EE 263 Homework 7

Luis A. Perez

Bow Tie Launch

Solution:

(a) Given the restrictions imposed by the problem, we begin by tackling it in pieces. First, note that we can write $x(t)$ in the following form:

$$\begin{aligned} x(t) &= \begin{bmatrix} A^{t-1}B & A^{t-2}B & \cdots & AB & B \end{bmatrix} u_{\text{seq}} \\ &\quad + (A^{t-1} + A^{t-2} + \cdots + A + I)B_g \\ \implies x(t) - (A^{t-1} + A^{t-2} + \cdots + A + I)B_g &= \begin{bmatrix} A^{t-1}B & A^{t-2}B & \cdots & AB & B \end{bmatrix} u_{\text{seq}} \\ \implies \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} [x(t) - (A^{t-1} + A^{t-2} + \cdots + A + I)B_g] &= \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} \begin{bmatrix} A^{t-1}B & A^{t-2}B & \cdots & AB & B \end{bmatrix} u_{\text{seq}} \\ \implies p(t) - \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{t-1} + A^{t-2} + \cdots + A + I)B_g &= \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} \begin{bmatrix} A^{t-1}B & A^{t-2}B & \cdots & AB & B \end{bmatrix} u_{\text{seq}} \end{aligned}$$

which is very nearly in the form we're trying to find, given by:

$$\hat{A}u_{\text{seq}} = \hat{y}$$

In fact, we can let:

$$\hat{y}_t = p(t) - \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{t-1} + A^{t-2} + \cdots + A + I)B_g \in \mathbb{R}^2$$

and then we can define:

$$\hat{y} = \begin{bmatrix} y_{10} \\ y_{20} \\ \vdots \\ y_{60} \\ y_{61} \\ \vdots \\ y_{65} \end{bmatrix} \in \mathbb{R}^{22}$$

With the above defined, we can similarly create our matrix $\hat{A}$. First, let us defined:

$$\hat{A}_t = \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} \begin{bmatrix} A^{t-1}B & A^{t-2}B & \cdots & AB & B & 0 & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{2\times 130}$$

(Where the front is padded by 0s)

Then we define our matrix $\hat{A}$ simply as:

$$\hat{A} = \begin{bmatrix} \hat{A}_{10} \\ \hat{A}_{20} \\ \vdots \\ \hat{A}_{60} \\ \hat{A}_{61} \\ \vdots \\ \hat{A}_{65} \end{bmatrix} \in \mathbb{R}^{22 \times 130}$$

Then note that we immediately have the system as desired:

$$\hat{y} = \hat{A} u_{\text{seq}}$$

Solving for the minimum norm solutin, we have:

$$u_{ln} = \hat{A}^T (\hat{A}\hat{A}^T)^{-1} \hat{y}$$

A plot of the applied forces can be seen in Figure 1, and a plot of the calculated drone trajectory and waypoints in Figure 2.

(b) This is just a weighed sum objective problem. The weighed sum objective can be rewritten as:

$$J_1 + \mu J_2 = \left|\left| \begin{bmatrix} \hat{A} \\ \sqrt{\mu} I \end{bmatrix} u_{\text{seq}} - \begin{bmatrix} \hat{y} \\ 0 \end{bmatrix} \right|\right|_2^2$$

This should be immediately obvious, but for the wary, we show the argument below. Basically, expanding out we have:

$$\begin{aligned} J_1 + \mu J_2 &= \sum_{i=1}^{11} ||q(t_i) - p_i||^2 + \mu \sum_{t=0}^{64} ||u(t)||^2 \\ &= \sum_{i=1}^{11} ||q(t_i) - X_{t_i} + X_{t_i} - p_i||^2 + \mu \sum_{t=0}^{64} ||u(t)||^2 \\ &\qquad (X_{t_i} = \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{t_i - 1} + A^{t_i - 2} + \cdots + A + I) B_g) \\ &= \sum_{i=1}^{11} ||\hat{A}_{t_i} u(t_i) - \hat{y}_i||^2 + \mu \sum_{t=0}^{64} ||u(t)||^2 \end{aligned}$$

From the above, we can see how we can rewrite as described. As such, we have that the solutions are given directly by:

$$u^* = \begin{bmatrix} \hat{A} \\ \sqrt{\mu} I \end{bmatrix}^{\dagger} \begin{bmatrix} \hat{y} \\ 0 \end{bmatrix}$$

where we note that] $\in \mathbb{R}^{87\times 130}$, so it is fat and full-rank. As such we know exactly how to compute the solution. As such, we compute the above and vary the values of μ from $1to10^7$. The resulting trade-off curve is shown in Figure TODO (incomplete).

(c) We plot the five trajectories. See the attached figures.

(d) We find the matrices F and G so that we can rewrite the cost functions. We can immediately see that the matrix

$$\begin{aligned}
J_3 &= \sum_{t=0}^{65} ||q(t)||^2 \\
&= \sum_{t=0}^{65} ||A_t u_{\text{seq}} + \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{t-1} + A^{t-2} + \cdots + A + I)B_g||^2 \qquad \text{(Results from (a))} \\
&= ||W u_{\text{seq}} + G||^2
\end{aligned}$$

where we have G given by:

$$Z = \begin{bmatrix} 0 \\ \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} B_g \\ \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A+I)B_g \\ \vdots \\ \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{t-1} + A^{t-2} + \cdots + A + I)B_g \vdots \\ \begin{bmatrix} e_1 \\ e_3 \end{bmatrix} (A^{64} + A^{63} + \cdots + A + I)B_g \end{bmatrix} \in \mathbb{R}^{130\times 1}$$

and

$$W = \begin{bmatrix} 0 \\ A_1 \\ A_2 \\ \vdots \\ A_{65} \end{bmatrix} \in \mathbb{R}^{130\times 130}$$

With the above, we have the full objetive as:

$$J_3 + \gamma J_2 = ||W u_{\text{seq}} + Z||^2 + \gamma ||u_{\text{seq}}||^2$$

We can do the same trick we did for the multi-objective function, and combine the above into one norm as follows:

$$J_3 + \gamma J_2 = ||F u_{\text{seq}} + G||^2$$

where we write:

$$F = \begin{bmatrix} W \\ \sqrt{\gamma} I \end{bmatrix}$$

and

$$G = \begin{bmatrix} Z \\ 0 \end{bmatrix}$$

so that it matches the output dimensions of F.

