Manual for Generic Trans-dimensional code for 2D Problems

Rhys Hawkins

November 11, 2021

Contents

1 Introduction

This software contains a general framework for the inversion of 2D surface reconstructions using a Bayesian Trans-dimensional approach with a Voronoi Cell, Delaunay Triangulation or Clough-Tocher interpolant. For an introduction to the Trans-dimensional Tree method, see **?**.

2 Installation

The main framework is in the source code in the `base` and `lib` directories, with a demonstration of a regression problem in the `generalregressioncpp` directory and the code for the synthetic Tide Gauge example in the `tides` directory.

This code is expected to be run on a Unix machine and requires the following packages to compile:

- GNU g++ Version 6.x or greater
- GNU Make version 4.x or greater
- GNU Scientific Library (GSL) version 1 or 2
- OpenMPI version 1.10

The source code with example scripts and data is called TransTessellate2D.tar.gz and once the environment is properly configued, extraction and compilation can proceed as follows:

```
> tar -xzf TransTessellate2D.tar.gz
> cd TransTessellate2D
> make
```

which should compile the individual components and example programs. The code defaults to GNU/OpenMPI compilation tools, but has been tested also using the Intel compiler suite. See the Makefile's in the lib, base, generalregressioncpp and tides directories to use Intel compilers.

3 Tutorial on Running the Regression Example

In the `generalregressioncpp` directory there are two examples that correspond to the two regression models inverted in **?**. The "franke" directory is for the smooth model and the "tesselation" directory is for the discontinuous model. We describe an overview of the steps required in the following sub sections and point out that in each of these directories is a Makefile which will run smaller versions of the inversions that appear in the publication (smaller number of parallel chains). The `transcale` sub directory contains the SLURM scripts that were used to run the actual inversions in the manuscripts on our cluster.

3.1 Creating a synthetic dataset

The synthetic observation generating program requires a set of sythetic points. A python script is available to do this:

```
python2 ../../scripts/generatetempatepoints.py -N 100 \
  -o datatemplate.txt \
  --seed 983 \
  --xmin 0.0 --xmax 1.0 \
  --ymin 0.0 --ymax 1.0
```

The `-N` option specifies the number of points. By default the domain of the points will be -1 ... 1 for both x and y directions and this can be overidden with the x/y min/max command line options. The points are generated randomly with a uniform distribution and the seed can be changed to give different models.

To generate synthetic observations, the mksynthetic program is used with the data points as input:

```
../mksynthetic \
  -m Franke \
  -i datatemplate.txt \
  -x 0.0 -X 1.0 \
  -y 0.0 -Y 1.0 \
  -o synthetic_franke.txt \
  -O synthetic_franke.txt.true \
  -n 0.05 \
  -I syntheticobs_franke.img \
  -W 1024 -H 1024
```

The first parameter is the synthetic model name to generate. To find a list of available synthetic models, you can run "mksynthetic -l". The -i input parameter is the template list of points. The -x/X and -y/Y are the bounds of the region and should be the same as above. The -o option indicates the filename to save the observations and similarly the -O option indicates the filename to save the true observations, that is before the addition of noise. The -n option specifies the standard deviation of the independent Gaussian noise to be added. The last three options specify an optional image to be output of the actual model image and its width and height.

Not shown here are two other options that can be used to change the range of values of the synthetic model. By default the synthetic models output values in the range 0 . . . 1. This can be changed by scale and offset parameters:

```
  --scale 0.5 \
  --offset 3.0 \
```

which would remap 0 . . . 1 of the model to 2.75 . . . 3.25 which may be more realistic for surface wave tomography for example.

3.2 Prior/Proposal Specification

Priors and Proposals are specified with small text files. There are three types: value prior/proposals, position prior/proposals and hierarchical prior/proposals.

For prior/proposal files, the format is quite straight forward with a prior definition followed by a proposal definition, for example

```
prior Uniform
-0.5 2.0
proposal Gaussian
0.05
```

The first line must have “prior” followed by a valid prior name. See the source code base/prior.hpp for a list of available priors. The following two lines specify the lower and upper bounds of the prior.

Then the proposal definition similarly must have “proposal” then a valid proposal name. This proposal is only used for McMC inversions as value proposals in HMC inversion are computed differently. The following line specifies the standard deviation of the Gaussian perturbation in this case and can be tuned to obtain a better acceptance rate.

For the position priors, at present the file will always be of the form:

```
positionprior UniformPosition
positionproposal GaussianPosition
0.1
```

The last line contains the standard deviation for movement of vertices/nodes and this number can be changed to improve acceptance rates of move proposals. In general, reducing this number will result in improved acceptance.

Lastly the hierarchical prior/proposal file is of the same form as for the value prior/proposal file.

