
User Manual: Local and Parallel Computation Toolbox for Gaussian Process Regression (GPLP)

Version 1.0

Chiwoo Park

March 13, 2012

Abstract

The **GPLP** Version 1.0 is the **Octave** and **Matlab** implementation of several fast computation algorithms for efficiently computing Gaussian process regression with a large amount of data. The main contribution of this toolbox is two-folds. First, the toolbox implements several localized Gaussian process regression methods, which are not included as a part of **GPML** toolbox version 3.1 developed by Rasmussen and Nickisch. Specifically, **GPLP** includes the domain decomposition method (only applicable for spatial datasets), partial independent conditional, localized probabilistic regression, and bagging for Gaussian process regression. Second, it provides two parallel computation codes of the domain decomposition method, which are executable on both of **Unix** and **Windows** operating systems with several open source softwares (**Octave**, **MatMPI** and **TORQUE** resource manager). Based on the parallel implementations, the spatial regression (regression in two dimensional domain) with a huge size of dataset can be handled in timely manner. This documentation provides several examples to show how the general users utilize this toolbox. It also includes how the advanced users could extend the functions of this toolbox.

Contents

1	GPLP Toolbox	3
2	Installation	4
2.1	Basic Installation	4
2.2	Setting Up a Parallel Computation Environment	6
3	Usage and Demonstration	7
3.1	General Usage	7
3.2	Case 1: Execution of Domain Decomposition Method with Local Hyperparameters	8
3.3	Case 2: Execution of Domain Decomposition Method with Global Hyperparameters	11
3.4	Case 3: Parallel Execution of Domain Decomposition Method in Single-Machine-Multi-Core Environment	11
3.5	Case 4: Parallel Execution of Domain Decomposition Method in Multi-Machine Environment	13
3.6	Case 5: Execution of Partial Independent Conditional	16
3.7	Case 6: Execution of Localized Probabilistic Regression	18
3.8	Case 7: Execution of Bagging for Gaussian Process Regression	20
4	User Extension of GPLP Toolbox	21
4.1	Adding a new covariance function	21
4.2	Adding a new mesh generation function	23
5	Conclusion	26

1 GPLP Toolbox

The GPLP Version 1.0 is a software package written in the **Octave** and **Matlab** to provide the implementation of several localized and parallelized computation algorithms for computing the Gaussian process regression problem with massive amount of data. The algorithms implemented includes the domain decomposition method (Park et al., 2011, DDM), two parallel computation versions of the domain decomposition methods, partial independent conditional (Snelson and Ghahramani, 2007, PIC), localized probabilistic regression (Urtasun and Darrell, 2008, LPR), and bagging for Gaussian process regression (Chen and Ren, 2009, BGP). Most of these methods work in general setting of regression, while DDM only works with spatial datasets having two dimensional predictor variables. The GPLP is released under GNU General Public License version 3.0 (GPL-3.0).

The basic concept of the localized computation methods is based on first decomposing the large amount of data into several smaller chunks and then learning a localized regression function per chunk. Subsequently, the localized regression functions are stitched together or summed with some weightings to form a global regression function. Computing and learning the localized regression functions is as cheap as $O(NM^2)$ in computation, much cheaper than learning the original regression function ($O(N^3)$), where $M \ll N$ is the number of data points in the small local chunk. In addition, the localized computation methods are more adaptive to non-stationary change in data. Therefore, the implementation of the localized methods is helpful to practitioners who want to apply the Gaussian process regression for large-scale datasets. All of these implementations are tested on both of **Octave** 3.2.4 and **Matlab** 7.7.0, so they should be executable in those versions or later versions. One exception is the implementation of LPR that only works in **Matlab** 7.12.0, in **Matlab** 7.7.0 or later versions with a compiler supporting mex-compile, or in **Octave** 3.2.4 or later versions. For information on the list of compilers to support the mex-compile in **Matlab**, please refer to http://www.mathworks.com/support/compilers/previous_releases.html.

The package also includes two implementation versions of the parallel computation of the domain decomposition method, shortly PDDM. The parallel versions have time complexity of $O(M^3)$ so they are much more suitable to handling large datasets. The first implementation of PDDM is based on the parallel computation toolbox of **Matlab**, so it is executable only in **Matlab**. If users do not buy enough number of **Matlab** licenses, they cannot take full advantage of the parallelization using this implementation. The second implementation of PDDM is based on the open source implementation of message passing interface standard, **MatMPI**¹ (Kepner, 2001), and it is executable in both of **Octave** and **Matlab**. Since **Octave** is licensed with free of charge, PDDM can be executed without limitation in the software license, but only limited by the number of hardware processors. We also expect that the second implementation becomes a reference for other researchers or software developers who implement the parallel computation versions of their algorithms in **Matlab** or **Octave**.

The GPML toolbox² (Rasmussen and Nickisch, 2010) has already been available for solving the Gaussian process regression, but none of the localized computation methods is included in the existing toolbox. The

¹ Available at <http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html>

² Available at <http://mloss.org/software/view/263/>

GPLP should be a nice complement to the GPML toolbox.

This documentation has three sections. Section 2 is about how to install the GPML toolbox and it also includes how to set up the parallel computation environment for making PDDM executable with an open-source job scheduler, TORQUE resource manager³. Section 3 describes the general usage of the toolbox with several examples for illustrating the usage. Section 4 is only for advanced users or developers, explaining how to extend the functions of this package through the addition of new functional modules.

2 Installation

The GPLP should be installed in an operating system where all of the following softwares can be installed:

- Any OS-dependent C compiler
- **Matlab** Version 7.7.0 or later versions, or **Octave** Version 3.2.4 or later versions; any **Matlab** Version 7.x or any **Octave** Version 3.2.x might work but **GPLP** has not been tested on those versions.
- (Only for PDDM) **MatMPI** Version 1.2
- (Only for PDDM) **TORQUE** resource manager or any batch workload manager that works in a distributed computing environment

The softwares listed above are required to execute GPLP. The softwares work in most of the operating systems, so GPLP is basically OS-independent.

2.1 Basic Installation

The installation of GPLP consists of three simple steps. First, **Matlab** should be properly configured so that it can compile mex files. **Matlab** supports the use of a variety of compilers for building MEX-files. Users can specify which compiler they want to use. For a list of compilers supported by **Matlab**, please refer to the link at http://www.mathworks.com/support/compilers/previous_releases.html. Once you have verified that you are using a supported C, C++, or FORTRAN compiler, you are ready to configure your system to build MEX-files. In order to do this, run the following command from the MATLAB command prompt:

```
mex -setup
```

When you run this command, a series of questions are asked regarding the location of the C or C++ compiler you would like to use to compile your code. Answering these questions completes the configuration of **Matlab**.

Second, the GPLP package files should be extracted from `gplp.ver1.0.zip` (Windows) or `gplp.ver1.0.tgz` (Unix). The extracted files are organized in the following sub-directories:

- `./` : directory contains all demo files and data files necessary for executing the demo files.
- `./doc` : directory contains this documentation.
- `./cov` : directory contains all source files defining the covariance functions for Gaussian process; this folder is simply a copy of the covariance

³ Available at <http://www.clusterresources.com/downloads/torque/>

function implementations provided as a part of the early version of GPML.

