Inverse Square Wave Blind Source Separation

Alexander Glandon
aglan001@odu.edu

Abstract: This work describes a method for blind signal separation with no knowledge of the signal characteristics. We prove the zero knowledge separation using simulation that converges on random signal sources with 0 autocorrelation.

This work is the completion of an idea developed under the supervision of Khan Iftekharuddin.

Inverse square waves decay at 1/r^2, where r is the radius from the source.

We perform a simulation on signals with no prior knowledge, proving results for a general class of signals. Blind signal separation convergence is obtained on signals with 0 autocorrelation.

The idea is given in [1], but the approach is modified from gradient descent to root finding. This update allowed us to achieve convergence in simulation, which was not done using the theory in [1].

Consider sources 1,2,...,N. The sources emit signals Qnt with values at time steps 1,2,..,T. This is equally valid for samples at t=(time×frequency) steps. We will use the notation Qnt to refer to either case.

Consider M receivers (M>=4 for uniqueness given geometric considerations). The receivers sample signals Pmt at time steps 1,2,...,T.

The receiver locations are known as Rmc, where c is dimension index 1, 2, or 3 of our coordinate system.

The source locations are unknown as Snc.

We wish to estimate Qnt given the receiver locations and the receiver samples.

The distance between known receiver locations and unknown source locations is
Dmn = (Σ(Rmc-Snc)^2)^0.5 [eq 1]

Given this formulation, for inverse square waves
P=UQ, where Umn=Dmn^-2

Naive matrix factorization of UQ has m×n + n×t unknowns and m×t knowns.

Using the constraint in eq 1, we reduce the number of unknowns to n×3 + n×t.

This allows a unique solution as we show in simulation using nonlinear least squares to find the roots of f(S,Q)=P-U(S)Q.

[1] Glandon, Alexander M. "Recurrent neural networks and matrix methods for cognitive radio spectrum prediction and security." (2017).

