1 Introduction

I first came across the need to understand the Jordan form when studying cointegration techniques. Not being okay with making use of a thing I did not understand, and it not being a trivial yet an essential thing for me to understand, I ended up devoting some time to it. Putting a matrix in Jordan form has numerous applications. It is somewhat foundational to linear algebra, and by extension it finds its place in numerous computational applications. As far as data science goes, aside from being a necessary tool in powerful methods such as co-integration and Markov-chain analysis, I believe that possessing a firm understanding of the subspaces of a matrix is an indispensable means of extracting meaningful information from it. At the very least it is a start. It is, for example, no coincidence that eigenvalues find their place on almost every other page of many quantum physics text-books; eigenspaces are essential to our ability to bridge the gap between the world of numbers and physically observable phenomena.

This code is more or less a computational version of the detailed analysis given in reference [1]. I will not bother with rigorous proofs which are heavy in esoteric math lingo as they can be found in reference [1] and elsewhere. Here I aim to describe in [as much as possible] laymans terms both the essence of the mathematical steps being executed and how they relate to the code which follows the theoretical steps executed in reference [1].

Note that there are a number of codes available on the internet which perform Jordan decomposition in a much more succinct fashion. Part of the reason for this is because I utilize my own algorithms for solving linear systems, to find range and null basis, performing Gaussian elimination, finding the minimum polynomial, etc. Were it not for these files/ functions the code would largely boil down to just two functions contained in the $funcs\_transforms.py$ file. The finding of the minimum polynomial function can and probably should be kept in some file other than the main one ($Jordan_decomposition.py$), but I thought it useful to have all the print statements which guide the user through the steps being executed in one single file.

2 Matrix polynomials

A polynomial of order n is a function of the form,

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n$$

By the fundamental theory of algebra this polynomial has n roots, so it can be factored into *monic polynomials*, e.g. terms of the form (x-λ) where λ is some constant.

Perhaps a bit counter intuitively, matrices can also be used as arguments in a polynomial. Given a matrix **A** we have.

$$p(\mathbf{A}) = a_0 + a_1\mathbf{A} + a_2\mathbf{A}^2 + a_3\mathbf{A}^3 + ... + a_n\mathbf{A}^n \tag{1}$$

If there exist a polynomial such that (1) equals zero, and furthermore if this is the *absolute lowest* degree polynomial which exists such that $p(\mathbf{A}) = 0$ then we've found what is known as the minimum polynomial of **A** denoted as $P_{min}(\mathbf{A})$.

In many cases we find this polynomial is actually the product of smaller polynomial factors. In the example followed in the code and in reference [1] we get,

$$P_{min}(x) = (x-1) * (x^4 + 2x^2 + 1) = (x-1) * (x^2+1)^2 = p_1 * p_2 = 0 \tag{2}$$

This example is derived from a square matrix with dimension (length) of 6. This matrix is,

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & -1 & -1 \\ -3 & 8 & 5 & 5 & 2 & -2 \\ 1 & 0 & -1 & 0 & -1 & 0 \\ 4 & -10 & -7 & -6 & -3 & 3 \\ -1 & 3 & 2 & 2 & 2 & -1 \\ -2 & 6 & 4 & 4 & 2 & -1 \end{pmatrix}$$

It is of no coincidence that the order of the polynomial in eqn. (2) is 6 – a thing we'd find were we to expand it. This has relation to the fact that the null space of a matrix stabilizes (or does not increase) beyond the point of being raised to the power of its dimension $N = 6$ (or sometimes L in coding because it is the length of a given matrix). See ref [2] for details on this.

We can make use of the fundamental theorem of algebra to factor a gien polynomial even further down monic terms, albeit at the cost that they will often involve imaginary terms. In the general case we have,

$$P_{min} = (\mathbf{A}) = (\mathbf{A} - \lambda_0)^{k_0} * (\mathbf{A} - \lambda_1)^{k_1} * ... * (\mathbf{A} - \lambda_n)^{k_n} = 0$$

For the minimum polynomial only the λ values are referred to as $eigenvalues$. The powers to which the terms are raised to is referred to as the *algebraic multiplicity* of the monic polynomial terms. In the code I refer to the algebraic multiplicity as the degeneracies of the factors. This is a misnomer which I at first mistakenly equated to the concept of degeneracy of eigenvalues in physics. As it turns out degeneracy of eigenvectors is in math terms the *geometric multiplicity* of eigenvectors.

