

GDBP Technical Report: Compliance Study of a Formula Student Car

Department of Mechanical Engineering

Word Count: 9,756

Candidate Number: 11415

May 12, 2026

Faculty of
Engineering & Design



UNIVERSITY OF
BATH



Contents

1	Acknowledgements	4
2	Abstract	5
3	Introduction	6
4	Design Specification	8
5	Data Pipeline Construction	9
5.1	Comparison of FEA and MATLAB Non-Linear Solvers	9
5.1.1	Operation of an FEA solver	9
5.1.2	Operation of MATLAB's fsolve (trust-region dogleg method)	12
5.1.3	Hybrid Approach	13
5.2	Final Pipeline	14
6	FEA Work	15
6.1	Mesh and Geometry preparation	15
6.2	Element type selection	16
6.3	Materials and Composite layup	18
6.4	Contacts and Constraints	21
6.5	Outcome of the full-assembly model	22
7	MATLAB Solver	27
7.1	Geometric framework	27
7.2	Rocker local coordinate system	28
7.3	Construction of the local-to-global rotation matrix	30
7.4	Camber and Toe relative to the upright	31
7.5	FEA map import and interpolation	33
7.6	User input and the staged solve	34
7.7	Residual function and equation count	37
8	Results	38
8.1	Validation	38
8.2	Compliance through suspension travel	40
8.3	Lateral load cases	44
8.4	Sensitivity to compliance and material choice	48

9 Solution Lifecycle	54
10 Future Work	54
11 Conclusion	55
A MATLAB Code	59
A.1 Main Compliance Solver Script	59
B Python Code	85
B.1 Ansys Load Sweep Script	85

1 Acknowledgements

This project was undertaken with Team Bath Racing Electric as the primary clients and they provided support throughout the project giving access to extensive car data and feedback on proposed avenues of design.

This report should also be read in conjunction with the rest of the team's technical reports as much of the work is interlinked. Particularly for this report VD-01 & VD-05 provided important information to facilitate the analysis carried out.

It should also be recognised that generative AI has been used for brainstorming, research and to co-generate some code in accordance with the university guidelines but all work presented as my own within this report is as such.

2 Abstract

In this project a hybrid kinematics & compliance solving software was produced which combined the strengths of both FEA matrix methods and the trust-region dogleg non-linear optimisation technique to reduce model run time from over 20 hours to approximately 100 minutes. At the same time it was found that out of camber and toe, toe is approximately 4 times more sensitive to changes in compliance than camber whilst vehicle lap time is approximately 5 times more sensitive to changes in toe than camber. A potential improvement to the compliance of the suspension was proposed which despite increasing the car's mass was predicted to reduce the lap time at Formula Student Germany (FSG) Endurance by almost five thousandths of a second. The groundwork was also laid to expand the software into a comprehensive kinematics design tool in the future.

3 Introduction

“If you broke down everything you could think of that goes into [cycling], and then improved it by 1%, you will get a significant increase when you put them all together [1].” Sir Dave Brailsford’s mantra for Team Sky has since been adopted across motorsport, with Toto Wolff describing Formula 1 as “all about marginal gains [2].” Many factors that look negligible in isolation become substantial in combination, and suspension compliance is a clear example of this principle in race car design.

In Formula Student (FS), the mechanical design of suspension typically follows a fixed sequence. The architecture is settled first, normally a double wishbone arrangement, as this offers far greater control over tyre attitude through bump and steer than alternatives such as MacPherson struts or trailing arms. The double wishbone setup used on TBrE25 is shown in figure 1.



Figure 1: Double wishbone set-up on TBrE25

Kinematic analysis then sets targets for camber and toe variation through wheel travel, which fixes the suspension geometry. The dynamic components, the springs and dampers, are then selected to avoid exciting natural frequencies and to optimise transient handling.

Throughout the kinematic and dynamic design stages the suspension members are treated as infinitely stiff: their lengths and dimensions are assumed constant regardless of load. Once the geometry is finalised, finite element analysis (FEA) is used to determine the materials and cross-sections required to ensure that no member yields during its expected loading cycle. Because the materials commonly used in suspension members, aluminium alloys and carbon fibre composites, both have a high Young’s modulus, the FEA generally shows that members deflect by only a small amount under typical loads. It is here, however, that the marginal-

gains argument applies most directly. Deflections of a couple of millimetres in each member can stack up to produce significant geometry changes, such as undesirable positive camber under hard cornering when negative camber was intended.

The same argument applies to suspension dynamics. Claude Rouelle frequently emphasises the springs-in-series principle: “A soft spring in series with a stiff spring is still a pretty soft spring. There is no point in having a very stiff chassis and compliant suspension wishbones [3].” And likewise, a stiff corner spring combined with compliant wishbones gives an installed stiffness considerably lower than the theoretical value.

Compliance is rarely ignored because teams are unaware of it; it is more often left out because it is difficult to quantify [4]. The deflection of a suspension member under load alters the load distribution within the assembly, which in turn alters the deflection. The problem is therefore inherently non-linear and requires an iterative solver to address.

The work presented in this report develops a pipeline that combines FEA with a custom non-linear iterative solver to quantify suspension compliance. Each suspension component is simulated individually in Ansys to generate a map of force against deflection for a range of loading conditions. These maps are then passed to a MATLAB-based non-linear solver that combines the components and resolves the forces, yielding the change in kinematic properties of the assembled suspension under load. The pipeline has been built to be modular, so that individual components can be exchanged without re-running the FEA on the entire assembly, saving significant time and computational resources for the team in subsequent design iterations.

Within the wider Team Bath Racing electric (TBRe) vehicle dynamics package, the compliance pipeline has a clearly defined role. Report VD-01 supplies the kinematic targets, around which the compliance solver is run to determine the range of installed kinematics actually seen on the car. These ranges are then passed to the full-vehicle model developed in report VD-05, which performs a sensitivity study on the lap time impact of these kinematic deviations. The results are fed back so that geometry, materials and component dimensions can be revised, closing the design loop laid out in figure 2.

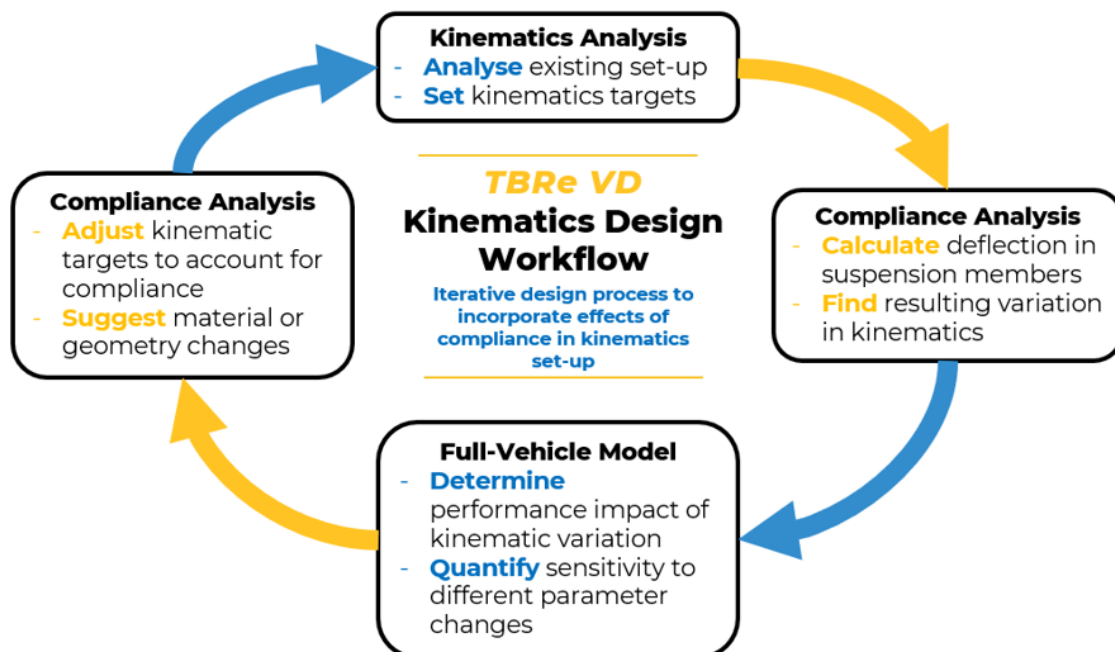


Figure 2: Proposed kinematics design workflow

This combined analysis also enables the trade-off between mass and compliance to be quantified. Suspension design is not strictly about minimising compliance: provided the compliance is known, it can be incorporated into the design so that there are no surprises at the track. Increasing stiffness to reduce compliance generally increases mass, which is detrimental to performance. Working with the rest of the team, the pipeline therefore makes it possible to identify an optimal balance between mass and stiffness to achieve peak performance.

The final part of the report considers how the theoretical results can be validated in physical testing, since simulation results are of limited use without correlation against measurement. The industry standard for this type of work is a Kinematics and Compliance (K&C) rig, but at a cost in the order of millions of pounds these are not realistic for FS-level teams [5]. Lower-cost alternatives suitable for a student team are therefore proposed.

4 Design Specification

The technical task addressed in this report was to develop a tool capable of quantifying the compliance of the suspension system using purely digital simulation and analysis. The tool was required to provide an early indication of car performance and the impact of compliance on suspension geometry, so that, in conjunction with the kinematic design and full-vehicle modelling work documented elsewhere in the team's reports, an iterative design process could be supported through early validation and testing.

The first constraint of the project was the choice of software environment. The tool was required to take inputs from Ansys and provide its interface and outputs in MATLAB, since these are the two software packages used routinely by the TBRe team. Restricting the toolchain to these packages ensured that all team members would have access to the required toolboxes, and that the tool would persist as a usable resource for the team in subsequent years.

A second major requirement was to reduce the runtime and computational load associated with running the analysis directly in Ansys. The analysis of a mechanism involves multiple non-linearities and is not the primary use case for a software package such as Ansys, with the result that convergence on a reasonable solution can take a very long time on the full assembly. The target was therefore to produce a more modular approach where if a single component is changed, only that component needs to be updated in FEA, rather than the entire assembly. This would facilitate much more rapid iteration in the early design phase.

The third specification was that the tool needed to be usable by team members who were not experts in mechanical design or FEA. Its primary purpose is as a vehicle dynamics (VD) design aid, and so the end user should not be required to interact with Ansys directly. Maintaining a separation between the end user and Ansys would also allow FEA and VD workflows to operate in parallel, with simulation engineers developing the latest models and the VD team only needing to pull the updates in at a convenient moment.

Beyond the data flow, the tool was required to produce charts showing the deflection of individual members, in order to identify the dominant sources of compliance, together with toe and camber sweeps that could be compared with those generated by the kinematic design and passed on to the full-vehicle simulation. This gave the final set of specifications listed in table 1.

Table 1: Design specifications for the compliance analysis tool

No.	Specification
1	Material and geometry data input through Ansys
2	Output results in MATLAB
3	Reduce computational time compared to running whole assembly in Ansys
4	Modular arrangement so individual components can be changed without having to re-run all FEA
5	No direct interaction with Ansys for end user
6	FEA and processing workflows can run independently
7	Graphs produced for deflection of all components along with sweeps of toe/camber through bump

The development of the tool itself, and the specific design decisions made during implementation, are detailed in the sections that follow.

5 Data Pipeline Construction

5.1 Comparison of FEA and MATLAB Non-Linear Solvers

With the design specification fixed, the structure of the tool itself had to be decided. The principal obstacle to reducing run time was the FEA's difficulty converging on large mechanisms, which raised the question of whether a different class of solver might be better suited to the problem. Although general-purpose FEA solvers and MATLAB's non-linear optimisation routines are often grouped together as "non-linear solvers," they are designed for different tasks and behave very differently in practice. Understanding these algorithmic differences is necessary to justify the structure of the pipeline, hence the following section lays out their comparative functionality.

5.1.1 Operation of an FEA solver

In a typical FEA solver such as Ansys, the structure is approximated by a large number of elements connected at nodes. For each element a local stiffness matrix is constructed from the material properties and element geometry, capturing the relationship between nodal forces and nodal displacements [6]. Hooke's law in matrix form, force equal to stiffness multiplied by displacement, gives equations 1 & 2.

$$[K] \times \{D\} = \{P\} \quad (1)$$

$$[\text{stiffness matrix}] \times \{\text{displacement vector}\} = \{\text{load vector}\}$$

$$\begin{Bmatrix} p_{x1} \\ p_{x2} \end{Bmatrix} = \begin{bmatrix} AE/l & -AE/l \\ -AE/l & AE/l \end{bmatrix} \begin{Bmatrix} \delta_{x1} \\ \delta_{x2} \end{Bmatrix} \quad (2)$$

where:

p = force

A = cross-sectional area

E = Young's modulus

l = member length

δ = displacement

The local stiffness matrices are transformed into the global coordinate system using rotation matrices and assembled into a global stiffness matrix that captures the connectivity of the entire mesh. The angles between the local and global coordinate system are shown in figure 3 and the resulting rotation matrices equation given in equations 3 & 4.

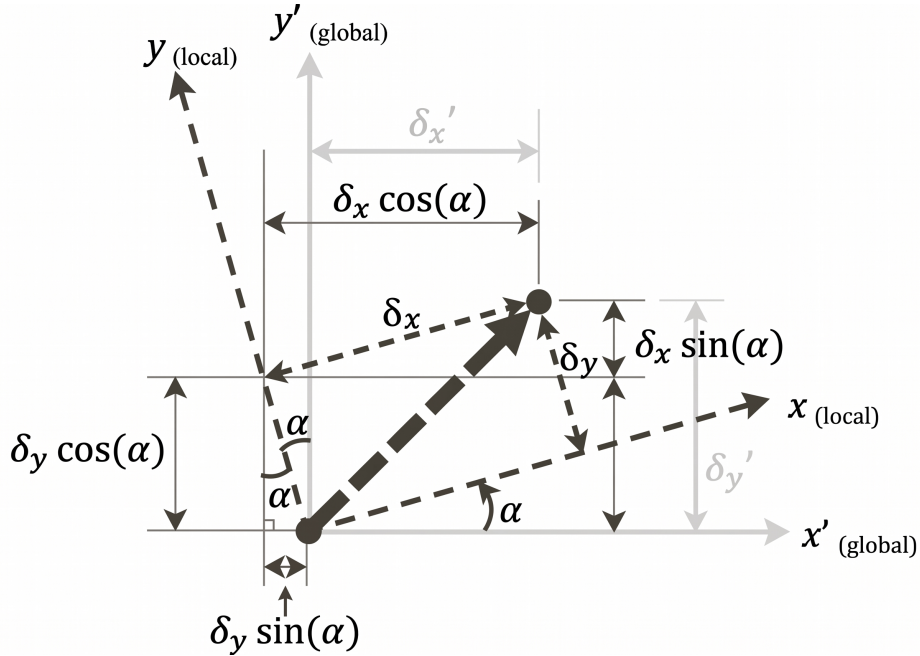


Figure 3: Transforming from local to global coordinates system [6]

$$\begin{bmatrix} \delta'_x \\ \delta'_y \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} \quad (3)$$

$$\begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} \delta'_x \\ \delta'_y \end{bmatrix} \quad (4)$$

Three equations per node are required, corresponding to the translational degrees of freedom in x , y and z , and sufficient boundary conditions must be supplied to render the system solvable. For a linear problem, Gaussian elimination on the global stiffness matrix forms an upper triangular matrix as shown in equation 5 which can be solved for the unknown nodal forces and displacements directly.

$$\frac{EA}{l} \begin{bmatrix} 2 & 0 & 1/2 \\ 0 & 2 & -1/2 \\ 0 & 0 & 1/4 \end{bmatrix} \begin{Bmatrix} \delta'_{x1} \\ \delta'_{y1} \\ \delta'_{y2} \end{Bmatrix} = \begin{Bmatrix} -W \\ 0 \\ -\frac{3W}{4} \end{Bmatrix} \quad (5)$$

where:

W = arbitrary load

Where the system is non-linear, the global stiffness matrix changes with displacement and the solver must iterate. The local stiffness matrix is generally constant within the elastic region of the material, but the global stiffness matrix becomes non-linear once components undergo rigid-body motion (in which whole components pivot or translate without internal deformation) because the relative orientation of components in the global coordinate system changes continually. Hence the rotation matrix given in 3 must be re-defined at each iteration. Ansys handles an iterative solve using the Newton–Raphson method, making a prediction about the equilibrium point based on the current stiffness matrix [7]. This method can risk divergence in highly non-linear scenarios as shown in figure 4.

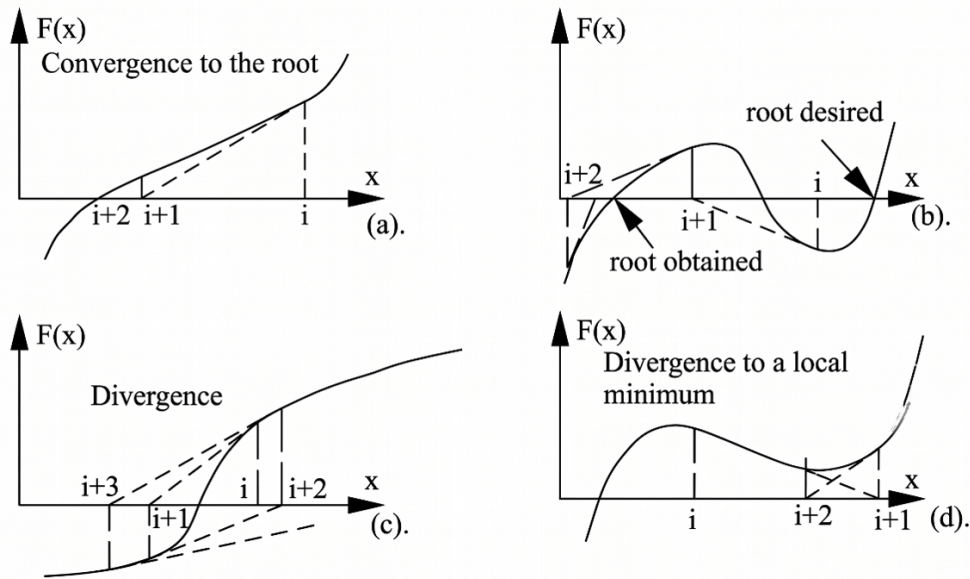


Figure 4: Potential Newton-Raphson method responses to different function shapes [7]

A line-search algorithm therefore scales back the step length, typically by a factor of 0.5, if the predicted step fails to reduce the residual in an effort to avoid this divergence. The starting point of any FEA analysis is the unloaded CAD model, which by construction is geometrically valid, and so iterations from this state are likely to remain feasible provided the load is applied gradually.

5.1.2 Operation of MATLAB's fsolve (trust-region dogleg method)

MATLAB has an inbuilt function for solving general non-linear problems - called fsolve - and its default algorithm is the trust-region dogleg method [8]. As an optimisation function rather than a root finding routine the program aims to minimise the sum of the square of the residuals. From the initial guess of unknowns, the solver probes small increments in each degree of freedom and uses finite differences to construct the Jacobian matrix shown in equation 6, which contains the partial derivatives of the residuals with respect to the each of the unknowns [9].

$$\mathbf{J}_f(x, y) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} \quad (6)$$

Next the Hessian (the matrix of second partial derivatives of the residuals) is approximated using the Gauss-Newton approximation given in equation 7.

$$\mathbf{H} \approx \mathbf{J}\mathbf{J}^T \quad (7)$$

A local minimum is found by locating a point where the gradient in all directions is 0. The Hessian describes how the gradient changes with position. The gradient at a nearby point is therefore approximately the current gradient plus the Hessian multiplied by a displacement. The current gradient of the residuals is defined using the chain rule and given in equation 8.

$$\nabla S = \mathbf{J}^T \mathbf{r} \quad (8)$$

where:

$\mathbf{r}$ = vector of the residuals

Therefore, combining equations 7 & 8 and setting to zero gives a displacement to move to the Gauss-Newton (stationary) point as shown in equation 9.

$$\mathbf{H}\delta = -\nabla S \quad (9)$$

If the Gauss-Newton step lies inside the current trust region the solver accepts it. The size of the trust region is updated based on the agreement between the predicted reduction in residual and the actual reduction observed in the previous step: if the residual decreased significantly less than expected, the trust region is contracted; if the prediction was accurate, it is expanded. If the Gauss-Newton point lies outside the trust region, the solver finds the Cauchy point, which is the minimum of the square residual sum along the steepest-descent direction within the trust region – determined from the Jacobian. The dogleg path interpolates between these two directions as shown in figure 5.

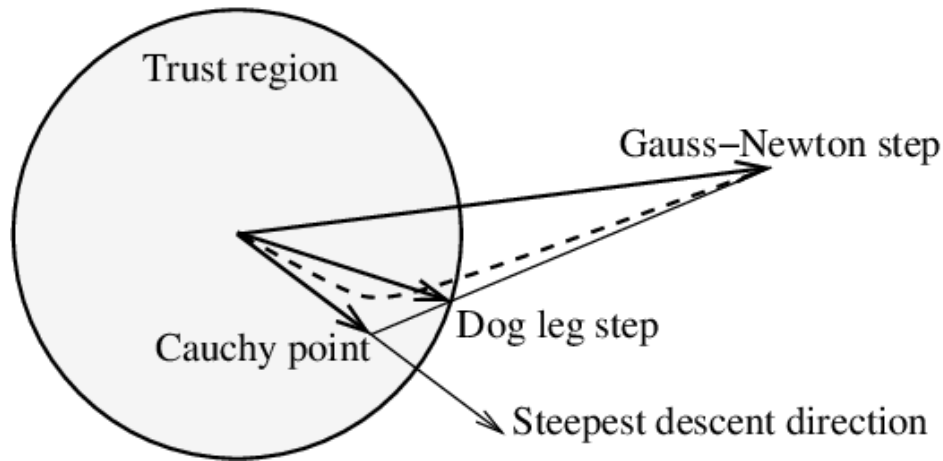


Figure 5: Graphical illustration of the trust-region-dogleg method [10]

Compared with Newton–Raphson plus line search, the trust-region dogleg is appreciably more stable. In a residual surface with large changes in gradient, the Newton–Raphson method can become divergent as shown in figure 4, whereas the trust-region dogleg always converges to a local minimum. It should be noted that this guarantee is local rather than global; in the present application the suspension system has effectively a single physical equilibrium for a given loading, so this is not problematic, but the implications for global optimisation are discussed in the future work section.

5.1.3 Hybrid Approach

Given the stability advantages of the trust-region method, it is reasonable to ask why FEA solvers do not use it as standard. The answer lies in scale. The Jacobian of an FEA problem is highly sparse, since each element has stiffness in only a few directions and connects to only a few other elements [11]. FEA solvers are written to exploit this sparsity, and use matrix decomposition techniques that preserve it during row and column operations. In the trust-region dogleg algorithm, however, the Hessian is approximated as the Jacobian multiplied by its transpose, an operation that fills in many of the previously zero entries. Even a small percentage of fill-in causes a large increase in memory usage when the Jacobian contains millions of entries, and the algorithm therefore becomes infeasible on full-scale FEA problems.

FEA solvers also build the global stiffness matrix from known local stiffness matrices which are simply a function of each component’s material properties and geometry, whereas fsolve has no inherent knowledge of the structure it is working on. Instead it must probe each direction with finite differences to construct the Jacobian numerically, which involves many additional function evaluations. For a small system this overhead is negligible; for a million-element model it would be prohibitive.

These observations explain why a combined approach would be effective. FEA is well suited to the analysis of large numbers of small elements moving in predictable ways, where the sparse-matrix functionality delivers rapid calculations. The trust-region dogleg method is well suited to small numbers of larger components moving along non-linear trajectories, such as the rigid-body arcs traced by a suspension assembly, due to its increased stability.

5.2 Final Pipeline

As described above MATLAB's fsolve function and the processes in Ansys have different strengths and as such the data pipeline was defined to utilise these whilst meeting the design specifications in table 1. The schematic given in figure 6 lays out the final pipeline.

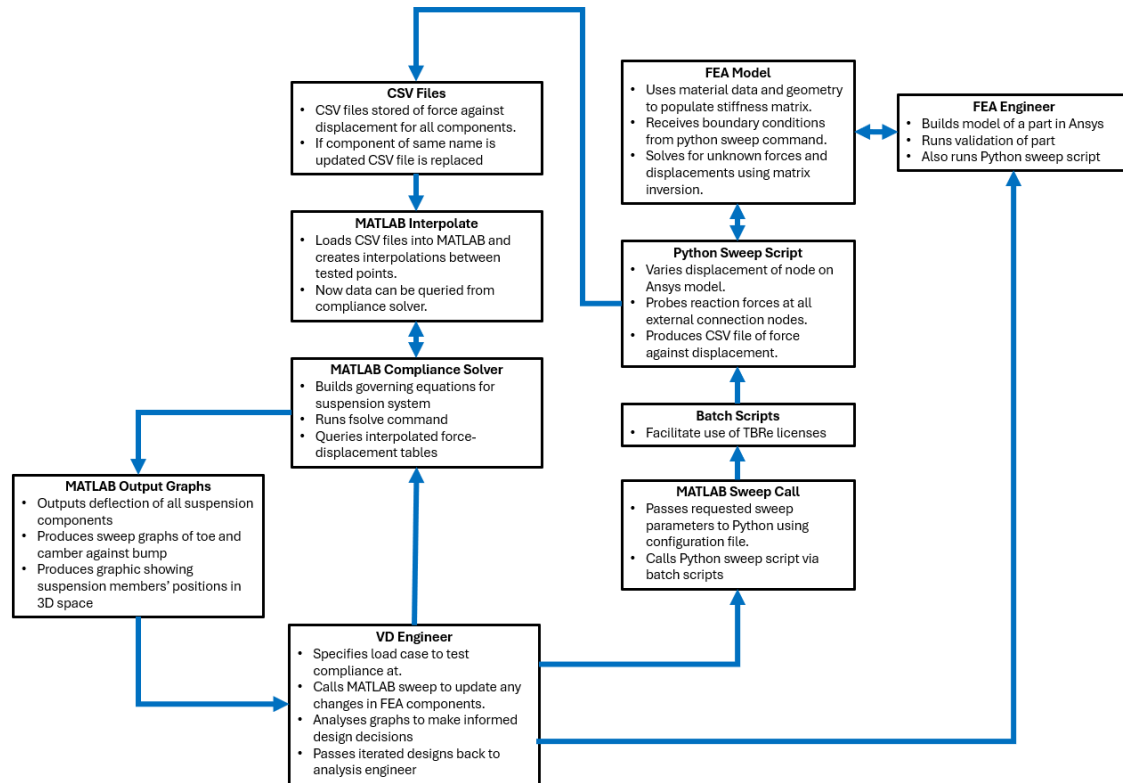


Figure 6: Final structure decided upon for analysis pipeline

The FEA solver takes care of the high element count linear analysis of each component whilst the MATLAB program combines each component and solves the non-linear motion when the suspension moves as a mechanism.

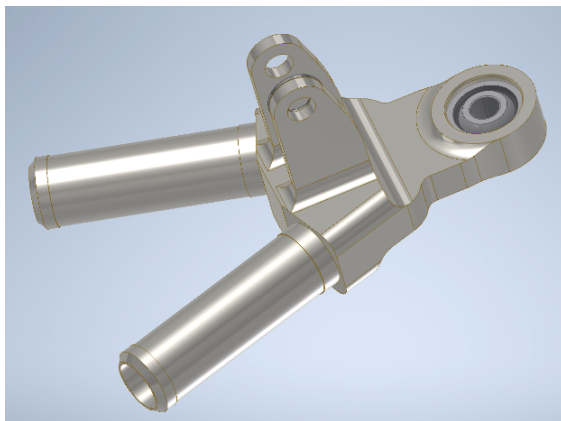
By adopting this workflow it clearly split up the work of the FEA engineer in Ansys and the VD engineer in MATLAB meaning that the specification for parallel workflow is met. Also, by dividing up the work according to the two software's strengths it is likely to result in a significantly reduced computational time. If the workflow detailed in figure 6 can be executed it should satisfy all criteria from table 1.

6 FEA Work

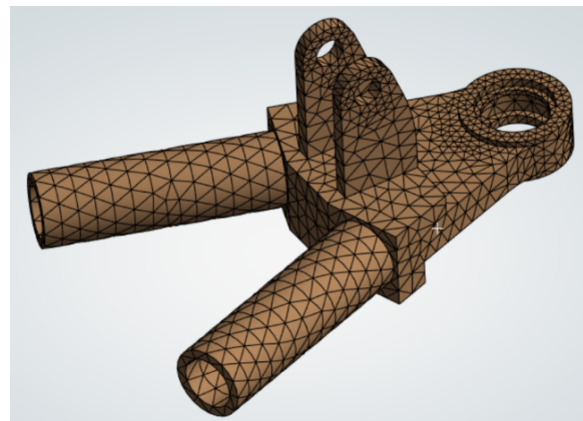
Having settled on a structure for the final data pipeline it was decided to set a benchmark by running an analysis of the entire suspension in Ansys to serve as a reference against which the modular pipeline could be compared. As described in this section, the full-assembly model proved infeasible to run within the project timeframe, but the work undertaken in setting it up was still essential for running the sweeps on individual components.

6.1 Mesh and Geometry preparation

Since FEA mesh density trades accuracy against computational cost – and small CAD features such as chamfers, fillets and pockets consume disproportionate mesh resources for little gain in fidelity – time was taken to de-feature the components in CAD, removing small geometries that were either off the load path or too small to influence the overall response [12]. This process can clearly be seen between the lower wishbone joint in figure 7a and the corresponding part de-featured in Ansys in figure 7b.



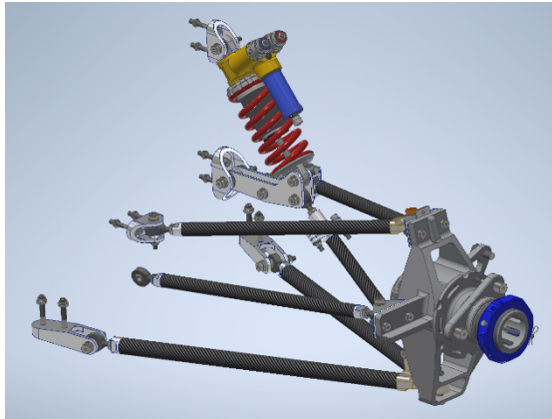
(a) CAD



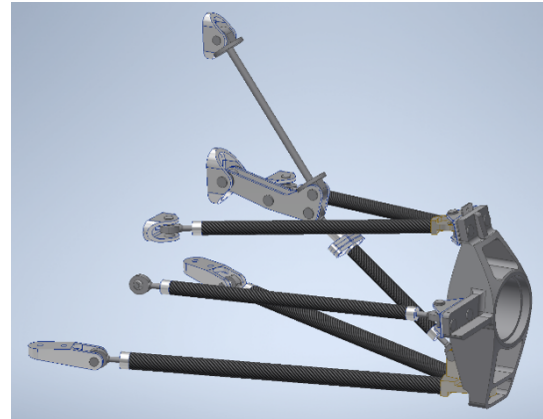
(b) Ansys

Figure 7: Lower wishbone joint in CAD and then after de-featuring and meshing in Ansys

The TBRe team supplied the CAD geometry for the 2026 car currently in build; however on inspection, several components did not align to a sufficient tolerance to allow contacts to be built in FEA. Minor adjustments were made to ensure the assembly could be processed by Ansys at the same time as defeaturing. The original CAD and that passed to Ansys can be seen in figures 8a & 8b.



