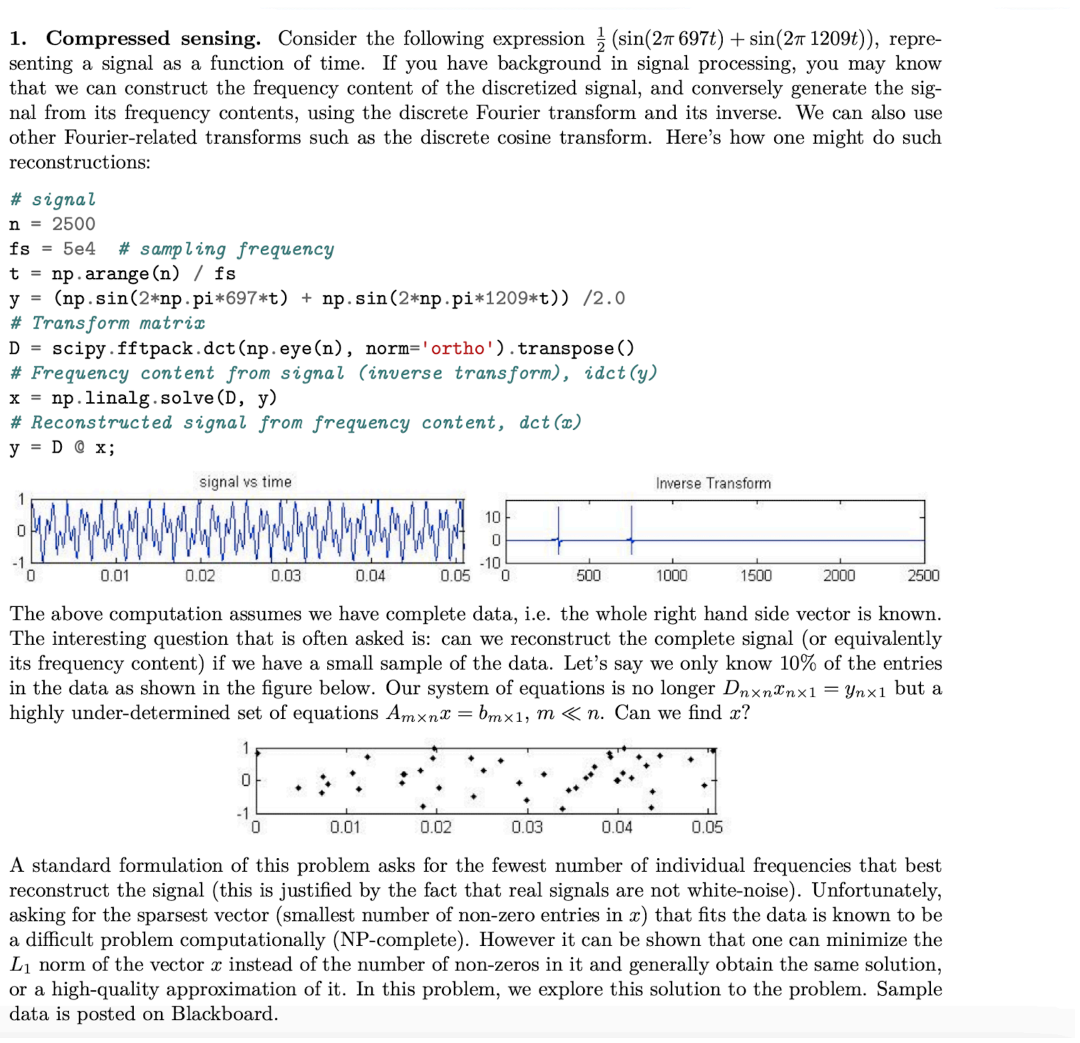
Numerical Optimization - Assignment 8

David Alvear(187594) - Nouf Farhoud(189731)

In [10]:
```python
import numpy as np
import scipy.fftpack
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import cvxpy as cp
```

1. Compressed sensing



signal vs time
Inverse Transform

1. Compressed sensing. Consider the following expression $\frac{1}{2}(\sin(2\pi\, 697t) + \sin(2\pi\, 1209t))$, representing a signal as a function of time. If you have background in signal processing, you may know that we can construct the frequency content of the discretized signal, and conversely generate the signal from its frequency contents, using the discrete Fourier transform and its inverse. We can also use other Fourier-related transforms such as the discrete cosine transform. Here's how one might do such reconstructions:

```python
# signal
n = 2500
fs = 5e4  # sampling frequency
t = np.arange(n) / fs
y = (np.sin(2*np.pi*697*t) + np.sin(2*np.pi*1209*t)) /2.0
# Transform matrix
D = scipy.fftpack.dct(np.eye(n), norm='ortho').transpose()
# Frequency content from signal (inverse transform), idct(y)
x = np.linalg.solve(D, y)
# Reconstructed signal from frequency content, dct(x)
y = D @ x;
```

The above computation assumes we have complete data, i.e. the whole right hand side vector is known. The interesting question that is often asked is: can we reconstruct the complete signal (or equivalently its frequency content) if we have a small sample of the data. Let's say we only know 10% of the entries in the data as shown in the figure below. Our system of equations is no longer $D_{n\times n}x_{n\times 1} = y_{n\times 1}$ but a highly under-determined set of equations $A_{m\times n}x = b_{m\times 1}$, $m \ll n$. Can we find x?

A standard formulation of this problem asks for the fewest number of individual frequencies that best reconstruct the signal (this is justified by the fact that real signals are not white-noise). Unfortunately, asking for the sparsest vector (smallest number of non-zero entries in x) that fits the data is known to be a difficult problem computationally (NP-complete). However it can be shown that one can minimize the L_1 norm of the vector x instead of the number of non-zeros in it and generally obtain the same solution, or a high-quality approximation of it. In this problem, we explore this solution to the problem. Sample data is posted on Blackboard.

- Formulate the problem as a least squares problem to minimize the l_2 norm of x

$$\begin{aligned} \text{minimize} \quad & \|x\|_2^2 = \sum_{i=1}^{n} x_i^2 \\ \text{subject to} \quad & Ax = b \end{aligned}$$

  - this is a quadratic optimization problem with equality-only constraints, therefore the KKT conditions are linear equations. Write and solve these equations.
  - comment on the quality of the solution you obtain.

- Formulate the problem as the problem of minimizing the l_1 norm of x

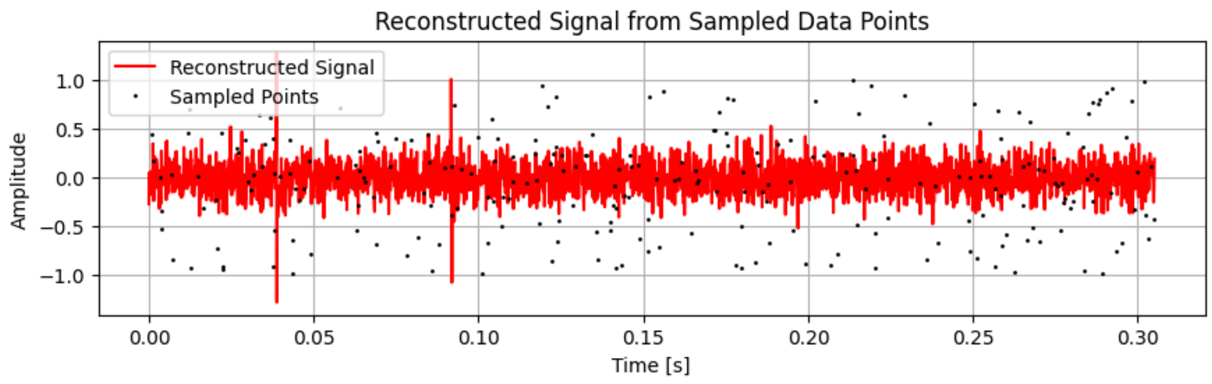
$$\begin{aligned} \text{minimize} \quad & \|x\|_1 = \sum_{i=1}^{n} |x_i| \\ \text{subject to} \quad & Ax = b \end{aligned}$$

  - show how this problem may be expressed as an LP.
  - solve the problem using `cvxpy` and comment on the quality of the solution you obtain.