2.1 Linear independence

given a set of vectors $v_1, v_2, ..., v_n$ if the *only* set of coefficients which solve the equation,

$$\sum_{i=0}^{n} \alpha_i v_i = 0$$

are $\alpha_0, \alpha_1, ..., \alpha_n = 0, 0, ..., 0$ we say the set of vectors $v_1, v_2, ..., v_k$ are *linearly independent*. if even one of the α coefficients can be given a non-zero value we say this set of vectors are *linearly dependent*.

2.2 Finding the Minimum Polynomial

We seek a polynomial P_{min} such that $P_{min}(\mathbf{A}) = 0$. This is to say that we seek a polynomial of $\mathbf{A}$ which annihilates the matrix $\mathbf{I}$. More specifically, we seek the lowest degree polynomial which does so. To this end consider the matrix $\mathbf{I}$ as an array of unit column vectors e_j;

$$\mathbf{I} = [e_1, e_2, ..., e_n]$$

We first seek to annihilate the vector e_1 with the lowest number of vectors of the form $A^i e_1$ possible. The set of vectors $\{e_1, Ae_1, A^2e_1, ...A^m e_1\}$ is the minimal linearly dependent set of vectors of this form, which is to say there will be only one dependent variable. We then have found our first factor of P_{min} and it is,

$$p_1(\mathbf{A}) = \sum_{i=0}^{m} x_i A^i$$

where x_i are real or imaginary coefficients. $p_1(\mathbf{A})$ has the property that $p_1(\mathbf{A})e_1 = [0]$.

Now we move onto e_2. It may happen to be the case that $p_1(\mathbf{A})e_2 = [0]$ in which case we'd move onto e_3. If $p_1(\mathbf{A})$ annihilates all the unit vectors e_i which span the dimension of our matrix then we've found the minimum polynomial.

Assuming this not to be the case, we move onto e_2 in a way which does not disregard the work we've done thus far. Instead of taking e_2 as our test vector, lets take $v = p_1(\mathbf{A})e_2$ (notice how we carried with us the originally derived polynomial factor which we found to annihilate e_1). We construct a polynomial term $p_2(\mathbf{A})$ such that $p_2(\mathbf{A})v = p_2(\mathbf{A})[p_1(\mathbf{A})e_2] = [0]$. Ergo, were we to have need to obtain a distinct polynomial factor for every unit vector annihilated, our test vector v at the i^{th} step in the process will have the general form

$$v = p_{i-1}(\mathbf{A}) * ... * p_2(\mathbf{A})p_1(\mathbf{A})e_i$$

This is accomplished by updating the string P_{min} on every iteration then evaluating the expression with inputs $\mathbf{A}, \mathbf{I}$ where $\mathbf{I}$ is the identity matrix (used for multiplying constant terms in *scypy* strings). Iterating the procedure we eventually get,

$$P_{min}(\mathbf{A}) = p_n(\mathbf{A}) * ... * p_2(\mathbf{A}) * p_1(\mathbf{A}) = \{\mathbf{0})\}$$

This all is what lines 20-130 of the code accomplishes. Once a minimally dependent set of vectors are found which annihilate our test vector v for all e_i unit vectors then the coefficients of the polynomial factors are returned in a list,

$$X = [[x_1], [x_2], ..., [x_n]] \tag{3}$$

where $[x_i]$ stands for the coefficients (np.array) of the polynomial factor p_i. As we will want to factor these polynomial terms even further (down to their monic roots), from here we *could* make use of the *scypy* factor function, but we'd find this often does not factor expressions down to their monic terms. If factoring requires floating point numbers (a thing we ought to expect in real life data analysis) then scypy won't factor it. So we instead rely on *np.roots* to solve for the roots then construct a *scypy* polynomial string ourselves (c.f. the functions utilized on lines 68-69). We can always implement a more robust root finding algorithm if and when needed – plenty are available out there.

