order to perform vector operations on some rows or on some columns of a dense matrix, `frovedis` provides some sliced data structures. `sliced_colmajor_vector_local<T>` is one of them. It is actually not a real vector, rather it only contains some slicing information of a physical `colmajor_matrix_local<T>`. Thus any

changes performed on the sliced vector, would actually make changes on the specific row or column of the physical matrix from which the slice was made.

Like `colmajor_matrix_local<T>`, a `sliced_colmajor_vector_local<T>` is also a template based structure with type “T”. This has the below structure:

```
template <class T>
struct sliced_colmajor_vector_local {
    T* data;          // pointer pointing to the begining of the row or column to be sliced
    size_t size;      // number of elements in the sliced vector
    size_t stride;    // stride between two consecutive elements in the sliced vector
};
```

E.g., if a physical `colmajor_matrix_local<T>` M has dimensions 4x4 and its 2nd row needs to be sliced, then

“data” will hold the address of `M[0][1]` (not `M[1][0]`, since matrix is stored in colmajor order),

“size” would be 4 (number of elements in 2nd row),

and “stride” would be 4 (since matrix is stored in colmajor order, the stride between two consecutive elements in a row would be equal to leading dimension of that matrix, i.e., number of rows in that matrix).

On the other hand, if 2nd column needs to be sliced, then

“data” will hold the address of `M[1][0]` (not `M[0][1]`, since matrix is stored in colmajor order),

“size” would be 4 (number of elements in 2nd column),

and “stride” would be 1 (since matrix is stored in colmajor order, the consecutive elements in a column would be placed one after another).

Such vectors are very useful in operations of external libraries like blas etc.

### 10.3.1 Constructor Documentation

#### 10.3.1.1 `sliced_colmajor_vector_local ()`

This is the default constructor which creates an empty sliced vector with `size = stride = 0` and “data” points to NULL. In general of no use, unless it is needed to manipulate the slice information explicitly.

#### 10.3.1.2 `sliced_colmajor_vector_local (const colmajor_matrix_local<T>& m)`

This is a special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced vector. Thus the created `sliced_colmajor_vector_local<T>` would have “size” equals to number of rows in the input `colmajor_matrix_local<T>`, “stride” equals to 1 and “data” pointing to the base address of the input `colmajor_matrix_local<T>`. Please note that such conversion can only be possible if the input matrix can be treated as a column vector (a matrix with multiple rows and single column), else it throws an exception.

#### 10.3.1.3 `sliced_colmajor_vector_local (const std::vector<T>& v)`

This is a special constructor for implicit conversion. This constructor treats an entire physical vector as a sliced vector. Thus the created `sliced_colmajor_vector_local<T>` would have “size” equals to the length of the input `std::vector<T>`, “stride” equals to 1 and “data” pointing to the base address of the input vector.

## 10.3.2 Public Member Function Documentation

### 10.3.2.1 bool is\_valid () const

This function returns true, if the caller object is a valid sliced vector, else it returns false.

Kindly note that an empty sliced vector is also an invalid sliced vector, since no valid operation can be performed on its data pointing to NULL.

### 10.3.2.2 void debug\_print () const

It prints the contents of the sliced row or column of the original (physical) `colmajor_matrix_local<T>` on the user standard output terminal.

## 10.3.3 Public Data Member Documentation

### 10.3.3.1 data

A pointer of type “T” pointing to the beginning of the row or column of a physical `colmajor_matrix_local<T>` from which slice has been made.

### 10.3.3.2 size

A `size_t` attribute to contain the number of elements in the sliced vector.

### 10.3.3.3 stride

A `size_t` attribute to contain the stride between two consecutive elements in a sliced vector.

## 10.3.4 Public Global Function Documentation

### 10.3.4.1 make\_row\_vector (mat, row\_index)

#### Parameters

*mat*: An object of either `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` type.

*row\_index*: A `size_t` attribute to indicate the row index to be sliced.

#### Purpose

This function accepts a valid `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` with the row index to be sliced (index starts with 0). On receiving valid inputs, it outputs a `sliced_colmajor_vector_local<T>` object containing the slicing information, else it throws an exception.

#### Example:

If a physical `colmajor_matrix_local<T>` “mat” has the dimensions 4x4 and its 2nd row needs to be sliced, then this function should be called like:

```
auto rvec = make_row_vector(mat,1); // row index of second row is 1
```

Input (mat):		Output (rvec):
-----		-----
1 2 3 4	=>	5 6 7 8
5 6 7 8		

```
8 7 6 5
4 3 2 1
```

Now if it is needed to slice the 2nd row from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat = make_sliced_colmajor_matrix_local(mat,2,2,2,2);
auto s_rvec = make_row_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd row is to be sliced from the sub-matrix.

Kindly note that 2nd row of “smat” is actually the 4th row of the physical matrix “mat”, but this function takes care of it internally. Thus user only needs to take care of the index of the input sliced matrix, not the actual physical matrix.

Input (mat):	Output (smat):	Output (s_rvec):
-----	-----	-----
1 2 3 4	6 5	=> 2 1
5 6 7 8	=> 2 1	
8 7 6 5		
4 3 2 1		

### Return Value

On success, this function returns the sliced row vector of the type `sliced_colmajor_vector_local<T>`. Otherwise it throws an exception.

#### 10.3.4.2 make\_col\_vector (mat, col\_index)

##### Parameters

*mat*: An object of either `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` type.

*col\_index*: A `size_t` attribute to indicate the column index needs to be sliced.

##### Purpose

This function accepts a valid `colmajor_matrix_local<T>` or `sliced_colmajor_matrix_local<T>` with the column index to be sliced (index starts with 0). On receiving the valid inputs, it outputs a `sliced_colmajor_vector_local<T>` object containing the slicing information, else it throws an exception.

##### Example:

If a physical `colmajor_matrix_local<T>` “mat” has the dimensions 4x4 and its 2nd column needs to be sliced, then this function should be called like:

```
auto cvec = make_col_vector(mat,1); // column index of second column is 1
```

Input (mat):	Output (cvec):
-----	-----
1 2 3 4	=> 2 6 7 3
5 6 7 8	
8 7 6 5	
4 3 2 1	

Now if it is needed to slice the 2nd column from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat = make_sliced_colmajor_matrix_local(mat,2,2,2,2);
auto s_cvec = make_col_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd column is to be sliced from the sub-matrix.



Kindly note that 2nd column of “smat” is actually the 4th column of the physical matrix “mat”, but this function takes care of it internally. Thus user only needs to take care of the index of the input sliced matrix, not the actual physical matrix.

Input (mat):		Output (smat):		Output (s_cvec):
-----		-----		-----
1 2 3 4		6 5	=>	5 1
5 6 7 8	=>	2 1		
8 7 6 5				
4 3 2 1				

#### Return Value

On success, it returns the sliced column vector of the type `sliced_colmajor_vector_local<T>`. Otherwise it throws an exception.

## 10.4 SEE ALSO

`colmajor_matrix`, `sliced_colmajor_matrix_local`



# Chapter 11

## blas\_wrapper

### 11.1 NAME

blas\_wrapper - a frovedis module provides user-friendly interfaces for commonly used blas routines in scientific applications like machine learning algorithms.

### 11.2 SYNOPSIS

```
#include <frovedis/matrix/blas_wrapper.hpp>
```

### 11.3 WRAPPER FUNCTION

```
void swap (const sliced_colmajor_vector_local<T>& v1,
           const sliced_colmajor_vector_local<T>& v2)
void copy (const sliced_colmajor_vector_local<T>& v1,
           const sliced_colmajor_vector_local<T>& v2)
void scal (const sliced_colmajor_vector_local<T>& v,
           T al)
void axpy (const sliced_colmajor_vector_local<T>& v1,
           const sliced_colmajor_vector_local<T>& v2,
           T al = 1.0)
T dot (const sliced_colmajor_vector_local<T>& v1,
       const sliced_colmajor_vector_local<T>& v2)
T nrm2 (const sliced_colmajor_vector_local<T>& v)
void gemv (const sliced_colmajor_matrix_local<T>& m,
           const sliced_colmajor_vector_local<T>& v1,
           const sliced_colmajor_vector_local<T>& v2,
           char trans = 'N',
           T al = 1.0,
           T be = 0.0)
void ger (const sliced_colmajor_vector_local<T>& v1,
          const sliced_colmajor_vector_local<T>& v2,
          const sliced_colmajor_matrix_local<T>& m,
          T al = 1.0)
void gemm (const sliced_colmajor_matrix_local<T>& m1,
```

```

const sliced_colmajor_matrix_local<T>& m2,
const sliced_colmajor_matrix_local<T>& m3,
char trans_m1 = 'N',
char trans_m2 = 'N',
T al = 1.0,
T be = 0.0)

```

## 11.4 OVERLOADED OPERATORS

```

colmajor_matrix_local<T>
operator* (const sliced_colmajor_matrix_local<T>& m1,
          const sliced_colmajor_matrix_local<T>& m2)
colmajor_matrix_local<T>
operator~ (const sliced_colmajor_matrix_local<T>& m1)

```

## 11.5 DESCRIPTION

BLAS is a high-performance external library written in Fortran language. It provides rich set of functionalities on vectors and matrices. Like PBLAS, computation loads of these functionalities **are not parallelized** over the available processes in a system, thus they operate on *non-distributed* data. But like PBLAS, the user interfaces of this library are also very detailed and a bit complex in nature. It requires a strong understanding on each of the input parameters before using these functionalities correctly.

Frovedis provides a wrapper module for some commonly used BLAS subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed input parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a BLAS routine can also be performed using Frovedis BLAS wrapper of that routine.

These wrapper routines are global functions in nature. Thus they can be called easily from within the “frovedis” namespace. As an input vector, they accept “colmajor\_matrix\_local<T> with single column” or “sliced\_colmajor\_vector\_local<T>”. And as an input matrix, they accept “colmajor\_matrix\_local<T>” or “sliced\_colmajor\_matrix\_local<T>”. “T” is a template type which can be either “float” or “double”. The individual detailed descriptions can be found in the subsequent sections. Please note that the term “inout”, used in the below section indicates a function argument as both “input” and “output”.

### 11.5.1 Detailed Description

#### 11.5.1.1 swap (v1, v2)

##### Parameters

*v1*: A colmajor\_matrix\_local<T> with single column or a sliced\_colmajor\_vector\_local<T> (inout)  
*v2*: A colmajor\_matrix\_local<T> with single column or a sliced\_colmajor\_vector\_local<T> (inout)

##### Purpose

It will swap the contents of v1 and v2, if they are semantically valid and are of same length.

##### Return Value

On success, it returns void. If any error occurs, it throws an exception.

**11.5.1.2 copy (v1, v2)****Parameters**

*v1*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*v2*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (output)

**Purpose**

It will copy the contents of *v1* in *v2* ( $v2 = v1$ ), if they are semantically valid and are of same length.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**11.5.1.3 scal (v, al)****Parameters**

*v*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (inout)

*al*: A “T” typed argument (float or double) to specify the value to which the input vector needs to be scaled. (input)

**Purpose**

It will scale the input vector with the provided “*al*” value, if it is semantically valid. On success, input vector “*v*” would be updated (in-place scaling).

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**11.5.1.4 axpy (v1, v2, al=1.0)****Parameters**

*v1*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*v2*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (inout)

*al*: A “T” typed argument (float or double) to specify the value to which “*v1*” needs to be scaled (not in-place scaling) [Default: 1.0] (input/optional)

**Purpose**

It will solve the expression  $v2 = al * v1 + v2$ , if the input vectors are semantically valid and are of same length. On success, “*v2*” will be updated with desired result, but “*v1*” would remain unchanged.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**11.5.1.5 dot (v1, v2)****Parameters**

*v1*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*v2*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

**Purpose**

It will perform dot product of the input vectors, if they are semantically valid and are of same length. Input vectors would not get modified during the operation.

**Return Value**

On success, it returns the dot product result of the type “float” or “double”. If any error occurs, it throws an exception.

### 11.5.1.6 nrm2 (v)

#### Parameters

*v*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

#### Purpose

It will calculate the norm of the input vector, if it is semantically valid. Input vector would not get modified during the operation.

#### Return Value

On success, it returns the norm value of the type “float” or “double”. If any error occurs, it throws an exception.

### 11.5.1.7 gemv (m, v1, v2, trans='N', al=1.0, be=0.0)

#### Parameters

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*v1*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*v2*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (inout)

*trans*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)

*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

*be*: A “T” typed (float or double) scalar value [Default: 0.0] (input/optional)

#### Purpose

The primary aim of this routine is to perform simple matrix-vector multiplication.

But it can also be used to perform any of the below operations:

(1)  $v2 = al * m * v1 + be * v2$

(2)  $v2 = al * \text{transpose}(m) * v1 + be * v2$

If *trans*='N', then expression (1) is solved. In that case, the size of “v1” must be at least the number of columns in “m” and the size of “v2” must be at least the number of rows in “m”.

If *trans*='T', then expression (2) is solved. In that case, the size of “v1” must be at least the number of rows in “m” and the size of “v2” must be at least the number of columns in “m”.

Since “v2” is used as input-output both, memory must be allocated for this vector before calling this routine, even if simple matrix-vector multiplication is required. Otherwise, this routine will throw an exception.

For simple matrix-vector multiplication, no need to specify values for the input parameters “trans”, “al” and “be” (leave them at their default values).

On success, “v2” will be overwritten with the desired output. But “m” and “v1” would remain unchanged.

#### Return Value

On success, it returns void. If any error occurs, it throws an exception.

### 11.5.1.8 ger (v1, v2, m, al=1.0)

#### Parameters

*v1*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*v2*: A `colmajor_matrix_local<T>` with single column or a `sliced_colmajor_vector_local<T>` (input)

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

#### Purpose

The primary aim of this routine is to perform simple vector-vector multiplication of the sizes “a” and “b” respectively to form an axb matrix. But it can also be used to perform the below operations:

```
m = al*v1*v2' + m
```

This operation can only be performed if the inputs are semantically valid and the size of “v1” is at least the number of rows in matrix “m” and the size of “v2” is at least the number of columns in matrix “m”.

Since “m” is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple vector-vector multiplication is required. Otherwise it will throw an exception.

For simple vector-vector multiplication, no need to specify the value for the input parameter “al” (leave it at its default value).

On success, “m” will be overwritten with the desired output. But “v1” and “v2” will remain unchanged.

### Return Value

On success, it returns void. If any error occurs, it throws an exception.

#### 11.5.1.9 gemm (m1, m2, m3, trans\_m1='N', trans\_m2='N', al=1.0, be=0.0)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*m3*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*trans\_m1*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)

*trans\_m2*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)

*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

*be*: A “T” typed (float or double) scalar value [Default: 0.0] (input/optional)

**Purpose** The primary aim of this routine is to perform simple matrix-matrix multiplication.

But it can also be used to perform any of the below operations:

- (1)  $m3 = al*m1*m2 + be*m3$
- (2)  $m3 = al*transpose(m1)*m2 + be*m3$
- (3)  $m3 = al*m1*transpose(m2) + be*m3$
- (4)  $m3 = al*transpose(m1)*transpose(m2) + be*m3$

(1) will be performed, if both “trans\_m1” and “trans\_m2” are ‘N’.

(2) will be performed, if trans\_m1=‘T’ and trans\_m2 = ‘N’.

(3) will be performed, if trans\_m1=‘N’ and trans\_m2 = ‘T’.

(4) will be performed, if both “trans\_m1” and “trans\_m2” are ‘T’.

If we have four variables nrowa, nrowb, ncola, ncolb defined as follows:

```
if(trans_m1 == 'N') {
    nrowa = number of rows in m1
    ncola = number of columns in m1
}
else if(trans_m1 == 'T') {
    nrowa = number of columns in m1
    ncola = number of rows in m1
}

if(trans_m2 == 'N') {
    nrowb = number of rows in m2
    ncolb = number of columns in m2
}
```

```

else if(trans_m2 == 'T') {
    nrowb = number of columns in m2
    ncolb = number of rows in m2
}

```

Then this function can be executed successfully, if the below conditions are all true:

- (a) "ncola" is equal to "nrowb"
- (b) number of rows in "m3" is equal to or greater than "nrowa"
- (b) number of columns in "m3" is equal to or greater than "ncolb"

Since “m3” is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple matrix-matrix multiplication is required. Otherwise it will throw an exception.

For simple matrix-matrix multiplication, no need to specify the value for the input parameters “trans\_m1”, “trans\_m2”, “al”, “be” (leave them at their default values).

On success, “m3” will be overwritten with the desired output. But “m1” and “m2” will remain unchanged.

**Return Value** On success, it returns void. If any error occurs, it throws an exception.

#####operator\* (m1, m2) **Parameters**

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

#### **Purpose**

This operator operates on two input matrices and returns the resultant matrix after successful multiplication. Both the input matrices remain unchanged.

#### **Return Value**

On success, it returns the resultant matrix of the type `colmajor_matrix_local<T>`. If any error occurs, it throws an exception.

#####operator~ (m1) **Parameters**

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

#### **Purpose**

This operator operates on single input matrix and returns its transposed matrix. E.g., if “m” is a matrix of type `colmajor_matrix_local<T>`, then “~m” will return transposed of matrix “m” of the type `colmajor_matrix_local<T>`.

#### **Return Value**

On success, it returns the resultant matrix of the type `colmajor_matrix_local<T>`. If any error occurs, it throws an exception.

## 11.6 SEE ALSO

`sliced_colmajor_matrix_local`, `sliced_colmajor_vector_local`, `pblas_wrapper`



# Chapter 12

## lapack\_\_wrapper

### 12.1 NAME

lapack\_\_wrapper - a frovedis module provides user-friendly interfaces for commonly used lapack routines in scientific applications like machine learning algorithms.

### 12.2 SYNOPSIS

```
#include <frovedis/matrix/lapack_wrapper.hpp>
```

### 12.3 WRAPPER FUNCTIONS

```
int getrf (const sliced_colmajor_matrix_local<T>& m,
           std::vector<int>& ipiv)
int getri (const sliced_colmajor_matrix_local<T>& m,
           const std::vector<int>& ipiv)
int getsr (const sliced_colmajor_matrix_local<T>& m1,
           const sliced_colmajor_matrix_local<T>& m2,
           const std::vector<int>& ipiv,
           char trans = 'N')
int gesv (const sliced_colmajor_matrix_local<T>& m1,
           const sliced_colmajor_matrix_local<T>& m2)
int gesv (const sliced_colmajor_matrix_local<T>& m1,
           const sliced_colmajor_matrix_local<T>& m2,
           std::vector<int>& ipiv)
int gels (const sliced_colmajor_matrix_local<T>& m1,
           const sliced_colmajor_matrix_local<T>& m2,
           char trans = 'N')
int gesvd (const sliced_colmajor_matrix_local<T>& m,
           std::vector<T>& sval,
           char option = 'N')
int gesvd (const sliced_colmajor_matrix_local<T>& m,
           std::vector<T>& sval,
           const sliced_colmajor_matrix_local<T>& svec,
           char vtype = 'L',
```

```

    char part = 'A',
    char opt_a = 'N')
int gesvd (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& sval,
    const sliced_colmajor_matrix_local<T>& lsvect,
    const sliced_colmajor_matrix_local<T>& rsvect,
    char part_l = 'A',
    char part_r = 'A')
int gesdd (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& sval)
int gesdd (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& sval,
    const sliced_colmajor_matrix_local<T>& svec)
int gesdd (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& sval,
    const sliced_colmajor_matrix_local<T>& lsvect,
    const sliced_colmajor_matrix_local<T>& rsvect,
    char part_lr = 'A')
int gelsy (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    T rcond = -1)
int gelsy (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    int& rank,
    T rcond = -1)
int gelss (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    T rcond = -1)
int gelss (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    std::vector<T>& sval,
    int& rank,
    T rcond = -1)
int gelsd (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    T rcond = -1)
int gelsd (const sliced_colmajor_matrix_local<T>& m1,
    const sliced_colmajor_matrix_local<T>& m2,
    std::vector<T>& sval,
    int& rank,
    T rcond = -1)
int geev (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& eval)
int geev (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& eval,
    const sliced_colmajor_matrix_local<T>& evec,
    char vtype = 'L')
int geev (const sliced_colmajor_matrix_local<T>& m,
    std::vector<T>& eval,
    const sliced_colmajor_matrix_local<T>& levec,
    const sliced_colmajor_matrix_local<T>& revec)

```

## 12.4 SPECIAL FUNCTIONS

```
colmajor_matrix_local<T> inv (const sliced_colmajor_matrix_local<T>& m)
```

## 12.5 DESCRIPTION

LAPACK is a high-performance external library written in Fortran language. It provides rich set of linear algebra functionalities. Like ScaLAPACK, computation loads of these functionalities **are not parallelized** over the available processes in a system, thus they operate on *non-distributed* data. But like ScaLAPACK, the user interfaces of this library are also very detailed and a bit complex in nature. It requires a strong understanding on each of the input parameters before using these functionalities correctly.

Frovedis provides a wrapper module for some commonly used LAPACK subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed input parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a LAPACK routine can also be performed using Frovedis LAPACK wrapper of that routine.

These wrapper routines are global functions in nature. Thus they can be called easily from within the “frovedis” namespace. As an input matrix, they accept “colmajor\_matrix\_local<T>” or “sliced\_colmajor\_matrix\_local<T>”. “T” is a template type which can be either “float” or “double”. The individual detailed descriptions can be found in the subsequent sections. Please note that the term “inout”, used in the below section indicates a function argument as both “input” and “output”.

### 12.5.1 Detailed Description

#### 12.5.1.1 getrf (m, ipiv)

##### Parameters

*m*: A colmajor\_matrix\_local<T> or a sliced\_colmajor\_matrix\_local<T> (inout)

*ipiv*: An empty object of the type std::vector<int> (output)

##### Purpose

It computes an LU factorization of a general M-by-N matrix, “m” using partial pivoting with row interchanges.

On successful factorization, matrix “m” is overwritten with the computed L and U factors. Along with the input matrix, this function expects user to pass an empty object of the type “std::vector<int>” as a second argument, named as “ipiv” which would be updated with the pivoting information associated with input matrix “m” by this function while computing factors. This “ipiv” information will be useful in computation of some other functions (like getri, getsr etc.)

##### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.2 getri (m, ipiv)

##### Parameters

*m*: A colmajor\_matrix\_local<T> or a sliced\_colmajor\_matrix\_local<T> (inout)

*ipiv*: An object of the type std::vector<int> (input)

##### Purpose

It computes the inverse of a square matrix using the LU factorization computed by getrf(). So in order to

compute inverse of a matrix, first compute it's LU factor (and ipiv information) using `getrf()` and then pass the factored matrix, "m" along with the "ipiv" information to this function.

On success, factored matrix "m" is overwritten with the inverse (of the matrix which was passed to `getrf()`) matrix. "ipiv" will be internally used by this function and will remain unchanged.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.3 `getrs (m1, m2, ipiv, trans='N')`

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*ipiv*: An object of the type `std::vector<int>` (input)

*trans*: A character containing either 'N' or 'T' [Default: 'N'] (input/optional)

##### Purpose

It solves a real system of linear equations,  $AX=B$  with a general square matrix (A) using the LU factorization computed by `getrf()`. Thus before calling this function, it is required to obtain the factored matrix "m1" (along with "ipiv" information) by calling `getrf()`.

If `trans='N'`, the linear equation  $AX=B$  is solved.

If `trans='T'` the linear equation  $\text{transpose}(A)X=B$  ( $A'X=B$ ) is solved.

The matrix "m2" should have number of rows  $\geq$  the number of rows in "m1" and at least 1 column in it.

On entry, "m2" contains the right-hand-side (B) of the equation and on successful exit it is overwritten with the solution matrix (X).

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.4 `gesv (m1, m2)`

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

##### Purpose

It solves a real system of linear equations,  $AX=B$  with a general square matrix, "m1" by computing it's LU factors internally. This function internally computes the LU factors and ipiv information using `getrf()` and then solves the equation using `getrs()`.

The matrix "m2" should have number of rows  $\geq$  the number of rows in "m1" and at least 1 column in it.

On entry, "m1" contains the left-hand-side square matrix (A), "m2" contains the right-hand-side matrix (B) and on successful exit "m1" is overwritten with it's LU factors, "m2" is overwritten with the solution matrix (X).

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.5 gesv (m1, m2, ipiv)****Parameters**

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*ipiv*: An empty object of the type `std::vector<int>` (output)

**Purpose**

The function serves the same purpose as explained in above version of `gesv` (with two parameters). Only difference is that this version accepts an extra parameter “*ipiv*” of the type `std::vector<int>` which would be allocated and updated with the pivoting information computed during factorization of “*m1*”. Along with the factored matrix, it might also be needed to know the associated pivot values. In that case, this version of `gesv` (with three parameters) can be used.

On entry, “*m1*” contains the left-hand-side square matrix (A), “*m2*” contains the right-hand-side matrix (B), and “*ipiv*” is an empty object. On successful exit “*m1*” is overwritten with its LU factors, “*m2*” is overwritten with the solution matrix (X), and “*ipiv*” is updated with the pivot values associated with factored matrix, “*m1*”.

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.6 gels (m1, m2, trans=‘N’)****Parameters**

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*trans*: A character containing either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)

**Purpose**

It solves overdetermined or underdetermined real linear systems involving an M-by-N matrix (A) or its transpose, using a QR or LQ factorization of (A). It is assumed that matrix (A) has full rank.

If *trans*=‘N’ and  $M \geq N$ : it finds the least squares solution of an overdetermined system.

If *trans*=‘N’ and  $M < N$ : it finds the minimum norm solution of an underdetermined system.

If *trans*=‘T’ and  $M \geq N$ : it finds the minimum norm solution of an underdetermined system.

If *trans*=‘T’ and  $M < N$ : it finds the least squares solution of an overdetermined system.

The matrix “*m2*” should have number of rows  $\geq \max(M, N)$  and at least 1 column.

On entry, “*m1*” contains the left-hand-side matrix (A) and “*m2*” contains the right-hand-side matrix (B). On successful exit, “*m1*” is overwritten with the QR or LQ factors and “*m2*” is overwritten with the solution matrix (X).

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.7 gesvd (m, sval, option=‘N’)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

*option*: A character containing either ‘L’, ‘R’ or ‘N’ [Default: ‘N’] (input/optional)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. Optionally, it can also compute part of left or right singular vectors.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`. And on exit, if option=‘L’, then “m” is overwritten with the first min(M,N) columns of left singular vectors (stored columnwise).

If option=‘R’, then “m” is overwritten with the first min(M,N) rows of right singular vectors (stored rowwise in transposed form).

And if option=‘N’, neither right nor left singular vectors are computed and the contents of “m” is destroyed (used as workspace internally by this function).

“sval” is updated with the singular values in sorted order, so that  $\text{sval}(i) \geq \text{sval}(i+1)$ .

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.8 gesvd (m, sval, svec, vtype=‘L’, part=‘A’, opt\_a=‘N’)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

*sval*: An empty vector of the type `std::vector<T>` (output)

*svec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*vtype*: A character value containing either ‘L’ or ‘R’ [Default: ‘L’] (input/optional)

*part*: A character value containing either ‘A’ or ‘S’ [Default: ‘A’] (input/optional)

*opt\_a*: A character value containing either ‘O’ or ‘N’ [Default: ‘N’] (input/optional)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. Additionally, it also computes *left and/or right singular vectors*.

If vtype=‘L’ and part=‘A’, “svec” will be updated with all the M columns of left singular vectors. In that case, “svec” should have at least M number of rows and M number of columns.

If vtype=‘L’ and part=‘S’, “svec” will be updated with first min(M,N) columns of left singular vectors (stored columnwise). In that case, “svec” should have at least M number of rows and min(M,N) number of columns.

If vtype=‘R’ and part=‘A’, “svec” will be updated with all the N rows of right singular vectors (in transposed form). In that case, “svec” should have at least N number of rows and N number of columns.