- `./mesh` : directory contains all source files defining the partitioning schemes that decompose the large input data into a number of small chunks for localized computation.
- `./ddm` : directory contains all source files that implement the sequential computation version of the domain decomposition method.
- `./pddm-multicore` : directory contains all source files that implement the parallel version of the domain decomposition method using **Matlab** Parallel Computing Toolbox.
- `./pddm-multinode` : directory contains all source files that implement the parallel version of the domain decomposition method using **MatMPI**.
- `./pic` : directory contains all source files that implement the partial independent conditional.
- `./bpg` : directory contains all source files that implement the bagging for Gaussian process regression.
- `./lpr` : directory contains all source files that implement the localized probabilistic regression.
- `./etc` : directory contains the sample configuration files necessary for setting up the computation environment with **GPLP**.
- `./etc/kdtree` : directory contains the KD-tree package **kdtree1.2**¹.

The last step of the installation is to compile eight mex files in `./mesh`, `./cov`, and `./etc/kdtree` sub-directories. If you are using **Matlab**, please check a list of the compilers that support the mex-compilation before executing the following commands in **Matlab** prompts:

- **If you are using Matlab 7.12.0 or later versions:**

```
>> mex -outdir ./mesh ./mesh/dist.c
>> mex -outdir ./cov ./cov/sq_dist.c
```

- **If you are NOT using Matlab 7.12.0 or later versions:**

```
>> mex -outdir ./mesh ./mesh/dist.c
>> mex -outdir ./cov ./cov/sq_dist.c
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_ball_query.cpp
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_build.cpp
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_delete.cpp
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_k_nearest_neighbors.cpp
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_nearest_neighbor.cpp
>> mex -outdir ./etc/kdtree \
    ./etc/kdtree/kdtree_range_query.cpp
```

The first two commands are the same for using all versions of **Matlab**. But when using a version of **Matlab** earlier than 7.12.0, you need to compile six additional mex files, which are a part of **kdtree1.2** package.

¹ **kdtree1.2** is **Matlab** implementation of kd-tree, which is available at <http://www.mathworks.com/matlabcentral/fileexchange/21512>

In **Matlab** 7.12.0 or later versions, **Matlab** provides the built-in **kdtree** package for our use so that compiling these mex files is no longer needed. In our testing, the Visual Studio 2008 produced a complied version of **kdtree1.2** package to work with **Matlab** 7.7.0. It might work with other versions of Visual Studio and Matlab, but we have not tested it. For the details of installing Visual Studio 2008, please see [http://msdn.microsoft.com/en-us/library/ms246609\(v=VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ms246609(v=VS.90).aspx).

In order to compile the mex files in **Octave**, please use the following commands in **Octave** prompts:

```
octave-3.2.4.exe:1> mkcofile --mex ./mesh/dist.c
octave-3.2.4.exe:2> mkcofile --mex ./cov/sq_dist.c
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_ball_query.cpp
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_build.cpp
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_delete.cpp
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_k_nearest_neighbors.cpp
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_nearest_neighbor.cpp
octave-3.2.4.exe:2> mkcofile --mex \
    ./etc/kdtree/kdtree_range_query.cpp
```

If the mex-compile is successful, the files having ‘.o’ and ‘.mex’ file extensions (for **Octave**), or ‘.mexw32’ or ‘.mexw64’ file extension (for **Matlab**) will be generated. You might see some warnings during the compilation but you can ignore the warnings.

2.2 Setting Up a Parallel Computation Environment

This section explains how to set up the parallel computation environment for executing PDDM with **MatMPI** and a batch workload manager. Before following this section, one should make sure that a high performance cluster of multiple compute nodes has already been built up with a batch workload manager. After that, the set-up of the parallel computation environment is in the following two steps:

- Installing **MatMPI**: The **MatMPI** is a set of **Matlab** scripts that implement a subset of the message passing interface (MPI), which is the de-facto standard for doing point-to-point communications among parallel programs. The parallel computation version of the domain decomposition method (PDDM) is partially based on **MatMPI**. Thus, before running PDDM, one should install **MatMPI** on a directory that is visible to every machine in the cluster. For the installation,

1. Please download **MatlabMPI_v1.2.tar.gz** at <http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html>.
2. Extract the files at the directory where you want to locate **MatMPI**, using the following command in Unix:

```
tar -xzf MatlabMPI_v1.2.tar.gz
```

or using some freewares in Windows to unzip.

- Configuring the startup file: one should configure a startup file of **Matlab** or **Octave** so that **Matlab** or **Octave** can be aware of the installation location of **MatMPI**. The startup file is a file containing the commands to execute when **Matlab** or **Octave** starts. The file's name must be `startup.m` for **Matlab** and `.octaverc` for **Octave**, and the file should locate either in your home directory or in the directory where **Matlab** or **Octave** is started. Before using **GPLP**, the users need to add **MatlabMPI/src** directory to the **Matlab** or **Octave** library path by adding the following line in the startup file:

```
addpath('[MatMPI install directory name]/src')
```

The sample startup files are found at `./etc` directory with their names of 'startup.m' or '.octaverc'.

3 Usage and Demonstration

This section presents the general usage of **GPLP** and the seven different usage cases.

3.1 General Usage

Running several different methods in **GPLP** involves calling the twelve different functional modules from six different methods, two modules per each method (one for training and one for prediction). However, the two modules for every method are in the following common forms:

```
[model, train, test, mu, s2, mse, nlpd]
    = training_module_name(x, y, param1, param2,..., xs, ys)
[mu,s2, elapsed] = prediction_module_name(xs, model)
```

training_module_name: This function trains the Gaussian process regression model with training data **x**, **y**, and it returns **model** output parameter as the learned regression model. There is a naming convention for this function; the name should start with the method short name and should end with 'GP'. For example, if the method is the domain decomposition method, the method short name is 'ddm' so the name of the training function should be 'ddmGP'. The following arguments can be used in **training_module_name**:

- **x**: n training inputs ($n \times d$ matrix)
- **y**: n training targets (a column vector of length n)
- **paramx** (e.g. **param1**, **param2**,...): any additional parameters necessary for training. The number of parameters can be more than or equal to one. Each **paramx** is a **structure** data type in **Matlab**, so it has multiple parameter elements. The contents in each **paramx** varies depending on the method used. For more details, please refer to the examples presented in Section 3.2 through 3.8.
- **xs**: (Optional) ns test inputs ($ns \times d$ matrix); if this argument is specified, **training_module_name** function will perform the prediction on the test inputs using the trained regression model and the function will return the predictive mean **mu** and the predictive variance **s2**.
- **ys**: (Optional) ns test targets (a column vector of length ns); if this argument and **xs** are specified, **training_module_name**

function will perform the prediction on the test inputs **xs** using the trained regression model, and it will compute and return the mean squared error **mse** and the negative log predictive density **nlpd**.

- **model**: trained regression model
- **train**: time elapsed for training
- **test**: time elapsed for testing; only available if **xs** is specified
- **mu**: predictive mean at **xs**; only available if **xs** is specified
- **s2**: predictive variance at **xs**; only available if **xs** is specified
- **mse**: mean squared error; only available if **xs** and **ys** are specified
- **nlpd**: negative log predictive density defined as

$$\text{nlpd} = \frac{1}{ns} \sum_{i=1}^{ns} \frac{(ys_i - \mu_i)^2}{2s2_i} + \frac{1}{2} \log(2\pi s2_i),$$

where ys_i, μ_i and $s2_i$ are the i th element of **ys**, **mu** and **s2** respectively. **nlpd** is only available when **xs** and **ys** are specified.

prediction_module_name: This function uses the trained regression model, which can be obtained by calling **training_module_name**, for performing the prediction at test inputs **xs**. The name of **prediction_module_name** is always 'short method name' + '_pred' (+ postfix). For example, if the method name is the domain decomposition method and we execute its parallel version in multi-core environment, the function named 'ddm_pred_mc' should be called. The following arguments can be used in **prediction_module_name**:

- **model**: trained regression model
- **xs**: ns test inputs ($ns \times d$ matrix)
- **mu**: predictive mean at **xs**
- **s2**: predictive variance at **xs**
- **elapsed**: time elapsed for prediction

3.2 Case 1: Execution of Domain Decomposition Method with Local Hyperparameters

This example illustrates applying the domain decomposition method for predicting the total column of ozone at unobserved locations.