2.3 Constructing Null-spaces: Lemma 1

Instead of matrix $\mathbf{A}$ lets consider a more general matrix $\mathbf{T}$ which is often used to represent one of the monic polynomial terms in the minimum polynomial of $\mathbf{A}$, e.g. $\mathbf{T} = (\mathbf{A} - \lambda\mathbf{I})$. Here I paraphrase without proof the essential principle of lemma 1 from ref. [1];

Lemma 1: *Given a vector u, a matrix $\mathbf{T}$ and integer k such that $\mathbf{T}^k u = [0]$ we can construct vectors $u_0, u_1, u_2, ..., u_k = u_0, Tu_0, T^2u_0, ..., T^{k-1}u_k$ and these vectors are guaranteed to form a basis of the* nullspace *of T, which is the set of vectors w which satisfy the equation $\mathbf{T}w = [0]$.*

It is not difficult to see then that each of the vectors u_i satisfy the null-space criteria, if only we recognize that they do so with their own unique k_i value. For example, $\mathbf{T}^{k_i}u_i = \mathbf{T}^{k_i}(\mathbf{T}^i u_0) = 0$ if we have $k = k_i + i$ – a thing which is assured by our choice of basis – then we also have what we were given in the first place; $\mathbf{T}^k u = [0]$.

**Note: There are [at least] two ways of finding a basis for the null-space. One is the more common way of looking at the Gaussian reduced form of* $\mathbf{T}$*. This is what the function* $null\_basis$ *in the file* $range\_null\_basis.py$ *does. But this method* will not *yield for us a transformation matrix that results in the Jordan form. We instead need to rely on lemma 1 – when appropriate. But first, as in line with the steps followed in ref. [1], we* do *decompose the matrix* $\mathbf{T}$ *with use of the function* $null\_basis$*. So we'll perform not one but two transformations on matrix* $\mathbf{T}$*, as in line with reference [1].*

2.4 Relating Null-spaces to Polynomials

As demonstrated in the additional section at the very end of ref. [1], if we have that $P_min(\mathbf{A}) = p_1(\mathbf{A}) * p_2(\mathbf{A}) = \mathbf{0}$ and furthermore, that the greatest common divisor of $p1(\mathbf{A})$ and $p_2(\mathbf{A})$ is 1, then the set of vectors $\{v_0^1, v_1^1, v_2^1, ...\} \in V_1$ and $\{v_0^2, v_1^2, v_2^2, ...\} \in V_2$ defined such that $p_1(\mathbf{A})v_i^1 = 0$ and $p_2(\mathbf{A})v_i^2 = 0$ form a set of basis vectors which are are A) invariant, and B) together span the entire dimension of $\mathbf{A}$.

Formally stated, invariance of a subspace means that if $v \in V$ where v is a vector and V is a subspace, then $f(v) \in V$ as well. We can see the relevance of seeking invariant subspaces to eigen-decomposiont (our ultimate aim) by considering the operation $f(v) = \mathbf{A}v = \lambda v \in V$.

Invariance of two subspaces implies they can be written as a direct sum, i.e. $V_1 + V_2 = V_1 \oplus V_2$. In terms of matrices this can be visualized as,

$$\mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \left(\begin{array}{cc|ccccccc} & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & V_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & & & & & & & \\ 0 & 0 & & & & & & & \\ 0 & 0 & & & V_2 & & & & \\ 0 & 0 & & & & & & & \end{array}\right)$$

So if $p_1(\mathbf{A}) * p_2(\mathbf{A}) = \{\mathbf{0}\}$ then we are free to pick column vectors from the matrix $p_2(\mathbf{A})$ as vectors v_i^1 which satisfy $p_1(\mathbf{A})v_i^1 = 0$. But how do we pick the v_i^2 vectors? Thankfully, our polynomial is composed of only one argument ($\mathbf{A}$), that is to say it is *univariate*. In this case only polynomial factors do commute, which is to say they can be shuffled around: $p_1(\mathbf{A}) * p_2(\mathbf{A}) = \{\mathbf{0}\} \implies p_2(\mathbf{A}) * p_1(\mathbf{A}) = \{\mathbf{0}\}$. So we now are able to pick some column vector(s) from $p_1(\mathbf{A})$ as the v_i^2 vectors.

Problem: how do we know we've found *all* vectors v_i^1 and v_i^2 which satisfy $p_1(\mathbf{A})v_i^1 = 0$ and $p_2(\mathbf{A})v_i^2 = 0$? Simply picking some column vectors from either of the matrices is not exactly a rigorous approach. As demonstrated by the author of ref. [1] demonstrates, this approach happens to work out for the example considered, and she does show her choice to be in line with some of the more rigorous criteria involved in lemma 1 which I've not mentioned. But in terms of coding we need a *systematic* means of selecting a transformation basis – preferably one which also happens to be in line with theory.