If vtype=‘R’ and part=‘S’, “svec” will be updated with first min(M,N) rows of right singular vectors (stored rowwise in transposed form). In that case, “svec” should have at least min(M,N) number of rows and N number of columns.

This function expects that required memory would be allocated for the output matrix “svec” beforehand. If it is not allocated, an exception will be thrown.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “svec” is a valid sized (as explained above) matrix.

And on exit, If opt\_a=‘N’, then the contents of “m” will be destroyed (internally used as workspace).

If opt\_a=‘O’ and vtype=‘L’, then “m” will be overwritten with first min(M,N) rows of right singular vectors (stored rowwise in transposed form).

And If opt\_a=‘O’ and vtype=‘R’, then “m” will be overwritten with first min(M,N) columns of left singular vectors (stored columnwise).

“sval” is updated with the singular values in sorted order, so that  $\text{sval}(i) \geq \text{sval}(i+1)$  and “svec” will be updated with the desired singular vectors (as explained above).

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.9 gesvd (m, sval, lsvect, rsvect, part\_l='A', part\_r='A')****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

*lsvect*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*rsvect*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*part\_l*: A character containing either 'A' or 'S' [Default: 'A'] (input/optional)

*part\_r*: A character containing either 'A' or 'S' [Default: 'A'] (input/optional)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. Additionally, it also computes *left and right singular vectors*.

This function expects that required memory would be allocated for the output matrices “lsvect” and “rsvect” beforehand, to store the left and right singular vectors respectively. If they are not allocated, an exception will be thrown.

If *part\_l*='A', “lsvect” will be updated with all the M columns of left singular vectors. Thus, “lsvect” should have at least M number of rows and M number of columns.

If *part\_l*='S', “lsvect” will be updated with first min(M,N) columns of left singular vectors (stored columnwise). Thus, “lsvect” should have at least M number of rows and min(M,N) number of columns.

If *part\_r*='A', “rsvect” will be updated with all the N rows of right singular vectors (in transposed form). Thus, “rsvect” should have at least N number of rows and N number of columns.

If *part\_r*='S', “rsvect” will be updated with first min(M,N) rows of right singular vectors (stored rowwise in transposed form). Thus, “rsvect” should have at least min(M,N) number of rows and N number of columns.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “lsvect” and “rsvect” are valid sized (as explained above) matrices.

And on exit, the contents of “m” is destroyed (internally used as workspace), “sval” is updated with the singular values in sorted order, so that  $sval(i) \geq sval(i+1)$ , and “lsvect” and “rsvect” are updated with the left and right singular vectors respectively (as explained above).

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.10 gesdd (m, sval)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. But neither left nor right singular vectors are computed. Please refer to *lapack guide* to know the algorithmic differences between `gesvd()` and `gesdd()`.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`. And on successful exit, the contents of “m” is destroyed (used as workspace internally by this function) and “sval” is updated with the singular values in sorted order, so that  $sval(i) \geq sval(i+1)$ .

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.11 gesdd (m, sval, svec)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

*svec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. Additionally, it also computes *full or some part of left and right singular vectors* using divide-and-conquer algorithm. Please refer to *lapack documentation* to know the algorithmic differences between `gesvd()` and `gesdd()`.

If  $M \geq N$ , matrix “m” will be overwritten with the first N columns of the left singular vectors and “svec” will be updated with all the N rows of right singular vectors (in transposed form). In that case, “svec” should have at least N number of rows and N number of columns.

Otherwise, matrix “m” will be overwritten with first M rows of the right singular vectors (in transposed form) and “svec” will be updated with all the M columns of the left singular vectors. In that case, “svec” should have at least M number of rows and M number of columns.

This function expects that required memory would be allocated for the output matrix “svec” beforehand. If it is not allocated, an exception will be thrown.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “svec” is a valid sized (as explained above) matrix. And on successful exit, “m” and “svec” will be updated with the values (as explained above) and “sval” will be updated with singular values in sorted order, so that  $sval(i) \geq sval(i+1)$ .

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.12 gesdd (m, sval, lsvec, rsvec, part\_lr='A')****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

*lsvec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*rsvec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*part\_lr*: A character containing either ‘A’ or ‘S’ [Default: ‘A’] (input/optional)

**Purpose**

It computes the singular value decomposition (SVD) of an M-by-N matrix. Additionally, it also computes *full or some part of left and right singular vectors* using divide-and-conquer algorithm. But like the previous version of `gesdd` (with three parameters), it does not overwrite the results on matrix “m” (since it accepts memory locations for both the left and right singular vectors separately). Please refer to *lapack guide* to know the algorithmic differences between `gesvd()` and `gesdd()`.

If *part\_lr*='A', all the M columns of left singular vectors and all the N rows of right singular vectors (in transposed form) are stored in output matrix “lsvec” and “rsvec” respectively. In that case “lsvec” should have at least M number of rows and M number of columns and “rsvec” should have at least N number of rows and N number of columns.

If *part\_lr*='S', the first  $\min(M,N)$  columns of left singular vectors are stored in “lsvec” and the first  $\min(M,N)$  rows of right singular vectors are stored in “rsvec” (in transposed form). In that case “lsvec” should have at least M number of rows and  $\min(M,N)$  number of columns and “rsvec” should have at least  $\min(M,N)$  number of rows and N number of columns.



This function expects that required memory would be allocated for the output matrices “lsvec” and “rsvec” beforehand. If they are not allocated, an exception will be thrown.

On entry “m” contains the matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “lsvec” and “rsvec” are valid sized (as explained above) matrices. And on successful exit, the contents of “m” will be destroyed (used internally as workspace), “lsvec” and “rsvec” will be updated with the values (as explained above) and “sval” will be updated with singular values in sorted order, so that  $sval(i) \geq sval(i+1)$ .

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.13 gelsy (m1, m2, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

It computes the minimum-norm solution to a real linear least squares problem:

`minimize || A * X - B ||`

using a complete orthogonal factorization of A. A is an M-by-N matrix which may be rank-deficient.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B). On successful exit, “m1” is overwritten with its complete orthogonal factorization and “m2” is overwritten with the solution matrix (X).

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.14 gelsy (m1, m2, rank, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*rank*: An empty integer object passed by reference (output)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

The native lapack routine can also determine the rank of the matrix “m1” while finding the minimum-norm solution. If it is required to know the rank determined by this function, it is recommended to use this version of `gelsy()`.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B), “rank” is just an empty integer passed by reference to this routine. On successful exit, “m1” is overwritten

with its complete orthogonal factorization, “m2” is overwritten with the solution matrix (X) and “rank” is overwritten with the determined effective rank of matrix “m1”.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.15 gelss (m1, m2, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

It solves overdetermined or underdetermined systems for general matrices. It computes the minimum-norm solution to a real linear least squares problem:

`minimize 2-norm (| B - AX |)`

using the singular value decomposition (SVD) of A. A is an M-by-N general matrix which may be rank-deficient.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B). On successful exit, first  $\min(M,N)$  rows of “m1” is overwritten with its right singular vectors (stored rowwise) and “m2” is overwritten with the solution matrix (X).

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.16 gelss (m1, m2, sval, rank, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty object of the type `std::vector<T>` (output)

*rank*: An empty integer object passed by reference (output)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

It solves overdetermined or underdetermined systems for general matrices. It computes the minimum-norm solution to a real linear least squares problem:

`minimize 2-norm (| B - AX |)`

using the singular value decomposition of A. A is an M-by-N general matrix which may be rank-deficient.

It might also be needed to obtain the computed singular values and effective rank of the matrix A. In that case, this version of gelss(with five arguments) is recommended to use. It accepts an empty vector (3rd argument) and an empty integer (4th argument) which are passed by reference to this function.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B). On successful exit, first  $\min(M,N)$  rows of “m1” is overwritten with its right singular vectors (stored rowwise), “m2” is overwritten with the solution matrix (X), computed singular values of “m1” are stored in “sval” in decreasing order and “rank” is updated with the computed effective rank of “m1”.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.17 gelsd (m1, m2, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

It solves overdetermined or underdetermined systems for general matrices. It computes the minimum-norm solution to a real linear least squares problem:

`minimize 2-norm (| B - AX |)`

using the singular value decomposition (SVD) of A. A is an M-by-N general matrix which may be rank-deficient. Please refer to *lapack guide* to know the algorithmic differences between `gelsd()` and `gelss()`.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B). On successful exit, first  $\min(M,N)$  rows of “m1” is overwritten with its right singular vectors (stored rowwise) and “m2” is overwritten with the solution matrix (X).

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.18 gelsd (m1, m2, sval, rank, rcond=-1)

##### Parameters

*m1*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*m2*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*sval*: An empty object of the type `std::vector<T>` (output)

*rank*: An empty integer object passed by reference (output)

*rcond*: A T type object (float or double) [Default: -1] (input/optional)

##### Purpose

It solves overdetermined or underdetermined systems for general matrices. It computes the minimum-norm solution to a real linear least squares problem:

`minimize 2-norm (| B - AX |)`

using the singular value decomposition of A. A is an M-by-N general matrix which may be rank-deficient. Please refer to *lapack guide* to know the algorithmic differences between `gelsd()` and `gelss()`.

It might also be needed to obtain the computed singular values and effective rank of the matrix A. In that case, this version of `gelsd`(with five arguments) is recommended to use. It accepts an empty vector (3rd argument) and an empty integer (4th argument) which are passed by reference to this function.

The input parameter “rcond” is used to determine the effective rank of matrix “m1”. If “rcond” is less than zero, machine precision is used instead. The matrix “m2” should have number of rows  $\geq \max(M,N)$  and at least 1 column.

On entry, “m1” contains the left-hand-side matrix (A) and “m2” contains the right-hand-side matrix (B). On successful exit, first  $\min(M,N)$  rows of “m1” is overwritten with its right singular vectors (stored rowwise), “m2” is overwritten with the solution matrix (X), computed singular values of “m1” are stored in “sval” in decreasing order and “rank” is updated with the computed effective rank of “m1”.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.19 geev (m, eval)

##### Parameters

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*eval*: An empty object of the type `std::vector<T>` (output)

##### Purpose

It computes eigenvalues for an N-by-N real nonsymmetric matrix.

The input matrix, “m” must be a square matrix. Else it will throw an exception.

On entry, “m” is the square matrix whose eigenvalues are to be computed and “eval” is an empty vector. On successful exit, the contents of “m” is destroyed, and the computed eigenvalues are stored in “eval”.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 12.5.1.20 geev (m, eval, evec, vtype='L')

##### Parameters

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*eval*: An empty object of the type `std::vector<T>` (output)

*evec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*vtype*: A character value containing either ‘L’ or ‘R’ [Default: ‘L’] (input/optional)

##### Purpose

It computes eigenvalues for an N-by-N real nonsymmetric matrix. Additionally, it also computes the left or right eigenvectors.

The input matrix, “m” must be a square matrix. Else it will throw an exception. If *vtype*=‘L’, then left-eigenvectors will be computed.

If *vtype*=‘R’, then right-eigenvectors will be computed. The output matrix “evec” must have at least N number of rows and N number of columns. This function expects that “evec” is already allocated before its call. Thus if it is not allocated, an exception will be thrown.

On entry, “m” is the square matrix whose eigenvalues are to be computed, “eval” is an empty vector, “evec” is an empty matrix with valid size (as mentioned above). On successful exit, the contents of “m” is destroyed, the computed eigenvalues are stored in “eval” and “evec” is updated with the desired (left/right) eigenvectors.

### Return Value

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.21 geev (m, eval, levec, revec)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (inout)

*eval*: An empty object of the type `std::vector<T>` (output)

*levec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

*revec*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (output)

**Purpose**

It computes eigenvalues for an N-by-N real nonsymmetric matrix. Additionally, it also computes the left and right eigenvectors.

The input matrix, “m” must be a square matrix. Else it will throw an exception. The output matrices “levec” and “revec” must have at least N number of rows and N number of columns. This function expects that these output matrices are already allocated before its call. Thus if they are not allocated, an exception will be thrown.

On entry, “m” is the square matrix whose eigenvalues are to be computed, “eval” is an empty vector, “levec” and “revec” are an empty matrices with valid size (as mentioned above). On successful exit, the contents of “m” is destroyed, the computed eigenvalues are stored in “eval”, “levec” is updated with the left eigenvectors and “revec” is updated with right eigenvectors.

**Return Value**

On success, it returns the exit status of the lapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

**12.5.1.22 inv (m)****Parameters**

*m*: A `colmajor_matrix_local<T>` or a `sliced_colmajor_matrix_local<T>` (input)

**Purpose**

It computes the inverse of a square matrix “m” by using `getrf()` and `getri()` internally. Thus it is a kind of short-cut function to obtain the inverse of a non-distributed matrix.

On successful exit, it returns the resultant inversed matrix. The input matrix “m” remains unchanged. Since it returns the resultant matrix, it can be used in any numerical expressions, along with other operators. E.g., if a and b are two colmajor matrices, then the expresion like, “a\*(~b)\*inv(a)” can easily be performed.

**Return Value**

On success, it returns the resultant matrix of the type `colmajor_matrix_local<T>`. If any error occurs, it throws an exception explaining cause of the error.

**12.6 SEE ALSO**

`sliced_colmajor_matrix_local`, `sliced_colmajor_vector_local`, `scalapack_wrapper`



## Chapter 13

# frovedis::blockcyclic\_matrix<T>

### 13.1 NAME

`frovedis::blockcyclic_matrix<T>` - two-dimensional blockcyclic distribution of a dense matrix over a MxN process grid (MxN = number of parallel processes)

### 13.2 SYNOPSIS

```
#include <frovedis/matrix/blockcyclic_matrix.hpp>
```

#### 13.2.1 Constructors

```
blockcyclic_matrix ()  
blockcyclic_matrix (size_t nrow, size_t ncol, size_t type)  
blockcyclic_matrix (frovedis::node_local<blockcyclic_matrix_local<T>>&& data)  
blockcyclic_matrix (const blockcyclic_matrix<T>& m)  
blockcyclic_matrix (blockcyclic_matrix<T>&& m)  
blockcyclic_matrix (const colmajor_matrix<T>& m, size_t type=2)  
blockcyclic_matrix (colmajor_matrix<T>&& m, size_t type=2)  
blockcyclic_matrix (const std::vector<T>& v, size_t type=1)  
blockcyclic_matrix (std::vector<T>&& v, size_t type=1)
```

#### 13.2.2 Overloaded Operators

```
blockcyclic_matrix<T>& operator= (const blockcyclic_matrix<T>& m)  
blockcyclic_matrix<T>& operator= (blockcyclic_matrix<T>&& m)
```

#### 13.2.3 Public Member Functions

```
std::vector<T> to_vector ()  
std::vector<T> moveto_vector ()  
colmajor_matrix<T> to_colmajor ()  
colmajor_matrix<T> moveto_colmajor ()  
rowmajor_matrix<T> to_rowmajor ()
```

```

blockcyclic_matrix<T> transpose ()
void set_num (size_t nrow, size_t ncol, size_t type=2)
void save (const std::string& file)
void savebinary (const std::string& dir)
void debug_print ()
size_t get_nrows ()
size_t get_ncols ()

```

### 13.2.4 Public Data Members

```

frovedis::node_local<blockcyclic_matrix_local<T>> data
size_t num_row
size_t num_col
size_t type

```

## 13.3 DESCRIPTION

`frovedis::blockcyclic_matrix<T>` is a special type of `frovedis::colmajor_matrix<T>` distributed in two-dimensional blockcyclic manner over a  $M \times N$  process grid ( $M \times N$  = number of parallel processes). This is a template based dense matrix with type “**T**” which can be either “**float**” or “**double**” (at this moment). Specifying other types as template argument may lead to invalid results. Currently `frovedis` only supports creation of two types of blockcyclic matrices.

*type-1 blockcyclic-matrix:*

In case of type-1 blockcyclic-matrix, the global matrix is distributed over a  $N \times 1$  process grid, where  $N$  is the number of parallel processes. This type of distribution is preferred while distributing a column-vector (a matrix with many rows and 1 column), in order to achieve a better load balance.

*type-2 blockcyclic-matrix:*

In case of type-2 blockcyclic-matrix, the global matrix is distributed over a  $M \times N$  process grid, where  $M = \sqrt{\text{number of parallel processes}}$  and  $N = (\text{number of parallel processes} / M)$ .

The specifications of the block size ( $MB \times NB$ , where  $MB$  is the number of rows in a block and  $NB$  is the number of columns in a block) are decided by the algorithm depending upon the global matrix dimensions and number of parallel processes. Some constructors also support user defined block size.

A `blockcyclic_matrix<T>` contains public member “data” of the type `node_local<blockcyclic_matrix_local<T>>`. The actual distributed matrices are contained in all the worker nodes locally, thus named as `blockcyclic_matrix_local<T>`. Each of these local matrices have the below structure:

```

template <class T>
struct blockcyclic_matrix_local {
    std::vector<T> val;          // the actual local distributed matrix
    std::vector<int> descA;      // the distributed information mapping array
    size_t local_num_row;       // number of rows in local matrix
    size_t local_num_col;       // number of columns in local matrix
    size_t type;                // type of the local matrix (Nx1 or MxN)
};

```

The global version of the matrix at master node contains only three information, the reference to these local matrices at worker nodes, the dimensions of the global matrix, i.e., number of rows and columns and the type of the distributed matrix.



```
template <class T>
struct blockcyclic_matrix {
    frovedis::node_local<blockcyclic_matrix_local<T>> data; // local matrix information
    size_t num_row; // number of rows in global matrix
    size_t num_col; // number of columns in global matrix
    size_t type; // type of the blockcyclic-matrix (Nx1 or MxN)
};
```

### 13.3.1 Constructor Documentation

#### 13.3.1.1 `blockcyclic_matrix ()`

This is the default constructor which creates an empty blockcyclic matrix with `num_row = num_col = 0` and `type = 2`.

#### 13.3.1.2 `blockcyclic_matrix (size_t nrow, size_t ncol, size_t t=2)`

This is the parameterized constructor which creates an empty blockcyclic matrix of the given dimension and type (default type=2).

#### 13.3.1.3 `blockcyclic_matrix (frovedis::node_local<blockcyclic_matrix_local<T>>&& data)`

This is the parameterized constructor which accepts an rvalue of the type `node_local<blockcyclic_matrix_local<T>>` and *moves* the contents to the created blockcyclic matrix. In general, this constructor is used internally by some other functions. But user may need this constructor while constructing their own blockcyclic matrix using the return value of some function (returning a `blockcyclic_matrix_local<T>`) called using “`frovedis::node_local::map`” (thus returned value would be an object of type `node_local<blockcyclic_matrix_local<T>`).

#### 13.3.1.4 `blockcyclic_matrix (const blockcyclic_matrix<T>& m)`

This is the copy constructor which creates a new blockcyclic matrix by deep-copying the contents of the input blockcyclic matrix.

#### 13.3.1.5 `blockcyclic_matrix (blockcyclic_matrix<T>&& m)`

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

#### 13.3.1.6 `blockcyclic_matrix (const colmajor_matrix<T>& m, size_t type=2)`

This is a special constructor for implicit conversion. It converts an input colmajor matrix to a blockcyclic matrix of the same global dimensions. The input matrix is unchanged after the conversion. Default type of the created blockcyclic matrix is 2 (desired type can be specified in second argument).

**13.3.1.7 blockcyclic\_matrix (colmajor\_matrix<T>&& m, size\_t type=2)**

This is a special constructor for implicit conversion. It converts an input colmajor matrix to a blockcyclic matrix of the same global dimensions. But during the conversion the memory buffer of input rvalue matrix is reused, thus the input colmajor matrix becomes invalid after this conversion. Default type of the created blockcyclic matrix is 2 (desired type can be specified in second argument).

**13.3.1.8 blockcyclic\_matrix (const std::vector<T>& v, size\_t type=1)**

This is a special constructor for implicit conversion. It converts an input lvalue `std::vector<T>` to `blockcyclic_matrix<T>` with global dimensions  $N \times 1$ , where  $N$  = size of the input vector. The input vector is unchanged after the conversion. Default type of the created blockcyclic matrix is 1 to support load balancing (desired type can be specified in second argument).

**13.3.1.9 blockcyclic\_matrix (std::vector<T>&& v, size\_t type=1)**

This is a special constructor for implicit conversion. It converts an input rvalue `std::vector<T>` to `blockcyclic_matrix<T>` with global dimensions  $N \times 1$ , where  $N$  = size of the input vector. But during the conversion, the memory buffer of the input rvalue vector is reused, thus it becomes invalid after this conversion. Default type of the created blockcyclic matrix is 1 to support load balancing (desired type can be specified in second argument).

**13.3.2 Overloaded Operator Documentation****13.3.2.1 blockcyclic\_matrix<T>& operator= (const blockcyclic\_matrix<T>& m)**

It deep-copies the input blockcyclic matrix into the left-hand side matrix of the assignment operator “=”.

**13.3.2.2 blockcyclic\_matrix<T>& operator= (blockcyclic\_matrix<T>&& m)**

Instead of copying, it moves the contents of the input rvalue blockcyclic matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**13.3.3 Public Member Function Documentation****13.3.3.1 std::vector<T> to\_vector ()**

If `num_col = 1`, it converts the blockcyclic matrix to `std::vector<T>` and returns the same, else it throws an exception. The blockcyclic matrix is unchanged.

**13.3.3.2 std::vector<T> moveto\_vector ()**

If `num_col = 1` and `type = 1`, it converts the blockcyclic matrix to `std::vector<T>` and returns the same, else it throws an exception. Due to move operation, input matrix becomes invalid after the conversion.

**13.3.3.3 colmajor\_matrix<T> to\_colmajor ()**

It converts the blockcyclic matrix to colmajor matrix and returns the same. Input matrix is unchanged.

**13.3.3.4 colmajor\_matrix<T> moveto\_colmajor ()**

Only when `type = 1`, it converts the blockcyclic matrix to colmajor matrix and returns the same, else it throws an exception. During the conversion it reuses the memory buffer of the blockcyclic matrix, thus it would become invalid.

**13.3.3.5 rowmajor\_matrix<T> to\_rowmajor ()**

It converts the blockcyclic matrix to rowmajor matrix and returns the same. The blockcyclic matrix is unchanged.

**13.3.3.6 blockcyclic\_matrix<T> transpose ()**

It returns the transposed blockcyclic matrix of the source matrix object.

**13.3.3.7 void set\_num (size\_t nrow, size\_t ncol, size\_t type=2)**

It sets the global matrix information as specified. Default type is considered as 2, if *type* value is not provided.

**13.3.3.8 void save (const std::string& file)**

It writes the blockcyclic matrix to the specified file in rowmajor format with text data.

**13.3.3.9 void savebinary (const std::string& dir)**

It writes the blockcyclic matrix to the specified directory in rowmajor format with binary data (in little-endian form).

**13.3.3.10 void debug\_print ()**

It prints the contents and other information of the local matrices node-by-node on the user terminal. It is mainly useful for debugging purpose.

**13.3.3.11 size\_t get\_nrows ()**

It returns the global number of rows in the source blockcyclic matrix object.

**13.3.3.12 size\_t get\_ncols ()**

It returns the global number of columns in the source blockcyclic matrix object.

**13.3.4 Public Data Member Documentation****13.3.4.1 data**

An instance of `node_local<blockcyclic_matrix_local<T>>` which contains the references to the local matrices in the worker nodes.

### 13.3.4.2 num\_row

A `size_t` attribute to contain the number of rows in the global blockcyclic matrix.

### 13.3.4.3 num\_col

A `size_t` attribute to contain the number of columns in the global blockcyclic matrix.

## 13.3.5 Public Global Function Documentation

### 13.3.5.1 make\_blockcyclic\_matrix\_loadbinary(dirname, type, MB, NB)

#### Parameters

*dirname*: A string object containing the name of the directory having binary data to be loaded.

*type*: A `size_t` attribute containing the desired type of the matrix to be created (optional, default=2).

*MB*: A `size_t` attribute containing the desired number of rows in a block (optional, default=0).

*NB*: A `size_t` attribute containing the desired number of columns in a block (optional, default=0).

#### Purpose

This function loads the (little-endian) binary data from the specified directory and creates a blockcyclic matrix of default type = 2 and algorithm decided block size (if not defined by the user, i.e., MB=NB=0). The required type and block size can be specified.

#### Return Value

On success, it returns the created blockcyclic matrix of the type `blockcyclic_matrix<T>`. Otherwise, it throws an exception.

### 13.3.5.2 make\_blockcyclic\_matrix\_load(fname, type, MB, NB)

#### Parameters

*fname*: A string object containing the name of the data file.

*type*: A `size_t` attribute containing the desired type of the matrix to be created (optional, default=2).

*MB*: A `size_t` attribute containing the desired number of rows in a block (optional, default=0).

*NB*: A `size_t` attribute containing the desired number of columns in a block (optional, default=0).

#### Purpose

This function loads the data from the specified text file and creates a blockcyclic matrix of default type = 2 and algorithm decided block size (if not defined by the user, i.e., MB=NB=0). The required type and block size can be specified.

#### Return Value

On success, it returns the created blockcyclic matrix of the type `blockcyclic_matrix<T>`. Otherwise, it throws an exception.

### 13.3.5.3 make\_blockcyclic\_matrix\_scatter(rmat, type, MB, NB)

#### Parameters

*rmat*: An object of the type `rowmajor_matrix_local<T>` containing the data to be scattered.

*type*: A `size_t` attribute containing the desired type of the matrix to be created (optional, default=2).

*MB*: A `size_t` attribute containing the desired number of rows in a block (optional, default=0).

*NB*: A `size_t` attribute containing the desired number of columns in a block (optional, default=0).

**Purpose**

This function scatters an input `frovedis::rowmajor_matrix_local<T>` as per the active number of parallel processes and from the scattered data it creates a blockcyclic matrix of default type = 2 and algorithm decided block size (if not defined by the user, i.e., MB=NB=0). The required type and block size can be specified.

**Return Value**

On success, it returns the created blockcyclic matrix of the type `blockcyclic_matrix<T>`. Otherwise, it throws an exception.

**13.3.5.4 vec\_to\_bcm(vec, type, MB, NB)****Parameters**

*vec*: An object of the type `std::vector<T>` containing the data values.

*type*: A `size_t` attribute containing the desired type of the matrix to be created (optional, default=1).

*MB*: A `size_t` attribute containing the desired number of rows in a block (optional, default=0).

*NB*: A `size_t` attribute containing the desired number of columns in a block (optional, default=0).

**Purpose**

This function scatters the input vector as per the active number of parallel processes and from the scattered data it creates a blockcyclic matrix of default type = 1 (for a better load balancing) and algorithm decided block size (if not defined by the user, i.e., MB=NB=0). The required type and block size can be specified. If the input vector is an *lvalue*, it copies the data before scattering. But if the vector is an *rvalue*, it ignores copying the data.

**Return Value**

On success, it returns the created blockcyclic matrix of the type `blockcyclic_matrix<T>`. Otherwise, it throws an exception.

**13.4 SEE ALSO**

`colmajor_matrix`, `rowmajor_matrix`, `sliced_blockcyclic_matrix`, `sliced_blockcyclic_vector`



## Chapter 14

# frovedis::sliced\_blockcyclic\_matrix<T>

### 14.1 NAME

`frovedis::sliced_blockcyclic_matrix<T>` - a data structure containing the slicing information of a two-dimensional `frovedis::blockcyclic_matrix<T>`

### 14.2 SYNOPSIS

```
#include <frovedis/matrix/sliced_matrix.hpp>
```

#### 14.2.1 Constructors

```
sliced_blockcyclic_matrix ()  
sliced_blockcyclic_matrix (const blockcyclic_matrix<T>& m)
```

#### 14.2.2 Public Member Functions

```
void set_num (size_t nrow, size_t ncol)
```

#### 14.2.3 Public Data Members

```
node_local<sliced_blockcyclic_matrix_local<T>> data  
size_t num_row  
size_t num_col
```

### 14.3 DESCRIPTION

In order to perform matrix operations on sub-matrices instead of entire physical matrix, `frovedis` provides some sliced data structures. `sliced_blockcyclic_matrix<T>` is one of them. It is actually not a real matrix, rather it only contains some slicing information of a physical `blockcyclic_matrix<T>`. Thus any changes performed on the sliced matrix, would actually make changes on the physical matrix from which slice was made.

