# **Weekly Report – W3 Spring 2023**

## **Problem & Task**

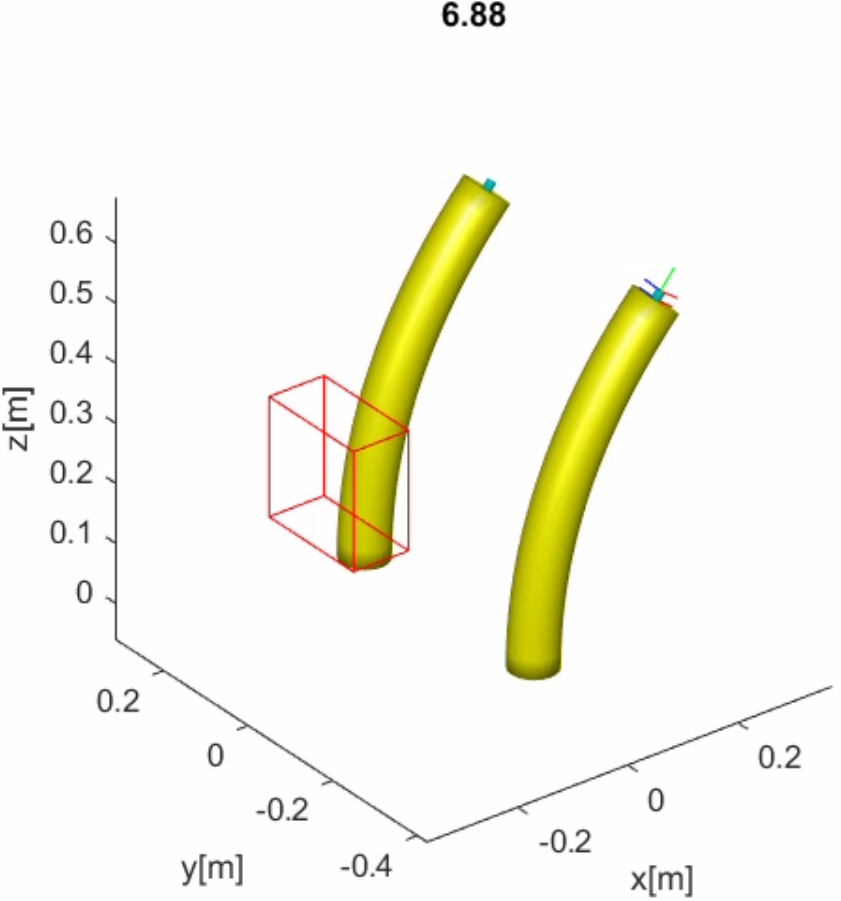
1. About adding another SRA in the existing TMTDyn simulation package;
2. Some other optimizations about TMTDyn package.

