Contents

These are my (ie the idiot) introduction to Stein Points, and the two associated computational problems I will investigate. These will be fleshed out over time as I come to understand more theory, and will include a number of examples that I found helpful on the way. Essentially everything in here is taken from Chen Et Al (2018).

Stein Point Concepts

Motivation

A common statistical problem is to approximate a probability distribution $P(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^d$ with a sequence of points $\{\mathbf{x}_i\}_{i=1}^n$. Such a sequence of points is an *empirical distribution* and is useful because it permits efficient calculation of various quantities (moments, quantiles etc). In addition they have well known and usefull theoretical properties. The critical goal is to produce a sequence which achieves *convergence* in the sense that

$$\frac{1}{n}\sum_{i=1}^{n} h(\mathbf{x}_i) \ \rightarrow \int h dP \tag{1}$$

when $n \rightarrow \infty$, where we are subject to various technical conditions in the above.

This projects concerns a particular sequence $\{\mathbf{x}_i\}_{i=1}^n$ of points called **Stein Points**, which are chosen carefully to provide a *best approximation* to $P(\mathbf{x})$, where we will define exactly what we mean by *best* in a later section. There are many techniques for generating convergent sequences of points from a given distribution, but most require all normalisation constants to be known. This requirement is commonly violated in practice, where we know the *functional form* of the distribution but may not be able to compute multiplicative constants. MCMC techniques can circumvent this problem by defining transition probabilities as ratios, so that these unknown constants cancel. Stein Points also do not require multiplicative constants to be known, so exist in a smaller set of plausible methods for this very common problem.

Definitions

A *discrepency* quantifies how well the points $\{\mathbf{x}_i\}$ cover the domain of the random variable $\mathbf{x}$ with respect to the distribution $p(\mathbf{x})$. The set of points which minimise a particular type of discrepency (a *Kernel Stein Discrepency*) with respect to the target $p(\mathbf{x})$ are referred to as *Stein Points*. The Stein Point methodology exists in the more general framework of *Reproducing Kernel Hilbert Spaces* (RKHS), which is a popular framework because analytic formulas for discrepencies are available. However, these general results suffer from the usual challenge that the target distribution's normalisation constant is required. It can be shows that particular choices of *Reproducing Kernels* (those chosen from a *Stein Set*) both simplify the maths in the more general RHKS and do not require knowledge of the normalisation constants. Such discrepencies are called *Kernel Stein Discrepencies* (KSD), and involve particular kernels referred to as *Stein Reproducing Kernels*. In particular the *Kernel Stein Discrepency* (KSD) is defined

$$\mathcal{D} = \sqrt{\frac{1}{n^2}\sum_i\sum_j k_0(\mathbf{x}_i, \mathbf{y}_j)} \tag{2}$$

where $k_0(\mathbf{x}, \mathbf{y})$ is the *Stein Reproducing Kernel* and is defined in a subsequent section.

Process

The process of approximating a target $p(\mathbf{x})$ using Stein Points proceeds by

1. Choosing a particular kernel $k(\mathbf{x}, \mathbf{y})$, which measures distances between points. This choice is subject to various technical conditions.
2. Choose a *Stein Operator* τ, which should be chosen cleverly in conjunction with the kernel above.
3. Compute the *Stein Reproducing Kernel* (SRK) $k_0(\mathbf{x}, \mathbf{y})$ by solving the integral equation $\int \tau[k]dP = 0$
4. Use the *Stein Reproducing Kernel* (SRK) to compute the *Kernel Stein Discrepency*

$$\boxed{\mathcal{D} = \sqrt{\frac{1}{n^2}\sum_i\sum_j k_0(\mathbf{x}_i, \mathbf{y}_j)}}$$

5. Choose an optimisation strategy to minimise the KSD above

Note that *Stein Operators* τ are chosen to

- Produce discrepencies that do not require normalisation constants
- Simplify formulas for discrepencies that are required when using the more general RKHS framework
- Combine with the choice of kernal $k(\mathbf{x}, \mathbf{y})$ to guarantee that as the KSD $\rightarrow 0$ the empirical distribution produced by our point sequence is convergent to the target $p(\mathbf{x})$.

