A toy example

Purpose of this demo: Motivate that abstract notions, such as sparse projection, are useful in practice.

- Disclaimer: I'm not expert in Python - I use Python/Matlab as tools to validate algorithms and theorems.
- Thus, my implementations are not the most efficient ones + there might be bugs

Problem definition: Linear regression.

$$y = Ax^\star + w$$

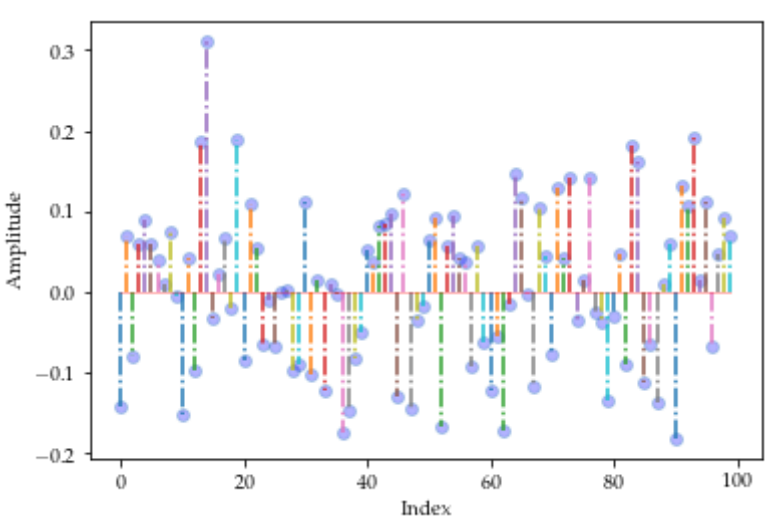
- $A \in \mathbb{R}^{n \times p}$
- $x^\star \in \mathbb{R}^p$
- $w \in \mathbb{R}^n$

Assume $n = p$, and A is in general position. Given y and A:

$$\min_{x \in \mathbb{R}^p} \qquad f(x) \triangleq \|y - Ax\|_2^2$$

```
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import random
from scipy import stats
from scipy.optimize import fmin

from matplotlib import rc
#rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
## for Palatino and other serif fonts use:
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

from PIL import Image

import random
from numpy import linalg as la

p = 100  # Ambient dimension
n = 100  # Number of samples

# Generate a p-dimensional zero vector
x_star = np.random.randn(p)
# Normalize
x_star = (1 / la.norm(x_star, 2)) * x_star


# Plot
xs = range(p)
markerline, stemlines, baseline = plt.stem(xs, x_star, '-.')
plt.setp(markerline, 'alpha', 0.3, 'ms', 6)
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'color', 'r', 'linewidth', 1, 'alpha', 0.3)
plt.rc('text', usetex=True)
```

```
#plt.rc('font', family='serif')
plt.xlabel('Index')
plt.ylabel('Amplitude')
plt.show()
```



How would you solve this problem?

Closed form solution using matrix inverse

$$\widehat{x} = A^{-1} y$$

```
A = np.random.randn(n, p)
y = A.dot(x_star)

A_inv = la.inv(A)
widehat_x = A_inv.dot(y)
# Plot
xs = range(p)
markerline, stemlines, baseline = plt.stem(xs, widehat_x, '-.')
plt.setp(markerline, 'alpha', 0.3, 'ms', 6)
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'color', 'r', 'linewidth', 1, 'alpha', 0.3)
plt.xlabel('Index')
plt.ylabel('Amplitude')
plt.show()

print('\|x^\star - x\|_2 = {0}'.format(la.norm(x_star - widehat_x)))
```

Problem definition: Sparse linear regression.