## **Solution**

1. Add another SRA in the simulation package

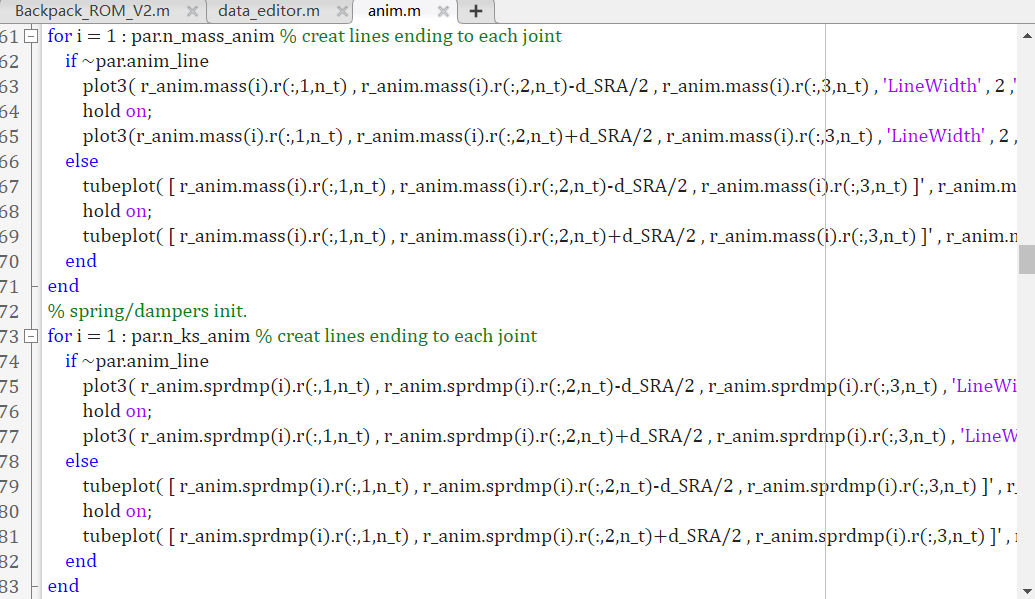
#### (1). Animation itself

Since last week I successfully added a cube in the 3D space in the original simulation package using exactly the same function (which is plot3) as how the SRA was animated (it failed for patch and fill function), so I guess the basic principles should be the same though there are more input values for the SRA structuring. The fundamental principle of plot3 function is to plot different coordinates into a same frame, and the input stores the position information, in this case we should be able to achieve this goal.



**Fig. W3-1** The schematic of 2 arms added into the animation frame

By adding another parameter d\_SRA which denotes the distance between the two arms, we can adjust their position relative to the origin, and the example codes are shown in the following figure.

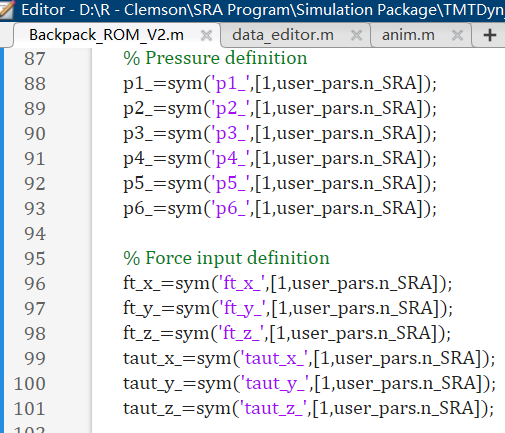


**Fig. W3-2** Example of modification towards the source code

#### (2). Change the size of input variables

In step 1, what we did is just to directly modify the code and duplicate another arm which has exactly the same properties as arm 1 in *anim.m*, the input of this function is actually coming from the first SRA, that is to say, we firstly need to make up another set of data for arm 2 in *data\_editor.m* which stores all the input including actuation pressure/torque and external force/pressure/torque and physical experimental data recording the tip position changes of the SRA (the file was named in Version 002, which was originally *sample\_exp\_data.m*); initially I assembled all the input values into a single matrix, accordingly we have to change the functions in which *data\_editor.m* has been called, and considering that in the future the case would become more complex with more SRAs, we need to come up with a more general way to set up new input variables in the main function (which is *Backpack\_ROM\_V2.m*, its old version name is *exp4\_ROM.m*), I brought in a new variable *n\_SRA* which denote the number of SRAs (or bodies by the definition in the code) for the simulation under the struct *user\_pars* so that when the struct was written into *mex* file, the newly built variable can be passed to any other functions called.

To distinguish the inputs of arm 2 from arm 1, symbolic arrays can be set up in the figure below just like what I did in my own simulation package.