The dataset used here contains data collected by NIMBUS-7/TOMS satellite to measure the total column of ozone over the globe on Oct 1 1988. This set consists of 48,331 measurements, and each measurement consists of latitude, longitude and the total column of ozone value at the latitude / longitude. The dataset files are entitled with **ozone_x.csv** and **ozone_y.csv** in **./** directory. We first load the dataset files into the memory space using the following command:

```
1 clc
2 clear
3
4 % add library path
```

```

5 addpath('./ddm')
6 addpath('./mesh')
7 addpath('./cov')
8 addpath('./lik')
9 addpath('./etc')
10
11 % load data & subsample to a training dataset
12 x = csvread('ozone_x.csv');
13 y = csvread('ozone_y.csv');
14 [dum,I] = sort(rand(size(x,1),1)); clear dum;
15 t_idx = ones(size(x,1),1); t_idx(I(1:round(size(x,1) ...
    * 0.1))) = 0;
16 t_idx = logical(t_idx);
17
18 xs = x(~t_idx, :); ys = y(~t_idx, :);
19 x = x(t_idx, :); y = y(t_idx, :);

```

The line 5 through 8 is for loading the common library files to execute the domain decomposition method. The line 11 through 18 load two csv files into **x** and **y** variables, and the lines split the variables into a training dataset (90% of the whole dataset) and a test dataset (10%).

Before training the Gaussian process regression model with the domain decomposition method, one should define several parameters for the training module. The domain decomposition method has two groups of parameters: domain decomposition parameter group **dd_param** and covariance function parameter group **cv_param**. Each parameter group is typed as a **Matlab** structure data-type, containing the following sub-parameter values.

- **dd_param.meshfunc**: the string name of the function used to partition the big training dataset into many smaller chunks (by decomposing the domain of regression function into small subdomains and then by splitting the training data points according to which subdomain they belong to). Such function is called a *mesh generation function*. The mesh generation functions available are **rectMesh** and **rectGrid**. Users can define their own mesh generation function (please refer to Section 4.2).
- **dd_param.mparam**: input parameters for meshfunc (e.g. the size of subdomains)
- **dd_param.p**: degrees of freedom for representing each boundary function
- **dd_param.q**: the number of locations to check the continuity of predictions on a boundary
- **cv_param.covfunc**: prior covariance function. There are many covariance functions available, all start with the three letters **cov** and reside in the **./cov** directory. Users can write their own covariance function (please refer to Section 4.1).
- **cv_param.local**: (optional) 1: use the local hyperparameters; default, 0: use the global one
- **cv_param.frachyper**: (optional) the fraction of the training dataset used for learning the hyperparameters (range: 0.0 - 1.0, default: 1.0)
- **cv_param.nIter**: (optional) the maximum number of iterations for optimizing the hyperparameters (default: 50)

- `cv_param.logtheta`: (optional) used as hyperparameters of the prior covariance function. If it is specified, the hyperparameter learning step is skipped.
- `cv_param.logtheta0`: (optional) initial guess of hyperparameters of the prior covariance function

In this example, we use `rectGrid` as the mesh generation function such that each mesh is sized as 14-by-21 grid points. The degree of freedom for the boundary function and the number of locations to check the boundary continuity are set to three. We use the sum of the squared exponential covariance and the noise covariance as our prior covariance function and let the hyperparameters of the function be learned using only 60% of the training dataset with some initial guess `logtheta0`. The corresponding source code for the parameter specification is as follows:

```

1
2 % set the input parameters regarding domain decomposition
3 dd_param.meshfunc = 'rectGrid'; % mesh generation function
4 dd_param.mparam   = [14 21]; % mesh generation ...
5 % function parameters
6 dd_param.p       = 3; % (mesh size)
7 % boundary element
8 dd_param.q       = 3; % degree of freedom for ...
9 % the number of ...
10 % locations to check the
11 % continuity of ...
12 % prediction over boundary
13
14 % set the input parameters regarding prior covariance function
15 % cv_param.covfunc: prior covariance function = squared ...
16 % covariance function
17 % + noise covariance function
18 cv_param.covfunc = {'covSum', {'covSEard', 'covNoise'}};
19 d               = size(x, 2);
20 logtheta0       = 0*ones(d+2,1); % starting values of log ...
21 % hyperparameters:
22 % log ...
23 % (\lambda_1,...,\lambda_d, ...
24 % rho)
25 % for square covariance ...
26 % function
27 logtheta0(d+2) = -1.15; % starting value for log(noise ...
28 % std dev): log
29 % sigma^2 for noise covariance ...
30 % function
31 cv_param.logtheta0 = logtheta0;
32 % fraction of training data used for learning hyperparameters
33 cv_param.frachyper = 0.6;

```

Last, we call the main training module named `ddmGP` to train the Gaussian process regression model based on the domain decomposition method. Two modes are possible — training or prediction: if no test input is provided, the training module fits the hyperparameters of the covariance function and does some pre-computation necessary for prediction. If test inputs are given, then the predictive mean and variance at the test inputs are returned. Usage:

```

training: [model train test
          = ddmGP(x,y, dd_param, cv_param);
prediction: [model train test mu s2
           = ddmGP(x,y, dd_param, cv_param, xs);
           or: [model train test mu s2 mse nlpd]

```

```
= ddmGP(x,y, dd_param, cv_param, xs, ys);
```

In this example, we use the third mode to produce the predictive mean, variance, mean squared error and the negative log predictive density as follows:

```
1 % execute the main logic
2 [model, train, test, mu, s2, mse, nlpd] = ddmGP(x, y, ...
    dd_param, cv_param, xs, ys);
3 fprintf('training time = %-8.4f seconds\n', train);
4 fprintf('test time = %-8.4f seconds\n', test);
5 fprintf('mse = %-8.4f\n', mse);
6 fprintf('nlpd = %-8.4f\n', nlpd);
```

The code of this example can be found at the `./` directory (please find `demo_ddm.m` file). The execution of the example will print the time elapsed by training, the time elapsed by testing, the mean squared error and the negative log predictive density in the program console as the summary result.

3.3 Case 2: Execution of Domain Decomposition Method with Global Hyperparameters

This example shows how to solve the same problem as Case 1 in the previous section with the hyperparameters learned globally, while the hyperparameter values were learned separately for each mesh in Case 1. To allow the training module to know that the hyperparameters should be learned globally, one should specify the `cv_param.local` parameter value as follows:

```
cv_param.local = 0;
```

The default value of the parameter is 1 so if it is not given, the training module learns the hyperparameters for each mesh.

The rest of the example code is the same as Case 1, available as `demo_ddm_global.m` at `./` directory. The execution of the example will print the time elapsed by training, the time elapsed by testing, the mean squared error and the negative log predictive density in the program console as the summary result. The mean squared error of this example is higher than that of Case 1 example because the hyperparameters learned globally in this example do not reflect underlying local variabilities in the dataset.

