# Meeting : 2 May 2019

Kristyn Pantoja

5/1/2019

# Last Meeting

# Last Time

- We saw that `seq` function makes the minimum energy designs generated from the fast algorithm and the one-at-a-time algorithm look more similar to each other (but still not quite the same!)
- We also talked about how the design is possibly not unique!
- We saw some histograms, summary statistics, evaluations of the designs

# Today’s Meeting

# Goals

- Try parallel and reproducible code in rmarkdown!
- Mainly, though, our goal is to evaluate the model based on:
  1. Robustness
  2. Objective Improvement
  3. Limiting Behavior

# Stuff from Last Time

# Revisit Comparisons of 2 Algorithms' Design Outputs

- Parameters

```
mean_beta0 = 1 # slope of null model
mean_beta1 = 1 / 2 # slope of alternative model
var_mean = 0.001 # variance on beta
var_e = 0.01 # variance on error
```

- Settings

```
N = 67
# for fast algorithm:
K = 40 # ceiling(4* sqrt(p))
numParameters = 1 # number of parameters (just slope!)
p = numParameters * 2
# for one-at-a-time algorithm:
numCandidates = 10^5 # suggested 10^5
k = 4
```

## Timing each:

```
## Fast Algorithm
set.seed(1)
ptm <- proc.time()
X_test_efficient = SMED_ms_fast(mean_beta0 = mean_beta0, mean_beta1 = mean_beta1, var_e = var_e, var_mean
proc.time() - ptm # elapsed = 581.156
```

```
##    user  system elapsed
## 606.625   4.684 623.369
```

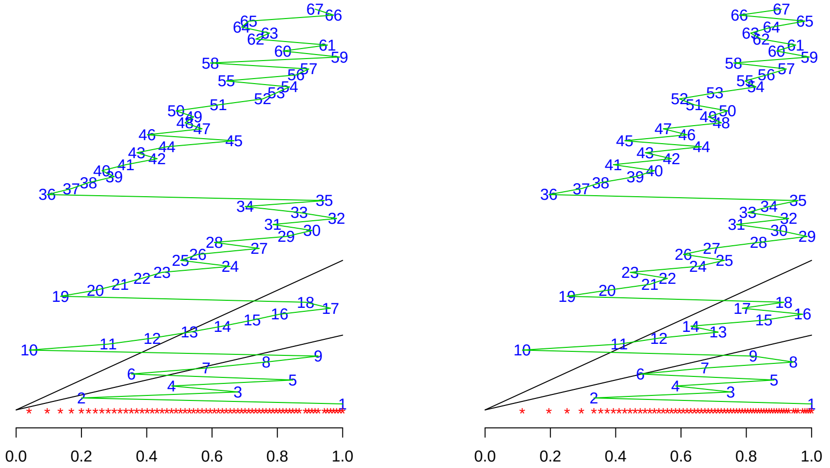
```
# One-at-a-Time Algorithm
set.seed(1)
ptm <- proc.time()
X_test_1atatime = SMED_ms(mean_beta0 = mean_beta0, mean_beta1 = mean_beta1, var_e = var_e, var_mean = var_
proc.time() - ptm # 1022.885
```

```
##     user   system  elapsed
## 1077.992    6.938 1119.272
```

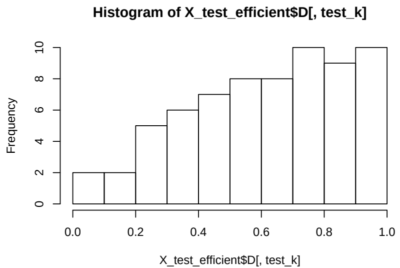
- note: don't actually need seed here
- also the "fast" algorithm could possibly be faster by matricizing and Rcpp

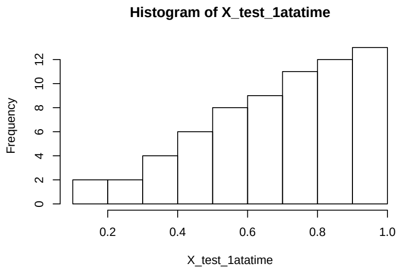
# Designs

Designs for Fast Algorithm and One-at-a-Time Algorithm, resp. Look similar!