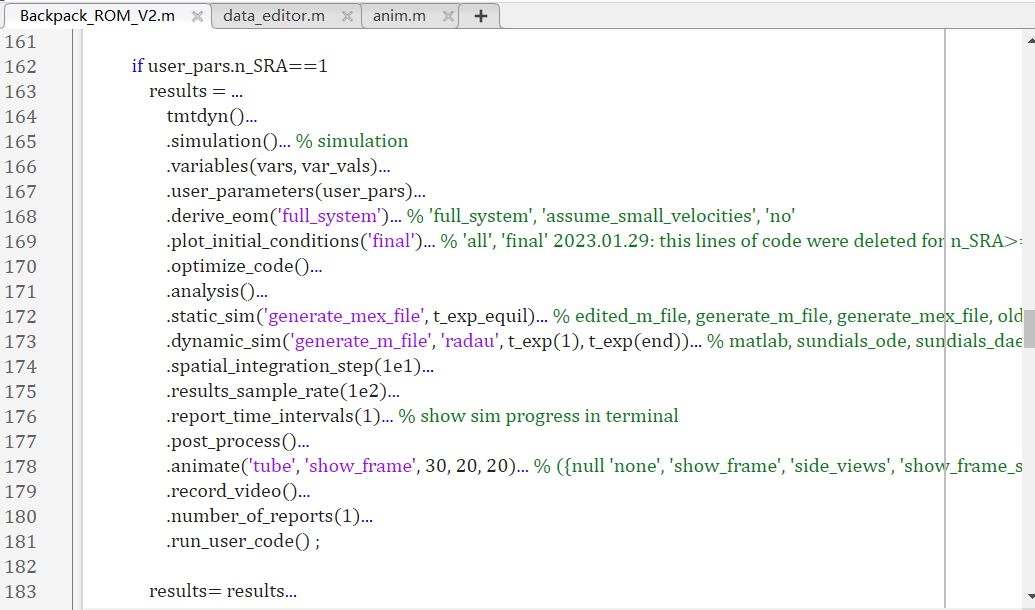
**Fig. W3-3** The example of input setting up using symbolic variables for multiple arms

The advantage of this formatting is that it is convenient to use in a for loop, and the logic for EOMs derivation is the same as mine, using symbolic variables to derive an expression, then passing the specific values to them to trigger the ODE solvers. The assembled variable array will be 12 greater for each SRA increased in terms of the size, and I realized another issue, how do other functions recognize what kind of inputs they are going to accept? That means I have to modify all the input and output format in related functions and redefine them, which will be a huge project. However, what’s more, the side effect of this method is it will take up a lot of memory allocated since all the parameters were predefined before they enter the real simulation process, for example, for deriving EOMs of SRA 1, even though some of the parameters were not used during that process, which belong to SRA 2, extra memory still will be allocated to them, which slows the simulation process to some extent. The good thing is I haven’t found any influence of these redundant parameters on the final results.

Thus I decided to change the size of symbolic variable array into its original size, and here comes another vital question: because it has been proved that for loop must be used for multiple SRAs’ simulation, so where should the for loop start?

#### (3). The position of for loop start in the main function

Initially I thought the lines of code in the figure below are executed in a certain sequence, they do, but they are more like bound with each other, the *results* is written in struct, which broken down into 3 sections just to avoid that earlier version MATLAB could not cope with such complicated command. As we can see for the first section of *results* which is followed by two bodies’ definition (the SRA itself and a force sensor located at the tip of the SRA), all the related symbolic variables and specific values are sent to the EOM derivation function, in our case, the two arms are exactly the same, which means that I do not need to derive the EOM twice, for the following two sections, because the input force might be different, I started the for loop between section 1 and



**Fig. W3-4** The example of functions called in the format of struct

2, but I found that for each new body established (which has no attachment to others), section 1 will be executed once more, the animation of a single arm will be displayed along with its tip position change, obviously this is not what we want, we would like both arms to be animated in the same frame; moreover, the former arm’s results will be over written by the latter one, I will leave this problem in chapter (4). results processing.

I have tried a lot of methods, and eventually I decided to move the beginning of the for loop to the very front of the main function, just before every parameter was defined and meanwhile I placed the iteration count *i* under the struct *user\_pars* as well so that the subsequent functions called should be able to recognize which inputs they should use, and deleted the *prot\_process* and *animate* because we need the results of two arms plotted in a single frame.