After writing $k = k(\mathbf{x}, \mathbf{y})$ to save notation, the particular choice of τ considered here is the *Langevin Stein Operator*

$$\tau\left[k\right] = \nabla \cdot \left(pk\right)/p \tag{3}$$

which generates a Stein Reproducing Kernel (SRK)

$$\boxed{k_0 = \nabla_{\mathbf{x}} \cdot \nabla_{\mathbf{y}}\, k + \nabla_{\mathbf{x}}\, k \cdot \nabla_{\mathbf{y}} \log p(\mathbf{y}) + \nabla_{\mathbf{y}}\, k \cdot \nabla_{\mathbf{x}} \log p(\mathbf{x}) + k\nabla_{\mathbf{x}} \log p(\mathbf{x}) \cdot \nabla_{\mathbf{y}} \log p(\mathbf{y})} \tag{4}$$

The `div` $\cdot$ `grad` structure of the SRK results in a computation where partial derivatives are evaluated at each of the d components of $\mathbf{x}$ before being summed, that is, $k_0(\mathbf{x}, \mathbf{y})$ can be written

$$\boxed{k_0(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d}\left(\frac{\partial^2 k}{\partial x_i \partial y_i} + \frac{\partial k}{\partial x_i}\frac{\partial g}{\partial y_i} + \frac{\partial k}{\partial y_i}\frac{\partial g}{\partial x_i} + k(\mathbf{x}, \mathbf{y})\frac{\partial g}{\partial x_i}\frac{\partial g}{\partial y_i}\right)} \tag{5}$$

where I have substituted $g = \log p$ to again save notation.

Examples Part 1: Multivariate Normal Distribution with IMQ Kernel

In this section I will work through a few simple examples to clarify my understanding of the problem. 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

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

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

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

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

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

with $\alpha > 0$ and $-1 < \beta < 0$.

The multivariate normal is implemented in `mvn.py` and the IMQ kernel in `util.py`. Note that when run `mvn.py` tests the gradient calculations

- $\nabla g(\mathbf{z})$
- $\nabla_x k(\mathbf{x},\mathbf{y})$ and $\nabla_y k(\mathbf{x},\mathbf{y})$
- The d terms in $\nabla_x \cdot \nabla_y k(\mathbf{x},\mathbf{y})$

against numeric approximations.

Example 1.1: Univariate Normal Distribution

In this example I take $d = 1$ and investigate:

- What do the KSD and SRK formulas actually look like?
- What kind of objective functions result from the Greedy Minimisation strategy?

With $d = 1$ the objective function for the greedy minimisation strategy are easy to visualise. This is the univariate normal target $p(x)$. This will be the only case where the bold font will not be used for the random variable. The SRK is

$$k_0(x, y) = \frac{\partial^2 k}{\partial x \partial y} + \frac{\partial k}{\partial x}\frac{\partial g}{\partial y} + \frac{\partial k}{\partial y}\frac{\partial g}{\partial x} + k(x, y)\frac{\partial g}{\partial x}\frac{\partial g}{\partial y} \tag{6}$$

The log gradient for this choice of p is

$$\frac{\partial g}{\partial z} = -(z - \mu)/\Sigma \tag{7}$$

where both $\mu, \Sigma \in \mathbb{R}$, and the kernel derivatives are

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

The code implementing these formulas is in `stein/notes/learning/mvn/uvn.py`.

Plotting the Greedy Objective

Using the greedy optimisation method (Section 3.1 in Chen Et Al) we set the initial Stein Point to be $x_1 = \mu$ ie the distribution mode. Eventually this will require a global optimisation call but in this simple example lets just assign it. For each subsequent x_n for $n > 1$ take x_n to be the value which minimises the objective

$$\text{argmin}_z \left\{ \frac{1}{2} k_0(z,z) + \sum_{j=1}^{n-1} k_0(x_j, z) \right\} \tag{8}$$

Since we are in one dimension we can do an exhaustive grid search, and plot the objective function at each iteration to get a feel for the minimisation surface. The script that creates this plot is `stein/notes/learning/mvn/normal1-1.py`. Using the log gradient and SRK defined in `uvn.py` it is simple to write the greedy objective

```python
def objective(z, ksr, points):
    # evaluate the greedy objective for this point z \in R
    # where points is a sequence of previously computed points
    # use ksr(x,y) to evaluate ksr between any two points
    out = 0.5*ksr(z, z)
    if points is not None:
        for xi in points:
            out += ksr(xi, z)
    return out
```

In the plot below each curve plots the greedy objective for point x_j where $j = 2, 3, \ldots, n$. No objective is plotted for the initial point but the point itself is plotted in red. Subsequent Stein Points are plotted in black. This example would appear to be the *simplest possible example* of the computation of Stein Points and yet the scale of the optimisation problem is already apparent.