# Histograms of Design Points




```
# Fast Algorithm
mean(X_test_efficient$D[ ,test_k])
```

```
## [1] 0.6138501
```

```
sd(X_test_efficient$D[ ,test_k])
```

```
## [1] 0.2551005
```

```
# One-at-a-Time Algorithm
mean(X_test_1atatime)
```

```
## [1] 0.6882634
```

```
sd(X_test_1atatime)
```

```
## [1] 0.2182556
```

# Fast Algorithm : Distances b/t Points and Evaluations

```r
sum(diffX); sum(diffY) # 0.9599299, 20.5807
```

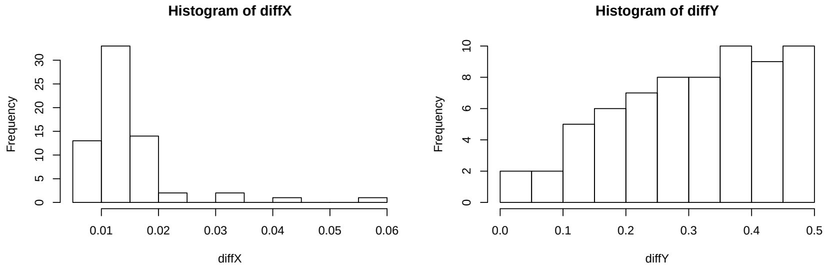
```
## [1] 0.959982

## [1] 20.56398
```

```r
mean(diffX); mean(diffY) # 0.01454439, 0.3071747
```

```
## [1] 0.01454518

## [1] 0.3069251
```

```r
sd(diffX); sd(diffY) # 0.007635743, 0.1278703
```

```
## [1] 0.007676371

## [1] 0.1275502
```



# One-at-a-Time : Distances b$\widehat{P}$oints and Evaluations

```r
sum(diffX); sum(diffY) # 0.9599299, 20.5807
```

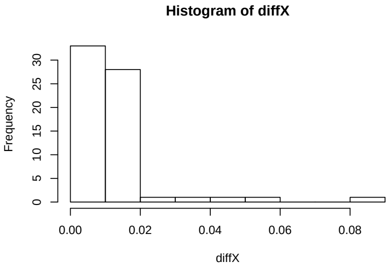
```
## [1] 0.8859489

## [1] 23.05683
```

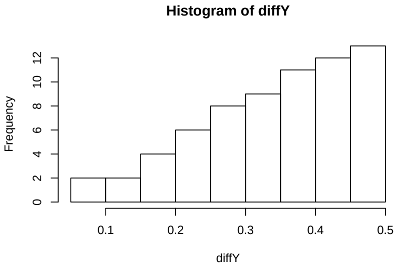
```r
mean(diffX); mean(diffY) # 0.01454439, 0.3071747
```

```
## [1] 0.01342347

## [1] 0.3441317
```

```r
sd(diffX); sd(diffY) # 0.007635743, 0.1278703
```

```
## [1] 0.01187598

## [1] 0.1091278
```



# Fast Algorithm : Bayes Factors

## From 1000 simulations of $Y|H_0$

```
# calculate expected marginal y for each model by averaging
expected_marginalY_H0 = mean(marginalY_H0) # 2.756918
expected_marginalY_H1 = mean(marginalY_H1) # 0.6176464

# calculate expected posterior probability of each model (equal prior on models)
expected_post_H0 = expected_marginalY_H0 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.81697
expected_post_H1 = expected_marginalY_H1 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.18303

BayesFactor_01 = expected_marginalY_H0 / expected_marginalY_H1
BayesFactor_01 # since 4.536117 > 1, H0 is supported
```

```
## [1] 4.485721
```

## From 1000 simulations of $Y|H_1$

```
# calculate expected marginal y for each model by averaging
expected_marginalY_H0 = mean(marginalY_H0) # 0.6140411
expected_marginalY_H1 = mean(marginalY_H1) # 2.768687

# calculate expected posterior probability of each model (equal prior on models)
expected_post_H0 = expected_marginalY_H0 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.1815225
expected_post_H1 = expected_marginalY_H1 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.8184775

BayesFactor_01 = expected_marginalY_H0 / expected_marginalY_H1
BayesFactor_01 # since 0.2217807 < 1, H0 is not supported
```

```
## [1] 0.2198247
```

# One-at-a-Time Algorithm : Bayes Factors

## From 1000 simulations of $Y|H_0$

```
# calculate expected marginal y for each model by averaging
expected_marginalY_H0 = mean(marginalY_H0) # 2.746371
expected_marginalY_H1 = mean(marginalY_H1) # 0.3862057

# calculate expected posterior probability of each model (equal prior on models)
expected_post_H0 = expected_marginalY_H0 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.8767131
expected_post_H1 = expected_marginalY_H1 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.1232869

BayesFactor_01 = expected_marginalY_H0 / expected_marginalY_H1
BayesFactor_01 # since 7.11116 > 1, H0 is supported
```

```
## [1] 7.179607
```