Appendix

Here is the code for inverse square wave blind source separation.

```
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('Agg')
import numpy as np
from scipy.optimize import least_squares
from sklearn.decomposition import PCA

# 8 of receivers in a cube
M = 8
R = np.zeros(shape=(M,3))
R[:,0] = [0, 0, 0, 0, 1, 1, 1, 1]
R[:,1] = [0, 0, 1, 1, 0, 0, 1, 1]
R[:,2] = [0, 1, 0, 1, 0, 1, 0, 1]
R = R

# number of sources, 10 would be a hard separation problem
N = 5

# 100 measurements over time
T = 100

# 100 ground truth source power samples
Q_target = np.square(np.random.normal(loc=0,scale=1,size=(N,T))) # maybe I don't need to square if I'm not trying NNMF (power not needed, can use raw signal)
S_target = np.random.uniform(low=0,high=1,size=(N,3))

# 100 receieved power samples
D_target = np.zeros((M,N))
for m in range(M):
  for n in range(N):
    D_target[m,n] = np.power(np.sum(np.square(R[m,:]-S_target[n,:])),0.5)
```

```
P = np.matmul(np.divide(1,np.square(D_target)),Q_target)


# solution:

def f(X_guess):
  S_vector = X_guess[0:(N*3)]
  Q_vector = X_guess[N*3:]

  S_guess = S_vector.reshape(N,3)
  Q_guess = Q_vector.reshape(N,T)

  U_guess = np.zeros((M,N))
  for m in range(M):
    for n in range(N):
      U_guess[m,n] = np.power(np.sum(np.square(R[m,:]-S_guess[n,:])),-1)

  error = P-np.matmul(U_guess,Q_guess)
  error_vector = error.reshape((M*T,))
  return error_vector



# check that f(x) == 0
S_vector = S_target.reshape((N*3,))
Q_vector = Q_target.reshape((N*T,))
X_target = np.concatenate((S_vector,Q_vector))
errors = f(X_target)
print(0.5*np.sum(np.square(errors)))


#max_nfev = 100

X0 = np.random.normal(loc=0,scale=1,size=(N*(T+3),))
X_guess=least_squares(fun=f,x0=X0,verbose=2)


S_vector = X_guess['x'][0:(N*3)]
Q_vector = X_guess['x'][N*3:]
S_guess = S_vector.reshape(N,3)
Q_guess = Q_vector.reshape(N,T)

# check that f(x) == 0
```

```
S_guess_vector = S_guess.reshape((N*3,))
Q_guess_vector = Q_guess.reshape((N*T,))
X_guess = np.concatenate((S_vector,Q_vector))
errors = f(X_guess)
print(0.5*np.sum(np.square(errors)))

# find solution permutation
S_temp = np.zeros((N,3))
Q_temp = np.zeros((N,T))
used_indices = []
for n in range(N):
  best_score = 0
  for n_guess in range(N):
    already_used = False
    for n_used in used_indices:
      if n_guess == n_used:
        already_used = True
        break
    if not already_used:
      score = np.power(np.sum(np.square(S_target[n,:]-S_guess[n_guess,:])),-1)
      if score > best_score:
        best_score = score
        best_index = n_guess

  S_temp[n,:] = S_guess[best_index,:]
  Q_temp[n,:] = Q_guess[best_index,:]

S_guess = S_temp
Q_guess = Q_temp


# check that f(x) == 0
S_guess_vector = S_guess.reshape((N*3,))
Q_guess_vector = Q_guess.reshape((N*T,))
X_guess = np.concatenate((S_vector,Q_vector))
errors = f(X_guess)
print(0.5*np.sum(np.square(errors)))


#print solution

#print("number of iterations = ", max_nfev)
S_target_array = []
for n in range(N):
```

```
  for c in range(3):
    S_target_array.append(S_target[n,c])

S_guess_array = []
for n in range(N):
  for c in range(3):
    S_guess_array.append(S_guess[n,c])


Q_target_array = []
for n in range(N):
  for t in range(T):
    Q_target_array.append(Q_target[n,t])

Q_guess_array = []
for n in range(N):
  for t in range(T):
    Q_guess_array.append(Q_guess[n,t])

plt.plot(Q_target_array, label = "Source Power Target")
plt.plot(Q_guess_array, label = "Source Power Prediction")
plt.xlabel("Time")
plt.ylabel("Power")
plt.title("Source Power Error all Transmitters")
plt.legend()
plt.show()
plt.savefig("Source Power Error all Transmitters.png")
plt.close()


plt.plot(S_target_array, label = "Source Locations Target")
plt.plot(S_guess_array, label = "Source Locations Prediction")
plt.xlabel("All Sources and Spatial Dimensions")
plt.ylabel("Spatial Position")
plt.title("Source Locations Error all Transmitters")
plt.legend()
plt.show()
plt.savefig("Source Locations Error all Transmitters.png")
plt.close()

pca = PCA(n_components=2)
pca.fit(S_target)
S_guess_2D = pca.transform(S_guess)
S_target_2D = pca.transform(S_target)
```

```
for n in range(N):
  plt.scatter(x=[S_target_2D[n,0],S_guess_2D[n,0]],y=[S_target_2D[n,1],S_guess_2D[n,1]],
label="Source "+str(n))
plt.xlabel("Spatial Dimension 1")
plt.ylabel("Spatial Dimension 2")
plt.title("Source Locations Error Projection in 2D Space")
plt.legend()
plt.show()
plt.savefig("Source Locations Error Projection in 2D Space.png")
plt.close()

for n in range(N):
  plt.plot(Q_target[n,:], label = "Source Power Target")
  plt.plot(Q_guess[n,:], label = "Source Power Prediction")
  plt.xlabel("Time")
  plt.ylabel("Power")
  plt.title("Source "+str(n+1)+" Power Error")
  plt.legend()
  plt.show()
  plt.savefig("Source "+str(n+1)+" Power Error.png")
  plt.close()
```