3.3 Serial Inversion

The way these codes work is that they run for a specified number of iterations and output statistical information to a series of files. To run a short inversion of the synthetic data, the following could be used:

```
mkdir -p results_franke
../hmc -i syntheticobs_franke.txt \
  -P priorproposal.txt \
  -M positionpriorproposal.txt \
  -H hierarchicalpriorproposal.txt \
  -o results_franke/ \
  -v 1000 \
  -t 100000 \
  -A 0 \
  -x 0.0 -X 1.0 \
  -y 0.0 -Y 1.0
```

The parameters are described as follows:

-i The observations input file

-P The value prior/proposal file described in the previous section

-M The position prior/proposal file described in the previous section

-H The hierarchical prior/proposal file described in the previous section

-o The results file(s) output prefix (append a "/") to output to a directory.

-v Verbosity or how often the running diagnostics are output

-x/X/y/Y The bounds of the 2D region to invert. All points must be within this region.

-t The total number of iterations

-A The parameterization to use, 0 for Voronoi, 1 for Delaunay linear, and 2 for Delaunay Clough-Tocher.

In the `results_franke` directory, there will be a number of files written, described as follows:

ch.dat the chain history (binary)

finalmodel.txt the final model (useful for continuing Markov chains)

khistogram.txt the histogram of the number of vertices/nodes

residuals.txt the mean residuals for each observation.

The most important of these for analysis/visualization of the results is the chain history file `ch.dat` that contains all the models of the ensemble. Information can be extracted from the chain history file with post processing commands, for example, to obtain a mean image:

```
../post_mean -i results_franke/ch.dat \
  -s 50000 -t 10 \
  -x 0.0 -X 1.0 -y 0.0 -Y 1.0 \
  -A 0 \
  -W 100 -H 100 \
  -o results_franke/mean.txt
```

The various parameters here are descibed as follows

-i The input chain history file

-s The number of models to skip from the processing (i.e. burnin samples that are thrown away).

-t Thinning, or in the example above, only every tenth model is used.

-x/X/y/Y The region of the image (must match inversion).

-A The parameterization to use (this must match the inversion).

-o The output mean image to write.

The mean image is writen in text format and can be visualized simply with python, for example

```
import numpy
import matplotlib.pyplot as P

img = numpy.loadtxt('results_franke/mean.txt')
fig, ax = P.subplots()
ax.imshow(img, origin = 'lower')
P.show()
```

3.4 Further examples

Several other use cases of the commands described above are shown in the two example sub directories in Makefiles.

3.5 Diagnostics

The program will output log messages periodically depending on the verbosity level (default is every 1000 iterations). An example is shown below.

```
99000: Likelihood  45.672329 (-245.160863) Lambda(s)   1.875096 Cell   7
          HMC:|   33665/   47094 :  71.48 |
        Birth:|      88/    2292 :   3.84 |
        Death:|      82/    2339 :   3.51 |
         Move:|    6727/   23649 :  28.45 |
  Hierarchical:|   17791/   23626 :  75.30 |
```

The first line contains, the current iteration number (99000), the current negative log likelihood with the misfit (45.67) and normalization shown in brackets (-245.16). The remainded of the line outputs the current hierarchical parameter values (Lambda(s)) and the number of cells in each model.

A key component of monitoring these inversions is the acceptance rates. In the example output above, the last 5 lines show the statistics for each of the proposal types.

To tune these acceptance rates, editing of the `priorproposal.txt` file may be required. To lower the acceptance rate, in the case where it is too high, the standard deviation of the particular level must be increased. The reverse is true when the acceptance rate is too low, i.e. the standard deviation of the perturbation should be decreased.

4 Parallel chains and Parallel Tempering

With the basics covered above, there are parallel versions of the base applications which allow both parallel chains, parallel tempering and paralellization of the forward model calculations. In the above example, we ran `hmc` and its parallel counter part is `hmcpt`. When running this version it should be executed using the `mpirun` program. Repeating the example above, we could run

```
mkdir -p results_franke_parallel
mpirun -np 12 ../hmcpt -i syntheticobs_franke.txt \
  -P priorproposal.txt \
  -M positionpriorproposal.txt \
  -H hierarchicalpriorproposal.txt \
  -o results_franke_parallel/ \
  -v 1000 \
  -t 100000 \
  -A 0 \
  -x 0.0 -X 1.0 \
  -y 0.0 -Y 1.0 \
  -c 3 \
  -K 4 -m 5.0 -e 10
```

Here we have configured the inversion to run on 12 cores. The extra parameters which determine how these cores are used are

-c The number of independent chains to run at each temperature

-K The number of temperature levels for parallel tempering (1 == no parallel tempering)

-m The maximum temperature for the parallel tempering log temperature scale

-e The number of iterations between parallel tempering exchange attempts

A constraint here is that the number of independent chains times the number of temperatures must be an integer factor of the number of cores. Here we have used 3 chains and 4 temperatures which requires 12, 24, 36, etc total cores. The number of cores used per chain is then simply the total number of cores divided by product of the numbers of chains and temperatures. Hence with 12 total cores we have a single core per chain, whereas with 24 we would have two cores processing each chain.