## From 1000 simulations of $Y|H_1$

```
# calculate expected marginal y for each model by averaging
expected_marginalY_H0 = mean(marginalY_H0) # 0.3827373
expected_marginalY_H1 = mean(marginalY_H1) # 2.758056

# calculate expected posterior probability of each model (equal prior on models)
expected_post_H0 = expected_marginalY_H0 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.1218601
expected_post_H1 = expected_marginalY_H1 / (expected_marginalY_H0 + expected_marginalY_H1) # 0.8781399

BayesFactor_01 = expected_marginalY_H0 / expected_marginalY_H1
BayesFactor_01 # since 0.1387706 < 1, H0 is not supported
```

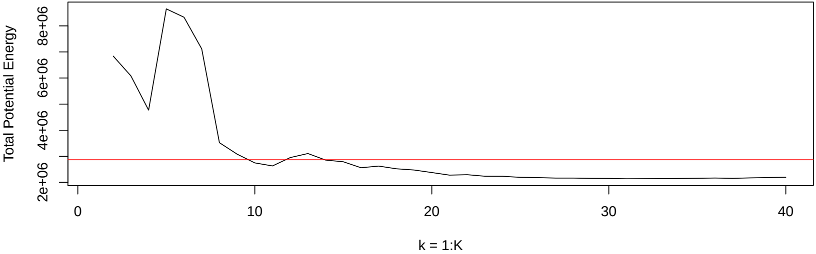
```
## [1] 0.1372213
```

# Robustness & Objective Improvement

# Evaluating the Criterion

The total potential energy is given by:

$$\sum_{i \neq j} \frac{q(\mathbf{x}_i)q(\mathbf{x}_j)}{d(\mathbf{x}_i, \mathbf{x}_j)}$$




- For `K = 40`, it seems that the efficient algorithm is better at minimizing the criterion than the one-at-a-time algorithm
- Also looks like it starts to stabilize by `k = 20`

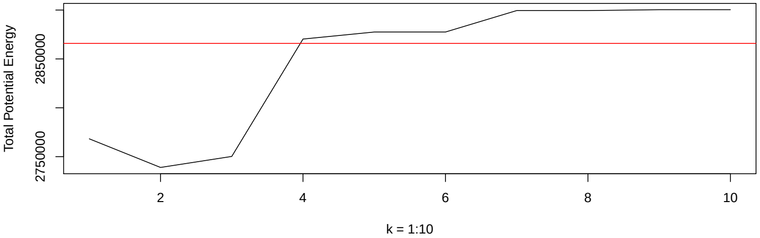
# Robustness / Objective Improvement : One-at-a-Time Algorithm

- When `k = 4` (as suggested in Joseph et. al. 2015), the Total Potential Energy criterion evaluates to:

```
totalPE_1atatime
```

```
## [1] 2865942
```

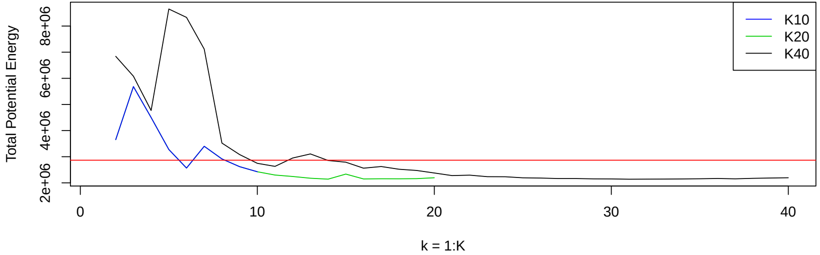
- But what happens when we change this value?
- When `k = 1:10` we get:




- Somehow, `k = 2` is better. (`k = 4` was a heuristic anyways.)