(e) We show that the optimal solution to a problem of the form

$$\begin{aligned} &\text{minimize} && ||Fu + G||^2 \\ &\text{subject to} && \hat{A}u = \hat{y} \end{aligned}$$

is given by:

$$u_{\text{opt}} = \Sigma^{-1}\hat{A}^T(\hat{A}\Sigma^{-1}\hat{A}^T)^{-1}(\hat{y} + \hat{A}\Sigma^{-1}F^TG) - \Sigma^{-1}F^TG$$

We can do this using Lagrange multipliers. The Lagrangian for our problem is given by:

$$\begin{aligned} L(u, \lambda) &= (Fu + G)^T(Fu + G) + \lambda^T(\hat{A}u - \hat{y}) \\ &= u^T\Sigma u + G^TG + u^TFG + GFu + \lambda^T(\hat{A}u - \hat{y}) \end{aligned}$$

This gives the optimality conditions as:

$$\begin{aligned} \nabla_u L = 2u^T\Sigma + \mathbf{1}^TFG + \mathbf{1}^TGF + \lambda^TA = 0 \\ \nabla_\lambda L = \hat{A}u - \hat{y} = 0 \end{aligned}$$

The first condition immediate gives:

$$u = -\frac{1}{2}\Sigma^{-1}(\hat{A}^T\lambda + 2F^TG)$$

We can then subtitute this solution into the second equation to give us:

$$\begin{aligned} -\frac{1}{2}\hat{A}(\Sigma^{-1}(\hat{A}^T\lambda + 2F^TG)) = \hat{y} \\ \implies \lambda = -2(\hat{A}\Sigma^{-1}\hat{A}^T)^{-1}(\hat{y} + \hat{A}\Sigma^{-1}F^TG) \end{aligned}$$

and finally, subtituting λ back into the first equations we have:

$$u_{\text{opt}} = \Sigma^{-1}\hat{A}^T(\hat{A}\Sigma^{-1}\hat{A}^T)^{-1}(\hat{y} + \hat{A}\Sigma^{-1}F^TG) - \Sigma^{-1}F^TG$$

just as we wanted.

(f) The plots for the values are shown in the attached code. We can see in the plots the clear tradeoff for γ. A low value of γ causes the drone to try extremely hard to stay close to the origin. The case with. the $\gamma = 10$ seems to match the requirements the best.

(g) See attached code for attempt at plotting the trade-off curves.

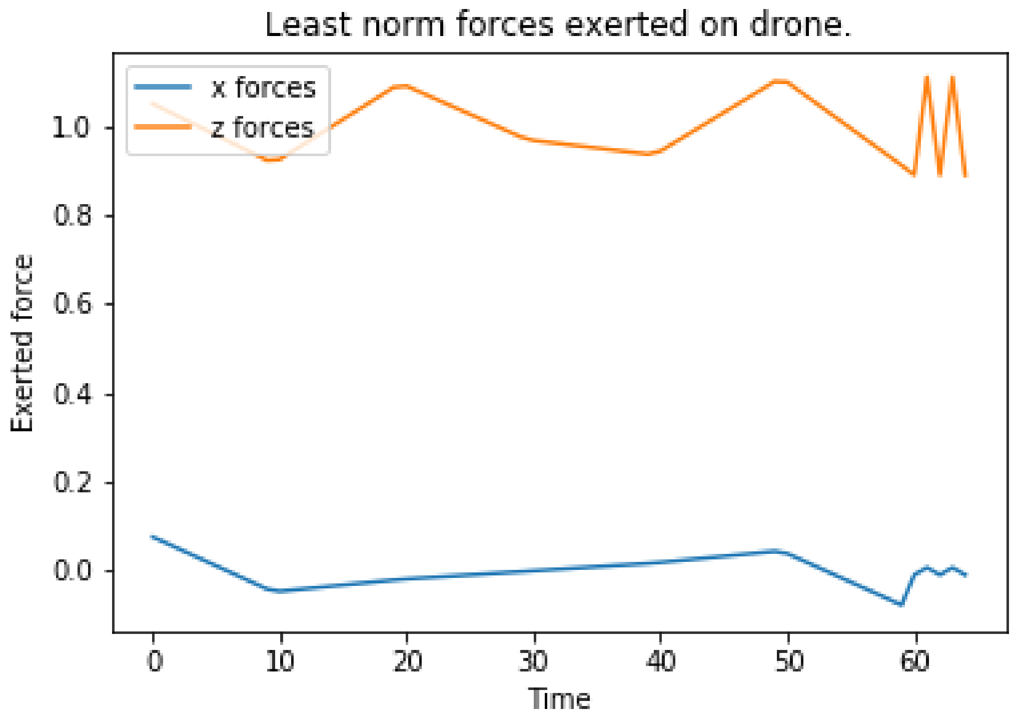


Figure 1: Plot of least norm forces over time.

Attack position of Fighter Jets

Solution:

(a) Following the hint, we try to express each of the control schemas as a linear dynamical system. That is to say, we try to frame them in the form:

$$\dot{x} = Ax$$

where x is our state vector.

- *Right looking control*:
  For this control schema, we begin by defining our state as follows:

$$x = \begin{bmatrix} y_n \\ s_{n-1} \\ s_{n-2} \\ \vdots \\ s_1 \\ v_n \\ v_{n-1} \\ \vdots \\ v_1 \\ 1 \end{bmatrix} \in \mathbb{R}^{2n+1}$$

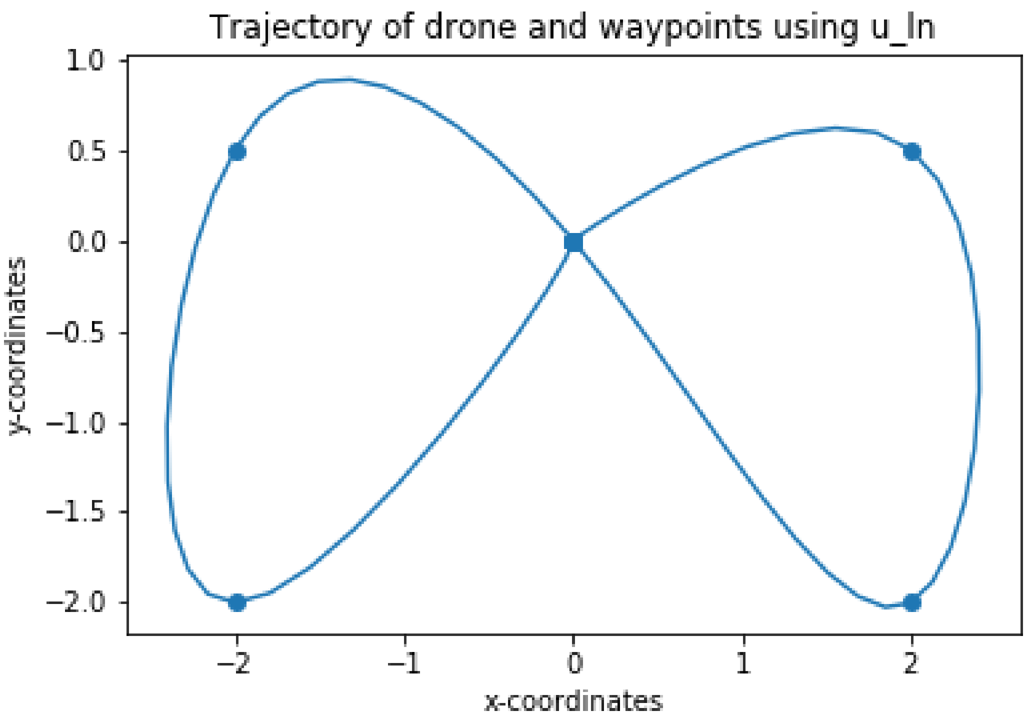


Figure 2: Waypoint and drone trajectory

This leads us to the matrix A (in our case $n = 7$) defined as:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{15\times 15}$$

This matrix gives us the exact dynamics we want. Specifically, we have:

$$\begin{aligned} \dot{y}_n &= v_n \\ \dot{s}_i &= \dot{y}_{i+1} - \dot{y}_i = v_{i+1} - v_i \\ \dot{v}_n &= u_n - v_n = 7 - y_n - v_n \\ \dot{v}_i &= u_i - v_i = s_i - 1 - v_i \end{aligned}$$

Note that in this paradigm, we're looking for the state vector given by:

$$x_{\text{aligned}} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \in \mathbb{R}^{15}$$

which indicates that $y_n(t) = 7$ (so the right-most jet is in the correct position) and that $s_i(t) = 1$ (so the spacing is 1) and that $v_i(t) = 0$ (so velocity is 0).