$\mathcal{L}(x, \lambda) = \|x\|^2 - \lambda^\top (Ax - b)$
Solving KKT Conditions:
1 Stationarity:
$\nabla_x \mathcal{L}(x, \lambda) = 2x - A^\top \lambda = 0$
$x = \dfrac{A^\top \lambda}{2}$
2 Primal feasibility: $\quad$ Constraint should be satisfied
$Ax = b$
$A\left(\dfrac{A^\top \lambda}{2}\right) = b$
$\lambda = 2(AA^\top)^{-1} b$
We do not check duality nor complementary slackness since we just have equality

In [ ]:
```
# Seed for reproducibility
seed = 211
rng = np.random.default_rng(seed)

# Signal parameters
n = 2500  # Total number of points
m = 250   # Number of points to reconstruct
fs = 8192  # Sampling frequency

# Time vector
t = np.arange(n) / fs

# Original signal
y = (np.sin(2 * np.pi * 521 * t) + np.sin(2 * np.pi * 1233 * t)) / 2.0

# DCT matrix
D = scipy.fftpack.dct(np.eye(n), norm='ortho').transpose()

# Randomly selected indices for reconstruction
k = np.sort(rng.choice(n, m, replace=False))
```

```
A = D[k, :]
b = y[k]

# Now we solve the least squares problem: min ||x||_2 subject to Ax = b
# Using the pseudo-inverse for numerical stability
x_ls = np.linalg.pinv(A) @ b

# Plot the reconstructed signal using the least squares solution
plt.figure(figsize=(10.24, 2.56))
plt.plot(t, x_ls, 'r-', label='Reconstructed Signal')
plt.plot(t[k], b, 'ko', markersize=1, label='Sampled Points')
plt.title('Reconstructed Signal from Sampled Data Points')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)
plt.show()

# Return the least squares solution and the residual norm
x_ls, np.linalg.norm(A @ x_ls - b)
```



```
Out[ ]: (array([-0.27056606,  0.04792385, -0.06810129, ..., -0.25329519,
               -0.11469387,  0.19300071]),
         1.2166998267461423e-14)
```

- The red line represents the reconstructed signal from the sampled data points, which are shown as black dots.
- The signal appears to be very noisy, indicating that the reconstruction might not be capturing the true underlying sine waves precisely.

Expressing the problem as a linear program, we introduce an auxiliary variable u :

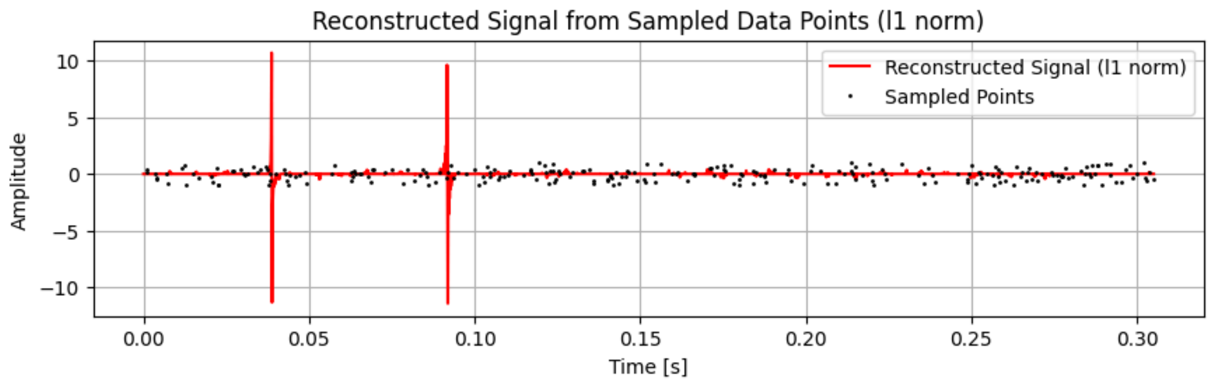
$$\min ||x||_1 = \min \sum_{i=1}^{n} |x_i| = \min \sum_{i=1}^{n} u_i$$

$$\text{s.t.} \quad -u_i \leq x_i \leq u_i, \quad \forall i = 1, \ldots, n$$

$$Ax = b$$

Given:

In [ ]:
```python
# Using previously defined variables from the user's problem setup
# A, b, n are already defined in the problem context
# A = DCT matrix at selected points, b = sampled points from the signal, n =

# Define the optimization variables
x = cp.Variable(n)
u = cp.Variable(n)

# Objective function is the sum of u which represents the l1 norm of x
objective = cp.Minimize(cp.sum(u))

# Constraints include the absolute value constraints and the equality constr
constraints = [cp.abs(x) <= u, A @ x == b]

# Define and solve the problem
problem = cp.Problem(objective, constraints)
problem.solve()

# Extract the solution if the problem was solved successfully
if problem.status not in ["infeasible", "unbounded"]:
    # The problem has an optimal solution
    x_value = x.value  # Optimal x
    # Quality of the solution
    quality_of_solution = np.linalg.norm(A @ x_value - b, ord=1)  # L1 norm
else:
    x_value = None
    quality_of_solution = "Problem is infeasible or unbounded"

# Plot the reconstructed signal using the l1 minimization solution
plt.figure(figsize=(10.24, 2.56))
plt.plot(t, x_value, 'r-', label='Reconstructed Signal (l1 norm)')
plt.plot(t[k], b, 'ko', markersize=1, label='Sampled Points')
plt.title('Reconstructed Signal from Sampled Data Points (l1 norm)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)
plt.show()

x_value, quality_of_solution, problem.status
```

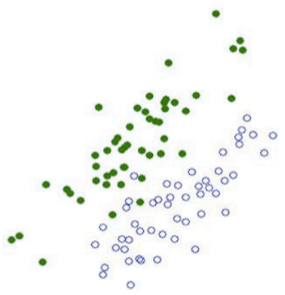


```
Out[ ]: (array([-8.30533969e-11, -3.78217609e-11, -2.76650550e-11, ...,
                 4.15790890e-12,  8.09879964e-12,  2.54794699e-11]),
         3.3242419233969756e-13,
         'optimal')
```

- The X-value(s): The numbers in the array are very close to zero, which implies that the solution is sparse.
- The L1 norm of the resiual: The second element is a very small number (3.3242419233969756e-13), representing the norm of the residual Ax−b. In the context of the optimization problem, this value measures how well the solution satisfies the constraint Ax=b. A smaller value indicates a better fit. Since this number is extremely close to zero, it suggests that the solution x found by CVXPY almost perfectly satisfies the constraint.
- 'optimal': This status indicates that the solver successfully found a solution that minimizes the objective function under the given constraints.

2. Classification

2. Classification. Given two sets of points in R^n, the classification problem asks to which set a new point belongs. The answer may be found by first finding a hyperplane $a^T x - b$ that "best" separates the two sets, being positive on the first set and negative on the second. A new point can be classified by checking which side of the hyperplane it lies on. This problem appears in a wide variety of contexts in machine learning, pattern recognition, data mining, and related domains where one seeks to learn from a set of observations or samples. The figure below shows an illustration of the problem in R^2.