3.4 Case 3: Parallel Execution of Domain Decomposition Method in Single-Machine-Multi-Core Environment

This example illustrates the use of a parallel computation version of the domain decomposition method for handling larger datasets in a multi-core machine. Note that this example works only with **Matlab**, not **Octave**, because the multi-core version of DDM is partially based on the **Matlab** Parallel Computing Toolbox. In this example, we use two different datasets: one for a training data and the other for a test data. The dataset for training is the remote sensing data collected by NIMBUS-7/TOMS satellite to measure the total column of ozone over the globe on Oct 1 1988. The dataset is much closer to raw satellite data than what was used in Case 1, and the number of the measurements is 182,591, which is much larger than the dataset used in Case 1. The files storing the dataset are `ozone_L2_x.csv` (training input file) and `ozone_L2_y.csv` (training target file) and both are located in `./` directory.

There is a separate test dataset, which is the total column of ozone values measured at 89 ground stations. The files storing the test dataset are `ozone_gr_x.csv` (ground station location) and `ozone_gr_y.csv` (measurement) in `./` directory. We will use the training dataset to learn the Gaussian process regression model and will perform the prediction at where the ground stations locate. Finally, we will compare the prediction to the direct measurement from the ground stations. Such comparison is widely used in geo-statistics to verify the measurements of ground station using satellite data or vice versa.

We first need to set up the library path and to initiate the **Matlab** process pool for parallel computations, using the commands in the bottom box. In line 11, we initiated eight processes in the process pool, which is the maximum number allowed for the local scheduler (the number is four if you use **Matlab** Version 7.7.0 or the older version). Please note that adding `./pddm_multicore` in the library path (line 5) is necessary.

```

1 % add library path
2 addpath('./pddm-multicore')
3 addpath('./ddm')
4 addpath('./mesh')
5 addpath('./cov')
6 addpath('./lik')
7 addpath('./etc')
8
9 % open pool of MATLAB sessions for parallel computation
10 if matlabpool('size') == 0
11     matlabpool open 8
12 end

```

Next, we load the four data files for the training and test datasets.

```

1 % load data & subsample to a training dataset
2 x = csvread('ozone_L2_x.csv');
3 y = csvread('ozone_L2_y.csv');
4 xs = csvread('ozone_gr_x.csv');
5 ys = csvread('ozone_gr_y.csv');

```

Third, we define meshes for decomposing the large chunk of the training data into many small chunks based on the spatial locations of data. Different from Case 1 and 2, we use `rectMesh` as the mesh generation function instead of `rectGrid`. The `rectMesh` is more appropriate to decomposing spatially irregular data, while the `rectGrid` is more appropriate for spatially regular data (e.g. gridded data). The remote sensing data used in this example is spatially irregular so we use `rectMesh` function, setting its parameters such that it produces 30-by-20 meshes totally. The source code for meshing is as follows:

```

1 % set the input parameters regarding domain decomposition
2 dd.param.meshfunc = 'rectMesh'; %mesh generation function
3 dd.param.mparam = [30 20]; %mesh generation ...
4     function parameters
5                                     % (the total number of ...
6                                     meshes per each
7 dd.param.p = 2; % spatial dimension)
8     boundary element %degree of freedom for ...
9 dd.param.q = 2; %the number of locations ...
10     to check the

```

```

8                                     %continuity of ...
                                     prediction over boundary

```

We also need to specify the covariance function parameters for Gaussian process. We can do that in the same way as what we did in Case 1.

```

1 % set the input parameters regarding prior covariance function
2 cv_param.covfunc = {'covSum', {'covSEard', 'covNoise'}}; % ...
  prior covariance function
3 d = size(x, 2);
4 logtheta0 = 0*ones(d+2,1); % starting values ...
  of log hyperparameters: log (\lambda_{d+1}, ..., \lambda_d, ...
  rho) for Square Covariance Function
5 logtheta0(d+2) = -1.15; % starting value ...
  for log(noise std dev): log sigma^2 for Noise ...
  Covariance Function
6 cv_param.logtheta0 = logtheta0;
7 cv_param.frachyper = 0.2; % fraction of ...
  training data used for learning hyperparameters

```

Finally, we train the Gaussian process regression model using the multicore version of the domain decomposition method. The name of the training module is `ddmGP_mc`. To illustrate how to use the trained model for future prediction, we called the prediction module separately from the training module. The function name of the prediction module is `ddm_pred_mc`.

```

1 % train the GP model
2 [model, train] = ddmGP_mc(x, y, dd_param, cv_param);
3 fprintf('training time = %-8.4f seconds\n', train);
4
5 % test the model
6 [mu, s2, test] = ddm_pred_mc(xs, model);
7
8 %compute MSE & NLPD
9 ns = size(xs,1);
10 se = (ys - mu)' * (ys - mu);
11 nlpd = sum(0.5*(log(2*pi*s2) + ((ys - mu).^2)./s2));
12 mse = se / ns;
13 nlpd = nlpd / ns;
14
15 fprintf('training time = %-8.4f seconds\n', train);
16 fprintf('test time = %-8.4f seconds\n', test);
17 fprintf('mse = %-8.4f\n', mse);
18 fprintf('nlpd = %-8.4f\n', nlpd);

```

The last four lines print out the summary result. The source code file for this example is `demo_ddm_multicore.m` located at `./` directory.

3.5 Case 4: Parallel Execution of Domain Decomposition Method in Multi-Machine Environment

This example illustrates how to execute the domain decomposition method in a high performance cluster of multiple machines (HPC). Before running this example, please make sure that `MatMPI` is properly installed as described in Section 2.2. We also assume that the cluster of the user system uses the `TORQUE`² resource manager as a batch workload

² The `TORQUE` resource manager is a open-source batch job scheduler and a workload manager that distributes parallel computation tasks over computation nodes in parallel computing environment. The resource manager is an open-source implementation of a de-facto standard interface for the batch job scheduler, `OpenPBS`,

manager, but this does not imply that the implementation of the domain decomposition method works only with the **TORQUE** resource manager. Indeed, it is compatible with every batch workload manager installed in the user system.

To execute the domain decomposition method in HPC, users need to write three script files: (1) a PBS (Portable Batch System) script file for submitting a parallel job to **TORQUE** resource manager, (2) a **Matlab** or **Octave** script file for initiating the **MatMPI** over multiple machines, and (3) a **Matlab** or **Octave** script file for describing the job to be performed.

The example PBS script file is at `./` directory with name `pbs_multinode.pbs`. The example file is launching **Octave** to execute the domain decomposition method as follows.

```
#!/bin/sh
#PBS -l walltime=00:40:00
#PBS -q iamcs
#PBS -l nodes=2:ppn=8
#PBS -j oe
#PBS -N octavempi

Nprocs=16

source ~/bashrc

### go to your working directory
cd /home/chiwoo.park/pddm

### copy the node list file to your working directory
cp $PBS_NODEFILE ./machines.m

### run the program
time /apps/octave-3.2.4/bin/octave -q < main_multinode.m >& ddm.out
### time /apps/matlab/R2010b/bin/matlab -nojvm -nosplash -nodisplay -nodes
###                                     main_multinode.m >& ddm.out
```

In the PBS script, the line 2 to 6 are for defining PBS job script options. The description about the options is as follows:

- `#PBS -N jobname`: Assign a name to job
- `#PBS -l walltime=runtime`: Set wallclock time limit (format = hh:mm:ss)
- `#PBS -q queue name`: Specify job queue to be used
- `#PBS -l nodes=node number:ppn=process per node`: Specify the total number of nodes (machines) used and the total number of processes evoked per node.
- `#PBS -k oe`: Keep the standard output file and error log file in your home directory while the job runs.