(a) Full Suspension

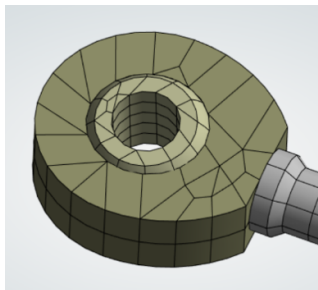


(b) Defeatured Suspension

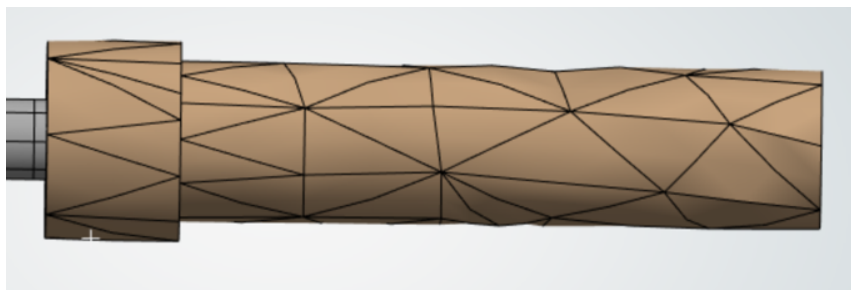
Figure 8: Full front left corner from TBRe team before and after defeaturing

6.2 Element type selection

Once defeatured, the components were meshed in Ansys. Automatic mesh generation in Ansys tends to default to tetrahedral elements, which, as the simplest 3D element type, can be fitted to almost any geometry [13]. Hexahedral elements are more restrictive: distorted angles must be avoided, since a concave corner can drive the determinant of the Jacobian matrix to zero or below, producing a singularity that prevents the element from being solved [14]. The singularities that a hex mesh risks producing can be seen on figure 9a and figure 9b shows a coarse tetrahedral mesh automatically fitted by Ansys.



(a) Singularities in hex mesh



(b) Coarse tetrahedral mesh

Figure 9: Automatic meshes generated by Ansys

Hexahedral elements have two significant advantages over linear tetrahedra. They are computationally cheaper per node, since a hex element captures the same volume as several tetrahedra, reducing the total node count for a given region. They also capture bending correctly, whereas linear tetrahedra suffer from shear locking: their constant-strain formulation forces the element edges to remain straight, so bending can only be represented by stretching, which artificially stiffens the model [15]. This is shown in figure 10 where each tetrahedron is at a constant strain causing large discontinuities across the model.

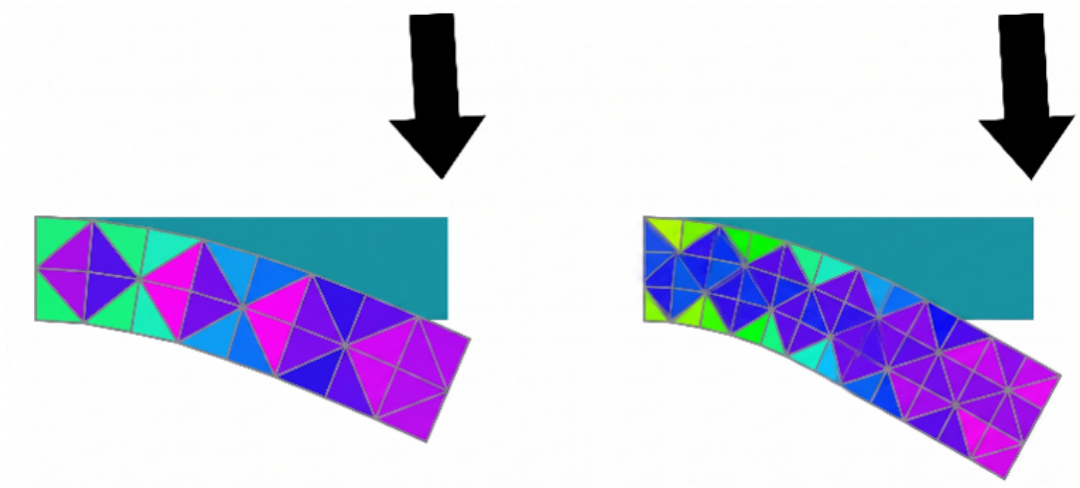


Figure 10: Demonstration of shear locking in tetrahedral mesh. Each triangle is a single colour denoting constant strain [15]

Hexahedral elements were therefore used wherever the geometry permitted. Because they are harder to fit to complex shapes, considerable time was spent building up the meshes manually: bodies were split to allow face meshes to be swept through prismatic regions, with topology sharing applied at the body splits so that the nodes connect across the joins to give a continuous mesh for the solver. The same components from figures 9a & 9b are shown in figure 11 with a manually constructed hexahedral mesh.

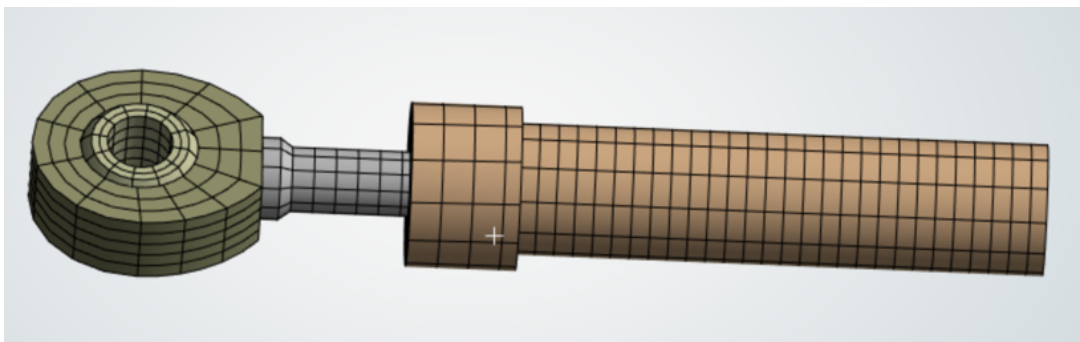


Figure 11: Manually constructed mesh utilising hexahedral elements

The final overall mesh is displayed in figure 12.



Figure 12: Final mesh of full suspension system

6.3 Materials and Composite layup

The analysis was performed in Ansys' Static Structural module. The aim of the model was to determine the change in suspension geometry under representative load cases, and for this purpose the assembly could be assumed to have reached static equilibrium at each load point. This assumption excludes inertia and material damping, both of which would matter under high-speed transient cases.

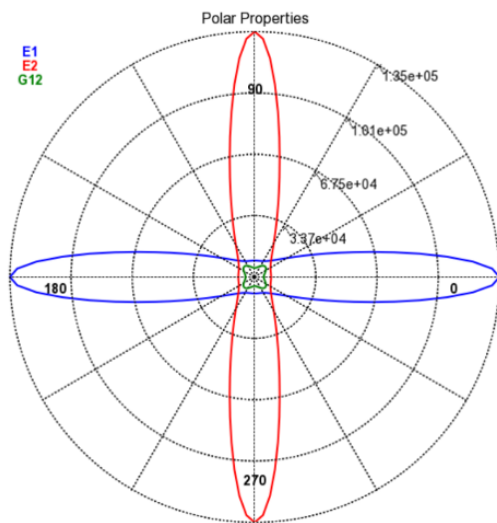
The next task in Ansys was to define the materials correctly. The clevises and inserts are predominantly steel or aluminium alloy, both of which are typically homogeneous and isotropic, with stiffness independent of direction. The wishbones, push rods and tie rods, however, are carbon-fibre composite tubes. Carbon fibre is highly anisotropic, with very different stiffness along and perpendicular to the fibre direction, and it must therefore be modelled correctly to obtain accurate results.

In Ansys this is handled in the ACP (Ansys Composite PrepPost) module, where the properties of each ply are defined together with the layup, including ply thickness and orientation across the structure [16]. The carbon tubes used in the suspension were manufactured by roll-wrapping with the layup given in table 2 where 0 degrees is axially along the tube.

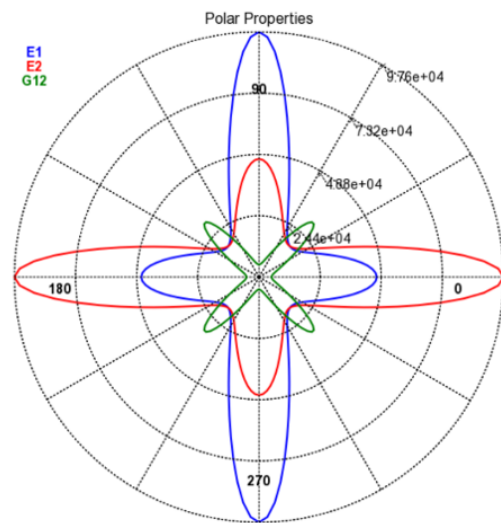
Table 2: Composite layup orientation by ply number

Ply Number	Orientation
Inner Surface	
1	90
2	0
3	90
4	0
5	90
6	0
7	0
8	0
9	0
10	0
Outer Surface	

The modulus properties of each individual ply are displayed in figure 13a and of the total stackup in figure 13b.



(a) Single Ply



(b) Stackup

Figure 13: Graphs of Young's and Shear modulus by direction for the individual carbon plys and overall stackup

The effect of the layup on tube performance is visible in the stress distribution shown in figure 14.

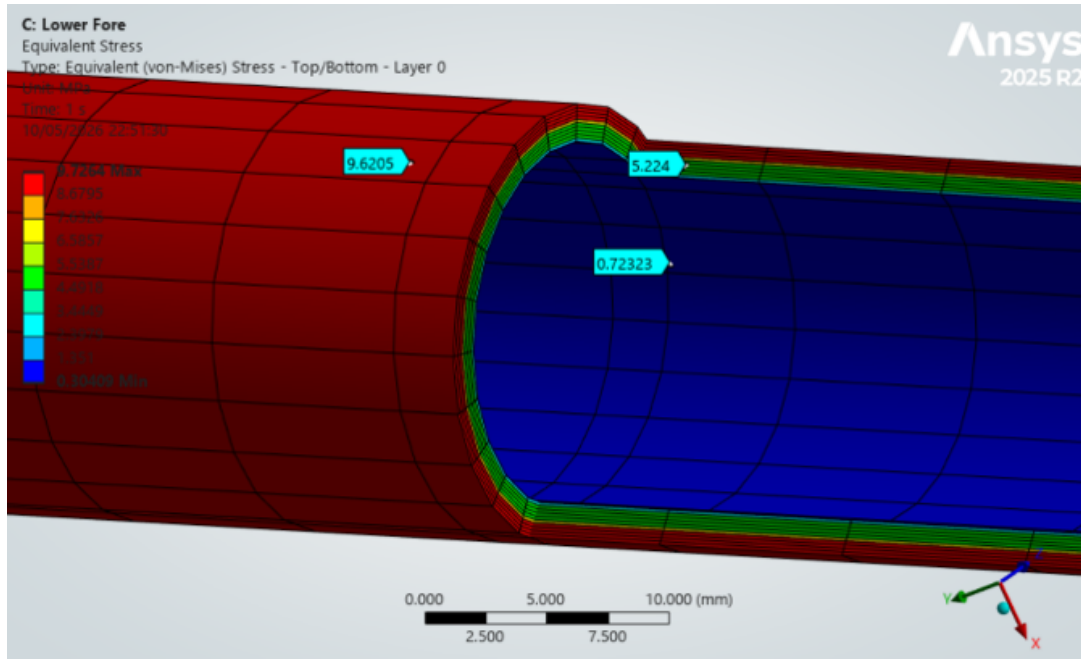


Figure 14: Stress distribution by ply for carbon tube at mid-span

Because all of the plies are bonded together, the strain across the wall thickness is approximately constant as seen in figure 15.

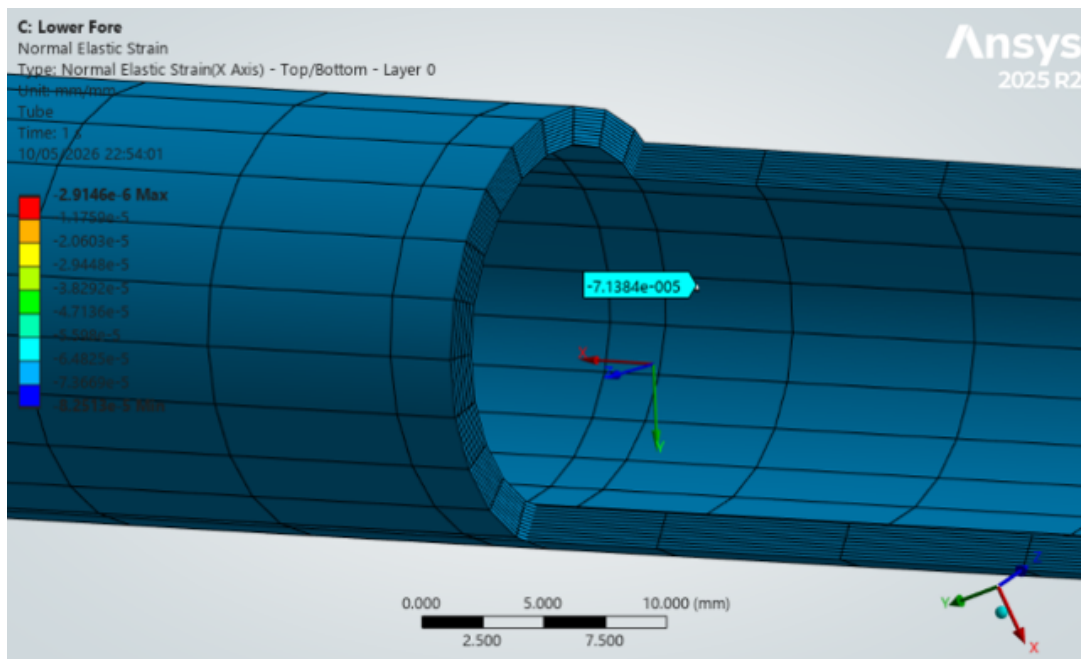


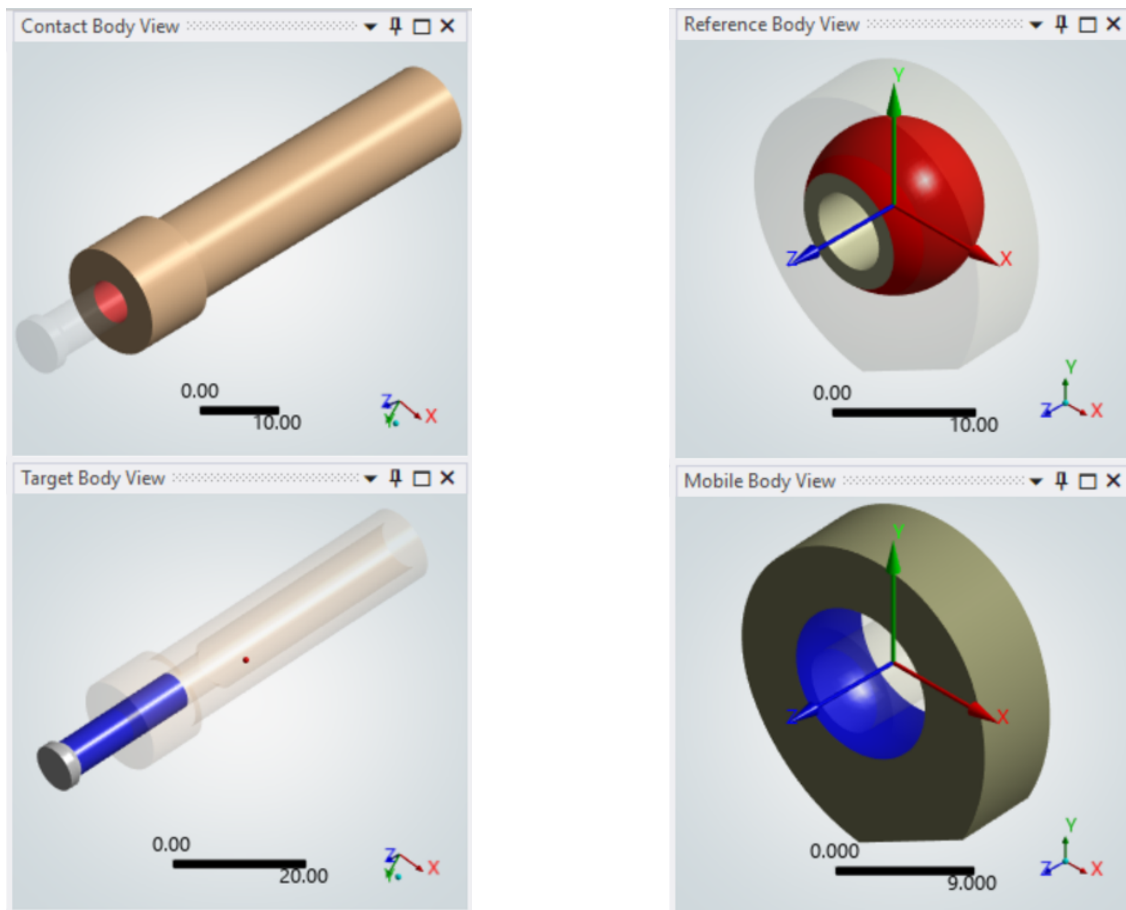
Figure 15: Strain distribution by ply for carbon tube at mid-span

Therefore, the load is shared in proportion to the local stiffness. The inner plies, oriented at 90 degrees, have a much lower axial stiffness and therefore carry a smaller proportion of the axial load; the outer plies, oriented axially, carry the majority.

6.4 Contacts and Constraints

With materials defined, contacts between components were specified. Contact definition is another area where computational time can grow significantly without due care. In the real world most interfaces correspond to what Ansys terms a frictional contact: parts cannot penetrate, there are equal and opposite forces where they are in contact and the stiffness drops to zero on separation. Sliding is permitted, with the resisting force dependant on the normal load. This makes the system non-linear: the global stiffness matrix is no longer constant as the contact forces change.

The contact type with the lowest computational overhead is a bonded contact, in which the solver simply connects the nearest nodes on each surface and effectively merges the meshes. The stiffness matrix then depends only on the elements themselves, not on the relative position of the parts. A no-separation contact performs a similar role, keeping the meshes locked together but allowing the specific node pairings to update on each iteration to permit sliding. Contacts in this work were specified to use bonded or no-separation (including joints) behaviour wherever the physics permitted, an example of each is given in figures 16a & 16b.



(a) Bonded Contact

(b) No Separation Contact

Figure 16: Examples of bonded and no-separation contacts

Some non-linearity is unavoidable in a suspension analysis. The corner spring and damper are represented by a spring connection in the Ansys model as shown in figure 17.

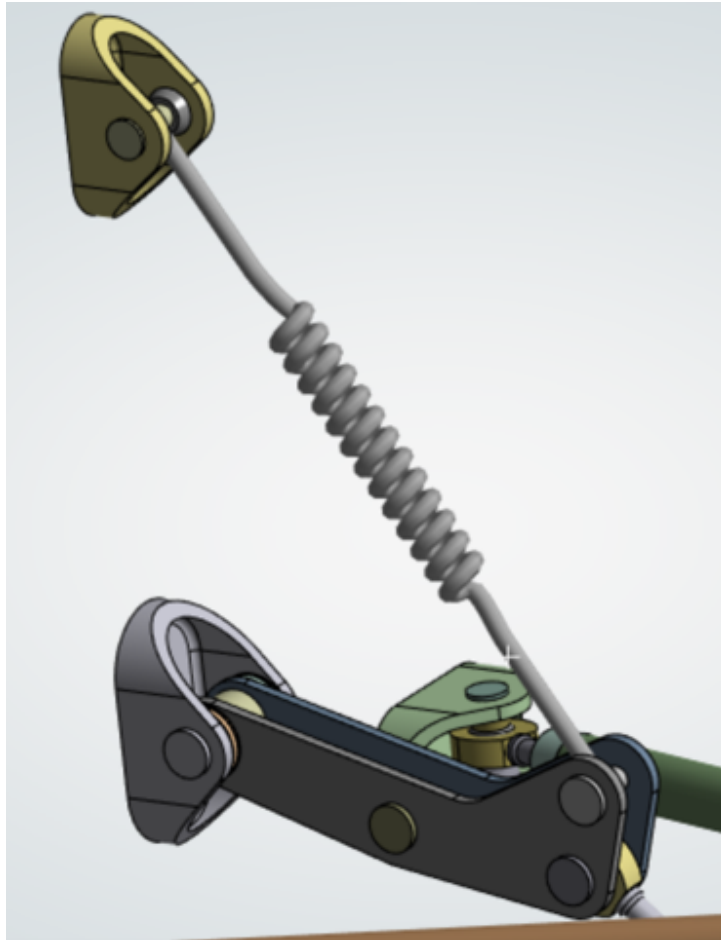


Figure 17: Spring connection used to replicate corner spring

This generates substantial rigid-body motion and the global stiffness matrix changes significantly through the application of load. Large-deflection behaviour was therefore enabled so Ansys could update the global stiffness matrix after each iteration. Because each tube can rotate freely about its end ball joint its perpendicular stiffness is effectively zero which also risks solver instability due to small reaction forces, therefore weak springs were added to provide nominal stiffness in all directions, stabilising the solve at a small computational cost.

6.5 Outcome of the full-assembly model

Even with these measures, an attempt to run the full-assembly model showed that 1/10th of a single load case took two hours to solve. The mesh was relatively detailed, capturing most key features, and although the mesh was further coarsened and the load applied in many small substeps, the solver still struggled to converge as the system's rigid-body motion caused the global stiffness matrix to change significantly between iterations. Each load step that proved too large produced several wasted iterations before the solver bisected the step, lengthening the runtime further. This behaviour is shown in figures 18 & 19.

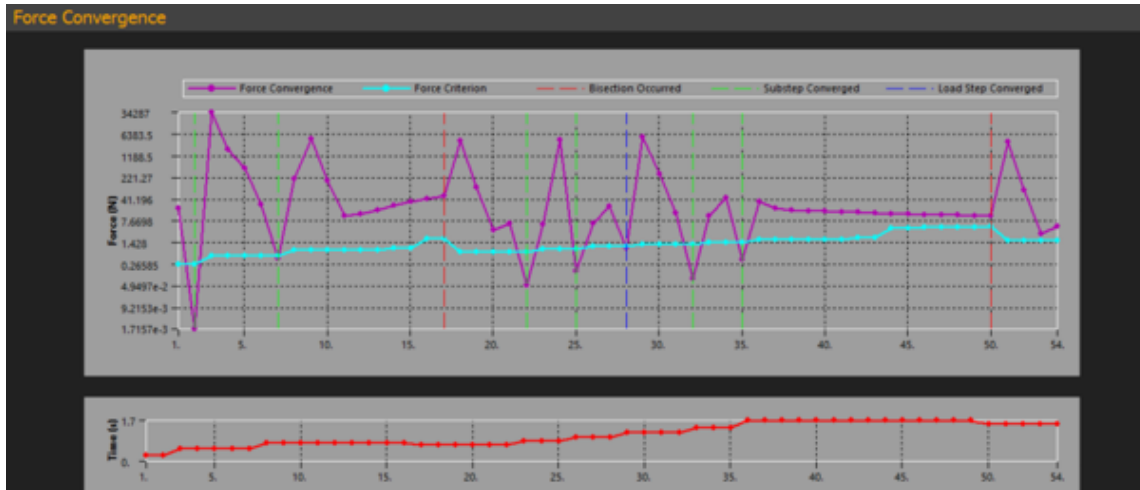


Figure 18: Ansys force convergence plot

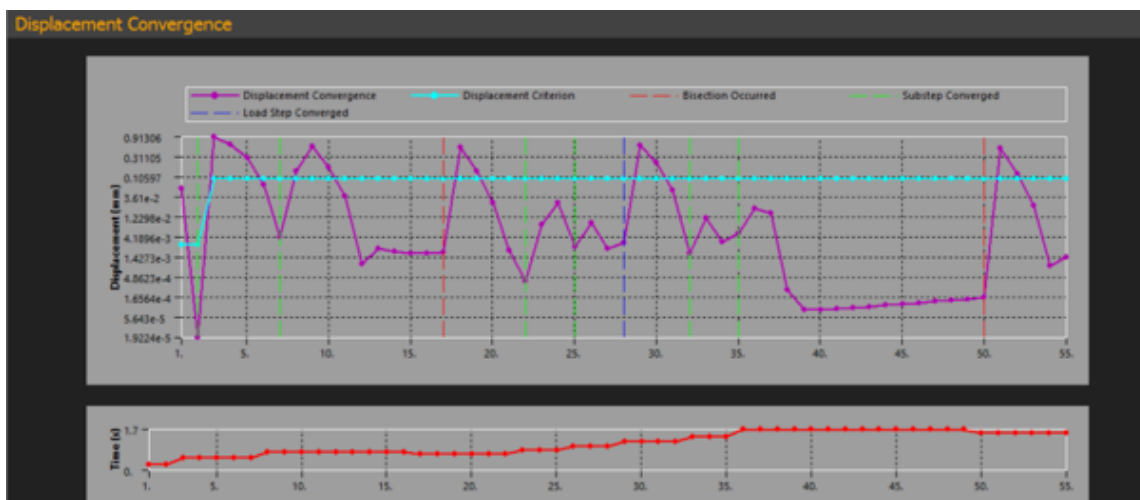


Figure 19: Ansys displacement convergence plot

The intermediate result (for the stress distribution) when the simulation was stopped is shown in figure 20.

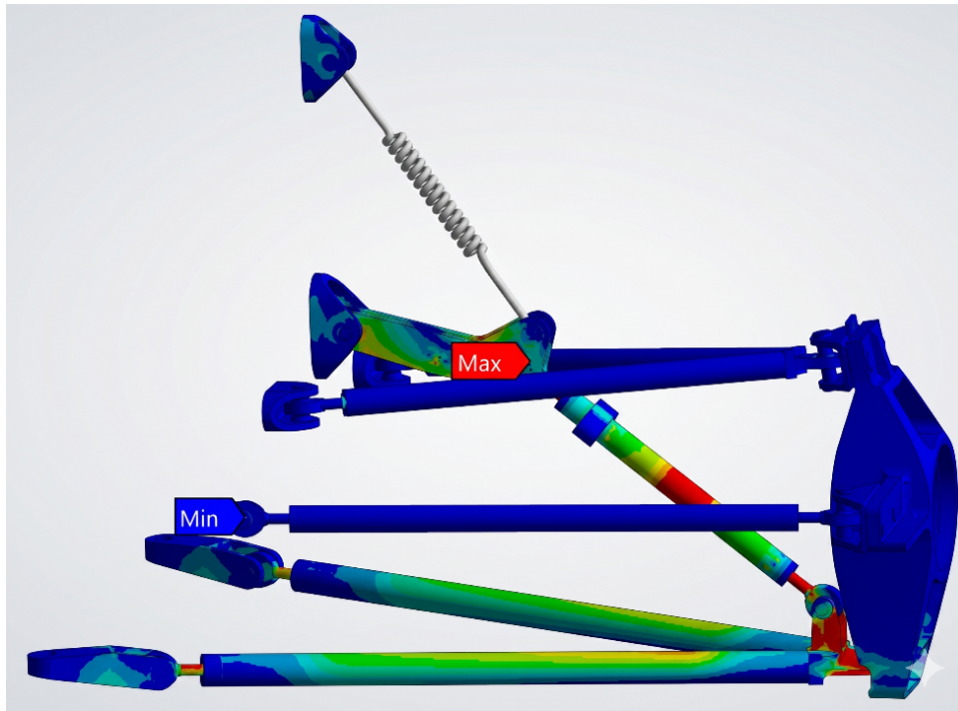


Figure 20: Intermediate full suspension stress distribution on termination of Ansys solve

The load at the point the simulation stopped was a vertical force of 500N. As can be seen in figure 20 this was mainly reacted by a high compressive stress in the push rod and corresponding tension in the lower wishbones, which is a result consistent with that obtained by the MATLAB solver in section 8.

This experience confirmed the research in section 5 that FEA is most effective when applied to predominantly linear problems rather than mechanism-level analyses dominated by rigid-body motion. The analysis therefore transitioned to the individual component sweeps required to generate force-displacement maps for the MATLAB solver. Most components in the suspension have only two attachment points: each clevis connects the chassis to a wishbone, and each wishbone connects a clevis to an outboard point on the upright. By fixing the inboard side of a component and sweeping the outboard side through a range of displacements, the reaction force required to achieve each displacement could be recorded directly. This process is shown through figures 21, 22 & 23.