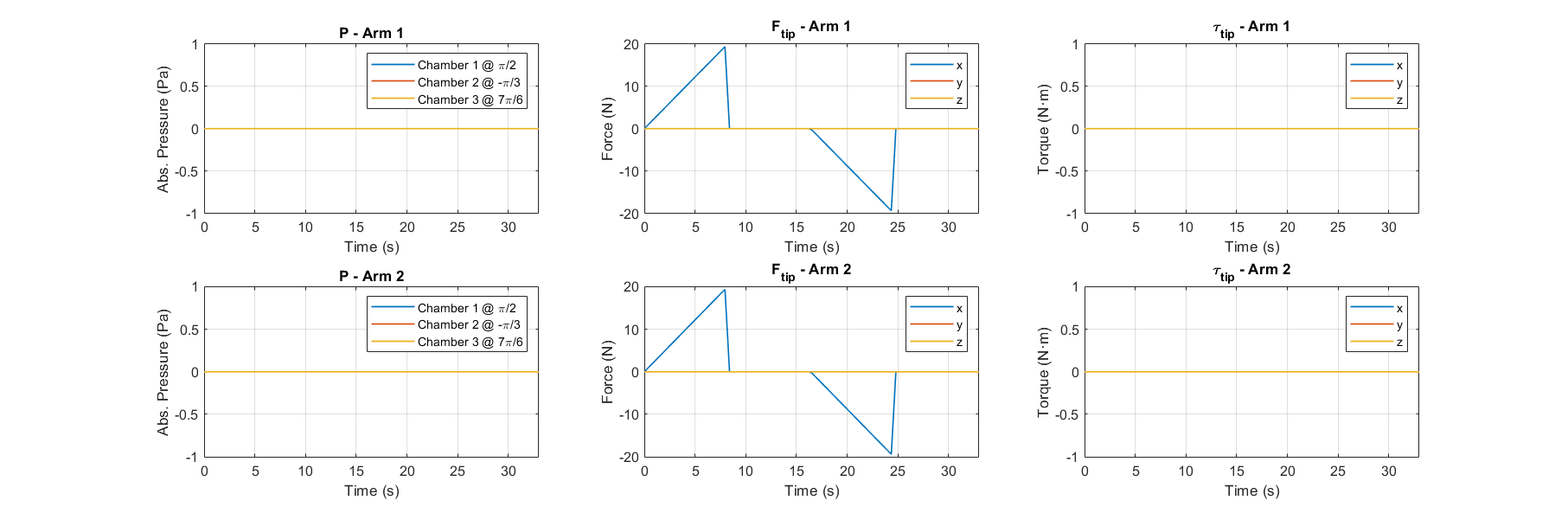
**Limitations:**

If we choose to use for loop to complete the EOMs derivation of both arms, we need to switch the .derive\_eom(‘no’) to .derive\_eom(‘full\_system’) because the properties of the arms might be different in the future, we need to over write the governing equations for the former one; recall that for very large shear force, there will be some special issues with the EOMs themselves (the sqrt() problem, we have to add another complex() inside the bracket), we cannot save it because in the next iteration the second arm will still use the EOMs we just saved, which is apparently not correct. And for EOMs of different arms, we cannot save them into separated files currently, the only solution when facing very large shear force is to add a “pause” before each arm’s simulation is done and export the results individually.