- When the two sets are separable by a hyperplane $a^T x - b$, the problem can be formulated as that of finding the "thickest slab" that achieves the separation. Formulate this problem as a quadratic optimization problem of the form:

$$\begin{array}{ll} \underset{a,b}{\text{minimize}} & \frac{1}{2}a^T a \\ \text{subject to} & D(Xa - b\mathbf{1}) \geq \mathbf{1} \end{array}$$

  and use `cvxpy` or `scipy.optimize.minimize` to solve it for the data posted. Show your solution as the pair $a^T x - b = \pm 1$.

- Add a data point that prevents the sets from being separable and verify (with `scipy.optimize` or `cvxpy`) that the problem above becomes infeasible.

- In general the two sets of points cannot be separated by a hyperplane, so we seek to find the best classifier that maximizes the width of a slab that separates the two point sets while minimizing the amount of misclassification (as measured, for example, by the violation of the constraints above). A weighted combination of these objectives can be used. Formulate the problem as:

$$\begin{array}{ll} \underset{a,b,u}{\text{minimize}} & \frac{1}{2}a^T a + \alpha\mathbf{1}^T u \\ \text{subject to} & D(Xa - b\mathbf{1}) \geq \mathbf{1} - u \\ & u \geq 0 \end{array}$$

  and solve it to generate the tradeoff curve of Pareto optimal points for the given data. Comment on the shape of the curve. Plot the solution. Does it make sense?

Problem Formulation

Find the hyperplane $a^T x - b$ to find the thickest slab to separate the points.
Find function that maximizes the gap of the points x_i and y_i

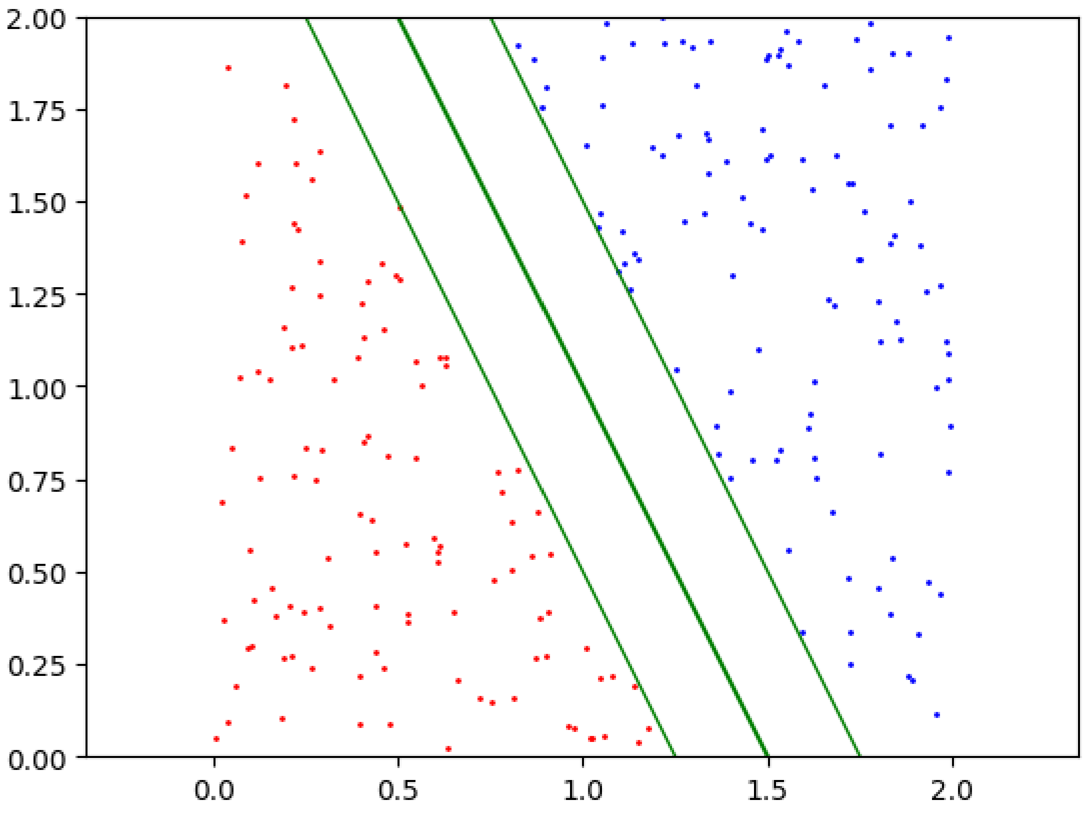
maximize t
s.t. $\quad a^T x_i - b \geq t \quad i = 1 \ldots N$
$a^T y_i - b \leq -t \quad i = 1 \ldots N$
$\|a\| = 1$

For t^* optimum we can find the line that separates x_i and y_i.
Then, at optimality
$\|a\| = 1$ a tight bound. So geometrically we can write the problem
that finds the hyperplane that separates the points. At t^* represent the half of the s

minimize $a, b \quad \frac{1}{2}\|a\| = \frac{1}{2}a^T a$
Subject to $D(Xa - b1) \leq 1$

where D is the diagonal matrix with the points class labels.
For $y_i \in \{-1, 1\}$

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt

# generate test data
seed = 211          # seed the random number generator (for repeatability)
rng = np.random.default_rng(seed)
nd = 300           # 300 random data poins
P = 2 * rng.random((nd, 2))

a = np.array([4, 2])
b = 6

blue = (P @ a.transpose() - b * np.ones(nd)) > 1.0
red  = (P @ a.transpose() - b * np.ones(nd)) < -1.0
X = np.vstack( (P[blue, :], P[red, :]) )
mblue = P[blue, :].shape[0]
mred  = P[red, :].shape[0]
m = mblue + mred                # total number of test points


# plot test data
plt.figure('Test data')
plt.plot(X[0:mblue, 0], X[0:mblue, 1], 'bo', markersize=1)
plt.plot(X[mblue:m, 0], X[mblue:m, 1], 'ro', markersize=1)

plt.plot([0.0, b/a[0]], [b/a[1], 0.0],  color='g')
plt.plot([0.0, (b+1)/a[0]], [(b+1)/a[1], 0.0],  color='g', lw = 1)
plt.plot([0.0, (b-1)/a[0]], [(b-1)/a[1], 0.0],  color='g', lw = 1)

plt.axis('equal')
plt.xlim( 0.0, 2.0 )
plt.ylim( 0.0, 2.0 )

plt.show()