For this scheme to 'work', we must have that no matter the initial conditions, $e^{tA}x(0)$ converges to x_{aligned}. However, the only eigenvalues of the matrix A are given by:

$$\begin{aligned}\lambda_1 &= -0.5 + 0.8660254i \\ \lambda_2 &= -0.5 - 0.8660254i \\ \lambda_3 &= 0\end{aligned}$$

where λ_1 and λ_2 have multiplicity of 7 each. We can immediately see that since the real-part part of the non-zero eigenvalues are all negative, these directions will eventually be killed of, leaving only the eigenvector v_3 with corresponding eigenvalue $\lambda_3 = 0$. This means that we have:

$$\begin{aligned}\lim_{t\to\infty} e^{tA}x(0) &= \lim_{t\to\infty} e^{\lambda_3 t}(w_3^T x(0))v_3 \\ &= w_3^T x(0) v_3\end{aligned}$$

Computing the given eigenvectors (see code for reference), we have the below (each vector has been normalized):

$$v_3 = \begin{bmatrix} 0.935 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.134 \end{bmatrix} \in \mathbb{R}^{15}$$

$$w_3 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \in \mathbb{R}^{15}$$

Now, given the representation of our system, we must always have our state with the last entry equal to 1. This means that for any $x(0)$, we know that $x(0)_{15} = 1$. As such, we can remove the last degree of freedom from v_3 by making sure it's last entry is 1. Doing this, we see that we have:

$$\begin{aligned}\lim_{t\to\infty} e^{tA}x(0) &= \hat{v}_3 \\ &= \begin{bmatrix} 7 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} = x_{\text{aligned}}\end{aligned}$$

As such, we conclude with the statement that **this scheme works**. Note that we only don't achieve our desired state if we have a zero starting state, but this is not possible since we know that the starting positions of the planes are all distinct.

- Following a similar process as for the previous scheme, we can define our LDS with the same state $x \in \mathbb{R}^{15}$ (with similar restrictions) but with a slightly modified matrix A given below:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0.5 & -0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & -0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & -0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & -0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & -0.5 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in$$

Running through the same process but with this matrix, we once again find that the eigenvalues correspond to:

$$\begin{aligned}
\lambda_1 &= -0.5 + 1.31j \\
\lambda_2 &= -0.5 - 1.31j \\
\lambda_3 &= -0.5 + 1.207j \\
\lambda_4 &= -0.5 - 1.207j \\
\lambda_5 &= -0.035 \\
\lambda_6 &= -0.965 \\
\lambda_7 &= -0.5 + 1.004j \\
\lambda_8 &= -0.5 - 1.004j \\
\lambda_9 &= -0.5 + 0.701j \\
\lambda_{10} &= -0.5 - 0.701j \\
\lambda_{11} &= -0.5 + 0.866j \\
\lambda_{11} &= -0.5 - 0.866j \\
\lambda_{12} &= 0
\end{aligned}$$

Some of the above values have non-singular multiplicity. However, the important thing to note is that all but-one have real parts which are < 0, which indicates that as $t \to \infty$, their corresponding eigenvectors will be killed off. And again, similarly to the previous part, we actually have the v_{12} given by:

$$v_{12} = \begin{bmatrix} 0.935 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0.134 \\ 0 \\ \vdots \\ 0 \\ 0.134 \end{bmatrix} \in \mathbb{R}^{15}$$

Again, this normalizes so that the final state is equal to x_{aligned}. This tells us that **this control schema works**, since for any non-zero input vector (which we must have since the initial positions are all distinct), we will achieve the final position given enough time.

- *Independent alignment* We follow a slightly different approach to representing this system. In this case, we have the input state given as:

$$x = \begin{bmatrix} y_n \\ y_{n-1} \\ y_{n-2} \\ \vdots \\ y_1 \\ v_n \\ v_{n-1} \\ \vdots \\ v_1 \\ 1 \end{bmatrix} \in \mathbb{R}^{2n+1}$$

With the above, the dynamics are given by the block matrix:

$$A = \begin{bmatrix} 0_7 & I_7 & 0 \\ -I_7 & -I_7 & b \\ 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{15\times 15}$$

where $b \in \mathbb{R}^7 = \begin{bmatrix} 7 \\ 6 \\ 5 \\ \vdots \\ 1 \end{bmatrix}$.

Given this system, we can find the corresponding eigenvalues. The are given by:

$$\lambda_1 = -0.5 + 0.866$$
$$\lambda_2 = -0.5 - 0.866$$
$$\lambda_3 = 0$$

Again, the fact that all the eigenvalues have real parts less than zero indicates that we will converge to v_3 in the long-run. Normalizing given that we must have the last element in our state be 1, we have:

$$\lim_{t\to\infty} e^{tA}x(0) = \begin{bmatrix} 7 \\ 6 \\ 5 \\ 4 \\ 3 \\ 2 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \in \mathbb{R}^{15}$$

As such, this **schema also works**.

For scheme 1 and 2, the asymptotic convergence properties are essentially the same (their eigenvalues are exactly the same, with the same multiplicity). As such, these two schemes have the same overall convergence properties.

However, scheme 2 has the potential to converge faster, since some of its eigenvectors actually have much smaller (more negative) real-parts. This means that these sections will collpse to 0 more quickly.

(b) Given that all of the schemes converge, we approach this problem by simply running a simulation until a collision occurs (which might not be the case) or we've reached a stable point. Given the precision requested of 0.1, we increment in intervals of 0.05. We now present the results:

- *Right looking control*
  Collision occured at $t = 8.35$ between jet 2 and jet 1.
- *Left and right looking control*
  Closest collision occured at $t = 2.35$ between jet 7 and jet 6 with spacing of 0.31462333372486384.
- *Independent control*
  Closest collisions occured at $t = 3.65$ between jet 5 and jet 4 with spacing of 0.8370070662304583 as well as between jets 3 and 4 with the same spacing.

UN General Assembly voting

Solution:

(a) The requested stem plot can be seen in Figure 3. We can see that we have two significant values (much larger than the others, by about a factor of 5 or 6). Computing the fraction of the variation in the voting data associated with these first two singular values, we have:

$$f_2 = \frac{\sigma_1^2 + \sigma_2^2}{\sum_{i=1}^r \sigma_i^2} = 0.2044$$

(b) We follow the process described until convergence. The total number of $+1$'s is 44934, with the number of -1's as 19632.

(c)
- The requested scatter plot can be seen in Figure 4.
- The requested scatter plot cab bse seen in Figure 5.

From the two plots, we can give a guess for the meaning of u_1 and u_2.

From Figure 4, we can see that u_2 essentially divices countries into Capitalist and Communist. If the value of u_2 is positive, the country is Capitalists (blue), and if it is negative, the country is Communist (red). However, this measure is not very good for the socialist countries.