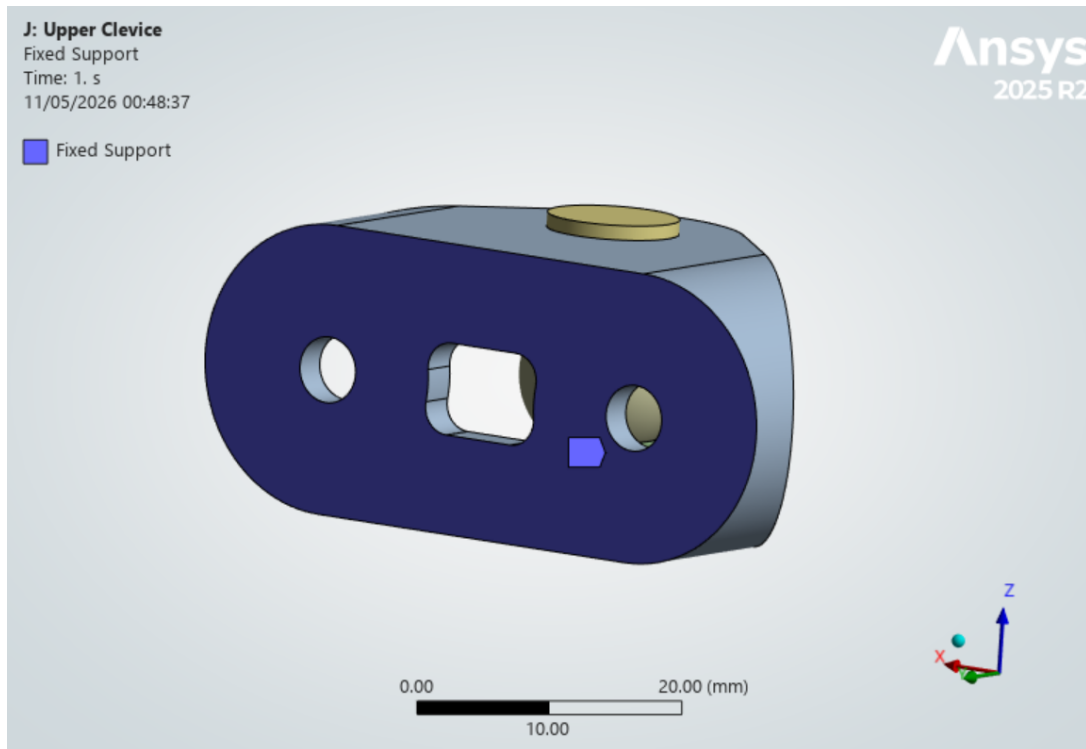


Figure 21: Chassis side of clevis supported with fixed constraint

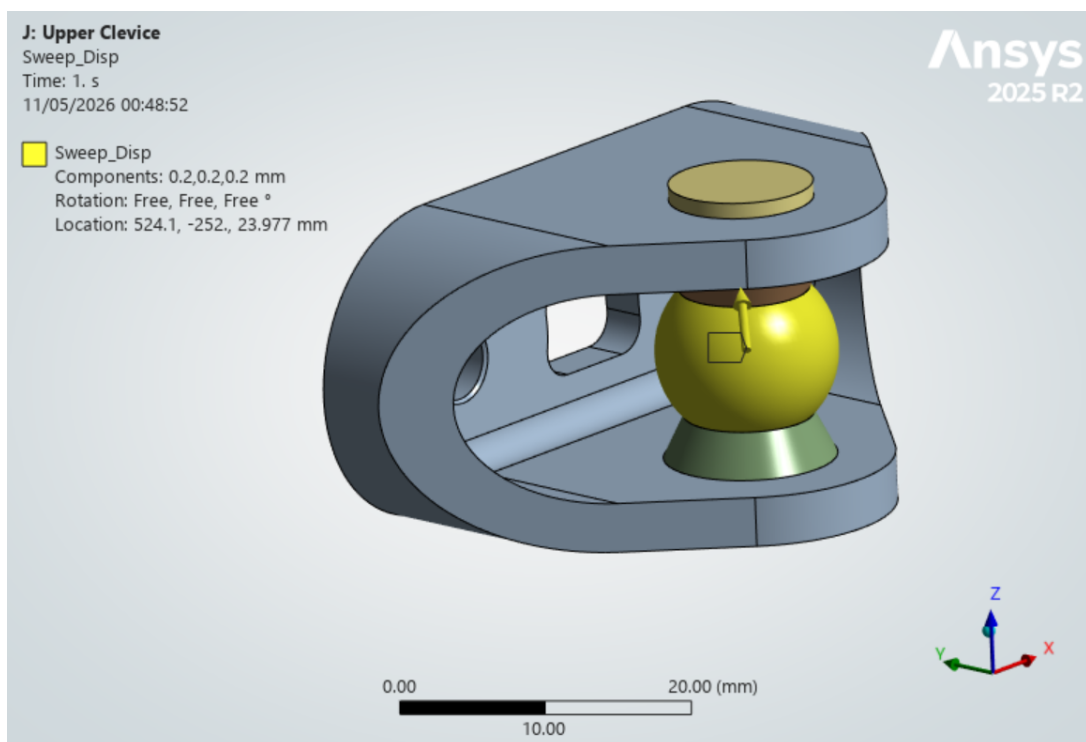


Figure 22: Node on outboard side of clevis swept through range of displacements

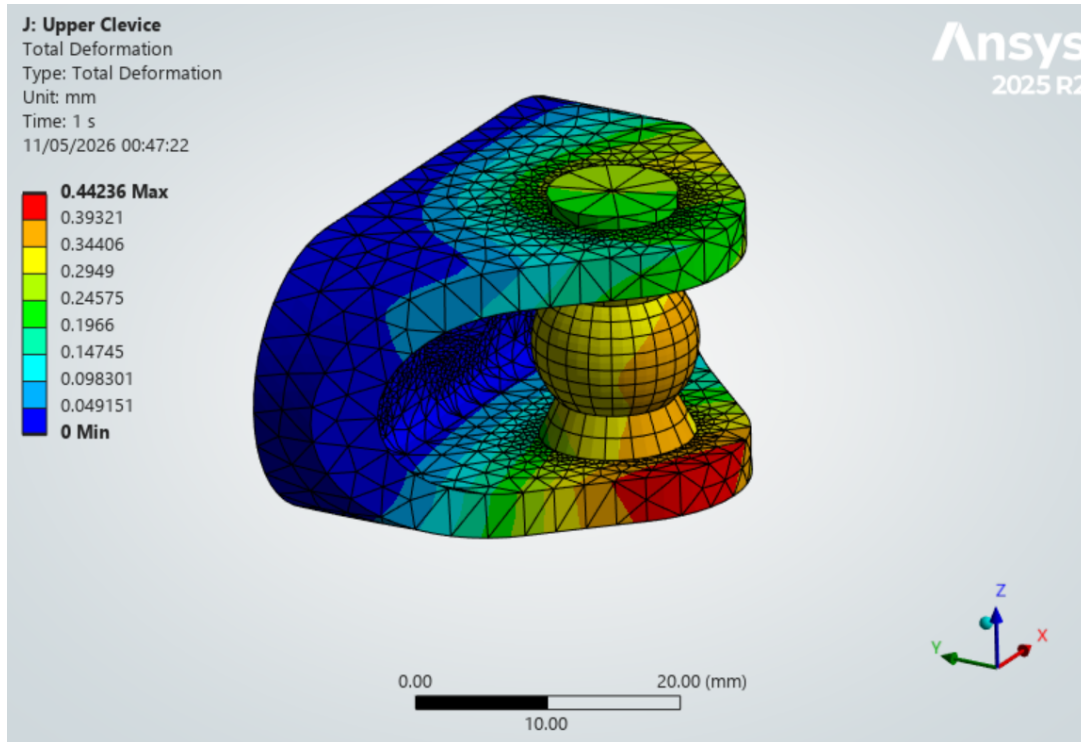


Figure 23: Upper clevis deformation as a result of node displacement sweep

The displacement sweep was performed using the Python script in appendix B.1, and the resulting reaction forces recorded in CSVs like that in figure 24.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	X	Y	Z	Fx	Fy	Fz	Status	Message					
2	-0.2	-0.2	-0.2	-51167.3	-80864.1	-13381.4	Success	None					
3	-0.2	-0.2	0	-51104.5	-80830.1	-96.7238	Success	None					
4	-0.2	-0.2	0.2	-51041.7	-80796.2	13187.95	Success	None					
5	-0.2	0	-0.2	-51179.9	-21.3837	-13347.5	Success	None					
6	-0.2	0	0	-51117.1	12.5496	-62.7905	Success	None					
7	-0.2	0	0.2	-51054.3	46.4829	13221.89	Success	None					
8	-0.2	0.2	-0.2	-51192.4	80821.29	-13313.5	Success	None					
9	-0.2	0.2	0	-51129.6	80855.23	-28.8572	Success	None					
10	-0.2	0.2	0.2	-51066.8	80889.16	13255.82	Success	None					
11	0	-0.2	-0.2	-50.2409	-80876.6	-13318.6	Success	None					
12	0	-0.2	0	12.5496	-80842.7	-33.9333	Success	None					
13	0	-0.2	0.2	75.3401	-80808.7	13250.74	Success	None					
14	0	0	-0.2	-62.7905	-33.9333	-13284.7	Success	None					
15	0	0	0	0	0	0	FAILED	You need at least one structural load to proceed with the solution.					
16	0	0	0.2	62.7905	33.9333	13284.68	Success	None					
17	0	0.2	-0.2	-75.3401	80808.74	-13250.7	Success	None					
18	0	0.2	0	-12.5496	80842.68	33.9333	Success	None					
19	0	0.2	0.2	50.2409	80876.61	13318.61	Success	None					
20	0.2	-0.2	-0.2	51066.83	-80889.2	-13255.8	Success	None					
21	0.2	-0.2	0	51129.62	-80855.2	28.8572	Success	None					
22	0.2	-0.2	0.2	51192.41	-80821.3	13313.53	Success	None					
23	0.2	0	-0.2	51054.28	-46.4829	-13221.9	Success	None					
24	0.2	0	0	51117.07	-12.5496	62.7905	Success	None					
25	0.2	0	0.2	51179.86	21.3837	13347.47	Success	None					
26	0.2	0.2	-0.2	51041.73	80796.19	-13188	Success	None					
27	0.2	0.2	0	51104.52	80830.13	96.7238	Success	None					
28	0.2	0.2	0.2	51167.31	80864.06	13381.4	Success	None					

Figure 24: Upper clevis force-displacement CSV file

The number of runs required scales rapidly with the resolution of the sweep. For three sample points along each of the three axes, $3^3 = 27$ simulations are required; for four points, this rises

to $4^3 = 64$. Since the maps are intended to be built once and re-used, finer resolutions remain feasible in the long-term, but a $3 \times 3 \times 3$ grid was used in this project as a proof of concept. Each 27 point sweep took 20 minutes to run meaning with 5 discrete components it took 1 hour 40 minute to define all the maps.

7 MATLAB Solver

With the FEA-side of the pipeline producing a CSV map of force against displacement for each component, the MATLAB solver was developed to assemble these components into a complete suspension and resolve the forces and displacements under user-defined load cases. The structure of the program follows the following data flow: geometry definition, coordinate-system construction, map import and interpolation, user input, non-linear solve and output.

7.1 Geometric framework

All operations in the program are expressed in linear algebra, with vectors representing positions and directions, and matrices representing rotations and projections. Dot and cross products are used to construct normals and parallels and to project vectors into specific directions, in line with the standard treatment of geometry in mechanical software [17].

Each suspension node is defined initially in its unloaded position, with the suspension at full droop where the spring is at its natural length. The coordinates are taken directly from the CAD model and expressed in the global TBRe coordinate system, with the origin at the centre top of the chassis at the front wing mounting point and axes as illustrated in figure 25.

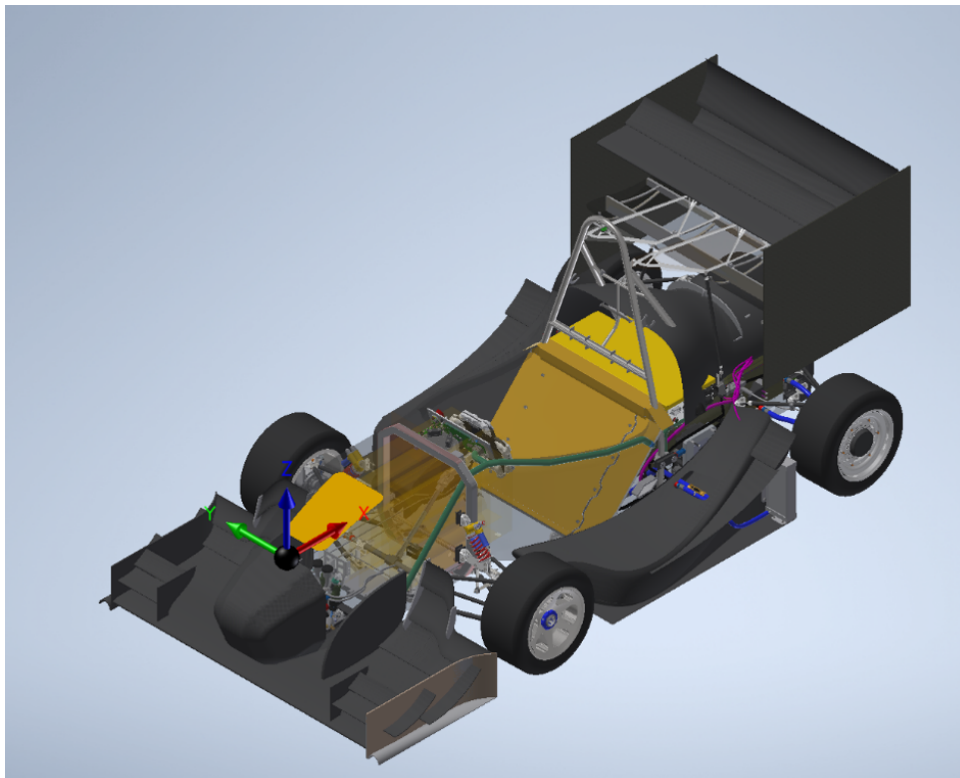


Figure 25: TBRe global coordinate system

The work in this report focuses on the front-left suspension corner, so all y-coordinates are negative. As the assembly does not include a wheel/tire, the loading point is specified at the centre of the upright hub mounting as shown in figure 26.

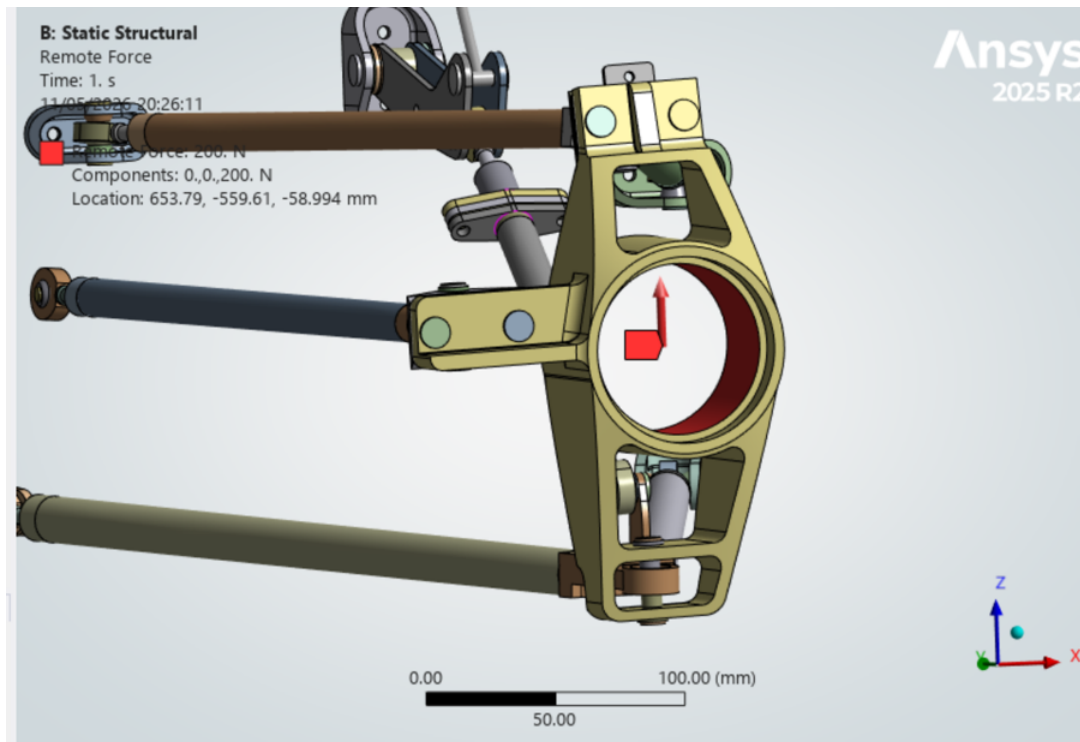


Figure 26: Load point position on upright

7.2 Rocker local coordinate system

The rocker is the component that presents the most difficulty when modelling in MATLAB. The wishbones each have only two mounting points and carry essentially axial loads, so the force in a wishbone is determined entirely by the change in length between its two attachment points and is independent of its global position. The clevises have three-dimensional loading, but again have only two attachment points, one of which is fixed in the global coordinate system; only the displacement of the non-fixed point is required to query the force-displacement map.

The rocker is different in two respects. Its global position changes substantially through suspension travel, simply by virtue of its location in the mechanism, and it has three attachment points (the pivot, the spring eye, and the push-rod attachment), all of which can move in global space and whose relative positions must be resolved. The stiffness map for the rocker therefore must be expressed in a local coordinate system, and rather than sweeping a single point's displacement, the interconnected forces and displacements at all three attachment points must be characterised.

To handle this, the pivot point was set as the local origin, with the local x-axis pointing from the pivot toward the spring attachment, and the local z-axis normal to the rocker plane (aligned with the global x-axis at rest) as shown in figure 27.

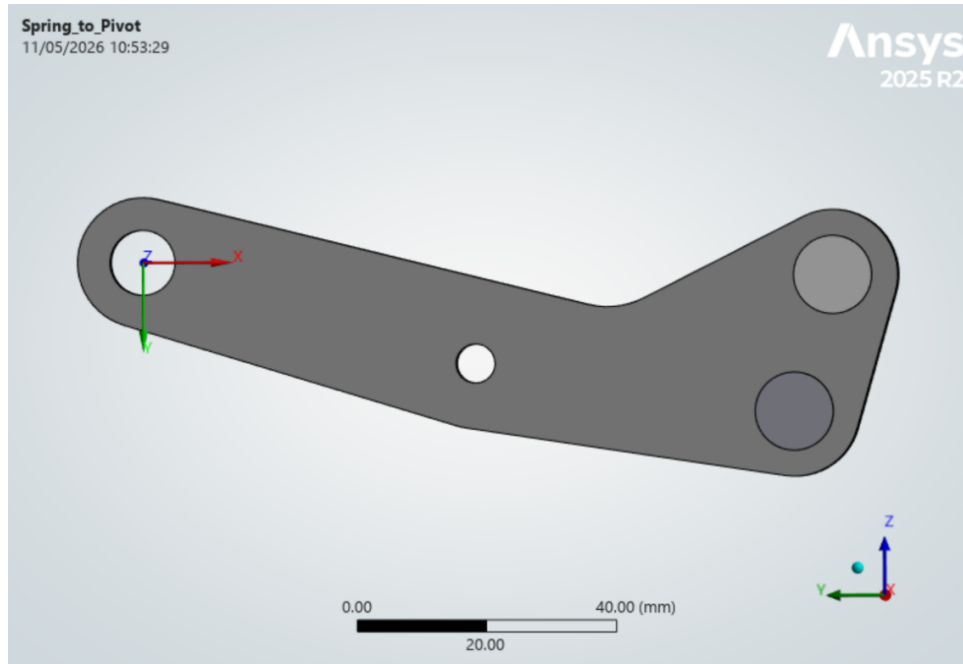


Figure 27: Rocker local coordinate system

The spring and push-rod attachment points were then swept in turn relative to the pivot, with the pivot and remaining point held at a fixed displacement during each sweep, whilst the reaction forces at the pivot and target point were recorded. The reaction force directions at these points when sweeping the spring connection are shown in figure 28, producing the deformation in figure 29.

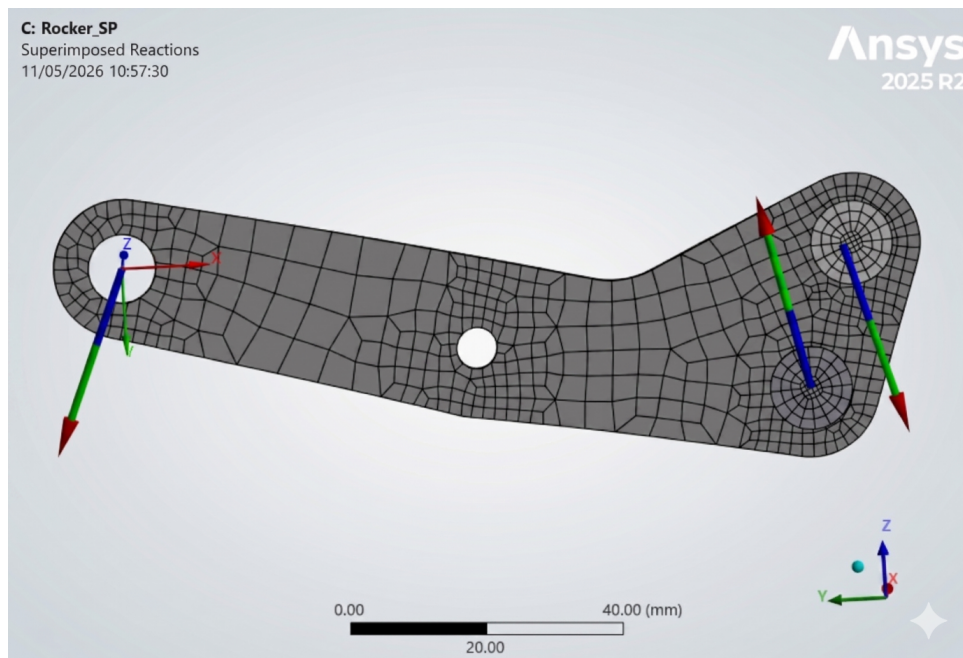


Figure 28: Reaction forces at each rocker connection node

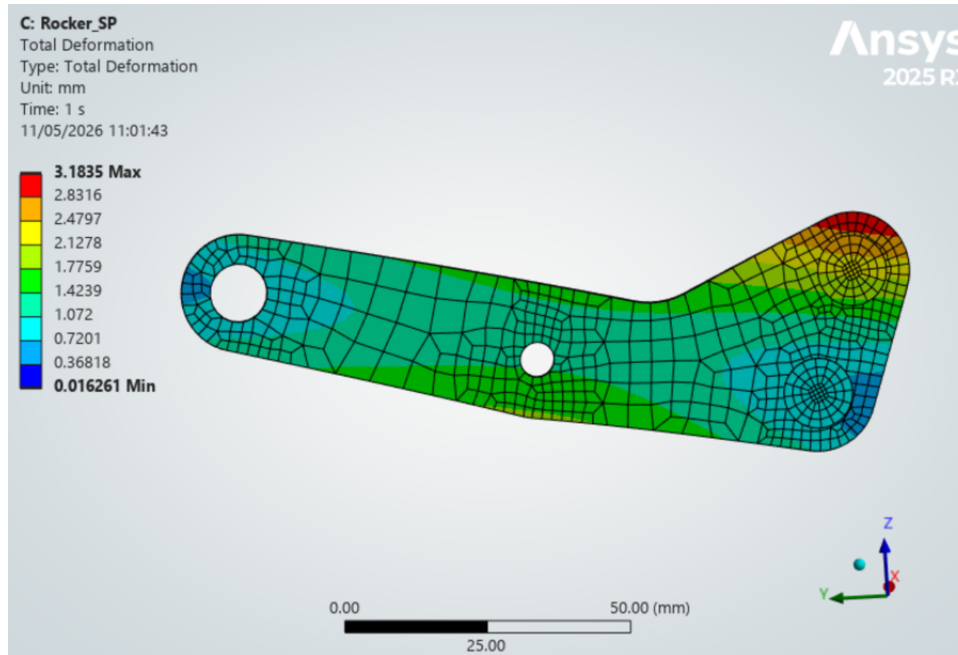


Figure 29: Rocker deformation as a result of sweeping spring connection node

Knowing the positions of the spring and push-rod attachment relative to the pivot fully defines the rocker in its local coordinate space, and the reaction forces required to maintain a given configuration can be obtained by superposing the contributions from each individual node's force-displacement map.

7.3 Construction of the local-to-global rotation matrix

The local axes of the rocker described in section 6.2 are constructed in MATLAB as follows. The plane of the rocker is defined via its normal: two in-plane direction vectors are constructed, one pointing from the pivot to the push-rod attachment and one from the spring attachment to the push-rod attachment, and their cross product gives the rocker plane normal. The local y-axis is defined as orthogonal to both the local x-axis and the rocker plane normal, and is obtained by taking the cross product of these two vectors. The local z-axis is then orthogonal to both the local x and y axes, again obtained by cross product. The corresponding code is given in figure 30.

```

47
48 % Find local push rod attachment position
49 local_x = Rocker_Spring_Attachment - Rocker_Clevice_Attachment;
50 rocker_plane_1 = Rocker_Push_Rod_Attachment - Rocker_Spring_Attachment;
51 rocker_plane_2 = Rocker_Push_Rod_Attachment - Rocker_Clevice_Attachment;
52 rocker_normal = cross(rocker_plane_1,rocker_plane_2);
53 local_y = cross(local_x,rocker_normal);
54 local_z = cross(local_x,local_y);
55

```

Figure 30: Code to define rocker local coordinate system

Once the three direction vectors have been computed, they are normalised to unit magnitude. Because the three normalised vectors are the directions of the local axes expressed in the global

frame, they are concatenated as columns to form the local-to-global rotation matrix. The resulting rotation matrix has determinant equal to one and so does not scale the magnitude of any vector it rotates. The transpose of this matrix is the global-to-local rotation, which is what is required by the program. The starting positions of the spring and push-rod attachment can then be expressed in local coordinates, and the initial rocker dimensions computed as shown in figure 31.

```

55
56      % normalise local axis vectors
57      local_x = local_x / norm(local_x);
58      local_y = local_y / norm(local_y);
59      local_z = local_z / norm(local_z);
60
61      % concatenate local axis vectors to form rotation matrix
62      local_to_global = [local_x,local_y,local_z];
63
64      % transpose
65      global_to_local = local_to_global';
66
67      % find initial rocker local dimensions
68      global_push_pos = Rocker_Push_Rod_Attachment - Rocker_Clevice_Attachment;
69      global_spring_pos = Rocker_Spring_Attachment - Rocker_Clevice_Attachment;
70
71      local_push_pos Og = global_to_local * global_push_pos;
72      local_spring_pos Og = global_to_local * global_spring_pos;
73

```

Figure 31: Code showing formation of the global-to-local rotation matrix and finding initial local rocker dimensions

It should be noted that, although the original intention was to include compliance of the rocker (and the upright) within the model, this could not be achieved within the project timeframe. As described in Section 5.5, generating a map for a component with more than two relevant nodes requires sweeping each node against every other node in three dimensions, leading to a sweep size of $27^3 = 19,683$ runs for three points in each dimension. This was prohibitive for the available computational time. The full pipeline does however accept such maps once they are generated, so the tool could be easily extended to include them in future. For the purposes of this study the rocker and upright were therefore treated as rigid.

7.4 Camber and Toe relative to the upright

The next stage establishes the relationship between the upright inclination and the static camber and toe of the wheel. In a model that included the tyre, the camber and toe could be obtained directly by projecting the wheel-plane normal onto the global yz-plane and comparing with the y-axis (for camber), or onto the xy-plane and comparing with the y-axis (for toe) as shown in figure 32.

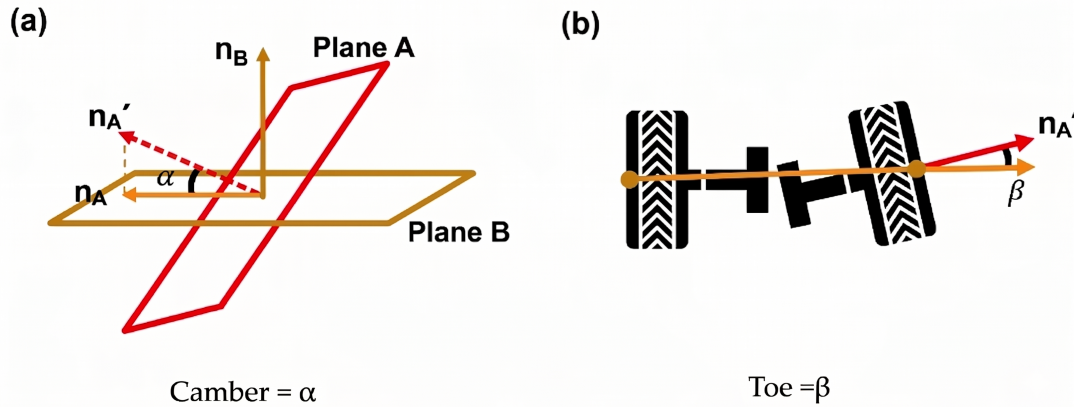


Figure 32: Plane A shows plane of wheel and corresponding projections required to find a) camber and b) toe [18]

In this work the model terminates at the upright, and the upright plane is in general slightly offset from the true wheel plane, so the offset between the two normals must first be calculated.

This is achieved by defining the suspension at full droop, where the camber and toe are known. These angles are converted to rotation matrices, and the y-axis unit vector is rotated by their composition to give the direction vector that the wheel-plane normal would adopt if a tyre were attached to the upright in the model. The angle between this direction and the upright normal yields a fixed offset, which is then applied at every subsequent step so that wheel camber and toe can be calculated from the upright normal alone. The code to achieve this is displayed in figure 33.

```

114
115 % For camber and toe use plane of upright with upper, lower and tie rod
116 % points
117
118 % first find normal to this plane as unit direction vector
119 vector_1 = Upright_Tie_Rod - Upright_Lower;
120 vector_2 = Upright_Tie_Rod - Upright_Upper;
121 normal_upright = cross(vector_1, vector_2);
122
123 % normal to tyre with no toe or camber
124 normal_to_standard_tire = [0;-1;0];
125
126 % correct camber and toe values at full droop
127 full_droop_camber = 1.15;
128 full_droop_toe = -0.65;
129
130 alpha = deg2rad(full_droop_camber);
131 gamma = deg2rad(full_droop_toe);
132
133 % Form rotation matrices
134 Rx = [1, 0, 0;
135       0, cos(alpha), -sin(alpha);
136       0, sin(alpha), cos(alpha)];
137
138 Rz = [cos(gamma), -sin(gamma), 0;
139       sin(gamma), cos(gamma), 0;
140       0, 0, 1];
141
142 % Combine both rotations into single matrix
143 R = Rz * Rx;
144
145 % Create and normalise vector normal to full droop wheel plane
146 normal_to_full_droop_tyre = R * normal_to_standard_tire;
147 unit_1 = normal_to_full_droop_tyre / norm(normal_to_full_droop_tyre);
148
149 % Find offset between upright plane normal and tyre plane normal
150 static_offset_rotation = rot_between_vectors(normal_upright, normal_to_full_droop_tyre);
151

```

Figure 33: Code for setup of camber and toe static offset between upright and wheel normal

7.5 FEA map import and interpolation

The next section of the code calls functions that load the FEA maps from their CSV files and constructs interpolations of them. For the carbon tubes the map is one-dimensional, since the tubes should never see a meaningful bending load. For components such as the clevises a three-dimensional map is required for each axis of force. The displacement of the attachment point in x, y and z can then be passed as arguments to each of the three interpolated maps, returning the reaction force in each global direction. An example of the map for the Fy reaction force of the upper clevis is given in figure 34.



Y

solver in the wrong direction.

7.6 User input and the staged solve

indicating which has been provided in each axis as shown in figure 35.

199
200
201
202
203
204
205
206
207
208
209
210
211
212

'sweep'. Either force or displacement boundary conditions must be specified too.

This ensures that exactly three unknowns remain at the load point, that the assembly is fully determined, and that a unique solution exists. The program then calls `fsolve` in two stages. The

non-linear solver finds local minima, so the initial guesses for the suspension node positions must be in the correct region; otherwise members can become inverted and the suspension can flip into a different equilibrium configuration that has no physical meaning as demonstrated by figures 36a & 36b.