# Formulate and solve the problem below
# you should try to use cvxpy or the scipy.optimize.minimize() routine.
```

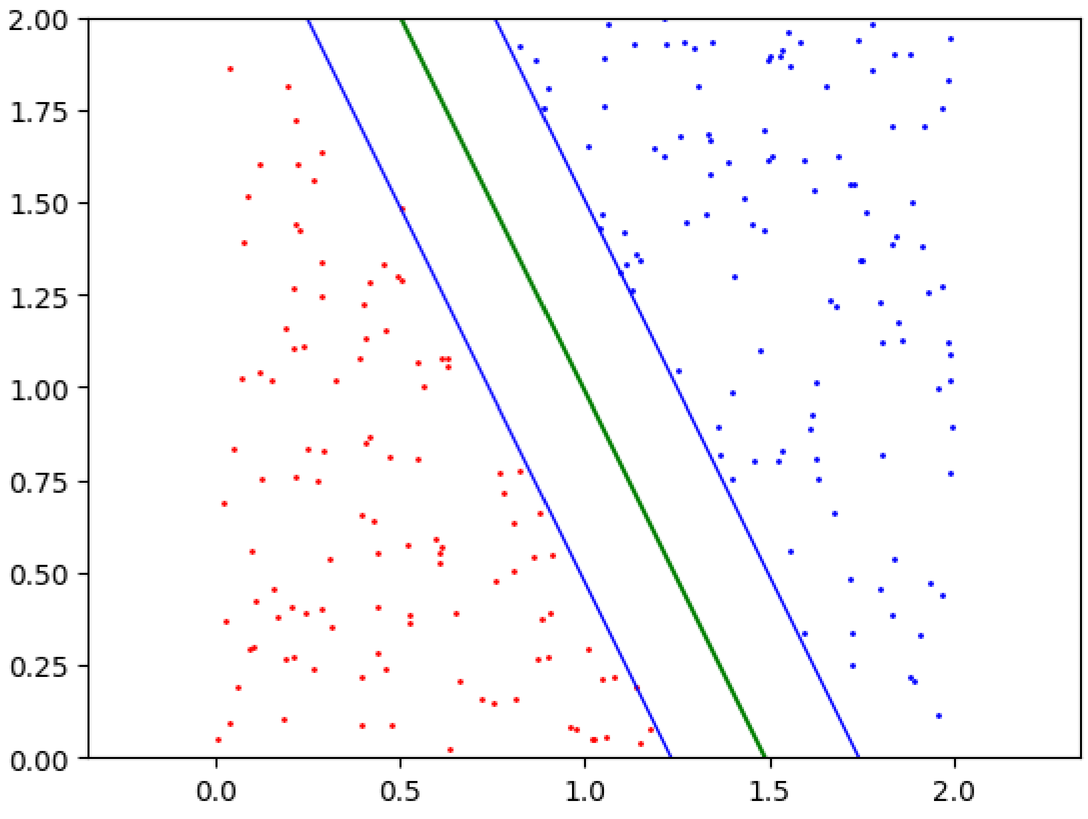


In [3]:
```
# Define the variables
# a, b variables
# X points
# D Class labels
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
a = cp.Variable(2)
b = cp.Variable()

# Define objective
objective = cp.Minimize(0.5*cp.norm(a,2)**2)

# constraints
constraints = [D @ (X @ a - b*np.ones(m)) >= 1]
# constraints = [cp.multiply(D, X @ a - b) >= 1]

# Define problem and solve
problem = cp.Problem(objective, constraints)

# Solve the problem
```

```
problem.solve()

# get the params values
a = a.value
b = b.value

print(f"Primal optimum value: {problem.value}")
print(f"Primal optimum a value: {a}")
print(f"Primal optimum b value: {b}")
```

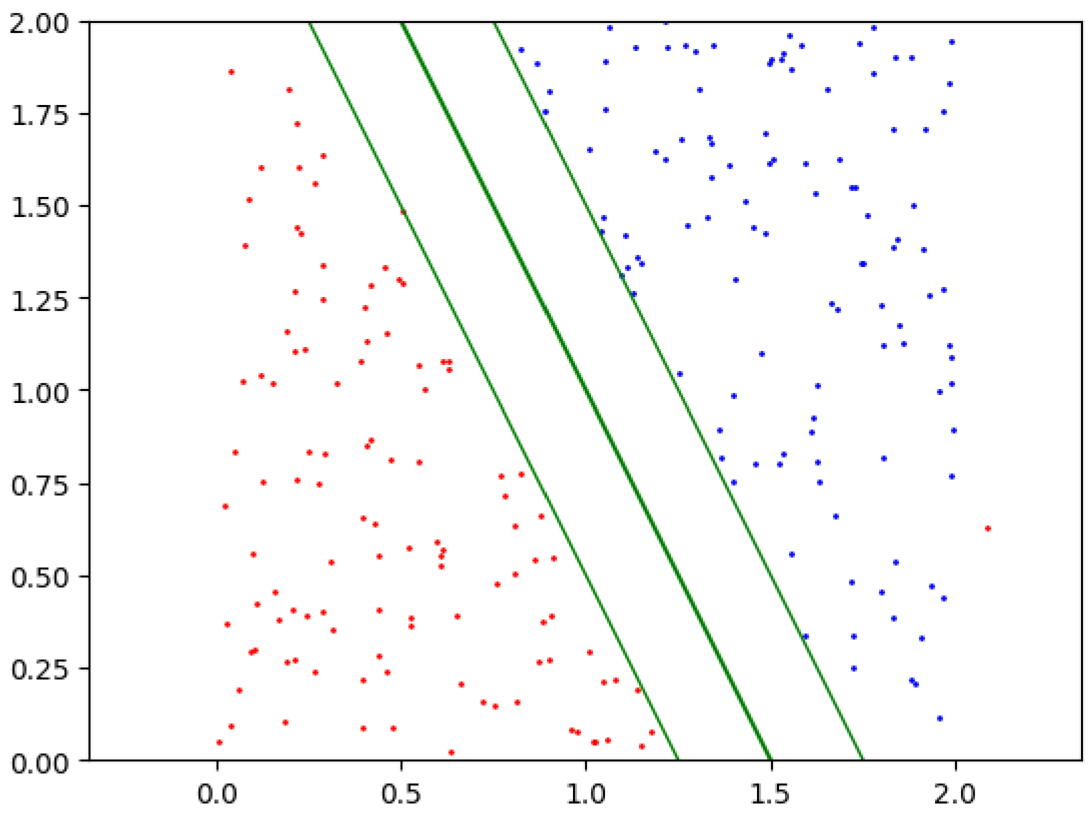
```
Primal optimum value: 9.636554516630637
Primal optimum a value: [3.93872405 1.93895897]
Primal optimum b value: 5.85729627660868
```

In [4]:
```
# plot test data
plt.figure('Test data')
plt.plot(X[0:mblue, 0], X[0:mblue, 1], 'bo', markersize=1)
plt.plot(X[mblue:m, 0], X[mblue:m, 1], 'ro', markersize=1)

plt.plot([0.0, b/a[0]], [b/a[1], 0.0],  color='g')
plt.plot([0.0, (b+1)/a[0]], [(b+1)/a[1], 0.0],  color='b', lw = 1)
plt.plot([0.0, (b-1)/a[0]], [(b-1)/a[1], 0.0],  color='b', lw = 1)

plt.axis('equal')
plt.xlim( 0.0, 2.0 )
plt.ylim( 0.0, 2.0 )

plt.show()
```



- Add a point to make the set of points non-linear separable and solve.

In [5]:
```python
import numpy as np
import matplotlib.pyplot as plt

# generate test data
seed = 211          # seed the random number generator (for repeatability)
rng = np.random.default_rng(seed)
nd = 300           # 300 random data poins
P = 2 * rng.random((nd, 2))

a = np.array([4, 2])
b = 6

blue = (P @ a.transpose() - b * np.ones(nd)) > 1.0
red  = (P @ a.transpose() - b * np.ones(nd)) < -1.0
X = np.vstack( (P[blue, :], P[red, :]) )
mblue = P[blue, :].shape[0]
mred  = P[red, :].shape[0]
m = mblue + mred                # total number of test points

# Adding a non-separable point
# Generate a random blue point
np.random.seed(seed)  # Ensure reproducibility for the example
non_separable_point = P[blue, :][np.random.choice(mblue)]
# Modify it to be inside the margin of the red class
non_separable_point += (0.15 * np.random.random(2) + 0.01) * a
# Append it to the dataset
X = np.vstack((X, non_separable_point))
# Now, this point is added to the red class
mred += 1
m += 1

# plot test data
plt.figure('Test data')
plt.plot(X[0:mblue, 0], X[0:mblue, 1], 'bo', markersize=1)
plt.plot(X[mblue:m, 0], X[mblue:m, 1], 'ro', markersize=1)

plt.plot([0.0, b/a[0]], [b/a[1], 0.0],  color='g')
plt.plot([0.0, (b+1)/a[0]], [(b+1)/a[1], 0.0],  color='g', lw = 1)
plt.plot([0.0, (b-1)/a[0]], [(b-1)/a[1], 0.0],  color='g', lw = 1)

plt.axis('equal')
plt.xlim( 0.0, 2.0 )
plt.ylim( 0.0, 2.0 )

plt.show()