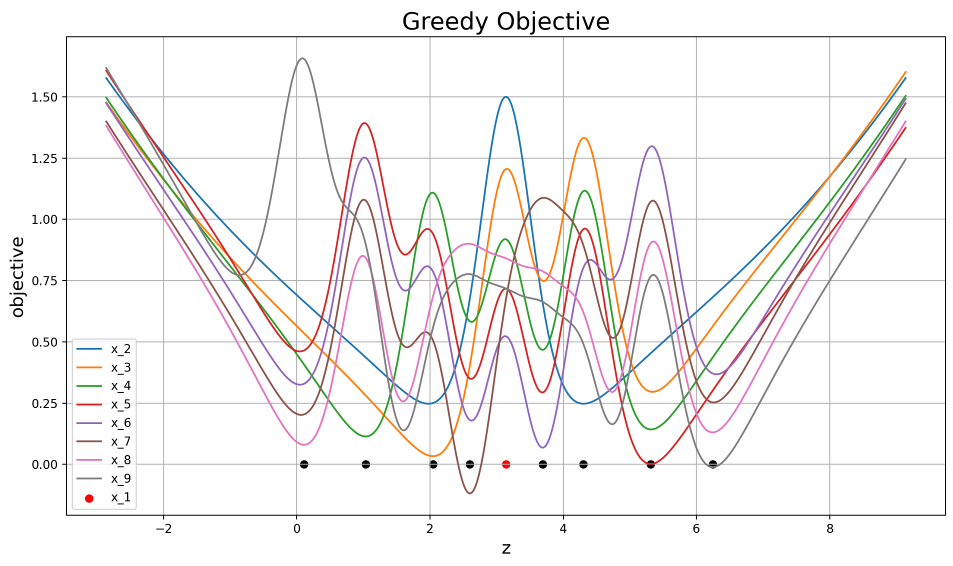


Figure 1: Greedy Objective for a Univariate Normal Distribution

Plotting Empirical CDFs and Comparing the numpy

In `/stein/notes/learning/mvn/normal1-2.py` we look at the empirical cdfs. Running the script produces output

```
**** Found Stein Point 2 at z=1.976 after 0:00:00.179566 (H:M:S) ****
**** Found Stein Point 3 at z=4.236 after 0:00:00.425367 (H:M:S) ****
**** Found Stein Point 4 at z=5.248 after 0:00:00.751985 (H:M:S) ****
**** Found Stein Point 5 at z=0.968 after 0:00:01.159569 (H:M:S) ****
...
...
**** Found Stein Point 18 at z=4.420 after 0:00:13.798995 (H:M:S) ****
**** Found Stein Point 19 at z=3.304 after 0:00:15.341222 (H:M:S) ****
**** Found Stein Point 20 at z=1.568 after 0:00:16.960261 (H:M:S) ****
Computation complete.  Found:
 -- 20 Stein Points.
 -- Wall Time: 0:00:16.960291 (H:M:S)
```

for the small $n = 20$ case. Using these samples we compare

- Stein Points
- Two samples of normal rvs generated from `numpy.random.normal`
- True normal CDF

to gain insight into the quality of the convergence of the Stein Point approximation. I do not know the method `numpy` uses to generate normal rvs but I presume it is high quality. No rigorous testing is performed but from a cursory look the Stein Points appear to be a highly efficient approximation compared to the normal rvs generated from `numpy`.