Similarly when post processing the chain histories for parallel chains, we run a parallel version of the `post_mean` program with `mpirun`. The number of processors requested here should match the number of chains (ie below we have -np 3 and above we have -c 3).

```
mpirun -np 3 ../post_mean_mpi \
  -x 0.0 -X 1.0 -y 0.0 -Y 1.0 \
  -i results_franke_parallel/ch.dat \
  -o results_franke_parallel/mean.txt \
  -D results_franke_parallel/stddev.txt \
  -t 10 \
  -s 50000 \
  -W 100 -H 100
```

This will compute the mean and standard deviations across the three chains. To compute means etc on a single chain, the parallel version outputs files for the form `ch.dat-000`, `ch.dat-001` etc so to compute the mean and standard deviation of the first chain one could do the following

```
../post_mean \
  -x 0.0 -X 1.0 -y 0.0 -Y 1.0 \
  -i results_franke_parallel/ch.dat-000 \
  -o results_franke_parallel/mean.txt-000 \
  -D results_franke_parallel/stddev.txt-000 \
  -t 10 \
  -s 50000 \
  -W 100 -H 100
```

5 Writing Custom Applications

Each of the example applications are written as custom applications and as such can be used as templates for incorporating your own physical modelling or likelihood functions into an trans-dimensional tree inversion. To create your own custom application it is recommended you copy the Makefile and generalregression.cpp files into a new subdirectory and use this as a starting template.

There are a number of functions that need to be defined and we give a brief description here before giving some background information and more detail on each.

gvcart_initialise_ The first function is an initialisation routine that simply returns the number of hierarchical parameters to create and the number of 2D surfaces to invert for.

gvcart_loaddata_ In the framework, the model and inversion is kept separate from the data and this function loads and stores the data, but registers sampling points required for each observation.

gvcart_compute_prediction_ This function needs to compute the prediction based upon model values at prediction points.

gvcart_compute_likelihood_ This function computes the likelihood and normalization constant based on hierarchical parameters and predictions.

gvcart_savedata_ This function needs to save data in a format that your own `gvcart_loaddata_` can read but with modified predictions and errors. This is used for generating synthetic observations.

5.1 Background

For more detailed background, see the acompanying paper **?** and references there in.

Conceptually, this software provides a framework for Bayesian inferences of 2D fields and is divided into the modelling side, which comprises of the trans-dimensional Voronoi cell or Delaunay triangulation and a user supplied prediction and likelihood function.

In a Bayesian inversion, we seek a posterior distribution of the model $\mathbf{m}$ using Bayes' law

$$p(\mathbf{m}|\mathbf{d}) \propto p(\mathbf{m})p(\mathbf{d}|\mathbf{m}) \tag{1}$$

where $p(\mathbf{m}|\mathbf{d})$ is the posterior probability distribution of interest, $p(\mathbf{m})$ is the prior on model parameters, and $p(\mathbf{d}|\mathbf{m})$. In this software, the priors are specified using configuration files as explained in the tutorial above. The likelihood is custom written by the user.

Using the Gaussian likelihood function as an example, this is written as

$$p(\mathbf{d}|\mathbf{m}) = \frac{1}{\sqrt{2\pi|\mathbf{C}_d|}} \exp -\frac{1}{2}(g(\mathbf{m}) - \mathbf{d})^T \mathbf{C}_d^{-1}(g(\mathbf{m}) - \mathbf{d}) \tag{2}$$

where $g(\mathbf{m})$ is the forward model operator that computes predictions to compare to the data $\mathbf{d}$, and $\mathbf{C}_d$ is the data covariance matrix of errors. In the software, the likelihood is split into two distinct parts where the predictions, that is the vector $g(\mathbf{m})$, is computed by the `gvcart_compute_predictions_` function, and the likelihood by the `gvcart_compute_likelihood_`.

5.2 Initialisation

The initialisation function is simply used to tell the framework how many hierarchical parameters and models to invert for.

```
int gvcart_initialise_(int *nmodels,
                       int *nhierarchical)
```

5.3 Loading data

The data file format is entirely in the control of the user. The command line application is passed a filename to load, and this is passed onto the load function directly. This file could be a raw data file or a configuration file that points to a series of files that require loading for the inverse problem. This function must load all necessary data for subsequent calculation of predictions and the likelihood. The interface is as follows:

```
int gvcart_loaddata_(int *n,
                     const char *filename,
                     gvcart_addobservation_t addobs);
```

The parameters are

n The length of the filename string

filename The filename to load

addobs A callback function to register points

height The height of the model image

The first parameter is only needed for Fortran problems.