# Formulate and solve the problem below
# you should try to use cvxpy or the scipy.optimize.minimize() routine.
```



In [6]:
```
# Define the variables
# a, b variables
# X points
# D Class labels
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
a = cp.Variable(2)
b = cp.Variable()

# Define objective
objective = cp.Minimize(0.5*cp.norm(a,2)**2)

# constraints
constraints = [D @ (X @ a - b*np.ones(m)) >= 1]
# constraints = [cp.multiply(D, X @ a - b) >= 1]

# Define problem and solve
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()

# get the params values
a = a.value
b = b.value

print(f"Primal optimum value: {problem.value}")
print(f"Primal optimum a value: {a}")
print(f"Primal optimum b value: {b}")
```

```
Primal optimum value: inf
Primal optimum a value: None
Primal optimum b value: None
```

We can see that the outlier red point added to the blue zone prevented to find a separable line between the set of points. We have a problem value inf and none solution.

- Find the best classiier that maximizes the width of a slab that separates the two point sets while minimizing the amount of misclassiication.

Problem Formulation

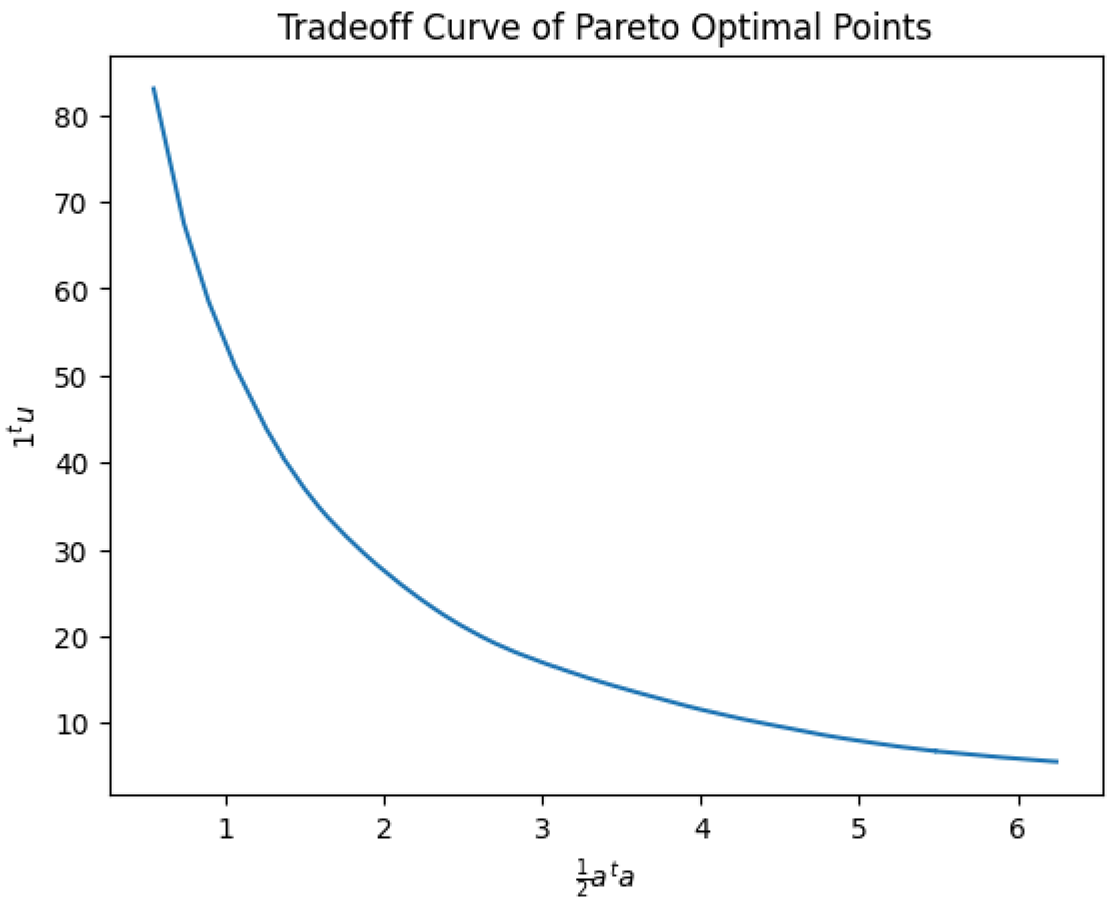
To find the best classifier that maximizes the width of the slab but minimizes the n
Relax the constraints adding u_i where $i = 1 \ldots N$ and v_i where $i = 1 \ldots M, u_i \geq 0\forall$
Then for the constraints:
$a^T x_i - b \geq 1 - u_i \quad i = 1 \ldots N$
$a^T y_i - b \leq -(1 - v_i) \quad i = 1 \ldots M$
For this case define $u = u_i + v_i$ the sum of the set of all u_i and v_i
to formulate the new problem maximizing the gap and minimizing the misclassifica
minimize $\frac{1}{2}\|a\|^2 + \alpha \sum u$
S.t. $D(xa - b) \geq 1 - u$
$u \geq 0$

In [7]:
```
# Define the variables
# a, b variables
# X points
# D Class labels
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
a = cp.Variable(2)
b = cp.Variable()
u = cp.Variable(m)

# Define objective
objective = cp.Minimize(0.5*cp.norm(a,2)**2 + np.ones(m).T @ u)

# constraints
constraints = [D @ (X @ a - b*np.ones(m)) >= np.ones(m) - u, u>=0]
# constraints = [cp.multiply(D, X @ a - b) >= 1]

# Define problem and solve
```

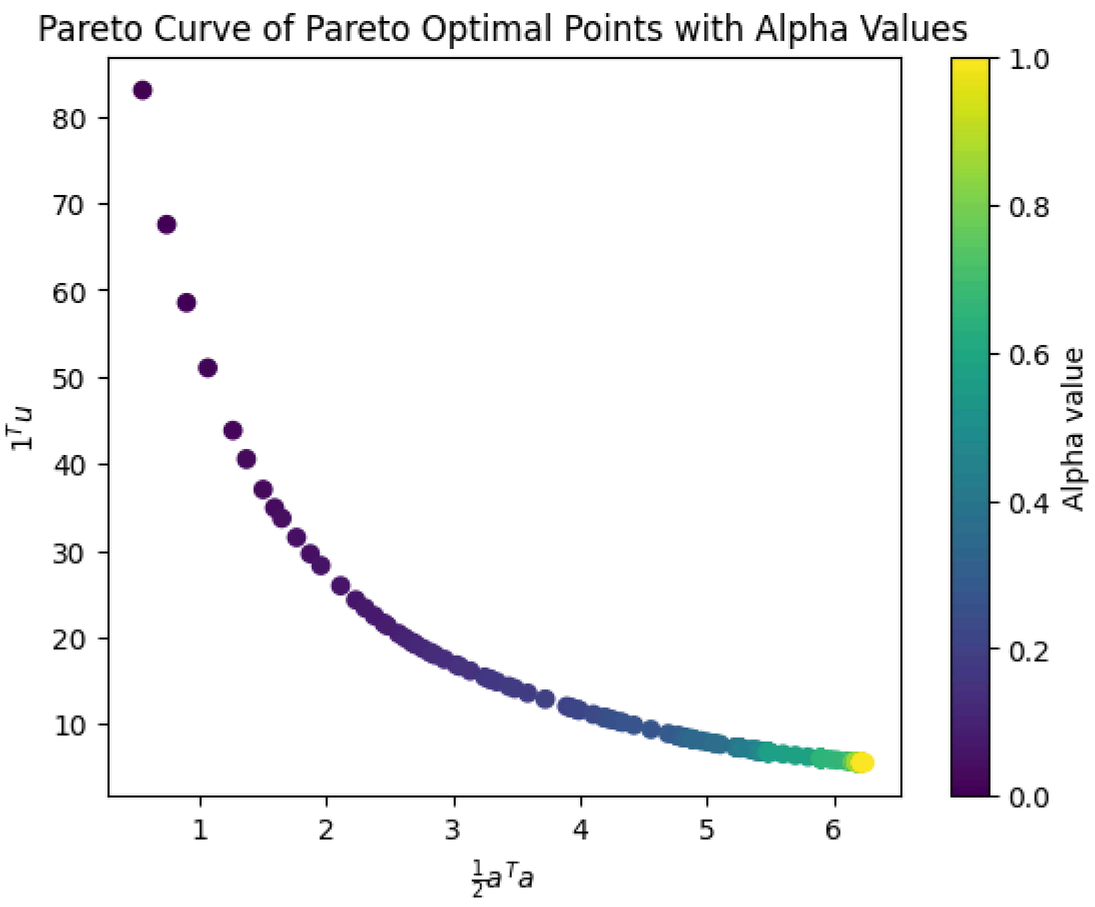
```
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()