$$y = Ax^\star + w$$

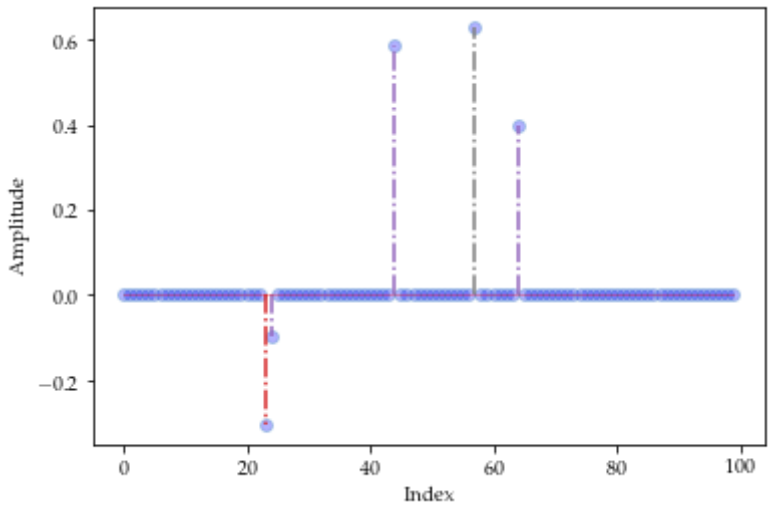
- $A \in \mathbb{R}^{n \times p}$, but now $n \ll p$
- $x^\star \in \mathbb{R}^p$ but k-sparse, where $k \ll p$
- $w \in \mathbb{R}^n$

Assume $n = p$, and A is in general position. Given y and A:

$$\min_{x \in \mathbb{R}^p} \quad f(x) \triangleq \|y - Ax\|_2^2$$

Would a similar technique solve the problem?

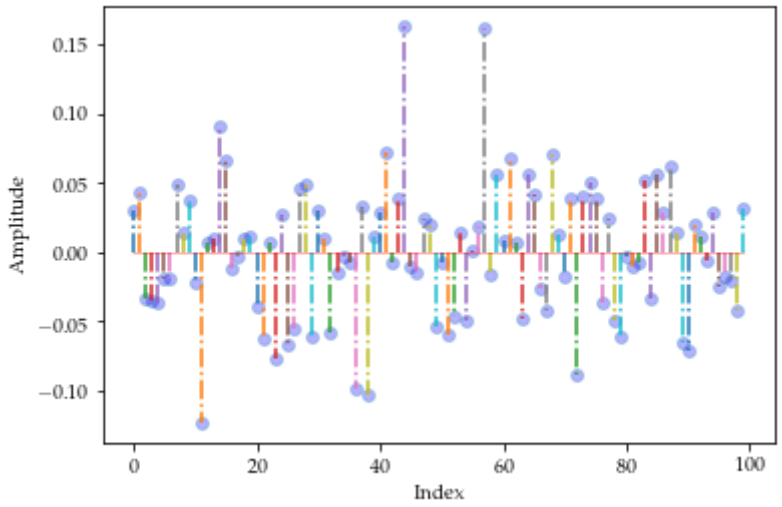
```
p = 100  # Ambient dimension
n = 40   # Number of samples
k = 5    # Sparsity level

# Generate a p-dimensional zero vector
x_star = np.zeros(p)
# Randomly sample k indices in the range [1:p]
x_star_ind = random.sample(range(p),  k)
# Set x_star_ind with k random elements from Gaussian distribution
x_star[x_star_ind] = np.random.randn(k)
# Normalize
x_star = (1 / la.norm(x_star, 2)) * x_star

# Plot
xs = range(p)
markerline, stemlines, baseline = plt.stem(xs, x_star, '-.')
plt.setp(markerline, 'alpha', 0.3, 'ms', 6)
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'color', 'r', 'linewidth', 1, 'alpha', 0.3)
plt.xlabel('Index')
plt.ylabel('Amplitude')
plt.show()
```



We will use the pseudo-inverse of A:

$$A^\dagger = A^\top (AA^\top)^{-1}$$

```
A = np.random.randn(n, p)
y = A.dot(x_star)

A_inv = la.pinv(A)
widehat_x = A_inv.dot(y)
# Plot
xs = range(p)
markerline, stemlines, baseline = plt.stem(xs, widehat_x, '-.')
plt.setp(markerline, 'alpha', 0.3, 'ms', 6)
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'color', 'r', 'linewidth', 1, 'alpha', 0.3)
plt.xlabel('Index')
plt.ylabel('Amplitude')
plt.show()

la.norm(x_star - widehat_x)
```




```
0.8712800492517689
```