From Figure 5, we can see that u_1 appears to be a measure of the tendency for a country to vote with the majority vote (eg, how aligned is the country with the majority). Smaller of u_1 imply higher alignment (a higher fraction of votes matching the majority), while larger values imply lower alignment (a lower fraction of votes matching the majority).

(d)
- The requested scatter plot can be seen in Figure 6.
- The requested scatter plot cab bse seen in Figure 7.

From the two plots, we can give a guess for the meaning of v_1 and v_2.

From Figure 6, we can see that v_1 correlated heavily with the amount of support a vote receives. Lower values of v_1 seem to indicate higher support (pink), while higher values indicate lower support.

From Figure 7, we can see that v_2 is correlated heavily with the partisan support received by a vote. More negative values of v_2 seem to indicate large values of z, so these are votes which are primarily supported by capitalist countries. More positive values of v_2 seem to indicate much lower values of z, so these are votes which are primarily supported by Communist countries.

`un_voting_patterns`

```
% ======== Part (a) ========
S = svd(votes);
stem(diag(S));
title('Stem plot of singular values of votes')
ylabel('Singular value')
saveas(gcf, 'stem_plot.jpg');

% compute variance.
f2 = (S(1) + S(2)) / sum(S);

% ======== Part (b) ========
prevA = votes;
A = zeros(size(votes));
while  norm(prevA - A) > 0.0001
    prevA = A;
    [U, S, V] = svd(A);
    lowRankApprox = S(1,1) .* U(:,1) * V(:,1)' + S(2,2) .*  U(:,2) * V(:,2)';
    A = lowRankApprox;
    A(votes == 1) = 1;
    A(A > 0)= 1;
    A(votes == -1) = -1;
    A(A < 0) = -1;
end;
% Count the number of 1s and -1s.
numOnes = sum(A(:) == 1);
numNegOnes = sum(A(:) == -1);

% ======== Part (c) ========
% Get singular values
[U, S, V] = svd(A);
% 0 represents 'Cap'
class = zeros(size(countries(:,2)));
u1 = U(:, 1);
u2 = U(:, 2);
for i = 1:size(class)
    if strcmp(countries(i, 2), 'Soc')
        class(i) = 1;
    elseif strcmp(countries(i,2),'Com')
        class(i) = 2;
    elseif strcmp(countries(i,2), 'Cap')
        class(i) = 0;
    else
        disp('Error!')
    end
```

```
end
spatial_plot(u1, u2, class, 3, eye(3));
title('Scatter plot of primary diads colored by country classification')
xlabel('u_1')
ylabel('u_2')
saveas(gcf, 'country_scatter_classification.jpg');

% sum columns and take sign to know which was majority vote.
majorityVote = sign(sum(A, 1));
fractionMaj = zeros(size(countries(:,2)));
for i = 1:size(fractionMaj)
    fractionMaj(i) = sum(A(i, :) == majorityVote) / 633;
end

spatial_plot(u1 , u2 , fractionMaj , 10);
title('Scatter plot colored by fraction of votes aligned with majority.')
xlabel('u_1')
ylabel('u_2')
saveas(gcf, 'country_majority_scatter_plot.jpg')

% ======== Part (d) ========
v1 = V(:, 1);
v2 = V(:, 2);
totalSupport = sum(A, 1);
spatial_plot(v1, v2, totalSupport, size(unique(totalSupport), 2));
title('Scatter plot colored by support received by vote j.')
xlabel('v_1')
ylabel('v_2')
saveas(gcf, 'votes_by_total_support.jpg')

capVotes = sum(A(class == 0, :), 1) ./ sum(class == 0);
comVotes = sum(A(class == 2, :), 1) ./ sum(class == 2);
partisanSupport = capVotes - comVotes;
spatial_plot(v1, v2, partisanSupport, size(unique(partisanSupport), 2));
title('Scatter plot colored by partisan support received by the vote.')
xlabel('v_1')
ylabel('v_2')
saveas(gcf, 'votes_by_partisan_support.jpg')
```

Final

August 16, 2019

1 Final

author: Luis Perez
email: luis0@stanford.edu

[375]:
```
import numpy as np
from scipy import linalg
import seaborn as sns
import math
from matplotlib import pyplot as plt
```

[20]:
```
np.set_printoptions(suppress=True, precision=3)
```

1.1 Problem 1

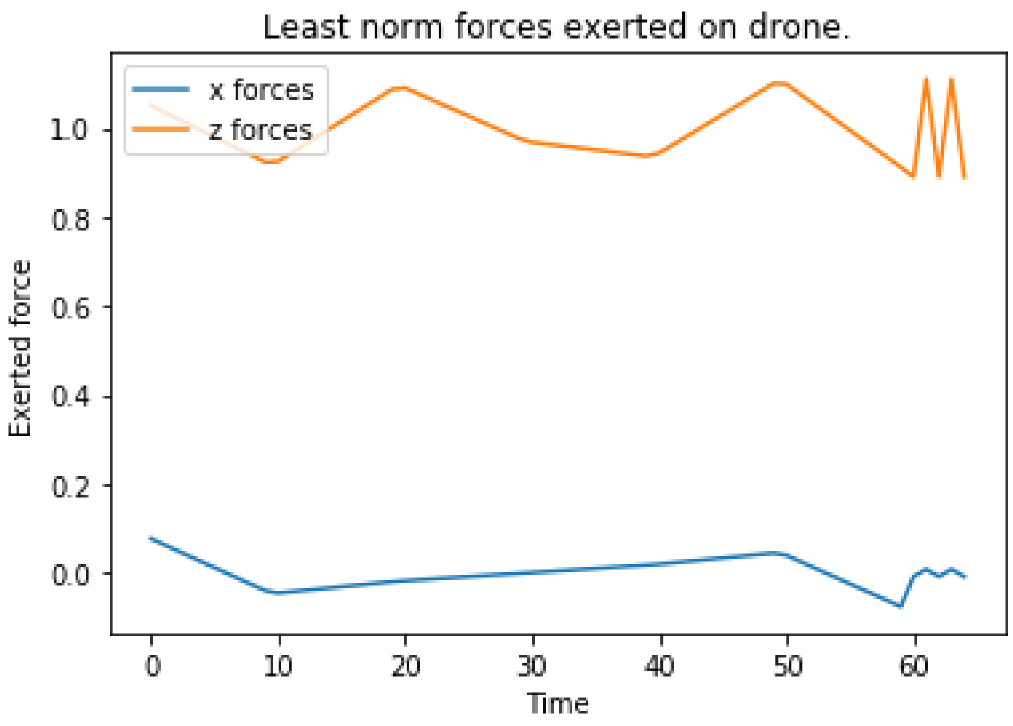
[437]:
```
def getProblem1Inputs():
    p = np.array([
        [2,2,0,-2,-2,0,0,0,0,0,0],
        [0.5, -2, 0, 0.5, -2, 0,0,0,0,0,0]
    ])
    T = np.array([10, 20, 30, 40, 50, 60, 61, 62, 63, 64, 65])
    A = np.array([
        [1, 1, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 1],
        [0, 0, 0, 1]
    ])
    B = np.array([
        [0.5, 0],
        [1, 0],
        [0, 0.5],
        [0, 1]
    ])
    Bg = np.array([
        [0],
        [0],
        [-0.5],
        [-1]
    ])
    return A, B, Bg, T, p
```

