Contents

This document continues from those in `stein/notes/learning/mvn`, where we compute Stein Points for the multivariate normal distribution using the IMQ kernal.

Multivariate Normal Examples in C++

In this section I will work through a few simple examples to clarify my understanding of the problem and develop the outline of a `C++` solution.. These will focus on the IMQ kernel and the target distribution will be the multivariate normal distribution.

The multivariate normal distribution with mean μ, covariance Σ and dimension d is

$$p(\mathbf{z}) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left[-\frac{1}{2}(\mathbf{z}-\mu)^T \Sigma^{-1}(\mathbf{z}-\mu)\right]$$

with corresponding log target $g(\mathbf{z})$

$$g(\mathbf{z}) = -\frac{1}{2}\log\left((2\pi)^d \det(\Sigma)\right) - \frac{1}{2}(\mathbf{z}-\mu)^T \Sigma^{-1}(\mathbf{z}-\mu)$$

where $\mathbf{z} \in \mathbb{R}^d$ and of course the additive constant can be ignored for our purposes. The gradient of the log target is

$$\nabla \log p(\mathbf{z}) = -(\mathbf{z}-\mu)^T \Sigma^{-1}$$

.

The IMQ kernel is

$$k(\mathbf{x},\mathbf{y}) = \left(\alpha + \|\mathbf{x}-\mathbf{y}\|^2\right)^\beta$$

with $\alpha > 0$ and $-1 < \beta < 0$, which has gradients given by

$$\begin{aligned}
\frac{\partial k}{\partial x_i} &= 2\beta(x_i - y_i)\left(\alpha + \|\mathbf{x}-\mathbf{y}\|^2\right)^{\beta-1} \\
\frac{\partial k}{\partial y_i} &= -2\beta(x_i - y_i)\left(\alpha + \|\mathbf{x}-\mathbf{y}\|^2\right)^{\beta-1} \\
\frac{\partial^2 k}{\partial x_i \partial y_i} &= -2\beta\left(\alpha + \|\mathbf{x}-\mathbf{y}\|^2\right)^{\beta-2}\left[2(\beta-1)(x_i-y_i)^2 + \left(\alpha + \|\mathbf{x}-\mathbf{y}\|^2\right)\right]
\end{aligned}$$

Implementing a Simple Library for Use in OptimLib

Eigen is a header only linear algebra library used in large production libraries and system, such as the `Ceres` optimisation library used in production at Google and elsewhere. Eigen is the basis for the `OptimLib` optimisation library I propose to use, so I will use Eigen throughout. It has a nice api making development quick and easy while ensuring good quality performance. Mirroring the `python/lib` setup the small header only library is in `code/c++/lib`. Like the `python` lib it is written in three parts all under the namespace `stein`:

- `stein::kernel` for defining `kernel` objects, of which the `IMQ` kernel is the only one currently implemented.
- `stein::target` for defining `target` objects, of which the `MVN` target is the only one currently implemented.
- `stein` for defining generic objects required for Stein Point calculations, like the KSD, SRK etc.

I have chosen to use row major matrices (non-standard for Eigen) so defined the `stein::Matrix_t` and `stein::Vector_t` types along with other commonly used types in `types.H`. Eigen also has a defined framework for capturing arbitrary matrix-like types, which I use extensively. As an example the `operator()` for the MVN class template looks like:

```
[[nodiscard]] inline Scalar_t operator()(const Eigen::MatrixBase<VectorT> &
    ↪ z) const
{
    return _norm_const * std::exp(-0.50 * (z - _mu).transpose() *
        ↪ _inv_sigsq * (z - _mu));
}
```

`OptimLib` is very particular about signatures for its algorithms. Since I intend to use them regularly, I can `typedef` the required `stein::` types to fit in more easily. A final note for now is that `OptimLib` using a a `void*` argument to pass auxiliary data into the objective function. This means writing functions like

```
double target_obj(const stein::dVector_t& x, stein::dVector_t* grad, void*
    ↪ opt_data)
{
    stein::MVN_t* _mvn_obj_ = reinterpret_cast<stein::MVN_t*> (opt_data);
    return -1.0*_mvn_obj_->operator()(x);
}
```

where the use of `void*` is a bit scary. Alternatively we can write these function-like objects needed by `OptimLib` in `lambdas` or `functors`, which I think is safer. An example replacement of the call above would be

```
stein::target::MVN mvn {mu, sigsq};
auto target_obj = [&mvn](const stein::dVector_t&x, stein::dVector_t* g,
    ↪ void* optdata){ return -1.0*mvn(x); };
```

which is much nicer I think. Let's see how we go on this in the future.

Finally, performance in the `OptimLib` libraries appears to rely heavily on improved (I wont say optimised at this stage) compiler settings. Without obvious effort or profiling the library appears at least an order of magnitude faster than the `python` version. For example the same MVN problem as considered in the `python` notes is repeated here for $n = 100$. This following calculation takes roughly `6:45` mins

```
(base) optimlib % g++ -std=c++20 -O3 -march=native -ffp-contract=fast mvn1.
    ↪ C  -I/Users/patricknoble/Documents/Library/Eigen  -I/Users/
    ↪ patricknoble/Documents/Projects/stein/code/c++/lib  -I/Users/
    ↪ patricknoble/Documents/Library/OptimLib -o mvn1
(base) optimlib % ./mvn1
Compute MVN Mode with OptimLib.
 ** Stein Point (1) Found: -0.5  0.5
Begin Solving for 100 Stein Points.
 ** Stein Point (2) Found: 0.00604159    1.72169 in 0.17923 seconds.
```

```
 ** Stein Point (3) Found:   -1.0073 -0.724721 in 0.263277 seconds.
 ** Stein Point (4) Found: -1.38235 0.860729 in 0.33346 seconds.
 ** Stein Point (5) Found: 0.434139 0.110601 in 0.412709 seconds.
 ** Stein Point (6) Found: -0.831251   1.87337 in 0.485283 seconds.
 ** Stein Point (7) Found: 0.747009  1.20209 in 0.569836 seconds.
 ** Stein Point (8) Found:  -1.67736 -0.241186 in 0.651067 seconds.
 ** Stein Point (9) Found: -0.144906 -0.860912 in 0.723276 seconds.
 ...
 ...
 ** Stein Point (97) Found: -0.279761  0.692954 in 7.79265 seconds.
 ** Stein Point (98) Found: -0.455465   2.39737 in 7.7982 seconds.
 ** Stein Point (99) Found: 1.71781 2.29144 in 7.98523 seconds.
 ** Stein Point (100) Found: -1.34332  1.25429 in 7.98582 seconds.
Solver Completed in 6.73898 minutes.
```