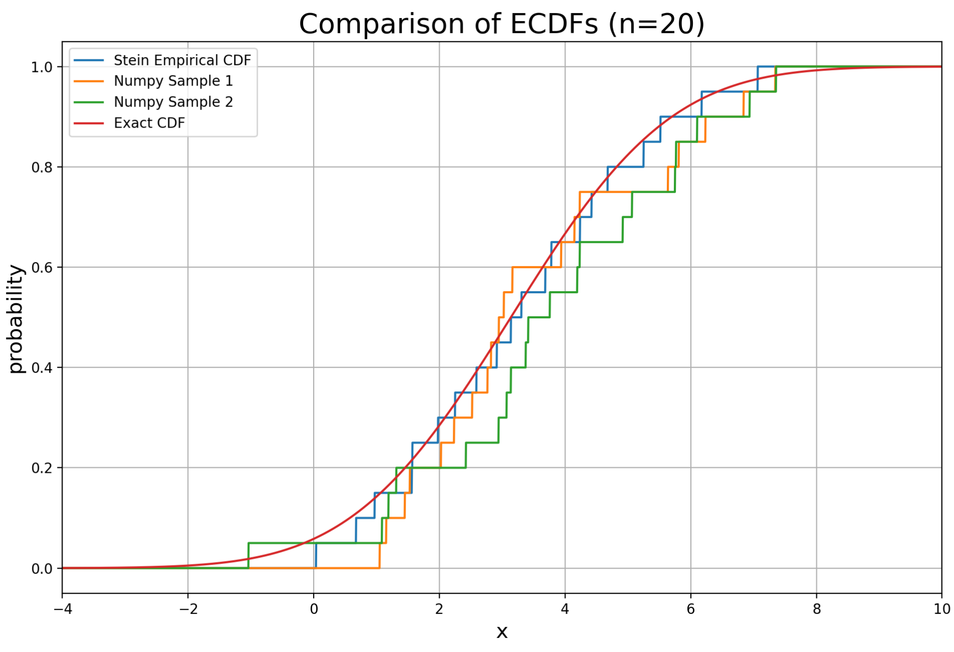


Figure 2: Empirical CDF Comparison for n=20

The effectiveness of the Stein Points is even more clear after repeating the experiment with $n = 50$.

```
 **** Found Stein Point 2 at z=1.978 after 0:00:00.944684 (H:M:S) ****
 **** Found Stein Point 3 at z=4.234 after 0:00:02.308628 (H:M:S) ****
 ...
 ...
 **** Found Stein Point 48 at z=3.438 after 0:09:26.052780 (H:M:S) ****
 **** Found Stein Point 49 at z=1.642 after 0:09:50.222214 (H:M:S) ****
 **** Found Stein Point 50 at z=4.417 after 0:10:08.421444 (H:M:S) ****
Computation complete.  Found:
 -- 50 Stein Points.
 -- Wall Time: 0:10:08.421573 (H:M:S)
```

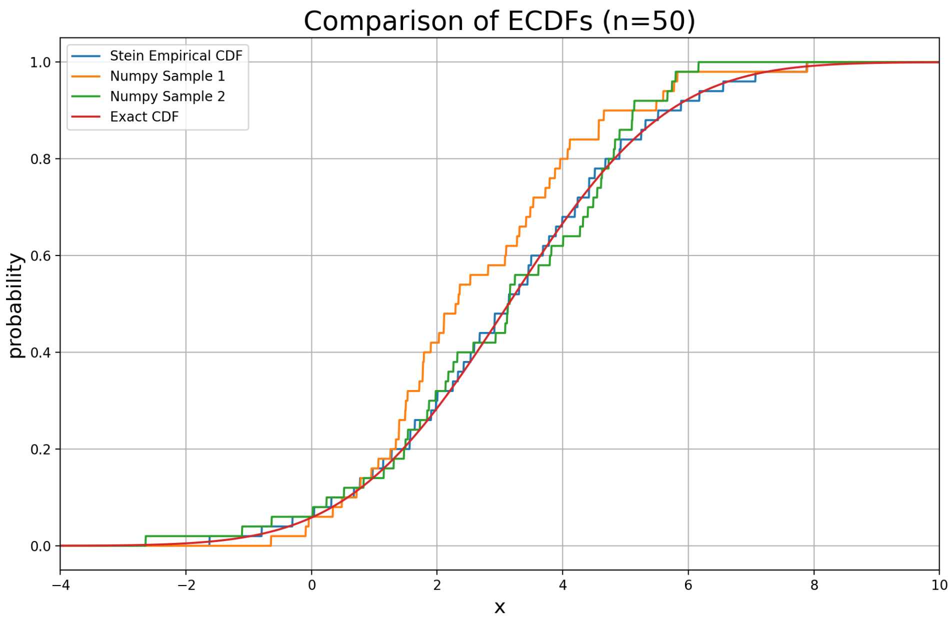


Figure 3: Empirical CDF Comparison for n=50

I note that the greedy objective appears to get progressively slower as we compute more points. This makes sense as the objective requires repeated re-calculation of k_0 against previous points. This appears to be a possible optimisation strategy for the future. For example, when we minimize the objective for point x_2 we evaluate