In this example, we are invoking eight processes in a single machine node. Next, the PBS script has a couple of lines for specifying the environment

so it is widely used as a job scheduler for building parallel grids. For this reason, this section explains how to set up the environment when we use **TORQUE** resource manager as the batch job scheduler, but it does not imply that **TORQUE** resource manager is the only option for the batch job scheduler.

variables of `MatMPI` (line 8-11), and the script changes the working directory (line 14). The PBS script is then storing `$PBS_NODEFILE` environment variable into `./machines.m` file so that we later see which machines are involved in the parallel job (line 17). Finally, the script let `Octave` execute `main_multinode.m` script file, recording the output into `ddm.out` file.

The `main_multinode.m` script file is the `Matlab` (compatible with `Octave`) script file for initiating the `MatMPI` over multiple machines. The file packaged in `GPLP` is written such that it internally executes `demo_ddm_multinode.m` over multiple machines. The `main_multinode.m` file should look as follows:

```
1 addpath('./pddm-multinode');
2
3 %This script is for running main.m (matlabmpi) with batch mode
4 % Read the node list file into a array
5 machines = readnodes('machines.m');
6
7 % Get the length of node list
8 node_num = size(machines, 2);
9
10 % Clean up matlabmpi
11 MatMPI.Delete_all
12
13 % Run your program
14 eval(MPI_Run('demo_ddm_multinode',node_num, machines));
```

The script `demo_ddm_multinode.m` is the job description file for this example. If users want to execute a different job, they must write their own `Matlab` or `Octave` script file for their purpose. After that, they should change the last line of `main_multinode.m` such that the newly written script file is executed as follows:

```
1 eval(MPI_Run('your script file name',node_num, machines));
```

The job description script in this example, `demo_ddm_multinode.m`, performs the exactly same job as what we did in Case 3 (, but based on `MatMPI`). First, let the `MatMPI` channel connect every machines involved in the parallel job as follows:

```
1 global comm; % communication channel
2 global my_rank
3
4 % Initialize MPI (Message Passing Interface) for MATLAB or ...
   Octave.
5 MPI_Init;
6
7 % Create communicator.
8 comm = MPI_COMM_WORLD;
9
10 % Get rank.
11 my_rank = MPI_Comm_rank(comm);
```

Next, we should load the data files and perform the subsampling on the training dataset such that only a small sample will be used for learning the covariance hyperparameters. Please note that the following function should be used to generate the subsamples, instead of directly calling `Matlab` sampling functions (e.g `randi`):

```
[indices] = gen_random_subset(N, fracs);
```

The function generates the subsample of $\{1, 2, \dots, N\}$ with size `fracs` × `N`, where `fracs` is a value between 0 and 1. If `fracs` parameter is a list of values, the function generates as many subsamples as the size of the list. In this example, we will use 20% of the training dataset for learning the hyperparameters so we have the following code:

```
1 % load data
2 disp('loading data files...');
3 x = csvread('ozone_L2.x.csv');
4 y = csvread('ozone_L2.y.csv');
5 xs = csvread('ozone_gr.x.csv');
6 ys = csvread('ozone_gr.y.csv');
7
8 % Subsample the training dataset for efficient ...
   hyperparameter learning
9 % Make sure that every node has the same training/test ...
   dataset as well as
10 % the same subsample used for the hyperparameter learning.
11 disp('generating a random subset used for learning ...
   hyperparameters...');
```

The specification of `dd_param` and `cv_param` is the same as that of Case 3 except specifying an additional parameter `cv_param.idx` as:

```
1 cv_param.idx = idx; % the row indices of the training ...
   dataset, which corresponds to a subset of the training ...
   dataset used for learning the hyperparameters
```

Last, we call `ddmGP_mn`, which is the main training module for the MatMPI-based implementation of the domain decomposition method. We also need to close the MatMPI channel.

```
1 cv_param.idx      = idx;
2
3 disp('starting the main logic...');
4 % execute the main logic
5 if my_rank == 0
6     [model, train, test, mu, s2, mse, nlpd] = ddmGP_mn(x, ...
   y, dd_param, cv_param, xs, ys);
7     fprintf('training time = %-8.4f seconds\n', train);
8     fprintf('test time = %-8.4f seconds\n', test);
9     fprintf('mse = %-8.4f\n', mse);
10    fprintf('nlpd = %-8.4f\n', nlpd);
11 else
12    ddmGP_mn(x, y, dd_param, cv_param, xs, ys);
13 end
14
15 % Finalize Matlab MPI.
16 MPI_Finalize;
17 disp('SUCCESS');
```

The last script file of this example can be found at the `./` directory (please find `demo_ddm_multinode.m` file).

3.6 Case 5: Execution of Partial Independent Conditional

This example shows how to run the partial independent conditional (PIC) method with the ozone dataset used in Case 1. The code for loading the data files is in the same way as Case 1.

Before training the Gaussian process regression model with PIC, one should add `./pic` to the library path and needs to define several parameters for the training module. The PIC has a group of parameters: `param`. The parameter group is typed as a `Matlab` structure data-type, containing the following sub-parameter values.

- `param.M`: the number of pseudo inputs
- `param.K`: the number of local regions
- `param.logtheta0`: initial guess of hyperparameters of the prior covariance function
- `param.frachyper`: (optional) the fraction of the training dataset used for learning the hyperparameters (range: 0.0 - 1.0, default: 1.0)
- `param.nIter`: (optional) the maximum number of iterations for optimizing the hyperparameters (default: 50)

In this example, we learn PIC with 200 pseudo inputs and 300 local regions. The sum of the squared exponential covariance and the noise covariance is only allowed for the prior covariance function in PIC so we do not need to specify the covariance function. We let the hyperparameters of the function be learned using only 60% of the training dataset with some initial guess `logtheta0`. The corresponding source code for the parameter specification is as follows:

```

1 % starting values of log hyperparameters:
2 logtheta0(1:d,1) = -2*log((max(x)-min(x))'/2); % log ...
   1/(lengthscales)^2
3 logtheta0(d+1,1) = log(var(y,1)); % log size
4 logtheta0(d+2,1) = log(var(y,1)/4); % log noise
5 param.logtheta0 = logtheta0;
6
7 % fraction of training data used for learning hyperparameters
8 param.frachyper = 0.6;
9 param.M         = 200;
10 param.K         = 300;
```

Last, we call the main training module named `picGP` to train the Gaussian process regression model based on PIC. Two modes are possible: training or prediction: if no test input is provided, the training module fits the hyperparameters of the covariance function and does some pre-computation necessary for prediction. If test inputs are given, then the predictive mean and variance at the test inputs are returned. Usage:

```

training: [model train test          ]
           = picGP(x,y, param);
prediction: [model train test mu s2    ]
            = picGP(x,y, param, xs);
            or: [model train test mu s2 mse nlpd]
                 = picGP(x,y, param, xs, ys);
```

In this example, we use the third mode to produce the predictive mean, variance, mean squared error and the negative log predictive density as follows:

```

1 % execute the main logic
2 [model, train, test, mu, s2, mse, nlpd] = picGP(x, y, ...
   param, xs, ys);
```

```

3 fprintf('training time = %-8.4f seconds\n',train);
4 fprintf('test time = %-8.4f seconds\n',test);
5 fprintf('mse = %-8.4f\n',mse);
6 fprintf('nlpd = %-8.4f\n',nlpd);

```

The code of this example can be found at the `./` directory (please find `demo_pic.m` file). The execution of the example will print the time elapsed by training, the time elapsed by testing, the mean squared error and the negative log predictive density in the program console as the summary result.