[472]:
```
def createMatrixA(A, B, T):
    allAts = []
    for t in T:
        matrices = []
        currPow = np.eye(A.shape[0])
        for _ in range(t):
            matrices.append(np.dot(currPow, B))
            currPow = np.dot(currPow, A)
        matrices.reverse()
        zeros = np.zeros((A.shape[0], 2*np.max(T) - 2*t))
        At = np.concatenate(matrices + [zeros], axis=1)
        At = np.dot(np.array([
            [1, 0, 0, 0],
            [0, 0, 1, 0]
        ]), At)
        assert At.shape == (2, 130)
        allAts.append(At)
    Ahat = np.vstack(allAts)
    assert Ahat.shape == (22, 130)
    return Ahat

def createYHat(A, Bg, T, p):
    yts = []
    for i, t in enumerate(T):
        Asum = np.zeros(A.shape)
        At = np.eye(A.shape[0])
        for _ in range(t):
            Asum += At
            At = np.dot(At, A)
        pt = p[:, i]
        pt.shape = (2,1)
        yt = pt - np.dot(np.array([
            [1, 0, 0, 0],
            [0, 0, 1, 0]
        ]), np.dot(Asum, Bg))
        assert yt.shape == (2,1)
        yts.append(yt)
    yhat = np.vstack(yts)
    assert yhat.shape == (22, 1)
    return yhat
```

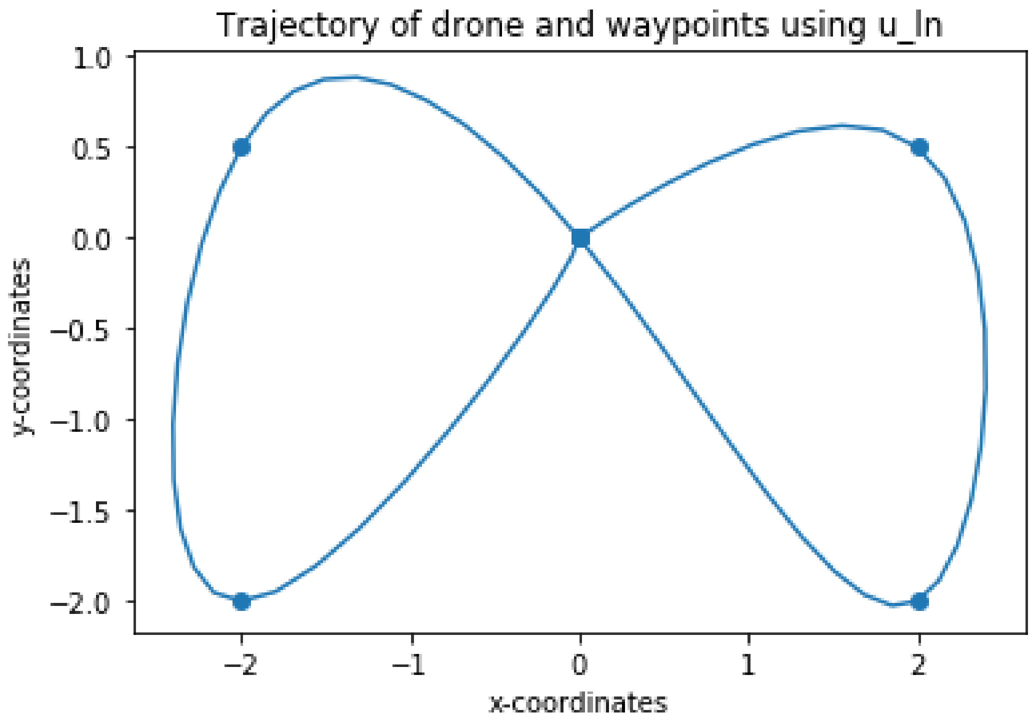
[591]:
```
A, B, Bg, T, p = getProblem1Inputs()
Ahat = createMatrixA(A, B, T)
yhat = createYHat(A, Bg, T, p)
uln = np.dot(np.dot(Ahat.T, np.linalg.inv(np.dot(Ahat, Ahat.T))), yhat)
```

```python
# plot optimal u against time
plt.plot(range(65), uln[:len(uln):2], label="x forces")
plt.plot(range(65), uln[1:len(uln):2], label="z forces")
plt.legend(loc='upper left')
plt.xlabel('Time')
plt.ylabel('Exerted force')
plt.title('Least norm forces exerted on drone.')
plt.savefig('../final/least_norm_forces.png')
```



[592]:
```python
def plotTrajectory(A, B, Bg, u):
    state = np.zeros((4,1))
    xs = [0]
    zs = [0]
    for t in range(0,130,2):
        ut = np.array([u[t],u[t+1]])
        state = np.dot(A, state) + np.dot(B, ut) + Bg
        xs.append(state[0])
        zs.append(state[2])
    plt.plot(xs, zs)
```

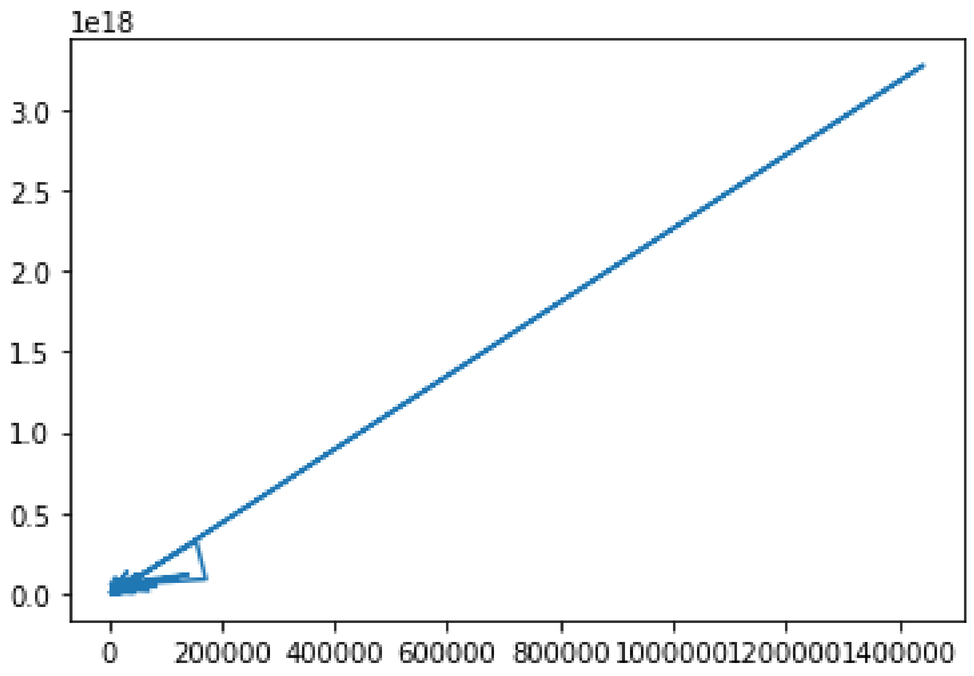
[593]:
```python
plt.scatter(p[0,:], p[1,:])
plotTrajectory(A, B, Bg, uln)
plt.title('Trajectory of drone and waypoints using u_ln')
plt.xlabel("x-coordinates")
plt.ylabel("y-coordinates")
plt.savefig('../final/drone_trajectory.png')
```