To be clear, the transformation currently being discussed is a transformation which is based on the null-spaces (not *necessarily* the same thing as eigen-spaces!) of the p_n factors.

To this end consider the previous arguments in a slightly different way; if $p_1(\mathbf{A}) * p_2(\mathbf{A}) = 0$, and if the two polynomials commute, then whatever the true and complete null-space happens to be for $p_1(\mathbf{A})$, the columns of $p_2(\mathbf{A})$ are certainly included in it. As mentioned previously, using Gaussian reduction it is possible to find a basis of the null-space for a matrix. The file $null\_range\_basis.py$ has functions which compute the range or null-space basis for an input matrix.

As demonstrated in ref. [2] the range and null-space of a matrix together form invariant subspaces, which ought to come as no surprise to us considering the above arguments. We can then simply find the range and null-space of just $p_1(\mathbf{A})$. The *rank–nullity theorem* guarantees that,

$$dim\,[\mathbf{A}] = dim\,[\mathcal{R}(\mathbf{p}_1(\mathbf{A}))] + dim\,[\mathcal{N}(\mathbf{p}_1(\mathbf{A}))]$$

Furthermore, the null and range together span the entire space of $\mathbf{p}_1(\mathbf{A})$, and are invariant which means they may be written as a direct sum,

$$\mathcal{S}(\mathbf{p}_1(\mathbf{A})) = \mathcal{R}(\mathbf{p}_1(\mathbf{A})) \oplus \mathcal{N}(\mathbf{p}_1(\mathbf{A}))$$

The dimension theorem assures us that the number of independent column vectors we are able to pull from the range and null-space of $\mathbf{p}_1(\mathbf{A})$ together should exactly equal the dimension of this matrix (which of course has the same dimension as $\mathbf{A}$ – presumed to be a square matrix). Thus, we ought to expect to be able to construct a transformation matrix $\mathbf{S}$ with N columns. Were we to perform a change of basis on $P_{min}(\mathbf{A})$ by the matrix $\mathbf{S}_{null} = [\mathcal{N}(p_1(\mathbf{A}))), \mathcal{R}(p_1(\mathbf{A}))]$, then we ought to see the invariance once we perform this transformation. Lets the call $\mathbf{A}^1$ the transformed equivalent of $\mathbf{A}$

$$P_{min}(\mathbf{A}^1) = \mathbf{S}_{null}^{-1} P_{min}(\mathbf{A}) \mathbf{S}_{null} = \left(\begin{array}{cc|ccccccc} & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \multicolumn{2}{c|}{\mathbf{p}_1(\mathbf{A}^1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & & & & & & & \\ 0 & 0 & & & & & & & \\ 0 & 0 & \multicolumn{7}{c}{p_2(\mathbf{A}^1) .. * p_n(\mathbf{A}^1)} \\ 0 & 0 & & & & & & & \end{array}\right) = \{\mathbf{0}\}$$

Iterating this procedure we eventually get,

$$P_{min}(\mathbf{A}^n) = p1(\mathbf{A}^n) \oplus p2(\mathbf{A}^n) \oplus p3(\mathbf{A}^n) \oplus ... \oplus pn(\mathbf{A}^n) = \{\mathbf{0}\} \oplus \{\mathbf{0}\} \oplus \{\mathbf{0}\} \oplus ... \oplus \{\mathbf{0}\} \tag{4}$$

At each step i we update the transformation basis $\mathbf{S}_{null} = \mathbf{S}_{null}^i * \mathbf{S}_{null}$ so as to obtain the final resulting transformation matrix, which when operating on $\mathbf{P}_{min}(\mathbf{A})$ gives us the transformed matrix,

$$\mathbf{p}_{min}(\mathbf{A}^n) = \mathbf{S}_{null}^{-1} \mathbf{p}_{min} = \left[\begin{array}{cc|cccccc} & & 0 & 0 & \cdots & \cdots & 0 & 0 \\ \multicolumn{2}{c|}{p_1(\mathbf{A}^n)} & 0 & 0 & \cdots & \cdots & 0 & 0 \\ \hline 0 & 0 & \multicolumn{2}{|c}{\multirow{2}{*}{$p_2(\mathbf{A}^n)$}} & \cdots & \cdots & 0 & 0 \\ 0 & 0 & \multicolumn{2}{|c}{} & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \multicolumn{2}{c|}{\multirow{2}{*}{$\ddots$}} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \multicolumn{2}{c|}{} & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \multicolumn{2}{c}{\multirow{2}{*}{$p_n(\mathbf{A}^n)$}} \\ 0 & 0 & 0 & 0 & 0 & 0 & \multicolumn{2}{c}{} \end{array}\right]. \tag{5}$$

