Data Driven Models for Engineering Problems
2nd Assignment

Nikos Stamatis
MSc Student
Graduate Programme in Data Science and Machine Learning
SN: 03400115
nikolaosstamatis@mail.ntua.gr

1) The unknown quantities are the temperatures at each grid point. We first unravel the points of the 39×39 grid into a one dimensional vector $x = (x_1, \ldots, x_{39 \cdot 39})$ as follows: The first 39 entries of x contain the first line of the grid, the following 39 entries the second grid line and so on. In a more concrete format, if we denote the grid points as $y_{i,j}$ for $i, j = 1, \ldots, 39$, then $x_m = y_{k+1,l}$, where $m = 39k + l$ and $l = 1, \ldots, 39$.

Having defined the vector $x \in \mathbb{R}^{39\times 39}$ it is easy to derive the form of the matrix K that corresponds to the desired system $Kx = b$:

```
K = np.zeros((39*39, 39*39))
for x in range(39*39):
  for y in range(39*39):
    if x==y: K[x,y] = 4
    elif y==x+1: K[x,y] = -1
    elif y==x-1: K[x,y] = -1
    elif y==x+39: K[x,y] = -1
    elif y==x-39: K[x,y] = -1

K_inv = LA.inv(K)
```

Next we define the vector b. We allow the parameter r to vary, as we intend to simulate several vectors b that correspond to different values of it.

```
def candle(x, x_0=np.array([0.55, 0.45]), r=1):
  # The heat source function.

  y = x - x_0
  return 100 * np.exp(-np.dot(y, y) / r)
```

```
def create_b(rr, x_0=np.array([0.55, 0.45])):
  # Creates the vector b for a given value of r.

  b = np.zeros(39*39)
  count = 0
  for x in range(39):
    for y in range(39):
      z = (1/40) * np.array([x+1, y+1])
      b[count] = (1/40) * (1/40) * candle(z, x_0, r=rr)
      count = count + 1
  return(b)
```

2) For the simulation part, we rely on the following two functions. They both draw values for r according to the given distribution, create the vector b and then they solve the system $Kx = b$. For each value of r, the first function only returns x_{761}, which is the temperature at the center of the plate, whereas the second one returns the whole solution vector x.

```
def simulation(n_samples=10000):
  # Simulates n_samples values for b and computes
  # the temperature at point (0.5, 0.5).

  result = np.zeros(n_samples)
  mu, sigma = 0.05, 0.005
  for i in range(n_samples):
    r = np.random.normal(mu, sigma, 1)
    b = create_b(r)
    b.shape = (1521,1)
    sol = np.matmul(K_inv, b)
    result[i] = sol[760]
    if sol[760] == np.inf: print(r)
  return(result)

def full_simulation(n_samples=10000):
  # Simulates n_samples values for b and computes
  # the temperature at every point of the grid.

  simulation_results = np.zeros((n_samples, 39*39))
  mu, sigma = 0.05, 0.005
  for i in range(n_samples):
    r = np.random.normal(mu, sigma, 1)
    b = create_b(r)
    b.shape = (1521,1)
    sol = np.matmul(K_inv, b)
    simulation_results[i, :] = sol.T

  return(simulation_results)
```

The parameter r was simulated from $N(0.05, 0.005^2)$, instead of the given distribution $N(0.05, 0.005)$ for both mathematical and physical reasons. A glance at the formula $f(x) = 100\exp\left(-\frac{\|x-x_0\|^2}{r}\right)$, where $x \in \mathbb{R}^2$ and $x_0 = (0.55, 0.45)$, reveals that r should be nonnegative in order for the whole solution to make physical sense.