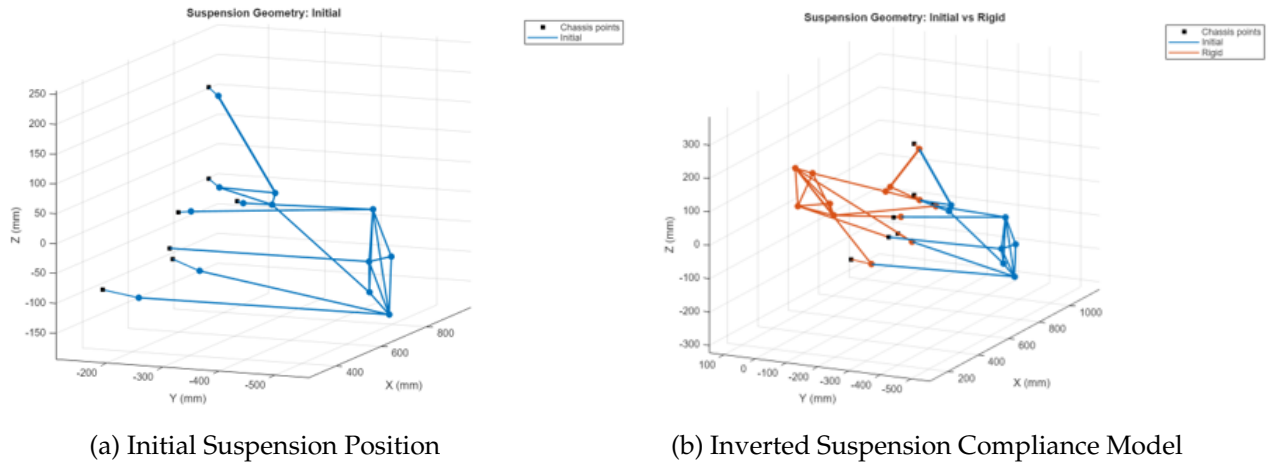


Figure 36: Illustration of the issue if initial node position guesses are all set to 0 when running the compliance model; suspension can invert.

In the first call to fsolve, the static suspension positions at full droop are passed as the starting guess, and only the upright attachment points are treated as unknowns, with no compliance permitted at individual components. The residual equations are formulated such that the entire vertical load is reacted by the z-component of the suspension corner spring, which is a good approximation in the no-compliance scenario since the spring carries almost all of the vertical force. This stage gives a reliable starting point for the second solve. The second call to fsolve then treats all of the clevis, rocker and upright attachment points as unknowns and permits compliance of the suspension members. Because the previous stage has placed the solver close to the true solution, convergence is rapid and reliable. An illustration of how the solution progresses from the initialisation through the two solves is given in figures 37, 38 & 39.

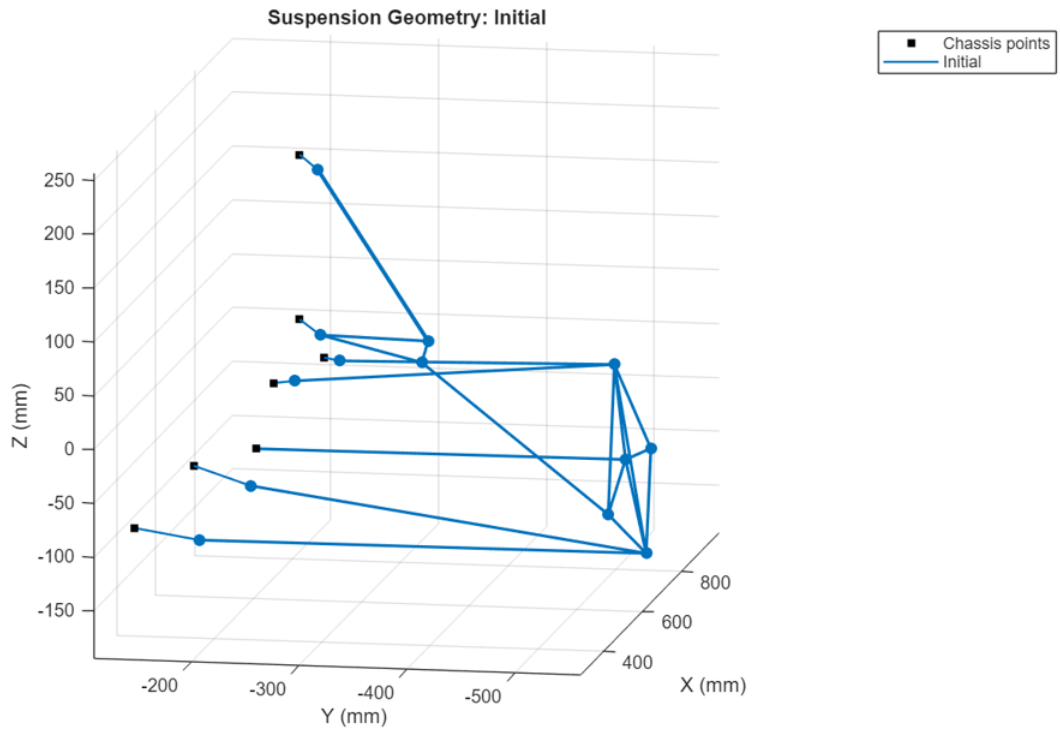


Figure 37: Initial Suspension Position

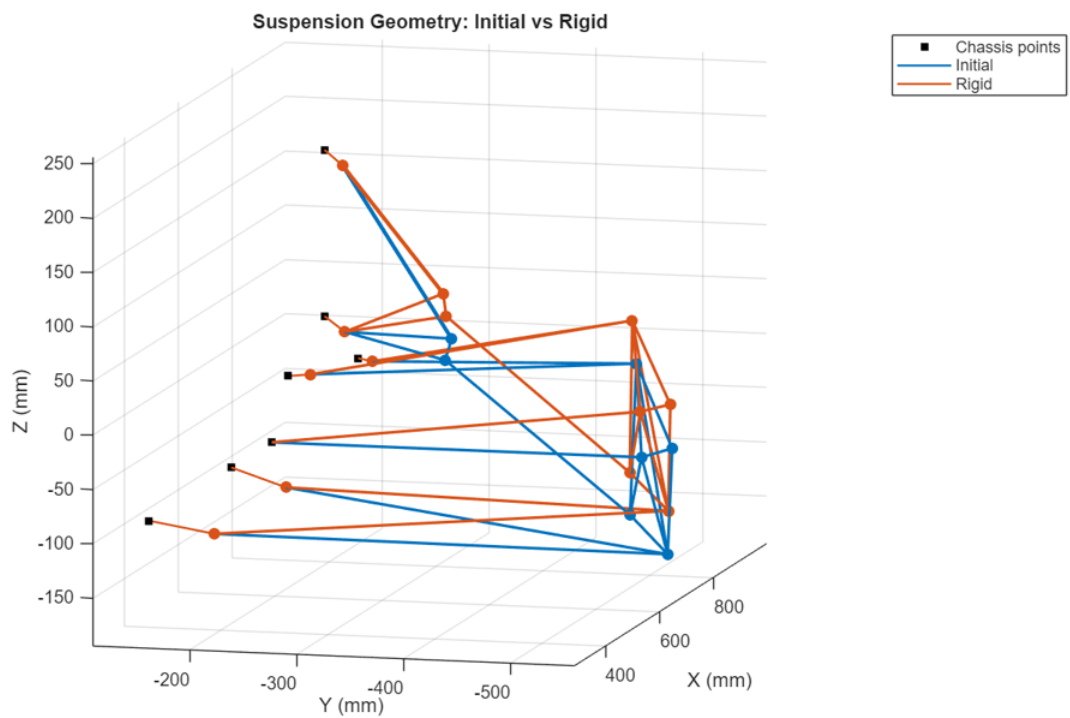


Figure 38: Suspension position with rigid members and all vertical force reacted by spring

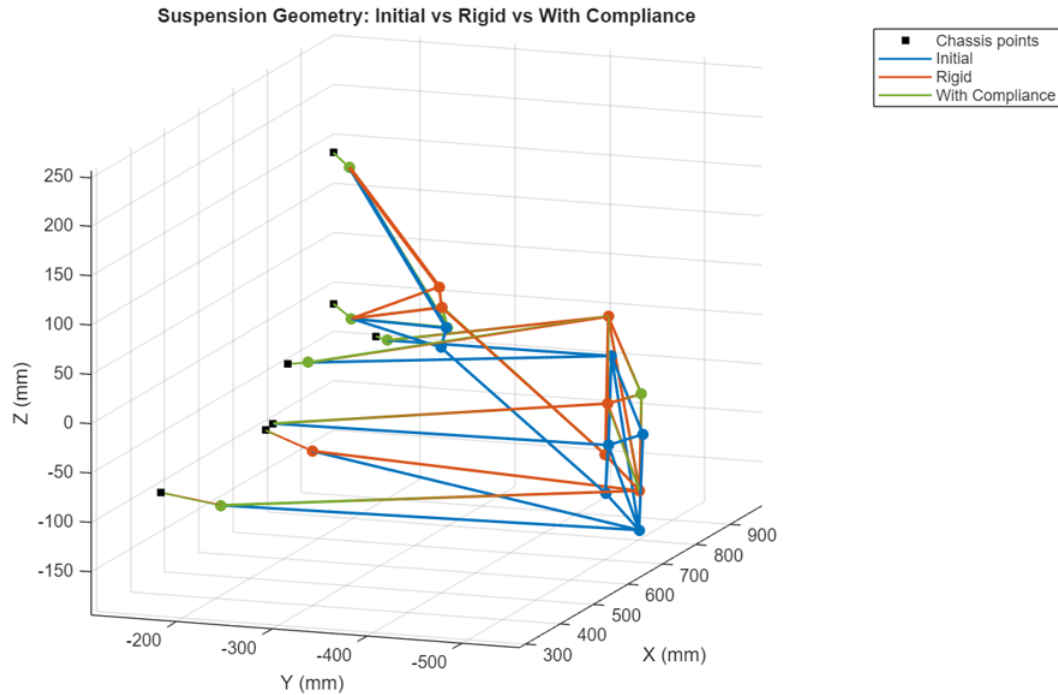


Figure 39: Final solve with compliant suspension methods, does not move far from rigid solve

7.7 Residual function and equation count

Within the residual function passed to `fsolve`, three times as many equations as nodes are required to constrain the problem in x , y and z . At any node that connects two components, such as a clevis attachment, the equations are simple force balances in each of the three global axes, with the reaction force in the clevis found by passing its displacement through the interpolated map. An example of this for the lower front wishbone and clevis is given in figures 40 & 41.

```

966 % Lower Front Force
967 new_LF = Upright_Lower_temp - Front_Lower_Clevice_Attachment_temp;
968 delta = (norm(new_LF)) - LF;
969 Force = interps.Tube_Fx_interp(delta);
970 LF_Force = -Force * (new_LF / norm(new_LF));
971

```

Figure 40: Force calculated in lower front wishbone from extension of wishbone and interpolation map

```

1007 % Lower Front
1008 delta_vector = (Front_Lower_Clevice_Attachment_temp - LFC)';
1009 LFC_Force(1) = interps.Lower_Clevice_Fx_interp(delta_vector);
1010 LFC_Force(2) = interps.Lower_Clevice_Fy_interp(delta_vector);
1011 LFC_Force(3) = interps.Lower_Clevice_Fz_interp(delta_vector);
1012 res(10:12) = LFC_Force + LF_Force;
1013
1014

```

Figure 41: Wishbone force balanced in all 3-dimensions with force at clevis found from interpolation map

For the upright and rocker, force and moment balances are derived across the whole component, to capture the multi-point connection as in figure 42.

```

1053
1054      % Total moment balance about rocker pivot
1055      between = Rocker_Spring_temp - Rocker_Clevice_Attachment_temp;
1056      moment_1 = cross(between, Spring_Force);
1057      between = Rocker_Push_temp - Rocker_Clevice_Attachment_temp;
1058      moment_2 = cross(between, -PR_Force);
1059      M_Total = moment_1 + moment_2;
1060      res(28:30) = 0 - M_Total;
1061
1062      % Total force balance on rocker
1063      F_Total = Spring_Force - RC_Force - PR_Force;
1064      res(31:33) = 0 - F_Total;
1065

```

Figure 42: Moment and force balance for rocker as it has more than 2 connections to other members

The remainder of the code (which can be found in appendix A.1) processes and plots the outputs, which are presented in the following section.

8 Results

As detailed in report VD-01, the kinematic behaviour of the suspension is one of the most important determinants of car performance, and achieving the right balance of castor, camber and toe requires careful selection of both inboard and outboard hard-points. The pipeline developed in this work is intended to inform that process by quantifying how the installed kinematics differ from the rigid-body design targets under realistic loading.

8.1 Validation

Validating the MATLAB model is a prerequisite for any credible conclusions to be drawn. The most direct route would have been to compare its outputs against the full-assembly Ansys simulation. As noted in section 6, however, this model was found to take approximately 20 hours to complete a single load case. As an alternative, a suspension load spreadsheet developed by the TBRe team was used as a cross-check on the loads predicted by the MATLAB solver. The spreadsheet uses rigid-body assumptions, so an exact match with the compliant model is not expected, but it provides a useful indication that the solver is operating in the correct region. The ideal validation route would be a Kinematics and Compliance rig, as discussed under future work, but that is not available to the team at the moment.

It should also be noted that as mentioned in section 6 some small changes did have to be made to the initial CAD geometry received from TBRe, which will likely contribute to slight discrepancies with theoretical results.

The starting point was to compare the static loads found in each member from the MATLAB program with those predicted in the suspension load sheet. The static condition is considered to be a vertical force of 1337N at each corner. The wishbone load sheet results are given in table 3.

Table 3: Suspension member forces from TBRe Suspension Load Analysis Spreadsheet for static load condition

Link Forces			Inboard Component Forces		
		Resultant	x	y	z
Up-Fore		-233.36	-101.09	208.90	-24.43
Up-Aft		-93.40	32.00	85.96	-17.63
Low-Fore		1126.45	383.55	-1059.05	13.68
Low-Aft		962.63	-305.27	-912.86	11.79
Push/Pull		-2102.78	1.16	1603.98	1359.75
Tie/Toe		-74.05	-10.35	73.08	-5.99

Then the loads in the members according to the MATLAB program are given in figure 43.

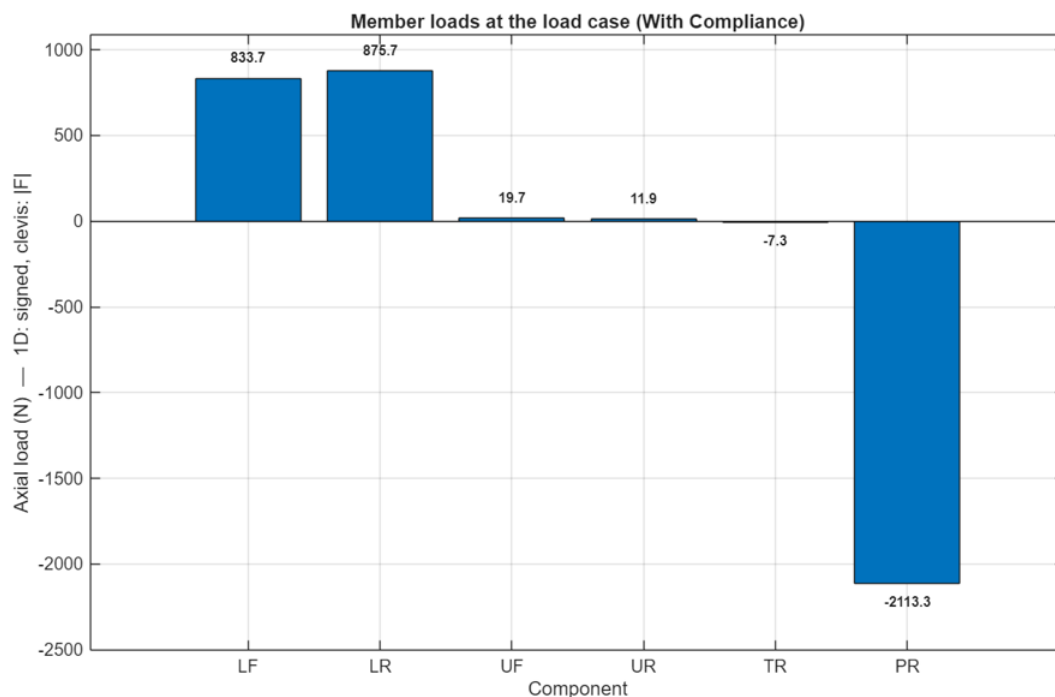


Figure 43: Loads according to MATLAB compliance solver for static load condition

What this shows is that the MATLAB solver closely correlates with the load sheet in the static state – the simulated push-rod force being within 11 N of the predicted – suggesting it correctly captures all the force balances and the geometries align sufficiently. The small variations such as the total of the lower wishbone compressive forces being ~300 N lower in MATLAB is likely down to the slight redistribution of forces due to compliance which is reflected by the corresponding 300 N decrease in tensile forces across the upper wishbones.

The static load case captures purely vertical loading, so a second case involving lateral loading was used to extend the validation before moving on to quantify the effects of compliance. This time the load case used from the wishbone spreadsheet was for steady state cornering and hitting a bump (1170N +y, 2883N +z). The results from the spreadsheet and MATLAB program are given in table 4 and figure 44 below.

Table 4: Suspension member forces from TBrE Suspension Load Analysis Spreadsheet for steady state cornering with bump load condition

Link Forces			Inboard Component Forces		
		Resultant	x	y	z
Up-Fore		-241.29	-104.53	216.00	-25.26
Up-Aft		194.65	-66.69	-179.14	36.74
Low-Fore		1329.02	452.53	-1249.50	16.14
Low-Aft		930.93	-295.21	-882.80	11.40
Push/Pull		-4387.30	2.42	3346.59	2837.03
Tie/Toe		82.13	11.48	-81.05	6.64

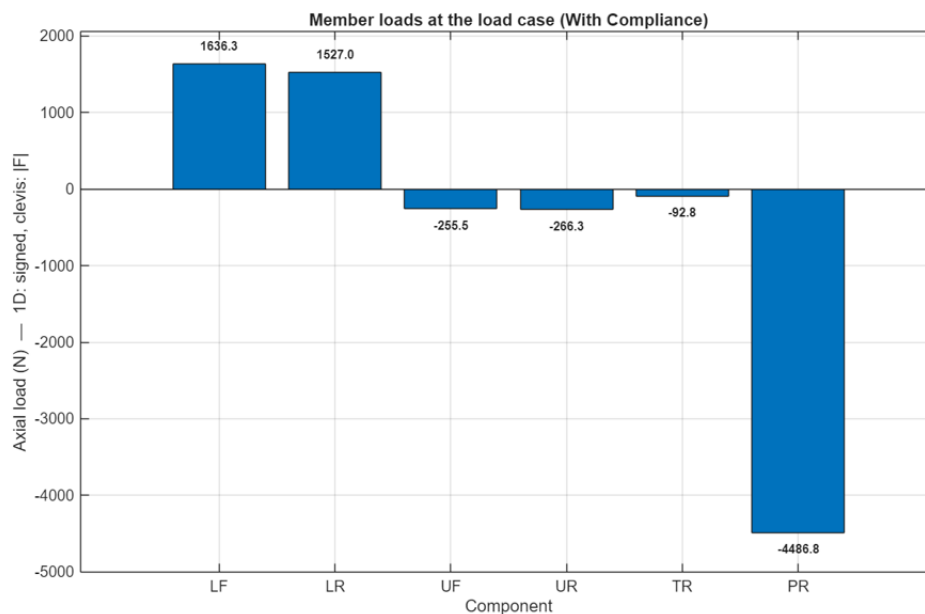


Figure 44: Loads according to MATLAB compliance solver for steady state cornering with bump load condition

The results again correlate closely – the push rod forces being within 100 N. The most notable difference is that the tie rod and upper aft wishbone show opposite signs in the two analyses. However, the lower aft wishbone carries a comparatively higher force which means the total load balance is preserved but its distribution has shifted slightly because of the introduced compliance. The agreement is sufficient to support the further analysis presented below; the question of more rigorous validation is returned to in the future work section.

8.2 Compliance through suspension travel

The next comparison considers how camber and toe vary through the wheel travel, evaluated against the theoretical prediction for the TBrE 26 car as documented in report VD-01. The theoretical graphs for TBrE 26 are given in figures 45 & 46. Note - all analysis will be conducted in degrees.

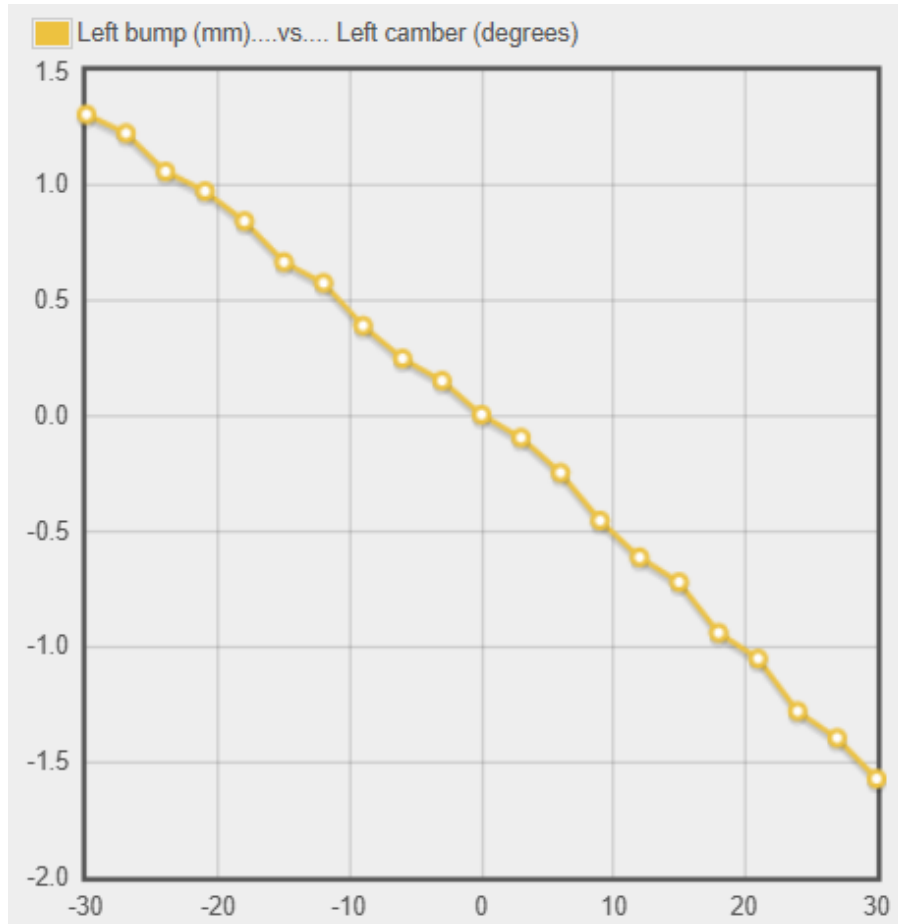


Figure 45: Graph of theoretical camber variation with wheel travel from report #

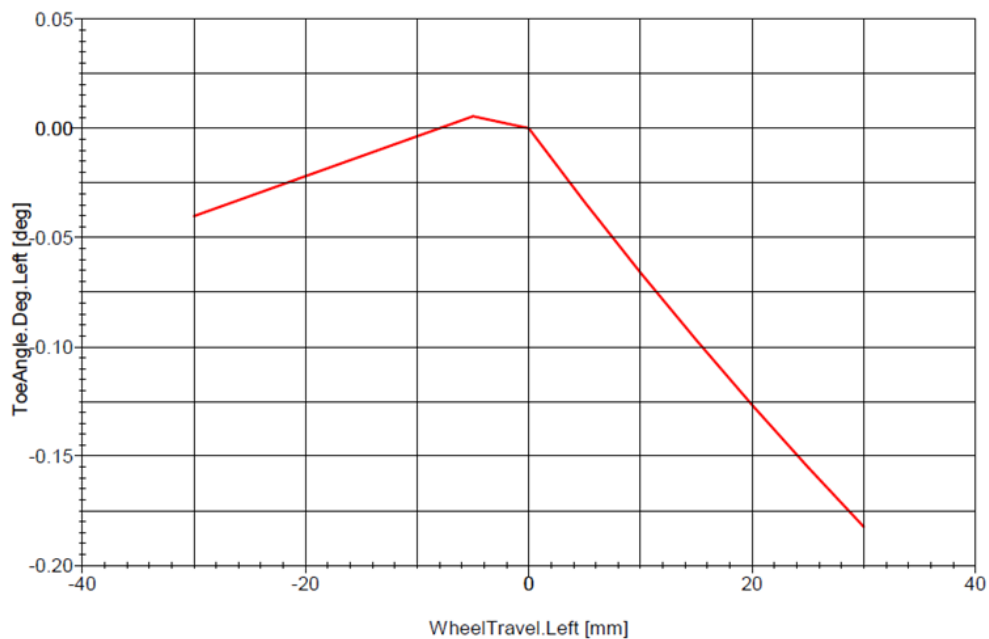


Figure 46: Graph of theoretical toe variation with wheel travel from report VD-01

The corresponding graphs obtained from MATLAB are shown in figure 47. It should be noted

that as the suspensions in MATLAB were defined at full droop the z-displacement values sweep from 0–60 mm for the full travel of the wheel. This corresponds exactly to the –30 to 30 mm sweep shown on the theoretical graphs.

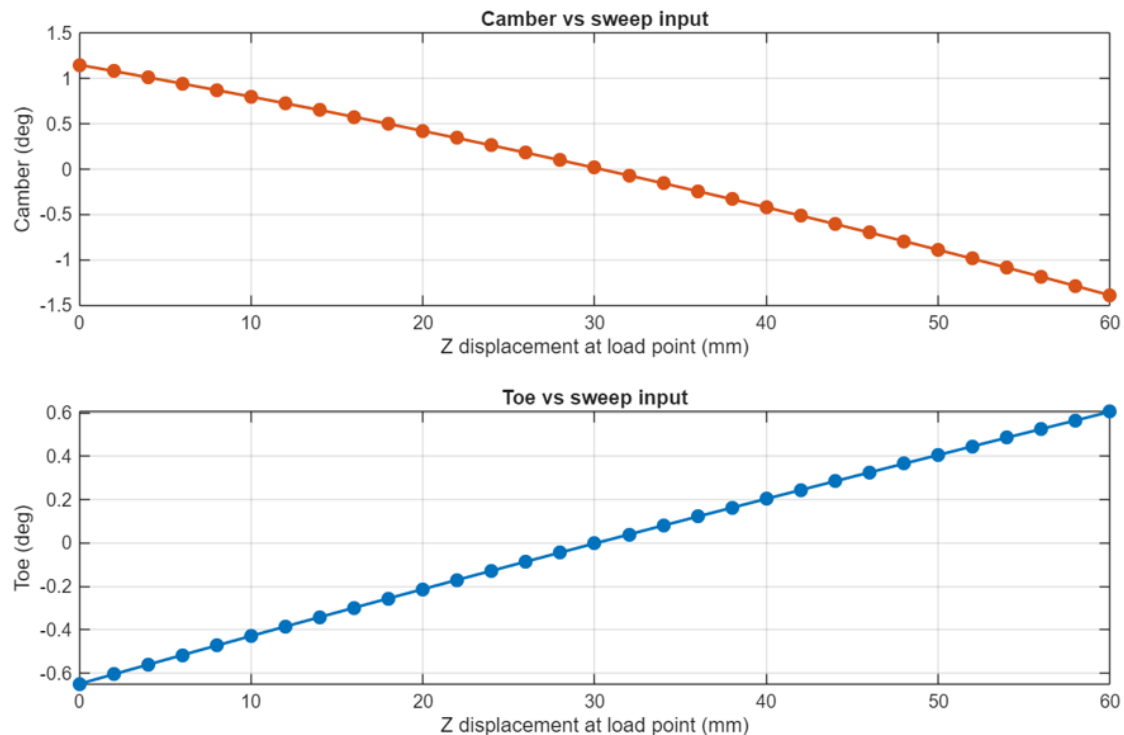


Figure 47: Camber and toe variation against wheel travel according to MATLAB compliance solver

The camber curve from the compliance model is very similar in form to the theoretical prediction: an approximately straight line crossing through the origin, with greater magnitude in bump than in droop. The range is slightly compressed, running from approximately 1.2 to –1.4 degrees rather than the theoretical 1.3 to –1.6 degrees. An early indication of the effect of compliance.

The toe curve, by contrast, shows a more pronounced deviation. As the wheel is loaded and swept through its travel, an unexpected toe-in develops, with the front of the tyres turning toward the centre of the car. The magnitude of this effect, around 0.6 degrees at the extremes of travel, is larger than the theoretical value of ~ 0.02 degrees, but the order of magnitude becomes plausible when the close spacing of the constraining hard-points on the upright is considered. The x-distance between the tie-rod mounting and the average location of the two wishbone attachments is approximately 70 mm, and applying basic trigonometry shows that to achieve an angular change of 0.6 degrees a relative y-displacement of only 0.73 mm is required between these points.

This is reflected in the graph of component deflection at maximum bump which shows a deflection in the lower wishbones of approximately 0.65 mm in figure 48.

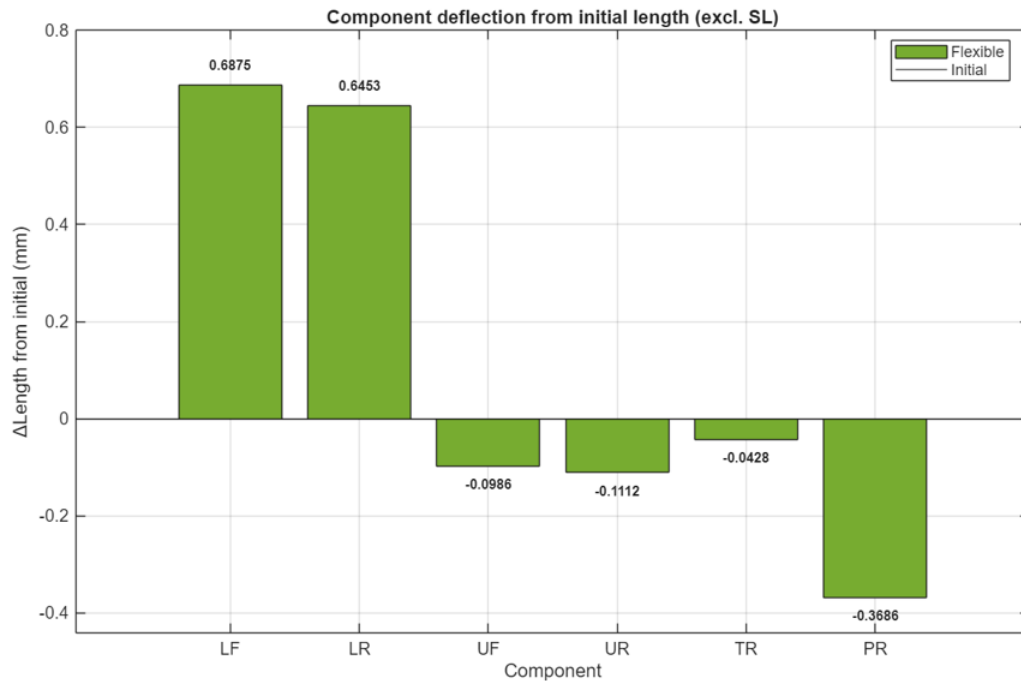


Figure 48: Suspension tubes deflection at maximum bump (60mm)

This value is close to the expected deflection magnitude to result in the observed change in toe. This gives an initial suggestion that the toe of the wheel is much more sensitive to compliance than the camber.

In fact this effect is further compounded when observing deflections in all the clevises too. Figure 49 shows the loads in the suspension members from the sweep used to produce figure 47.