What we have just accomplished is as follows:

$$\begin{aligned} p_1(\mathbf{A}) &= \{\mathbf{0}\} \\ p_2(\mathbf{A}) &= \{\mathbf{0}\} \\ &\vdots \\ p_n(\mathbf{A}) &= \{\mathbf{0}\} \end{aligned}$$

Ultimately it is our goal to decompose the matrix $\mathbf{p}_{min}(\mathbf{A}^n)$ and like terms according to their most irreducible eigen-spaces. This we will see in the next section is equivalent utilizing lemma 1 so as to find the eigen-spaces of each of the *monic* polynomial terms. Here we've established a useful

method for reducing the minimum polynomial down to its larger terms, which may or may not happen to be monic terms.

2.5 Eigen Decomposition

Consider the factorization of the larger term in equation 2, which now can be set to the zero matrix

$$p_2(\mathbf{A}) = (\mathbf{A}^2 + \mathbf{I})^2 = \mathbf{0}$$

can instead be represented as

$$(\mathbf{A} - \mathbf{I}i)^2(\mathbf{A} + \mathbf{I}i)^2 = \mathbf{0} \tag{6}$$

In the code it is messy because $np.roots$ will return something like $(-0-1j)$ (j being the equivalent of i in python). So we will end up constructing a polynomial factor term of the form $(x-(-0-1j))$ where x is the symbol we use for constructing $scypy$ strings to be evaluated with inputs $\mathbf{A}, \mathbf{I}$. This is the cost of relying on root-finding algorithms to perform factorization, but as previously described we must do this as $scypy$ cannot be trusted to factorize anything which involves floating point numbers.

With each p_n term factored to its monic roots and raised to the power of their algebraic multiplicity (mistakenly referred to as their degeneracies in the code) we are in a position to find a basis which is in line with lemma 1. We will construct a set of basis vectors – an eigen-base – which allows us to perform a second transformation, the end result of which will be the Jordan form.

To caste the above example in terms of lemma 1 we identify the matrix $\mathbf{T}$ as $\mathbf{A}-\mathbf{I}i$ and the algebraic multiplicity $k = 2$ with the property that $\mathbf{T}^k u = [0]$ where u is any column picked from the matrix $(\mathbf{A} + \mathbf{I}i)^2$. We then have as an eigen basis vector from which we can build the set $\{u, \mathbf{T}u\}$ which forms the [eigen] subspace corresponding to eigen-value i, so lets call the eigen-vector $u = u_{+i}$. The corresponding basis vectors are,

$$\{u_{+i}, \mathbf{T}u_{+i}\} = \left\{\left[(\mathbf{A} + \mathbf{I}i)^2\right]_j, (\mathbf{A} - \mathbf{I}i)\left[(\mathbf{A} + \mathbf{I}i)^2\right]_j\right\}$$

where the subscript j indicates we've picked some colum from the matrix. Recalling that univariate matrix polynomial commute, we can shuffle eq. 2.5 to get,

$$(\mathbf{A} + \mathbf{I}i)^2(\mathbf{A} - \mathbf{I}i)^2 = \mathbf{0}$$

repeating the previous steps we get for a basis for the eigenvalue $-i$,

$$\{u_{-i}, \mathbf{T}u_{-i}\} = \left\{\left[(\mathbf{A} - \mathbf{I}i)^2\right]_j, (\mathbf{A} + \mathbf{I}i)\left[(\mathbf{A} - \mathbf{I}i)^2\right]_j\right\}$$

Algorithmically, how to achieve the above requires we loop through a given list of polynomial coefficients (in this case [1, -i], [1, i]). These np.arrays are the $[x_i]$ values in 3. We then construct a list of scypy strings from he coefficients in $[x_i]$ and their corresponding degeneracies (stored in a similar list). We loop through this list of factors, bring the factor in question to the left of all other factors, evaluate it, and label it as the matrix $\mathbf{T}$. Simultaneously, we put together and evaluate all the other factors, evaluate them as a $scypy$ string, evaluate them, and label this the matrix $\mathbf{U}$. We then manage to find one non-zero column of $\mathbf{U}$ through the function $range\_basis$ which gives a [non-zero] basis set for the range of input matrix U – this will serve as our eigen-base vector u_i.