$$\frac{1}{2}k_0(z,z) + k_0(x_1,z)$$

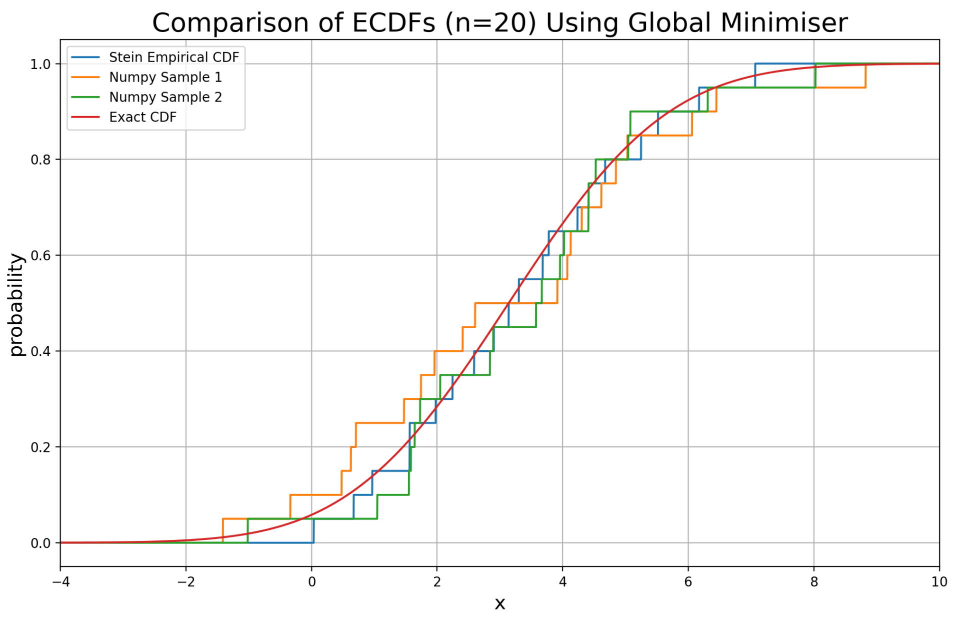
but for each subsequent x_i the expression $k_0(x_1,z)$ will be evaluated again. It may be faster to evaluate $k_0(x_1,z)$ at more points than required by the 2^{nd} iteration of the minimiser, but save and tabulate them to be interpolated against later. Further, say $x_{10} \approx x_1$, then can we interpolate from $k(x_1,z)$ to $k(x_{10},z)$?

Testing a scipy Global Optimiser

The `scipy` package provides a suite of global optimizers in the `scipy.optimize` namespace. An example is the `dual_annealing` routine which is used below. This routine appears to return essentially the identical points to the direct search method above, and produces output for $n = 20$

```
 **** Found Stein Point 2 at z=1.976 after 0:00:00.185857 (H:M:S) ****
 **** Found Stein Point 3 at z=4.234 after 0:00:00.406755 (H:M:S) ****
 ...
 ...
 **** Found Stein Point 18 at z=4.414 after 0:00:10.198553 (H:M:S) ****
 **** Found Stein Point 19 at z=3.300 after 0:00:11.266055 (H:M:S) ****
 **** Found Stein Point 20 at z=1.562 after 0:00:12.393725 (H:M:S) ****
Computation complete.  Found:
 -- 20 Stein Points.
 -- Wall Time: 0:00:12.393767 (H:M:S)
```

The resulting ECDF is identical to that produced in the direct search in the line above.



Example 2.1: Bivariate Normal Distribution

Here we continue on from Example 1.1 and compute Stein Points for a multivariate normal in $\mathbb{R}^2$. This is an interesting example for investigating more general formulas for the SRK and log gradients, as well as how `scipy` routines cope with the nasty optimisation problem in $d = 2$. The SRK is

$$k_0(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d=2} \left(\frac{\partial^2 k}{\partial x_i \partial y_i} + \frac{\partial k}{\partial x_i}\frac{\partial g}{\partial y_i} + \frac{\partial k}{\partial y_i}\frac{\partial g}{\partial x_i} + k(\mathbf{x}, \mathbf{y})\frac{\partial g}{\partial x_i}\frac{\partial g}{\partial y_i} \right) \tag{9}$$

the kernel derivatives are

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

and finally the log gradients are

$$\nabla \log p(\mathbf{z}) = -(\mathbf{z} - \mu)^T \Sigma^{-1} \tag{10}$$

These formulas are implemented in `stein/notes/learning/mvn/mvn.py` and are analgous to those computed in `uvn.py` discussed previously. The output is vectorised but the component wise nature of the calculations is clear.

The d=2 Greedy Objective Function

As one expects the objective function in two dimensions is complicated. In this example I take

$$\mu = (-0.5, 0.5)^T \tag{11}$$

and correlation

$$\Sigma = \begin{pmatrix} 1.0 \; 0.5 \\ 0.5 \; 2.0 \end{pmatrix} \tag{12}$$

As per the one dimension examples I choose the initial Stein Point to be the mode $\mathbf{x}_1 = \mu$, and use the `scipy` `scipy.optimize.dual_annealing` minimiser which appears to also work well in two dimension. The following plot shows contours of the objective for the first eight Stein Points (after the initial choice) and the points chosen by the minimiser are the red crosses. These points appear plausible (without doing rigorous checking).