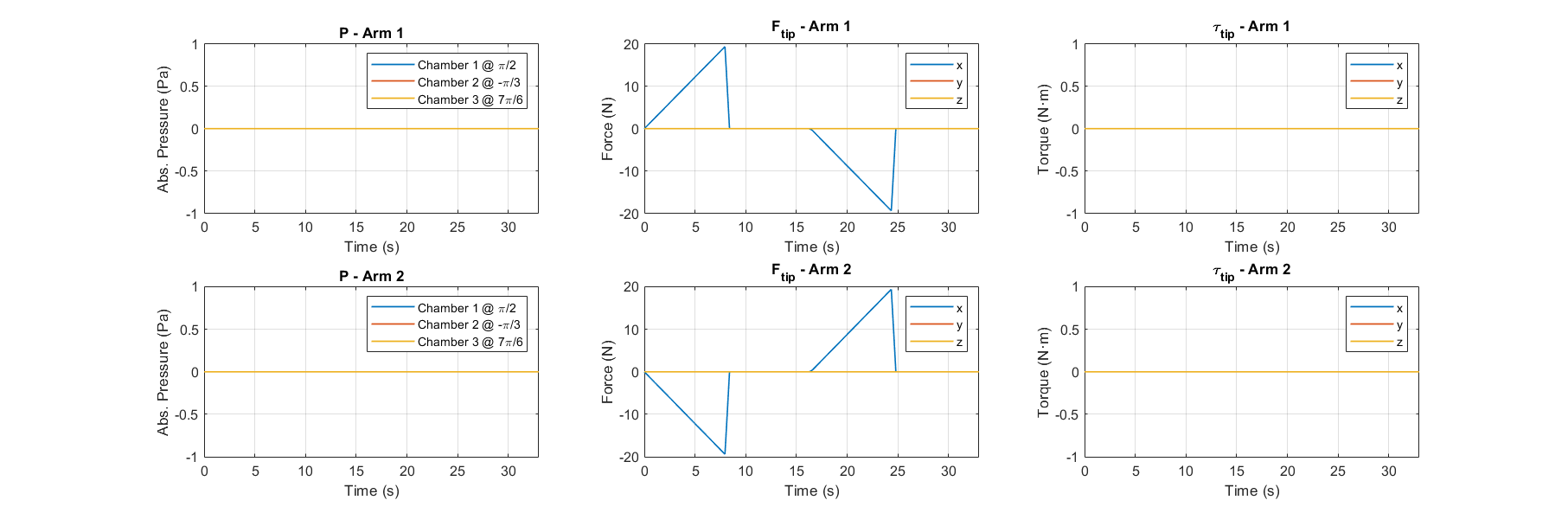
#### (4). Results processing

To cope with the issue that the former results would be over written by the latter one, I would bring in another struct only for section 3 of the broken down *results* struct, such as *intm(i).results* instead of *results=results…*, since *results* contains so many struct, it can not be written into a double array or matrix, certainly we can do it manually by using bracket [ ], however the code will lose property of autonomy, my solution is to write the results of different SRAs into a cell. To assess if the input was truly applied, my test plan was arranged as follows.

Similar as what I have done in the past few weeks, I can apply a gradually increased force exerted on the tip along a certain axis, when it achieves the maximum value, it will drop down to zero suddenly, then do the same thing for the opposite direction. The first test is to apply exactly the same force input to both arms no matter for the magnitude or direction; the second one is for arm 1, the input will be the same as test 1, but for arm 2, the force direction will be opposite as shown in the figures below.