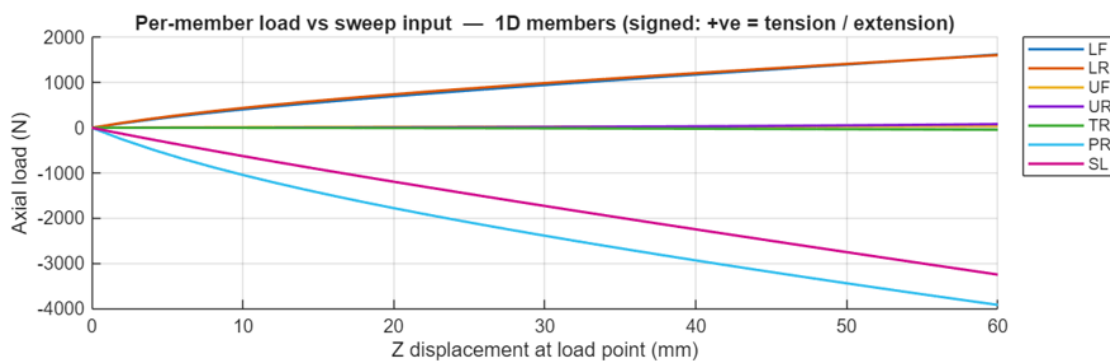


Figure 49: Sweep of suspension tube loads against wheel travel

A significant tensile force is observed on the lower rear wishbone (corresponding with the extension of the wishbones shown in figure 49), indicating that this member is pulling its clevis away from the chassis. The corresponding clevis attachment displacement exceeds 0.1 mm in magnitude as shown in figure 50.

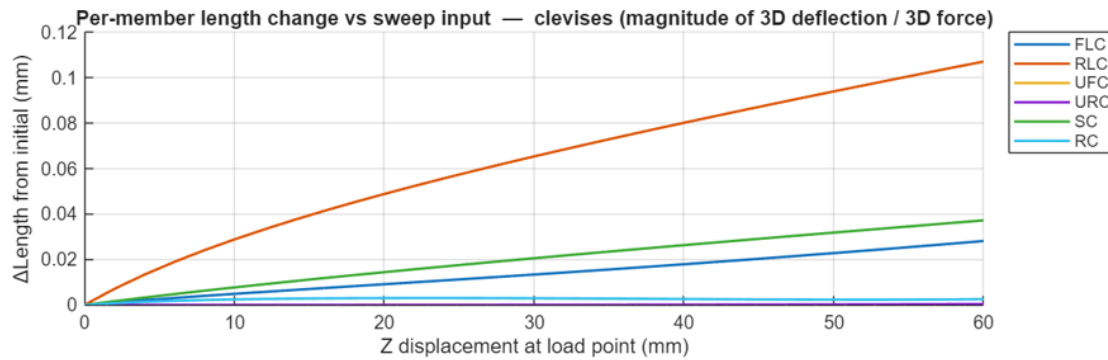


Figure 50: Sweep of clevis deflections (magnitude) against wheel travel

Added to the 0.65 mm deflection of the wishbones and the value approaches the predicted 0.73 mm. Some of these displacements will compete and cancel out in their contribution to toe, but the principle still stands: combining similar displacements across all the components that constrain the upright shows how readily the cumulative effect of small compliances produces a measurable change in wheel attitude.

Since the static toe of TBRe26 has been set to 0 deg at mid-travel, it is to be expected that the modelled toe crosses zero at mid-travel. As load increases and components such as the rear lower clevis deflect further, the toe continues to drift toward toe-out. Minimising the change in toe with bump is a standard objective of kinematic design, and the compliance results indicate that this is the suspension characteristic most affected by the present compliance distribution.

8.3 Lateral load cases

Where vertical loading is reacted predominantly through the push-rod and spring, lateral loading is supported almost entirely by the “rigid” members, and so lateral cases are also important when investigating the effect of compliance.

The first lateral case applies a 3,000 N lateral force at mid-travel, slightly higher than the maximum value calculated in the wishbone load sheet of 2,750 N. First the graph with no lateral force is given in figure 51 for comparison purposes.

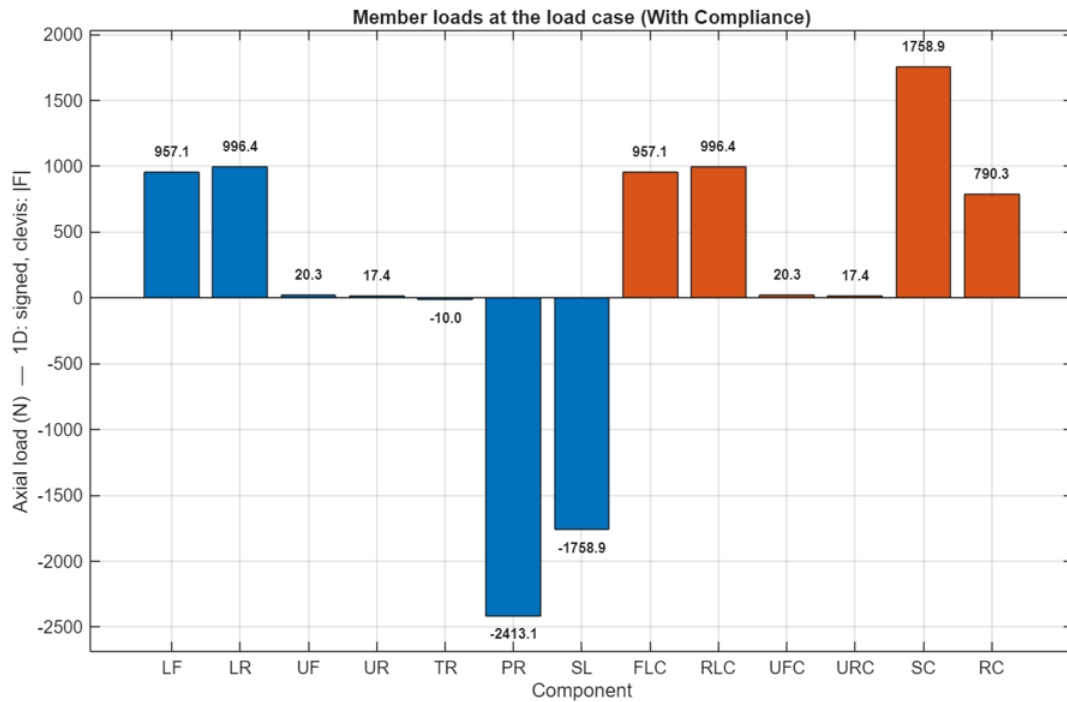


Figure 51: Loads in all suspension components at mid-travel with no lateral force

Then figure 52 shows loads with a 3,000 N lateral force in the positive-y direction.

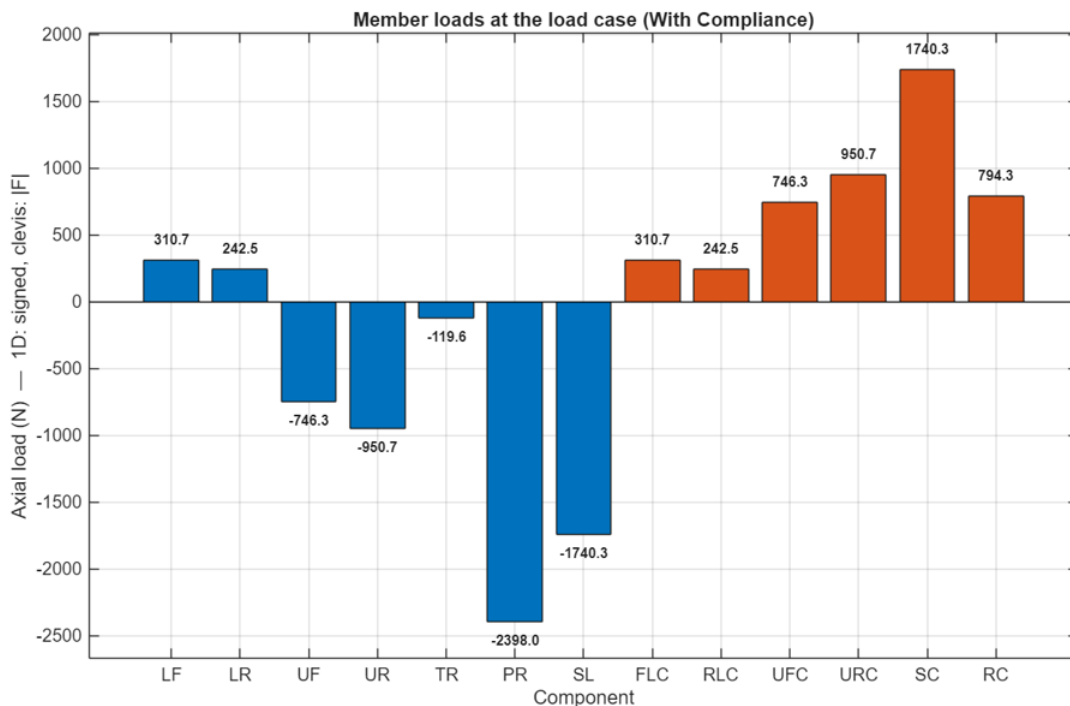


Figure 52: Loads in all suspension components at mid-travel with 3,000N lateral force in positive-y direction

As expected, the upper wishbones now carry significantly more load – increasing by over 600 N each – while the lower wishbones actually see a reduction of similar amounts. The

previous tensile load in the lower wishbones existed primarily to react the lateral component induced by the push-rod; with a direct lateral load applied at the wheel, this internal reaction reduces, and the tension in the lower wishbones falls. Correspondingly the deflections in the suspension are given in figures 53 & 54 without and with load respectively.

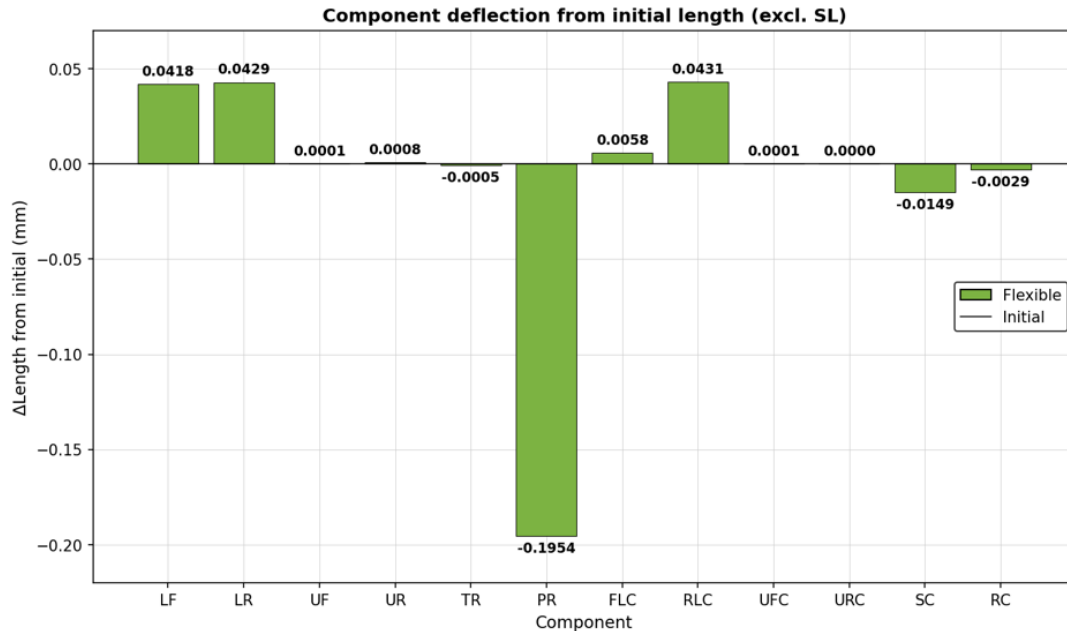


Figure 53: Deflection in all suspension components at mid-travel with no lateral force

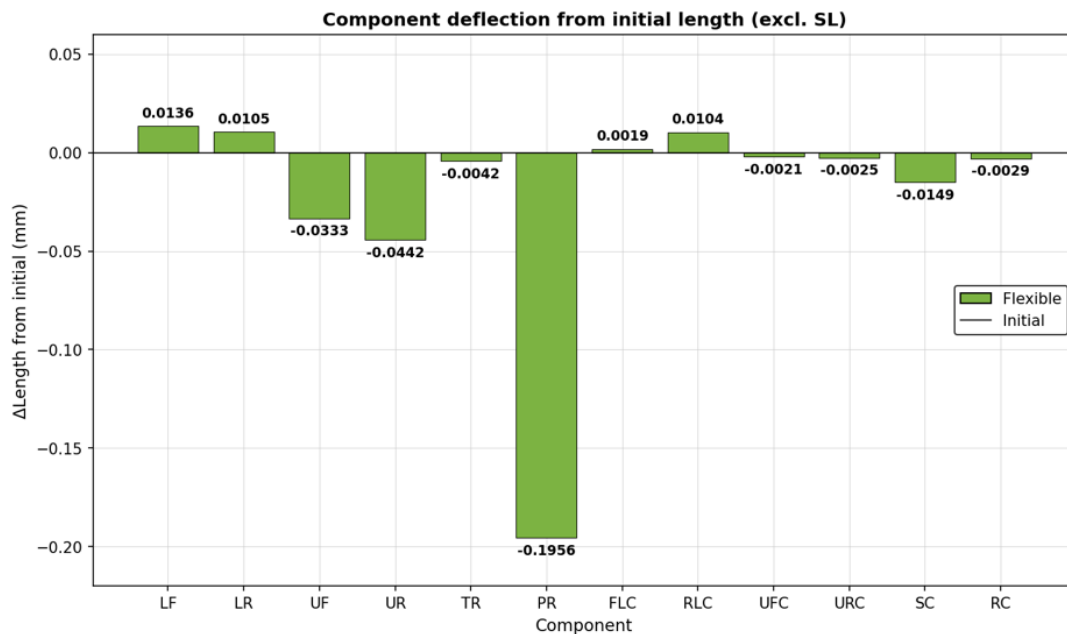


Figure 54: Deflection in all suspension components at mid-travel with 3,000N lateral force in positive-y direction

It is worth noting that as the rear lower wishbone is unloaded the rear lower clevis sees the biggest change in deflection – reducing by 0.03 mm, which seems a small change. But most important is not the displacements themselves but how they affect the wheel's toe and camber

which are given in table 5.

Table 5: Effect of lateral force on toe and camber at mid-travel

3,000 N lateral force in +y direction	No Load		With Load		Delta	
	Toe	Camber	Toe	Camber	Toe	Camber
Mid-travel (30 mm)	-0.002	0.016	0.026	0.018	+0.028	+0.002

The change in camber is negligible at 0.002 degrees whilst the change in toe is still very small approaching only 0.03 degrees. As noted before it is clear that there is a greater impact from compliance on toe than camber.

For additional verification two further load cases were examined: full bump and droop with 3,000 N lateral force, the results are given without and with the load in table 6.

Table 6: Effect of lateral force on toe and camber at extremes of travel

3,000 N lateral force in +y direction	No Load		With Load		Delta	
	Toe	Camber	Toe	Camber	Toe	Camber
Full droop (0 mm)	-0.65	1.15	-0.624	1.151	+0.026	+0.001
Full bump (60 mm)	0.61	-1.39	0.636	-1.387	+0.026	+0.003

Here were similar findings: very little effect on camber – changes of 0.001 deg and 0.003 deg – with toe showing the dominant response – changes of 0.026 deg. The loads in the different suspension components are given for the suspension in full droop with the 3,000 N lateral force in figure 55.

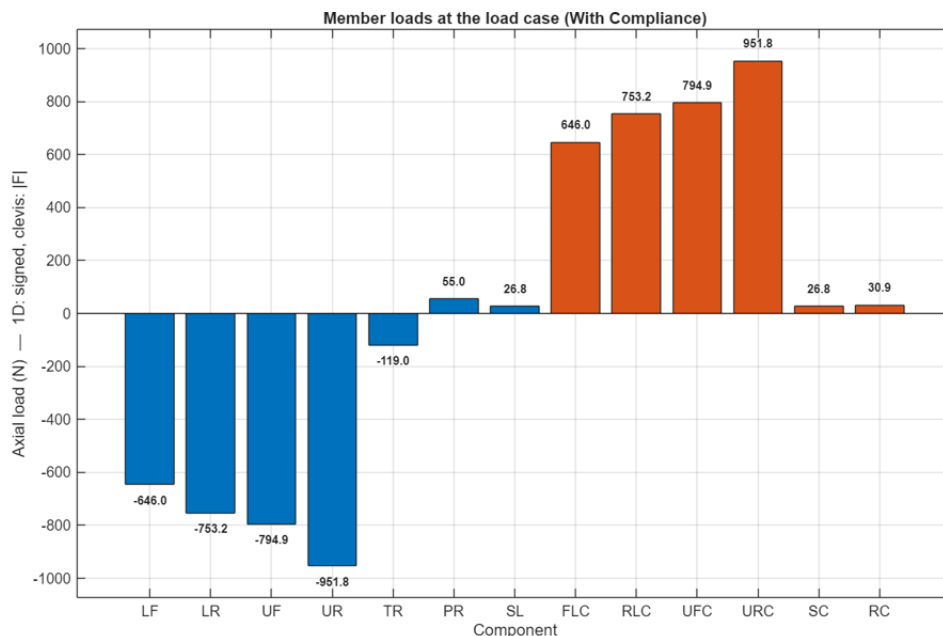


Figure 55: Loads in all suspension components at full droop with 3,000N lateral force in positive-y direction

It can be seen that the force in the tie rod is very low – approximately 85% lower than the average wishbone load – when a lateral force is applied because the load point from the wheel

is almost directly between the upper and lower wishbone attachment points on the upright. It is therefore unsurprising that toe is much more sensitive to lateral force as the wishbones are deflecting and the tie rod staying pretty much in the same place which means the upright is effectively rotating about the z-axis in the car coordinate frame.

Having analysed various different loading scenarios of the suspension and compared between the rigid and compliant set-ups, tables 7 & 8 summarise the findings.

Table 7: Variation of toe and camber with suspension travel

Load Case	Theoretical		Simulation		Delta	
	Toe	Camber	Toe	Camber	Toe	Camber
Full droop (0 mm)	-0.04	1.3	-0.65	1.15	-0.61	-0.15
Mid-travel (30 mm)	0	0	-0.002	0.016	-0.002	+0.016
Full bump (60 mm)	-0.18	-1.6	0.61	-1.39	+0.77	+0.21

Table 8: Variation of toe and camber with lateral force

3,000 N lateral force in +y direction	No Load		With Load		Delta	
	Toe	Camber	Toe	Camber	Toe	Camber
Full droop (0 mm)	-0.65	1.15	-0.624	1.151	+0.026	+0.001
Mid-travel (30 mm)	-0.002	0.016	0.026	0.018	+0.028	+0.002
Full bump (60 mm)	0.61	-1.39	0.636	-1.387	+0.026	+0.003

The largest effects on camber and toe were observed between the rigid-kinematics prediction and the compliance model through travel itself - a 0.77 deg change in toe and 0.21 deg change in camber being the peak variation from the theoretical - indicating that vertical load excitation is the most significant driver of compliance in this suspension configuration. Hence the maximum source of compliance is likely to be from the pushrod as shown in figures 53 & 54 where its length changes much more than other components.

8.4 Sensitivity to compliance and material choice

Quantifying compliance is only useful in so far as its effect on car performance can also be quantified. The sensitivity of lap time to the compliance ranges identified above was therefore assessed using the full vehicle model built in report VD-05. In parallel, a small set of material substitutions were performed, replacing the clevis material with a stiffer, heavier variant and with a lighter, more compliant variant, to assess the sensitivity of the kinematic outputs to these choices and inform the cost-performance trade-offs discussed in the next section.

Whilst the model from report VD-05 had been thoroughly validated when the sensitivity study was carried out it had not yet been used to optimise a baseline trim for the TBRe 26 car. Therefore, whether the changes made in the sensitivity study make the car faster/slower will be negated and the focus will instead be on the magnitude of the changes.

The main metrics used to quantify the changes in performance will be the lap time alongside the peak lateral and longitudinal accelerations all run around a model of the Formula Student Germany Endurance Circuit.

The baseline trim was run with the set-up displayed in table 9:

Table 9: Full Vehicle Model Baseline Trim

Camber		Toe		Mass (kg)
Front	Rear	Front	Rear	
-2.5	-1.5	0.25	0	275

Then the results of all the runs and parameter changes are given in table 10.

Table 10: Sensitivity study results

Run	Lap Time		Peak Ax+		Peak Ax-		Peak Ay	
	Value	Delta	Value	Delta	Value	Delta	Value	Delta
Baseline	82.188	-	0.68	-	-2.07	-	2.09	-
Toe +0.5	82.040	-0.148	0.67	-0.01	-2.06	+0.01	2.13	+0.04
Camber +0.3	82.220	+0.032	0.68	0	-2.07	0	2.09	0
Mass +0.7	82.308	+0.120	0.67	-0.01	-2.06	+0.01	2.08	-0.01

The changes in parameters were made in the full vehicle model by simply offsetting the static camber/toe by a given amount and this offset was then maintained through all the steering angles used in the model. The magnitudes chosen by which to change the parameters was based on the largest deltas in toe and camber from tables 7 and 8. Clearly, it is unrealistic that the car would ever be running at full bump or droop but under high cornering forces there is significant lateral load transfer which can result in the suspension moving a long way from its static position. This is therefore treated as a worst-case scenario.

First looking at the results for a 0.5 degree increase in toe angle front and rear: this gave a magnitude of change of 0.148 s. This is largely down to an increase in peak lateral acceleration of 0.04 g which clearly outweighs a slight decrease in max deceleration of 0.01 g.

In comparison changing camber has an impact on lap time of 0.032 s, whilst presenting no discernible change in either lateral or longitudinal peak acceleration. Clearly the car performance has a greater sensitivity to changes in toe angle (it appears 5 times as much as camber) and therefore it is important to control this within a tolerance of the theoretical optimum found in report VD-01.

In terms of how to deal with this compliance, the next question is whether the mass penalty of choosing a stiffer material to decrease the suspension compliance is beneficial in terms of the increased control of toe. Or whether going to a lighter and less stiff material is actually beneficial for lap time at the sacrifice of some wheel attitude control. Therefore, a sensitivity study was also run with a 0.7 kg mass increase which is approximately what the mass penalty would be for switching out the aluminium clevises to steel. The result gave a magnitude change of 0.120 s which is actually less than that experienced for the change in toe. Hence potentially the weight penalty of moving to steel clevises may be worthwhile if it can maintain the desired toe within a tighter window.

A case study is then performed below to utilise these sensitivity studies to understand the impact of changing the clevis materials to a much lighter and less stiff magnesium alloy or a much stiffer and heavier steel alloy. The materials used and their properties are given in table 11.

Table 11: Potential clevis material properties

Material	Young's Modulus	Density
AL 7075-T6	72 GPa	2.8 g/cm ³
17-4 PH Stainless Steel	197 GPa	7.8 g/cm ³
Magnesium Alloy (AZ80A-T5)	45 GPa	1.8 g/cm ³

The materials will be compared at max bump as from table 7 this seems to be the condition which most excites compliance modes. As before the focus will be on the relative performance of the different set-ups rather than their absolute values. Progressing from most to least compliant the plots of deflection in all the suspension members are given for the magnesium, aluminium and steel alloys in figures 56, 57 & 58 respectively.

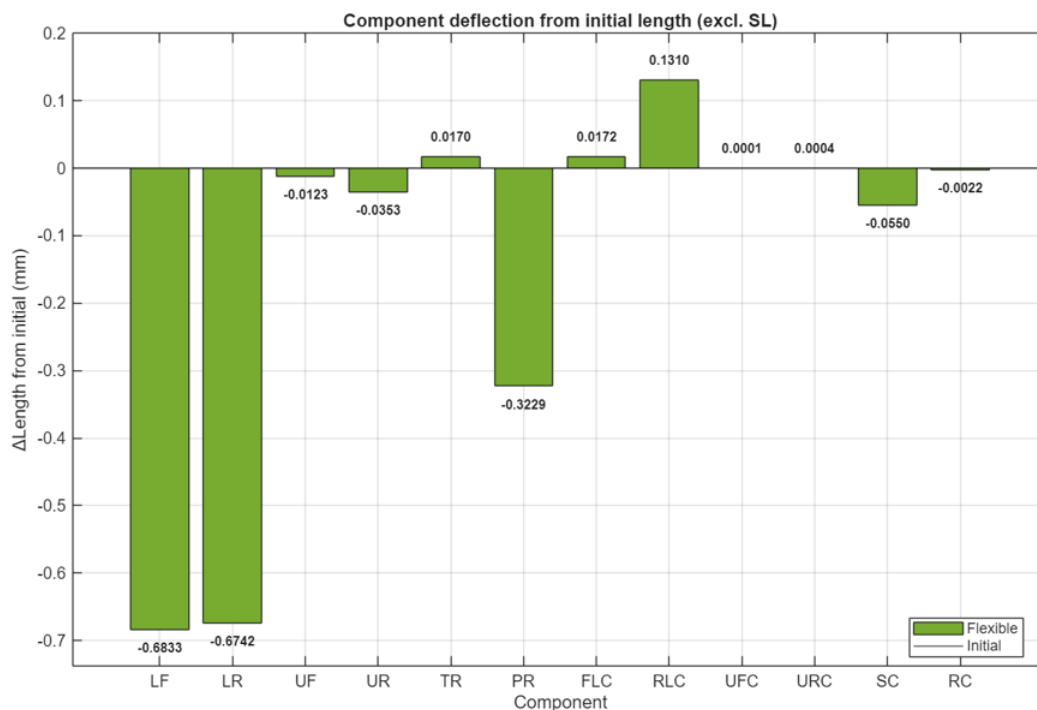


Figure 56: Deflection in all suspension components at maximum bump with magnesium clevises

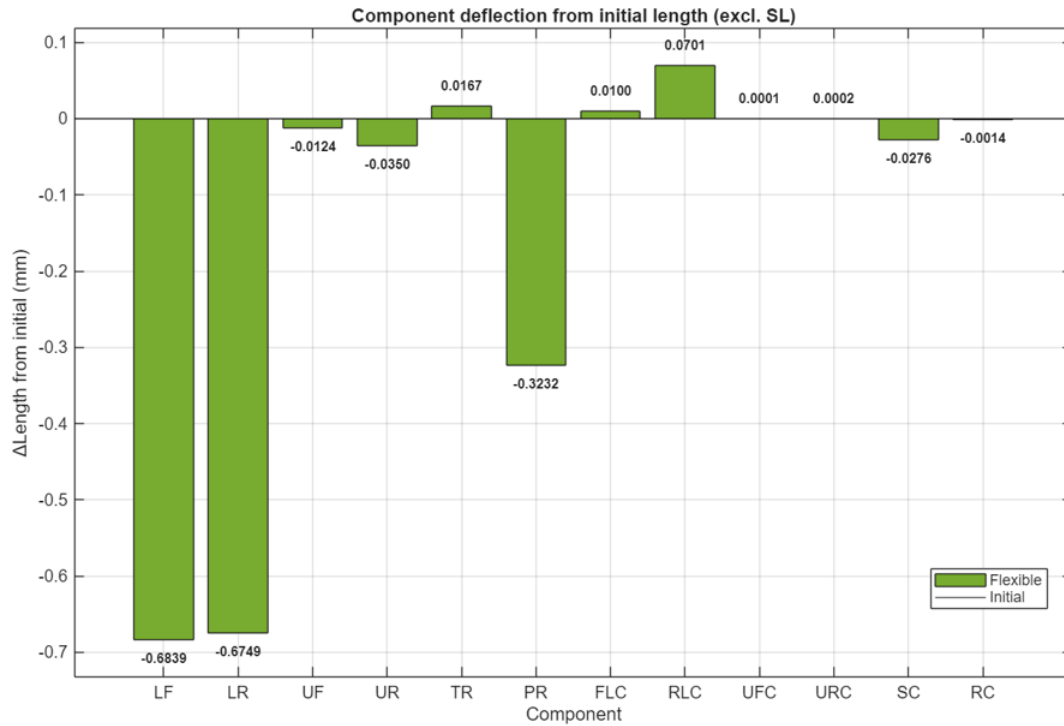


Figure 57: Deflection in all suspension components at maximum bump with aluminium clevises

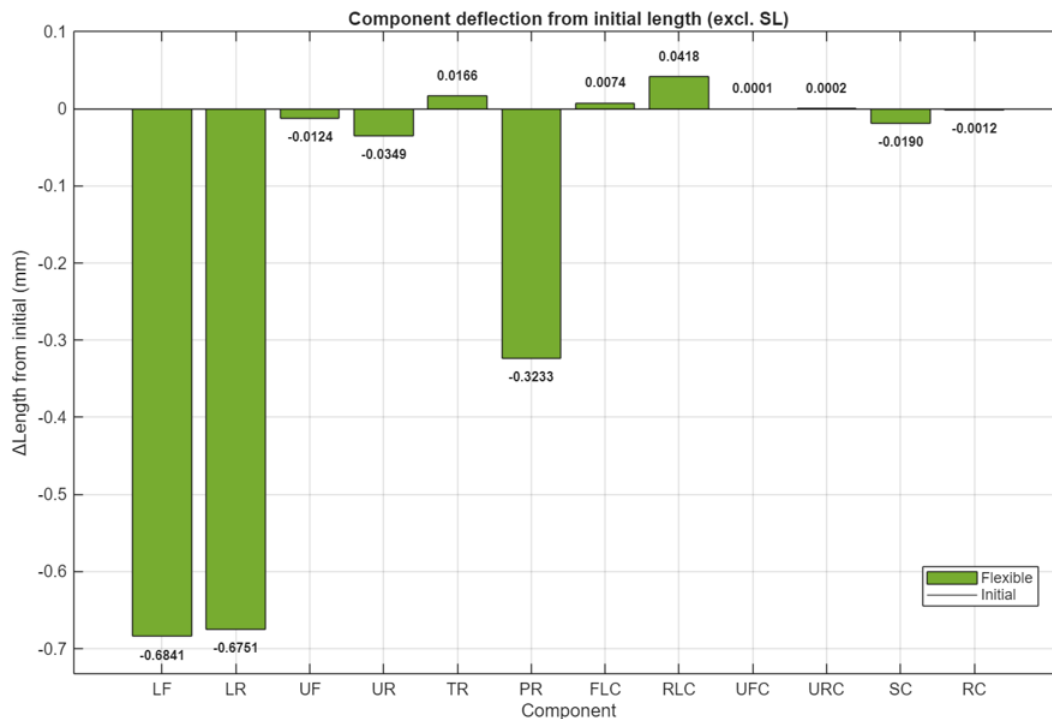


Figure 58: Deflection in all suspension components at maximum bump with steel clevises

And the effects on toe and camber are summarised in table 12.