# get the params values
a = a.value
b = b.value

print(f"Primal optimum value: {problem.value}")
print(f"Primal optimum a value: {a}")
print(f"Primal optimum b value: {b}")
```

```
Primal optimum value: 11.768739081697
Primal optimum a value: [3.09878287 1.7078903 ]
Primal optimum b value: 4.747770557563012
```

In [8]:
```
# Define the variables
# a, b variables
# X points
# D Class labels
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
a = cp.Variable(2)
b = cp.Variable()
u = cp.Variable(m)
alpha = cp.Parameter(nonneg=True)

# Define objective
objective = cp.Minimize(0.5*cp.norm(a,2)**2 + alpha * np.ones(m).T @ u)

# constraints
constraints = [D @ (X @ a - b*np.ones(m)) >= np.ones(m) - u, u>=0]

# Define problem and solve
problem = cp.Problem(objective, constraints)

# Loop
tradeoff_points = []

for val in np.arange(0.01,1,0.005):
    alpha.value = val
    problem.solve()
    tradeoff_points.append((a.value, b.value, u.value))


# Plotting the tradeoff curve.
# For simplicity, we will plot the norm of 'a' vs. the sum of 'u' to see the
norm_a_values = [0.5*a_val.T @ a_val for a_val, _, _ in tradeoff_points]
sum_u_values = [np.sum(u_val) for _, _, u_val in tradeoff_points]

plt.figure('Tradeoff Curve')
plt.plot(norm_a_values, sum_u_values)
plt.xlabel(r'$\frac{1}{2}a^ta$')
plt.ylabel(r'$1^tu$')
plt.title('Tradeoff Curve of Pareto Optimal Points')
plt.show()
```



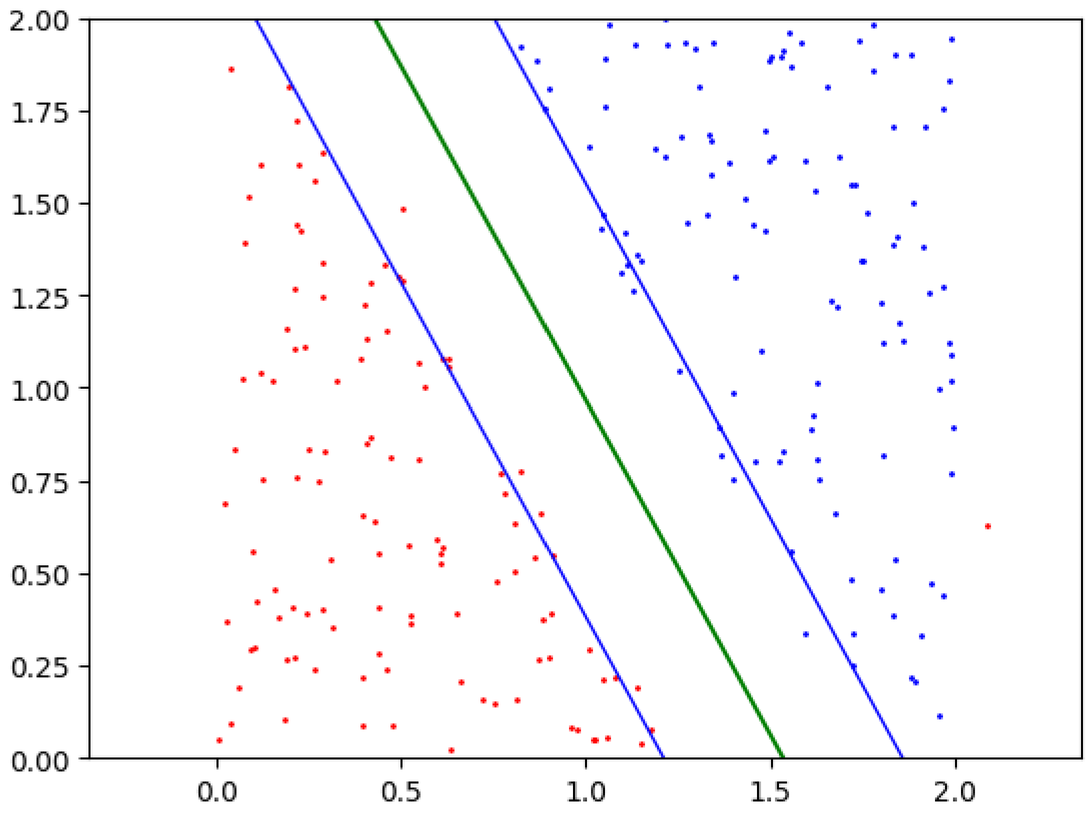
In [11]:
```
# Define the alpha values we tested
alpha_values = np.arange(0.01, 1, 0.005)

# Calculate the norm of 'a' and the sum of 'u' for the trade-off curve
norm_a_values = [0.5 * np.linalg.norm(a_val)**2 for a_val, _, _ in tradeoff_
sum_u_values = [np.sum(u_val) for _, _, u_val in tradeoff_points]

# Create a color map based on the alpha values
colors = cm.viridis((alpha_values - min(alpha_values)) / (max(alpha_values)

# Plot the tradeoff curve with a color map
plt.scatter(norm_a_values, sum_u_values, c=colors)
cbar = plt.colorbar()
cbar.set_label('Alpha value')

# Add labels and title
plt.xlabel(r'$\frac{1}{2}a^T a$')
plt.ylabel(r'$1^T u$')
plt.title('Pareto Curve of Pareto Optimal Points with Alpha Values')