- The reconstruction of $x^\star$ from y is an ill-posed problem since $n < p$ and there is no hope in finding the *true vector* without ambiguity.
- Additional prior information is needed.
- We might want to use the fact that $\|x\|_0 \leq k$ where $k \ll p$ and $\|\cdot\|_0$ is the ℓ_0-"norm".
- It turns out that, under proper assumptions on the sensing matrix A and the sparsity level k, one can still recover $x^\star$!

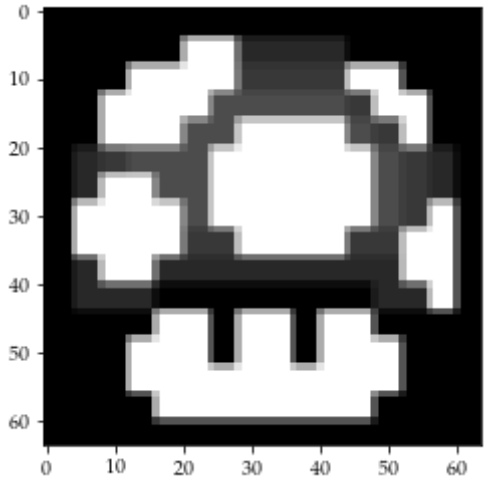
Why sparsity?

Let us consider the following practical case: image processing.

```
%matplotlib inline
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import random
from scipy import stats
from scipy.optimize import fmin

from PIL import Image

# Open image using Image package
x_mush_orig = Image.open("./SupportFiles/mushroom.png").convert("L")
```

```
# Transform to a np array
x_mush_star = np.fromstring(x_mush_orig.tobytes(), np.uint8)
# Set the shape of np array
x_mush_star.shape = (x_mush_orig.size[1], x_mush_orig.size[0])
# Show the image
plt.imshow(x_mush_star, interpolation = "nearest", cmap = plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x1a13cd69e8>
```



Obviously, this is a simple image case: the "mushroom" image is sparse by itself (do you see the black pixels? Yes, they are zeros). To see this more clearly, let's sort the true coeffients in decreasing order.

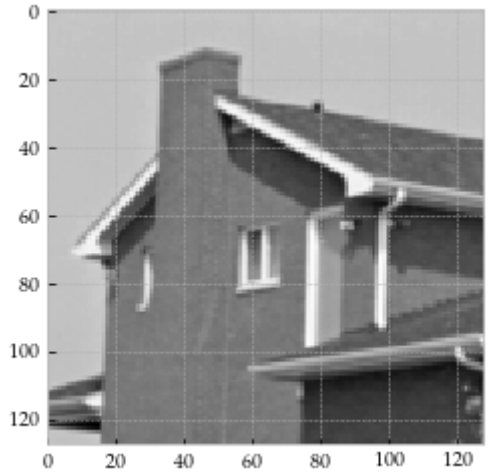
```
from bokeh.plotting import figure, show, output_file
from bokeh.palettes import brewer

# Get the absolute value of a flatten array (vectorize)
x_mush_abs = abs(x_mush_star.flatten())
# Sort the absolute values (ascending order)
x_mush_abs.sort()
# Descending order
x_mush_abs_sort = np.array(x_mush_abs[::-1])

plt.style.use('bmh')
fig, ax = plt.subplots()
# Generate an array with elements 1:len(...)
xs = np.arange(len(x_mush_abs_sort))
# Fill plot - alpha is transparency (might take some time to plot)
ax.fill_between(xs, 0, x_mush_abs_sort, alpha = 0.2)
# Plot - alpha is transparency (might take some time to plot)
ax.plot(xs, x_mush_abs_sort, alpha = 0.8)
plt.show()
```

For this 64 x 64 image, the total number of pixels sums up to 4096. As you can observe, by default almost half of the pixels are zero, which constitutes "mushroom" image sparse (but still the sparsity level is quite high: more than half the ambient dimension).