# Robustness / Objective Imrovement : Fast Algorithm

- Compare their criterion evaluations at the `K`th design:




```
totalPE_each_k[40]
```

```
## [1] 2194655
```

```
totalPE_K20_each_k[20]
```

```
## [1] 2194592
```

```
totalPE_K10_each_k[10]
```

```
## [1] 2424573
```

# Limiting Behavior

# Computational issues

- Well, we saw that, for the one-at-a-time algorithm, when `k` is large, there were some computational issues: it was getting stuck at particular locations.
- This seemed to be the case in both the original implementation (for densities) and for the model selection case.
- We talked about potentially working around this by mixing the two algorithms:
  - use the criterion of the one-at-a-time algorithm
  - with the method of the fast algorithm (re: annealing)

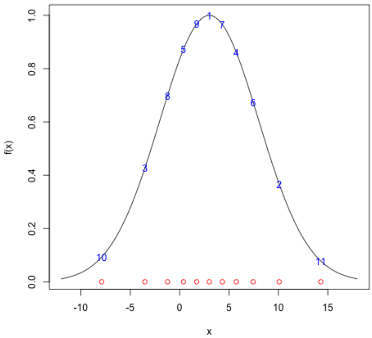
One-at-a-Time Algorithm: choose the next $(n+1)$th design point:

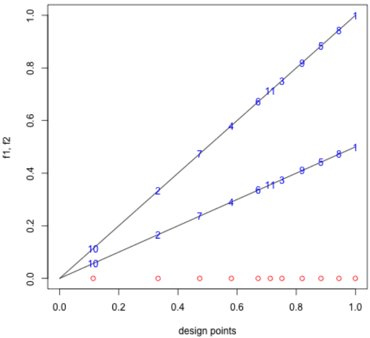
$$\mathbf{x}_{n+1} = \arg\min_{\mathbf{C}} \sum_{i=1}^{n} \left( \frac{q(\mathbf{x}_i) q(\mathbf{x})}{d(\mathbf{x}_i, \mathbf{x})} \right)^k$$

Fast Algorithm: At design $k \in 2 : (K-1)$, choose the location of $x_j$ in the $(k+1)$th design:

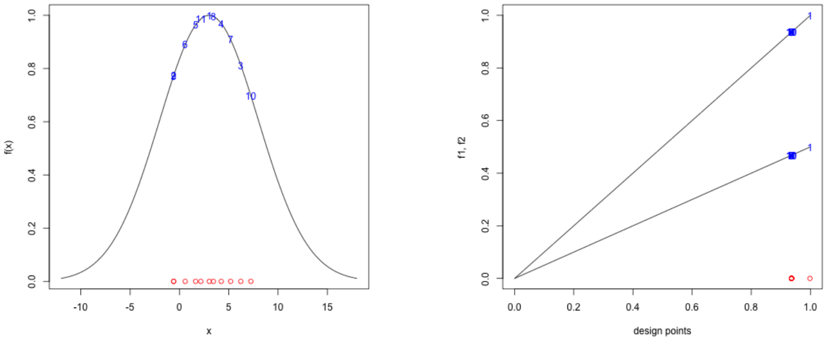
$$\mathbf{x}_j^{k+1} = \arg\min_{\mathbf{x} \in \mathbf{C}_{k+1}^j} \max_{i=1:(j-1)} \frac{1}{f^{\gamma_k}(\mathbf{x}_i) f^{\gamma_k}(\mathbf{x}) d(\mathbf{x}_i, \mathbf{x})}$$

# When k = 4



# When k = 5000



# Revisiting the Mixed Algorithm Idea

- Mixing the two algorithms:
  - use the criterion of the one-at-a-time algorithm
  - with the method of the fast algorithm (re: annealing)

One-at-a-Time Algorithm: choose the next $(n+1)$th design point:

$$\mathbf{x}_{n+1} = \arg\min_{\mathbf{C}} \sum_{i=1}^{n} \left( \frac{q(\mathbf{x}_i) q(\mathbf{x})}{d(\mathbf{x}_i, \mathbf{x})} \right)^k$$

Fast Algorithm: At design $k \in 2 : (K-1)$, choose the location of $x_j$ in the $(k+1)$th design:

$$\mathbf{x}_j^{k+1} = \arg\min_{\mathbf{x} \in \mathbf{C}_{k+1}^j} \max_{i=1:(j-1)} \frac{1}{f^{\gamma_k}(\mathbf{x}_i) f^{\gamma_k}(\mathbf{x}) d^{2p}(\mathbf{x}_i, \mathbf{x})}$$

- But what would this look like? Idea: at design $k \in 2 : (K-1)$, choose the location of $x_j$ in the $(k+1)$th design:

$$\mathbf{x}_j^{k+1} = \arg\min_{\mathbf{x} \in \mathbf{C}_{k+1}^j} \sum_{i=1}^{j-1} \left( \frac{1}{f^{\gamma_k}(\mathbf{x}_i) f^{\gamma_k}(\mathbf{x}) d^{2p}(\mathbf{x}_i, \mathbf{x})} \right)^k$$