3.7 Case 6: Execution of Localized Probabilistic Regression

This example shows how to run the localized probabilistic regression (LPR) method with the ozone dataset used in Case 1. The code for loading the data files is in the same way as Case 1.

Before training the Gaussian process regression model with LPR, one should add `./lpr` to the library path and needs to define several parameters for the training module. The LPR has a group of parameters: `param`. The parameter group is typed as a `Matlab` structure data-type, containing the following sub-parameter values.

- `param.T`: the number of local experts used for prediction; the prediction at a test input is obtained by taking the weighted average of `param.T` local expert's predictions.
- `param.R`: the number of random sites chosen for learning local hyperparameters; if a test input is given, the LPR dynamically generates `param.T` local experts which are closest to the test input. Since learning hyperparameters for the dynamically chosen local experts per test input is too computationally expensive, the LPR randomly chooses `param.R` sites so that they distribute uniformly over the whole input space, and it learns one set of hyperparameters using the neighborhood of each random site. When a test input is given, the LPR finds the random site closest to the test input and uses the hyperparameter learned at the chosen random site.
- `param.S`: the number of training inputs allocated to each local expert
- `cv_param.covfunc`: prior covariance function. There are many covariance functions available, all start with the three letters `cov` and reside in the `./cov` directory. Users can write their own covariance function (please refer to Section 4.1).
- `param.logtheta0`: initial guess of hyperparameters of the prior covariance function
- `param.frachyper`: (optional) the fraction of the training dataset used for learning the hyperparameters (range: 0.0 - 1.0, default: 1.0)
- `param.nIter`: (optional) the maximum number of iterations for optimizing the hyperparameters (default: 50)

In this example, we learn LPR with 20 local experts, 200 training inputs allocated per each expert and 1000 random sites chosen for learning hyperparameters. The sum of the squared exponential covariance and the noise covariance is used. We let the hyperparameters of the function be learned using only 50% of the training dataset (i.e. set

`param.frachyper` to its default value), and the hyperparameters are optimized with `logtheta0` as the initial values through maximally 100 iterations of the optimization. The corresponding source code for the parameter specification is as follows:

```

1
2 % set the input parameters
3 param.covfunc = {'covSum', {'covSEard','covNoise'}}; % ...
    prior covariance function
4 d = size(x, 2);
5 logtheta0 = 0*ones(d+2,1); % starting values of ...
    log hyperparameters:
6 %                                     log ...
    (\lambda_1,...,\lambda_d, rho) for
7 %                                     Square Covariance ...
    Function
8 logtheta0(d+2) = -1.15; % starting value for log(noise ...
    std dev):
9                                     %log sigma^2 for Noise ...
                                     Covariance Function
10 param.logtheta0 = logtheta0;
11 param.T = 20; % number of local experts
12 param.R = 1000; % number of random sites chosen ...
    for local
13 % hyperparameter learning
14 param.S = 200; % number of training inputs ...
    allocated to each local
15 % expert

```

Last, we call the main training module named `lprGP` to train the Gaussian process regression model based on LPR. Two modes are possible: training or prediction: if no test input is provided, the training module fits the hyperparameters of the covariance function and does some pre-computation necessary for prediction. If test inputs are given, then the predictive mean and variance at the test inputs are returned. Usage:

```

training: [model train test          ]
           = lprGP(x,y, param);
prediction: [model train test mu s2    ]
            = lprGP(x,y, param, xs);
            or: [model train test mu s2 mse nlpd]
                 = lprGP(x,y, param, xs, ys);

```

In this example, we use the third mode to produce the predictive mean, variance, mean squared error and the negative log predictive density as follows:

```

1
2 % execute the main logic
3 [model, train, test, mu, s2, mse, nlpd] = lprGP(x, y, ...
    param, xs, ys);
4 fprintf('training time = %-8.4f seconds\n',train);
5 fprintf('test time = %-8.4f seconds\n',test);
6 fprintf('mse = %-8.4f\n',mse);

```

The code of this example can be found at the `./` directory (please find `demo_lpr.m` file). The execution of the example will print the time elapsed by training, the time elapsed by testing, the mean squared error and the negative log predictive density in the program console as the summary result.

3.8 Case 7: Execution of Bagging for Gaussian Process Regression

This example shows how to run the bagging method for Gaussian process regression (BGP) with the ozone dataset used in Case 1. The code for loading the data files is in the same way as Case 1.

Before training the Gaussian process regression model with BGP, one should add `./bgp` to the library path and needs to define several parameters for the training module. The BGP has a group of parameters: `param`. The parameter group is typed as a `Matlab` structure data-type, containing the following sub-parameter values.

- `param.M`: the size of each bagging sample
- `param.K`: the total number of bagging samples
- `cv_param.covfunc`: prior covariance function. There are many covariance functions available, all start with the three letters `cov` and reside in the `./cov` directory. Users can write their own covariance function (please refer to Section 4.1).
- `param.logtheta0`: initial guess of hyperparameters of the prior covariance function
- `param.frachyper`: (optional) the fraction of the training dataset used for learning the hyperparameters (range: 0.0 - 1.0, default: 1.0)
- `param.nIter`: (optional) the maximum number of iterations for optimizing the hyperparameters (default: 50)

In this example, we learn BGP with 40 bagging samples and 300 training inputs in each bagging sample. The sum of the squared exponential covariance and the noise covariance is used. We let the hyperparameters of the function be learned using only 50% of the training dataset (i.e. set `param.frachyper` to its default value), and the hyperparameters are optimized with `logtheta0` as the initial values through maximally 100 iterations of the optimization. The corresponding source code for the parameter specification is as follows:

```
1
2 % set the input parameters
3 param.covfunc = {'covSum', {'covSEard','covNoise'}}; % ...
   prior covariance function
4 d             = size(x, 2);
5 logtheta0     = 0*ones(d+2,1); % starting values of ...
   log hyperparameters:
6 %                                     log ...
   (\lambda_1,...,\lambda_d, rho) for
7 %                                     Square Covariance ...
   Function
8 logtheta0(d+2) = -1.15; % starting value for log(noise ...
   std dev):
9                                     %log sigma^2 for Noise ...
                                     Covariance Function
10 param.logtheta0 = logtheta0;
11 param.frachyper = 0.5; % fraction of training data used ...
   for learning
12                                     % hyperparameters
13 param.M         = 300;
```

Last, we call the main training module named `baggingGP` to train the Gaussian process regression model based on BGP. Two modes are possi-

ble: training or prediction: if no test input is provided, the training module fits the hyperparameters of the covariance function and does some pre-computation necessary for prediction. If test inputs are given, then the predictive mean and variance at the test inputs are returned. Usage:

```
training: [model train test          ]
           = baggingGP(x,y, param);
prediction: [model train test mu s2    ]
            = baggingGP(x,y, param, xs);
            or: [model train test mu s2 mse nlpd]
                = baggingGP(x,y, param, xs, ys);
```

In this example, we use the third mode to produce the predictive mean, variance, mean squared error and the negative log predictive density as follows:

```
1
2 % execute the main logic
3 [model, train, test, mu, s2, mse, nlpd] = baggingGP(x, y, ...
   param, xs, ys);
4 fprintf('training time = %-8.4f seconds\n',train);
5 fprintf('test time = %-8.4f seconds\n',test);
6 fprintf('mse = %-8.4f\n',mse);
```

The code of this example can be found at the `./` directory (please find `demo_bgp.m` file). The execution of the example will print the time elapsed by training, the time elapsed by testing, the mean squared error and the negative log predictive density in the program console as the summary result.