Since this seems to be a "cooked"-up example, let us consider a more *realistic* scenario: a brick house. (*Does anyone know where is this house?*)

```
x_house_orig = Image.open("./SupportFiles/house128.png").convert("L")
x_house_star = np.fromstring(x_house_orig.tobytes(), np.uint8)
x_house_star.shape = (x_house_orig.size[1], x_house_orig.size[0])
plt.imshow(x_house_star, interpolation = "nearest", cmap = plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x1a16735358>
```

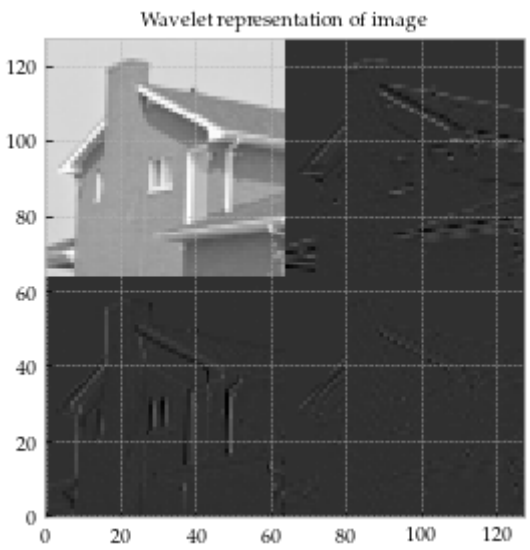


...and here is the bar plot of the coefficients.

```
x_house_abs = abs(x_house_star.flatten())
x_house_abs.sort()
x_house_abs_sort = np.array(x_house_abs[::-1])

plt.style.use('bmh')
fig, ax = plt.subplots()
xs = np.arange(len(x_house_abs_sort))
ax.fill_between(xs, 0, x_house_abs_sort, alpha = 0.2)
plt.plot(xs, x_house_abs_sort, alpha=0.8)
plt.show()
```

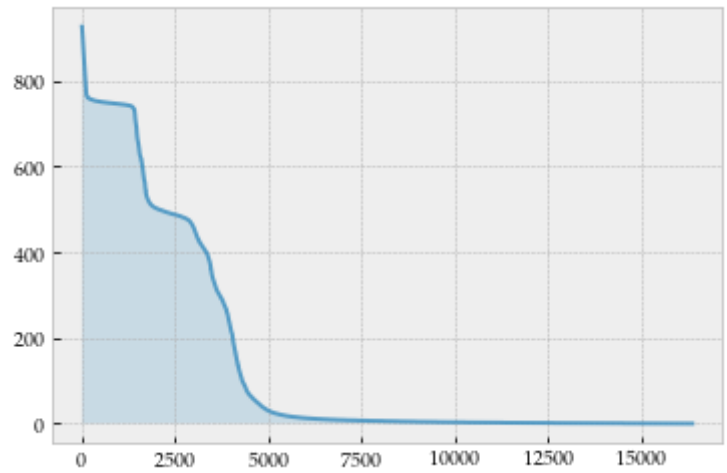
- All the coefficients are non-zero! Is there anything we can do in this case?
- However: under proper orthonormal transformations, natural images become sparse.

```
import pywt

x_house_orig = Image.open("./SupportFiles/house.png").convert("L")
x_house_star = np.fromstring(x_house_orig.tobytes(), np.uint8)
x_house_star.shape = (x_house_orig.size[1], x_house_orig.size[0])

# Defines a wavelet object - 'db1' defines a Daubechies wavelet
wavelet = pywt.Wavelet('db1')

# Multilevel decomposition of the input matrix
coeffs = pywt.wavedec2(x_house_star, wavelet, level=2)
cA2, (cH2, cV2, cD2), (cH1, cV1, cD1) = coeffs

# Concatenate the level-2 submatrices into a big one and plot
x_house_star_wav = np.bmat([[cA2, cH2], [cV2, cD2]])
plt.imshow(np.flipud(x_house_star_wav), origin='image', interpolation="nearest", cmap=
plt.title("Wavelet representation of image", fontsize=10)
plt.tight_layout()
```



After wavelet transformation, let's plot the wavelet coefficients.