Like `blockcyclic_matrix<T>`, a `sliced_blockcyclic_matrix<T>` is also a template based structure with type “**T**” which can be either “**float**” or “**double**” (at this moment). Specifying other types can cause floating point exception issues.

A `sliced_blockcyclic_matrix<T>` contains public member “data” of the type `node_local<sliced_blockcyclic_matrix_local<T>>`. The actual distributed sliced matrices are contained in all the worker nodes locally, thus named as `sliced_blockcyclic_matrix_local<T>`. Each of this local matrix has the below structure:

```
template <class T>
struct sliced_blockcyclic_matrix_local {
    T *data; // pointer to the beginning of the physical local matrix
    int *descA; // pointer to the descriptor array of the physical local matrix
    size_t IA; // row-id of the physical matrix starting row from which to slice
    size_t JA; // col-id of the physical matrix starting col from which to slice
    size_t sliced_num_row; // number of rows in the global sliced matrix
    size_t sliced_num_col; // number of columns in the global sliced matrix
};
```

E.g., if a physical `blockcyclic_matrix<T>` M has dimension 4x4 and slice is needed from 2nd row and 2nd column till 3rd row and 3rd column, then

“data” in each node will hold the address of local blockcyclic matrix of that node (data -> &local\_m[0][0]),  
“descA” in each node will hold the address of the array descriptor of the local blockcyclic matrix of that node (descA -> &local\_m.descA[0]),

“IA” will be 2 (starting from 2nd row - so row id is 2),

“JA” will be 2 (starting from 2nd column - so col id is 2),

From 2nd row till 3rd row, number of rows to be sliced is 2, thus “sliced\_num\_row” would be 2.

From 2nd column till 3rd column, number of columns to be sliced is 2, thus “sliced\_num\_col” would be 2.

Kindly note that IA and JA do not contain the index, instead they contain the id. And also in each local sliced matrices the sliced information IA, JA, sliced\_num\_row, sliced\_num\_col would be same. The only difference would be in the pointer values, i.e., in data and descA.

The global version, `sliced_blockcyclic_matrix<T>` at master node actually contains nothing but the reference to these local sliced matrices at worker nodes and the global matrix dimension, i.e., the actual number of rows and columns in the physical global blockcyclic matrix. It has the below structure:

```
template <class T>
struct sliced_blockcyclic_matrix {
    node_local<sliced_blockcyclic_matrix_local<T>> data; // local matrix information
    size_t num_row; // actual number of rows in physical global matrix
    size_t num_col; // actual number of columns in physical global matrix
};
```

Such matrices are very useful in operations of external libraries like pblas, scalapack etc.

## 14.3.1 Constructor Documentation

### 14.3.1.1 sliced\_blockcyclic\_matrix ()

Default constructor. It creates an empty sliced matrix with num\_row = num\_col = 0 and local data pointers pointing to NULL. Basically of no use, unless it is needed to manipulate the slice information manually.



**14.3.1.2 sliced\_blockcyclic\_matrix (const blockcyclic\_matrix<T>& m)**

Special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced matrix. Thus the created `sliced_blockcyclic_matrix<T>` would have same dimension as with the input `blockcyclic_matrix<T>` and local data pointers pointing to the base address of the local blockcyclic matrices.

**14.3.2 Public Member Function Documentation****14.3.2.1 void set\_num (size\_t nrow, size\_t ncol)**

This function sets the “num\_row” and “num\_col” fields with the actual number of rows and columns in the global physical blockcyclic matrix. Only useful when manual manipulation is required.

**14.3.3 Public Data Member Documentation****14.3.3.1 data**

An instance of `node_local<sliced_blockcyclic_matrix_local<T>>` which contains the references to the local sliced matrices in the worker nodes.

**14.3.3.2 num\_row**

A `size_t` attribute to contain the actual number of rows in the physical global blockcyclic matrix.

**14.3.3.3 num\_col**

A `size_t` attribute to contain the actual number of columns in the physical global blockcyclic matrix.

**14.3.4 Public Global Function Documentation****14.3.4.1 make\_sliced\_blockcyclic\_matrix ()**

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and slicing information like row and column index from which slicing is to be started, and the size of the output sliced matrix, i.e., number of rows and columns to be sliced from the starting location. On receiving the valid inputs, it outputs a `sliced_blockcyclic_matrix<T>` object containing the reference to the local sliced matrices, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_matrix<T>
make_sliced_blockcyclic_matrix (const sliced_blockcyclic_matrix<T>& mat,
                                size_t start_row_index,
                                size_t start_col_index,
                                size_t num_row,
                                size_t num_col);
```

Please note that in case a `blockcyclic_matrix<T>` is passed to this function, the entire matrix would be treated as a `sliced_blockcyclic_matrix<T>` because of the implicit conversion constructor (explained above). Thus this function can be used to obtain a sliced matrix from both a physical `blockcyclic_matrix<T>` and a valid `sliced_blockcyclic_matrix<T>`.

**Example:** If a physical `blockcyclic_matrix<T>` “mat” has the dimension 4x4 and slicing is required from its 2nd row and 2nd column till 4th row and 4th column, then this function should be called like:

```
auto smat = make_sliced_blockcyclic_matrix(mat,1,1,3,3);
```

Index of the 2nd row is 1, thus `start_row_index = 1`

Index of the 2nd column is 1, thus `start_col_index = 1`

From 2nd row till 4th row, number of rows to be sliced is 3, thus `num_row = 3`

From 2nd column till 4th column, number of columns to be sliced is 3, thus `num_col = 3`

Input (mat):		Output (smat):
-----		-----
1 2 3 4		6 7 8
5 6 7 8	=>	7 6 5
8 7 6 5		3 2 1
4 3 2 1		

Now if we need to slice further this sliced matrix, “smat” from its 2nd row and 2nd column till 3rd row and 3rd column, then we would call this function like below:

```
auto ssmat = make_sliced_blockcyclic_matrix(smat,1,1,2,2);
```

Index of the 2nd row of smat is 1, thus `start_row_index = 1`

Index of the 2nd column of smat is 1, thus `start_col_index = 1`

From 2nd row till 3rd row of smat, number of rows to be sliced is 2, thus `num_row = 2`

From 2nd column till 3rd column of smat, number of columns to be sliced is 2, thus `num_col = 2`

Kindly note that 2nd row of “smat” is actually the 3rd row of the physical matrix “mat”, but this function takes care of it internally. Thus you just need to take care of the index of the input sliced matrix, not the actual physical matrix

Input (smat):		Output (ssmat):
-----		-----
6 7 8		6 5
7 6 5	=>	2 1
3 2 1		

The above input/output is presented just to explain the slicing concept. The internal storage representation of these sliced blockcyclic matrices would be a bit different and complex in nature.

## 14.4 SEE ALSO

`blockcyclic_matrix`, `sliced_blockcyclic_vector`

## Chapter 15

# frovedis::sliced\_blockcyclic\_vector<T>

### 15.1 NAME

`frovedis::sliced_blockcyclic_vector<T>` - a data structure containing the row or column wise slicing information of a two-dimensional `frovedis::blockcyclic_matrix<T>`

### 15.2 SYNOPSIS

```
#include <frovedis/matrix/sliced_vector.hpp>
```

#### 15.2.1 Constructors

```
sliced_blockcyclic_vector ()  
sliced_blockcyclic_vector (const blockcyclic_matrix<T>& m)
```

#### 15.2.2 Public Member Functions

```
void set_num (size_t len)
```

#### 15.2.3 Public Data Members

```
node_local<sliced_blockcyclic_vector_local<T>> data  
size_t size
```

### 15.3 DESCRIPTION

In order to perform vector operations on some rows or on some columns of a dense matrix, `frovedis` provides some sliced data structures. `sliced_blockcyclic_vector<T>` is one of them. It is actually not a real vector, rather it only contains some slicing information of a physical `blockcyclic_matrix<T>`. Thus any changes performed on the sliced vector, would actually make changes on the specific row or column of the physical matrix from which the slice was made.

Like `blockcyclic_matrix<T>`, a `sliced_blockcyclic_vector<T>` is also a template based structure with type “**T**” which can be either “**float**” or “**double**” (at this moment). Specifying other types can cause floating point exception issues.

A `sliced_blockcyclic_vector<T>` contains public member “data” of the type `node_local<sliced_blockcyclic_vector_local<T>>`. The actual distributed sliced vectors are contained in all the worker nodes locally, thus named as `sliced_blockcyclic_vector_local<T>`. Each of this local vector has the below structure:

```
template <class T>
struct sliced_blockcyclic_vector_local {
    T *data;    // pointer to the beginning of the physical local matrix
    int *descA; // pointer to the descriptor array of the physical local matrix
    size_t IA;  // row-id of the physical matrix starting row from which to slice
    size_t JA;  // col-id of the physical matrix starting col from which to slice
    size_t size; // number of elements in the sliced vector
    size_t stride; // stride between two consecutive elements of the sliced vector
};
```

E.g., if a physical `blockcyclic_matrix<T>` M has dimension 4x4 and its 2nd row needs to be sliced, then  
 “data” in each node will hold the address of local blockcyclic matrix of that node (data -> &local\_m[0][0]),  
 “descA” in each node will hold the address of the array descriptor of the local blockcyclic matrix of that node (descA -> &local\_m.descA[0]),  
 “IA” will be 2 (row id of the 2nd row is 2),  
 “JA” will be 1 (since complete 2nd row needs to be sliced, column id of the first element in the 2nd row would be 1),  
 “size” will be 4 (number of elements in 2nd row),  
 “stride” will be 4 (since matrix is stored in colmajor order, the stride between two consecutive elements in a row would be equal to leading dimension of that matrix, i.e., number of rows in that matrix)

On the other hand, if 2nd column needs to be sliced, then  
 “data” in each node will hold the address of local blockcyclic matrix of that node (data -> &local\_m[0][0]),  
 “descA” in each node will hold the address of the array descriptor of the local blockcyclic matrix of that node (descA -> &local\_m.descA[0]),  
 “IA” will be 1 (since complete 2nd column needs to be sliced, row id of the first element in the 2nd column would be 1),  
 “JA” will be 2 (column id of the 2nd column is 2),  
 “size” will be 4 (number of elements in 2nd column),  
 “stride” will be 1 (since matrix is stored in colmajor order, the consecutive elements in a column would be placed one after another)

Kindly note that IA and JA do not contain the index, instead they contain the id. And also in each local sliced vectors the sliced information IA, JA, size, stride would be same. The only difference would be in the pointer values, i.e., in data and descA.

The global version, `sliced_blockcyclic_vector<T>` at master node actually contains nothing but the reference to these local sliced vectors at worker nodes and the global size of the distributed row or column vector. It has the below structure:

```
template <class T>
struct sliced_blockcyclic_vector {
    node_local<sliced_blockcyclic_vector_local<T>> data; // local vector information
    size_t size; // actual no. of elements in the target row/col in the physical matrix
};
```

Such vectors are very useful in operations of external libraries like pblas etc.

### 15.3.1 Constructor Documentation

#### 15.3.1.1 `sliced_blockcyclic_vector ()`

Default constructor. It creates an empty sliced vector with `size = 0` and local data pointers pointing to `NULL`. Basically of no use, unless it is needed to manipulate the slice information manually.

#### 15.3.1.2 `sliced_blockcyclic_vector (const blockcyclic_matrix<T>& m)`

Special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced vector. Thus the created `sliced_blockcyclic_vector<T>` would have “size” equals to number of rows in the input `blockcyclic_matrix<T>` and “data” pointer pointing to the base address of the local blockcyclic matrices. Please note that such conversion can only be possible if the input matrix can be treated as a column vector (a matrix with multiple rows and single column), else it throws an exception.

### 15.3.2 Public Member Function Documentation

#### 15.3.2.1 `void set_num (size_t len)`

This function sets the “size” field with the actual number of elements in the target row or column in the global physical blockcyclic matrix. Only useful when manual manipulation is required.

### 15.3.3 Public Data Member Documentation

#### 15.3.3.1 `data`

An instance of `node_local<sliced_blockcyclic_vector_local<T>>` which contains the references to the local sliced vectors in the worker nodes.

#### 15.3.3.2 `size`

A `size_t` attribute to contain the actual number of elements in the target row or column in the physical global blockcyclic matrix.

### 15.3.4 Public Global Function Documentation

#### 15.3.4.1 `make_row_vector ()`

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and the row index to be sliced. On receiving the valid inputs, it outputs a `sliced_blockcyclic_vector<T>` object containing the reference to the local sliced vectors, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_vector<T>
make_row_vector (const sliced_blockcyclic_matrix<T>& mat,
                 size_t row_index);
```

Please note that in case a `blockcyclic_matrix<T>` is passed to this function, the entire matrix would be treated as a `sliced_blockcyclic_matrix<T>` because of the implicit conversion constructor (as explained in *sliced\_blockcyclic\_matrix* manual page). Thus this function can be used to obtain a row vector from both a physical `blockcyclic_matrix<T>` and a valid `sliced_blockcyclic_matrix<T>`.

**Example:** If a physical `blockcyclic_matrix<T>` “mat” has the dimension 4x4 and its 2nd row needs to be sliced, then this function should be called like:

```
auto rvec = make_row_vector(mat,1); // row index of second row is 1
```

Input (mat):		Output (rvec):
-----		-----
1 2 3 4	=>	5 6 7 8
5 6 7 8		
8 7 6 5		
4 3 2 1		

Now if it is needed to slice the 2nd row from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat = make_sliced_blockcyclic_matrix(mat,2,2,2,2);
auto s_rvec = make_row_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd row is to be sliced from the sub-matrix.

Kindly note that 2nd row of “smat” is actually the 4th row of the physical matrix “mat”, but this function takes care of it internally. Thus you just need to take care of the index of the input sliced matrix, not the actual physical matrix.

Input (mat):		Output (smat):		Output (s_rvec):
-----		-----		-----
1 2 3 4		6 5	=>	2 1
5 6 7 8	=>	2 1		
8 7 6 5				
4 3 2 1				

#### 15.3.4.2 make\_col\_vector ()

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and the column index to be sliced. On receiving the valid inputs, it outputs a `sliced_blockcyclic_vector<T>` object containing the reference to the local sliced vectors, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_vector<T>
make_col_vector (const sliced_blockcyclic_matrix<T>& mat,
                 size_t col_index);
```

Please note that in case a `blockcyclic_matrix<T>` is passed to this function, the entire matrix would be treated as a `sliced_blockcyclic_matrix<T>` because of the implicit conversion constructor (as explained in *sliced\_blockcyclic\_matrix* manual page). Thus this function can be used to obtain a column vector from both a physical `blockcyclic_matrix<T>` and a valid `sliced_blockcyclic_matrix<T>`.

**Example:** If a physical `blockcyclic_matrix<T>` “mat” has the dimension 4x4 and its 2nd column needs to be sliced, then this function should be called like:

```
auto cvec = make_col_vector(mat,1); // col index of second col is 1
```

Input (mat):		Output (cvec):
-----		-----
1 2 3 4	=>	2 6 7 3
5 6 7 8		
8 7 6 5		
4 3 2 1		

Now if it is needed to slice the 2nd column from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat = make_sliced_blockcyclic_matrix(mat,2,2,2,2);
auto s_cvec = make_col_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd column is to be sliced from the sub-matrix.

Kindly note that 2nd column of “smat” is actually the 4th column of the physical matrix “mat”, but this function takes care of it internally. Thus you just need to take care of the index of the input sliced matrix, not the actual physical matrix.

Input (mat):	Output (smat):	Output (s_cvec):
-----	-----	-----
1 2 3 4	6 5	=> 5 1
5 6 7 8	=> 2 1	
8 7 6 5		
4 3 2 1		

The above input/output is presented just to explain the slicing concept. The internal storage representation of these sliced blockcyclic vectors would be a bit different and complex in nature.

## 15.4 SEE ALSO

blockcyclic\_matrix, sliced\_blockcyclic\_matrix





# Chapter 16

## pblas\_\_wrapper

### 16.1 NAME

pblas\_\_wrapper - a frovedis module provides user-friendly interfaces for commonly used pblas routines in scientific applications like machine learning algorithms.

### 16.2 SYNOPSIS

```
#include <frovedis/matrix/pblas_wrapper.hpp>
```

### 16.3 WRAPPER FUNCTIONS

```
void swap (const sliced_blockcyclic_vector<T>& v1,
           const sliced_blockcyclic_vector<T>& v2)
void copy (const sliced_blockcyclic_vector<T>& v1,
           const sliced_blockcyclic_vector<T>& v2)
void scal (const sliced_blockcyclic_vector<T>& v,
           T al)
void axpy (const sliced_blockcyclic_vector<T>& v1,
           const sliced_blockcyclic_vector<T>& v2,
           T al = 1.0)
T dot (const sliced_blockcyclic_vector<T>& v1,
       const sliced_blockcyclic_vector<T>& v2)
T nrm2 (const sliced_blockcyclic_vector<T>& v)
void gemv (const sliced_blockcyclic_matrix<T>& m,
           const sliced_blockcyclic_vector<T>& v1,
           const sliced_blockcyclic_vector<T>& v2,
           char trans = 'N',
           T al = 1.0,
           T be = 0.0)
void ger (const sliced_blockcyclic_vector<T>& v1,
          const sliced_blockcyclic_vector<T>& v2,
          const sliced_blockcyclic_matrix<T>& m,
          T al = 1.0)
void gemm (const sliced_blockcyclic_matrix<T>& m1,
```

```

    const sliced_blockcyclic_matrix<T>& m2,
    const sliced_blockcyclic_matrix<T>& m3,
    char trans_m1 = 'N',
    char trans_m2 = 'N',
    T al = 1.0,
    T be = 0.0)
void geadd (const sliced_blockcyclic_matrix<T>& m1,
    const sliced_blockcyclic_matrix<T>& m2,
    char trans = 'N',
    T al = 1.0,
    T be = 1.0)

```

## 16.4 OVERLOADED OPERATORS

```

blockcyclic_matrix<T>
operator* (const sliced_blockcyclic_matrix<T>& m1,
    const sliced_blockcyclic_matrix<T>& m2)
blockcyclic_matrix<T>
operator+ (const sliced_blockcyclic_matrix<T>& m1,
    const sliced_blockcyclic_matrix<T>& m2)
blockcyclic_matrix<T>
operator~ (const sliced_blockcyclic_matrix<T>& m1)

```

## 16.5 DESCRIPTION

PBLAS is a high-performance external library written in Fortran language. It provides rich set of functionalities on vectors and matrices. The computation loads of these functionalities are parallelized over the available processes in a system and the user interfaces of this library is very detailed and complex in nature. It requires a strong understanding on each of the input parameters, along with some distribution concepts.

Frovedis provides a wrapper module for some commonly used PBLAS subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed distribution parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a PBLAS routine can also be performed using Frovedis PBLAS wrapper of that routine.

These wrapper routines are global functions in nature. Thus they can be called easily from within the “frovedis” namespace. As a distributed input vector, they accept “`blockcyclic_matrix<T>`” with single column” or “`sliced_blockcyclic_vector<T>`”. And as a distributed input matrix, they accept “`blockcyclic_matrix<T>`” or “`sliced_blockcyclic_matrix<T>`”. “T” is a template type which can be either “float” or “double”. The individual detailed descriptions can be found in the subsequent sections. Please note that the term “inout”, used in the below section indicates a function argument as both “input” and “output”.

### 16.5.1 Detailed Description

#### 16.5.1.1 swap (v1, v2)

##### Parameters

*v1*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (inout)  
*v2*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (inout)

**Purpose**

It will swap the contents of `v1` and `v2`, if they are semantically valid and are of same length.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.2 copy (v1, v2)****Parameters**

`v1`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

`v2`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (output)

**Purpose**

It will copy the contents of `v1` in `v2` ( $v2 = v1$ ), if they are semantically valid and are of same length.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.3 scal (v, al)****Parameters**

`v`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (inout)

`al`: A “T” typed argument (float or double) to specify the value to which the input vector needs to be scaled. (input)

**Purpose**

It will scale the input vector with the provided “al” value, if it is semantically valid. On success, input vector “v” would be updated (in-place scaling).

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.4 axpy (v1, v2, al=1.0)****Parameters**

`v1`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

`v2`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (inout)

`al`: A “T” typed argument (float or double) to specify the value to which “v1” needs to be scaled (not in-place scaling) [Default: 1.0] (input/optional)

**Purpose**

It will solve the expression  $v2 = al * v1 + v2$ , if the input vectors are semantically valid and are of same length. On success, “v2” will be updated with desired result, but “v1” would remain unchanged.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.5 dot (v1, v2)****Parameters**

`v1`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

`v2`: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

**Purpose**

It will perform dot product of the input vectors, if they are semantically valid and are of same length. Input vectors would not get modified during the operation.

**Return Value**

On success, it returns the dot product result of the type “float” or “double”. If any error occurs, it throws an exception.

**16.5.1.6 nrm2 (v)****Parameters**

*v*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

**Purpose**

It will calculate the norm of the input vector, if it is semantically valid. Input vector would not get modified during the operation.

**Return Value**

On success, it returns the norm value of the type “float” or “double”. If any error occurs, it throws an exception.

**16.5.1.7 gemv (m, v1, v2, trans='N', al=1.0, be=0.0)****Parameters**

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*v1*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)

*v2*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input/output)

*trans*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)

*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

*be*: A “T” typed (float or double) scalar value [Default: 0.0] (input/optional)

**Purpose**

The primary aim of this routine is to perform simple matrix-vector multiplication. But it can also be used to perform any of the below operations:

$$(1) \ v2 = al*m*v1 + be*v2$$

$$(2) \ v2 = al*transpose(m)*v1 + be*v2$$

If *trans*=‘N’, then expression (1) is solved. In that case, the size of “v1” must be at least the number of columns in “m” and the size of “v2” must be at least the number of rows in “m”.

If *trans*=‘T’, then expression (2) is solved. In that case, the size of “v1” must be at least the number of rows in “m” and the size of “v2” must be at least the number of columns in “m”.

Since “v2” is used as input-output both, memory must be allocated for this vector before calling this routine, even if simple matrix-vector multiplication is required. Otherwise, this routine will throw an exception.

For simple matrix-vector multiplication, no need to specify values for the input parameters “trans”, “al” and “be” (leave them at their default values).

On success, “v2” will be overwritten with the desired output. But “m” and “v1” would remain unchanged.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.8 ger (v1, v2, m, al=1.0)****Parameters**

*v1*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)  
*v2*: A `blockcyclic_matrix<T>` with single column or a `sliced_blockcyclic_vector<T>` (input)  
*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)  
*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

**Purpose**

The primary aim of this routine is to perform simple vector-vector multiplication of the sizes “a” and “b” respectively to form an axb matrix. But it can also be used to perform the below operations:

```
m = al*v1*v2' + m
```

This operation can only be performed if the inputs are semantically valid and the size of “v1” is at least the number of rows in matrix “m” and the size of “v2” is at least the number of columns in matrix “m”.

Since “m” is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple vector-vector multiplication is required. Otherwise it will throw an exception.

For simple vector-vector multiplication, no need to specify the value for the input parameter “al” (leave it at its default value).

On success, “m” will be overwritten with the desired output. But “v1” and “v2” will remain unchanged.

**Return Value**

On success, it returns void. If any error occurs, it throws an exception.

**16.5.1.9 gemm (m1, m2, m3, trans\_m1='N', trans\_m2='N', al=1.0, be=0.0)****Parameters**

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)  
*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)  
*m3*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)  
*trans\_m1*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)  
*trans\_m2*: A character value can be either ‘N’ or ‘T’ [Default: ‘N’] (input/optional)  
*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)  
*be*: A “T” typed (float or double) scalar value [Default: 0.0] (input/optional)

**Purpose**

The primary aim of this routine is to perform simple matrix-matrix multiplication. But it can also be used to perform any of the below operations:

- (1)  $m3 = al*m1*m2 + be*m3$
- (2)  $m3 = al*transpose(m1)*m2 + be*m3$
- (3)  $m3 = al*m1*transpose(m2) + be*m3$
- (4)  $m3 = al*transpose(m1)*transpose(m2) + be*m3$

(1) will be performed, if both “trans\_m1” and “trans\_m2” are ‘N’.

(2) will be performed, if trans\_m1=‘T’ and trans\_m2 = ‘N’.

(3) will be performed, if trans\_m1=‘N’ and trans\_m2 = ‘T’.

(4) will be performed, if both “trans\_m1” and “trans\_m2” are ‘T’.

If we have four variables nrowa, nrowb, ncola, ncolb defined as follows:

```

if(trans_m1 == 'N') {
    nrowa = number of rows in m1
    ncola = number of columns in m1
}
else if(trans_m1 == 'T') {
    nrowa = number of columns in m1
    ncola = number of rows in m1
}

if(trans_m2 == 'N') {
    nrowb = number of rows in m2
    ncolb = number of columns in m2
}
else if(trans_m2 == 'T') {
    nrowb = number of columns in m2
    ncolb = number of rows in m2
}

```

Then this function can be executed successfully, if the below conditions are all true:

- (a) "ncola" is equal to "nrowb"
- (b) number of rows in "m3" is equal to or greater than "nrowa"
- (b) number of columns in "m3" is equal to or greater than "ncolb"

Since “m3” is used as input-output both, memory must be allocated for this matrix before calling this routine, even if simple matrix-matrix multiplication is required. Otherwise it will throw an exception.

For simple matrix-matrix multiplication, no need to specify the value for the input parameters “trans\_m1”, “trans\_m2”, “al”, “be” (leave them at their default values).

On success, “m3” will be overwritten with the desired output. But “m1” and “m2” will remain unchanged.

### Return Value

On success, it returns void. If any error occurs, it throws an exception.

#### 16.5.1.10 geadd (m1, m2, trans='N', al=1.0, be=1.0)

##### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)  
*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)  
*trans*: A character value can be either 'N' or 'T' [Default: 'N'] (input/optional)  
*al*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)  
*be*: A “T” typed (float or double) scalar value [Default: 1.0] (input/optional)

##### Purpose

The primary aim of this routine is to perform simple matrix-matrix addition. But it can also be used to perform any of the below operations:

- (1)  $m2 = al * m1 + be * m2$
- (2)  $m2 = al * \text{transpose}(m1) + be * m2$

If trans='N', then expression (1) is solved. In that case, the number of rows and the number of columns in “m1” should be equal to the number of rows and the number of columns in “m2” respectively.

If trans='T', then expression (2) is solved. In that case, the number of columns and the number of rows in “m1” should be equal to the number of rows and the number of columns in “m2” respectively.

If it is needed to scale the input matrices before the addition, corresponding “al” and “be” values can be provided. But for simple matrix-matrix addition, no need to specify values for the input parameters “trans”,

“al” and “be” (leave them at their default values).

On success, “m2” will be overwritten with the desired output. But “m1” would remain unchanged.

#### Return Value

On success, it returns void. If any error occurs, it throws an exception.

#####operator\* (m1, m2) **Parameters**

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

#### Purpose

This operator operates on two input matrices and returns the resultant matrix after successful multiplication. Both the input matrices remain unchanged.

#### Return Value

On success, it returns the resultant matrix of the type `blockcyclic_matrix<T>`. If any error occurs, it throws an exception.

#####operator+ (m1, m2) **Parameters**

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

#### Purpose

This operator operates on two input matrices and returns the resultant matrix after successful addition. Both the input matrices remain unchanged.

#### Return Value

On success, it returns the resultant matrix of the type `blockcyclic_matrix<T>`. If any error occurs, it throws an exception.

#####operator~ (m1)

#### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

#### Purpose

This operator operates on single input matrix and returns its transposed matrix. E.g., if “m” is a matrix of type `blockcyclic_matrix<T>`, then “~m” will return transposed of matrix “m” of the type `blockcyclic_matrix<T>`.

**Return Value** On success, it returns the resultant matrix of the type `blockcyclic_matrix<T>`. If any error occurs, it throws an exception.

## 16.6 SEE ALSO

`sliced_blockcyclic_matrix_local`, `sliced_blockcyclic_vector_local`, `blas_wrapper`





# Chapter 17

## scalapack\_\_wrapper

### 17.1 NAME

scalapack\_\_wrapper - a frovedis module provides user-friendly interfaces for commonly used scalapack routines in scientific applications like machine learning algorithms.

### 17.2 SYNOPSIS