4 User Extension of GPLP Toolbox

The intended reader of this section is advanced users of GPLP or software developers who want to extend the basic functions of GPLP. The GPLP toolbox supports the package extension interfaces for users to add a new covariance function or to add a new mesh generation function. The interface to add a new covariance function is necessary because the flexibility and property of a covariance function determines the quality of the solution of the Gaussian process regression. In addition, for the localized computation methods, specifying the mesh generation function to define the appropriate local regions can be very critical in terms of computation efficiency and accuracy. The next two subsections are the ‘How-to’ style documentation of the package extension interfaces for software developers.

4.1 Adding a new covariance function

A Gaussian process f is a stochastic process $\{f(x) : \mathbf{x} \in \mathcal{X}\}$, any finite samples of which will be distributed as multivariate Gaussian. The Gaussian process is fully defined by the pointwise mean function and the covariance function. In solving the Gaussian process regression, the mean function is usually assumed to be zero (simple kriging) and the choice of the covariance function is important to the quality of solution. The GPLP allows users to use various parametric covariance functions which were originally available at the early version of GPML. In addition, the GPLP provides the software interface to allow users to define their own covariance functions. This section will present how to define a new covariance function.

A covariance function is a (positive definite) scalar function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$, which maps a pair of two domain variables to the covariance between the two realizations of f at the two domain variables: for $\mathbf{x}, \mathbf{z} \in \mathcal{X}$,

$$K(\mathbf{x}, \mathbf{z}) = \text{Cov}(f(\mathbf{x}), f(\mathbf{z})).$$

The covariance function K is mostly defined as a parametric form that depends on a set of hyperparameters $\boldsymbol{\theta}$. For example, the anisotropic version of the squared exponential covariance function is defined as:

$$K_{se}(\mathbf{x}, \mathbf{z}; \boldsymbol{\theta}) = \theta_{D+1}^2 \exp \left\{ -\frac{1}{2} \sum_{d=1}^D \left(\frac{x_d - z'_d}{\theta_d} \right)^2 \right\},$$

where D is the dimension of \mathcal{X} , $\mathbf{x} = (x_1, \dots, x_D)^t$, and $\boldsymbol{\theta} = (\theta_1, \dots, \theta_D, \theta_{D+1})^t$.

If you want to define a new covariance function for GPLP, you should write a new `Matlab` function which supports all of the following four modes:

```
[H]      = covName()
[A]      = covName(loghyper, x)
[R]      = covName(loghyper, x, i)
[v, B]   = covName(loghyper, x, z)
```

- **H** is the total number of hyperparameters required by evaluating the covariance function; it is a string formatted number including a special character 'D', which is the dimension of the domain variable (i.e. dimension of \mathcal{X}). For example, if '(D+1)' is returned as the value of H , the number of the hyperparameters required is the dimension (of the domain variable) plus one.
- **loghyper** is the H -by-1 column vector of $\log(\theta_1), \log(\theta_2), \dots, \log(\theta_H)$.
- **x** is the m -by- D matrix of m domain variables.
- **z** is the n -by- D matrix of n domain variables.
- **A** is the m -by- m covariance matrix with A_{ij} as the covariance value between the i th and j th domain variables in **x**.
- **i** is the integer index pointing an element in **loghyper**, with respect to which the partial derivative of the training covariance matrix (**A**) is computed.
- **R** is the m -by- m matrix of partial derivatives of the training covariance matrix (**A**) with respect to the i th element of **loghyper**.
- **v** is the n -dimensional column vector of the self covariances for **z**, i.e. v_i is the covariance between \mathbf{z}_i and \mathbf{z}_i .
- **B** is the m -by- n matrix of cross covariances between **x** and **z**.

For example, we present how to define the anisotropic version of the squared exponential covariance function. We named the function `covSEard`. The following line is the first line of the function definition:

```
1 function [A, B] = covSEard(loghyper, x, z)
```

The function should return string '(D+1)' if there is no input according to the convention of the covariance function (first mode).

```

1  if nargin == 0, A = '(D+1)'; return; end           % report ...
    number of parameters

```

If there are one or more inputs, the function computes the several types of covariance functions or their partial derivatives:

```

1  persistent K;
2
3  [n D] = size(x);
4  ell = exp(loghyper(1:D)); % ...
    characteristic length scale
5  sf2 = exp(2*loghyper(D+1)); ... % signal variance
6
7  if nargin == 2
8      K = sf2*exp(-sq_dist(diag(1./ell)*x')/2);
9      A = K;
10 elseif nargin == 2 % compute ...
    test set covariances
11     A = sf2*ones(size(z,1),1);
12     B = sf2*exp(-sq_dist(diag(1./ell)*x',diag(1./ell)*z')/2);
13 else % ...
    compute derivative matrix
14
15     % check for correct dimension of the previously ...
    calculated kernel matrix
16     if any(size(K)~=n)
17         K = sf2*exp(-sq_dist(diag(1./ell)*x')/2);
18     end
19
20     if z <= D % ...
    length scale parameters
21         A = K.*sq_dist(x(:,z)'/ell(z));
22     else % ...
    % magnitude parameter
23         A = 2*K;
24         clear K;
25     end
26 end

```

The line 8 and 9 computes the m -by- m covariance matrix between \mathbf{x} and \mathbf{x} , and the line 11 and 12 computes the self covariances for \mathbf{z} , and the cross covariance matrix between \mathbf{x} and \mathbf{z} . The remaining lines are for computing the partial derivative of the training covariance matrix.

The example file explained is `covSEard.m` at `./cov` directory. Additional explanation about ‘How to use covariance functions’ can be found at `covFunctions.m` at the same directory.

4.2 Adding a new mesh generation function

This section describes how to define a new mesh generation function for the use of the domain decomposition method.

The domain decomposition method partitions the domain of Gaussian process into polygonal subdomains, and formulates a small Gaussian process regression problem for each subdomain. The mesh generation function (`mgf`) defines the polygonal subdomains and returns the structure of the polygonal subdomains. We first define some terminology used for explaining the structure. The structure is defined by three sets of sub-structures as follows:

- *subdomain*: this structure defines the interior part of a subdomain.

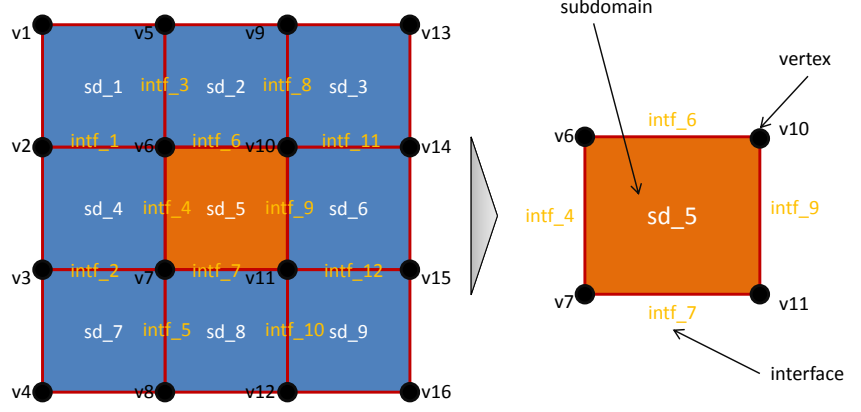


Figure 1: Structure of domain decomposition

- *interface*: this structure defines a line segment of the polygonal boundary for a subdomain.
- *vertex*: this structure contains information about vertices of the polygonal boundary.