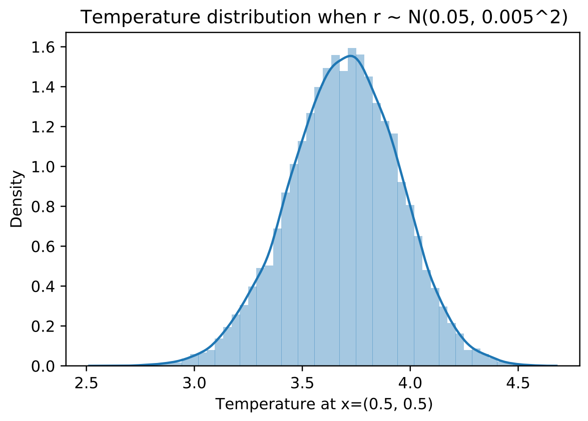

Figure 1. Temperature distribution at the center of the plate based on 10000 simulations from $r \sim N(0.05, 0.005^2)$.

Even if we ignore the physical interpretation and attempt to solve the system for negative values of r, python overflow limitations won't allow us to do so for $r \in I = (-0.00077, 0)$.

Drawing from the original distribution $X \sim N(0.05, 0.005)$ results to an extremely high probability of picking a value in I even for moderate sample sizes. Some obvious remedies include drawing from $|X|$, X^2 or even the truncated normal distribution, where negative values are being discarded, until a nonnegative sample appears.

However, given the output temperature histograms and density plot of the aforementioned distributions, the distribution $N(0.05, 0.005^2)$ seemed more plausible. Additionally, $N(0.05, 0.005^2)$ has an extremely low probability of giving negative values ($p = 7.6 \cdot 10^{-24}$ for obtaining a negative value in a single draw).

3) First we create a sample of size 300 using the previously defined function full_simulation:

```
def generate_data(n_samples=5):
  # Generates a small sample of solutions.

  data = np.zeros((n_samples, 39*39))
  for i in range(n_samples):
    data[i, :] = full_simulation(1)
  return data

hello = generate_data(300)
```

Then, we apply the PCA method on the created dataset.

```
pca_full = PCA(n_components=1, random_state=42)
pca_full.fit(hello)
pca_full.explained_variance_ratio_ * 100
```

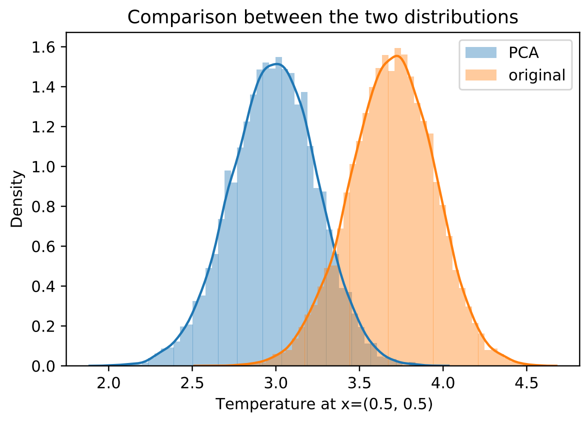

Figure 2. Comparison of the temperature distributions between the regular simulation method and the reduced PCA method, based on 10000 simulations each.

We observe that 99.99% of the variance is explained using just one component. The eigenvector can be found using the .components_ function:

```
phi = pca_full.components_
K_red = np.matmul(K, phi.T)
K_red = np.matmul(phi, K_red)
K_red_inv = 1 / K_red
```

The simulations are performed with the following function:

```
def simulation_pca(n_samples=10000):
  result = np.zeros(n_samples)
  mu, sigma = 0.05, 0.005
  U = np.matmul(phi.T, phi)
  for i in range(n_samples):
    r = np.random.normal(mu, sigma, 1)
    b = create_b(r)
    b.shape = (1521,1)
    W = np.matmul(U, b)
    W = K_red_inv * W
    result[i] = W[760]
    if W[760] == np.inf: print(r)
  return(result)

pca_sim = simulation_pca(10000)
```

We observe a high discrepancy between the two distributions (Figure 2). Although the variances are the same ($v_{\text{reg}} = v_{\text{PCA}} = 0.065$), their means differ significantly ($\mu_{\text{reg}} = 3.69$ vs $\mu_{\text{PCA}} = 2.99$.)