```
#include <frovedis/matrix/scalapack_wrapper.hpp>
```

### 17.3 WRAPPER FUNCTIONS

```
int getrf (const sliced_blockcyclic_matrix<T>& m,  
          lvec<int>& ipiv)  
int getri (const sliced_blockcyclic_matrix<T>& m,  
          const lvec<int>& ipiv)  
int getsr (const sliced_blockcyclic_matrix<T>& m1,  
          const sliced_blockcyclic_matrix<T>& m2,  
          const lvec<int>& ipiv,  
          char trans = 'N')  
void lacpy (const sliced_blockcyclic_matrix<T>& m1,  
          const sliced_blockcyclic_matrix<T>& m2,  
          char uplo = 'A')  
int gesv (const sliced_blockcyclic_matrix<T>& m1,  
          const sliced_blockcyclic_matrix<T>& m2)  
int gesv (const sliced_blockcyclic_matrix<T>& m1,  
          const sliced_blockcyclic_matrix<T>& m2,  
          lvec<int>& ipiv)  
int gels (const sliced_blockcyclic_matrix<T>& m1,  
          const sliced_blockcyclic_matrix<T>& m2,  
          char trans = 'N')  
int gesvd (const sliced_blockcyclic_matrix<T>& m,  
          std::vector<T>& sval)  
int gesvd (const sliced_blockcyclic_matrix<T>& m,  
          std::vector<T>& sval,
```

```

        const sliced_blockcyclic_matrix<T>& svec,
        char vtype = 'L')
int gesvd (const sliced_blockcyclic_matrix<T>& m,
        std::vector<T>& sval,
        const sliced_blockcyclic_matrix<T>& lsvect,
        const sliced_blockcyclic_matrix<T>& rsvec)

```

## 17.4 SPECIAL FUNCTIONS

```
blockcyclic_matrix<T> inv (const sliced_blockcyclic_matrix<T>& m)
```

## 17.5 DESCRIPTION

ScaLAPACK is a high-performance external library written in Fortran language. It provides rich set of linear algebra functionalities whose computation loads are parallelized over the available processes in a system and the user interfaces of this library is very detailed and complex in nature. It requires a strong understanding on each of the input parameters, along with some distribution concepts.

Frovedis provides a wrapper module for some commonly used ScaLAPACK subroutines in scientific applications like machine learning algorithms. These wrapper interfaces are very simple and user needs not to consider all the detailed distribution parameters. Only specifying the target vectors or matrices with some other parameters (depending upon need) are fine. At the same time, all the use cases of a ScaLAPACK routine can also be performed using Frovedis ScaLAPACK wrapper of that routine.

These wrapper routines are global functions in nature. Thus they can be called easily from within the “frovedis” namespace. As a distributed input matrix, they accept “`blockcyclic_matrix<T>`” or “`sliced_blockcyclic_matrix<T>`”. “T” is a template type which can be either “float” or “double”. The individual detailed descriptions can be found in the subsequent sections. Please note that the term “inout”, used in the below section indicates a function argument as both “input” and “output”.

### 17.5.1 Detailed Description

#### 17.5.1.1 getrf (m, ipiv)

##### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*ipiv*: An empty object of the type `frovedis::lvec<int>` (output)

##### Purpose

It computes an LU factorization of a general M-by-N distributed matrix, “m” using partial pivoting with row interchanges.

On successful factorization, matrix “m” is overwritten with the computed L and U factors. Along with the input matrix, this function expects user to pass an empty object of the type “`frovedis::lvec<int>`” as a second argument, named as “ipiv” which would be updated with the pivoting information associated with input matrix “m” by this function while computing factors. This “ipiv” information will be useful in computation of some other functions (like `getri`, `getrs` etc.)

##### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

### 17.5.1.2 getri (m, ipiv)

#### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*ipiv*: An object of the type `frovedis::lvec<int>` (input)

#### Purpose

It computes the inverse of a distributed square matrix using the LU factorization computed by `getrf()`. So in order to compute inverse of a matrix, first compute it's LU factor (and *ipiv* information) using `getrf()` and then pass the factored matrix, “*m*” along with the “*ipiv*” information to this function.

On success, factored matrix “*m*” is overwritten with the inverse (of the matrix which was passed to `getrf()`) matrix. “*ipiv*” will be internally used by this function and will remain unchanged.

#### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

### 17.5.1.3 getsr (m1, m2, ipiv, trans='N')

#### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*ipiv*: An object of the type `frovedis::lvec<int>` (input)

*trans*: A character containing either 'N' or 'T' [Default: 'N'] (input/optional)

#### Purpose

It solves a real system of distributed linear equations,  $AX=B$  with a general distributed square matrix (*A*) using the LU factorization computed by `getrf()`. Thus before calling this function, it is required to obtain the factored matrix “*m1*” (along with “*ipiv*” information) by calling `getrf()`.

If *trans*='N', the linear equation  $AX=B$  is solved.

If *trans*='T' the linear equation  $\text{transpose}(A)X=B$  ( $A'X=B$ ) is solved.

The matrix “*m2*” should have number of rows  $\geq$  the number of rows in “*m1*” and at least 1 column in it.

On entry, “*m2*” contains the distributed right-hand-side (*B*) of the equation and on successful exit it is overwritten with the distributed solution matrix (*X*).

#### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

### 17.5.1.4 lacpy (m1, m2, uplo='A')

#### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (output)

*uplo*: A character containing either 'U', 'L' or 'A' [Default: 'A'] (input/optional)

#### Purpose

It copies a distributed M-by-N matrix, “*m1*” in another distributed M-by-N matrix, “*m2*” ( $m2=m1$ ). No communication is performed during this copy. Only local versions are copied in each other.

If *uplo*='U', only upper-triangular part of “*m1*” will be copied in upper-triangular part of “*m2*”.

If *uplo*='L', only lower-triangular part of “*m1*” will be copied in lower-triangular part of “*m2*”.

And if *uplo*='A', all part of “*m2*” will be copied in “*m2*”.

This function expects a valid M-by-N distributed matrix “m2” to be passed as second argument which will be updated with the copy of “m2” on successful exit. Thus a user is needed to allocate the memory for “m2” and pass to this function before calling it. If dimension of “m2” is not matched with dimension of “m1” or “m2” is not allocated beforehand, this function will throw an exception.

### Return Value

On success, it returns void. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.5 gesv (m1, m2)

##### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

##### Purpose

It solves a real system of distributed linear equations,  $AX=B$  with a general distributed square matrix, “m1” by computing its LU factors internally. This function internally computes the LU factors and ipiv information using `getrf()` and then solves the equation using `getrs()`.

The matrix “m2” should have number of rows  $\geq$  the number of rows in “m1” and at least 1 column in it.

On entry, “m1” contains the distributed left-hand-side square matrix (A), “m2” contains the distributed right-hand-side matrix (B) and on successful exit “m1” is overwritten with its LU factors, “m2” is overwritten with the distributed solution matrix (X).

### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.6 gesv (m1, m2, ipiv)

##### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*ipiv*: An empty object of the type `frovedis::lvec<int>` (output)

##### Purpose

The function serves the same purpose as explained in above version of `gesv` (with two parameters). Only difference is that this version accepts an extra parameter “ipiv” of the type `lvec<int>` which would be allocated and updated with the pivoting information computed during factorization of “m1”. Along with the factored matrix, it might also be needed to know the associated pivot values. In that case, this version of `gesv` (with three parameters) can be used.

On entry, “m1” contains the distributed left-hand-side square matrix (A), “m2” contains the distributed right-hand-side matrix (B), and “ipiv” is an empty object. On successful exit “m1” is overwritten with its LU factors, “m2” is overwritten with the distributed solution matrix (X), and “ipiv” is updated with the pivot values associated with factored matrix, “m1”.

### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.7 gels (m1, m2, trans=‘N’)

##### Parameters

*m1*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*m2*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*trans*: A character containing either 'N' or 'T' [Default: 'N'] (input/optional)

### Purpose

It solves overdetermined or underdetermined real linear systems involving an M-by-N distributed matrix (A) or its transpose, using a QR or LQ factorization of (A). It is assumed that distributed matrix (A) has full rank.

If *trans*='N' and  $M \geq N$ : it finds the least squares solution of an overdetermined system.

If *trans*='N' and  $M < N$ : it finds the minimum norm solution of an underdetermined system.

If *trans*='T' and  $M \geq N$ : it finds the minimum norm solution of an underdetermined system.

If *trans*='T' and  $M < N$ : it finds the least squares solution of an overdetermined system.

The matrix "m2" should have number of rows  $\geq \max(M, N)$  and at least 1 column.

On entry, "m1" contains the distributed left-hand-side matrix (A) and "m2" contains the distributed right-hand-side matrix (B). On successful exit, "m1" is overwritten with the QR or LQ factors and "m2" is overwritten with the distributed solution matrix (X).

### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.8 gesvd (m, sval)

##### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

##### Purpose

It computes the singular value decomposition (SVD) of an M-by-N distributed matrix.

On entry "m" contains the distributed matrix whose singular values are to be computed, "sval" is an empty object of the type `std::vector<T>`. And on successful exit, the contents of "m" is destroyed (internally used as workspace) and "sval" is updated with the singular values in sorted order, so that  $sval(i) \geq sval(i+1)$ .

##### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.9 gesvd (m, sval, svec, vtype)

##### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (inout)

*sval*: An empty vector of the type `std::vector<T>` (output)

*svec*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (output)

*vtype*: A character value containing either 'L' or 'R' [Default: 'L'] (input/optional)

##### Purpose

It computes the singular value decomposition (SVD) of an M-by-N distributed matrix. Additionally, it also computes *left or right singular vectors*.

If *vtype*='L', "svec" will be updated with first  $\min(M, N)$  columns of left singular vectors (stored columnwise). In that case "svec" should have at least M number of rows and  $\min(M, N)$  number of columns.

If *vtype*='R', "svec" will be updated with first  $\min(M, N)$  rows of right singular vectors (stored rowwise in transposed form). In that case "svec" should have at least  $\min(M, N)$  number of rows and N number of columns.

This function expects that required memory would be allocated for the output matrix “svec” beforehand. If it is not allocated, an exception will be thrown.

On entry “m” contains the distributed matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “svec” is a valid sized (as explained above) distributed matrix.

And on successful exit, the contents of “m” is destroyed (internally used as workspace), “sval” is updated with the singular values in sorted order, so that  $\text{sval}(i) \geq \text{sval}(i+1)$  and “svec” is updated with the desired singular vectors (as explained above).

### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.10 gesvd (m, sval, lsvec, rsvec)

##### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

*sval*: An empty vector of the type `std::vector<T>` (output)

*lsvec*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (output)

*rsvec*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (output)

##### Purpose

It computes the singular value decomposition (SVD) of an M-by-N distributed matrix. Additionally, it also computes *left and right singular vectors*.

This function expects that required memory would be allocated for the output matrices “lsvec” and “rsvec” beforehand, to store the left and right singular vectors respectively. If they are not allocated, an exception will be thrown.

Output matrix “lsvec” will be updated with first  $\min(M,N)$  columns of left singular vectors (stored columnwise). Thus, “lsvec” should have at least M number of rows and  $\min(M,N)$  number of columns.

Output matrix “rsvec” will be updated with first  $\min(M,N)$  rows of right singular vectors (stored rowwise in transposed form). Thus, “rsvec” should have at least  $\min(M,N)$  number of rows and N number of columns.

On entry “m” contains the distributed matrix whose singular values are to be computed, “sval” is an empty object of the type `std::vector<T>`, “lsvec” and “rsvec” are valid sized (as explained above) distributed matrices. And on successful exit, the contents of “m” is destroyed (internally used as workspace), “sval” is updated with the singular values in sorted order, so that  $\text{sval}(i) \geq \text{sval}(i+1)$ , “lsvec” and “rsvec” are updated with the left and right singular vectors respectively (as explained above).

### Return Value

On success, it returns the exit status of the scalapack routine itself. If any error occurs, it throws an exception explaining cause of the error.

#### 17.5.1.11 inv (m)

##### Parameters

*m*: A `blockcyclic_matrix<T>` or a `sliced_blockcyclic_matrix<T>` (input)

##### Purpose

It computes the inverse of a distributed square matrix “m” by using `getrf()` and `getri()` internally. Thus it is a kind of short-cut function to obtain the inverse of a distributed matrix.

On successful exit, it returns the resultant inversed matrix. The input matrix “m” remains unchanged. Since it returns the resultant matrix, it can be used in any numerical expressions, along with other operators. E.g., if a and b are two blockcyclic matrices, then the expression like, “ $a*(\sim b)*\text{inv}(a)$ ” can easily be performed.

**Return Value**

On success, it returns the resultant matrix of the type `blockcyclic_matrix<T>`. If any error occurs, it throws an exception explaining cause of the error.

**17.6 SEE ALSO**

`sliced_blockcyclic_matrix_local`, `sliced_blockcyclic_vector_local`, `lapack_wrapper`





## Chapter 18

# frovedis::crs\_matrix\_local<T,I,O>

### 18.1 NAME

`frovedis::crs_matrix_local<T,I,O>` - A two-dimensional non-distributed sparse matrix with compressed row storage.

### 18.2 SYNOPSIS

```
#include <frovedis/matrix/crs_matrix.hpp>
```

#### 18.2.1 Constructors

```
crs_matrix_local ();  
crs_matrix_local (size_t nrow, size_t ncol);  
crs_matrix_local (const crs_matrix_local<T,I,O>& m);  
crs_matrix_local (crs_matrix_local<T,I,O>&& m);
```

#### 18.2.2 Overloaded Operators

```
crs_matrix_local<T,I,O>& operator= (const crs_matrix_local<T,I,O>& m);  
crs_matrix_local<T,I,O>& operator= (crs_matrix_local<T,I,O>&& m);
```

#### 18.2.3 Public Member Functions

```
void set_local_num (size_t ncol);  
void savebinary (const std::string& dir);  
void debug_print ();  
void debug_pretty_print ();  
crs_matrix_local<T,I,O> transpose () const;  
sparse_vector<T,I> get_row(size_t r);
```

## 18.2.4 Public Data Members

```
std::vector<T> val;
std::vector<I> idx;
std::vector<O> off;
size_t local_num_row;
size_t local_num_col;
```

## 18.3 DESCRIPTION

A CRS (Compressed Row Storage) matrix is one of the most popular sparse matrices. It has three major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

val: a vector containing the non-zero elements of the matrix (in row-major order).  
 idx: a vector containing the column indices for each non-zero elements.  
 off: a vector containing the row-offsets.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

then its CRS representation would be:

```
val: {1, 2, 4, 1, 2, 3, 1, 2, 4, 1, 2, 3}
idx: {0, 4, 7, 3, 4, 7, 0, 4, 7, 3, 4, 7}
off: {0, 3, 6, 9, 12}
```

row-offset starts with 0 and it has n+1 number of elements, where n is the number of rows in the sparse matrix. The difference between i+1th element and ith element in row-offset indicates number of non-zero elements present in ith row.

crs\_matrix\_local<T,I,O> is a two-dimensional template based non-distributed sparse data storage supported by frovedis. The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t>
struct crs_matrix_local {
    std::vector<T> val;        // to contain non-zero elements of type "T"
    std::vector<I> idx;        // to contain column indices of type "I" (default: size_t)
    std::vector<O> off;        // to contain row-offsets of type "O" (default: size_t)
    size_t local_num_row;      // number of rows in the sparse matrix
    size_t local_num_col;      // number of columns in the sparse matrix
};
```

### 18.3.1 Constructor Documentation

#### 18.3.1.1 crs\_matrix\_local ()

This is the default constructor which creates an empty crs matrix with local\_num\_row = local\_num\_col = 0.

**18.3.1.2 crs\_matrix\_local (size\_t nrow, size\_t ncol)**

This is the parameterized constructor which creates an empty crs matrix of the given dimension without any memory allocation for the matrix elements.

**18.3.1.3 crs\_matrix\_local (const crs\_matrix\_local<T,I,0>& m)**

This is the copy constructor which creates a new crs matrix by deep-copying the contents of the input crs matrix.

**18.3.1.4 crs\_matrix\_local (crs\_matrix\_local<T,I,0>&& m)**

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**18.3.2 Overloaded Operator Documentation****18.3.2.1 crs\_matrix\_local<T,I,0>& operator= (const crs\_matrix\_local<T,I,0>& m)**

It deep-copies the input crs matrix into the left-hand side matrix of the assignment operator “=”.

**18.3.2.2 crs\_matrix\_local<T,I,0>& operator= (crs\_matrix\_local<T,I,0>&& m)**

Instead of copying, it moves the contents of the input rvalue crs matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**18.3.3 Public Member Function Documentation****18.3.3.1 sparse\_vector<T,I> get\_row(size\_t r)**

It returns the requested row of the target sparse matrix in the form of `sparse_vector<T,I>` which contains a vector of type “T” for the non-zero elements in the requested row and a vector of type “I” for their corresponding column indices. If `r > local_num_row`, then it will throw an exception.

**18.3.3.2 void set\_local\_num (size\_t ncol)**

It sets the matrix information related to number of rows and number of columns as specified by the user. It assumes the user will provide the valid information related to the number of columns. Number of rows value is set as `off.size()-1`.

**18.3.3.3 void debug\_print ()**

It prints the information related to the compressed row storage (val, idx, off, number of rows and number of columns) on the user terminal. It is mainly useful for debugging purpose.

**18.3.3.4 void debug\_pretty\_print ()**

Unlike debug\_print(), it prints the compressed row storage as a view of a two dimensional dense storage on the user terminal. It is mainly useful for debugging purpose.

**18.3.3.5 crs\_matrix\_local<T,I,O> transpose ()**

It returns the transposed crs\_matrix\_local of the source matrix object.

**18.3.3.6 void savebinary (const std::string& dir)**

It writes the elements of a crs matrix to the specified directory as little-endian binary data.

The output directory will contain four files, named “nums”, “val”, “idx” and “off”. “nums” is a text file containing the number of rows and number of columns information in first two lines of the file. And rest three files contain the binary data related to compressed row storage.

**18.3.4 Public Data Member Documentation****18.3.4.1 val**

An instance of `std::vector<T>` type to contain the non-zero elements of the sparse matrix.

**18.3.4.2 idx**

An instance of `std::vector<I>` type to contain the column indices of the non-zero elements of the sparse matrix.

**18.3.4.3 off**

An instance of `std::vector<O>` type to contain the row offsets.

**18.3.4.4 local\_num\_row**

A `size_t` attribute to contain the number of rows in the 2D matrix view.

**18.3.4.5 local\_num\_col**

A `size_t` attribute to contain the number of columns in the 2D matrix view.

**18.3.5 Public Global Function Documentation****18.3.5.1 crs\_matrix\_local<T,I,O> make\_crs\_matrix\_local\_load(filename)****Parameters**

*filename*: A string object containing the name of the text file having the data to be loaded.

**Purpose**

This function loads the text data from the specified file and creates a `crs_matrix_local<T,I,0>` object filling the data loaded.

The input file for the sparse data should be in the below format:

```
1:2 3:2
2:5
1:3 3:4 6:3
3:2 4:5
```

Where each sparse row is represented as “column\_index:value” (column\_index starts at 0). Note that there can be empty rows in the given file indicating no non-zero elements in that row. The desired type triplet of the matrix `<T,I,0>` needs to be explicitly specified when loading the matrix data from reading a file.

Default types for “I” and “O” is “size\_t”. But “T” type must be mandatorily specified. While loading the matrix data, it will consider number of columns as the maximum value of the column index read.

For example, considering “./data” is a text file having the sparse data to be loaded, then

```
auto m1 = make_crs_matrix_local_load<int>("./data");
auto m2 = make_crs_matrix_local_load<float>("./data");
```

“m1” will be a `crs_matrix_local<int,size_t,size_t>`, whereas

“m2” will be a `crs_matrix_local<float,size_t,size_t>`.

**Return Value**

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

**18.3.5.2 `crs_matrix_local<T,I,0> make_crs_matrix_local_load(filename, num_col)`****Parameters**

*filename*: A string object containing the name of the text file having the data to be loaded.

*num\_col*: A `size_t` attribute specifying the number of columns in the sparse matrix to be loaded.

**Purpose**

This function serves the same purpose as explained in above data loading function. But since it also accepts the number of columns information, it sets the loaded matrix column number with the given value (without computing the maximum column index as in previous case). Thus it expects, user will pass a valid column number for the loaded sparse matrix.

**Return Value**

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

**18.3.5.3 `crs_matrix_local<T,I,0> make_crs_matrix_local_loadbinary(dirname)`****Parameters**

*dirname*: A string object containing the name of the directory having the data to be loaded. It expects four files to be presented inside the specified directory, as follows:

- “nums” (containing number of rows and number of columns separated with new-line),
- “val” (containing binary data for non-zero elements),
- “idx” (containing binary column indices) and

- “off” (containing binary offset values)

### Purpose

This function loads the little-endian binary data from the specified directory and creates a `crs_matrix_local<T,I,0>` object filling the data loaded. The desired value type, “T” (e.g., int, float, double etc.) must be specified explicitly when loading the matrix data. If not specified, the other two types “I” and “O” would be `size_t` as default types.

For example, considering “./bin” is a directory having the binary data to be loaded,

```
auto m1 = make_crs_matrix_local_loadbinary<int>("./bin");
auto m2 = make_crs_matrix_local_loadbinary<float>("./bin");
```

“m1” will be a `crs_matrix_local<int,size_t,size_t>`, whereas

“m2” will be a `crs_matrix_local<float,size_t,size_t>`.

### Return Value

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

#### 18.3.5.4 `crs_matrix_local<T,I,0> make_crs_matrix_local_loadcoo(file,zero_origin)`

### Parameters

*file*: A string object containing the name of the file having the COO data to be loaded.

*zero\_origin*: A boolean attribute to indicate whether to consider 0-based indices while loading the COO data from file.

### Purpose

This function loads the text data from the specified file and creates a `crs_matrix_local<T,I,0>` object filling the data loaded.

The input file for the sparse data should be in the below COO format:

```
1 1 2.0
1 3 2.0
2 2 5.0
3 1 3.0
3 3 4.0
3 6 3.0
4 3 2.0
4 4 5.0
```

Where each row in the given file represents a triplet like `<row-index col-index value>`. The indices are 1-based by default. This file can be loaded as 0-based index, if “zero\_origin” parameter is passed as “true” while loading the file. The desired triplet type of the matrix `<T,I,0>` needs to be explicitly specified when loading the matrix data from reading a file.

Default types for “I” and “O” is “size\_t”. But “T” type must be mandatorily specified. While loading the matrix data, it will consider number of columns as the maximum value of the column index read.

For example, considering “./data” is a text file having the COO data to be loaded, then

```
auto m1 = make_crs_matrix_local_loadcoo<int>("./data");
auto m2 = make_crs_matrix_local_loadcoo<float>("./data");
```

“m1” will be a `crs_matrix_local<int,size_t,size_t>`, whereas

“m2” will be a `crs_matrix_local<float,size_t,size_t>`.

**Return Value**

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

**18.3.5.5 `std::ostream& operator<<(str, mat)`****Parameters**

*str*: A `std::ostream&` object representing the output stream buffer.

*mat*: An object of the type `crs_matrix_local<T,I,0>` containing the matrix to be handled.

**Purpose**

This function writes the contents of the sparse matrix in “index:value” format in the given output stream. Thus a crs matrix can simply be printed on the user terminal as “`std::cout << mat`”, where “mat” is the input matrix.

**Return Value**

On success, it returns a reference to the output stream.

**18.3.5.6 `std::vector<T> operator*(m,v)`****Parameters**

*m*: A `const&` object of the type `crs_matrix_local<T,I,0>`.

*v*: A `const&` object of the type `std::vector<T>`.

**Purpose**

This function performs matrix-vector multiplication between a sparse crs matrix object with a `std::vector` of same value (T) type. It expects the size of the input vector should be greater than or equal to the number of columns in the input crs matrix.

**Return Value**

On success, it returns the resultant vector of the type `std::vector<T>`. Otherwise, it throws an exception.

**18.3.5.7 `rowmajor_matrix_local<T> operator*(m1,m2)`****Parameters**

*m1*: A `const&` object of the type `crs_matrix_local<T,I,0>`.

*m2*: A `const&` object of the type `rowmajor_matrix_local<T>`.

**Purpose**

It performs matrix-matrix multiplication in between a sparse crs matrix and a dense rowmajor matrix of the same value (T) type.

**Return Value**

On success, it returns the resultant rowmajor matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.

**18.4 SEE ALSO**

`rowmajor_matrix_local`, `crs_matrix`





## Chapter 19

# frovedis::crs\_matrix<T,I,O>

### 19.1 NAME

`frovedis::crs_matrix<T,I,O>` - A two-dimensional row-wise distributed sparse matrix with compressed row storage.