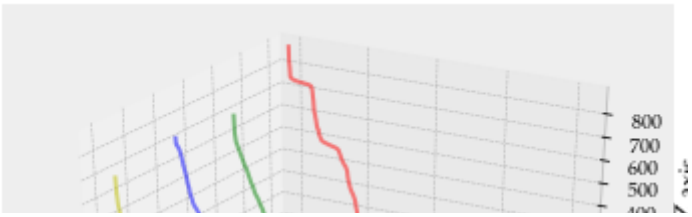
```
# Flatten and show the histogram
x_house_abs_wav = abs(x_house_star_wav.flatten())
x_house_abs_wav.sort()
x_house_abs_wav.flatten()
x_house_abs_wav_sort = np.array(x_house_abs_wav[::-1])

plt.style.use('bmh')
fig, ax = plt.subplots()
xs = np.arange(len(x_house_abs_wav_sort.flatten()))

ax.fill_between(xs, 0, np.flipud(x_house_abs_wav_sort.flatten()), alpha = 0.2)
plt.plot(xs, np.flipud(x_house_abs_wav_sort.transpose()), alpha = 0.8)
plt.show()
```



It is obvious that much less number of coefficients are non-zero! (...and this holds generally for naturally images.)

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
for c, z, zi in zip(['r', 'g', 'b', 'y'], ['./SupportFiles/house128.png', './SupportFi
    y = Image.open(z).convert("L")
    y_star = np.fromstring(y.tobytes(), np.uint8)
    y_star.shape = (y.size[1], y.size[0])

    # Multilevel decomposition of the input matrix
    y_coeffs = pywt.wavedec2(y_star, wavelet, level=2)
    y_cA2, (y_cH2, y_cV2, y_cD2), (y_cH1, y_cV1, y_cD1) = y_coeffs

    # Concatenate the level-2 submatrices into a big one and plot
    y_star_wav = np.bmat([[y_cA2, y_cH2], [y_cV2, y_cD2]])

    y_abs_wav = abs(y_star_wav.flatten())
    y_abs_wav.sort()
    y_abs_wav.flatten()
    y_abs_wav_sort = np.array(y_abs_wav[::-1])

    xs = np.arange(len(y_abs_wav_sort.flatten()))
    cs = c
    ys = [zi] * len(xs)
    ys = np.array(ys)

    ax.plot(xs, ys = ys.flatten(), zs = np.flipud(y_abs_wav_sort.flatten()), zdir = 'z

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show
```

```
<function matplotlib.pyplot.show(*args, **kw)>
```



In the above picture, the y values (1.0 to 4.0) correspond to four different image cases (for chanity check, observe that the red curve is the same curve for the house.png case, presented above).

One can observe that most of the coeffs are close to zero and only few of them (compared to the ambient dimension) are significantly large. This has led to the observation that keeping only the most important coefficients (even truncating the non-zero entries further) leads to a significant compression of the image. At the same time, only these coefficients can lead to a pretty good reconstruction of the original image.

Using sparse projections