# Show the plot
plt.show()
```



Comments

- The curve is monotonically decreasing and concave downward. We can see that the effect of alpha in the trade-off between reducing missclassification and maximizing the width of the slab that separate the points. This curves makes sense because relaxing the constraints will increase the missclassification and penalizing harder with the alpha value we will have less missclassification but reducing the slab width. An apropiate value for alpha is the point in the curve around 0.15 and 0.2.

In [12]:
```
a = a.value
b = b.value
# plot test data
plt.figure('Test data')
plt.plot(X[0:mblue, 0], X[0:mblue, 1], 'bo', markersize=1)
plt.plot(X[mblue:m, 0], X[mblue:m, 1], 'ro', markersize=1)

plt.plot([0.0, b/a[0]], [b/a[1], 0.0],  color='g')
plt.plot([0.0, (b+1)/a[0]], [(b+1)/a[1], 0.0],  color='b', lw = 1)
plt.plot([0.0, (b-1)/a[0]], [(b-1)/a[1], 0.0],  color='b', lw = 1)

plt.axis('equal')
plt.xlim( 0.0, 2.0 )
plt.ylim( 0.0, 2.0 )

plt.show()
```



Overall, if we have an outlier we can still design a classifier to separate the data as we demonstrated in the last problem.

3. Dual

3. Dual. Consider the dual of the classification problem above.

- Write the Lagrangian:

$$L(a, b, u, \lambda, \sigma) = \tfrac{1}{2}a^T a - \lambda^T DXa + \lambda^T D\mathbf{1}b + (\alpha\mathbf{1}^T - \lambda^T - \sigma^T)u + \lambda^T\mathbf{1}$$

  where λ and σ are the Lagrange multipliers of the classification and non-negativity constraints, respectively.

- Express the dual problem as:

$$\begin{array}{ll} \underset{\lambda}{\text{maximize}} & -\tfrac{1}{2}\lambda^T DXX^T D\lambda + \mathbf{1}^T\lambda \\ \text{subject to} & \mathbf{1}^T D\lambda = 0 \\ & 0 \le \lambda \le \alpha\mathbf{1} \end{array}$$

  (Hint: The dual variables σ can be eliminated)

- Choose a value of α (say, $\alpha = 0.5$) and solve the primal and dual problems. Do you get the same optimal value of the objective value? Comment.

Solution

$$\begin{aligned}\mathcal{L}(a,b,u,\lambda) &= \frac{1}{2}a^\top a + \alpha 1^\top u - \lambda(D(xa-b1) - 1 + u) - \delta^\top u\\ &= \frac{1}{2}a^\top a + \alpha 1^\top u - \lambda^\top D(xa - b1) + \lambda^\top 1 - \lambda^\top u - \delta^\top u\\ &= \frac{1}{2}a^\top a - \lambda^\top Dxa + \lambda^\top D1b + (\alpha 1^\top - \lambda^\top - \delta^\top)u + \lambda^\top 1\end{aligned}$$

Find the dual problem

$$\nabla_{a,b,u}\mathcal{L}(a,b,u,\lambda,\delta) = \begin{bmatrix} a - \lambda^\top Dx \\ \lambda^\top D1 \\ \alpha 1^\top - \lambda^\top - \delta^\top \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \implies \begin{matrix} \lambda^\top D1 = 0 \\ \lambda^\top Dx = a \\ \alpha^\top 1 - \lambda^\top - \delta^\top = 0 \end{matrix}$$

$$\begin{aligned}&\alpha 1^\top - \lambda^\top = \delta^\top \quad \text{but } \delta \geq 0\\ &\alpha 1^\top - \lambda^\top \geq 0\end{aligned}$$

Dual Problem replace in $\mathcal{L}(a,b,u,\lambda,\delta)$

$$g(\lambda,\delta) = \begin{cases} -\frac{1}{2}(\lambda^\top Dx)^\top(\lambda^\top Dx) + \lambda^\top 1 & \text{if } \lambda^\top D1 = 0 \text{ and } 0 \leq \lambda \leq \alpha 1^\top \\ -\infty & \text{otherwise}\end{cases}$$

$$\text{maximize}_{\ \lambda} \quad -\frac{1}{2}X^\top D^\top \lambda\lambda^\top DX + \lambda^\top 1$$

$$\begin{aligned}&\text{Subject to } \lambda^\top D1 = 0\\ &\quad 0 \leq \lambda \leq \alpha 1^\top\end{aligned}$$

- Solve primal and dual problems for $\alpha = 0.5$.

Primal Problem

In [13]:
```
# Define the variables
# a, b variables
# X points
# D Class labels
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
a = cp.Variable(2)
b = cp.Variable()
u = cp.Variable(m)
alpha = cp.Parameter(nonneg=True)

# Define objective
objective = cp.Minimize(0.5*cp.norm(a,2)**2 + alpha * np.ones(m).T @ u)

# constraints
constraints = [D @ (X @ a - b*np.ones(m)) >= np.ones(m) - u, u>=0]
```

```
# Define problem and solve
problem = cp.Problem(objective, constraints)
alpha.value = 0.5
problem.solve()
a = a.value
b = b.value
print(f"Primal optimum value: {problem.value}")
print(f"Primal optimum a value: {a}")
print(f"Primal optimum b value: {b}")
```

```
Primal optimum value: 8.84700818185702
Primal optimum a value: [2.89435198 1.60016787]
Primal optimum b value: 4.464484759367421
```

Dual Problem

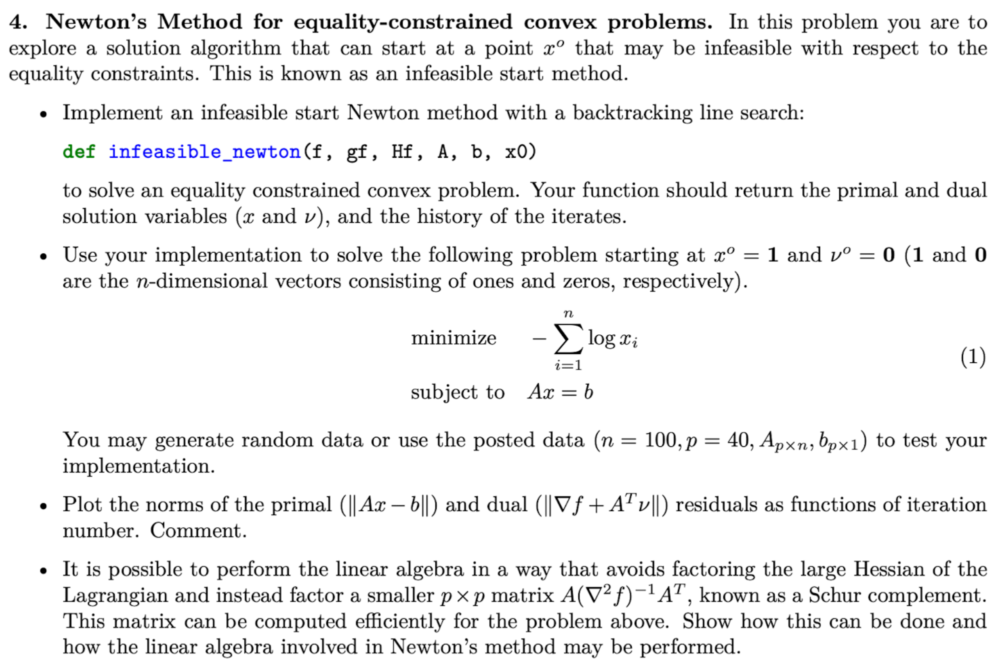
In [15]:
```
# Define the variables
# l lambda
# X points
# D Class labels
y = X_labels.reshape(-1, 1)
X_labels = np.ones(m)
X_labels[mblue:] = -1
D = np.diag(X_labels)

# Optimization problem
l = cp.Variable(m)
alpha_value = 0.5
Q = cp.Parameter((m, m), PSD=True)
Q.value = np.matmul(y * X, (y * X).T)

# Define objective
# objective = cp.Minimize(-(0.5*(X.T @ D.T @ l @ l.T @ D @ X) + l.T @ np.one
objective = cp.Maximize(cp.sum(l) - 0.5 * cp.quad_form(l, Q))

# # constraints
# constraints = [l.T @ D @ np.ones(m) == 0, l>=0, l<=alpha]
constraints = [y.T @ l == 0, l >= 0, l <= alpha_value]