### 19.2 SYNOPSIS

```
#include <frovedis/matrix/crs_matrix.hpp>
```

#### 19.2.1 Constructors

```
crs_matrix ();  
crs_matrix (frovedis::node_local<crs_matrix_local<T,I,O>>&& d);
```

#### 19.2.2 Public Member Functions

```
void save (const std::string& file);  
void savebinary (const std::string& dir);  
void debug_print ();  
void debug_pretty_print ();  
crs_matrix<T,I,O> transpose ();  
sparse_vector<T,I> get_row(size_t r);  
void clear();
```

#### 19.2.3 Public Data Members

```
frovedis::node_local<crs_matrix_local<T,I,O>> data;  
size_t num_row;  
size_t num_col;
```

## 19.3 DESCRIPTION

A CRS (Compressed Row Storage) matrix is one of the most popular sparse matrices. It has three major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

val: a vector containing the non-zero elements of the matrix (in row-major order).  
 idx: a vector containing the column indices for each non-zero elements.  
 off: a vector containing the row-offsets.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

then its CRS representation would be:

```
val: {1, 2, 4, 1, 2, 3, 1, 2, 4, 1, 2, 3}
idx: {0, 4, 7, 3, 4, 7, 0, 4, 7, 3, 4, 7}
off: {0, 3, 6, 9, 12}
```

row-offset starts with 0 and it has n+1 number of elements, where n is the number of rows in the sparse matrix. The difference between i+1th element and ith element in row-offset indicates number of non-zero elements present in ith row.

crs\_matrix<T,I,O> is a two-dimensional template based distributed sparse data storage supported by frovedis. It contains public member “data” of the type node\_local<crs\_matrix\_local<T,I,O>>. The actual distributed matrices are contained in all the worker nodes locally, thus named as crs\_matrix\_local<T,I,O> (see manual of crs\_matrix\_local) and “data” is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original sparse matrix. The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t>
struct crs_matrix {
    frovedis::node_local<crs_matrix_local<T,I,O>> data;    // local matrix information
    size_t num_row;    // number of rows in the global sparse matrix
    size_t num_col;    // number of columns in the global sparse matrix
};
```

For example, if the above sparse matrix with 4 rows and 8 columns is distributed row-wise over two worker nodes, then the distribution can be shown as:

master	worker0	worker1
-----	-----	-----
crs_matrix<int,size_t,size_t>	-> crs_matrix_local<int, size_t,size_t>	-> crs_matrix_local<int, size_t,size_t>
*data: node_local< crs_matrix _local<int, size_t,size_t>>	val: vector<int> ({1,2,4,1,2,3}) idx: vector<size_t> ({0,4,7,3,4,7}) off: vector<size_t> ({0,3,6})	val: vector<int> ({1,2,4,1,2,3}) idx: vector<size_t> ({0,4,7,3,4,7}) off: vector<size_t> ({0,3,6})
num_row: size_t (4)	local_num_row: size_t (2)	local_num_row: size_t (2)
num_col: size_t (8)	local_num_col: size_t (8)	local_num_col: size_t (8)

The node\_local<crs\_matrix\_local<int,size\_t,size\_t>> object “data” is simply a (\*)handle of the (->)local matrices at worker nodes.

## 19.3.1 Constructor Documentation

### 19.3.1.1 `crs_matrix ()`

This is the default constructor which creates an empty distributed crs matrix without any memory allocation at worker nodes.

### 19.3.1.2 `crs_matrix (frovedis::node_local<crs_matrix_local<T,I,0>>&& data)`

This is the parameterized constructor which accepts an rvalue of the type `node_local<crs_matrix_local<T,I,0>>` and *moves* the contents to the created distributed crs matrix.

In general, this constructor is used internally by some other functions. But user may need this constructor while constructing their own distributed crs matrix using the return value of some function (returning a `crs_matrix_local<T,I,0>`) called using “`frovedis::node_local::map`”.  
(thus returned value would be an object of type `node_local<crs_matrix_local<T,I,0>`)

## 19.3.2 Public Member Function Documentation

### 19.3.2.1 `sparse_vector<T,I> get_row(size_t r)`

It returns the requested row of the target sparse matrix in the form of `sparse_vector<T,I>` which contains a vector of type “T” for the non-zero elements in the requested row and a vector of type “I” for their corresponding column indices. If `r > local_num_row`, then it will throw an exception.

### 19.3.2.2 `void debug_print ()`

It prints the information related to the distributed compressed row storage (val, idx, off, number of rows and number of columns) on the user terminal node-by-node. It is mainly useful for debugging purpose.

### 19.3.2.3 `void debug_pretty_print ()`

Unlike `debug_print()`, it prints the distributed compressed row storage as a view of a two dimensional dense storage on the user terminal node-by-node. It is mainly useful for debugging purpose.

### 19.3.2.4 `crs_matrix<T,I,0> transpose ()`

It returns the transposed `crs_matrix` of the source matrix object.

### 19.3.2.5 `void save (const std::string& file)`

It writes the elements of a distributed crs matrix to the specified file as text data with the format “index:value” for each non-zero elements.

### 19.3.2.6 void savebinary (const std::string& dir)

It writes the elements of a distributed crs matrix to the specified directory as little-endian binary data.

The output directory will contain five files, named “nums”, “val”, “idx”, “off”, and “type”. “nums” is a text file containing the number of rows and number of columns information in first two lines of the file. And rest three files contain the binary data related to compressed row storage. The file “type” is a text file containing the type information of “val”, “idx”, and “off”.

### 19.3.2.7 void clear()

It clears the memory space for the allocated `crs_matrix_local<T,I,0>` per worker.

### 19.3.2.8 crs\_matrix<TT,II,00> change\_datatype()

This function can be used in order to change the triplet type of the target `crs_matrix` from `<T, I, 0>` to `<TT, II, 00>`, where these two type triplets must be compatible.

## 19.3.3 Public Data Member Documentation

### 19.3.3.1 data

An instance of `node_local<crs_matrix_local<T,I,0>>` type to contain the reference information related to local matrices at worker nodes.

### 19.3.3.2 num\_row

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

### 19.3.3.3 num\_col

A `size_t` attribute to contain the total number of columns in the 2D matrix view.

## 19.3.4 Public Global Function Documentation

### 19.3.4.1 crs\_matrix<T,I,0> make\_crs\_matrix\_load(filename)

#### Parameters

*filename*: A string object containing the name of the text file having the data to be loaded.

#### Purpose

This function loads the text data from the specified file and creates a `crs_matrix<T,I,0>` object filling the data loaded.

The input file for the sparse data should be in the below format:

```
1:2 3:2
2:5
1:3 3:4 6:3
3:2 4:5
```

Where each sparse row is represented as “column\_index:value” (column\_index starts at 0). Note that there can be empty rows in the given file indicating no non-zero elements in that row. The desired type triplet of the matrix  $\langle T, I, 0 \rangle$  needs to be explicitly specified when loading the matrix data from reading a file.

Default types for “I” and “O” is “size\_t”. But “T” type must be mandatorily specified. While loading the matrix data, it will consider number of columns as the maximum value of the column index read.

For example, considering “./data” is a text file having the sparse data to be loaded, then

```
auto m1 = make_crs_matrix_load<int>("./data");
auto m2 = make_crs_matrix_load<float>("./data");
```

“m1” will be a `crs_matrix<int,size_t,size_t>`, whereas “m2” will be a `crs_matrix<float,size_t,size_t>`.

### Return Value

On success, it returns the created matrix of the type `crs_matrix<T,I,0>`. Otherwise, it throws an exception.

#### 19.3.4.2 `crs_matrix<T,I,0> make_crs_matrix_load(filename, num_col)`

### Parameters

*filename*: A string object containing the name of the text file having the data to be loaded.

*num\_col*: A `size_t` attribute specifying the number of columns in the sparse matrix to be loaded.

### Purpose

This function serves the same purpose as explained in above data loading function. But since it also accepts the number of columns information, it sets the loaded matrix column number with the given value (without computing the maximum column index as in previous case). Thus it expects, user will pass a valid column number for the loaded sparse matrix.

### Return Value

On success, it returns the created matrix of the type `crs_matrix<T,I,0>`. Otherwise, it throws an exception.

#### 19.3.4.3 `crs_matrix<T,I,0> make_crs_matrix_loadbinary(dirname)`

### Parameters

*dirname*: A string object containing the name of the directory having the data to be loaded. It expects four files to be presented inside the specified directory, as follows:

- “nums” (containing number of rows and number of columns separated with new-line),
- “val” (containing binary data for non-zero elements),
- “idx” (containing binary column indices) and
- “off” (containing binary offset values)

### Purpose

This function loads the little-endian binary data from the specified directory and creates a `crs_matrix<T,I,0>` object filling the data loaded. The desired value type, “T” (e.g., int, float, double etc.) must be specified explicitly when loading the matrix data. If not specified, the other two types “I” and “O” would be `size_t` as default types.

For example, considering “./bin” is a directory having the binary data to be loaded,

```
auto m1 = make_crs_matrix_loadbinary<int>("./bin");
auto m2 = make_crs_matrix_loadbinary<float>("./bin");
```

“m1” will be a `crs_matrix<int,size_t,size_t>`, whereas “m2” will be a `crs_matrix<float,size_t,size_t>`.

### Return Value

On success, it returns the created matrix of the type `crs_matrix<T,I,0>`. Otherwise, it throws an exception.

#### 19.3.4.4 `crs_matrix<T,I,0> make_crs_matrix_loadcoo(file,zero_origin)`

##### Parameters

*file*: A string object containing the name of the file having the COO data to be loaded.

*zero\_origin*: A boolean attribute to indicate whether to consider 0-based indices while loading the COO data from file.

##### Purpose

This function loads the text data from the specified file and creates a `crs_matrix<T,I,0>` object filling the data loaded.

The input file for the sparse data should be in the below COO format:

```
1 1 2.0
1 3 2.0
2 2 5.0
3 1 3.0
3 3 4.0
3 6 3.0
4 3 2.0
4 4 5.0
```

Where each row in the given file represents a triplet like `<row-index col-index value>`. The indices are 1-based by default. This file can be loaded as 0-based index, if “zero\_origin” parameter is passed as “true” while loading the file. The desired triplet type of the matrix `<T,I,0>` needs to be explicitly specified when loading the matrix data from reading a file.

Default types for “I” and “O” is “size\_t”. But “T” type must be mandatorily specified. While loading the matrix data, it will consider number of columns as the maximum value of the column index read.

For example, considering “./data” is a text file having the COO data to be loaded, then

```
auto m1 = make_crs_matrix_loadcoo<int>("./data");
auto m2 = make_crs_matrix_loadcoo<float>("./data");
```

“m1” will be a `crs_matrix<int,size_t,size_t>`, whereas “m2” will be a `crs_matrix<float,size_t,size_t>`.

### Return Value

On success, it returns the created matrix of the type `crs_matrix<T,I,0>`. Otherwise, it throws an exception.

#### 19.3.4.5 `std::ostream& operator<<(str, mat)`

##### Parameters

*str*: A `std::ostream&` object representing the output stream buffer.

*mat*: An object of the type `crs_matrix<T,I,0>` containing the matrix to be handled.

##### Purpose

This function writes the contents of the sparse matrix in “index:value” format in the given output stream. Thus a crs matrix can simply be printed on the user terminal as “std::cout « mat”, where “mat” is the input matrix.

### Return Value

On success, it returns a reference to the output stream.

**19.3.4.6** `crs_matrix<T,I,0> make_crs_matrix_scatter (mat)`**Parameters**

*mat*: An object of the type `crs_matrix_local<T,I,0>` to be scattered among worker nodes.

**Purpose**

This function accepts a `crs_matrix_local<T,I,0>` object and scatters the same among participating worker nodes in order to create a `crs_matrix<T,I,0>`.

**Return Value**

On success, it returns the created matrix of the type `crs_matrix<T,I,0>`.

Otherwise, it throws an exception.

**19.4 SEE ALSO**

`crs_matrix_local`





## Chapter 20

# frovedis::ccs\_\_matrix\_\_local<T,I,O>

### 20.1 NAME

`frovedis::ccs_matrix_local<T,I,O>` - A two-dimensional non-distributed sparse matrix with compressed column storage.

### 20.2 SYNOPSIS

```
#include <frovedis/matrix/ccs_matrix.hpp>
```

#### 20.2.1 Constructors

```
ccs_matrix_local ();  
ccs_matrix_local (const ccs_matrix_local<T,I,O>& m);  
ccs_matrix_local (ccs_matrix_local<T,I,O>&& m);  
ccs_matrix_local (const crs_matrix_local<T,I,O>& m);
```

#### 20.2.2 Overloaded Operators

```
ccs_matrix_local<T,I,O>& operator= (const ccs_matrix_local<T,I,O>& m);  
ccs_matrix_local<T,I,O>& operator= (ccs_matrix_local<T,I,O>&& m);
```

#### 20.2.3 Public Member Functions

```
void set_local_num (size_t nrow);  
void debug_print ();  
ccs_matrix_local<T,I,O> transpose ();  
crs_matrix_local<T,I,O> to_crs();
```

#### 20.2.4 Public Data Members

```
std::vector<T> val;  
std::vector<I> idx;
```

```
std::vector<O> off;
size_t local_num_row;
size_t local_num_col;
```

## 20.3 DESCRIPTION

A CCS (Compressed Column Storage) matrix is one of the popular sparse matrices with compressed column. It has three major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

**val:** a vector containing the non-zero elements of the compressed columns (in column-major order) of the matrix.  
**idx:** a vector containing the row indices for each non-zero elements in "val".  
**off:** a vector containing the column offsets.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

then its CCS representation would be:

```
val: {1, 1, 1, 1, 2, 2, 2, 2, 4, 3, 4, 3}
idx: {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3}
off: {0, 2, 2, 2, 2, 4, 8, 8, 8, 12}
```

column offset starts with 0 and it has n+1 number of elements, where n is the number of columns in the sparse matrix. The difference between i+1th element and ith element in column offset indicates number of non-zero elements present in ith column.

`ccs_matrix_local<T,I,O>` is a two-dimensional template based non-distributed sparse data storage supported by frovedis. The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t>
struct ccs_matrix_local {
    std::vector<T> val;        // to contain non-zero elements of type "T"
    std::vector<I> idx;        // to contain row indices of type "I" (default: size_t)
    std::vector<O> off;        // to contain column offsets of type "O" (default: size_t)
    size_t local_num_row;      // number of rows in the sparse matrix
    size_t local_num_col;      // number of columns in the sparse matrix
};
```

This matrix can be loaded from a local crs matrix and also the matrix can be converted back to the local crs matrix. Thus loading/saving interfaces are not provided for local ccs matrix.

### 20.3.1 Constructor Documentation

#### 20.3.1.1 `ccs_matrix_local ()`

This is the default constructor which creates an empty ccs matrix with `local_num_row = local_num_col = 0`.

**20.3.1.2 `ccs_matrix_local (const ccs_matrix_local<T,I,0>& m)`**

This is the copy constructor which creates a new ccs matrix by deep-copying the contents of the input ccs matrix.

**20.3.1.3 `ccs_matrix_local (ccs_matrix_local<T,I,0>&& m)`**

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**20.3.1.4 `ccs_matrix_local (const crs_matrix_local<T,I,0>& m)`**

This is the implicit conversion constructor which creates a new ccs matrix by converting the input crs matrix of the same types.

**20.3.2 Overloaded Operator Documentation****20.3.2.1 `ccs_matrix_local<T,I,0>& operator= (const ccs_matrix_local<T,I,0>& m)`**

It deep-copies the input ccs matrix into the left-hand side matrix of the assignment operator “=”.

**20.3.2.2 `ccs_matrix_local<T,I,0>& operator= (ccs_matrix_local<T,I,0>&& m)`**

Instead of copying, it moves the contents of the input rvalue crs matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**20.3.3 Public Member Function Documentation****20.3.3.1 `void set_local_num (size_t nrow)`**

It sets the matrix information related to number of rows and number of columns as specified by the user. It assumes the user will provide the valid information related to the number of rows. Number of columns value is set as `off.size()-1`.

**20.3.3.2 `void debug_print ()`**

It prints the information related to the compressed column storage (`val`, `idx`, `off`, number of rows and number of columns) on the user terminal. It is mainly useful for debugging purpose.

**20.3.3.3 `ccs_matrix_local<T,I,0> transpose ()`**

It returns the transposed `ccs_matrix_local` of the source matrix object.

#### 20.3.3.4 `crs_matrix_local<T,I,O> to_crs ()`

It creates an equivalent crs matrix of the target ccs matrix of the same “val”, “idx” and “off” types. Target ccs matrix would remain unchanged.

### 20.3.4 Public Data Member Documentation

#### 20.3.4.1 `val`

An instance of `std::vector<T>` type to contain the non-zero elements of the compressed columns of the sparse matrix.

#### 20.3.4.2 `idx`

An instance of `std::vector<I>` type to contain the row indices of the non-zero elements of the compressed columns of the sparse matrix.

#### 20.3.4.3 `off`

An instance of `std::vector<O>` type to contain the column offsets.

#### 20.3.4.4 `local_num_row`

A `size_t` attribute to contain the number of rows in the 2D matrix view.

#### 20.3.4.5 `local_num_col`

A `size_t` attribute to contain the number of columns in the 2D matrix view.

### 20.3.5 Public Global Function Documentation

#### 20.3.5.1 `ccs_matrix_local<T,I,O> crs2ccs(m)`

##### Parameters

*m*: An object of the type `crs_matrix_local<T,I,O>`

##### Purpose

This function can be used to get a `ccs_matrix_local<T,I>` from a `crs_matrix_local<T,I,O>`. Input matrix would remain unchanged.

##### Return Value

On success, it returns the created matrix of the type `ccs_matrix_local<T,I>`. Otherwise, it throws an exception.

**20.3.5.2** `crs_matrix_local<T,I,0> ccs2crs(m)`**Parameters**

*m*: An object of the type `ccs_matrix_local<T,I,0>`

**Purpose**

This function can be used to get a `crs_matrix_local<T,I,0>` from a `ccs_matrix_local<T,I,0>`. Input matrix would remain unchanged.

**Return Value**

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

**20.3.5.3** `std::vector<T> operator*(m,v)`**Parameters**

*m*: An object of the type `ccs_matrix_local<T,I,0>`.

*v*: An object of the type `std::vector<T>`.

**Purpose**

This function performs matrix-vector multiplication between a sparse ccs matrix object with a `std::vector` of same value (T) type. It expects the size of the input vector should be greater than or equal to the number of columns in the input ccs matrix.

**Return Value**

On success, it returns the resultant vector of the type `std::vector<T>`. Otherwise, it throws an exception.

**20.3.5.4** `rowmajor_matrix_local<T> operator*(m1,m2)`**Parameters**

*m1*: An object of the type `ccs_matrix_local<T,I,0>`.

*m2*: An object of the type `rowmajor_matrix_local<T>`.

**Purpose**

It performs matrix-matrix multiplication in between a sparse ccs matrix and a dense rowmajor matrix of the same value (T) type.

**Return Value**

On success, it returns the resultant rowmajor matrix of the type `rowmajor_matrix_local<T>`. Otherwise, it throws an exception.

**20.4 SEE ALSO**

`crs_matrix_local`, `rowmajor_matrix_local`, `ccs_matrix`



## Chapter 21

# frovedis::ccs\_\_matrix<T,I,O>

### 21.1 NAME

`frovedis::ccs_matrix<T,I,O>` - A two-dimensional row-wise distributed sparse matrix with compressed column storage.

### 21.2 SYNOPSIS

```
#include <frovedis/matrix/ccs_matrix.hpp>
```

#### 21.2.1 Constructors

```
ccs_matrix ();  
ccs_matrix (const ccs_matrix<T,I,O>& m);
```

#### 21.2.2 Public Member Functions

```
void debug_print ();
```

#### 21.2.3 Public Data Members

```
frovedis::node_local<ccs_matrix_local<T,I,O>> data;  
size_t num_row;  
size_t num_col;
```

### 21.3 DESCRIPTION

A CCS (Compressed Column Storage) matrix is one of the popular sparse matrices. It has three major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

val: a vector containing the non-zero elements of the compressed columns (in column-major order) of the matrix.  
 idx: a vector containing the row indices for each non-zero elements in "val".  
 off: a vector containing the column offsets.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

then its CCS representation would be:

```
val: {1, 1, 1, 1, 2, 2, 2, 2, 4, 3, 4, 3}
idx: {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3}
off: {0, 2, 2, 2, 4, 8, 8, 8, 12}
```

column offset starts with 0 and it has n+1 number of elements, where n is the number of columns in the sparse matrix. The difference between i+1th element and ith element in column offset indicates number of non-zero elements present in ith column.

`ccs_matrix<T,I,O>` is a two-dimensional template based distributed sparse data storage supported by `frovedis`. It contains public member "data" of the type `node_local<ccs_matrix_local<T,I,O>>`. The actual distributed matrices are contained in all the worker nodes locally, thus named as `ccs_matrix_local<T,I,O>` (see manual of `ccs_matrix_local`) and "data" is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original sparse matrix. The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t>
struct ccs_matrix {
    frovedis::node_local<ccs_matrix_local<T,I,O>> data;    // local matrix information
    size_t num_row;    // number of rows in the global sparse matrix
    size_t num_col;    // number of columns in the global sparse matrix
};
```

For example, if the above sparse matrix with 4 rows and 8 columns is distributed row-wise over two worker nodes, then the distribution can be shown as:

master	worker0	worker1
-----	-----	-----
<code>ccs_matrix&lt;int,size_t,size_t&gt;</code>	<code>-&gt; ccs_matrix_local&lt;int,</code>	<code>-&gt; ccs_matrix_local&lt;int,</code>
	<code>size_t,size_t&gt;</code>	<code>size_t,size_t&gt;</code>
<code>*data: node_local&lt;</code>	<code>val: vector&lt;int&gt;</code>	<code>val: vector&lt;int&gt;</code>
<code>ccs_matrix</code>	<code>({1,1,2,2,4,3})</code>	<code>({1,1,2,2,4,3})</code>
<code>_local&lt;int,</code>	<code>idx: vector&lt;size_t&gt;</code>	<code>idx: vector&lt;size_t&gt;</code>
<code>size_t,size_t&gt;&gt;</code>	<code>({0,1,0,1,0,1})</code>	<code>({0,1,0,1,0,1})</code>
	<code>off: vector&lt;size_t&gt;</code>	<code>off: vector&lt;size_t&gt;</code>
	<code>({0,1,1,1,2,4,4,6})</code>	<code>({0,1,1,1,2,4,4,6})</code>
<code>num_row: size_t (4)</code>	<code>local_num_row: size_t (2)</code>	<code>local_num_row: size_t (2)</code>
<code>num_col: size_t (8)</code>	<code>local_num_col: size_t (8)</code>	<code>local_num_col: size_t (8)</code>

The `node_local<ccs_matrix_local<int,size_t,size_t>>` object "data" is simply a (\*)handle of the (->)local matrices at worker nodes.

This matrix can be loaded from a distributed crs matrix and also the matrix can be converted back to the distributed crs matrix. Thus loading/saving interfaces are not provided for distributed ccs matrix.



## 21.3.1 Constructor Documentation

### 21.3.1.1 `ccs_matrix ()`

This is the default constructor which creates an empty distributed ccs matrix without any memory allocation at worker nodes.

### 21.3.1.2 `ccs_matrix (crs_matrix<T,I,0>& m)`

This is the implicit conversion constructor to construct a distributed ccs matrix from the input distributed crs matrix of the same types.

## 21.3.2 Public Member Function Documentation

### 21.3.2.1 `void debug_print ()`

It prints the information related to the distributed compressed column storage (val, idx, off, number of rows and number of columns) on the user terminal node-by-node. It is mainly useful for debugging purpose.

## 21.3.3 Public Data Member Documentation

### 21.3.3.1 `data`

An instance of `node_local<ccs_matrix_local<T,I,0>>` type to contain the reference information related to local matrices at worker nodes.

### 21.3.3.2 `num_row`

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

### 21.3.3.3 `num_col`

A `size_t` attribute to contain the total number of columns in the 2D matrix view.

## 21.4 SEE ALSO

`ccs_matrix_local`, `crs_matrix`



## Chapter 22

# frovedis::ell\_matrix\_local<T,I>

### 22.1 NAME

`frovedis::ell_matrix_local<T,I>` - A two-dimensional non-distributed ELL sparse matrix.

### 22.2 SYNOPSIS

```
#include <frovedis/matrix/ell_matrix.hpp>
```

#### 22.2.1 Constructors

```
ell_matrix_local ();  
ell_matrix_local (const ell_matrix_local<T,I>& m);  
ell_matrix_local (ell_matrix_local<T,I>&& m);  
ell_matrix_local (const crs_matrix_local<T,I,0>& m);
```

#### 22.2.2 Overloaded Operators

```
ell_matrix_local<T,I>& operator= (const ell_matrix_local<T,I>& m);  
ell_matrix_local<T,I>& operator= (ell_matrix_local<T,I>&& m);
```

#### 22.2.3 Public Member Functions

```
void debug_print ();  
crs_matrix_local<T,I,0> to_crs();
```

#### 22.2.4 Public Data Members

```
std::vector<T> val;  
std::vector<I> idx;  
size_t local_num_row;  
size_t local_num_col;
```

## 22.3 DESCRIPTION

A ELL matrix is one of the most popular sparse matrices with elements stored in column-major order. In this matrix representation, all the non-zero elements of a row are shifted (packed) at left side and all the rows are padded with zeros on the right to give them equal length.

It has two major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

**val:** a vector containing the left-shifted (zero-padded) non-zero elements of the sparse matrix stored in column-major order.  
**idx:** a vector containing the corresponding column indices of the non-zero elements.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

Then its ELL image can be thought of as:

values		indices
1 2 4		0 4 7
1 2 3	=>	3 4 7
1 2 4		0 4 7
1 2 3		3 4 7

And its column-major memory representation would be:

**val:** {1, 1, 1, 1, 2, 2, 2, 2, 4, 3, 4, 3}  
**idx:** {0, 3, 0, 3, 4, 4, 4, 4, 7, 7, 7, 7}

`ell_matrix_local<T,I,0>` is a two-dimensional template based non-distributed sparse data storage supported by `frovedis`. The structure of this class is as follows:

```
template <class T, class I=size_t>
struct ell_matrix_local {
    std::vector<T> val;        // to contain non-zero elements of type "T"
    std::vector<I> idx;        // to contain column indices of type "I" (default: size_t)
    size_t local_num_row;     // number of rows in the sparse matrix
    size_t local_num_col;     // number of columns in the sparse matrix
};
```

This matrix can be loaded from a local crs matrix and also the matrix can be converted back to the local crs matrix. Thus loading/saving interfaces are not provided for local ell matrix.

### 22.3.1 Constructor Documentation

#### 22.3.1.1 `ell_matrix_local ()`

This is the default constructor which creates an empty ell matrix with `local_num_row = local_num_col = 0`.

#### 22.3.1.2 `ell_matrix_local (const ell_matrix_local<T,I>& m)`

This is the copy constructor which creates a new ell matrix by deep-copying the contents of the input ell matrix.

**22.3.1.3 ell\_matrix\_local (ell\_matrix\_local<T,I>&& m)**

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**22.3.1.4 ell\_matrix\_local (const crs\_matrix\_local<T,I,0>& m)**

This is the implicit conversion constructor to construct a local ell matrix from the input local crs matrix of same “val” and “idx” type.

**22.3.2 Overloaded Operator Documentation****22.3.2.1 ell\_matrix\_local<T,I>& operator= (const ell\_matrix\_local<T,I>& m)**

It deep-copies the input ell matrix into the left-hand side matrix of the assignment operator “=”.

**22.3.2.2 ell\_matrix\_local<T,I>& operator= (ell\_matrix\_local<T,I>&& m)**

Instead of copying, it moves the contents of the input rvalue ell matrix into the left-hand side matrix of the assignment operator “=”.

Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

**22.3.3 Public Member Function Documentation****22.3.3.1 crs\_matrix\_local<T,I,0> to\_crs()**

This method can be used to convert the target ell matrix into a local crs matrix of the same “val” and “idx” type.

**22.3.3.2 void debug\_print ()**

It prints the information related to the ELL storage (val, idx, number of rows and number of columns) on the user terminal. It is mainly useful for debugging purpose.

**22.3.4 Public Data Member Documentation****22.3.4.1 val**

An instance of `std::vector<T>` type to contain the non-zero elements of the ELL sparse matrix in column major order.

**22.3.4.2 idx**

An instance of `std::vector<I>` type to contain the column indices of the non-zero elements of the sparse matrix.

### 22.3.4.3 local\_num\_row

A `size_t` attribute to contain the number of rows in the 2D matrix view.

### 22.3.4.4 local\_num\_col

A `size_t` attribute to contain the number of columns in the 2D matrix view.

## 22.3.5 Public Global Function Documentation

### 22.3.5.1 ell\_matrix\_local<T,I> crs2ell(m)

#### Parameters

*m*: An object of the type `crs_matrix_local<T,I,0>`

#### Purpose

This function can be used to get a `ell_matrix_local<T,I>` from a `crs_matrix_local<T,I,0>`. Input matrix would remain unchanged.

#### Return Value

On success, it returns the created matrix of the type `ell_matrix_local<T,I>`. Otherwise, it throws an exception.

### 22.3.5.2 crs\_matrix\_local<T,I,0> ell2crs(m)

#### Parameters

*m*: An object of the type `ell_matrix_local<T,I>`

#### Purpose

This function can be used to get a `crs_matrix_local<T,I,0>` from a `ell_matrix_local<T,I>`. Input matrix would remain unchanged.

#### Return Value

On success, it returns the created matrix of the type `crs_matrix_local<T,I,0>`. Otherwise, it throws an exception.

### 22.3.5.3 std::vector<T> operator\*(m,v)

#### Parameters

*m*: A const& object of the type `ell_matrix_local<T,I>`

*v*: A const& object of the type `std::vector<T>`

#### Purpose

This function performs matrix-vector multiplication between a sparse ell matrix object with a `std::vector` of same value (T) type. It expects the size of the input vector should be greater than or equal to the number of columns in the input ell matrix.

#### Return Value

On success, it returns the resultant vector of the type `std::vector<T>`. Otherwise, it throws an exception.

**22.3.5.4** `std::vector<T> trans_mv(m,v)`**Parameters**

*m*: A const& object of the type `ell_matrix_local<T,I>`

*v*: A const& object of the type `std::vector<T>`

**Purpose**

This function performs transposed matrix-vector multiplication ( $m^T \cdot v$ ) between a sparse ell matrix object with a `std::vector` of same value (T) type. It expects the size of the input vector should be greater than or equal to the number of rows in the input ell matrix.

**Return Value**

On success, it returns the resultant vector of the type `std::vector<T>`. Otherwise, it throws an exception.

**22.4 SEE ALSO**

`crs_matrix_local`, `jds_matrix_local`, `ell_matrix`





## Chapter 23

# frovedis::ell\_matrix<T,I>

### 23.1 NAME

`frovedis::ell_matrix<T,I>` - A two-dimensional row-wise distributed ELL sparse matrix.

### 23.2 SYNOPSIS