It is up to the custom application to permanently store any data necessary for the likelihood or calculation of predictions in this function.

The function should return zero on success and negative one for any error. This will subsequently terminate the main program with an error condition.

One of the most important aspects of this function is the registering of points required for each observation. For each observation, the function `addobservation` (abbreviated `addobs` in the regression code) should be called, its interface is

```
int addobservation(int *n,
                   int *model_indices,
                   double *x,
                   double *y);
```

The first parameter is the number of points, then the following parameters of arrays of model index (starting from 0), x coordinate and y coordinate. It is necessary to call `addobservation` for each observation, that is, each element of the **d** vector.

5.4 Computing predictions

The next function that needs to be implemented is the function that computes the predictions for a single observation. The C interface for this function is shown below

```
int gvcart_compute_prediction_(int *nmodels,
                               int *observation,
                               int *npoints,
                               const double *value,
                               double *weight,
                               double *prediction);
```

The first parameter specifies the number of inverted for models. This function is required to compute the prediction for a given observation and this is indicated by the observation integer parameter. Observations are indexed from zero to the number observations minus one (C indexing) and in the order they are registered when loading with the `addobservation` callback above. This enables parallel forward modelling.

The next two parameters specify the number of points and the value of the model at each point.

The weight parameter is a 1D array with a length of the number of points. It represents the partial deriviatives of the prediction with respect to the model values. For McMC inversions, this can be safely ignored, but is required for HMC inversions.

Lastly, the prediction parameter is a single value that needs to be set and is equivalent to $g(\mathbf{m})$ above. For example, this should be a value for regression type problems or a travel time for tomography type problems.

The function should return zero on success and negative one if there is an error.

5.5 Computing the likelihood

The C++ interfaces for the function that needs to be implemented to compute the likelihood are below:

```
int gvcart_compute_likelihood_(int *nmodels,
                                int *nhierarchical,
                                int *nobservation,
                                double *hierarchical,
                                double *predictions,
                                double *residuals,
                                double *weight,
                                double *like,
                                double *norm);
```

The input variables are

nmodels The number of models inverted.

nhierarchical The number of hierarchical parameters inverted.

nobservation The number of observations (size of the arrays passed in)

hierarchical The value of the hierarchical parameters in an array nhierarchical in length.

predictions The vector of predictions as an array nobservation in length.

On output, the array residuals must be set which is an array the same length as the predictions input array. Generally this is equivalent to simply $g(\mathbf{m}) - \mathbf{d}$. For HMC inversion, the weight array, again the same length as the predictions array, needs to be set to the partial derivative of the -log of the likelihood with respect to the prediction, that is $\text{weight}_i = \frac{\partial}{\partial g(\mathbf{m})_i} - \log p(\mathbf{d}|\mathbf{m})$.

Lastly the negative log likelihood needs to be set as the like output variable and the normalization constant in the norm output variable.

Again, this function should return zero on success and negative one on error.

5.6 Synthetic Model Generation

For synthetic model generation useful for testing, an additional function needs to be implemented with the C++ fortran interfaces below

```
int gvcart_savedata_(int *n,
                     const char *filename,
                     double *noiselevel,
                     int *nobservations,
                     double *predictions);
```

The parameter `n` and `filename` specify the length and the name of the file to create. The `noiselevel` specifies the true Gaussian noise added to the synthetic observations (this will be zero when outputting the true observations). Lastly the `nobservations` and `predictions` specify the array size and the array of synthetic predictions that are to be output as the observations.

For synthetic model general (see the file `mksynthetic.cpp`), a template observation file will be loaded by calling `gvcart_loaddata_` followed by `gvcart_compute_predicti`

5.7 Parallel implementation

The generic interface is structured for observation parallelism so that if for example a single chain is run on ten processors using the parallel version, observations are evenly split between processors and residuals computed. The results are gathered on a single processor where the likelihood is computed. In summary

- `gvcart_loaddata_` is called by all processes at the beginning before the iterations begin
- `gvcart_compute_prediction_` is called for a subset of the observations on each processor
- `gvcart_compute_likelihood_` is called by one processor (as it is generally a trivial summation over the residuals).

5.8 Recommendations

The simplest path to starting a custom application is to copy the C++/Fortran regression example and use that as a template.

For each of these examples, there is an example subdirectory has a Makefile that

1. creates some synthetic template data using scripts included in this source bundle. This essentially creates empty observations but with random points (regression) or random paths (tomography) on the sphere.
2. creates synthetic data using the compute predictions implementation and adding some Gaussian noise
3. runs a short single process inversion
4. runs the post processing to compute the mean model and standard deviation.

Both the source code for these programs and the simple examples provide good starting points for creating custom applications.