Table 12: Effects of clevis material on toe and camber at maximum bump

Material	Toe		Camber		Total Mass	
	Value	Delta	Value	Delta	Value	Delta
Magnesium	0.870	−0.007	−1.139	−0.009	0.12	−0.06
Aluminium	0.877	–	−1.130	–	0.18	–
Steel	0.881	+0.004	−1.125	+0.005	0.50	+0.32

What this shows is for almost identical forces the clevis deflections decreased by approximately half going from aluminium to steel – the rear lower clevis going from 0.07 mm to 0.04 mm – and almost doubled going from aluminium to magnesium – the rear lower clevis going from 0.07 to 0.13 mm. This however has only a small effect on the toe and camber as shown in table 12. This is likely because as can be seen in the figures some suspension members, such as the lower wishbones and pushrod have deflections of up to 0.68 mm. This is an order of magnitude higher than those of the clevises and so are the major contributor to the toe and camber angles. In this respect the 60 g mass saving of switching from aluminium to magnesium clevises could well be worthwhile if they are having little impact on compliance.

This theory was put to test with a case study to try and minimise compliance whilst maximising performance. To improve the wheel attitude control the outer diameter of the carbon tubes was increased by 6.75 mm while maintaining the same wall thickness to give an effective 50% increase in stiffness. At the same time the mass of the system was reduced by switching from the aluminium to the magnesium clevises. The force and deflection graphs are given in figures 59 & 60.

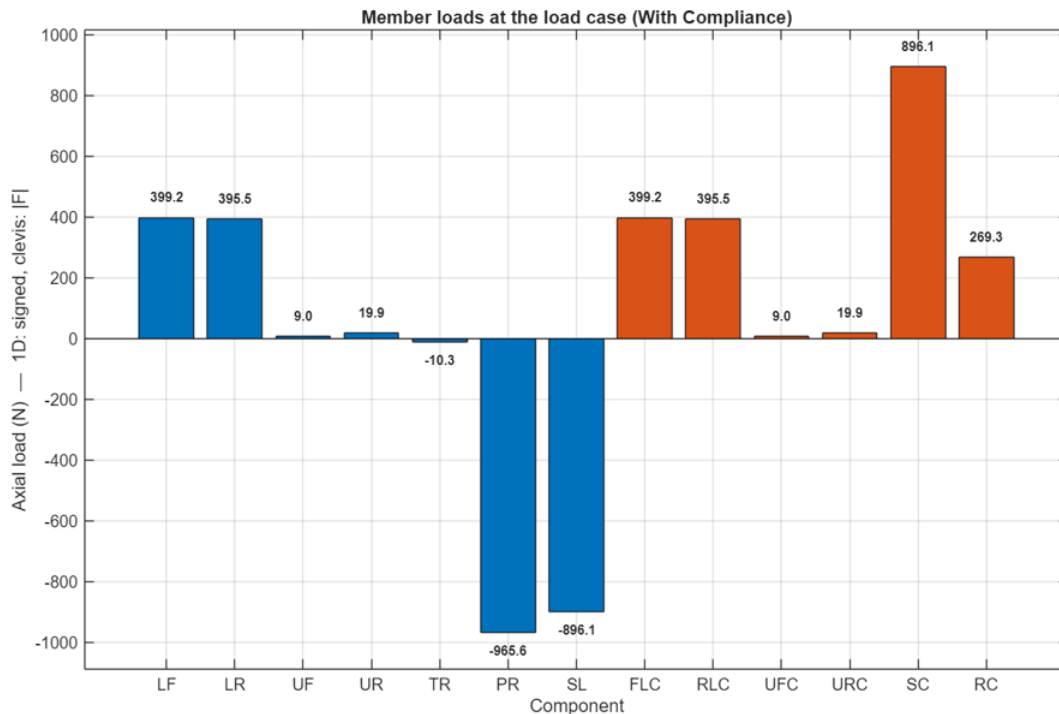


Figure 59: Loads in all suspension members with increased stiffness carbon tubes and magnesium clevises at maximum bump

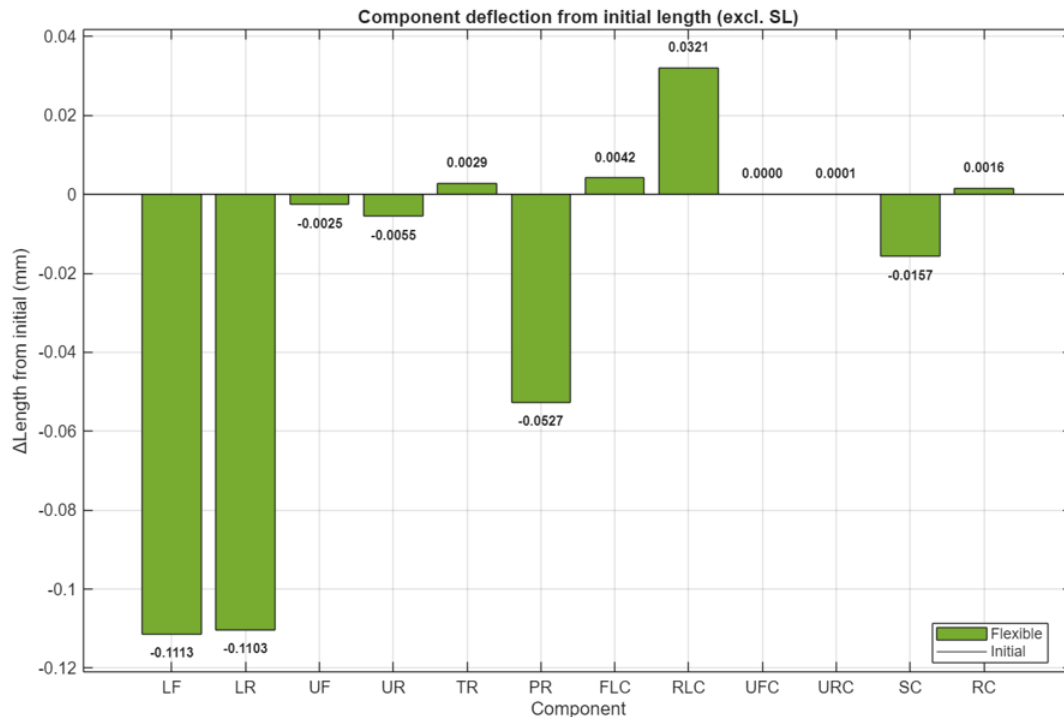


Figure 60: Deflections in all suspension members with increased stiffness carbon tubes and magnesium clevises at maximum bump

The resulting camber and toe values are given in table 13.

Table 13: Camber and toe values for increased stiffness carbon tubes and magnesium clevises at maximum bump

Camber	Toe
-1.501	0.483

As can be seen increasing the stiffness of the tubes reduces their deflection by a factor of 7 – from approximately 0.7 mm to 0.1 mm – which brings their order of magnitude much closer to that of the magnesium clevises. As mentioned earlier this highlights that the idea of quantifying compliance isn't necessarily to then reduce compliance everywhere but to change it judiciously to optimise performance.

These changes bring the toe and camber values much more in line with those in the theoretical graph laid out in figure 45. A change in camber of -0.111 deg from the original set-up in 7 and -0.127 deg in toe, both moving towards the theoretical values. Whilst the carbon tubes increase by 90g in weight, the clevises have reduced by 60g giving a net increase of 30g in mass.

If it is assumed that moving towards the theoretical camber and toe values is moving towards the optimum, the sensitivity study from table 10 can be used to convert these results into a change in lap time through linear interpolation - the change in each parameter in this case is divided by the change of the parameter in the sensitivity study, then multiplied by the change in lap time to give the following results for camber, toe and mass:

$$(0.111 \div 0.3) \times 0.032 = 0.012s \text{ decrease}$$

$$(0.127 \div 0.5) \times 0.148 = 0.038s \text{ decrease}$$

$$(0.03 \div 0.7) \times 0.12 = 0.005s \text{ increase}$$

This would give a total decrease in lap time of 0.045s which whilst small suggests the changes are a net benefit and demonstrates that the effects of compliance are quantifiable.

9 Solution Lifecycle

The pipeline developed in this work supports a more rigorous design process by quantifying compliance factors that would otherwise not be characterised until a physical prototype of the suspension had been built and run extensively. In the absence of a K&C rig, the conventional alternative is on-track running of the assembled car to infer compliance effects from observed kinematic behaviour, which is both expensive and time-consuming. The savings enabled by the digital pipeline are therefore both environmental and economic: less running of the car corresponds to fewer direct and indirect emissions, and the ability to iterate purely in software avoids the cost and material waste associated with physical prototypes that are quickly superseded.

The pipeline also establishes a consistent workflow for validating designs in FEA while collecting useful information for further design work. As noted in the introduction, current practice typically involves designing a part, checking that it passes FEA, signing it off, and then setting the FEA results aside. By generating the force-displacement maps as a routine output of the FEA, the simulation engineer's time becomes more effective: the same model that validates the absolute performance of the part also feeds back into the kinematic and dynamic design process. This also ensures that FEA is performed and recorded for all components in a structured manner, supporting design traceability and continuous improvement.

Costing is a recurring constraint for TBRe, which operates on a strict budget; design and performance ideals frequently have to be compromised on grounds of cost. For suspension components, more expensive materials generally correspond to higher stiffness and a better strength-to-weight ratio. The compliance tool makes it possible to identify when a component is significantly over-specified, or when a cheaper, lighter but more compliant material would be advantageous in the context of the full-vehicle sensitivity study. Costing decisions can then be made with engineering justification, maximising both performance and cost-effectiveness.

The MATLAB software has been written using a function-based modular structure. Each function can be developed and validated in isolation without affecting the rest of the script, supporting co-development by multiple team members. New generations of TBRe, or other clients, can adapt the software to new vehicles or new functionality without rewriting the core - which also give the tool scalable commercial potential. Since MATLAB was chosen specifically for its widespread use within the TBRe team, the resulting tool is accessible to all members and is expected to remain in use across multiple seasons.

10 Future Work

The pipeline as developed provides a workable method for quantifying compliance and feeding the results back into the kinematic design workflow. Several extensions of the work are

nevertheless worth highlighting, both to address current limitations and to broaden the tool's applicability.

The most immediate extension is to incorporate explicit kinematic design functionality into the same software. The compliance tool already produces graphs of camber and toe variation through wheel travel, which are precisely the curves that a kinematic engineer aims to shape when designing a suspension, as detailed in report VD-01. Rather than constraining the inboard hard-points to their as-built locations, as is done here, a parameter table could be utilised in which the engineer specifies allowable ranges for hard-point positions, track width, suspension member lengths, wheel travel and target camber and toe curves. These targets and constraints could then be passed as residuals to a non-linear solver, which would search for a feasible kinematic layout with quantified compliance.

The non-linear solver used in the present work locates only local minima. While this is acceptable for analysing a known suspension, it is unsuitable for full kinematic design, where many distinct hard-point arrangements may achieve similar residuals while favouring different characteristics. A weighting scheme would be required to ensure that the highest-priority constraints, such as physical or regulatory limits, are met, and a global-minimisation algorithm would be needed to avoid local minima. Methods such as genetic algorithms, simulated annealing or pattern search would be appropriate candidates.

Extending the compliance modelling itself is also a clear next step. As noted in section 6, the upright and rocker were treated as rigid in the present analysis because the per-component sweep required to map a three-node component scales as $27^3 = 19,683$ cases. With access to a more capable solver or a high-performance computing cluster, the rocker and upright maps could be generated, allowing their compliance to be incorporated into the assembly solve.

The most significant longer-term step would be the construction of a Kinematics and Compliance rig for physical validation. Physical testing is essential for validating any simulation tool, and is also a useful check on installation accuracy, since small geometric errors, for example in shim selection or clevis location, can be difficult to detect by inspection because the angles involved are small and depend on wheel travel. Commercial K&C rigs cost in the order of millions of pounds, but FS teams have been known to construct their own setups; the principal limitation is that FS cars are too small to fit on commercial rigs, so even short-term hire is generally not an option. A practical first step would be a quarter-car compliance rig, in which the suspension clevises bolt directly onto a small, braced backing structure and the wheel is moved through its travel by a combination of linear actuators and load-measuring elements such as strain gauges or string potentiometers. Such a rig would allow direct correlation against the simulation outputs across the full range of travel and applied lateral load.

11 Conclusion

The final data pipeline and associated tools had strict criteria to meet in terms of performing both a technical function as well as fulfilling various process targets to make it a useful tool for the client TBRe team.

As laid out in the design specification the tool needed to utilise FEA to provide an early iterative design aid as opposed to just a means of final validation. This was achieved by creating a modular program where each component could be updated individually without having to re-run a full ~ 20 hour Ansys simulation and the individual component data was stored as maps so that it could be easily called on when necessary.

This was achieved by a comprehensive study of the relative merits of non-linear solving meth-

ods in MATLAB and Ansys to create a hybrid system. Ansys performed the linear analysis of individual components with high fidelity models with a vast number of nodes and elements and advanced material properties. Then MATLAB excelled in the highly non-linear analysis with a much reduced number of elements and highly robust local optimisation algorithms.

At the same time this also significantly reduced the computational load which was another main target: the cumulative times compared to Ansys became around 20 minutes per component FEA sweep and 20 seconds to run the MATLAB program, so around 1 hour 40 minutes to run a full model from scratch, rather than over 20 hours when solving the whole model in Ansys.

By making the tool quick and easy to use it became accessible to those not normally accustomed with FEA software. Using a MATLAB interface to run the non-linear solver as well as post-process the data and produce graphs makes it easy to interact with by the TBRe team who are well accustomed to this software. Graphs such as 54 make it quick to identify the components which are most compliant and make design decisions accordingly. And by making the functions in MATLAB extremely modular this protects the future lifecycle of the tool as members of the team can repurpose it and create new post-processing features whilst maintaining the core functionality.

When using this tool TBRe's 2026 suspension system could be analysed to find its key characteristics and identify potential avenues for improvement. From table 10 it was found that out of toe and camber the lap time and hence performance of the car was almost 5 times more sensitive to changes in toe than camber. This was an important conclusion given that it was also found that toe was around 4 times more sensitive than camber to compliance according to table 7 meaning that real world values could stray far from theoretical targets.

Using these discoveries a case study was undertaken to aim to improve the suspension system's performance. This involved stiffening the tube members by increasing their OD whilst switching out the aluminium clevises for magnesium alloy ones. This brought all of the components into a similar window of deflection and was seen to dramatically improve the correlation between simulated and theoretical camber and toe values whilst bringing a theoretical FSG lap time improvement of almost 5 thousandths of a second.

What this highlighted is that knowing your compliance means you can be more judicious about how you make changes to the suspension. Just decreasing compliance everywhere doesn't necessarily increase performance but by stiffening the most compliant members and vice versa you can achieve a mass decrease and improved wheel attitude hence dramatically increasing the car's performance.

The hope is that as detailed in the future work section the simulation can be validated over the summer race season through either on-car data or by the construction of a customised K&C rig. This will allow the model to be improved further before being integrated into a wider kinematics design and optimisation package which can use global optimisation methods to test a wide range of hard points and other geometric and regulatory criteria to attain specified kinematic targets.

The tool also feeds into the work of the wider team in the design and analysis of TBRe27's suspension system with the metrics from report VD-01 giving targets around which to design the suspension and then in combination with this tool and the sensitivity studies from report VD-05 an iterative design process could be achieved.

Personal Reflections

As a team we had quite a broad scope to cover the analysis of a whole suspension system. Certainly early on this required a lot of discussion to ensure that our workstreams could continue in parallel and didn't create too much overlap between team members. My personal role was taking the TBRe 26 suspension metrics and the theoretical properties calculated in report VD-01 to then perform a compliance analysis and quantify how this affects the car's performance by utilising the model in report VD-05.

An early risk identified during the project was a circular dependency between the analysis pipeline and each other's report outputs, as each was required as an input to the other. This was resolved by using data from the TBRe 26 car to validate the underlying concepts of the tools, which can be readily applied to TBRe 27 in future.

Overall the team worked well together and achieved a comprehensive analysis of TBRe27's VD solution as laid out across all the technical reports.

References

- [1] Matt Slater. Olympics cycling: Marginal gains underpin team gb dominance. *BBC Sport*, 08 2012.
- [2] France proof that 'marginal gains' all important – mercedes, 2026.
- [3] Claude Rouelle. Claude rouelle's 101 tips for formula student competition participants | optimumg, 03 2018.
- [4] Pat and Steve Fox. Chassis stiffness and compliance, 12 2010.
- [5] Ab dynamics spmm plus — kinematics and compliance (kc) test equipment, 04 2026.
- [6] Andrew Rhead. *ME20023 Solid Mechanics 4*. University of Bath, 10 2023.
- [7] Position analysis of mechanisms, 2026.
- [8] MATLAB . Solution of 2-d nonlinear system, 2026.
- [9] MATLAB . Equation solving algorithms - matlab simulink, 2026.
- [10] M.L.A. Lourakis and A.A Argyros. Is levenberg-marquardt the most efficient optimization algorithm for implementing bundle adjustment? *CiteSeer X (The Pennsylvania State University)*, 01 2005.
- [11] D.J. Evans. *The mathematics of finite elements and applications: proceedings of the Brunel University conference of the Institute of Mathematics and Its Applications held in April 1972*, ed. Academic Press, 04 1972.
- [12] Tyler Ferris. 3 steps to improve fea models | ansys blog, 12 2020.
- [13] Rupak Biswas and Roger C. Strawn. Tetrahedral and hexahedral mesh adaptation for cfd problems. *Applied Numerical Mathematics Journal*, 06 1997.
- [14] Cadence CFD . Hexahedral mesh vs. tetrahedral: Comparing high-quality meshing.
- [15] Łukasz Skotny. Why is a triangular element stiffer?, 02 2019.
- [16] LEAP Australia. Introduction to ansys composites preppost (acp) - presentation - composites for fsae 1, 03 2018.
- [17] University of Washington . Dot product and normals to lines and planes.
- [18] Seong Han Kim and Kang In Lee. Wheel alignment of a suspension module unit using a laser module. *Sensors*, 20:1648, 03 2020.

A MATLAB Code

A.1 Main Compliance Solver Script

```

1 % COMPLIANCE TOOL
2
3 % Define initial positions
4
5 % Clevis bases
6 Front_Lower_Clevice_Base = [483.842; -111.348; -110];
7 Rear_Lower_Clevice_Base = [795.282; -109.646; -110];
8 Upper_Rear_Clevice_Base = [769.031; -235; -0.302];
9 Upper_Front_Clevice_Base = [510.353; -235; 23.977];
10 Susp_Clevice_Base = [644.48; -235; 211.2];
11
12 % Rocker
13 Rocker_Clevice_Base = [644.48; -235; 58.097];
14 Rocker_Clevice_Attachment = [644.48; -254; 44.347];
15 Rocker_Spring_Attachment = [644.48; -354; 42];
16 Rocker_Push_Rod_Attachment = [644.48; -348; 22];
17
18 % Lower Wishbone
19 Front_Lower_Wishbone_Attachment = [504.103; -167.293; -123];
20 Rear_Lower_Wishbone_Attachment = [775.546; -165.777; -123];
21 Upright_Lower = [647.373; -555.35; -148.904];
22
23 % Upper Wishbone
24 Front_Upper_Wishbone_Attachment = [524.103; -252; 23.977];
25 Rear_Upper_Wishbone_Attachment = [755.281; -252; -0.302];
26 Upright_Upper = [649.828; -525.472; 25.044];
27
28 % Tie Rod
29 Tie_Rod_Base = [512.603; -219; -38.023];
30 Upright_Tie_Rod = [573.905; -549.497; -48.313];
31
32 % Suspension
33 Suspension_Attachment = [644.48; -252; 197.45];
34 Upright_Suspension_Mount = [647.179; -520.235; -113.993];
35
36 % Load Point
37 Load_Point = [645.85; -560; -51];
38
39 % Find local push rod attachment position
40 local_x = Rocker_Spring_Attachment - Rocker_Clevice_Attachment;
41 rocker_plane_1 = Rocker_Push_Rod_Attachment - Rocker_Spring_Attachment;
42 rocker_plane_2 = Rocker_Push_Rod_Attachment - Rocker_Clevice_Attachment;
43 rocker_normal = cross(rocker_plane_1, rocker_plane_2);
44 local_y = cross(local_x, rocker_normal);
45 local_z = cross(local_x, local_y);
46
47 % normalise local axis vectors
48 local_x = local_x / norm(local_x);
49 local_y = local_y / norm(local_y);
50 local_z = local_z / norm(local_z);
51
52 % concatenate local axis vectors to form rotation matrix
53 local_to_global = [local_x, local_y, local_z];
54
55 % transpose
56 global_to_local = local_to_global';
57

```



```

58 % find initial rocker local dimensions
59 global_push_pos = Rocker_Push_Rod_Attachment - Rocker_Clevice_Attachment;
60 global_spring_pos = Rocker_Spring_Attachment - Rocker_Clevice_Attachment;
61
62 local_push_pos Og = global_to_local * global_push_pos;
63 local_spring_pos Og = global_to_local * global_spring_pos;
64
65
66 % Clevis reference positions (used as the "undeformed" clevis attachment
67 % positions for the flexible solver)
68 LFC = Front_Lower_Wishbone_Attachment;
69 LRC = Rear_Lower_Wishbone_Attachment;
70 UFC = Front_Upper_Wishbone_Attachment;
71 URC = Rear_Upper_Wishbone_Attachment;
72 RC = Rocker_Clevice_Attachment;
73 SC = Suspension_Attachment;
74
75 % Lower Front Length
76 LF = norm(Upright_Lower - Front_Lower_Wishbone_Attachment);
77
78 % Lower Rear Length
79 LR = norm(Upright_Lower - Rear_Lower_Wishbone_Attachment);
80
81 % Upper Front Length
82 UF = norm(Upright_Upper - Front_Upper_Wishbone_Attachment);
83
84 % Upper Rear Length
85 UR = norm(Upright_Upper - Rear_Upper_Wishbone_Attachment);
86
87 % Tie Rod Length
88 TR = norm(Upright_Tie_Rod - Tie_Rod_Base);
89
90 % All upright dimensions
91 UD1 = norm(Upright_Lower - Upright_Upper);
92 UD2 = norm(Upright_Lower - Upright_Tie_Rod);
93 UD3 = norm(Upright_Lower - Upright_Suspension_Mount);
94 UD4 = norm(Upright_Upper - Upright_Tie_Rod);
95 UD5 = norm(Upright_Upper - Upright_Suspension_Mount);
96 UD6 = norm(Upright_Tie_Rod - Upright_Suspension_Mount);
97 UD7 = norm(Load_Point - Upright_Lower);
98 UD8 = norm(Load_Point - Upright_Upper);
99 UD9 = norm(Load_Point - Upright_Tie_Rod);
100
101 % For camber and toe use plane of upright with upper, lower and tie rod
102 % points
103
104 % first find normal to this plane as unit direction vector
105 vector_1 = Upright_Tie_Rod - Upright_Lower;
106 vector_2 = Upright_Tie_Rod - Upright_Upper;
107 normal_upright = cross(vector_1, vector_2);
108
109 % normal to tyre with no toe or camber
110 normal_to_standard_tire = [0;-1;0];
111
112 % correct camber and toe values at full droop
113 full_droop_camber = 1.15;
114 full_droop_toe = -0.65;
115
116 alpha = deg2rad(full_droop_camber);
117 gamma = deg2rad(full_droop_toe);
118
119 % Form rotation matrices
120 Rx = [1, 0, 0;

```

```

121     0, cos(alpha), -sin(alpha);
122     0, sin(alpha),  cos(alpha)];
123
124 Rz = [cos(gamma), -sin(gamma), 0;
125       sin(gamma),  cos(gamma), 0;
126       0,          0,          1];
127
128 % Combine both rotations into single matrix
129 R = Rz * Rx;
130
131 % Create and normalise vector normal to full droop wheel plane
132 normal_to_full_droop_tyre = R * normal_to_standard_tire;
133 unit_1 = normal_to_full_droop_tyre / norm(normal_to_full_droop_tyre);
134
135 % Find offset between upright plane normal and tyre plane normal
136 static_offset_rotation = ...
137     rot_between_vectors(normal_upright, normal_to_full_droop_tyre);
138
139 % Spring Length
140 SL = norm(Rocker_Spring_Attachment - Suspension_Attachment);
141
142 % Rocker Dimensions
143 PTS = norm(Rocker_Spring_Attachment - Rocker_Clevice_Attachment);
144 PTP = norm(Rocker_Push_Rod_Attachment - Rocker_Clevice_Attachment);
145 STP = norm(Rocker_Push_Rod_Attachment - Rocker_Spring_Attachment);
146
147 % Push Rod Length
148 PR = norm(Upright_Suspension_Mount - Rocker_Push_Rod_Attachment);
149
150
151 %%
152 % Load FEA force interpolants
153
154 % Load in 3D Ansys maps
155 interps = loadAllForceInterpolants({'Upright.csv', 'Upper_Clevice.csv', ...
156     'Lower_Clevice.csv', 'Rocker_Push.csv', 'Rocker_Spring.csv', ...
157     'Rocker_PivotfromPush.csv', 'Rocker_PivotfromSpring.csv', ...
158     'Rocker_PushfromSpring.csv', 'Rocker_SpringfromPush.csv'});
159
160 % Load in 1D Ansys maps
161 interps = ...
162     loadAll1DForceInterpolants({'Tube.csv', 'Push_Rod.csv'}, interps);
163
164
165 %%
166 % Pack geometry and dimensions into structs for the solver function
167
168 geom.FLW = Front_Lower_Wishbone_Attachment;
169 geom.RLW = Rear_Lower_Wishbone_Attachment;
170 geom.FUW = Front_Upper_Wishbone_Attachment;
171 geom.RUW = Rear_Upper_Wishbone_Attachment;
172 geom.TRB = Tie_Rod_Base;
173 geom.SB = Suspension_Attachment;
174 geom.RCA = Rocker_Clevice_Attachment;
175
176 dims.LF = LF; dims.LR = LR; dims.UF = UF; dims.UR = UR;
177 dims.TR = TR; dims.PR = PR; dims.SL = SL;
178 dims.PTS = PTS; dims.PTP = PTP; dims.STP = STP;
179 dims.UD1 = UD1; dims.UD2 = UD2; dims.UD3 = UD3;
180 dims.UD4 = UD4; dims.UD5 = UD5; dims.UD6 = UD6;
181 dims.UD7 = UD7; dims.UD8 = UD8; dims.UD9 = UD9;
182 dims.LFC = LFC; dims.LRC = LRC; dims.UFC = UFC; dims.URC = URC;
183 dims.RC = RC; dims.SC = SC;

```

```

184 dims.local_push_pos_og    = local_push_pos_og;
185 dims.local_spring_pos_og  = local_spring_pos_og;
186
187
188 %%
189 % USER INPUTS
190
191 % true  => force is known on that axis (value in CP_Load is a force in N)
192 % false => displacement is known on that axis (value in CP_Load is a
193 %         displacement relative to the static Load_Point, in mm)
194 Known_Force = [true, true, true];
195
196 % In 'sweep' mode the swept axis's value here is the baseline and gets
197 % overwritten by the sweep values.
198 CP_Load = [0; 0; 1337];
199
200 mode = 'single';    % 'single' or 'sweep'
201
202
203 %%
204 % PLOT FILTER      pick which components appear on the per-member plots
205
206 % Set to 'all' (or {}) to show every component, or list any subset of:
207 %   {'LF','LR','UF','UR','TR','PR','SL','FLC','RLC','UFC','URC','SC','RC'}
208
209 components_to_show = {'LF','LR','UF','UR','TR','PR'};
210
211
212 % just for full graphs of suspension geometries
213 % list any subset of: {'Initial','Rigid','With Compliance'}.
214
215 setups_to_show = 'all';
216
217
218
219 %%
220 % MODE
221
222 switch mode
223
224 case 'single'
225     % Convert delta-displacement inputs into absolute coordinates that the
226     % residuals expect
227     CP_Load_abs = CP_Load;
228     for i = 1:3
229         if ~Known_Force(i)
230             CP_Load_abs(i) = CP_Load(i) + Load_Point(i);
231         end
232     end
233
234     % call function that mediates with fsolve
235     [sol1, sol2, toe, camber, F_resolved, LP_resolved] = ...
236         solve_suspension(Known_Force, CP_Load_abs, geom, dims, interps, ...
237             static_offset_rotation, [], [], true);
238
239     % store data to plot
240     sol1_plot = sol1;  sol1_plot(:,7) = LP_resolved;
241     sol2_plot = sol2;  sol2_plot(:,7) = LP_resolved;
242
243     % Call plotting functions
244     plot_suspension(Front_Lower_Wishbone_Attachment,
245         Rear_Lower_Wishbone_Attachment, ...
246         Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...

```

```

246     Tie_Rod_Base, Suspension_Attachment, ...
247     Upright_Lower, Upright_Upper, Upright_Tie_Rod, Upright_Suspension_Mount,
248     ...
249     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
250     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
251     Susp_Clevice_Base, Rocker_Clevice_Base, ...
252     Rocker_Clevice_Attachment, Rocker_Spring_Attachment,
253     Rocker_Push_Rod_Attachment, ...
254     Load_Point, sol1_plot, sol2_plot, setups_to_show);
255
256 plot_component_lengths(Front_Lower_Wishbone_Attachment,
257     Rear_Lower_Wishbone_Attachment, ...
258     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
259     Tie_Rod_Base, Suspension_Attachment, ...
260     Upright_Lower, Upright_Upper, Upright_Tie_Rod, Upright_Suspension_Mount,
261     ...
262     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
263     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
264     Susp_Clevice_Base, Rocker_Clevice_Base, ...
265     Rocker_Clevice_Attachment, Rocker_Spring_Attachment,
266     Rocker_Push_Rod_Attachment, ...
267     sol1_plot, sol2_plot, components_to_show);
268
269 % Bar chart of axial loads in each member (from sol2)
270 [~, loads] = compute_member_loads_and_dL(sol2, geom, dims, interps);
271 plot_component_loads_bar(loads, components_to_show);
272
273 fprintf('Toe:      %7.3f deg\n', toe);
274 fprintf('Camber: %7.3f deg\n', camber);
275 fprintf('Resolved load point (mm): [%3f %3f %3f]\n', LP_resolved);
276 fprintf('Resolved load force (N) : [%3f %3f %3f]\n', F_resolved);
277
278
279 case 'sweep'
280     % Pick which axis to sweep and the range
281     sweep_axis = 3; % 1=X, 2=Y, 3=Z
282     sweep_deltas = linspace(0, 60, 31); % values from CP_Load
283
284     % Build the baseline absolute CP_Load from the user's inputs
285     CP_Load_abs_base = CP_Load;
286     for i = 1:3
287         if ~Known_Force(i)
288             CP_Load_abs_base(i) = CP_Load(i) + Load_Point(i);
289         end
290     end
291
292     % Convert sweep values into the absolute convention if the swept axis
293     % is a displacement; leave them alone if it's a force.
294     if Known_Force(sweep_axis)
295         % forces in N
296         sweep_values_abs = sweep_deltas;
297     else
298         % abs coord
299         sweep_values_abs = sweep_deltas + Load_Point(sweep_axis);
300     end
301
302     % call fsolve mediating function
303     results = sweep_suspension(sweep_axis, sweep_values_abs, ...
304         Known_Force, CP_Load_abs_base, ...
305         geom, dims, interps, static_offset_rotation, false);
306
307     % Shift the x-axis on the plots back to "delta from static" so the
308     % graphs match what the user typed in.