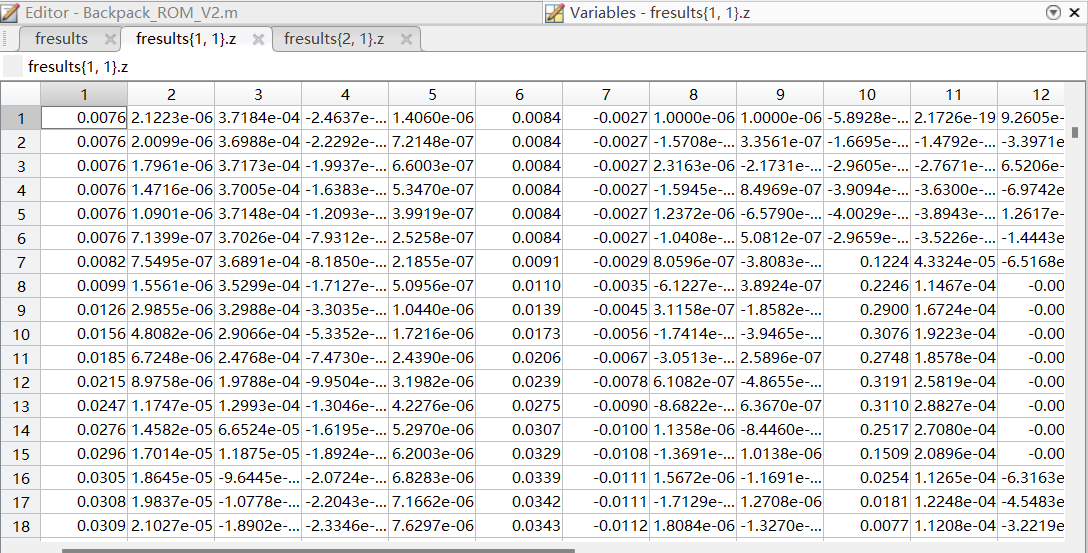


**Fig. W3-5** The input force for two arms in test 1

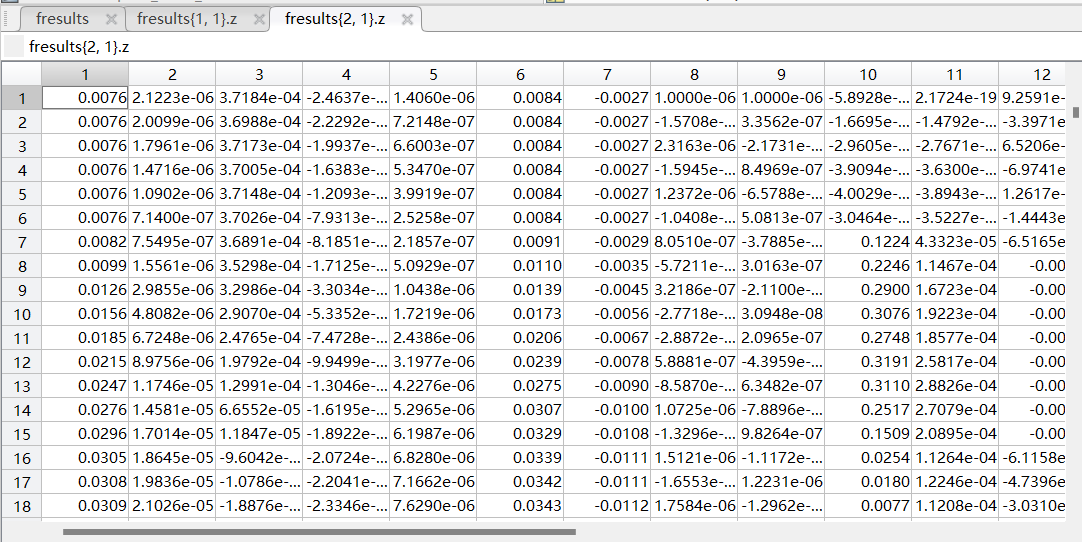


**Fig. W3-6** The input force for two arms in test 2

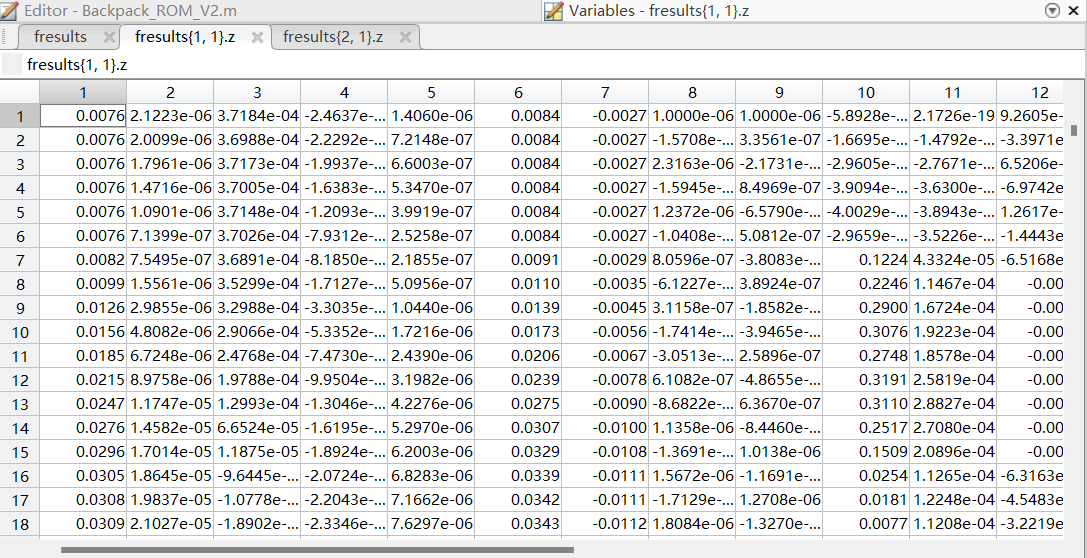
And accordingly the results are shown as follows.



