Standardisation Guide for "gp-emulators"

N.K. - kraemer@ins.uni-bonn.de

December 22, 2018

Abstract

This guide collects any standardisation I have come up with over time. On top of that, I mention other guidelines, like naming conventions, structure conventions and more, most of which I have learned the hard way. Many things are adapted from rbf-tools.

Contents

1. Purpose of the module

The purpose of the module is to collect most of the things I have programmed in the past 18 months with regard to radial basis functions. Resuability is a driving force of most of the modules. Many features have to be used in almost every script; for instance, building a covariance matrix. I got sick of doing it from scratch everytime, hence I started this collection.

In the context of Gaussian process emulators, one has to use a wide range of tools: PDE solvers, quadrature formulas, radial basis function interpolation, sampling according to probability distributions and more. This collection is supposed to allow combining any number of these easily.

2. Hierarchy

The hierarchy is supposed to be as flat as possible. However, we need to separate certain formulas for the aforementioned reusability purposes.

3. Naming and coding conventions

I follow naming conventions with two purposes in mind:

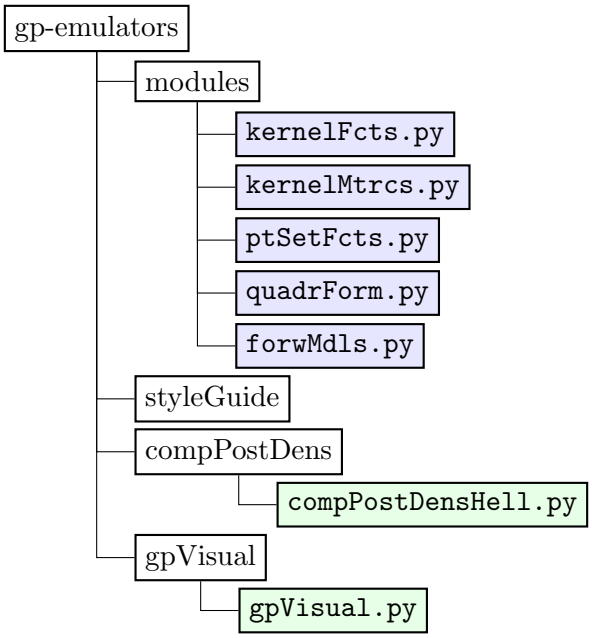

Figure 1: File structure of `rbf-tools`; modules in blue, scripts in green

1. Good programming practice
2. Unification

3.1. Good naming practice

The following is a list of most naming conventions regarding good practices:

1. **Variable naming:**
   - **Descriptive naming:** do not use `x`, `N` or `K`, but `pt`, `numPts` or `kernelMtrx`
   - **Short names:** do not use `standardKernelMatrixWithMaternKernel`, but `kernelMtrx`
   - No underscores (system variables)
   - No all-uppercase variables (system variables)
   - Indicate new "term" with a single uppercase letter: `kernelFct`, `kernelMtrx`, `ptSet`
2. **Commenting:** As long as the variables are named well, I do not need comments except for very few occasions
3. **Function naming:** verb-noun scheme, i.e. `buildKernelMtrx`, `getPtSet`, ...
4. **File naming:** Each file has to include the following information:
   (a) **Name:** e.g. `'interpolation.py'`
   (b) **Purpose:** Describe the purpose of the file in a single sentence (if that is not possible, think again about starting this file at all)

(c) **Description:** Describe the method in two or three sentences giving the main keywords

(d) **Author:** Usually me

An exemplary header is the following, taken from `'interpolMatern1d.py'`:

```
# NAME: 'interpolMatern1d.py'
#
# PURPOSE: Basic 1-dimensional interpolation using Matern functions
#
# DESCRIPTION: I solve a system involving a Matern-kernel matrix
# where the Matern kernel is based on scipy.special's functions
# and plot the 10-th Lagrange function.
#
# AUTHOR: NK, kraemer(at)ins.uni-bonn.de
```

3.2. Unification

The following is a list of most naming conventions regarding a unified system:

1. **Covariance functions:** I refer to covariance functions and covariance matrices using `cov`, not `kern` nor `kernel`; the functions themselves are the same, however.
2. **Common Abbreviations:** I use as common abbreviations:
   - Indices: `idx`, `jdx`, `kdx`, ...
   - Point: `pt`
   - Pointset: `ptSet`
   - Numer of points: `numPts`
   - Matrix: `mtrx`, matrices: `mtrcs`
   - Length of a vector called `vecAbc`: `lenVecAbc`
   - Pointset for evaluation (plotting): `evalPtSet`
   - Number of evaluation points: `numEvalPts`
   - Lebesgue constant: `lebCnst`
   - Expression: `expr` (as in `exprSol`)
   - Collocation: `coll` (as in `collMtrx`)
   - Gaussian: `gauss` (as in `gaussCov` instead of `gaussianCov`)
   - Higher order thin-plate splines (polyharmonic covariances): `tps1Cov` for $r^1 \log(r)$, `tps2Cov` for $r^2 \log(r)$, `tps3Cov` for $r^3 \log(r)$, ...