```
#include <frovedis/matrix/ell_matrix.hpp>
```

#### 23.2.1 Constructors

```
ell_matrix ();  
ell_matrix (crs_matrix<T,I,0>& m);
```

#### 23.2.2 Public Member Functions

```
void debug_print ();  
crs_matrix<T,I,0> to_crs();
```

#### 23.2.3 Public Data Members

```
frovedis::node_local<ell_matrix_local<T,I>> data;  
size_t num_row;  
size_t num_col;
```

### 23.3 DESCRIPTION

A ELL matrix is one of the most popular sparse matrices with elements stored in column-major order. In this matrix representation, all the non-zero elements of a row are shifted (packed) at left side and all the rows are padded with zeros on the right to give them equal length.

It has two major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

val: a vector containing the left-shifted (zero-padded) non-zero elements of the sparse matrix stored in column-major order.  
 idx: a vector containing the corresponding column indices of the non-zero elements.

For example, if we consider the below sparse matrix:

```
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
1 0 0 0 2 0 0 4
0 0 0 1 2 0 0 3
```

Then its ELL image can be thought of as:

```
values      indices
1 2 4      0 4 7
1 2 3  =>   3 4 7
1 2 4      0 4 7
1 2 3      3 4 7
```

And its column-major memory representation would be:

```
val: {1, 1, 1, 1, 2, 2, 2, 2, 4, 3, 4, 3}
idx: {0, 3, 0, 3, 4, 4, 4, 4, 7, 7, 7, 7}
```

`ell_matrix<T,I>` is a two-dimensional template based distributed sparse data storage supported by `frovedis`. It contains public member “data” of the type `node_local<ell_matrix_local<T,I>>`. The actual distributed matrices are contained in all the worker nodes locally, thus named as `ell_matrix_local<T,I>` (see manual of `ell_matrix_local`) and “data” is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original sparse matrix.

The structure of this class is as follows:

```
template <class T, class I=size_t>
struct ell_matrix {
    frovedis::node_local<ell_matrix_local<T,I>> data;    // local matrix information
    size_t num_row;    // number of rows in the global sparse matrix
    size_t num_col;    // number of columns in the global sparse matrix
};
```

For example, if the above sparse matrix with 4 rows and 8 columns is distributed row-wise over two worker nodes, then the distribution can be shown as:

master	worker0	worker1
----	----	----
<code>ell_matrix&lt;int,size_t&gt;</code>	<code>-&gt; ell_matrix_local&lt;int,size_t&gt;</code>	<code>-&gt; ell_matrix_local&lt;int,size_t&gt;</code>
<code>*data: node_local&lt;</code>	<code>val: vector&lt;int&gt;</code>	<code>val: vector&lt;int&gt;</code>
<code>ell_matrix</code>	<code>({1,1,2,2,4,3})</code>	<code>({1,1,2,2,4,3})</code>
<code>_local&lt;int,</code>	<code>idx: vector&lt;size_t&gt;</code>	<code>idx: vector&lt;size_t&gt;</code>
<code>size_t&gt;&gt;</code>	<code>({0,3,4,4,7,7})</code>	<code>({0,3,4,4,7,7})</code>
<code>num_row: size_t (4)</code>	<code>local_num_row: size_t (2)</code>	<code>local_num_row: size_t (2)</code>
<code>num_col: size_t (8)</code>	<code>local_num_col: size_t (8)</code>	<code>local_num_col: size_t (8)</code>

The `node_local<ell_matrix_local<int,size_t>>` object “data” is simply a (\*)handle of the (->)local matrices at worker nodes.

This matrix can be loaded from a distributed crs matrix and also the matrix can be converted back to the distributed crs matrix. Thus loading/saving interfaces are not provided for distributed ell matrix.

### 23.3.1 Constructor Documentation

#### 23.3.1.1 `ell_matrix ()`

This is the default constructor which creates an empty distributed ell matrix without any memory allocation at worker nodes.

#### 23.3.1.2 `ell_matrix (crs_matrix<T,I,0>& m)`

This is the implicit conversion constructor to construct a distributed ell matrix from the input distributed crs matrix of the same “val” and “idx” type.

### 23.3.2 Public Member Function Documentation

#### 23.3.2.1 `crs_matrix<T,I,0> to_crs()`

This method can be used to convert the target distributed ell matrix into a distributed crs matrix of the same “val” and “idx” type.

#### 23.3.2.2 `void debug_print ()`

It prints the information related to the ELL storage (val, idx, number of rows and number of columns) of the local matrices node-by-node on the user terminal. It is mainly useful for debugging purpose.

### 23.3.3 Public Data Member Documentation

#### 23.3.3.1 `data`

An instance of `node_local<ell_matrix_local<T,0>>` type to contain the reference information related to local matrices at worker nodes.

#### 23.3.3.2 `num_row`

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

#### 23.3.3.3 `num_col`

A `size_t` attribute to contain the total number of columns in the 2D matrix view.

## 23.4 SEE ALSO

`crs_matrix`, `jds_matrix`, `ell_matrix_local`



## Chapter 24

# frovedis::jds\_matrix\_local<T,I,O,P>

### 24.1 NAME

`frovedis::jds_matrix_local<T,I,O,P>` - A two-dimensional non-distributed sparse matrix with jagged diagonal storage.

### 24.2 SYNOPSIS

```
#include <frovedis/matrix/jds_matrix.hpp>
```

#### 24.2.1 Constructors

```
jds_matrix_local ();  
jds_matrix_local (const jds_matrix_local<T,I,O,P>& m);  
jds_matrix_local (jds_matrix_local<T,I,O,P>&& m);  
jds_matrix_local (const crs_matrix_local<T,I,O>& m);
```

#### 24.2.2 Overloaded Operators

```
jds_matrix_local<T,I,O,P>& operator= (const jds_matrix_local<T,I,O,P>& m);  
jds_matrix_local<T,I,O,P>& operator= (jds_matrix_local<T,I,O,P>&& m);
```

#### 24.2.3 Public Member Functions

```
void savebinary (const std::string& dir);  
void debug_print ();
```

#### 24.2.4 Public Data Members

```
std::vector<T> val;  
std::vector<I> idx;  
std::vector<O> off;  
std::vector<P> perm;
```

```
size_t local_num_row;
size_t local_num_col;
```

## 24.3 DESCRIPTION

In the CRS format, the rows of the matrix can be reordered decreasingly according to the number of non-zeros per row. Then the compressed and permuted diagonals can be stored in a linear array. The new data structure is called jagged diagonals. The number of jagged diagonals is equal to the number of non-zeros in the first row, i.e., the largest number of non-zeros in any row of the sparse matrix.

A JDS (Jagged Diagonal Storage) matrix is one of the popular sparse matrices with such jagged diagonals (the elements stored in column-major order). It has four major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

```
val: a vector containing the non-zero elements of the jagged diagonals
of the matrix (in column-major order).
idx: a vector containing the column indices for each non-zero elements
in the jagged diagonals.
off: a vector containing the jagged diagonal offsets.
perm: a vector containing the indices of the permuted rows.
```

For example, if we consider the below sparse matrix:

```
1 0 0 0 1 0
0 5 9 0 2 0
0 1 0 4 0 0
0 0 0 1 0 5
```

then its JDS image can be thought of as:

```
5 9 2
1 5
1 4
1 1
```

Note that 2nd row of the matrix is having maximum non-zero elements. So this matrix will have 3 jagged diagonals. Rest three rows are having 2 non-zero elements each which can be permuted in any order (in this case row: 4th -> 3rd -> 1st).

Now when storing the diagonals, its JDS representation would be:

```
val: {5, 1, 1, 1, 9, 5, 4, 1, 2}
idx: {1, 3, 1, 0, 2, 5, 3, 4, 4}
off: {0, 4, 8, 9}
perm: {1, 3, 2, 0}
```

Jagged diagonal offset starts with 0 and it has n+1 number of elements, where n is the number of jagged diagonals in the sparse matrix. The difference between i+1th element and ith element in offset indicates number of non-zero elements present in ith jagged diagonal.

`jds_matrix_local<T,I,O,P>` is a two-dimensional template based non-distributed sparse data storage supported by `frovedis`. The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t, class P=size_t>
struct jds_matrix_local {
    std::vector<T> val;      // to contain non-zero elements of type "T"
    std::vector<I> idx;      // to contain column indices of type "I" (default: size_t)
    std::vector<O> off;      // to contain offsets of type "O" (default: size_t)
```

```

    std::vector<P> perm      // to contain permuted row indices of type "P" (default: size_t)
    size_t local_num_row;    // number of rows in the sparse matrix
    size_t local_num_col;    // number of columns in the sparse matrix
};

```

## 24.3.1 Constructor Documentation

### 24.3.1.1 `jds_matrix_local ()`

This is the default constructor which creates an empty jds matrix with `local_num_row = local_num_col = 0`.

### 24.3.1.2 `jds_matrix_local (const jds_matrix_local<T,I,0,P>& m)`

This is the copy constructor which creates a new jds matrix by deep-copying the contents of the input jds matrix.

### 24.3.1.3 `jds_matrix_local (jds_matrix_local<T,I,0,P>&& m)`

This is the move constructor. Instead of copying the input matrix, it moves the contents of the input rvalue matrix to the newly constructed matrix. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

### 24.3.1.4 `jds_matrix_local (const crs_matrix_local<T,I,0>& m)`

This is the implicit conversion constructor which creates a new jds matrix by converting the input crs matrix.

## 24.3.2 Overloaded Operator Documentation

### 24.3.2.1 `jds_matrix_local<T,I,0,P>& operator= (const jds_matrix_local<T,I,0,P>& m)`

It deep-copies the input jds matrix into the left-hand side matrix of the assignment operator “=”.

### 24.3.2.2 `jds_matrix_local<T,I,0,P>& operator= (jds_matrix_local<T,I,0,P>&& m)`

Instead of copying, it moves the contents of the input rvalue jds matrix into the left-hand side matrix of the assignment operator “=”. Thus it is faster and recommended to use when input matrix will no longer be used in a user program.

## 24.3.3 Public Member Function Documentation

### 24.3.3.1 `void debug_print ()`

It prints the information related to the compressed jagged diagonal storage (val, idx, off, perm, number of rows and number of columns) on the user terminal. It is mainly useful for debugging purpose.

### 24.3.3.2 void savebinary (const std::string& dir)

It writes the elements of a jds matrix to the specified directory as little-endian binary data.

The output directory will contain four files, named “nums”, “val”, “idx”, “off” and “perm”. “nums” is a text file containing the number of rows and number of columns information in first two lines of the file. And rest four files contain the binary data related to compressed jagged diagonal storage.

## 24.3.4 Public Data Member Documentation

### 24.3.4.1 val

An instance of `std::vector<T>` type to contain the non-zero elements of the (jagged diagonals elements) of the sparse matrix.

### 24.3.4.2 idx

An instance of `std::vector<I>` type to contain the column indices of the jagged diagonal elements of the sparse matrix.

### 24.3.4.3 off

An instance of `std::vector<O>` type to contain the jagged diagonal offsets.

### 24.3.4.4 perm

An instance of `std::vector<P>` type to contain the permuted row indices.

### 24.3.4.5 local\_num\_row

A `size_t` attribute to contain the number of rows in the 2D matrix view.

### 24.3.4.6 local\_num\_col

A `size_t` attribute to contain the number of columns in the 2D matrix view.

## 24.3.5 Public Global Function Documentation

### 24.3.5.1 jds\_matrix\_local<T,I,O,P> make\_jds\_matrix\_local\_loadbinary(dirname)

#### Parameters

*dirname*: A string object containing the name of the directory having the data to be loaded. It expects five files to be presented inside the specified directory, as follows:

- “nums” (containing number of rows and number of columns separated with new-line),
- “val” (containing binary data for non-zero elements),



- “idx” (containing binary column indices),
- “off” (containing binary offset values) and
- “perm” (containing binary permuted row indices)

### Purpose

This function loads the little-endian binary data from the specified directory and creates a `jds_matrix_local<T,I,0,P>` object filling the data loaded. The desired value type, “T” (e.g., int, float, double etc.) must be specified explicitly when loading the matrix data. If not specified, the other three types “I”, “O” and “P” would be `size_t` as default types.

For example, considering “./bin” is a directory having the binary data to be loaded,

```
auto m1 = make_jds_matrix_local_loadbinary<int>("./bin");
auto m2 = make_jds_matrix_local_loadbinary<float>("./bin");
```

“m1” will be a `jds_matrix_local<int,size_t,size_t,size_t>`, whereas  
“m2” will be a `jds_matrix_local<float,size_t,size_t,size_t>`.

### Return Value

On success, it returns the created matrix of the type `jds_matrix_local<T,I,0,P>`. Otherwise, it throws an exception.

#### 24.3.5.2 `jds_matrix_local<T,I,0,P> crs2jds(m)`

##### Parameters

*m*: An object of the type `crs_matrix_local<T,I,0>`.

##### Purpose

This function converts an input crs storage into an equivalent jds storage of the same “val”, “num” and “off” type. The input matrix would remain unchanged.

##### Return Value

On success, it will return the converted `jds_matrix_local<T,I,0,P>`. Otherwise, it throws an exception.

#### 24.3.5.3 `std::vector<T> operator*(m,v)`

##### Parameters

*m*: A const& object of the type `jds_matrix_local<T,I,0,P>`.

*v*: A const& object of the type `std::vector<T>`.

##### Purpose

This function performs matrix-vector multiplication between a sparse jds matrix object with a `std::vector` of same value (T) type. It expects the size of the input vector should be greater than or equal to the number of columns in the input jds matrix.

##### Return Value

On success, it returns the resultant vector of the type `std::vector<T>`. Otherwise, it throws an exception.

## 24.4 SEE ALSO

`crs_matrix_local`, `ell_matrix_local`, `jds_matrix`



## Chapter 25

# frovedis::jds\_matrix<T,I,O,P>

### 25.1 NAME

`frovedis::jds_matrix<T,I,O,P>` - A two-dimensional row-wise distributed sparse matrix with jagged diagonal storage.

### 25.2 SYNOPSIS

```
#include <frovedis/matrix/jds_matrix.hpp>
```

#### 25.2.1 Constructors

```
jds_matrix ();  
jds_matrix (const crs_matrix<T,I,O>& m);
```

#### 25.2.2 Public Member Functions

```
void debug_print ();
```

#### 25.2.3 Public Data Members

```
frovedis::node_local<jds_matrix_local<T,I,O,P>> data;  
size_t num_row;  
size_t num_col;
```

### 25.3 DESCRIPTION

In the CRS format, the rows of the matrix can be reordered decreasingly according to the number of non-zeros per row. Then the compressed and permuted diagonals can be stored in a linear array. The new data structure is called jagged diagonals. The number of jagged diagonals is equal to the number of non-zeros in the first row, i.e., the largest number of non-zeros in any row of the sparse matrix.

A JDS (Jagged Diagonal Storage) matrix is one of the popular sparse matrices with such jagged diagonals (the elements stored in column-major order). It has four major components while storing the non-zero elements, as explained below along with the number of rows and the number of columns in the sparse matrix.

val: a vector containing the non-zero elements of the jagged diagonals of the matrix (in column-major order).  
 idx: a vector containing the column indices for each non-zero elements in the jagged diagonals.  
 off: a vector containing the jagged diagonal offsets.  
 perm: a vector containing the indices of the permuted rows.

For example, if we consider the below sparse matrix:

```
1 0 0 0 1 0
0 5 9 0 2 0
0 1 0 4 0 0
0 0 0 1 0 5
```

then its JDS image can be thought of as:

```
5 9 2
1 5
1 4
1 1
```

Note that 2nd row of the matrix is having maximum non-zero elements. So this matrix will have 3 jagged diagonals. Rest three rows are having 2 non-zero elements each which can be permuted in any order (in this case row: 4th -> 3rd -> 1st).

Now when storing the diagonals, its JDS representation would be:

```
val: {5, 1, 1, 1, 9, 5, 4, 1, 2}
idx: {1, 3, 1, 0, 2, 5, 3, 4, 4}
off: {0, 4, 8, 9}
perm: {1, 3, 2, 0}
```

Jagged diagonal offset starts with 0 and it has n+1 number of elements, where n is the number of jagged diagonals in the sparse matrix. The difference between i+1th element and ith element in offset indicates number of non-zero elements present in ith jagged diagonal.

jds\_matrix<T,I,O,P> is a two-dimensional template based distributed sparse data storage supported by frovedis. It contains public member “data” of the type node\_local<jds\_matrix\_local<T,I,O,P>>. The actual distributed matrices are contained in all the worker nodes locally, thus named as jds\_matrix\_local<T,I,O,P> (see manual of ell\_matrix\_local) and “data” is the reference to these local matrices at worker nodes. It also contains dimension information related to the global matrix i.e., number of rows and number of columns in the original sparse matrix.

The structure of this class is as follows:

```
template <class T, class I=size_t, class O=size_t, class P=size_t> struct jds_matrix {
  frovedis::node_local<jds_matrix_local<T,I,O,P>>
  data; // local matrix information
  size_t local_num_row; // number of rows in the sparse matrix
  size_t local_num_col; // number of columns in the sparse matrix
};
```

For example, if the above sparse matrix with 4 rows and 6 columns is distributed row-wise over two worker nodes, then the distribution can be shown as:

master	worker0	Worker1
----	----	----
jds_matrix<int,size_t, size_t,size_t>	-> jds_matrix_local<int, size_t,size_t,size_t>	-> jds_matrix_local<int, size_t,size_t,size_t>

<code>*data: node_local&lt;</code>	<code>val: vector&lt;int&gt;</code>	<code>val: vector&lt;int&gt;</code>
<code>  jds_matrix</code>	<code>({5,1,9,1,2})</code>	<code>({1,1,5,4})</code>
<code>  _local&lt;int,</code>	<code>idx: vector&lt;size_t&gt;</code>	<code>idx: vector&lt;size_t&gt;</code>
<code>  size_t,size_t,</code>	<code>({1,0,2,4,4})</code>	<code>({3,1,5,3})</code>
<code>  size_t&gt;&gt;</code>	<code>off: vector&lt;size_t&gt;</code>	<code>off: vector&lt;size_t&gt;</code>
	<code>({0,2,4,5})</code>	<code>({0,2,4})</code>
	<code>perm: vector&lt;size_t&gt;</code>	<code>perm: vector&lt;size_t&gt;</code>
	<code>({1,0})</code>	<code>({1,0})</code>
<code>  num_row: size_t (4)</code>	<code>  local_num_row: size_t (2)</code>	<code>  local_num_row: size_t (2)</code>
<code>  num_col: size_t (6)</code>	<code>  local_num_col: size_t (6)</code>	<code>  local_num_col: size_t (6)</code>

The `node_local<jds_matrix_local<int,size_t,size_t,size_t>>` object “data” is simply a (\*)handle of the (->)local matrices at worker nodes.

## 25.3.1 Constructor Documentation

### 25.3.1.1 `jds_matrix ()`

This is the default constructor which creates an empty distributed jds matrix without any memory allocation at worker nodes.

### 25.3.1.2 `jds_matrix (const crs_matrix<T,I,0>& m)`

This is the implicit conversion constructor which creates a new jds matrix by converting the input crs matrix.

## 25.3.2 Public Member Function Documentation

### 25.3.2.1 `void debug_print ()`

It prints the information related to the compressed jagged diagonal storage (val, idx, off, perm, number of rows and number of columns) node-by-node on the user terminal. It is mainly useful for debugging purpose.

## 25.3.3 Public Data Member Documentation

### 25.3.3.1 `data`

An instance of `node_local<jds_matrix_local<T,I,0,P>>` type to contain the reference information related to local matrices at worker nodes.

### 25.3.3.2 `num_row`

A `size_t` attribute to contain the total number of rows in the 2D matrix view.

### 25.3.3.3 `num_col`

A `size_t` attribute to contain the total number of columns in the 2D matrix view.

## 25.4 SEE ALSO

jds\_matrix\_local, crs\_matrix, ell\_matrix

## Chapter 26

# frovedis::linear\_regression\_model<T>

### 26.1 NAME

`linear_regression_model<T>` - A data structure used in modeling the outputs of the frovedis linear regression algorithms.

### 26.2 SYNOPSIS

```
#include <frovedis/ml/glm/linear_model.hpp>
```

#### 26.2.1 Constructors

```
linear_regression_model ()  
linear_regression_model (size_t num_ftr, T intercept=0.0)  
linear_regression_model (const linear_regression_model<T>& model)  
linear_regression_model (linear_regression_model<T>&& model)
```

#### 26.2.2 Overloaded Operators

```
linear_regression_model<T>& operator= (const linear_regression_model<T>& model)  
linear_regression_model<T>& operator= (linear_regression_model<T>&& model)  
linear_regression_model<T> operator+ (const linear_regression_model<T>& model) const  
linear_regression_model<T> operator- (const linear_regression_model<T>& model) const  
void operator+= (const linear_regression_model<T>& model)  
void operator-= (const linear_regression_model<T>& model)
```

#### 26.2.3 Public Member Functions

```
std::vector<T> predict (DATA_MATRIX& mat)  
size_t get_num_features () const  
void save (const std::string& path) const  
void savebinary (const std::string& path) const  
void load (const std::string& path) const  
void loadbinary (const std::string& path) const
```

```
void debug_print() const
node_local<linear_regression_model<T>> broadcast ()
```

## 26.3 DESCRIPTION

`linear_regression_model<T>` models the output of the frovedis linear regression algorithms, e.g., linear regression, lasso regression and ridge regression. Each of the trainer interfaces of these algorithms aim to optimize an initial model and output the same after optimization. This model has the below structure:

```
template <class T>
struct linear_regression_model {
    std::vector<T> weight; // the weight vector associated with each input training features
    T intercept;          // the bias intercept term
    SERIALIZE (weight, intercept)
};
```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

### 26.3.1 Constructor Documentation

#### 26.3.1.1 `linear_regression_model ()`

Default constructor. It creates an empty linear regression model with default “intercept” value as 0.0.

#### 26.3.1.2 `linear_regression_model (size_t num_ftr, T intercept=0.0)`

Parameterized constructor. It accepts the number-of-features input from the user and allocates the memory for the model of the same size. If no initial value of the “intercept” is provided, it considers the default value as 0.0.

#### 26.3.1.3 `linear_regression_model (const linear_regression_model<T>& model)`

Copy constructor. It accepts an lvalue object of the same type and deep-copies the same in the newly constructed object.

#### 26.3.1.4 `linear_regression_model (linear_regression_model<T>&& model)`

Move constructor. It accepts an rvalue object of the same type and instead of copying, it moves the contents in the newly constructed object.

### 26.3.2 Overloaded Operator Documentation

#### 26.3.2.1 `linear_regression_model<T>& operator= (const linear_regression_model<T>& model)`

It deep-copies the contents of the input lvalue model into the left-hand side model of the assignment operator “=”.



**26.3.2.2** `linear_regression_model<T>& operator= (linear_regression_model<T>&& model)`

Instead of copying, it moves the contents of the input rvalue model into the left-hand side model of the assignment operator “=”.

**26.3.2.3** `linear_regression_model<T> operator+ (const linear_regression_model<T>& model)  
const`

This operator is used to add two linear regression models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 + m2” can easily be evaluated on them.

**26.3.2.4** `linear_regression_model<T> operator- (const linear_regression_model<T>& model)  
const`

This operator is used to subtract two linear regression models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 - m2” can easily be evaluated on them.

**26.3.2.5** `void operator+= (const linear_regression_model<T>& model)`

This operator is used to add two linear regression models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 += m2” will add m2 with m1 and update m1 itself.

**26.3.2.6** `void operator-= (const linear_regression_model<T>& model)`

This operator is used to subtract two linear regression models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 -= m2” will subtract m2 from m1 and update m1 itself.

**26.3.3** Pubic Member Function Documentation**26.3.3.1** `std::vector<T> predict (DATA_MATRIX& mat)`

This function is used on a trained model (after training is done by respective trainers) to predict the unknown output labels based on the given input matrix. It uses prediction logic according to linear regression algorithm.

This function expects any input data matrix which provides an overloaded multiply “\*” operator with a vector type object. E.g., if “v” is an object of `std::vector<T>` type, then “mat \* v” should be supported and it should return the resultant vector of the type `std::vector<T>`. DATA\_MATRIX can be `frovedis::crs_matrix_local<T>`, `frovedis::ell_matrix_local<T>` etc.

On succesful prediction, this function returns the predicted values in the form of `std::vector<T>`. It will throw an exception, if any error occurs.

**26.3.3.2** `size_t get_num_features () const`

It returns the number-of-features in the target model.

**26.3.3.3 void save (const std::string& path) const**

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

**26.3.3.4 void savebinary (const std::string& path) const**

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

**26.3.3.5 void load (const std::string& path) const**

It loads the target linear regression model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

**26.3.3.6 void loadbinary (const std::string& path) const**

It loads the target linear regression model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

**26.3.3.7 void debug\_print() const**

It prints the contents of the model on the user terminal. It is mainly useful for debugging purpose.

**26.3.3.8 node\_local<linear\_regression\_model<T>> broadcast ()**

It broadcasts the target model to all the participating MPI processes (worker nodes) in the system. This is an efficient (as it does not involve the serialization overhead of the model weight vector) implementation than simple “frovedis:broadcast(model)” call.

**26.3.4 Public Data Member Documentation****26.3.4.1 weight**

An object of `std::vector<T>` type. It is used to store the weight/theta components associated with each training features.

**26.3.4.2 intercept**

A “T” type object (mainly “float” or “double”). It is used to store the bias intercept term of the model.

**26.4 SEE ALSO**

logistic\_regression\_model, svm\_model

## Chapter 27

# Linear Regression

### 27.1 NAME

Linear Regression - A regression algorithm supported by Frowedis to predict the continuous output without any regularization.

### 27.2 SYNOPSIS

```
#include <frowedis/ml/glm/linear_regression_with_sgd.hpp>

linear_regression_model<T>
linear_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

linear_regression_model<T>
linear_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    linear_regression_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

#include <frowedis/ml/glm/linear_regression_with_lbfgs.hpp>

linear_regression_model<T>
linear_regression_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
```

```

        size_t hist_size = 10,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

linear_regression_model<T>
linear_regression_with_lbfgs::train (crs_matrix<T>& data,
        dvector<T>& label,
        linear_regression_model<T>& initModel,
        size_t numIteration = 1000,
        T alpha = 0.01,
        size_t hist_size = 10,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

```

## 27.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

$$L(\mathbf{w}; \mathbf{x}, y) := 1/2(\mathbf{w}^T \mathbf{x} - y)^2$$

Where the vectors  $\mathbf{x}$  are the training data examples and  $y$  are their corresponding labels which we want to predict.  $\mathbf{w}$  is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of  $\mathbf{w}^T \mathbf{x}$  and  $y$ . Linear regression does not use any regularizer.

The gradient of the squared loss is:  $(\mathbf{w}^T \mathbf{x} - y) \cdot \mathbf{x}$

Frovedis provides implementation of linear regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form **min**  $f(\mathbf{w})$  is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense  $n \times n$  approximation to the inverse Hessian ( $n$  being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapid convergence compared with other first-order optimization.