For example, if you want to decompose the domain into 3-by-3 rectangular subdomains, the decomposition structure can be represented by nine subdomains (`sd_1,...,sd_9`), twelve interfaces (`intf_1,...,intf_12`) and sixteen vertices (`v1,...,v16`) as in Figure 1.

The mesh generation function (`mgf`) that returns any decomposition structure must have the following format:

```
[subdomains, interfaces, vertices, memberFunc, affine]
    = mgfName(param, x, y, idx)
```

which has the input parameters

- `x`: n by 2 matrix of training inputs
- `y`: column vector of length n of training targets
- `idx`: n by 1 logical mask to indicate the subset of training data used for learning the hyperparameters.
- `param`: a structure of the parameters necessary for generating the structure of domain decomposition.

The first job which `mgfName` needs to do is to define individual subdomains. A subdomain is polygonal, and its boundary is defined by a set of vertex coordinates of the polygon. The `vertices` output value of `mgfName` should be a $V \times D$ matrix containing a set of all polygon vertex coordinates of all subdomains, where V is the total number of vertices and D is the dimension of the domain. If we have the subdomains in Figure 1, the `vertices` should be a 16-by-2 matrix of the sixteen vertices in two dimensional space.

The definition of subdomains is stored in the first output value of `mgfName`, `subdomains`. The output is of cell array type in `Matlab`. Each element of the cell array is of 'structure' data type, which contains the definition of a subdomain. In other words, if `sd_i` is a structure containing the definition of the i th subdomain (`subdomains{i} = sd_i`). The `sd_i` should have the following elements:

- **sd_i.T**: a row vector of the indices pointing the vertices in **vertices** output value, which corresponds to the polygon vertices of the *i*th subdomain boundary.
- **sd_i.n**: the number of the training inputs inside the polygonal boundary specified by **sd_i.T**.
- **sd_i.x**: **sd_i.n** by 2 matrix of the training inputs inside the polygonal boundary specified by **sd_i.T**.
- **sd_i.y**: column vector of length **sd_i.n**, containing the training targets that corresponds to **sd_i.x**.
- **sd_i.hx**: a subset of **sd_i.x** that are used for learning hyperparameters; this can be obtained by subsetting **sd_i.x** with the input parameters **idx**.
- **sd_i.hy**: the training targets corresponding to **sd_i.hx**.
- **sd_i.neighbors**: a set of indices pointing elements in **subdomains**, which corresponds to the subdomains neighboring to **sd_i**. Its last element should be the total number of the neighbors. For example, in Figure 1, **sd_5** has four neighbors of **sd_2**, **sd_4**, **sd_6** and **sd_8**. **sd_5.neighbors** should have the value of [2, 4, 6, 8, 4].

The second job performed by **mgfName** is to define a set of the interfaces shared by pairs of neighboring subdomains. The second output value of **mgfName** is a M-by-1 cell array, containing a set of interfaces. The *j*th element of the cell array, denoted by **int_j**, is of structure data type, containing the definition of the *j*th interface (**interfaces{*j*} = int_j**). The **int_j** should have the following fields:

- **int_j.i**: the index pointing an element in **subdomains** output value, which corresponds to the first one of two neighboring subdomains.
- **int_j.j**: the index pointing an element in **subdomains** output value, which corresponds to the second one of two neighboring subdomains (always, **int_j.i < int_j.j**).
- **int_j.x**: a 1-by-2 vector of two indices pointing two elements in **vertices** output value, which corresponds to two end points of the line segment defined by **intf_j**.

In addition, the **mgfName** stores the indices of the interfaces related to **sd_i** as follows:

- **sd_i.int_idx**: a row vector of the indices pointing the elements in **interfaces**, which corresponds to parts of the polygonal boundaries of **sd_i**. The last element of this row vector is the total number of the interface indices stored in this row vector. For examples, in Figure 1, the polygonal boundary of **sd_5** consists of four interfaces (**intf_4**, **\intf_6**, **\int_7** and **int_9**). The **sd_5.int_idx** should be [4, 6, 7, 9, 4].

Last, the **mgfName** should return the following Matlab function references:

- **memberFunc**: a membership function to check if data **x** belongs to subdomain **sd_i**. Its usage is **idx = memberFunc(x, sd_i)** with parameters
 - **x**: *n*-by-*D* matrix of data inputs

- `sd_i`: the definition of a subdomain
- `idx`: n -by-1 column vector of indices pointing the elements in `x`, which belongs to `sd_i`.
- **affine**: affine transformation that transforms the global coordinates `x` into the local coordinates in subdomain `sd_i`. Its usage is `loc = affine(x, sd_i)` with parameters
 - `x`: n -by- D matrix of data inputs
 - `sd_i`: the definition of a subdomain
 - `loc`: n -by- D matrix of the local coordinates of `x` in `sd_i`; the local coordinate system can be defined in users' own ways, but the range of the local coordinate system should be the same for every subdomain.

For examples defining a new mesh function, please refer to `rectMesh.m` and `rectGrid.m` at `./mesh` directory. In the same directory, you can also find another documentation regarding the definition of mgf entitled `meshFunction.m`.

5 Conclusion

The GPLP Version 1.0 is a **Matlab** (or **Octave**) based software package that implements several localized computation methods of the Gaussian process regression problem, including the domain decomposition method (DDM), two parallel computation versions of DDM, partial independent conditional (PIC), local probabilistic regression (LPR) and bagging for Gaussian process regression (BGP). The methods implemented in GPLP have not been implemented in the general purpose computation toolbox for Gaussian process (GPML toolbox), so GPLP expects to be a nice complement to GPML. The GPLP also provides two parallel computation codes of the domain decomposition method, which is expected to solve much larger scale spatial regression in a timely manner. This documentation provides several examples to show how to use GPLP toolbox for general users. It also explains to advanced users how to extend the functions of this toolbox with computational language interfaces.

References

- Chen, T. and J. Ren (2009). Bagging for Gaussian process regression. *Neurocomputing* 72(7-9), 1605–1610.
- Kepner, J. (2001). Parallel programming with MatlabMPI. In *Proceedings of the High Performance Embedded Computing (HPEC 2001) workshop*.
- Park, C., J. Z. Huang, and Y. Ding (2011). Domain decomposition approach for fast gaussian process regression of large spatial data sets. *Journal of Machine Learning Research* 12, 1697–1728.
- Rasmussen, C. E. and H. Nickisch (2010). Gaussian processes for machine learning (GPML) toolbox. *Journal of Machine Learning Research* 11, 3011–3015.
- Snelson, E. and Z. Ghahramani (2007). Local and global sparse Gaussian process approximations. In *International Conference on Artificial Intelligence and Statistics 11*, pp. 524–531. Society for Artificial Intelligence and Statistics.

Urtasun, R. and T. Darrell (2008). Sparse probabilistic regression for activity-independent human pose inference. In *IEEE Conference on Computer Vision and Pattern Recognition 2008*, pp. 1–8.