[598]:
```
# Plot tradeoff curves.
def plotTradeoffCurves(A, y):
    J1s = []
    J2s = []
    for mu in np.logspace(0,7,50):
        Abig = np.vstack((A, mu * np.eye(65, M=A.shape[1])))
        ybig = np.vstack((y, np.zeros((65,1))))
        muStar = np.dot(np.dot(np.linalg.inv(np.dot(Abig.T, Abig)), Abig.T),␣
 ↪ybig)

        optJ1 = np.linalg.norm(np.dot(A, muStar) - y)
        optJ2 = np.linalg.norm(muStar)
        J1s.append(optJ1)
        J2s.append(optJ2)
    plt.plot(J1s, J2s)
    return (J1s, J2s)
```

[599]:
```
J1, J2 = plotTradeoffCurves(Ahat, yhat)
```



[585]:
```
J1
```

[585]:
```
[11770.778386332018]
```

[586]:
```
J2
```

[586]:
```
[3132869672841160.0]
```

1.2 Problem 2

[192]:
```
def getProblem2Inputs():
    A1 = np.array([
        [0] * 7 + [1] + [0] * 7,
        [0] * 7 + [1, -1] + [0] * 6,
        [0] * 8 + [1, -1] + [0] * 5,
        [0] * 9 + [1, -1] + [0] * 4,
        [0] * 10 + [1, -1] + [0] * 3,
        [0] * 11 + [1, -1] + [0] * 2,
        [0] * 12 + [1, -1] + [0] * 1,
        [-1] + [0] * 6 + [-1] + [0] * 6 + [7],
        [0] * 1 + [1] + [0] * 6 + [-1] + [0]*5 + [-1],
        [0] * 2 + [1] + [0] * 6 + [-1] + [0]*4 + [-1],
        [0] * 3 + [1] + [0] * 6 + [-1] + [0]*3 + [-1],
        [0] * 4 + [1] + [0] * 6 + [-1] + [0]*2 + [-1],
        [0] * 5 + [1] + [0] * 6 + [-1] + [0]*1 + [-1],
        [0] * 6 + [1] + [0] * 6 + [-1] + [-1],
        [0] * 15
    ])
    A2 = np.array([
        [0] * 7 + [1] + [0] * 7,
        [0] * 7 + [1, -1] + [0] * 6,
        [0] * 8 + [1, -1] + [0] * 5,
        [0] * 9 + [1, -1] + [0] * 4,
        [0] * 10 + [1, -1] + [0] * 3,
        [0] * 11 + [1, -1] + [0] * 2,
        [0] * 12 + [1, -1] + [0] * 1,
        [-1] + [0] * 6 + [-1] + [0] * 6 + [7],
        [0] * 1 + [0.5, -0.5] + [0] * 5 + [-1] + [0]*6,
        [0] * 2 + [0.5, -0.5] + [0] * 5 + [-1] + [0]*5,
        [0] * 3 + [0.5, -0.5] + [0] * 5 + [-1] + [0]*4,
        [0] * 4 + [0.5, -0.5] + [0] * 5 + [-1] + [0]*3,
        [0] * 5 + [0.5, -0.5] + [0] * 5 + [-1] + [0]*2,
        [0] * 6 + [1] + [0] * 6 + [-1] + [-1],
        [0] * 15
    ])
    A3 = np.array([
        [0] * 7 + [1] + [0] * 7,
        [0] * 8 + [1] + [0] * 6,
        [0] * 9 + [1] + [0] * 5,
        [0] * 10 + [1] + [0] * 4,
        [0] * 11 + [1] + [0] * 3,
        [0] * 12 + [1] + [0] * 2,
        [0] * 13 + [1] + [0] * 1,

        [-1] + [0]*6 + [-1] + [0]*6 + [7],
        [0]* 1 + [-1] + [0]*6 + [-1] + [0]*5 + [6],
        [0]* 2 + [-1] + [0]*6 + [-1] + [0]*4 + [5],
        [0]* 3 + [-1] + [0]*6 + [-1] + [0]*3 + [4],
        [0]* 4 + [-1] + [0]*6 + [-1] + [0]*2 + [3],
        [0]* 5 + [-1] + [0]*6 + [-1] + [0]*1 + [2],
        [0]* 6 + [-1] + [0]*6 + [-1] + [1],
        [0] * 15
    ])
    return A1, A2, A3
```

[193]:
```
A1, A2, A3 = getProblem2Inputs()
```

[194]:
```
w1, v1 = np.linalg.eig(A1)
indexMin = np.argmin(np.abs(np.real(w1)))
eigenVector = v1[:, indexMin]
np.real(eigenVector)
```

[194]:
```
array([ 0.935,  0.134,  0.134,  0.134,  0.134,  0.134,  0.134, -0.   ,
       -0.   , -0.   , -0.   , -0.   , -0.   , -0.   ,  0.134])
```

[195]:
```
w1
```

[195]:
```
array([-0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j, -0.5+0.866j,
       -0.5-0.866j, -0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j,
       -0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j,  0.  +0.j   ])
```

[196]:
```
w2, v2 = np.linalg.eig(A2)
indexMin2 = np.argmin(np.abs(np.real(w2)))
eigenVector2 = v2[:, indexMin2]
np.real(eigenVector2)
```

[196]:
```
array([ 0.935,  0.134,  0.134,  0.134,  0.134,  0.134,  0.134, -0.   ,
       -0.   , -0.   , -0.   , -0.   , -0.   , -0.   ,  0.134])
```

[197]:
```
w2
```

[197]:
```
array([-0.5  +1.31j , -0.5  -1.31j , -0.5  +1.207j, -0.5  -1.207j,
       -0.035+0.j   , -0.965+0.j   , -0.5  +0.207j, -0.5  -0.207j,
       -0.5  +1.004j, -0.5  -1.004j, -0.5  +0.701j, -0.5  -0.701j,
       -0.5  +0.866j, -0.5  -0.866j,  0.   +0.j   ])
```

[198]:
```
w3, v3 = np.linalg.eig(A3)
indexMin3 = np.argmin(np.abs(np.real(w3)))
eigenVector3 = v3[:, indexMin3]
np.real(eigenVector3)
```

[198]:
```
array([ 0.59 ,  0.505,  0.421,  0.337,  0.253,  0.168,  0.084, -0.   ,
       -0.   ,  0.   , -0.   , -0.   , -0.   , -0.   ,  0.084])
```

[260]:
```
w3
```

[260]:
```
array([-0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j, -0.5+0.866j,
       -0.5-0.866j, -0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j,
       -0.5+0.866j, -0.5-0.866j, -0.5+0.866j, -0.5-0.866j,  0.  +0.j   ])
```

[270]:
```
# Verify some of this stuff works.
for i in range(100):
    x = np.dot(linalg.expm(100*A1), np.concatenate((np.random.rand(14), [1])))
    assert np.allclose(x, np.array([7,1,1,1,1,1,1,0,0,0,0,0,0,0,1]))
```