### 27.3.1 Detailed Description

#### 27.3.1.1 linear\_regression\_with\_sgd::train()

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of `T` type containing the minibatch fraction (Default: 1.0)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default:

false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer, but without any regularizer. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 27.3.1.2 `linear_regression_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `linear_regression_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of T type containing the minibatch fraction (Default: 1.0)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer, but without any regularizer. Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 27.3.1.3 `linear_regression_with_lbfgs::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains a linear regression model with LBFGS optimizer, but without any regularizer. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

**27.3.1.4 linear\_regression\_with\_lbfgs::train()****Parameters**

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `linear_regression_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains a linear regression model with LBFGS optimizer, but without any regularizer. Instead of an initial guess of zeros, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

**27.4 SEE ALSO**

`linear_regression_model`, `lasso_regression`, `ridge_regression`

## Chapter 28

# Lasso Regression

### 28.1 NAME

Lasso Regression - A regression algorithm supported by Frowedis to predict the continuous output with L1 regularization.

### 28.2 SYNOPSIS

```
#include <frowedis/ml/glm/lasso_with_sgd.hpp>

linear_regression_model<T>
lasso_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

linear_regression_model<T>
lasso_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    linear_regression_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

#include <frowedis/ml/glm/lasso_with_lbfgs.hpp>

linear_regression_model<T>
lasso_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
```

```

        size_t numIteration = 1000,
        T alpha = 0.01,
        size_t hist_size = 10,
        T regParam = 0.01,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

linear_regression_model<T>
lasso_with_lbfgs::train (crs_matrix<T>& data,
        dvector<T>& label,
        linear_regression_model<T>& initModel,
        size_t numIteration = 1000,
        T alpha = 0.01,
        size_t hist_size = 10,
        T regParam = 0.01,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

```

## 28.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

$$L(\mathbf{w}; \mathbf{x}, y) := 1/2(\mathbf{w}^T \mathbf{x} - y)^2$$

Where the vectors  $\mathbf{x}$  are the training data examples and  $y$  are their corresponding labels which we want to predict.  $\mathbf{w}$  is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of  $\mathbf{w}^T \mathbf{x}$  and  $y$ . Lasso regression uses L1 regularization to address the overfit problem.

The gradient of the squared loss is:  $(\mathbf{w}^T \mathbf{x} - y) \cdot \mathbf{x}$

The gradient of the regularizer is:  $\text{sign}(\mathbf{w})$

Frovedis provides implementation of lasso regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form  $\min \mathbf{f}(\mathbf{w})$  is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense  $n \times n$  approximation to the inverse Hessian ( $n$  being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

### 28.3.1 Detailed Description

#### 28.3.1.1 lasso\_with\_sgd::train()

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix



*label*: A `dvector<T>` containing the output labels  
*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)  
*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*minibatchFraction*: A parameter of T type containing the minibatch fraction (Default: 1.0)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L1 regularization. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 28.3.1.2 `lasso_with_sgd::train()`

### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix  
*label*: A `dvector<T>` containing the output labels  
*initModel*: A `linear_regression_model<T>` containing the user provided initial model values  
*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)  
*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*minibatchFraction*: A parameter of T type containing the minibatch fraction (Default: 1.0)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L1 regularization. Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 28.3.1.3 `lasso_with_lbfgs::train()`

### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix  
*label*: A `dvector<T>` containing the output labels  
*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*hist\_size*: A parameter of size\_t type containing the number of gradient history to be stored (Default: 10)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with LBFGS optimizer and with L1 regularization. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 28.3.1.4 lasso\_with\_lbfgs::train()

### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix  
*label*: A `dvector<T>` containing the output labels  
*initModel*: A `linear_regression_model<T>` containing the user provided initial model values  
*numIteration*: A size\_t parameter containing the maximum number of iteration count (Default: 1000)  
*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*hist\_size*: A parameter of size\_t type containing the number of gradient history to be stored (Default: 10)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with LBFGS optimizer and with L1 regularizer. Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

## 28.4 SEE ALSO

`linear_regression_model`, `linear_regression`, `ridge_regression`

## Chapter 29

# Ridge Regression

### 29.1 NAME

Ridge Regression - A regression algorithm supported by Frowedis to predict the continuous output with L2 regularization.

### 29.2 SYNOPSIS

```
#include <frowedis/ml/glm/ridge_regression_with_sgd.hpp>

linear_regression_model<T>
ridge_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

linear_regression_model<T>
ridge_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    linear_regression_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

#include <frowedis/ml/glm/ridge_regression_with_lbfgs.hpp>

linear_regression_model<T>
ridge_regression_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
```

```

        size_t numIteration = 1000,
        T alpha = 0.01,
        size_t hist_size = 10,
        T regParam = 0.01,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

linear_regression_model<T>
ridge_regression_with_lbfgs::train (crs_matrix<T>& data,
        dvector<T>& label,
        linear_regression_model<T>& initModel,
        size_t numIteration = 1000,
        T alpha = 0.01,
        size_t hist_size = 10,
        T regParam = 0.01,
        bool isIntercept = false,
        T convergenceTol = 0.001,
        MatType mType = HYBRID)

```

## 29.3 DESCRIPTION

Linear least squares is the most common formulation for regression problems. It is a linear method with the loss function given by the **squared loss**:

$$L(\mathbf{w}; \mathbf{x}, y) := 1/2(\mathbf{w}^T \mathbf{x} - y)^2$$

Where the vectors  $\mathbf{x}$  are the training data examples and  $y$  are their corresponding labels which we want to predict.  $\mathbf{w}$  is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. The method is called linear since it can be expressed as a function of  $\mathbf{w}^T \mathbf{x}$  and  $y$ . Ridge regression uses L2 regularization to address the overfit problem.

The gradient of the squared loss is:  $(\mathbf{w}^T \mathbf{x} - y) \cdot \mathbf{x}$

The gradient of the regularizer is:  $\mathbf{w}$

Frovedis provides implementation of ridge regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form  $\min \mathbf{f}(\mathbf{w})$  is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense  $n \times n$  approximation to the inverse Hessian ( $n$  being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

### 29.3.1 Detailed Description

#### 29.3.1.1 ridge\_regression\_with\_sgd::train()

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of T type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L2 regularization. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 29.3.1.2 `ridge_regression_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `linear_regression_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of T type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with stochastic gradient descent with minibatch optimizer and with L2 regularization. Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 29.3.1.3 `ridge_regression_with_lbfgs::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*hist\_size*: A parameter of size\_t type containing the number of gradient history to be stored (Default: 10)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with LBFGS optimizer and with L2 regularization. It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

#### 29.3.1.4 ridge\_regression\_with\_lbfgs::train()

### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix  
*label*: A `dvector<T>` containing the output labels  
*initModel*: A `linear_regression_model<T>` containing the user provided initial model values  
*numIteration*: A size\_t parameter containing the maximum number of iteration count (Default: 1000)  
*alpha*: A parameter of T type containing the learning rate (Default: 0.01)  
*hist\_size*: A parameter of size\_t type containing the number of gradient history to be stored (Default: 10)  
*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)  
*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)  
*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)  
*mType*: frovedis::MatType parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a linear regression model with LBFGS optimizer and with L2 regularizer. Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `linear_regression_model<T>`.

## 29.4 SEE ALSO

`linear_regression_model`, `linear_regression`, `lasso_regression`

## Chapter 30

# frovedis::logistic\_regression\_model<T>

### 30.1 NAME

`logistic_regression_model<T>` - A data structure used in modeling the outputs of the frovedis logistic regression algorithm.

### 30.2 SYNOPSIS

```
#include <frovedis/ml/glm/linear_model.hpp>
```

#### 30.2.1 Constructors

```
logistic_regression_model ()  
logistic_regression_model (size_t num_ftr, T intercept=0.0, T threshold=0.5)  
logistic_regression_model (const logistic_regression_model<T>& model)  
logistic_regression_model (logistic_regression_model<T>&& model)
```

#### 30.2.2 Overloaded Operators

```
logistic_regression_model<T>& operator= (const logistic_regression_model<T>& model)  
logistic_regression_model<T>& operator= (logistic_regression_model<T>&& model)  
logistic_regression_model<T> operator+ (const logistic_regression_model<T>& model) const  
logistic_regression_model<T> operator- (const logistic_regression_model<T>& model) const  
void operator+= (const logistic_regression_model<T>& model)  
void operator-= (const logistic_regression_model<T>& model)
```

#### 30.2.3 Public Member Functions

```
std::vector<T> predict (DATA_MATRIX& mat)  
std::vector<T> predict_probability (DATA_MATRIX& mat)  
void set_threshold (T threshold)  
size_t get_num_features () const  
void save (const std::string& path) const  
void savebinary (const std::string& path) const
```

```

void load (const std::string& path) const
void loadbinary (const std::string& path) const
void debug_print() const
node_local<logistic_regression_model<T>> broadcast ()

```

### 30.3 DESCRIPTION

`logistic_regression_model<T>` models the output of the `frovedis` logistic regression algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization. This model has the below structure:

```

template <class T>
struct logistic_regression_model {
    std::vector<T> weight; // the weight vector associated with each input training features
    T intercept;           // the bias intercept term
    T threshold;           // the threshold value used in prediction
    SERIALIZE (weight, intercept, threshold)
};

```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

#### 30.3.1 Constructor Documentation

##### 30.3.1.1 `logistic_regression_model ()`

Default constructor. It creates an empty logistic regression model with default “intercept” value as 0.0 and default “threshold” value as 0.5.

##### 30.3.1.2 `logistic_regression_model (size_t num_ftr, T intercept=0.0, T threshold=0.5)`

Parameterized constructor. It accepts the number-of-features input from the user and allocates the memory for the model of the same size. If no initial value of the “intercept” is provided, it considers the default value as 0.0. If no “threshold” value is provided, it considers the default value as 0.5.

##### 30.3.1.3 `logistic_regression_model (const logistic_regression_model<T>& model)`

Copy constructor. It accepts an lvalue object of the same type and deep-copies the same in the newly constructed object.

##### 30.3.1.4 `logistic_regression_model (logistic_regression_model<T>&& model)`

Move constructor. It accepts an rvalue object of the same type and instead of copying, it moves the contents in the newly constructed object.



### 30.3.2 Overloaded Operator Documentation

#### 30.3.2.1 `logistic_regression_model<T>& operator= (const logistic_regression_model<T>& model)`

It deep-copies the contents of the input lvalue model into the left-hand side model of the assignment operator “=”.

#### 30.3.2.2 `logistic_regression_model<T>& operator= (logistic_regression_model<T>&& model)`

Instead of copying, it moves the contents of the input rvalue model into the left-hand side model of the assignment operator “=”.

#### 30.3.2.3 `logistic_regression_model<T> operator+ (const logistic_regression_model<T>& model) const`

This operator is used to add two logistic regression models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 + m2” can easily be evaluated on them.

#### 30.3.2.4 `logistic_regression_model<T> operator- (const logistic_regression_model<T>& model) const`

This operator is used to subtract two logistic regression models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 - m2” can easily be evaluated on them.

#### 30.3.2.5 `void operator+= (const logistic_regression_model<T>& model)`

This operator is used to add two logistic regression models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 += m2” will add m2 with m1 and update m1 itself.

#### 30.3.2.6 `void operator-= (const logistic_regression_model<T>& model)`

This operator is used to subtract two logistic regression models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 -= m2” will subtract m2 from m1 and update m1 itself.

### 30.3.3 Public Member Function Documentation

#### 30.3.3.1 `std::vector<T> predict (DATA_MATRIX& mat)`

This function is used on a trained model (after training is done) to predict the unknown output labels based on the given input matrix. It uses prediction logic according to logistic regression algorithm.

This function expects any input data matrix which provides an overloaded multiply “\*” operator with a vector type object. E.g., if “v” is an object of `std::vector<T>` type, then “mat \* v” should be supported and it should return the resultant vector of the type `std::vector<T>`. DATA\_MATRIX can be `frovedis::crs_matrix_local<T>`, `frovedis::ell_matrix_local<T>` etc.

On successful prediction, this function returns the predicted values in the form of `std::vector<T>`. Currently, it supports only binary prediction in the form of 1 (yes) and -1 (no). It will throw an exception, if any error occurs.

### 30.3.3.2 `std::vector<T> predict_probability (DATA_MATRIX& mat)`

This function is also used on trained model (after training is done) to predict the unknown output labels based on the given input matrix. But instead of returning yes/no predictions, it returns the raw probabilities in the form of `std::vector<T>` corresponding to each new feature vector in the given matrix. Like `predict()`, it can also accept any data matrix, if support of “\*” operator with a vector is provided for that matrix.

### 30.3.3.3 `void set_threshold (T threshold)`

It sets threshold value of the target model with the provided value. It will throw an exception, if negative value is provided.

### 30.3.3.4 `size_t get_num_features () const`

It returns the number-of-features in the target model.

### 30.3.3.5 `void save (const std::string& path) const`

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

### 30.3.3.6 `void savebinary (const std::string& path) const`

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

### 30.3.3.7 `void load (const std::string& path) const`

It loads the target logistic regression model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

### 30.3.3.8 `void loadbinary (const std::string& path) const`

It loads the target logistic regression model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

### 30.3.3.9 `void debug_print() const`

It prints the contents of the model on the user terminal. It is mainly useful for debugging purpose.

**30.3.3.10 node\_local<logistic\_regression\_model<T>> broadcast ()**

It broadcasts the target model to all the participating MPI processes (worker nodes) in the system. This is an efficient (as it does not involve the serialization overhead of the model weight vector) implementation than simple “frovedis:broadcast(model)” call.

**30.3.4 Public Data Member Documentation****30.3.4.1 weight**

An object of `std::vector<T>` type. It is used to store the weight/theta components associated with each training features.

**30.3.4.2 intercept**

A “T” type object (mainly “float” or “double”). It is used to store the bias intercept term of the model.

**30.3.4.3 threshold**

A “T” type object (mainly “float” or “double”). It is used to hold the threshold value used in prediction.

**30.4 SEE ALSO**

linear\_regression\_model, svm\_model



## Chapter 31

# Logistic Regression

### 31.1 NAME

Logistic Regression - A classification algorithm supported by Frovedis to predict the binary output with logistic loss.

### 31.2 SYNOPSIS

```
#include <frovedis/ml/glm/logistic_regression_with_sgd.hpp>

logistic_regression_model<T>
logistic_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

logistic_regression_model<T>
logistic_regression_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    logistic_regression_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

#include <frovedis/ml/glm/logistic_regression_with_lbfgs.hpp>
```

```

logistic_regression_model<T>
logistic_regression_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    size_t hist_size = 10,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

logistic_regression_model<T>
logistic_regression_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
    logistic_regression_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    size_t hist_size = 10,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

```

### 31.3 DESCRIPTION

Classification aims to divide items into categories. The most common classification type is binary classification, where there are two categories, usually named positive and negative. Frovedis supports binary classification algorithm only.

Logistic regression is widely used to predict a binary response. It is a linear method with the loss function given by the **logistic loss**:

$$L(\mathbf{w}; \mathbf{x}, y) := \log(1 + \exp(-y\mathbf{w}^T\mathbf{x}))$$

Where the vectors  $\mathbf{x}$  are the training data examples and  $y$  are their corresponding labels (can be either -1 for negative response or 1 for positive response) which we want to predict.  $\mathbf{w}$  is the linear model (also called as weight) which uses a single weighted sum of features to make a prediction. Logistic Regression supports ZERO, L1 and L2 regularization to address the overfit problem.

The gradient of the logistic loss is:  $-y(1 - 1 / (1 + \exp(-y\mathbf{w}^T\mathbf{x}))) \cdot \mathbf{x}$

The gradient of the L1 regularizer is:  $\text{sign}(\mathbf{w})$

And The gradient of the L2 regularizer is:  $\mathbf{w}$

For binary classification problems, the algorithm outputs a binary logistic regression model. Given a new data point, denoted by  $\mathbf{x}$ , the model makes predictions by applying the logistic function:

$$f(z) := 1 / (1 + \exp(-z))$$

Where  $z = \mathbf{w}^T\mathbf{x}$ . By default, if  $f(\mathbf{w}^T\mathbf{x}) > 0.5$ , the response is positive (1), else the response is negative (-1).

Frovedis provides implementation of logistic regression with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form  $\min \mathbf{f}(\mathbf{w})$  is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS

is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense  $n \times n$  approximation to the inverse Hessian ( $n$  being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

### 31.3.1 Detailed Description

#### 31.3.1.1 `logistic_regression_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of `T` type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of `T` type containing the regularization parameter (also called  $\lambda$ ) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

##### Purpose

It trains a logistic regression model with stochastic gradient descent with minibatch optimizer and with provided regularizer (if not ZERO). It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

##### Return Value

After the successful training, it returns a trained model of the type `logistic_regression_model<T>`.

#### 31.3.1.2 `logistic_regression_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `logistic_regression_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of `T` type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of `T` type containing the regularization parameter (also called  $\lambda$ ) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a logistic regression model with stochastic gradient descent with minibatch optimizer and with provided regularizer (if not ZERO). Instead of an initial guess of zeros, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `logistic_regression_model<T>`.

#### 31.3.1.3 `logistic_regression_with_lbfgs::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*regParam*: A parameter of `T` type containing the regularization parameter (also called lambda) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

### Purpose

It trains a logistic regression model with LBFGS optimizer and with provided regularizer (if not ZERO). It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

### Return Value

After the successful training, it returns a trained model of the type `logistic_regression_model<T>`.

#### 31.3.1.4 `logistic_regression_with_lbfgs::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `logistic_regression_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*regParam*: A parameter of `T` type containing the regularization parameter (also called lambda) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)



*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains a logistic regression model with LBFGS optimizer and with provided regularizer (if not ZERO). Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `logistic_regression_model<T>`.

## 31.4 SEE ALSO

`logistic_regression_model`, `linear_svm`



## Chapter 32

# frovedis::svm\_model<T>

### 32.1 NAME

`svm_model<T>` - A data structure used in modeling the outputs of the frovedis linear SVM (Support Vector Machine) algorithm.

### 32.2 SYNOPSIS

```
#include <frovedis/ml/glm/linear_model.hpp>
```

#### 32.2.1 Constructors

```
svm_model ()  
svm_model (size_t num_ftr, T intercpt=0.0, T threshold=0.0)  
svm_model (const svm_model<T>& model)  
svm_model (svm_model<T>&& model)
```

#### 32.2.2 Overloaded Operators

```
svm_model<T>& operator= (const svm_model<T>& model)  
svm_model<T>& operator= (svm_model<T>&& model)  
svm_model<T> operator+ (const svm_model<T>& model) const  
svm_model<T> operator- (const svm_model<T>& model) const  
void operator+= (const svm_model<T>& model)  
void operator-= (const svm_model<T>& model)
```

#### 32.2.3 Public Member Functions

```
std::vector<T> predict (DATA_MATRIX& mat)  
std::vector<T> predict_probability (DATA_MATRIX& mat)  
void set_threshold (T threshold)  
size_t get_num_features () const  
void save (const std::string& path) const  
void savebinary (const std::string& path) const
```

```

void load (const std::string& path) const
void loadbinary (const std::string& path) const
void debug_print() const
node_local<svm_model<T>> broadcast ()

```

## 32.3 DESCRIPTION

`svm_model<T>` models the output of the frovedis linear SVM (Support Vector Machine) algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization. This model has the below structure:

```

template <class T>
struct svm_model {
    std::vector<T> weight; // the weight vector associated with each input training features
    T intercept;           // the bias intercept term
    T threshold;           // the threshold value used in prediction
    SERIALIZE (weight, intercept, threshold)
};

```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

### 32.3.1 Constructor Documentation

#### 32.3.1.1 `svm_model ()`

Default constructor. It creates an empty logistic regression model with default “intercept” value as 0.0 and default “threshold” value as 0.0.

#### 32.3.1.2 `svm_model (size_t num_ftr, T intercept=0.0, T threshold=0.0)`

Parameterized constructor. It accepts the number-of-features input from the user and allocates the memory for the model of the same size. If no initial value of the “intercept” is provided, it considers the default value as 0.0. If no “threshold” value is provided, it considers the default value as 0.0.

#### 32.3.1.3 `svm_model (const svm_model<T>& model)`

Copy constructor. It accepts an lvalue object of the same type and deep-copies the same in the newly constructed object.

#### 32.3.1.4 `svm_model (svm_model<T>&& model)`

Move constructor. It accepts an rvalue object of the same type and instead of copying, it moves the contents in the newly constructed object.

### 32.3.2 Overloaded Operator Documentation

#### 32.3.2.1 `svm_model<T>& operator= (const svm_model<T>& model)`

It deep-copies the contents of the input lvalue model into the left-hand side model of the assignment operator “=”.

#### 32.3.2.2 `svm_model<T>& operator= (svm_model<T>&& model)`

Instead of copying, it moves the contents of the input rvalue model into the left-hand side model of the assignment operator “=”.

#### 32.3.2.3 `svm_model<T> operator+ (const svm_model<T>& model) const`

This operator is used to add two svm models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 + m2” can easily be evaluated on them.

#### 32.3.2.4 `svm_model<T> operator- (const svm_model<T>& model) const`

This operator is used to subtract two svm models and outputs the resultant model. If m1 and m2 are two models, expression like “m1 - m2” can easily be evaluated on them.

#### 32.3.2.5 `void operator+= (const svm_model<T>& model)`

This operator is used to add two svm models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 += m2” will add m2 with m1 and update m1 itself.

#### 32.3.2.6 `void operator-= (const svm_model<T>& model)`

This operator is used to subtract two svm models. But instead of returning a new model, it updates the target model with the resultant model. If m1 and m2 are two models, then “m1 -= m2” will subtract m2 from m1 and update m1 itself.

### 32.3.3 Public Member Function Documentation

#### 32.3.3.1 `std::vector<T> predict (DATA_MATRIX& mat)`

This function is used on a trained model (after training is done) to predict the unknown output labels based on the given input matrix. It uses prediction logic according to linear SVM algorithm.

This function expects any input data matrix which provides an overloaded multiply “\*” operator with a vector type object. E.g., if “v” is an object of `std::vector<T>` type, then “mat \* v” should be supported and it should return the resultant vector of the type `std::vector<T>`. `DATA_MATRIX` can be `frovedis::crs_matrix_local<T>`, `frovedis::ell_matrix_local<T>` etc.

On successful prediction, this function returns the predicted values in the form of `std::vector<T>`. Currently, it supports only binary prediction in the form of 1 (yes) and -1 (no). It will throw an exception, if any error occurs.

**32.3.3.2 std::vector<T> predict\_probability (DATA\_MATRIX& mat)**

This function is also used on trained model (after training is done) to predict the unknown output labels based on the given input matrix. But instead of returning yes/no predictions, it returns the raw probabilities in the form of `std::vector<T>` corresponding to each new feature vector in the given matrix. Like `predict()`, it can also accept any data matrix, if support of “\*” operator with a vector is provided for that matrix.

**32.3.3.3 void set\_threshold (T threshold)**

It sets threshold value of the target model with the provided value. It will throw an exception, if negative value is provided.

**32.3.3.4 size\_t get\_num\_features () const**

It returns the number-of-features in the target model.

**32.3.3.5 void save (const std::string& path) const**

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

**32.3.3.6 void savebinary (const std::string& path) const**

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

**32.3.3.7 void load (const std::string& path) const**

It loads the target svm model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

**32.3.3.8 void loadbinary (const std::string& path) const**

It loads the target svm model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

**32.3.3.9 void debug\_print() const**

It prints the contents of the model on the user terminal. It is mainly useful for debugging purpose.

**32.3.3.10 node\_local<svm\_model<T>> broadcast ()**

It broadcasts the target model to all the participating MPI processes (worker nodes) in the system. This is an efficient (as it does not involve the serialization overhead of the model weight vector) implementation than simple “`frovedis:broadcast(model)`” call.

### 32.3.4 Public Data Member Documentation

#### 32.3.4.1 `weight`

An object of `std::vector<T>` type. It is used to store the weight/theta components associated with each training features.

#### 32.3.4.2 `intercept`

A “T” type object (mainly “float” or “double”). It is used to store the bias intercept term of the model.

#### 32.3.4.3 `threshold`

A “T” type object (mainly “float” or “double”). It is used to hold the threshold value used in prediction.

## 32.4 *SEE ALSO*

`linear_regression_model`, `logistic_regression_model`





## Chapter 33

# Linear SVM

### 33.1 NAME

Linear SVM (Support Vector Machines) - A classification algorithm supported by Frovedis to predict the binary output with hinge loss.

### 33.2 SYNOPSIS

```
#include <frovedis/ml/glm/svm_with_sgd.hpp>

svm_model<T>
svm_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

svm_model<T>
svm_with_sgd::train (crs_matrix<T>& data,
    dvector<T>& label,
    svm_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    T miniBatchFraction = 1.0,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

#include <frovedis/ml/glm/svm_with_lbfgs.hpp>
```

```

svm_model<T>
svm_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
    size_t numIteration = 1000,
    T alpha = 0.01,
    size_t hist_size = 10,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

svm_model<T>
svm_with_lbfgs::train (crs_matrix<T>& data,
    dvector<T>& label,
    svm_model<T>& initModel,
    size_t numIteration = 1000,
    T alpha = 0.01,
    size_t hist_size = 10,
    T regParam = 0.01,
    RegType regtyp = ZERO,
    bool isIntercept = false,
    T convergenceTol = 0.001,
    MatType mType = HYBRID)

```

### 33.3 DESCRIPTION

Classification aims to divide items into categories. The most common classification type is binary classification, where there are two categories, usually named positive and negative. Frowedis supports binary classification algorithm only.

The Linear SVM is a standard method for large-scale classification tasks. It is a linear method with the loss function given by the **hinge loss**:

$$L(w; x, y) := \max\{0, 1 - ywTx\}$$

Where the vectors  $x$  are the training data examples and  $y$  are their corresponding labels (can be either -1 for negative response or 1 for positive response) which we want to predict.  $w$  is the linear model (also known as weight) which uses a single weighted sum of features to make a prediction. Linear SVM supports ZERO, L1 and L2 regularization to address the overfit problem.

The gradient of the hinge loss is:  $-y.x$ , if  $ywTx < 1$ , 0 otherwise.

The gradient of the L1 regularizer is:  $\text{sign}(w)$

And The gradient of the L2 regularizer is:  $w$

For binary classification problems, the algorithm outputs a binary svm model. Given a new data point, denoted by  $x$ , the model makes predictions based on the value of  $wTx$ .

By default, if  $wTx \geq 0$ , then the response is positive (1), else the response is negative (-1).

Frowedis provides implementation of linear SVM with two different optimizers: (1) stochastic gradient descent with minibatch and (2) LBFGS optimizer.

The simplest method to solve optimization problems of the form  $\min f(w)$  is gradient descent. Such first-order optimization methods well-suited for large-scale and distributed computation. Whereas, L-BFGS is an optimization algorithm in the family of quasi-Newton methods to solve the optimization problems of the similar form.

Like the original BFGS, L-BFGS (Limited Memory BFGS) uses an estimation to the inverse Hessian matrix to steer its search through feature space, but where BFGS stores a dense  $n \times n$  approximation to the inverse Hessian ( $n$  being the number of features in the problem), L-BFGS stores only a few vectors that represent the approximation implicitly. L-BFGS often achieves rapider convergence compared with other first-order optimization.

### 33.3.1 Detailed Description

#### 33.3.1.1 `svm_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of `T` type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of `T` type containing the regularization parameter (also known as  $\lambda$ ) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

##### Purpose

It trains an svm model with stochastic gradient descent with minibatch optimizer and with provided regularizer (if not ZERO). It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

##### Return Value

After the successful training, it returns a trained model of the type `svm_model<T>`.

#### 33.3.1.2 `svm_with_sgd::train()`

##### Parameters

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `svm_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*minibatchFraction*: A parameter of `T` type containing the minibatch fraction (Default: 1.0)

*regParam*: A parameter of `T` type containing the regularization parameter (also known as  $\lambda$ ) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of `T` type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains an svm model with stochastic gradient descent with minibatch optimizer and with provided regularizer (if not ZERO). Instead of an initial guess of zeros, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `svm_model<T>`.

**33.3.1.3 svm\_with\_lbfgs::train()****Parameters**

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains an svm model with LBFGS optimizer and with provided regularizer (if not ZERO). It starts with an initial guess of zeros for the model vector and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `svm_model<T>`.