# # Define problem and solve
problem = cp.Problem(objective, constraints)
problem.solve()

print(f"Dual optimum value: {problem.value}")
```

```
Dual optimum value: 8.847008181685172
```

Comments

After finding the dual problem from the primal implementation. We solved for both and we found that they hold a strong duality. Both problems result with the same objective value at optimality of 8.847008.

4. Newton's Method for equaity-constrained convex problems



4. Newton's Method for equality-constrained convex problems. In this problem you are to explore a solution algorithm that can start at a point x^o that may be infeasible with respect to the equality constraints. This is known as an infeasible start method.

- Implement an infeasible start Newton method with a backtracking line search:

  `def infeasible_newton(f, gf, Hf, A, b, x0)`

  to solve an equality constrained convex problem. Your function should return the primal and dual solution variables (x and ν), and the history of the iterates.

- Use your implementation to solve the following problem starting at $x^o = \mathbf{1}$ and $\nu^o = \mathbf{0}$ ($\mathbf{1}$ and $\mathbf{0}$ are the n-dimensional vectors consisting of ones and zeros, respectively).

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^{n} \log x_i \\ \text{subject to} & Ax = b \end{array} \qquad (1)$$

  You may generate random data or use the posted data ($n = 100, p = 40, A_{p\times n}, b_{p\times 1}$) to test your implementation.

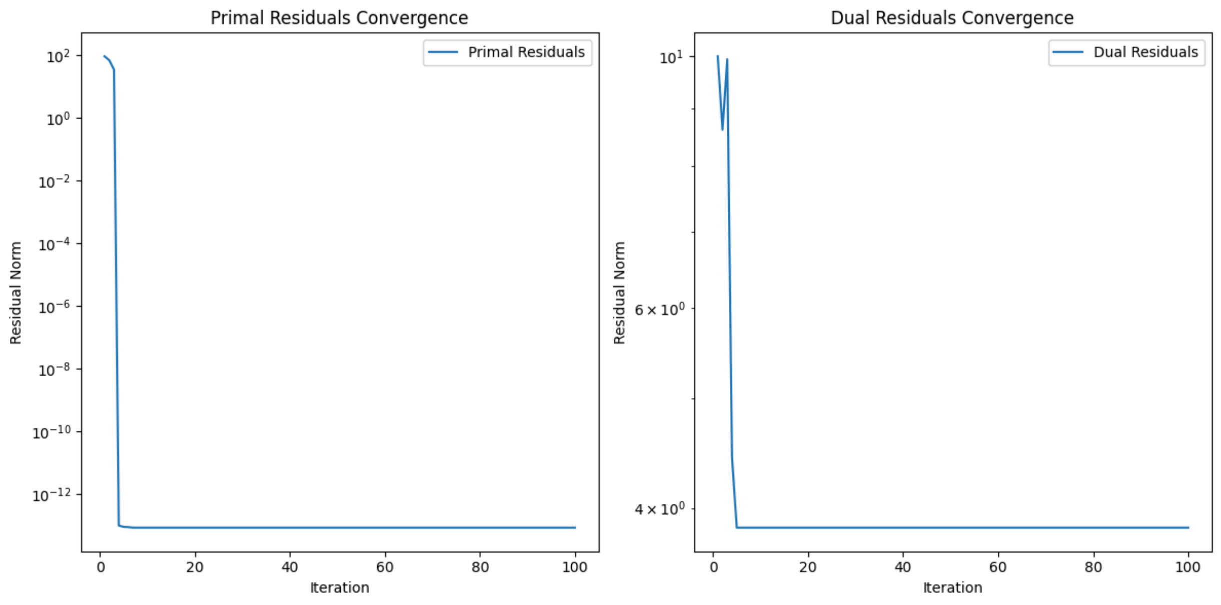
- Plot the norms of the primal ($\|Ax - b\|$) and dual ($\|\nabla f + A^T\nu\|$) residuals as functions of iteration number. Comment.

- It is possible to perform the linear algebra in a way that avoids factoring the large Hessian of the Lagrangian and instead factor a smaller $p \times p$ matrix $A(\nabla^2 f)^{-1}A^T$, known as a Schur complement. This matrix can be computed efficiently for the problem above. Show how this can be done and how the linear algebra involved in Newton's method may be performed.

In [ ]:

```
# Seed the random number generator for repeatability
seed = 211
rng = np.random.default_rng(seed)

n = 100
p = 40

# Generate a random p-by-n matrix A with independent rows
A = rng.standard_normal((p, n))
while np.linalg.matrix_rank(A) != p:
    A = rng.standard_normal((p, n))

# Generate a random right hand side b
b = A @ rng.standard_normal(n)

# Objective function f(x)
def f(x):
    return -np.sum(np.log(x))

# Gradient of the objective function gf(x)
def gf(x):
    return -1 / x

# Hessian of the objective function Hf(x)
def Hf(x):
    return np.diag(1 / x**2)
```

```
# Backtracking line search algorithm
def backtracking_line_search(x, delta_x, nu, delta_nu, f, gf, A, b, alpha=0.
    t = 1
    while np.min(x + t * delta_x) <= 0 or np.linalg.norm(A @ (x + t * delta_
        t *= beta
    return t

# Infeasible start Newton method
def infeasible_newton(f, gf, Hf, A, b, x0, tol=1e-10, max_iter=100):
    m, n = A.shape
    x = x0
    nu = np.zeros(m)
    residuals = []

    for i in range(max_iter):
        # Calculate the residuals and check the tolerance
        r_dual = gf(x) + A.T @ nu
        r_pri = A @ x - b
        residuals.append((np.linalg.norm(r_pri), np.linalg.norm(r_dual)))
        if np.linalg.norm(r_dual) < tol and np.linalg.norm(r_pri) < tol:
            break

        # Calculate the Newton direction
        H = Hf(x)
        M = np.block([[H, A.T], [A, np.zeros((m, m))]])
        delta = np.linalg.solve(M, -np.hstack([r_dual, r_pri]))
        delta_x = delta[:n]
        delta_nu = delta[n:]

        # Perform backtracking line search to find the step size
        t = backtracking_line_search(x, delta_x, nu, delta_nu, f, gf, A, b)

        # Update x and nu
        x += t * delta_x
        nu += t * delta_nu

    return x, nu, residuals

# Initial guess x0 and nu0
x0 = np.ones(n)

# Solve the problem using the infeasible start Newton method
x_star, nu_star, residuals = infeasible_newton(f, gf, Hf, A, b, x0)

# Plotting the residuals
primal_residuals, dual_residuals = zip(*residuals)
iterations = list(range(1, len(primal_residuals) + 1))

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.semilogy(iterations, primal_residuals, label='Primal Residuals')
plt.xlabel('Iteration')
plt.ylabel('Residual Norm')
plt.title('Primal Residuals Convergence')
plt.legend()
plt.subplot(1, 2, 2)
plt.semilogy(iterations, dual_residuals, label='Dual Residuals')
plt.xlabel('Iteration')
plt.ylabel('Residual Norm')
plt.title('Dual Residuals Convergence')
plt.legend()

plt.tight_layout()
plt.show()
```



- The left plot shows that the primal residuals decrease rapidly within the first few iterations. This shows how well the iterates are satisfying the primal constraints, is improving significantly with each iteration. After around 10 iterations, the primal residuals have decreased to below 10^-10, which suggests that the primal variables are converging to a feasible solution for the optimization problem.
- The right plot shows the rate at which the dual variables or the Lagrange multipliers are converging. There is an initial spike, which might be due to initialization or scaling issues, but it quickly resolves. By about 10 iterations, the dual residuals also have decreased significantly, reaching a value close to 10^-5.