[271]:
```
for i in range(100):
    x = np.dot(linalg.expm(1000*A2), np.concatenate((np.random.rand(14), [1])))
    assert np.allclose(x, np.array([7,1,1,1,1,1,1,0,0,0,0,0,0,0,1]))
```

[272]:
```
for i in range(100):
    x = np.dot(linalg.expm(100*A3), np.concatenate((np.random.rand(14), [1])))
    assert np.allclose(x, np.array([7,6,5,4,3,2,1,0,0,0,0,0,0,0,1]))
```

[322]:
```
def runSimulation(A, initialState, usesSpaces=True):
    if usesSpaces:
        xFinal = np.array([7,1,1,1,1,1,1,0,0,0,0,0,0,0,1])
    else:
        xFinal = np.array([7,6,5,4,3,2,1,0,0,0,0,0,0,0,1])
    t = 0.05
    state = initialState.flatten()
    closestCollision = None
    closestCollisionTime = None
    jetsInvolved = None
    while not np.allclose(state, xFinal):
        state = np.dot(linalg.expm(t * A), initialState).flatten()
        if usesSpaces:
            distances = state[1:7]
        else:
            distances = (state[:6] - state[1:7])

        # Find min space. If negative, a collision happend.
        minSpace = distances.min()
        if minSpace <= 0:
            for jet in np.array(range(7,1,-1))[distances < 0]:
                print('Collision occured at t = %s between jet %s and jet %s.'␣
↪% (t, jet, jet-1))
            return
        elif not closestCollision or (closestCollision > minSpace):
            closestCollision = minSpace
            closestCollisionTime = t
            jetsInvolved = np.array(range(7,1,-1))[np.abs(distances - minSpace)␣
↪< 1e-4]
        t += 0.05

    for jet in jetsInvolved:
        print('Closest collision occured at t = %s between jet %s and jet %s␣
↪with spacing of %s.' % (
            closestCollisionTime, jet, jet-1, closestCollision))
```

[323]:
```
initialConditionWithSpaces = np.array([
    [8],
    [1],
    [1],
    [2],
    [2],
    [1],
    [1],
    [0],
    [0],
    [0],
    [0],
    [0],
    [0],
    [0],
    [1]
])
```

```
runSimulation(A1, initialConditionWithSpaces)
```

```
Collision occured at t = 8.349999999999984 between jet 2 and jet 1.
```

[324]:
```
# Verify results. We see that the s_1 = -0.001 (a collision).
np.dot(linalg.expm(8.35 * A1), initialConditionWithSpaces)
```

[324]:
```
array([[ 7.016],
       [ 0.96 ],
       [ 1.188],
       [ 1.343],
       [ 0.977],
       [ 0.356],
       [-0.001],
       [-0.014],
       [ 0.04 ],
       [ 0.208],
       [ 0.089],
       [-0.341],
       [-0.714],
       [-0.607],
       [ 1.   ]])
```

[325]:
```
runSimulation(A2,initialConditionWithSpaces)
```

```
Closest collision occured at t = 2.3499999999999996 between jet 7 and jet 6 with
spacing of 0.31462333372486384.
```

[326]:
```
np.dot(linalg.expm(2.35 * A2), initialConditionWithSpaces)
```

[326]:
```
array([[ 7.021],
       [ 0.315],
       [ 1.222],
       [ 1.485],
       [ 1.501],
       [ 1.399],
       [ 1.081],
       [-0.319],
       [-0.326],
       [-0.211],
       [-0.002],
       [ 0.176],
       [ 0.111],
       [ 0.034],
       [ 1.   ]])
```

[329]:
```
initialConditionWithoutSpaces = np.array([
    [8],
    [7],
    [6],
    [4],
    [2],
    [1],
    [0],
    [0],
    [0],
    [0],
    [0],
    [0],
    [0],
    [0],
    [1]
])
runSimulation(A3,initialConditionWithoutSpaces, usesSpaces=False)
```

```
Closest collision occured at t = 3.649999999999995 between jet 5 and jet 4 with
spacing of 0.8370070662304583.
Closest collision occured at t = 3.649999999999995 between jet 4 and jet 3 with
spacing of 0.8370070662304583.
```

[330]:
```
np.dot(linalg.expm(3.65 * A3), initialConditionWithoutSpaces)
```

[330]:
```
array([[ 6.837],
       [ 5.837],
       [ 4.837],
       [ 4.   ],
       [ 3.163],
       [ 2.163],
       [ 1.163],
       [ 0.004],
       [ 0.004],
       [ 0.004],
       [-0.   ],
       [-0.004],
       [-0.004],
       [-0.004],
       [ 1.   ]])
```

[ ]:

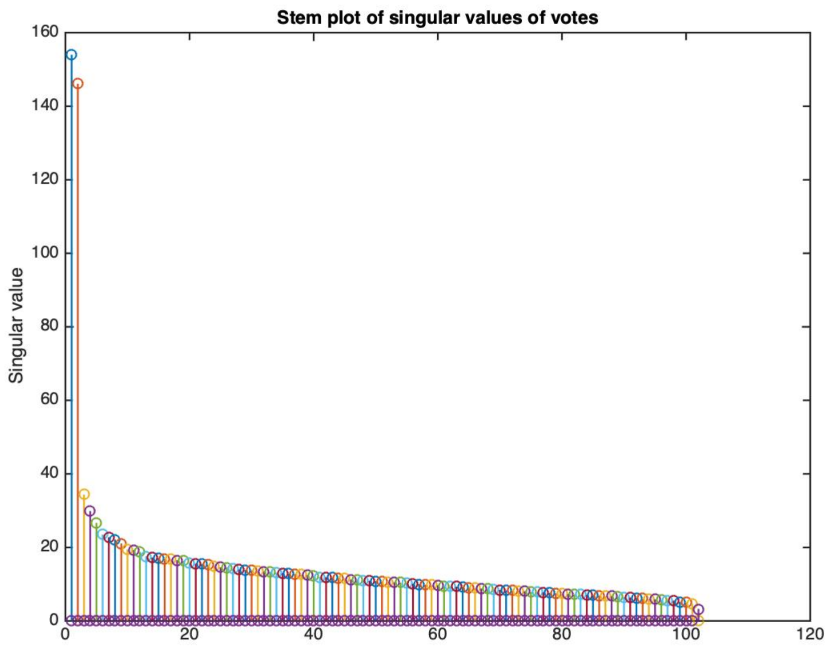


Figure 3: Stem plot of singular values for votes matrix

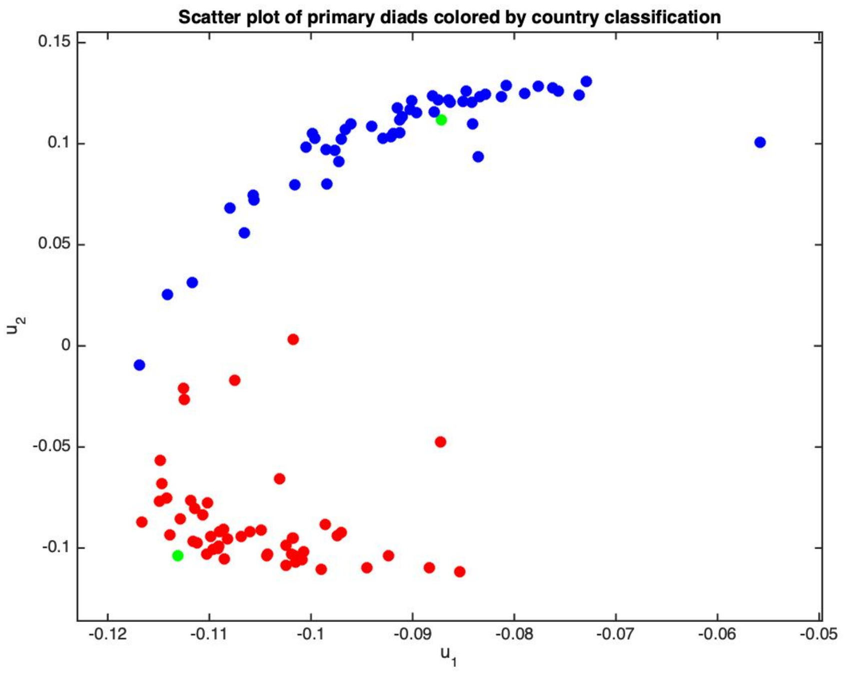


Figure 4: Scatter plot of u_1 and u_2 by country classification where Capitalists, Communists and Socialists are represented by red, blue, and green markers, respectively.

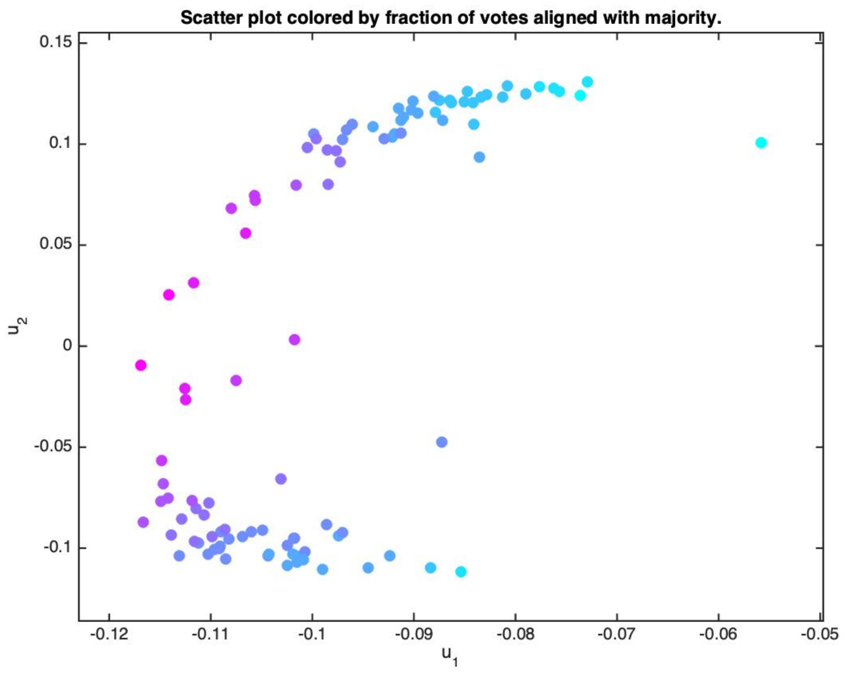


Figure 5: Scatter plot of u_1 and u_2 showing the fraction of votes in which each country voted with the majority of the other countries. Pink implies large values, and blue small values.

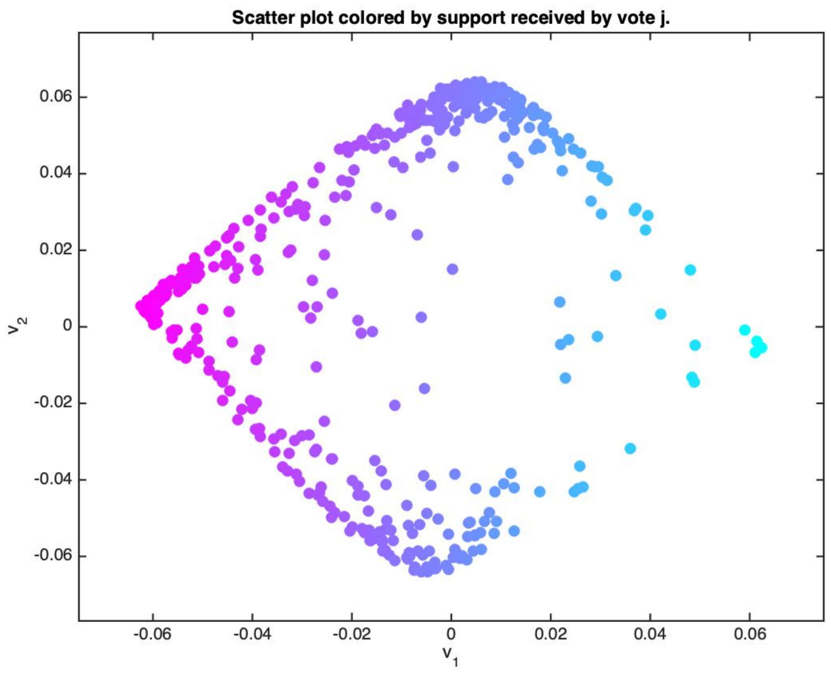


Figure 6: Scatter plot of v_1 and v_2 where colors indicate the total support received by vote j

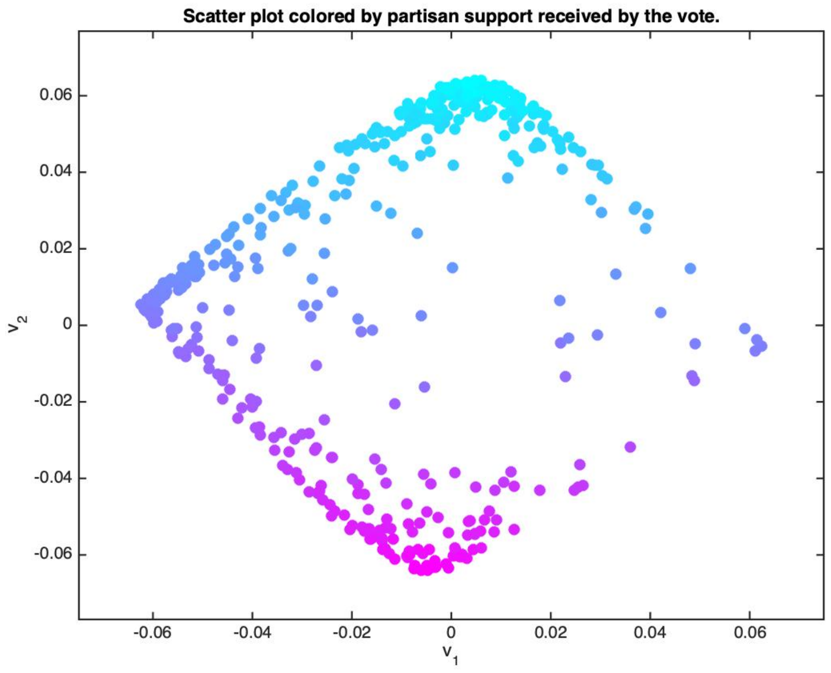


Figure 7: Scatter plot of v_1 and v_2 where the colors indicate the partisan support received by vote j