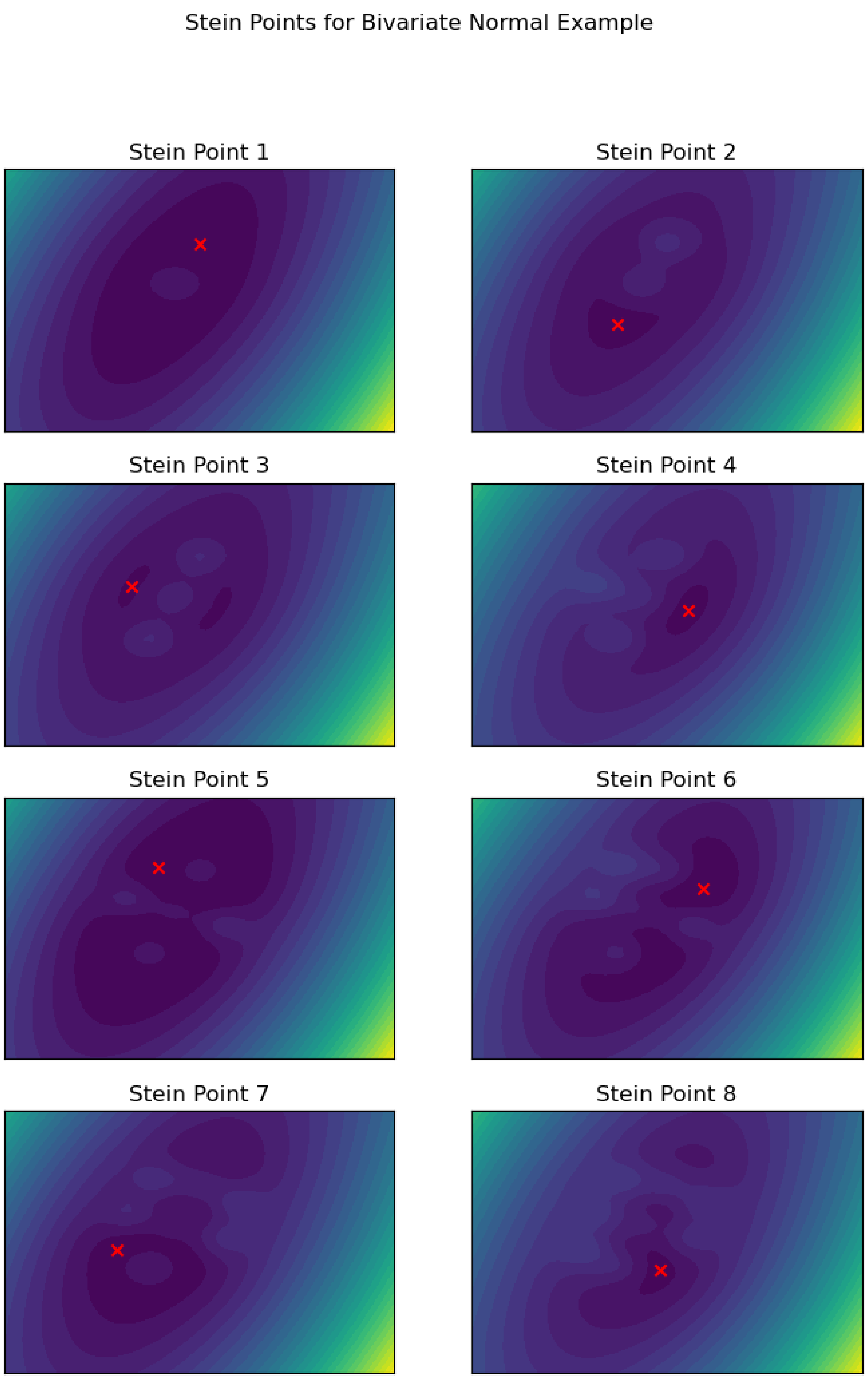


Figure 4: Countours of the Greedy Objective for the Bivariate Normal Target

The First n=20 Stein Points for the Bivariate Normal

Using the same bivariate normal example from above, I compute the first 20 Stein Points. This is found in the script `normal1-5.py`, which produces output like

```
*** Scipy found point (0.01. 1.72) after 0:00:03.419930 (H:M:S)
*** Scipy found point (-1.01. -0.72) after 0:00:04.777972 (H:M:S)
*** Scipy found point (-1.38. 0.86) after 0:00:05.857179 (H:M:S)
*** Scipy found point (0.43. 0.11) after 0:00:06.803905 (H:M:S)
*** Scipy found point (-0.83. 1.87) after 0:00:08.909907 (H:M:S)
*** Scipy found point (0.75. 1.20) after 0:00:09.986271 (H:M:S)
...
...
```

A contour plot of the target pdf and $n = 20$ Stein Points is plotted below.