3.3. Other good practices

1. **Functions:**
   - Each function should serve **a single** purpose which should be clear from the naming
   - Each function should be deterministic, i.e. two runs with the same input give the same output (this type of function is called pure function). In my case this often depends on random numbers; see next point
2. **Seeds for random numbers:** Each file should always give the same result as long as nothing is changed. Hence, start everything that includes random numbers with `np.random.seed(15051994)` and do not set another seed elsewhere
3. **Readability:** Readability of a program often trumps slight performance improvements
4. **Parameters:** If a simulation is to be run with different parameters, let the script ask for the parameter as in:

   ```
   print "\nWhich regularity of the Matern function? (e.g. 2.0)"
   maternReg = input("Enter: ")
   ```

   Do not change them in the script–it turned out to be quite confusing in the past

4. Modules

In the following I describe some module files and their conventions.

4.1. Covariance functions

I collect covariance functions in the file `covFcts.py`. They all take two points as inputs and give out a scalar. As an example, the Gaussian:

```
def gaussCov(ptOne, ptTwo, lengthScale = 1.0):
    distPts = np.linalg.norm(ptOne - ptTwo)
    return np.exp(-distPts**2/(2*lengthScale**2))
```

The distance of the two inputs, `ptOne` and `ptOne`, is computed inside the function. The purpose of this is that I can construct covariance matrices in a very easy manner; see subsection 4.2

4.2. Covariance matrices

I collect covariance matrices in the file `kernelMtrcs.py`. The all take two pointsets as inputs and return a matrix. As an example, the standard covariance matrix:

```
def buildCovMtrx(ptSetOne, ptSetTwo, covFct):
    lenPtSetOne = len(ptSetOne)
    lenPtSetTwo = len(ptSetTwo)
    covMtrx = np.zeros((lenPtSetOne, lenPtSetTwo))
    for idx in range(lenPtSetOne):
        for jdx in range(lenPtSetOne):
```

```
            covMtrx[idx,jdx] = covFct(ptSetOne[idx,:], ptSetTwo[jdx,:])
    return covMtrx
```

The input pointsets need to have the same dimension, but do not need to match in size. The covariance function `covFct` needs to be of the form I described in subsection 4.1

4.3. Pointsets

I collect different strategies to construct pointsets such as random points, Halton points and more in the file `ptSetFcts.py`. The input is always the **overall number of points** and the desired dimension as in:

```
def getPtsHalton(size, dim):
        [...]
        return np.array(seq).T
```

The output is always a pointset in $[0,1]^d$, if not specified otherwise. The function names start with `getPts`, as in for example `getPtsHalton` or `getPtsFibonacci`.

4.4. Quadratures

I collect different quadrature formulas, such as Quasi-Monte Carlo, Monte Carlo or covariance quadrature in the file `quadrForm.py`. The titles of the functions always start with `compQuad`, such as `compQuadQMC` or `compQuadMC`. The formulas take as an input the integrand and the pointset as in:

```
def compQuadQMC(integrand, ptSet):
        [...]
        for idx in range(numPts):
                qmcSum = qmcSum + integrand(ptSet[idx,:])
        return qmcSum/(N * 2.0**dim)
```

The reason for choosing this over an on-the-fly computation is the readibility of the code. The integrand is supposed to take a single vector as an input which is supposed to make it dimension-independent.

Do not forget to adjust the integrand according to the measure (especially factors like $\frac{1}{|A|^d}$).

4.5. Forward models

I collect different forward models, such as PDE-based forward models in the file `forwMdls.py`. The input is the vector of input parameters. The outputs are all **evaluation-based**, hence the output vector contains the evaluation points. The names of the forward models always start with `compForw`, as in `compForwFEM`.

5. Plotting

Visualisations (plots) tend to be more informative than printed results. The reason is that a plot can be stored easily and therefore, the information does not get lost. Every plot needs to have:

- **Axes labels:** not "`x`" and "`y`" but something like "`Distance to center`" and "`Lagrange function value`"
- **Title:** Keep it short, like "`Lagrange function decay`"
- **Grid:** Necessary as soon as numerical information is supposed to be taken from the plot
- **Legend:** Put details in the legend, like number of points or underlying regularity; it seems easiest to label each plot as in:

  ```
  plt.semilogy(..., linewidth = 2, label = "acc = %.1e"%(rileyAcc))
  ```

  and then to add a legend via

  ```
  plt.legend()
  ```

  The reason is that this way, the legend entries are consistent with the data.

On top of that, for aesthetic purposes, I tend to follow:

1. **Font size:** Change font size to e.g. 18 by adding

   ```
   plt.rcParams.update({'font.size': 18})
   ```

   to the top of each script
2. **Line width:** Increase line width and markersize, e.g. `linewidth = 2` and `markersize = 8`
3. **Color:** Main color is `darkslategray` which seems like a decent tradeoff between color and simplicity; as a complementary color I use `red` which is a tad lighter than its true complementary color
4. **Opacity:** Put the opacity value to `alpha = 0.8` which is less aggressive
5. **Legend:** A slight shadow seems nice:

   ```
   plt.legend(shadow = True)
   ```

6. **Saving:** Save plots into files automatically via for example

   ```
   plt.savefig("figures/titleOfScript/fctValues.png")
   ```