```

```

304     if ~Known_Force(sweep_axis)
305         results.values = results.values - Load_Point(sweep_axis);
306     end
307     plot_sweep_results(results, Known_Force, components_to_show);
308
309 end
310
311 %%
312 % FUNCTIONS
313
314
315
316 function [sol1, sol2, toe, camber, CP_Load_resolved, LoadPoint_resolved, dL,
        loads] = ...
317     solve_suspension(Known_Force, CP_Load_input, ...
318                     geom, dims, interps, static_offset_rotation, ...
319                     sol1_warm, sol2_warm, verbose)
320 % SOLVE_SUSPENSION Run the rigid + flexible solves for one load case.
321 %
322 % Inputs:
323 %   Known_Force      1x3 logical, true => force is known on that axis
324 %                   false => displacement is known on that axis
325 %   CP_Load_input    3x1, per-axis value: a force (N) where Known_Force(i)=true,
326 %                   or an *absolute* displacement coordinate (mm) where false.
327 %                   NOTE: this matches the convention the residual functions
328 %                   expect caller is responsible for adding any Load_Point
329 %                   offset before passing in.
330 %   geom              struct of geometry points (chassis-side + initial upright)
331 %   dims              struct of pre-computed lengths and reference frame data
332 %   interps           loaded force interpolants
333 %   static_offset_rotation 3x3 calibration rotation
334 %   sol1_warm, sol2_warm optional warm-start matrices ([] => cold start)
335 %   verbose           true/false for fsolve display
336 %
337 % Outputs:
338 %   sol1, sol2        raw solver outputs (column 7 may contain a force in the
339 %                   axes where Known_Force is false)
340 %   toe, camber        degrees, computed from sol2's upright points
341 %   CP_Load_resolved   3x1 actual force applied at the load point
342 %   LoadPoint_resolved 3x1 actual physical load-point coordinates
343
344     if nargin < 9, verbose = false; end
345     if nargin < 8, sol2_warm = []; end
346     if nargin < 7, sol1_warm = []; end
347
348     if verbose, disp_mode = 'iter'; else, disp_mode = 'off'; end
349     options = optimoptions('fsolve', ...
350         'MaxFunctionEvaluations', 100000, ...
351         'MaxIterations', 10000, ...
352         'Display', disp_mode);
353
354     % --- Build initial guesses ---
355     Locations_base = [647,649,573,647,644,644;...
356         -555,-525,-549,-520,-354,-348;...
357         -148,25,-48,-113,42,22];
358     Locations_2    = [483,795,769,510,644,644;...
359         -111,-109,-235,-235,-235,-254;...
360         -110,-110,0,24,211,44];
361
362     if isempty(sol1_warm)
363         % Column 7: position-guess where displacement is unknown,
364         %         zero (force) guess where displacement is known.
365         guesses = [0;0;0];

```

```

366     if Known_Force(1), guesses(1) = 645; end
367     if Known_Force(2), guesses(2) = -560; end
368     if Known_Force(3), guesses(3) = -51; end
369     Locations_k = [Locations_base, guesses];
370
371 else
372     Locations_k = sol1_warm;
373 end
374
375 % --- Rigid solve ---
376 sol1 = fsolve(@(x) Spring_Residuals(x, ...
377     geom.FLW, geom.RLW, geom.FUW, geom.RUW, ...
378     geom.TRB, geom.SB, geom.RCA, ...
379     dims.LF, dims.LR, dims.UF, dims.UR, dims.TR, dims.PR, ...
380     dims.UD1, dims.UD2, dims.UD3, dims.UD4, dims.UD5, ...
381     dims.UD6, dims.UD7, dims.UD8, dims.UD9, ...
382     dims.SL, dims.PTS, dims.PTP, dims.STP, ...
383     Known_Force, CP_Load_input), Locations_k, options);
384
385 % --- Flexible solve ---
386 if isempty(sol2_warm)
387     unknowns_k = [sol1, Locations_2];
388 else
389     unknowns_k = sol2_warm;
390     unknowns_k(:,1:7) = sol1; % refresh from the latest rigid solve
391 end
392
393 sol2 = fsolve(@(x) Susp_Residuals(x, geom.TRB, ...
394     dims.local_push_pos_og, dims.local_spring_pos_og, ...
395     dims.LFC, dims.LRC, dims.UFC, dims.URC, dims.RC, dims.SC, ...
396     dims.PR, dims.LF, dims.LR, dims.UF, dims.UR, dims.TR, ...
397     dims.UD1, dims.UD2, dims.UD3, dims.UD4, dims.UD5, ...
398     dims.UD6, dims.UD7, dims.UD8, dims.UD9, ...
399     dims.SL, dims.PTS, dims.PTP, dims.STP, ...
400     Known_Force, CP_Load_input, interps), unknowns_k, options);
401
402 % --- Reconstruct physical load point and applied force ---
403 LoadPoint_resolved = sol2(:,7);
404 CP_Load_resolved = CP_Load_input;
405 for i = 1:3
406     if ~Known_Force(i)
407         CP_Load_resolved(i) = sol2(i,7);
408         LoadPoint_resolved(i) = CP_Load_input(i);
409     end
410 end
411
412 % --- Toe and camber ---
413 [toe, camber] = compute_toe_camber(sol2(:,1), sol2(:,2), sol2(:,3), ...
414     static_offset_rotation);
415
416 % --- Per-member length change and load ---
417 [dL, loads] = compute_member_loads_and_dL(sol2, geom, dims, interps);
418
419 end
420
421 function results = sweep_suspension(sweep_axis, sweep_values, ...
422     Known_Force, CP_Load_base, ...
423     geom, dims, interps, ...
424     static_offset_rotation, plot_results)
425 % SWEEP_SUSPENSION Sweep one axis of CP_Load while holding the others fixed.
426 %
427 % Inputs:
428 % sweep_axis    1, 2, or 3      which axis to sweep
429 % sweep_values  vector of values for that axis (forces or absolute

```



```

429 % displacement coordinates, depending on Known_Force(sweep_axis))
430 % Known_Force 1x3 logical for the run
431 % CP_Load_base 3x1 baseline load (forces and/or absolute displacements
432 % in the convention the residuals expect)
433 % geom, dims, interps, static_offset_rotation : as for solve_suspension
434 % plot_results true/false (default true)
435 %
436 % Output:
437 % results struct with fields:
438 % .values sweep_values
439 % .toe vector of toe angles (deg)
440 % .camber vector of camber angles (deg)
441 % .CP_Load 3xN matrix of resolved forces
442 % .LP 3xN matrix of resolved load-point positions
443 % .axis sweep_axis
444
445 if nargin < 9, plot_results = true; end
446
447 n = numel(sweep_values);
448 results.values = sweep_values(:)';
449 results.toe = zeros(1,n);
450 results.camber = zeros(1,n);
451 results.CP_Load = zeros(3,n);
452 results.LP = zeros(3,n);
453 results.dL = zeros(13,n); % per-member length change (mm)
454 results.loads = zeros(13,n); % per-member axial load (N)
455 results.axis = sweep_axis;
456
457 sol1_prev = []; sol2_prev = [];
458
459 fprintf('\nSweeping axis %d over %d values...\n', sweep_axis, n);
460 for k = 1:n
461     CP_Load_k = CP_Load_base;
462     CP_Load_k(sweep_axis) = sweep_values(k);
463
464     try
465         [sol1_k, sol2_k, toe_k, camber_k, F_k, LP_k, dL_k, loads_k] = ...
466             solve_suspension(Known_Force, CP_Load_k, ...
467                 geom, dims, interps, static_offset_rotation, ...
468                 sol1_prev, sol2_prev, false);
469
470         results.toe(k) = toe_k;
471         results.camber(k) = camber_k;
472         results.CP_Load(:,k) = F_k;
473         results.LP(:,k) = LP_k;
474         results.dL(:,k) = dL_k;
475         results.loads(:,k) = loads_k;
476
477         sol1_prev = sol1_k;
478         sol2_prev = sol2_k;
479
480         fprintf(' step %2d/%d val = %+8.3f -> toe = %+7.4f, camber =
481             %+7.4f\n', ...
482                 k, n, sweep_values(k), toe_k, camber_k);
483     catch ME
484         warning('Step %d (val=%g) failed: %s', k, sweep_values(k), ME.message);
485         results.toe(k) = NaN;
486         results.camber(k) = NaN;
487         results.dL(:,k) = NaN;
488         results.loads(:,k) = NaN;
489         % Don't update warm starts on failure
490     end
491 end

```

```

491     if plot_results
492         plot_sweep_results(results, Known_Force);
493     end
494 end
495
496
497
498 function plot_sweep_results(results, Known_Force, components_to_show)
499     if nargin < 3, components_to_show = 'all'; end
500     axis_labels = {'X', 'Y', 'Z'};
501     if Known_Force(results.axis)
502         xlabel = sprintf('%s force at load point (N)', axis_labels{results.axis});
503     else
504         xlabel = sprintf('%s displacement at load point (mm)',
505             axis_labels{results.axis});
506     end
507
508     figure('Color','w');
509     subplot(2,1,1);
510     plot(results.values, results.camber, '-o', 'LineWidth', 1.6, ...
511         'Color', [0.85 0.325 0.098], 'MarkerFaceColor', [0.85 0.325 0.098]);
512     grid on; xlabel(xlabel); ylabel('Camber (deg)');
513     title('Camber vs sweep input');
514
515     subplot(2,1,2);
516     plot(results.values, results.toe, '-o', 'LineWidth', 1.6, ...
517         'Color', [0 0.447 0.741], 'MarkerFaceColor', [0 0.447 0.741]);
518     grid on; xlabel(xlabel); ylabel('Toe (deg)');
519     title('Toe vs sweep input');
520
521     % Gain plot (numerical derivative)
522     if numel(results.values) > 2
523         camber_gain = gradient(results.camber, results.values);
524         toe_gain = gradient(results.toe, results.values);
525
526         figure('Color','w');
527         subplot(2,1,1);
528         plot(results.values, camber_gain, '-o', 'LineWidth', 1.6, ...
529             'Color', [0.85 0.325 0.098]);
530         grid on; xlabel(xlabel); ylabel('Camber gain (deg per unit)');
531         title('Camber gain');
532         subplot(2,1,2);
533         plot(results.values, toe_gain, '-o', 'LineWidth', 1.6, ...
534             'Color', [0 0.447 0.741]);
535         grid on; xlabel(xlabel); ylabel('Toe gain (deg per unit)');
536         title('Toe gain');
537     end
538
539     % --- Per-member length change vs sweep ---
540     if isfield(results, 'dL')
541         plot_member_curves(results.values, results.dL, ...
542             xlabel, '\DeltaLength from initial (mm)', ...
543             'Per-member length change vs sweep input', true, components_to_show);
544     end
545
546     % --- Per-member load vs sweep ---
547     if isfield(results, 'loads')
548         plot_member_curves(results.values, results.loads, ...
549             xlabel, 'Axial load (N)', ...
550             'Per-member load vs sweep input', false, components_to_show);
551     end
552 end

```

```

553
554 function plot_member_curves(x, Y, xlabel, ylabel, ttl, also_plot_no_SL,
    components_to_show)
555 % PLOT_MEMBER_CURVES One line per member (rows of Y) vs x.
556 % Rows 1..7 are 1D members (tubes/PR/spring); rows 8..13 are clevis
557 % stand-offs (deflection magnitudes / 3D force magnitudes).
558 % If also_plot_no_SL is true, a second figure is produced with the SL
559 % (spring) row excluded so the smaller members are easier to read.
560 % components_to_show filters which rows are plotted.
561     if nargin < 6, also_plot_no_SL = false; end
562     if nargin < 7, components_to_show = 'all'; end
563
564     component_names = {'LF','LR','UF','UR','TR','PR','SL', ...
565                       'FLC','RLC','UFC','URC','SC','RC'};
566     show_mask = resolve_component_mask(component_names, components_to_show);
567
568     % If this is the loads plot, the clevis y-axis should read "Load Magnitude"
569     is_loads_plot = strcmpi(ylabel, 'Axial load (N)');
570     if is_loads_plot
571         ylabel_clevis = 'Load Magnitude (N)';
572     else
573         ylabel_clevis = ylabel;
574     end
575
576     rows_1D      = find(show_mask(1:7));      % indices into 1..7
577     rows_clevis  = find(show_mask(8:13)) + 7; % indices into 8..13
578
579     figure('Color','w');
580
581     subplot(2,1,1); hold on; grid on;
582     for i = rows_1D
583         plot(x, Y(i,:), '-', 'LineWidth', 1.4, ...
584              'DisplayName', component_names{i});
585     end
586     xlabel(xlabel); ylabel(ylabel);
587     title([ttl '      1D members (signed: +ve = tension / extension)']);
588     if ~isempty(rows_1D), legend('Location','bestoutside'); end
589     hold off;
590
591     subplot(2,1,2); hold on; grid on;
592     for i = rows_clevis
593         plot(x, Y(i,:), '-', 'LineWidth', 1.4, ...
594              'DisplayName', component_names{i});
595     end
596     xlabel(xlabel); ylabel(ylabel_clevis);
597     title([ttl '      clevises (magnitude of 3D deflection / 3D force)']);
598     if ~isempty(rows_clevis), legend('Location','bestoutside'); end
599     hold off;
600
601     % --- Second figure with SL excluded (top subplot only differs) ---
602     if also_plot_no_SL
603         rows_1D_no_SL = rows_1D(rows_1D ~= 7);
604         figure('Color','w');
605
606         subplot(2,1,1); hold on; grid on;
607         for i = rows_1D_no_SL
608             plot(x, Y(i,:), '-', 'LineWidth', 1.4, ...
609                  'DisplayName', component_names{i});
610         end
611         xlabel(xlabel); ylabel(ylabel);
612         title([ttl '      1D members excl. SL (signed: +ve = tension /
613                  extension)']);
614         if ~isempty(rows_1D_no_SL), legend('Location','bestoutside'); end

```

```

614     hold off;
615
616     subplot(2,1,2); hold on; grid on;
617     for i = rows_clevis
618         plot(x, Y(i,:), '-', 'LineWidth', 1.4, ...
619             'DisplayName', component_names{i});
620     end
621     xlabel(xlbl); ylabel(ylbl_clevis);
622     title([ttl '      clevises (magnitude of 3D deflection / 3D force)']);
623     if ~isempty(rows_clevis), legend('Location','bestoutside'); end
624     hold off;
625 end
626 end
627
628 function mask = resolve_component_mask(component_names, components_to_show)
629 % RESOLVE_COMPONENT_MASK Returns a logical mask the same length as
630 % component_names indicating which entries the user wants shown.
631 % - 'all' (or empty {}) => all true
632 % - a cell array of names => true only for matching names (case-insensitive)
633 if (ischar(components_to_show) && strcmpi(components_to_show, 'all')) ...
634     || (iscell(components_to_show) && isempty(components_to_show))
635     mask = true(1, numel(component_names));
636     return
637 end
638 if ~iscell(components_to_show)
639     components_to_show = cellstr(components_to_show);
640 end
641 requested_upper = upper(strtrim(components_to_show));
642 names_upper     = upper(component_names);
643 mask = ismember(names_upper, requested_upper);
644
645 % Warn about anything the user listed but that didn't match
646 unknown = setdiff(requested_upper, names_upper);
647 if ~isempty(unknown)
648     warning('components_to_show: unrecognized name(s): %s', strjoin(unknown,
649         ', '));
650 end
651 end
652
653 function add_bar_end_labels(b, values, fmt)
654 % ADD_BAR_END_LABELS Place horizontal, slightly bold numeric labels just
655 % past the end of each bar in a single bar series. Positive bars get a
656 % label above; negative bars get one below. The offset is a small fraction
657 % of the current y-axis range so it sits "just off" the bar end.
658 if nargin < 3, fmt = '%.2f'; end
659 ax = ancestor(b, 'axes');
660 yl = ylim(ax);
661 offset = 0.02 * (yl(2) - yl(1));
662 if offset == 0, offset = 0.02 * max(abs(values(:))) + eps; end
663
664 xtips = b.XEndPoints;
665 ytips = b.YEndPoints;
666 for k = 1:numel(values)
667     v = values(k);
668     if v >= 0
669         ypos = ytips(k) + offset;
670         va    = 'bottom';
671     else
672         ypos = ytips(k) - offset;
673         va    = 'top';
674     end
675 end

```

```

676         text(ax, xtips(k), ypos, sprintf(fmt, v), ...
677             'HorizontalAlignment','center', ...
678             'VerticalAlignment',va, ...
679             'FontSize', 8, ...
680             'FontWeight','bold', ...
681             'Rotation', 0);
682     end
683
684     % Pad the y-limits a touch so labels aren't clipped
685     new_yl = yl;
686     if any(values >= 0)
687         new_yl(2) = max(yl(2), max(values) + 3*offset);
688     end
689     if any(values < 0)
690         new_yl(1) = min(yl(1), min(values) - 3*offset);
691     end
692     ylim(ax, new_yl);
693 end
694
695
696 function plot_component_loads_bar(loads, components_to_show)
697 % PLOT_COMPONENT_LOADS_BAR Single-case bar chart of axial loads in each
698 % member. loads(1:7) are signed (tension +ve), loads(8:13) are 3D force
699 % magnitudes.
700     if nargin < 2, components_to_show = 'all'; end
701
702     component_names = {'LF','LR','UF','UR','TR','PR','SL', ...
703                       'FLC','RLC','UFC','URC','SC','RC'};
704     show_mask = resolve_component_mask(component_names, components_to_show);
705     keep_idx = find(show_mask);
706
707     if isempty(keep_idx)
708         warning('plot_component_loads_bar: no components selected, skipping
709             figure.');
```

```

709         return
710     end
711
712     loads_kept = loads(keep_idx);
713     names_kept = component_names(keep_idx);
714     is_1D = keep_idx <= 7;
715
716     figure('Color','w');
717     b = bar(loads_kept);
718     b.FaceColor = 'flat';
719     blue_row = [0 0.4470 0.7410];
720     orange_row = [0.8500 0.3250 0.0980];
721     cdata = zeros(numel(keep_idx), 3);
722     cdata(is_1D, :) = repmat(blue_row, sum(is_1D), 1);
723     cdata(~is_1D, :) = repmat(orange_row, sum(~is_1D), 1);
724     b.CData = cdata;
725
726     set(gca, 'XTick', 1:numel(names_kept), 'XTickLabel', names_kept);
727     ylabel('Axial load (N) 1D: signed, clevis: |F|');
728     xlabel('Component');
729     title('Member loads at the load case (With Compliance)');
730     grid on;
731     yline(0, 'k-');
732
733     % Numeric labels just off each bar end (horizontal, slightly bold)
734     add_bar_end_labels(b, round(loads_kept, 1), '%.1f');
735 end
736
737

```

```

738 function [dL, loads] = compute_member_loads_and_dL(sol2, geom, dims, interps)
739 % COMPUTE_MEMBER_LOADS_AND_DL Returns 13x1 vectors of length-change (mm)
740 % and axial load (N) for the 13 compliant members, in the same order as
741 % the bar charts elsewhere:
742 %   [LF LR UF UR TR PR SL FLC RLC UFC URC SC RC]
743 %
744 % For 1D members (rows 1:7) loads are signed: +ve = tension, -ve = compression.
745 % Sign convention: the interpolants return F = restoring force as a
746 % function of length change. A stretched tube (delta>0) is in tension,
747 % so we report +F(delta).
748 % For the 6 clevis stand-offs (rows 8:13) loads are the magnitude of the
749 % 3D restoring-force vector from the clevis interpolant (signed direction
750 % isn't meaningful in 1D for a 3D mount).
751
752 UL = sol2(:,1);   UU = sol2(:,2);   UT = sol2(:,3);   US = sol2(:,4);
753 RS = sol2(:,5);   RP = sol2(:,6);
754 FLC = sol2(:,8);   RLC = sol2(:,9);
755 RUC = sol2(:,10);  FUC = sol2(:,11);
756 SCa = sol2(:,12);  RCa = sol2(:,13);
757
758 TRB = geom.TRB;           % chassis-side tie-rod base
759 SB = geom.SB;             % chassis-side spring/suspension attachment
760
761 % --- 1D members: length change and signed axial load ---
762 % Tubes use the Tube interpolant, push rod uses Push_Rod, spring is k=60.
763 [L_LF, F_LF] = tube_dL_and_F(UL, FLC, dims.LF, interps.Tube_Fx_interp);
764 [L_LR, F_LR] = tube_dL_and_F(UL, RLC, dims.LR, interps.Tube_Fx_interp);
765 [L_UF, F_UF] = tube_dL_and_F(UU, FUC, dims.UF, interps.Tube_Fx_interp);
766 [L_UR, F_UR] = tube_dL_and_F(UU, RUC, dims.UR, interps.Tube_Fx_interp);
767 [L_TR, F_TR] = tube_dL_and_F(UT, TRB, dims.TR, interps.Tube_Fx_interp);
768 [L_PR, F_PR] = tube_dL_and_F(US, RP, dims.PR, interps.Push_Rod_Fx_interp);
769
770 % Spring: linear k = 60, compression positive sign convention here is
771 % "tension positive" so report -spring_magnitude when in compression.
772 new_SL_vec = RS - SCa;
773 L_SL = norm(new_SL_vec);
774 comp = dims.SL - L_SL;           % +ve when shorter than free length
775 spring_force_signed = -comp * 60; % compression -> -ve, extension -> +ve
776
777 % --- Clevis stand-offs: length change and 3D force magnitude ---
778 [dFLC, mFLC] = clevis_dL_and_Fmag(FLC, dims.LFC, ...
779     interps.Lower_Clevice_Fx_interp, interps.Lower_Clevice_Fy_interp,
780     interps.Lower_Clevice_Fz_interp);
781 [dRLC, mRLC] = clevis_dL_and_Fmag(RLC, dims.LRC, ...
782     interps.Lower_Clevice_Fx_interp, interps.Lower_Clevice_Fy_interp,
783     interps.Lower_Clevice_Fz_interp);
784 [dUFC, mUFC] = clevis_dL_and_Fmag(FUC, dims.UFC, ...
785     interps.Upper_Clevice_Fx_interp, interps.Upper_Clevice_Fy_interp,
786     interps.Upper_Clevice_Fz_interp);
787 [dURC, mURC] = clevis_dL_and_Fmag(RUC, dims.URC, ...
788     interps.Upper_Clevice_Fx_interp, interps.Upper_Clevice_Fy_interp,
789     interps.Upper_Clevice_Fz_interp);
790 [dSC, mSC] = clevis_dL_and_Fmag(SCa, dims.SC, ...
791     interps.Upper_Clevice_Fx_interp, interps.Upper_Clevice_Fy_interp,
792     interps.Upper_Clevice_Fz_interp);
793 [dRC, mRC] = clevis_dL_and_Fmag(RCa, dims.RC, ...
794     interps.Upper_Clevice_Fx_interp, interps.Upper_Clevice_Fy_interp,
795     interps.Upper_Clevice_Fz_interp);
796
797 % Pack outputs in canonical order
798 dL = [L_LF - dims.LF;
799     L_LR - dims.LR;
800     L_UF - dims.UF;

```

```

795     L_UR = dims.UR;
796     L_TR = dims.TR;
797     L_PR = dims.PR;
798     L_SL = dims.SL;
799     dFLC; dRLC; dUFC; dURC; dSC; dRC];
800
801     loads = [F_LF; F_LR; F_UF; F_UR; F_TR; F_PR;
802             spring_force_signed;
803             mFLC; mRLC; mUFC; mURC; mSC; mRC];
804 end
805
806
807 function [L, F] = tube_dL_and_F(p1, p2, L0, interp)
808     v = p1 - p2;
809     L = norm(v);
810     delta = L - L0;
811     F = interp(delta);    % +ve in tension by convention
812 end
813
814
815 function [dL_norm, Fmag] = clevis_dL_and_Fmag(pos, ref, fxI, fyI, fzI)
816     dvec = (pos - ref)';    % row vector for the 3D interpolant
817     dL_norm = norm(dvec);
818     Fmag = norm([fxI(dvec); fyI(dvec); fzI(dvec)]);
819 end
820
821 % Function for first solution with rigid members and just spring reacting
822 % force
823 function res = Spring_Residuals(x, Front_Lower_Wishbone_Attachment,
824     Rear_Lower_Wishbone_Attachment, Front_Upper_Wishbone_Attachment,...
825     Rear_Upper_Wishbone_Attachment, Tie_Rod_Base, Suspension_Attachment,
826     Rocker_Clevice_Attachment,...
827     LF, LR, UF, UR, TR, PR, UD1, UD2, UD3, UD4, UD5, UD6, UD7, UD8, UD9, SL, PTS,
828     PTP, STP, Known_Force, CP_Load)
829
830     % Processes inputs accordingly to whehther unknowns on each axis are
831     % force or displacement
832     Load_Point_temp = x(:,7);
833
834     if Known_Force(1) == false
835         Load_Point_temp(1) = CP_Load(1);
836         CP_Load(1) = x(1,7);
837     end
838     if Known_Force(2) == false
839         Load_Point_temp(2) = CP_Load(2);
840         CP_Load(2) = x(2,7);
841     end
842     if Known_Force(3) == false
843         Load_Point_temp(3) = CP_Load(3);
844         CP_Load(3) = x(3,7);
845     end
846
847     % Extract unknowns from matrix
848     Upright_Lower_temp = x(:,1);
849     Upright_Upper_temp = x(:,2);
850     Upright_Tie_Rod_temp = x(:,3);
851     Upright_Suspension_Mount_temp = x(:,4);
852     Rocker_Spring_Attachment_temp = x(:,5);
853     Rocker_Push_Attachment_temp = x(:,6);
854
855     % Lower Front Length
856     res(1) = LF - (norm(Upright_Lower_temp - Front_Lower_Wishbone_Attachment));

```



```

855 % Lower Rear Length
856 res(2) = LR - (norm(Upright_Lower_temp - Rear_Lower_Wishbone_Attachment));
857
858 % Upper Front Length
859 res(3) = UF - (norm(Upright_Upper_temp - Front_Upper_Wishbone_Attachment));
860
861 % Upper Rear Length
862 res(4) = UR - (norm(Upright_Upper_temp - Rear_Upper_Wishbone_Attachment));
863
864 % Tie Rod Length
865 res(5) = TR - (norm(Upright_Tie_Rod_temp - Tie_Rod_Base));
866
867 % Push Rod Length
868 res(6) = PR - (norm(Upright_Suspension_Mount_temp -
869                     Rocker_Push_Attachment_temp));
870
871 % All upright dimensions
872 res(7) = UD1 - (norm(Upright_Lower_temp - Upright_Upper_temp));
873 res(8) = UD2 - (norm(Upright_Lower_temp - Upright_Tie_Rod_temp));
874 res(9) = UD3 - (norm(Upright_Lower_temp - Upright_Suspension_Mount_temp));
875 res(10) = UD4 - (norm(Upright_Upper_temp - Upright_Tie_Rod_temp));
876 res(11) = UD5 - (norm(Upright_Upper_temp - Upright_Suspension_Mount_temp));
877 res(12) = UD6 - (norm(Upright_Tie_Rod_temp - Upright_Suspension_Mount_temp));
878 res(13) = UD7 - norm(Load_Point_temp - Upright_Lower_temp);
879 res(14) = UD8 - norm(Load_Point_temp - Upright_Upper_temp);
880 res(15) = UD9 - norm(Load_Point_temp - Upright_Tie_Rod_temp);
881
882 % Rocker Dimensions
883 res(16) = PTS - norm(Rocker_Spring_Attachment_temp -
884                     Rocker_Clevice_Attachment);
885 res(17) = PTP - norm(Rocker_Push_Attachment_temp - Rocker_Clevice_Attachment);
886 res(18) = STP - norm(Rocker_Push_Attachment_temp -
887                     Rocker_Spring_Attachment_temp);
888
889 % Find direction vector of spring
890 susp_vector = Suspension_Attachment - Rocker_Spring_Attachment_temp;
891 susp_direction = susp_vector / norm(susp_vector);
892 force_component = dot(CP_Load, susp_direction);
893
894 % Find spring compression
895 comp = SL - norm(Rocker_Spring_Attachment_temp - Suspension_Attachment);
896 spring_force = comp * 60;
897
898 % Force balance
899 res(19) = force_component - spring_force;
900
901 % Enforce rigid rocker
902 res(20) = Rocker_Clevice_Attachment(1) - Rocker_Spring_Attachment_temp(1);
903 res(21) = Rocker_Clevice_Attachment(1) - Rocker_Push_Attachment_temp(1);
904
905 end
906
907 % function for sol2 incorporating compliance
908 function res = Susp_Residuals(x, Tie_Rod_Base, local_push_pos_og,
909 local_spring_pos_og, LFC, LRC, UFC, URC, RC, SC,...
910 PR, LF, LR, UF, UR, TR, UD1, UD2, UD3, UD4, UD5, UD6, UD7, UD8, UD9, SL, PTS,
911 PTP, STP, Known_Force, CP_Load, interps)
912
913 % Loads unknown forces/displacements accordingly
914 Load_Point_temp = x(:,7);
915
916 if Known_Force(1) == false

```

```

913     Load_Point_temp(1) = CP_Load(1);
914     CP_Load(1) = x(1,7);
915 end
916 if Known_Force(2) == false
917     Load_Point_temp(2) = CP_Load(2);
918     CP_Load(2) = x(2,7);
919 end
920 if Known_Force(3) == false
921     Load_Point_temp(3) = CP_Load(3);
922     CP_Load(3) = x(3,7);
923 end
924
925 % Extracts unknowns from matrix
926 Upright_Lower_temp = x(:,1);
927 Upright_Upper_temp = x(:,2);
928 Upright_Tie_Rod_temp = x(:,3);
929 Upright_Suspension_Mount_temp = x(:,4);
930 Rocker_Spring_temp = x(:,5);
931 Rocker_Push_temp = x(:,6);
932 Front_Lower_Clevice_Attachment_temp = x(:,8);
933 Rear_Lower_Clevice_Attachment_temp = x(:,9);
934 Rear_Upper_Clevice_Attachment_temp = x(:,10);
935 Front_Upper_Clevice_Attachment_temp = x(:,11);
936 Susp_Clevice_Attachment_temp = x(:,12);
937 Rocker_Clevice_Attachment_temp = x(:,13);
938
939 % All upright dimensions
940 res(1) = UD1 - (norm(Upright_Lower_temp - Upright_Upper_temp));
941 res(2) = UD2 - (norm(Upright_Lower_temp - Upright_Tie_Rod_temp));
942 res(3) = UD3 - (norm(Upright_Lower_temp - Upright_Suspension_Mount_temp));
943 res(4) = UD4 - (norm(Upright_Upper_temp - Upright_Tie_Rod_temp));
944 res(5) = UD5 - (norm(Upright_Upper_temp - Upright_Suspension_Mount_temp));
945 res(6) = UD6 - (norm(Upright_Tie_Rod_temp - Upright_Suspension_Mount_temp));
946 res(7) = UD7 - (norm(Load_Point_temp - Upright_Lower_temp));
947 res(8) = UD8 - (norm(Load_Point_temp - Upright_Upper_temp));
948 res(9) = UD9 - (norm(Load_Point_temp - Upright_Tie_Rod_temp));
949
950 % Push Rod Force
951 new_PR = Upright_Suspension_Mount_temp - Rocker_Push_temp;
952 delta = (norm(new_PR)) - PR;
953 Force = interps.Push_Rod_Fx_interp(delta);
954 PR_Force = -Force * (new_PR / norm(new_PR));
955
956 % Lower Front Force
957 new_LF = Upright_Lower_temp - Front_Lower_Clevice_Attachment_temp;
958 delta = (norm(new_LF)) - LF;
959 Force = interps.Tube_Fx_interp(delta);
960 LF_Force = -Force * (new_LF / norm(new_LF));
961
962 % Lower Rear Force
963 new_LR = Upright_Lower_temp - Rear_Lower_Clevice_Attachment_temp;
964 delta = (norm(new_LR)) - LR;
965 Force = interps.Tube_Fx_interp(delta);
966 LR_Force = -Force * (new_LR / norm(new_LR));
967
968 % Upper Rear Force
969 new_UR = Upright_Upper_temp - Rear_Upper_Clevice_Attachment_temp;
970 delta = (norm(new_UR)) - UR;
971 Force = interps.Tube_Fx_interp(delta);
972 UR_Force = -Force * (new_UR / norm(new_UR));
973
974 % Upper Front Force
975 new_UF = Upright_Upper_temp - Front_Upper_Clevice_Attachment_temp;