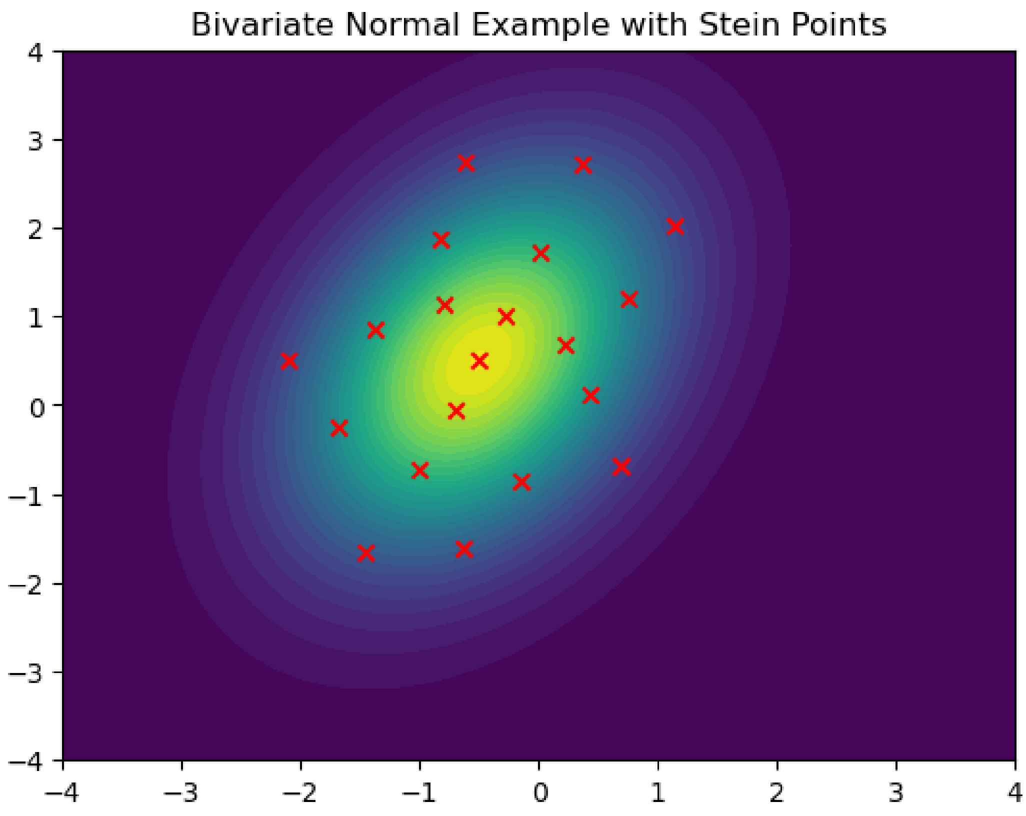


Figure 5: Countours of the Target PDF and Computed Stein Points

Example 2.2 Computing the KSD and Starting a Python Stein Point Library

The last example in this set is to outline the start of a simple Stein Point library in python. This is found at `stein/code/python/lib` and currently consists of four groups of files:

- `kernel.py` and `kernel_base.py` for defining general kernal functions $k(\mathbf{x}, \mathbf{y})$
- `target.py` and `target_base.py` for defining general targets $p(\mathbf{x})$
- `stein.py` where generic KSD, SRK and greedy objectives are defined
- `util.py` where a couple of utility routines I seem to use regularly are kept.

The script `stein/code/python/example/mvn1.py` demonstrates how to use the library, and provides an example of using kernals, targets and `scipy` global optimization routines to compute Stein Points. It is run (for $n = 25$ points for example) like

```
python mvn1.py --n 25
```

where `PYTHONPATH=<path_to_stein>/stein/code/python/lib` should be set for the imports required. This script produces output like

```
$ python mvn1.py --n 25
Computing first 25 Stein Points for MVN Example 1.
 ** Stein Point (1) found in 0:00:00.755472 (H:M:S) at x: [-0.50001774
    ↪ 0.49998333]
 ** Stein Point (2) found in 0:00:02.494034 (H:M:S) at x: [0.00604887
    ↪ 1.72167402]
 ** Stein Point (3) found in 0:00:03.273162 (H:M:S) at x: [-1.00729827
    ↪ -0.72474337]
 ** Stein Point (4) found in 0:00:04.227296 (H:M:S) at x: [0.37904274
    ↪ 0.13130274]
 ** Stein Point (5) found in 0:00:04.875321 (H:M:S) at x: [-1.43584947
    ↪ 0.88528364]
 ** Stein Point (6) found in 0:00:03.568377 (H:M:S) at x: [0.7054779
    ↪ 1.23707363]
 ** Stein Point (7) found in 0:00:01.400034 (H:M:S) at x: [-0.88527567
    ↪ 1.87834818]
 ....
```

and writes a plot of the target pdf, Stein Points, and KSD which is shown below:

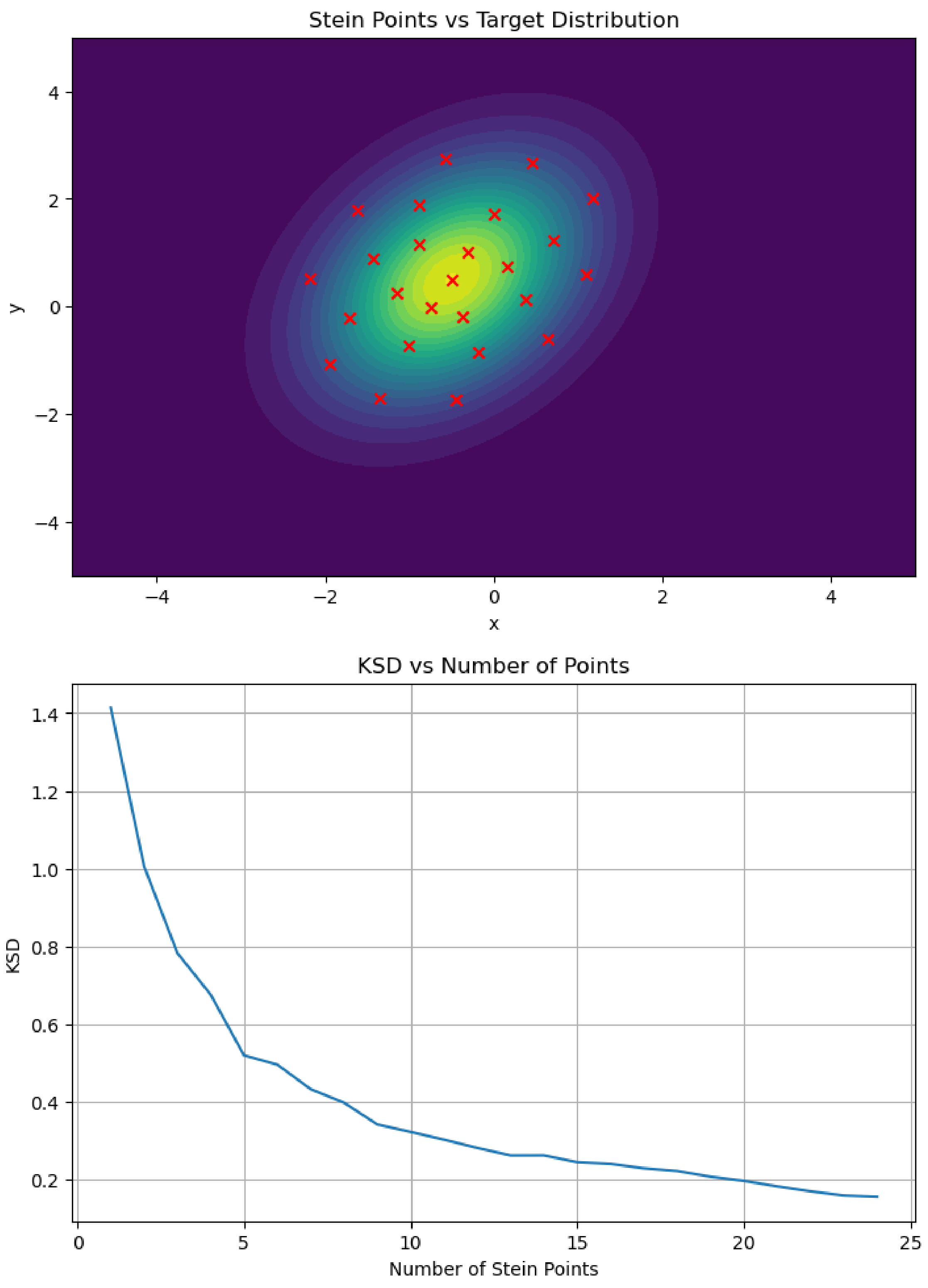


Figure 6: Stein Points and KSD from Small Python Library