Now, for the m^{th} block matrix in (5) we have systematic way to loop all factors involved in a given $p_m(\mathbf{A})$ term and identify a unique matrix $\mathbf{T}_i$, an algebraic multiplicity k_i, and a vector u_i with which we can construct a basis for whatever eigenvalue corresponds to what $\mathbf{T}_i$ is in every iteration of the loop. We construct this base, reshuffle and repeat until we've exhausted the list of all polynomial factors $p_m(\mathbf{A})$. At each block we add the eigen-basis column vectors we construct to a new transformation matrix $\mathbf{S}_{eigen}$. After doing so for each block matrix[1] in (5) the end number of columns of $\mathbf{S}_{eigen}$ will be $N = 6$.

Now all we need to do is multiply $S_{eigen} * S_{null}$ to get the final transformation matrix; $\mathbf{S} = \mathbf{S}_{eigen} * \mathbf{S}_{null}$ which is to operate on the original matrix $\mathbf{A}$. The final Jordan-Block transformation looks like,

$$\mathbf{J} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 1 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 \\ 0 & 0 & 0 & 0 & -i & 1 \\ 0 & 0 & 0 & 0 & 0 & -i \end{pmatrix}$$

Notice there is just one diagonal row above the diagonal – often referred to as the *supradiagonal* row. This is a characteristic which can always be expected of Jordan block matrices.

2.6 Some Room for Improvement?

There may exist more efficient means of arriving at the Jordan form. I believe it possible to simply find the eigen-vectors and use them as a basis, but for all the time and energy I put into this project, I never once bothered to follow up on whether there were an easier way.

Experimentally it was found that the order of the eigen-base need to be reversed from $\{u, \mathbf{T}u, ...\mathbf{T}^(k-1)\}$ to $\{textbfT^(k-1)\}, ..., \mathbf{T}u, u\}$, else instead of a supra-diagonal row (elements just above the diagonal) what resulted were elements just *below* the diagonal. Why this is so, I am unsure exactly, but it likely is related to the underlying theory of transformations. For a good start to this see chapter 5 of ref. [4].

Another point which a scrutinizing reader may have pondered is this; how do we justify the fact that the result of a similarity transform is block matrices? For diagonable matrices, there are trivial arguments to justify this. But doing so for the general case I believe takes a much deeper dive into the subject. Again, see chapter 5 for a start to the discussion on representing matrices in different basis. But I fear this might actually require the use of a more daunting subject called Singular value decomposition. On the other hand, one can always just roll up their sleeves and examine the properties of the inverse of the transformation matrix S^{-1}. I considered this, but I know this would run me into co-factor identities – a tedious subject I once dealt with and at the moment do not care to return to.

If nothing else, this project was a good lesson for me; sometimes, its OKAY to use a tool without understanding EVERY detail of it!

[1] Note that the eigen-basis column vectors we pull from each block of length l will of course need to have rows of zeros appended above and below as needed in order for the column positions to match their corresponding vertical height in (5).To this end, by keeping track of the dimensions (or ranks) of the null-spaces which were determined in the previous decomposition we are able to calculate the position and dimension of the current block matrix being dealt with, and so we know exactly how many zero rows need appending to the top and to the bottom.

References

[1] Attila M´at´e. *The Jordan canonical form*. Brooklyn College of the City University of New York, 2014.

[2] Taboga, Marco (2021). *"Range null-space decomposition", Lectures on matrix algebra. https://www.statlect.com/matrix-algebra/range-null-space-decomposition.*.

[3] Taboga, Marco (2021). *"Invariant subspace", Lectures on matrix algebra. https://www.statlect.com/matrix-algebra/invariant-subspace.*.

[4] Hans Schneider and George Philip Barker. *Matrices and Linear Algebra, 2nd ed.*. Dover Publications, New Yorkm 1973.