```
import math

# Generate sensing matrix
A = (1 / math.sqrt(n)) * np.random.randn(n, p)

# Observation model
y = A @ x_star
```

Gradient descent with sparse projections[7-8]. Solve the criterion

$$\min_x \quad f(x) := \frac{1}{2}\|y - Ax\|_2^2 \quad \text{s.t.} \quad \|x\|_0 \leq k$$

The IHT method

1: Choose initial guess x_0
2: **for** i = 0, 1, 2, ... **do**
3: Compuete $\nabla f(x_i) = -A^\top \cdot (y - Ax_i)$
4: $\widehat{x}_{i+1} = x_i - \nabla f(x_i)$
5: $x_{i+1} = \arg\min_{x \text{ is } k-\text{sparse}} \|\widehat{x}_{i+1} - x\|_2$
5: **end for**

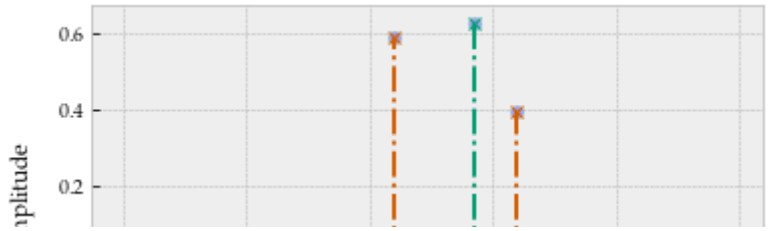
Let's use this algorithm and see how it performs in practice.

```
from numpy import linalg as la

# Hard thresholding function
def hardThreshold(x, k):
    p = x.shape[0]
    t = np.sort(np.abs(x))[::-1]
    threshold = t[k-1]
    j = (np.abs(x) < threshold)
    x[j] = 0
    return x

# Returns the value of the objecive function
def f(y, A, x):
    return 0.5 * math.pow(la.norm(y - A @ x, 2), 2)
```

```
def IHT(y, A, k, iters, epsilon, verbose, x_star):
    p = A.shape[1]   # Length of original signal
    n = A.shape[0]   # Length of measurement vector

    x_new = np.zeros(p)    # Initial estimate
    At = np.transpose(A)   # Transpose of A

    x_list, f_list = [1], [f(y, A, x_new)]

    for i in range(iters):
        x_old = x_new

        # Compute gradient
        grad = -At @ (y - A @ x_new)

        # Perform gradient step
        x_temp = x_old - 0.5 * grad

        # Perform hard thresholding step
        x_new = hardThreshold(x_temp, k)

        if (la.norm(x_new - x_old, 2) / la.norm(x_new, 2)) < epsilon:
            break

        # Keep track of solutions and objective values
        x_list.append(la.norm(x_new - x_star, 2))
        f_list.append(f(y, A, x_new))

        if verbose:
            print("iter# = "+ str(i) + ", ||x_new - x_old||_2 = " + str(la.norm(x_new

    print("Number of steps:", len(f_list))
    return x_new, x_list, f_list
```

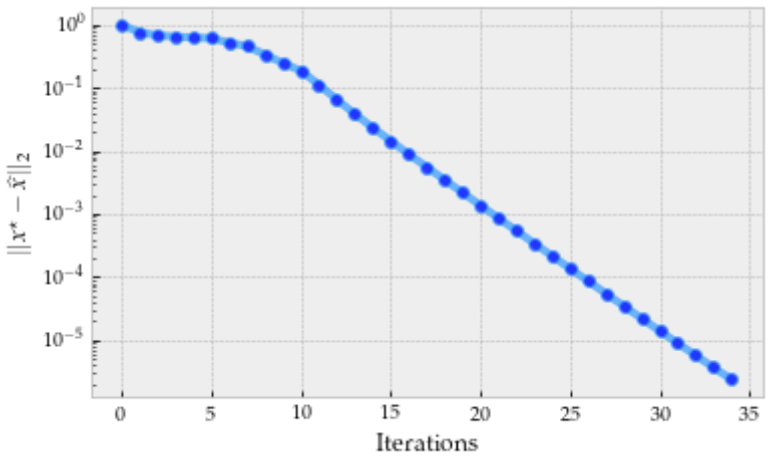
```
# Run algorithm
epsilon = 1e-6                # Precision parameter
iters = 100

x_IHT, x_list, f_list = IHT(y, A, k, iters, epsilon, True, x_star)