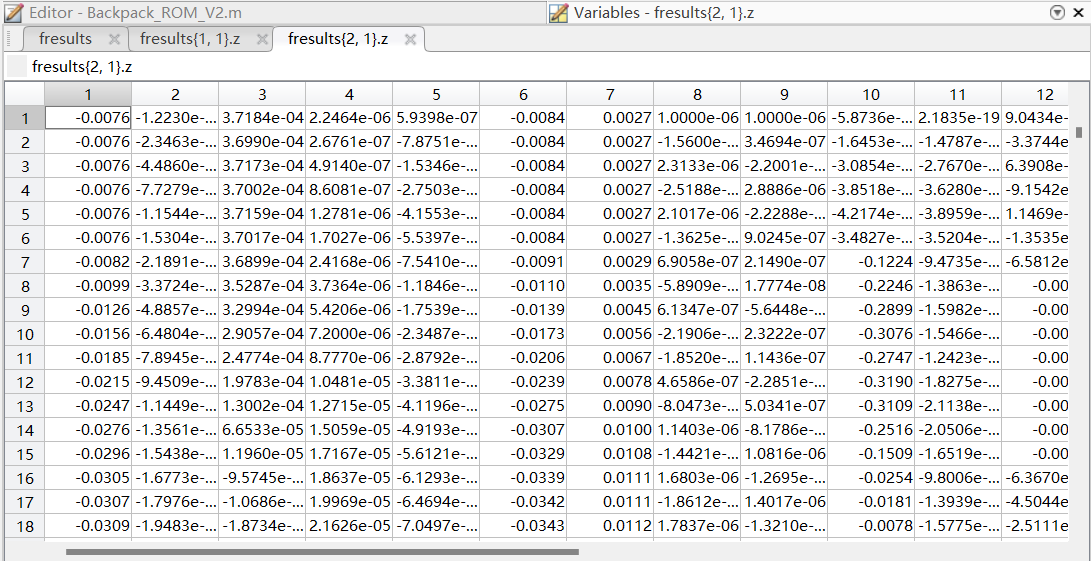
**Fig. W3-7** The results for arm 1 in test 1



**Fig. W3-8** The results for arm 2 in test 1



**Fig. W3-9** The results for arm 1 in test 2



**Fig. W3-10** The results for arm 2 in test 2

From the figures above, we can find that in test 1, though we cannot say that the results for both arms are 100% the same, they were extremely close, which can be seen as equivalent, the difference might be due to the variable step size for implicit ODE solver *ode15s*; for test 2, we can see that the results are totally different, which indicates that the input was truly passed to each arm, but I still need to see the animation to give a final conclusion about it.

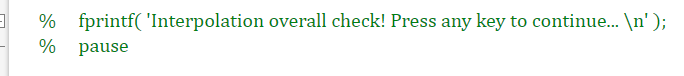
Apart from this, I will restate again, since all the results of both arms need to be plotted in the same frame, we cannot call the *anim.m* or *post\_process.m* functions for twice, the combination of all the results is a must no matter for what kind of the data class. So here, for example, for *anim.m* function, I added an if-else structure at the most outer layer of the original function to judge if the input is a struct or a cell, I’m still working on it and it will be another huge project.

1. Other optimizations about TMTDyn package

#### (1). Block the initial conditions plot

Specially for ROM, the code was designed to plot the initial conditions (which is unnecessary) for each new body establishment, and the simulation will pause after deriving EOMs of each body, we need to press any key to continue, which is quite inconvenient.

My solution was to find the related source code in tmtdyn\tmt\_eom.derive.m, section of “plot initial conditions and fitted splines” and comment out the following lines of code.



**Fig. W3-11** The commented out codes in *tmt\_eom.derive.m* to accelerate the simulation process

#### (2). Delete meaningless blank figure plotted

No matter for which modelling method we use in the simulation package, and no matter for calling the animation function or not, there will always be a blank figure window forged, I know what it is used for, its frame size is calculated by how far the SRA would be extended and the size will be determined by the maximum extension and then be used for the final frame of animation, but there is no reason for its appearance when the animation switch is off.

I haven’t found any codes directly related to it, and I’m not sure if there is any other influence to the animation if I comment out some of the related codes if possible, my solution was to add a *close* function at the end of *results* struct to close the current plot window, it has been verified to be success.

#### (3). Animation for multiple arms

Since for multiple arms, we don’t want the animation to be present in the for loop I set up in the main function, meanwhile the *anim.m* function does be called even though the animation switch is off, so for struct input of *anim.m*, we can just copy the first few lines of code inside and paste them into the case for multiple SRAs ignoring the rest ones just to guarantee the simulation will run smoothly, and this also has been tested to be success.

## **Plan**

1. Deal with the animation function if I have enough time;
2. Figure out which pin corresponds to which pneumatic chamber for the backpack as soon as possible; if so, modify the source code for physical test.