**33.3.1.4 svm\_with\_lbfgs::train()****Parameters**

*data*: A `crs_matrix<T>` containing the sparse feature matrix

*label*: A `dvector<T>` containing the output labels

*initModel*: A `svm_model<T>` containing the user provided initial model values

*numIteration*: A `size_t` parameter containing the maximum number of iteration count (Default: 1000)

*alpha*: A parameter of T type containing the learning rate (Default: 0.01)

*hist\_size*: A parameter of `size_t` type containing the number of gradient history to be stored (Default: 10)

*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)

*regtyp*: A parameter of the type `frovedis::RegType`, which can be either ZERO, L1 or L2 (Default: ZERO)

*isIntercept*: A boolean parameter to specify whether to include intercept term (bias term) or not (Default: false)

*convergenceTol*: A parameter of T type containing the threshold value to determine the convergence (Default: 0.001)

*mType*: `frovedis::MatType` parameter specifying the matrix type to be used for internal calculation (Default: HYBRID for SX architecture, CRS for other architectures)

**Purpose**

It trains an svm model with LBFGS optimizer and with provided regularizer (if not ZERO). Instead of an initial guess of zeors, it starts with user provided initial model values and keeps updating the model to minimize the cost function until convergence is achieved or maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `svm_model<T>`.

## 33.4 SEE ALSO

`svm_model`, `logistic_regression`



## Chapter 34

# frovedis::matrix\_factorization\_model<T>

### 34.1 NAME

`matrix_factorization_model<T>` - A data structure used in modeling the outputs of the frovedis matrix factorization using ALS algorithm

### 34.2 SYNOPSIS

```
#include <frovedis/ml/recommendation/matrix_factorization_model.hpp>
```

#### 34.2.1 Constructors

```
matrix_factorization_model ()  
matrix_factorization_model (size_t num_row, size_t num_col,  
                             size_t factor, size_t seed=0)  
matrix_factorization_model (const matrix_factorization_model<T>& model)  
matrix_factorization_model (matrix_factorization_model<T>&& model)
```

#### 34.2.2 Overloaded Operators

```
matrix_factorization_model<T>& operator= (const matrix_factorization_model<T>& model)  
matrix_factorization_model<T>& operator= (matrix_factorization_model<T>&& model)
```

#### 34.2.3 Public Member Functions

```
T predict (size_t uid, size_t pid)  
std::vector<T> predict_all (const std::vector<std::pair<size_t,size_t>> IDs)  
rowmajor_matrix_local<T> predict_all()  
  
std::vector<std::pair<size_t,T>> recommend_products(size_t uid, int num)  
std::vector<std::pair<size_t,T>> recommend_users(size_t pid, int num)  
  
void save (const std::string& path)  
void savebinary (const std::string& path)
```

```

void load (const std::string& path)
void loadbinary (const std::string& path)

size_t get_rank ()
void debug_print ()
node_local<matrix_factorization_model<T>> broadcast()

```

### 34.3 DESCRIPTION

`matrix_factorization_model<T>` models the output of the frovedis matrix factorization using ALS (alternating least square) algorithm, the trainer interface of which aims to optimize an initial model and outputs the same after optimization. This model has the below structure:

```

template <class T>
struct matrix_factorization_model {
    std::vector<T> X; // user-feature vector of the size numRows*factor
    std::vector<T> Y; // product-feature vector of the size numCols*factor
    size_t numRows;
    size_t numCols;
    size_t factor;
    SERIALIZE (X, Y, numRows, numCols, factor)
};

```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

#### 34.3.1 Constructor Documentation

##### 34.3.1.1 `matrix_factorization_model ()`

Default constructor. It creates an empty matrix factorization model with `numRows = numCols = factor = 0`.

##### 34.3.1.2 `matrix_factorization_model (size_t num_row, size_t num_col,`

`size_t factor, size_t seed)`

Parameterized constructor. It accepts number of rows(M), number of columns(N), latent factor(F) and seed value in order to create a model with “X” matrix of the dimension MxF and “Y” matrix of the dimension NxF initialized with random numbers according to the given seed.

##### 34.3.1.3 `matrix_factorization_model (const matrix_factorization_model<T>& model)`

Copy constructor. It accepts an lvalue object of the same type and deep-copies the same in the newly constructed object.

##### 34.3.1.4 `matrix_factorization_model (matrix_factorization_model<T>&& model)`

Move constructor. It accepts an rvalue object of the same type and instead of copying, it moves the contents in the newly constructed object.



### 34.3.2 Overloaded Operator Documentation

#### 34.3.2.1 `matrix_factorization_model<T>& operator= (const matrix_factorization_model<T>& model)`

It deep-copies the contents of the input lvalue model into the left-hand side model of the assignment operator “=”.

#### 34.3.2.2 `matrix_factorization_model<T>& operator= (matrix_factorization_model<T>&& model)`

Instead of copying, it moves the contents of the input rvalue model into the left-hand side model of the assignment operator “=”.

### 34.3.3 Pubic Member Function Documentation

#### 34.3.3.1 `T predict (size_t uid, size_t pid)`

This method can be used on a trained model in order to predict the rating confidence value for the given product id, by the given user id.

“uid” should be in between 0 to numRows-1.

And “pid” should be in between 0 to numCols-1. Otherwise exception will be thrown.

#### 34.3.3.2 `std::vector<T> predict_all (const std::vector<std::pair<size_t,size_t>> IDs)`

This method can be used to predict the rating confidence values for a given list of pair of some user ids and product ids.

In the list of pairs, “uid” should be in between 0 to numRows-1.

And “pid” should be in between 0 to numCols-1. Otherwise exception will be thrown.

On successful prediction, it returns the predicted scores in the form of `std::vector<T>`.

#### 34.3.3.3 `rowmajor_matrix_local<T> predict_all ()`

This method can be used in order to predict the rating confidence values for all the users and for all the products. Thus internally it performs a product of X and Y component of the model ( $X * Y^t$ ) and returns the resultant scores in the form of a `rowmajor_matrix_local<T>` with MxN dimension, where M is the number of rows in X component and N is the number of rows in Y component. This method is useful in case of debugging the model.

#### 34.3.3.4 `std::vector<std::pair<size_t,T>> recommend_products(size_t uid, int num)`

This method can be used to recommend given “num” number of products for the user with given user id in sorted order (highest scored products to lowest scored products).

“uid” should be in between 0 to numRows-1.

If `num > numCols`, then “numCols” number of products would be recommended. On success, it returns a vector of pairs containing recommended product ids and their corresponding rating confidence scores by the given user.

**34.3.3.5    `std::vector<std::pair<size_t,T>> recommend_users(size_t pid, int num)`**

This method can be used to recommend given “num” number of users for the product with given product id in sorted order (user with highest scores to user with lowest scores).

“pid” should be in between 0 to numCols-1.

If num > numRows, then “numRows” number of users would be recommended. On success, it returns a vector of pairs containing recommended user ids and their corresponding rating confidence scores for the given product.

**34.3.3.6    `size_t get_rank ()`**

It returns the latent factor of the target model.

**34.3.3.7    `void save (const std::string& path)`**

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

**34.3.3.8    `void savebinary (const std::string& path)`**

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

**34.3.3.9    `void load (const std::string& path)`**

It loads the target matrix factorization model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

**34.3.3.10    `void loadbinary (const std::string& path)`**

It loads the target matrix factorization model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

**34.3.3.11    `void debug_print()`**

It prints the contents of the X and Y components of the model on the user terminal. It is mainly useful for debugging purpose.

**34.3.3.12    `node_local<matrix_factorization_model<T>> broadcast ()`**

It broadcasts the target model to all the participating MPI processes (worker nodes) in the system. This is an efficient implementation (as it does not involve serialization overhead of the X and Y components of the model) than simple “frovedis:broadcast(model)” call.

### 34.3.4 Public Data Member Documentation

#### 34.3.4.1 X

An T type vector used to model the user-feature matrix of the model.

#### 34.3.4.2 Y

An T type vector used to model the product-feature matrix of the model.

#### 34.3.4.3 numRows

A size\_t attribute containing the number of rows in X component of the model.

#### 34.3.4.4 numCols

A size\_t attribute containing the number of rows in Y component of the model.

#### 34.3.4.5 factor

A size\_t attribute containing the latent factor of the model.



## Chapter 35

# Matrix Factorization using ALS

### 35.1 NAME

Matrix Factorization using ALS - A matrix factorization algorithm commonly used for recommender systems.

### 35.2 SYNOPSIS

```
#include <frovedis/ml/recommendation/als.hpp>

matrix_factorization_model<T>
matrix_factorization_using_als::train (crs_matrix<T>& data,
    size_t factor,
    int numIter = 100,
    T alpha = 0.01,
    T regParam = 0.01,
    size_t seed = 0)
```

### 35.3 DESCRIPTION

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. Frovedis currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. Frovedis uses the alternating least squares (ALS) algorithm to learn these latent factors. The algorithm is based on a paper “Collaborative Filtering for Implicit Feedback Datasets” by Hu, et al.

#### 35.3.1 Detailed Description

##### 35.3.1.1 `matrix_factorization_using_als::train()`

###### Parameters

*data*: A `crs_matrix<T>` containing the sparse rating matrix

*factor*: A `size_t` parameter containing the number of latent factors (also known as rank)

*numIter*: A `size_t` parameter containing the maximum number of iteration count (Default: 100)

*alpha*: A parameter of `T` type containing the learning rate (Default: 0.01)

*regParam*: A parameter of T type containing the regularization parameter (also known as lambda) (Default: 0.01)

*seed*: A size\_t parameter containing the seed value to initialize the model structures with random values (Default: 0)

**Purpose**

It trains a matrix factorization model with alternating least squares (ALS) algorithm. It starts with initializing the model structures of the size MxF and NxF (where MxN is the dimension of the input rating matrix and F is the latent factors count) with random values and keeps updating them until maximum iteration count is reached. After the training, it returns the trained output model.

**Return Value**

After the successful training, it returns a trained model of the type `matrix_factorization_model<T>` which can be used for predicting user choices or making recommendation.

## 35.4 SEE ALSO

`matrix_factorization_model`

# Chapter 36

## kmeans

### 36.1 NAME

kmeans - A clustering algorithm commonly used in EDA (exploratory data analysis).

### 36.2 SYNOPSIS

```
#include <frovedis/ml/clustering/kmeans.hpp>

rowmajor_matrix_local<T>
frovedis::kmeans (crs_matrix<T,I,0>& samples,
                  int k,
                  int iter,
                  T eps,
                  long seed = 0)

std::vector<int>
frovedis::kmeans_assign_cluster (crs_matrix_local<T,I,0>& mat,
                                 rowmajor_matrix_local<T>& centroid)
```

### 36.3 DESCRIPTION

Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity. K-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters (K).

#### 36.3.1 Detailed Description

##### 36.3.1.1 frovedis::kmeans()

###### Parameters

*samples*: A `crs_matrix<T,I,0>` containing the sparse data points

*k*: An integer parameter containing the number of clusters

*iter*: An integer parameter containing the maximum number of iteration count

*eps*: A parameter of T type containing the epsilon value

*seed*: A parameter of long type containing the seed value to generate the random rows from the given data samples (Default: 0)

**Purpose**

It clusters the given data points into a predefined number (k) of clusters.  
After the successful clustering, it returns the k centroids of the cluster.

**Return Value**

After the successful clustering it returns the centroids of the type `rowmajor_matrix_local<T>`, where each column shows each centroid vector.

**36.3.1.2 `frovedis::kmeans__assign__cluster()`****Parameters**

*mat*: A `crs_matrix_local<T,I,0>` containing the new sparse data points to be assigned to the cluster

*centroid*: A `rowmajor_matrix_local<T>` containing the centroids

**Purpose**

After getting the centroids from `kmeans()`, they can be used to assign data to the closest centroid using `kmeans__assign__cluster()`.

**Return Value**

It returns a `std::vector<int>` containing the assigned values.



## Chapter 37

# spectral\_clustering\_model

### 37.1 NAME

`spectral_clustering_model<T>` - A data structure used in modeling the outputs of the frovedis spectral clustering algorithm.

### 37.2 SYNOPSIS

```
#include <frovedis/ml/clustering/spectral_clustering_model.hpp>
```

#### 37.2.1 Constructors

```
spectral_clustering_model<T>()  
spectral_clustering_model<T>(rowmajor_matrix<T>& aff, std::vector<int>& lbl, int ncluster)
```

#### 37.2.2 Public Member Functions

```
void save(const std::string& dir)  
void savebinary(const std::string& dir)  
void load(const std::string& dir)  
void loadbinary(const std::string& dir)  
void debug_print()
```

#### 37.2.3 DESCRIPTION

`spectral_clustering_model<T>` models the output of the frovedis spectral clustering algorithms. This model has the below structure:

```
template <class T>  
struct spectral_clustering_model {  
    rowmajor_matrix<T> affinity_matrix,  
    std::vector<int> labels,  
    int nclusters) };
```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

### 37.2.4 Public Member Function Documentation

**void save(const std::string& dir)**

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

**void savebinary(const std::string& dir)**

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

**void load(const std::string& dir)**

It loads the target linear regression model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

**void loadbinary(const std::string& dir)**

It loads the target linear regression model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

**void debug\_print()**

It prints the contents of the model on the user terminal. It is mainly useful for debugging purpose.

### 37.2.5 Public Data Member Documentation

**affinity\_matrix:** A `rowmajor_matrix<T>` that return the values of distance co-related data values required for model building.

**labels:** It contains the final cluster output obtained from the successful creation of the model.

**nclusters:** An integer parameter containing the number of components for clusters.

## 37.3 SEE ALSO

`spectral_clustering`, `spectral_embedding_model`

## Chapter 38

# spectral clustering

### 38.1 NAME

spectral clustering - A clustering algorithm commonly used in EDA (exploratory data analysis), using the spectrum (eigenvalues) of the similarity matrix of the data to perform clustering.

### 38.2 SYNOPSIS

```
#include <frovedis/ml/clustering/spectral_clustering.hpp>
```

```
spectral_clustering_model<T>
frovedis::spectral_clustering(rowmajor_matrix<T>& mat,
    int ncluster = 2,
    int n_comp = 2,
    int niter = 100,
    double eps = 0.01,
    bool norm_laplacian = true,
    bool precomputed = false,
    bool drop_first = false,
    double gamma = 1.0,
    int mode = 1)

spectral_clustering_model<T>
frovedis::spectral_clustering(rowmajor_matrix<T>&& mat,
    int ncluster = 2,
    int n_comp = 2,
    int niter = 100,
    double eps = 0.01,
    bool norm_laplacian = true,
    bool precomputed = false,
    bool drop_first = false,
    double gamma = 1.0,
    int mode = 1)
```

## 38.3 DESCRIPTION

Clustering is an unsupervised learning problem whereby we aim to group subsets of entities with one another based on some notion of similarity. In spectral clustering, the data points are treated as nodes of a graph. Thus, clustering is treated as a graph partitioning problem. The user can provide input in form of co-ordinate values for graphical representation or as affinity matrix. The components or features are identified as per column order in matrix data. The nodes are then mapped to a low-dimensional space that can be easily segregated to form clusters.

### 38.3.1 Detailed Description

#### 38.3.1.1 Public Global Function Documentation

`spectral_clustering_model<T>`

`spectral_clustering(mat,ncluster,n_comp,niter,eps,norm_laplacian,precomputed,drop_first,gamma,mode)`

##### Parameters

*mat*: A rowmajor\_matrix of type “T”(where T can be either float or double) containing n-dimensional data points. It can be treated as either input data matrix or precomputed affinity matrix, based on the boolean parameter “precomputed”. The input can be passed as an lvalue or rvalue. The matrix is internally cleared to save computation memory during the algorithm, in case the input matrix is a rvalue.

*ncluster*: An integer parameter containing the number of required clusters(Default: 2)

*n\_comp*: An integer parameter containing the number of components for clusters(Default: 2)

*niter*: An integer parameter containing the maximum number of iteration count for kmeans(Default: 300)

*eps*: A parameter of double type containing the epsilon value for kmeans(Default: 0.1)

*norm\_laplacian*: A boolean parameter if set True, then compute normalized Laplacian else not(Default: true)

*precomputed*: A boolean parameter if set False, then internally affinity matrix is computed based on input matrix otherwise input matrix is treated as precomputed affinity matrix and internal affinity computation is skipped(Default: false)

*drop\_first*: A boolean parameter if set True, then drops the first eigenvector. The first eigenvector of a normalized laplacian is full of constants, thus if drop\_first is set true, compute (n\_comp+1) eigenvectors and will drop the first vector. Otherwise it will calculate n\_comp number of eigenvectors(Default: false)

*gamma*: The value required for computing nearby relational meaningful eigenvalues(Default: 1.0)

*mode*: A parameter required to set the eigen computation method. It can be either 1 or 3, 1 for generic and 3 for shift-invert mode(Default: 1)

##### Purpose

It computes the clusters with respect to the relational distance between the given data points, using normalized eigenvectors.

After the successful clustering, it returns `spectral_clustering_model<T>` containing the computed cluster values.

##### Return Value

After the successful clustering it returns the `spectral_clustering_model<T>` from the computed labels containing the centroids from kmeans.

## 38.4 SEE ALSO

`spectral_clustering_model`, `spectral_embedding`

## Chapter 39

# spectral\_embedding\_model

### 39.1 NAME

`spectral_embedding_model<T>` - A data structure used in modeling the outputs of the frovedis spectral embedding algorithm.

### 39.2 SYNOPSIS

```
#include <frovedis/ml/clustering/spectral_embedding_model.hpp>
```

#### 39.2.1 Constructors

```
spectral_embedding_model<T>()  
spectral_embedding_model<T>(rowmajor_matrix<T>& aff, rowmajor_matrix<T>& embed)
```

#### 39.2.2 Public Member Functions

```
void save(const std::string& dir)  
void savebinary(const std::string& dir)  
void load(const std::string& dir)  
void loadbinary(const std::string& dir)  
void debug_print()
```

#### 39.2.3 DESCRIPTION

`spectral_embedding_model<T>` models the output of the frovedis spectral embedding algorithms. This model has the below structure:

```
template <class T>  
struct spectral_embedding_model {  
    rowmajor_matrix<T> affinity_matrix,  
    rowmajor_matrix<T> embed_matrix };
```

This is a template based data structure, where “T” is supposed to be “float” (single-precision) or “double” (double-precision). Note this is a serialized data structure. The detailed description can be found in subsequent sections.

### 39.2.4 Public Member Function Documentation

**void save(const std::string& dir)**

It saves the target model in the specified path in simple text format. It will throw an exception, if any error occurs during the save operation.

**void savebinary(const std::string& dir)**

It saves the target model in the specified path in (little-endian) binary data format. It will throw an exception, if any error occurs during the save operation.

**void load(const std::string& dir)**

It loads the target linear regression model from the data in specified text file. It will throw an exception, if any error occurs during the load operation.

**void loadbinary(const std::string& dir)**

It loads the target linear regression model from the data in specified (little-endian) binary file. It will throw an exception, if any error occurs during the load operation.

**void debug\_print()**

It prints the contents of the model on the user terminal. It is mainly useful for debugging purpose.

### 39.2.5 Public Data Member Documentation

**affinity\_matrix:** A `rowmajor_matrix<T>` that return the values of distance co-related data values required for model building.

**embed\_matrix:** A `rowmajor_matrix<T>` containing meaningful patterns in normalized high dimensional data.

## 39.3 SEE ALSO

`spectral_embedding`, `spectral_clustering_model`

## Chapter 40

# spectral embedding

### 40.1 NAME

spectral embedding - Spectral embedding is useful for reducing the dimensionality of data that is expected to lie on a low-dimensional manifold contained within a high-dimensional space, it yields a low-dimensional representation of the data that best preserves the structure of the original manifold in the sense that points that are close to each other on the original manifold will also be close after embedding. At the same time, the embedding emphasizes clusters in the original data.

### 40.2 SYNOPSIS

```
#include <frovedis/ml/clustering/spectral_embedding.hpp>

spectral_embedding_model<T>
frovedis::spectral_embedding(rowmajor_matrix<T>& mat,
    int n_comp = 2,
    bool norm_laplacian = true,
    bool precomputed = false,
    bool drop_first = true,
    double gamma = 1.0,
    int mode = 1)

spectral_embedding_model<T>
frovedis::spectral_embedding(rowmajor_matrix<T>&& mat,
    int n_comp = 2,
    bool norm_laplacian = true,
    bool precomputed = false,
    bool drop_first = true,
    double gamma = 1.0,
    int mode = 1)
```

### 40.3 DESCRIPTION

Spectral embedding is the accurate method for extraction of meaningful patterns in high dimensional data. It forms an affinity matrix given by the specified function and applies spectral decomposition to the corresponding graph laplacian. The resulting transformation is given by the value of the eigenvectors for each data point.

### 40.3.1 Detailed Description

#### 40.3.1.1 Public Global Function Documentation

`spectral_embedding_model<T>`

`spectral_embedding(mat,n_comp,norm_laplacian,precomputed,drop_first,gamma,mode)`

##### Parameters

*mat*: A rowmajor\_matrix of type “T”(where T can be either float or double) containing n-dimensional data points. It can be treated as either input data matrix or precomputed affinity matrix, based on the boolean parameter “precomputed”. The input can be passed as an lvalue or rvalue. The matrix is internally cleared to save computation memory during the algorithm, in case the input matrix is a rvalue.

*n\_comp*: An integer parameter containing the number of components for clusters(Default: 2)

*norm\_laplacian*: A boolean parameter if set True, then compute normalized Laplacian else not(Default: true)

*precomputed*: A boolean parameter if set False, then internally affinity matrix is computed based on input matrix otherwise input matrix is treated as precomputed affinity matrix and internal affinity computation is skipped(Default: false)

*drop\_first*: A boolean parameter if set True, then drops the first eigenvector. The first eigenvector of a normalized laplacian is full of constants, thus if drop\_first is set true, compute (n\_comp+1) eigenvectors and will drop the first vector. Otherwise it will calculate n\_comp number of eigenvectors(Default: true)

*gamma*: The value required for computing nearby relational meaningful eigenvalues(Default: 1.0)

*mode*: A parameter required to set the eigen computation method. It can be either 1 or 3, 1 for generic and 3 for shift-invert mode(Default: 1)

##### Purpose

After getting the affinity matrix by computing distance co-relation, this is used to extract meaningful patterns using normalized eigenvectors.

##### Return Value

It returns a `spectral_embedding_model<T>` containing the values of the largest eigenvectors obtained from the normalized laplacian.

## 40.4 SEE ALSO

`spectral_clustering`, `spectral_embedding_model`



## Chapter 41

# Timeseries analysis using ARIMA

### 41.1 NAME

ARIMA - Autoregressive Integrated Moving Average model is a time series model that is used to forecast data based on the dataset of past to predict/forecast the future.

### 41.2 SYNOPSIS

```
#include <frovedis/ml/tsa/arma.hpp>
```

#### 41.2.1 Constructors

```
Arima(const node_local<std::vector>& sample_data_l,  
size_t ar_lag_l = 1,  
size_t diff_order_l = 0,  
size_t ma_lag_l = 0,  
size_t seasonal_l = 0,  
bool auto_arma_l = false,  
std::string solver_l = "lapack")
```

#### 41.2.2 Public Member Functions

```
void fit()  
  
std::vector  
predict(size_t start_step, size_t stop_step)  
  
std::vector  
forecast(size_t steps = 1)
```

### 41.3 DESCRIPTION

Frovedis provides a timeseries model in order to predict the future values based on the past values. Each component in ARIMA functions as a parameter with a standard notation. For ARIMA models, a

standard notation would be ARIMA with  $p(\text{AR})$ ,  $d(\text{I})$ , and  $q(\text{MA})$  which indicate the type of ARIMA model to be used. A 0 value can be used as a parameter and would mean that the particular component should not be used in the model. This way, the ARIMA model can be constructed to perform the function of an ARMA model, or even simple AR (1,0,0), I(0,1,0), or MA(0,0,1) models. **However, the current implementaion cannot be used to construct a pure MA model.**

Provedis ARIMA is comparatively faster than statsmodel ARIMA for large timeseries dataset as it is light weighted and having accuracy as good as statsmodel. Also, it provides the feature of auto ARIMA which can fit the best lag for AR and MA. This is a useful feature for the users who do not have knowledge about data analytics and those who do not want to use ACF, PACF and ADfuller tests in order to judge the data first before calling ARIMA.

Unlike statsmodel ARIMA, it does not use MLE (Maximum Likelihood Estimation), rather it uses OLS (Ordinary Least Squares).

**Note:-** Also, rather than converging around the mean after some number of predictions, it tends to follow the trend i.e it diverges towards increasing or decreasing trend.

### 41.3.1 Detailed Description

#### 41.3.1.1 1. Arima()

##### Parameters

**sample\_data\_l:** A `node_local<std::vector<T>>` containing the timeseries data.

**ar\_lag\_l:** A `size_t` parameter which specifies the order (lags used) of the Auto-Regressive Model. Currently, this parameter must be greater than or equal to 1.

**diff\_order\_l:** A `size_t` parameter which specifies the order of differencing. It must be a positive parameter.

**ma\_lag\_l:** A `size_t` parameter which specifies the order (lags used) of the Moving-Average Model. It must be a positive parameter.

**seasonal:** A `size_t` parameter which specifies the interval of seasonal differencing. In case the data has some seasonality, then it can handle it. It must be a positive parameter. This is an optional parameter.

**auto\_arima:** A `bool` parameter which specifies whether to use auto (brute) ARIMA. If set to `true`, it treats the `ar_lag_l` and `ma_lag_l` parameter as the highest limit for its iteration and auto fits the lags with the best RMSE score. This is an optional parameter.

**solver:** A `string` parameter which specifies the solver to be used for linear regression. By default, `lapack` solver will be used. It supports `lapack`, `scalapack`, `lbfgs` and `sgd` solver. When specified, e.g `lbfgs`, then it uses `lbfgs` solver for linear regression.

**Note:-** To get the best performance and accuracy from ARIMA, use `solver='lapack'`.

##### Purpose

It initializes the Arima instance with the given parameters.

Currently, the number of samples in the timeseries data must be greater than sum of ARIMA order (`ar_lag_l`, `diff_order_l`, `ma_lag_l`, `seasonal`).

```
`N_samples >= (ar_lag_l + diff_order_l + ma_lag_l + seasonal)`
```

##### Return Value

It simply returns the instance of Arima class.

#### 41.3.1.2 2. void fit()

##### Purpose

It is used to fit the model parameters on the basis of given parameters and data provided in Arima constructor.

##### Return Value

void

**41.3.1.3 3. std::vector predict(size\_t start\_step, size\_t stop\_step)**

**Parameters *start\_step*:** A `size_t` parameter which specifies the starting index from which the values are to be predicted. This parameter must be greater than or equal to 0.

***stop\_step*:** A `size_t` parameter which specifies the index till which the values are to be predicted. This parameter must be greater than or equal to 0.

**Purpose**

It is used to perform in-sample prediction and out-of-sample prediction.

During prediction, `stop_step` must not be less than `start_step`.

For prediction:

- When `start_step` and `stop_step` are both less than the order, then it will return a `std::vector<T>` having only the fitted values as predictions.
- When `start_step` is less than the sum of ARIMA order but `stop_step` is not, then it will return a `std::vector<T>` having a combination of the fitted values from `start_step` till order value and will use new prediction from `order + 1` to `stop_step`.
- When `start_step` and `stop_step` both are greater than the number of sample in timeseries data, then it will return a `std::vector<T>` having newly generated predictions.

**Return Value**

It returns a `std::vector<T>` of predicted values.

**41.3.1.4 4. std::vector forecast(size\_t steps = 1)****Parameters**

***steps*:** A `size_t` parameter which specifies the number of out of sample values to be predicted. By default, it will be set as 1. It must be greater than or equal to 1.

**Purpose**

It is used to perform out of sample forecasting. It is a wrapper function around `predict()`.

**Return Value**

It returns a `std::vector<T>` of out of sample forecasts.