# Plot
plt.rc('text', usetex=True)
plt.rc('font', family='serif')

xs = range(p)
markerline, stemlines, baseline = plt.stem(xs, x_IHT, '-.x')
plt.setp(markerline, 'alpha', 0.3, 'ms', 6)
plt.setp(markerline, 'markerfacecolor', 'b')
plt.setp(baseline, 'linewidth', 1, 'alpha', 0.3)
plt.xlabel('Index')
plt.ylabel('Amplitude')
#plt.title(r"$\|x^\star - \widehat{x}\|_2 = %s$" %(la.norm(x_star - x_IHT, 2)), fontsi

# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)
plt.show()
```

```
iter# = 0, ||x_new - x_old||_2 = 0.6177438798758598
iter# = 1, ||x_new - x_old||_2 = 0.26128157344738245
iter# = 2, ||x_new - x_old||_2 = 0.09361207881349505
iter# = 3, ||x_new - x_old||_2 = 0.04317711913212753
iter# = 4, ||x_new - x_old||_2 = 0.027527665985595813
iter# = 5, ||x_new - x_old||_2 = 0.1762479884559013
iter# = 6, ||x_new - x_old||_2 = 0.09143705811393904
iter# = 7, ||x_new - x_old||_2 = 0.1624625362367286
iter# = 8, ||x_new - x_old||_2 = 0.09667822951381577
iter# = 9, ||x_new - x_old||_2 = 0.06807979010614675
iter# = 10, ||x_new - x_old||_2 = 0.08967065929274189
iter# = 11, ||x_new - x_old||_2 = 0.04901045819886067
iter# = 12, ||x_new - x_old||_2 = 0.02697251556412546
iter# = 13, ||x_new - x_old||_2 = 0.015463272065968231
iter# = 14, ||x_new - x_old||_2 = 0.009139788634701333
iter# = 15, ||x_new - x_old||_2 = 0.005522395070714989
iter# = 16, ||x_new - x_old||_2 = 0.0033864055226010548
iter# = 17, ||x_new - x_old||_2 = 0.002096351483005982
iter# = 18, ||x_new - x_old||_2 = 0.0013054910680175874
iter# = 19, ||x_new - x_old||_2 = 0.000816053728203195
iter# = 20, ||x_new - x_old||_2 = 0.0005113669814988588
iter# = 21, ||x_new - x_old||_2 = 0.00032098769363336197
iter# = 22, ||x_new - x_old||_2 = 0.00020174487228618787
iter# = 23, ||x_new - x_old||_2 = 0.00012693398348486638
iter# = 24, ||x_new - x_old||_2 = 7.994107336965959e-05
iter# = 25, ||x_new - x_old||_2 = 5.039271838615745e-05
iter# = 26, ||x_new - x_old||_2 = 3.179686471715435e-05
iter# = 27, ||x_new - x_old||_2 = 2.008408028070075e-05
iter# = 28, ||x_new - x_old||_2 = 1.2700523690476049e-05
iter# = 29, ||x_new - x_old||_2 = 8.041995740638211e-06
iter# = 30, ||x_new - x_old||_2 = 5.100012942953152e-06
iter# = 31, ||x_new - x_old||_2 = 3.2401296811990317e-06
iter# = 32, ||x_new - x_old||_2 = 2.0629343032615233e-06
iter# = 33, ||x_new - x_old||_2 = 1.3168072675881744e-06
Number of steps: 35
```



This is great! IHT finds $\mathbf{x}^\star$ fast and 'accurately'. How fast? Let's create a convergence plot.

```
# Plot
plt.rc('text', usetex=True)
plt.rc('font', family='serif')

xs = range(len(x_list))
plt.plot(xs, x_list, '-o', color = '#3399FF', linewidth = 4, alpha = 0.7, markerfacecc
plt.yscale('log')
plt.xlabel('Iterations')
plt.ylabel(r"$\|x^\star - \widehat{x}\|_2$")

# Make room for the ridiculously large title.
plt.subplots_adjust(top=0.8)
plt.show()
```


10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
$||x^* - \hat{x}||_2$
0
5
10
15
20
25
30
35
Iterations