```

```

976 delta = (norm(new_UF)) - UF;
977 Force = interps.Tube_Fx_interp(delta);
978 UF_Force = -Force * (new_UF / norm(new_UF));
979
980 % Tie Rod Force
981 new_TR = Upright_Tie_Rod_temp - Tie_Rod_Base;
982 delta = (norm(new_TR)) - TR;
983 Force = interps.Tube_Fx_interp(delta);
984 TR_Force = -Force * (new_TR / norm(new_TR));
985
986 % Suspension Force - find spring compression
987 new_SL = Rocker_Spring_temp - Susp_Clevice_Attachment_temp;
988 comp = SL - norm(new_SL);
989 spring_magnitude = comp * 60;
990 Spring_Force = spring_magnitude * (new_SL / norm(new_SL));
991
992 % Combine wishbones that connect
993 Lower_Force = LF_Force + LR_Force;
994 Upper_Force = UF_Force + UR_Force;
995
996 [LFC_Force, LRC_Force, UFC_Force, URC_Force, RC_Force, SC_Force] =
997     deal(zeros(3,1));
998
999 % Lower Front
1000 delta_vector = (Front_Lower_Clevice_Attachment_temp - LFC)';
1001 LFC_Force(1) = interps.Lower_Clevice_Fx_interp(delta_vector);
1002 LFC_Force(2) = interps.Lower_Clevice_Fy_interp(delta_vector);
1003 LFC_Force(3) = interps.Lower_Clevice_Fz_interp(delta_vector);
1004 res(10:12) = LFC_Force + LF_Force;
1005
1006 % Lower Rear
1007 delta_vector = (Rear_Lower_Clevice_Attachment_temp - LRC)';
1008 LRC_Force(1) = interps.Lower_Clevice_Fx_interp(delta_vector);
1009 LRC_Force(2) = interps.Lower_Clevice_Fy_interp(delta_vector);
1010 LRC_Force(3) = interps.Lower_Clevice_Fz_interp(delta_vector);
1011 res(13:15) = LRC_Force + LR_Force;
1012
1013 % Upper Front
1014 delta_vector = (Front_Upper_Clevice_Attachment_temp - UFC)';
1015 UFC_Force(1) = interps.Upper_Clevice_Fx_interp(delta_vector);
1016 UFC_Force(2) = interps.Upper_Clevice_Fy_interp(delta_vector);
1017 UFC_Force(3) = interps.Upper_Clevice_Fz_interp(delta_vector);
1018 res(16:18) = UFC_Force + UF_Force;
1019
1020 % Upper Rear
1021 delta_vector = (Rear_Upper_Clevice_Attachment_temp - URC)';
1022 URC_Force(1) = interps.Upper_Clevice_Fx_interp(delta_vector);
1023 URC_Force(2) = interps.Upper_Clevice_Fy_interp(delta_vector);
1024 URC_Force(3) = interps.Upper_Clevice_Fz_interp(delta_vector);
1025 res(19:21) = URC_Force + UR_Force;
1026
1027 % Rocker Clevice
1028 delta_vector = (Rocker_Clevice_Attachment_temp - RC)';
1029 RC_Force(1) = interps.Upper_Clevice_Fx_interp(delta_vector);
1030 RC_Force(2) = interps.Upper_Clevice_Fy_interp(delta_vector);
1031 RC_Force(3) = interps.Upper_Clevice_Fz_interp(delta_vector);
1032
1033 % Suspension Clevice
1034 delta_vector = (Susp_Clevice_Attachment_temp - SC)';
1035 SC_Force(1) = interps.Upper_Clevice_Fx_interp(delta_vector);
1036 SC_Force(2) = interps.Upper_Clevice_Fy_interp(delta_vector);
1037 SC_Force(3) = interps.Upper_Clevice_Fz_interp(delta_vector);
1038 res(22:24) = SC_Force + Spring_Force;

```

```

1038
1039 % Rocker Dimensions
1040 res(25) = PTS - norm(Rocker_Spring_temp - Rocker_Clevice_Attachment_temp);
1041 res(26) = PTP - norm(Rocker_Push_temp - Rocker_Clevice_Attachment_temp);
1042 res(27) = STP - norm(Rocker_Push_temp - Rocker_Spring_temp);
1043
1044 % Total moment balance about rocker pivot
1045 between = Rocker_Spring_temp - Rocker_Clevice_Attachment_temp;
1046 moment_1 = cross(between, Spring_Force);
1047 between = Rocker_Push_temp - Rocker_Clevice_Attachment_temp;
1048 moment_2 = cross(between, -PR_Force);
1049 M_Total = moment_1 + moment_2;
1050 res(28:30) = 0 - M_Total;
1051
1052 % Total force balance on rocker
1053 F_Total = Spring_Force - RC_Force - PR_Force;
1054 res(31:33) = 0 - F_Total;
1055
1056 % Moment balance about load point
1057 between = Upright_Upper_temp - Load_Point_temp;
1058 moment_1 = cross(between, Upper_Force);
1059 between = Upright_Tie_Rod_temp - Load_Point_temp;
1060 moment_2 = cross(between, TR_Force);
1061 between = Upright_Suspension_Mount_temp - Load_Point_temp;
1062 moment_3 = cross(between, PR_Force);
1063 between = Upright_Lower_temp - Load_Point_temp;
1064 moment_4 = cross(between, Lower_Force);
1065 M_Total = moment_1 + moment_2 + moment_3 + moment_4;
1066 res(34:36) = 0 - M_Total;
1067
1068 % Total force balance at upright
1069 F_Total = Lower_Force + Upper_Force + TR_Force + PR_Force;
1070 res(37:39) = CP_Load + F_Total;
1071
1072 end
1073
1074
1075 function interps = loadForceInterpolants(filepath, interps)
1076 % LOADFORCEINTERPOLANTS Build 3D gridded interpolants for Fx, Fy, Fz from a CSV.
1077
1078     if nargin < 2
1079         interps = struct();
1080     end
1081
1082     [~, baseName, ~] = fileparts(filepath);
1083     T = readtable(filepath);
1084
1085     x1 = unique(T.X);
1086     x2 = unique(T.Y);
1087     x3 = unique(T.Z);
1088     n1 = numel(x1);
1089     n2 = numel(x2);
1090     n3 = numel(x3);
1091
1092     assert(height(T) == n1*n2*n3, ...
1093         'CSV does not contain a complete regular grid (%d rows, expected %d).', ...
1094         height(T), n1*n2*n3);
1095
1096     Fx = zeros(n1, n2, n3);
1097     Fy = zeros(n1, n2, n3);
1098     Fz = zeros(n1, n2, n3);
1099
1100     [~, ix] = ismember(T.X, x1);

```

```

1101     [~, iy] = ismember(T.Y, x2);
1102     [~, iz] = ismember(T.Z, x3);
1103     for k = 1:height(T)
1104         Fx(ix(k), iy(k), iz(k)) = T.Fx(k);
1105         Fy(ix(k), iy(k), iz(k)) = T.Fy(k);
1106         Fz(ix(k), iy(k), iz(k)) = T.Fz(k);
1107     end
1108
1109     interps.([baseName '_Fx_interp']) = griddedInterpolant({x1, x2, x3}, Fx,
1110         'cubic', 'linear');
1111     interps.([baseName '_Fy_interp']) = griddedInterpolant({x1, x2, x3}, Fy,
1112         'cubic', 'linear');
1113     interps.([baseName '_Fz_interp']) = griddedInterpolant({x1, x2, x3}, Fz,
1114         'cubic', 'linear');
1115 end
1116
1117 function interps = loadAllForceInterpolants(filepaths)
1118     interps = struct();
1119     for k = 1:numel(filepaths)
1120         interps = loadForceInterpolants(filepaths{k}, interps);
1121     end
1122 end
1123
1124 function interps = loadAll1DForceInterpolants(filepaths, interps)
1125     for k = 1:numel(filepaths)
1126         interps = load1DForce(filepaths{k}, interps);
1127     end
1128 end
1129
1130 function interps = load1DForce(filepath, interps)
1131 % LOAD1DFORCE Build a 1D gridded interpolant for Fx from a CSV.
1132
1133     if nargin < 2
1134         interps = struct();
1135     end
1136
1137     [~, baseName, ~] = fileparts(filepath);
1138     T = readtable(filepath);
1139
1140     x1 = unique(T.X);
1141     x2 = unique(T.Y);
1142     x3 = unique(T.Z);
1143     n1 = numel(x1);
1144     n2 = numel(x2);
1145     n3 = numel(x3);
1146
1147     assert(height(T) == n1*n2*n3, ...
1148         'CSV does not contain a complete regular grid (%d rows, expected %d).', ...
1149         height(T), n1*n2*n3);
1150
1151     Fx = zeros(n1, n2, n3);
1152     Fy = zeros(n1, n2, n3);
1153     Fz = zeros(n1, n2, n3);
1154
1155     [~, ix] = ismember(T.X, x1);
1156     [~, iy] = ismember(T.Y, x2);
1157     [~, iz] = ismember(T.Z, x3);
1158     for k = 1:height(T)
1159         Fx(ix(k), iy(k), iz(k)) = T.Fx(k);
1160         Fy(ix(k), iy(k), iz(k)) = T.Fy(k);

```

```

1161     Fz(ix(k), iy(k), iz(k)) = T.Fz(k);
1162 end
1163
1164     interp1s([baseName '_Fx_interp']) = griddedInterpolant(x1, Fx, 'cubic',
1165         'linear');
1166 end
1167
1168 function plot_suspension(Front_Lower_Wishbone_Attachment,
1169     Rear_Lower_Wishbone_Attachment, ...
1170     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1171     Tie_Rod_Base, Suspension_Attachment, ...
1172     Upright_Lower, Upright_Upper, Upright_Tie_Rod, Upright_Suspension_Mount, ...
1173     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
1174     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
1175     Susp_Clevice_Base, Rocker_Clevice_Base, ...
1176     Rocker_Clevice_Attachment, Rocker_Spring_Attachment,
1177     Rocker_Push_Rod_Attachment, ...
1178     Load_Point, ...
1179     sol1, sol2, setups_to_show)
1180 % PLOT_SUSPENSION Plots three suspension configurations on the same 3D axes.
1181
1182 if nargin < 23, setups_to_show = 'all'; end
1183 setup_names = {'Initial','Rigid','With Compliance'};
1184 setup_mask = resolve_component_mask(setup_names, setups_to_show);
1185
1186 figure('Color', 'w');
1187 hold on; grid on; axis equal;
1188 xlabel('X (mm)'); ylabel('Y (mm)'); zlabel('Z (mm)');
1189
1190 % Title reflects only the setups that are actually drawn
1191 if all(setup_mask)
1192     title('Suspension Geometry: Initial vs Rigid vs With Compliance');
1193 else
1194     kept = setup_names(setup_mask);
1195     if isempty(kept)
1196         title('Suspension Geometry');
1197     else
1198         title(['Suspension Geometry: ' strjoin(kept, ' vs ')]);
1199     end
1200 end
1201 view(3);
1202
1203 % --- INITIAL configuration ---
1204 init.UL = Upright_Lower;
1205 init.UU = Upright_Upper;
1206 init.UT = Upright_Tie_Rod;
1207 init.US = Upright_Suspension_Mount;
1208 init.RS = Rocker_Spring_Attachment;
1209 init.RP = Rocker_Push_Rod_Attachment;
1210 init.LP = Load_Point;
1211 init.FLC = Front_Lower_Wishbone_Attachment;
1212 init.RLC = Rear_Lower_Wishbone_Attachment;
1213 init.RUC = Rear_Upper_Wishbone_Attachment;
1214 init.FUC = Front_Upper_Wishbone_Attachment;
1215 init.SC = Suspension_Attachment;
1216 init.RC = Rocker_Clevice_Attachment;
1217
1218 % --- sol1 configuration ---
1219 s1.UL = sol1(:,1);
1220 s1.UU = sol1(:,2);
1221 s1.UT = sol1(:,3);
1222 s1.US = sol1(:,4);

```

```

1221 s1.RS = sol1(:,5);
1222 s1.RP = sol1(:,6);
1223 s1.LP = sol1(:,7);
1224 s1.FLC = Front_Lower_Wishbone_Attachment;
1225 s1.RLC = Rear_Lower_Wishbone_Attachment;
1226 s1.RUC = Rear_Upper_Wishbone_Attachment;
1227 s1.FUC = Front_Upper_Wishbone_Attachment;
1228 s1.SC = Suspension_Attachment;
1229 s1.RC = Rocker_Clevice_Attachment;
1230
1231 % --- sol2 configuration ---
1232 s2.UL = sol2(:,1);
1233 s2.UU = sol2(:,2);
1234 s2.UT = sol2(:,3);
1235 s2.US = sol2(:,4);
1236 s2.RS = sol2(:,5);
1237 s2.RP = sol2(:,6);
1238 s2.LP = sol2(:,7);
1239 s2.FLC = sol2(:,8);
1240 s2.RLC = sol2(:,9);
1241 s2.RUC = sol2(:,10);
1242 s2.FUC = sol2(:,11);
1243 s2.SC = sol2(:,12);
1244 s2.RC = sol2(:,13);
1245
1246 % Chassis-side fixed points
1247 chassis.FLW = Front_Lower_Wishbone_Attachment;
1248 chassis.RLW = Rear_Lower_Wishbone_Attachment;
1249 chassis.FUW = Front_Upper_Wishbone_Attachment;
1250 chassis.RUW = Rear_Upper_Wishbone_Attachment;
1251 chassis.TRB = Tie_Rod_Base;
1252 chassis.SB = Suspension_Attachment;
1253 chassis.FLCB = Front_Lower_Clevice_Base;
1254 chassis.RLCB = Rear_Lower_Clevice_Base;
1255 chassis.UFCB = Upper_Front_Clevice_Base;
1256 chassis.URCB = Upper_Rear_Clevice_Base;
1257 chassis.SCB = Susp_Clevice_Base;
1258 chassis.RCB = Rocker_Clevice_Base;
1259
1260 plot_chassis_points(chassis);
1261
1262 if setup_mask(1), plot_one_setup(init, chassis, [0 0.4470 0.7410],
    'Initial'); end
1263 if setup_mask(2), plot_one_setup(s1, chassis, [0.8500 0.3250 0.0980],
    'Rigid'); end
1264 if setup_mask(3), plot_one_setup(s2, chassis, [0.4660 0.6740 0.1880], 'With
    Compliance'); end
1265
1266 legend('Location', 'bestoutside');
1267 hold off;
1268 end
1269
1270
1271 function plot_chassis_points(chassis)
1272 pts = [chassis.FLW, chassis.RLW, chassis.FUW, chassis.RUW, ...
1273        chassis.TRB, chassis.SB, ...
1274        chassis.FLCB, chassis.RLCB, chassis.UFCB, chassis.URCB, ...
1275        chassis.SCB, chassis.RCB];
1276 plot3(pts(1,:), pts(2,:), pts(3,:), ...
1277        's', 'Color', 'k', 'MarkerFaceColor', 'k', 'MarkerSize', 5, ...
1278        'DisplayName', 'Chassis points');
1279 end
1280

```



```

1281
1282 function plot_one_setup(s, chassis, col, name)
1283
1284     moving_pts = [s.UL, s.UU, s.UT, s.US, s.RS, s.RP, s.LP, ...
1285                   s.FLC, s.RLC, s.RUC, s.FUC, s.SC, s.RC];
1286     plot3(moving_pts(1,:), moving_pts(2,:), moving_pts(3,:), ...
1287           'o', 'Color', col, 'MarkerFaceColor', col, 'MarkerSize', 6, ...
1288           'HandleVisibility', 'off');
1289
1290     legend_used = false;
1291     function draw(p, q, style, lw)
1292         if nargin < 3, style = '-'; end
1293         if nargin < 4, lw = 1.6; end
1294         if ~legend_used
1295             plot3([p(1) q(1)], [p(2) q(2)], [p(3) q(3)], ...
1296                   style, 'Color', col, 'LineWidth', lw, 'DisplayName', name);
1297             legend_used = true;
1298         else
1299             plot3([p(1) q(1)], [p(2) q(2)], [p(3) q(3)], ...
1300                   style, 'Color', col, 'LineWidth', lw, 'HandleVisibility', 'off');
1301         end
1302     end
1303
1304     % Clevis links
1305     draw(chassis.FLCB, s.FLC, '-', 1.2);
1306     draw(chassis.RLCB, s.RLC, '-', 1.2);
1307     draw(chassis.UFCB, s.FUC, '-', 1.2);
1308     draw(chassis.URCB, s.RUC, '-', 1.2);
1309     draw(chassis.SCB, s.SC, '-', 1.2);
1310     draw(chassis.RCB, s.RC, '-', 1.2);
1311
1312     % Lower wishbone
1313     draw(s.UL, s.FLC);
1314     draw(s.UL, s.RLC);
1315
1316     % Upper wishbone
1317     draw(s.UU, s.FUC);
1318     draw(s.UU, s.RUC);
1319
1320     % Tie rod
1321     draw(s.UT, chassis.TRB);
1322
1323     % Push rod
1324     draw(s.US, s.RP);
1325
1326     % Spring
1327     draw(s.RS, s.SC, '-', 2.2);
1328
1329     % Rocker triangle
1330     draw(s.RC, s.RS);
1331     draw(s.RC, s.RP);
1332     draw(s.RS, s.RP);
1333
1334     % Upright internal structure
1335     draw(s.UL, s.UU);
1336     draw(s.UL, s.UT);
1337     draw(s.UL, s.US);
1338     draw(s.UU, s.UT);
1339     draw(s.UU, s.US);
1340     draw(s.UT, s.US);
1341
1342     % Load point connections
1343     draw(s.LP, s.UL);

```

```

1344     draw(s.LP, s.UU);
1345     draw(s.LP, s.UT);
1346 end
1347
1348
1349 function plot_component_lengths(Front_Lower_Wishbone_Attachment,
    Rear_Lower_Wishbone_Attachment, ...
1350     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1351     Tie_Rod_Base, Suspension_Attachment, ...
1352     Upright_Lower, Upright_Upper, Upright_Tie_Rod, Upright_Suspension_Mount, ...
1353     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
1354     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
1355     Susp_Clevice_Base, Rocker_Clevice_Base, ...
1356     Rocker_Clevice_Attachment, Rocker_Spring_Attachment,
        Rocker_Push_Rod_Attachment, ...
1357     sol1, sol2, components_to_show)
1358
1359 if nargin < 22, components_to_show = 'all'; end
1360
1361 component_names = {'LF','LR','UF','UR','TR','PR','SL', ...
1362     'FLC','RLC','UFC','URC','SC','RC'};
1363 show_mask = resolve_component_mask(component_names, components_to_show);
1364
1365 L_init = compute_lengths( ...
1366     Upright_Lower, Upright_Upper, Upright_Tie_Rod, Upright_Suspension_Mount,
        ...
1367     Rocker_Spring_Attachment, Rocker_Push_Rod_Attachment, ...
1368     Front_Lower_Wishbone_Attachment, Rear_Lower_Wishbone_Attachment, ...
1369     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1370     Tie_Rod_Base, Suspension_Attachment, ...
1371     Front_Lower_Wishbone_Attachment, Rear_Lower_Wishbone_Attachment, ...
1372     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1373     Suspension_Attachment, Rocker_Clevice_Attachment, ...
1374     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
1375     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
1376     Susp_Clevice_Base, Rocker_Clevice_Base);
1377
1378 L_sol1 = compute_lengths( ...
1379     sol1(:,1), sol1(:,2), sol1(:,3), sol1(:,4), ...
1380     sol1(:,5), sol1(:,6), ...
1381     Front_Lower_Wishbone_Attachment, Rear_Lower_Wishbone_Attachment, ...
1382     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1383     Tie_Rod_Base, Suspension_Attachment, ...
1384     Front_Lower_Wishbone_Attachment, Rear_Lower_Wishbone_Attachment, ...
1385     Front_Upper_Wishbone_Attachment, Rear_Upper_Wishbone_Attachment, ...
1386     Suspension_Attachment, Rocker_Clevice_Attachment, ...
1387     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
1388     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
1389     Susp_Clevice_Base, Rocker_Clevice_Base);
1390
1391 L_sol2 = compute_lengths( ...
1392     sol2(:,1), sol2(:,2), sol2(:,3), sol2(:,4), ...
1393     sol2(:,5), sol2(:,6), ...
1394     sol2(:,8), sol2(:,9), ...
1395     sol2(:,11), sol2(:,10), ...
1396     Tie_Rod_Base, sol2(:,12), ...
1397     sol2(:,8), sol2(:,9), ...
1398     sol2(:,11), sol2(:,10), ...
1399     sol2(:,12), sol2(:,13), ...
1400     Front_Lower_Clevice_Base, Rear_Lower_Clevice_Base, ...
1401     Upper_Front_Clevice_Base, Upper_Rear_Clevice_Base, ...
1402     Susp_Clevice_Base, Rocker_Clevice_Base);
1403

```

```

1404 data = [L_init; L_sol1; L_sol2]';
1405
1406 delta_sol1 = L_sol1 - L_init;
1407 delta_sol2 = L_sol2 - L_init;
1408 delta_data = [delta_sol1; delta_sol2]';
1409
1410 % --- Apply filter to all bar-chart data ---
1411 keep_idx = find(show_mask);
1412 names_kept = component_names(keep_idx);
1413 data_kept = data(keep_idx, :);
1414 delta_data_kept = delta_data(keep_idx, :);
1415
1416 if isempty(keep_idx)
1417     warning('plot_component_lengths: no components selected, skipping
1418         figures.');
```

```

1418 else
1419     figure('Color', 'w');
1420     b = bar(data_kept, 'grouped');
1421     b(1).FaceColor = [0 0.4470 0.7410];
1422     b(2).FaceColor = [0.8500 0.3250 0.0980];
1423     b(3).FaceColor = [0.4660 0.6740 0.1880];
1424     set(gca, 'XTick', 1:numel(names_kept), 'XTickLabel', names_kept);
1425     ylabel('Length (mm)');
1426     xlabel('Component');
1427     title('Component lengths at each solve stage');
1428     legend({'Initial', 'Rigid', 'With Compliance'}, 'Location', 'best');
1429     grid on;
1430
1431     % Compute a single offset based on overall y-range (bars are
1432     % grouped, so we want consistent offset across all 3 series)
1433     ax = gca;
1434     yl = ylim(ax);
1435     offset = 0.02 * (yl(2) - yl(1));
1436     if offset == 0, offset = 0.02 * max(abs(data_kept(:))) + eps; end
1437     for k = 1:numel(b)
1438         xtips = b(k).XEndPoints;
1439         ytips = b(k).YEndPoints;
1440         yvals = b(k).YData;
1441         for j = 1:numel(yvals)
1442             v = yvals(j);
1443             if v >= 0
1444                 ypos = ytips(j) + offset; va = 'bottom';
1445             else
1446                 ypos = ytips(j) - offset; va = 'top';
1447             end
1448             text(ax, xtips(j), ypos, sprintf('%.2f', v), ...
1449                 'HorizontalAlignment', 'center', ...
1450                 'VerticalAlignment', va, ...
1451                 'FontSize', 7, ...
1452                 'FontWeight', 'bold', ...
1453                 'Rotation', 0);
1454         end
1455     end
1456     % Pad y-limits so labels don't get clipped
1457     new_yl = yl;
1458     if any(data_kept(:) >= 0)
1459         new_yl(2) = max(yl(2), max(data_kept(:)) + 3*offset);
1460     end
1461     if any(data_kept(:) < 0)
1462         new_yl(1) = min(yl(1), min(data_kept(:)) - 3*offset);
1463     end
1464     ylim(ax, new_yl);
1465

```

```

1466 % Deflection chart: drop the Rigid group (rigid lengths don't
1467 % change except for the spring), keep just the With-Compliance
1468 % deflections labelled "Flexible". The zero line represents the
1469 % initial (undeformed) state and is labelled "Initial".
1470 delta_data_kept_only = delta_data_kept(:, 2); % drop sol1 column
1471
1472 figure('Color', 'w');
1473 b2 = bar(delta_data_kept_only, 'DisplayName', 'Flexible');
1474 b2.FaceColor = [0.4660 0.6740 0.1880];
1475 set(gca, 'XTick', 1:numel(names_kept), 'XTickLabel', names_kept);
1476 ylabel('\DeltaLength from initial (mm)');
1477 xlabel('Component');
1478 title('Component deflection from initial length');
1479 grid on;
1480 yline(0, 'k-', 'DisplayName', 'Initial');
1481 legend('Location','best');
1482
1483 add_bar_end_labels(b2, round(b2.YData, 4), '%.4f');
1484
1485 % --- Same deflection chart but with SL row excluded (and filter) ---
1486 SL_idx = 7;
1487 keep_no_SL = keep_idx(keep_idx ~= SL_idx);
1488 names_no_SL = component_names(keep_no_SL);
1489 delta_data_no_SL = delta_data(keep_no_SL, 2); % drop sol1 column
1490
1491 if ~isempty(keep_no_SL)
1492     figure('Color', 'w');
1493     b3 = bar(delta_data_no_SL, 'DisplayName', 'Flexible');
1494     b3.FaceColor = [0.4660 0.6740 0.1880];
1495     set(gca, 'XTick', 1:numel(names_no_SL), 'XTickLabel', names_no_SL);
1496     ylabel('\DeltaLength from initial (mm)');
1497     xlabel('Component');
1498     title('Component deflection from initial length (excl. SL)');
1499     grid on;
1500     yline(0, 'k-', 'DisplayName', 'Initial');
1501     legend('Location','best');
1502
1503     add_bar_end_labels(b3, round(b3.YData, 4), '%.4f');
1504 end
1505 end
1506
1507 T = table(component_names', L_init', L_sol1', L_sol2', delta_sol1',
1508         delta_sol2', ...
1509         'VariableNames', {'Component','Initial_mm','sol1_mm','sol2_mm', ...
1510         'delta_sol1_mm','delta_sol2_mm'});
1511 disp(T);
1512 end
1513
1514 function L = compute_lengths(UL, UU, UT, US, RS, RP, ...
1515     FLW, RLW, FUW, RUW, TRB, SB, ...
1516     FLC, RLC, FUC, RUC, SC, RC, ...
1517     FLCB, RLCB, UFCB, URCB, SCB, RCB)
1518
1519 L = [norm(UL - FLW), ...
1520     norm(UL - RLW), ...
1521     norm(UU - FUW), ...
1522     norm(UU - RUW), ...
1523     norm(UT - TRB), ...
1524     norm(US - RP), ...
1525     norm(RS - SB), ...
1526     norm(FLC - FLCB), ...
1527     norm(RLC - RLCB), ...

```

```

1528     norm(FUC - UFCB), ...
1529     norm(RUC - URCB), ...
1530     norm(SC - SCB), ...
1531     norm(RC - RCB)];
1532 end
1533
1534
1535 function R = rot_between_vectors(a, b)
1536     a = a(:) / norm(a);
1537     b = b(:) / norm(b);
1538     v = cross(a, b);
1539     s = norm(v);
1540     c = dot(a, b);
1541
1542     if s < 1e-12
1543         if c > 0
1544             R = eye(3);
1545         else
1546             if abs(a(1)) < 0.9, tmp = [1;0;0]; else, tmp = [0;1;0]; end
1547             axis = cross(a, tmp); axis = axis / norm(axis);
1548             R = axis_angle(axis, pi);
1549         end
1550     end
1551     return
1552 end
1553
1554 K = [ 0    -v(3)  v(2);
1555       v(3)    0   -v(1);
1556      -v(2)  v(1)    0   ];
1557 R = eye(3) + K + K*K * ((1 - c) / (s^2));
1558 end
1559
1560 function R = axis_angle(k, theta)
1561     k = k(:) / norm(k);
1562     K = [ 0    -k(3)  k(2);
1563          k(3)    0   -k(1);
1564          -k(2)  k(1)    0   ];
1565     R = eye(3) + sin(theta)*K + (1-cos(theta))*(K*K);
1566 end
1567
1568
1569 function [toe_deg, camber_deg] = compute_toe_camber(Upright_Lower, Upright_Upper,
    Upright_Tie_Rod, static_offset_rotation)
1570 % COMPUTE_TOE_CAMBER Returns toe and camber in degrees from the three
1571 % upright points, using the calibration rotation derived from the static state.
1572
1573     vector_1 = Upright_Tie_Rod - Upright_Lower;
1574     vector_2 = Upright_Tie_Rod - Upright_Upper;
1575     normal_upright = cross(vector_1, vector_2);
1576
1577     actual_normal = static_offset_rotation * normal_upright;
1578     v = actual_normal / norm(actual_normal);
1579
1580     n_toe = [0;0;1];
1581     v_in_xy = v - dot(v, n_toe) * n_toe;
1582     a = [0;-1;0];
1583     toe_deg = rad2deg( atan2(dot(cross(a, v_in_xy), n_toe), dot(a, v_in_xy)) );
1584
1585     n_camber = [1;0;0];
1586     v_in_yz = v - dot(v, n_camber) * n_camber;
1587     a = [0;-1;0];
1588     camber_deg = rad2deg( atan2(dot(cross(a, v_in_yz), n_camber), dot(a, v_in_yz))
    );

```

1589 **end**

B Python Code

B.1 Ansys Load Sweep Script

```

1 import System
2 import os
3
4 # --- 1. Define your sweep ranges (in mm) ---
5 x_steps = [-0.2, 0, 0.2]
6 y_steps = [-0.2, 0, 0.2]
7 z_steps = [-0.2, 0, 0.2]
8
9 # --- 2. Setup the output file ---
10 file_path = r"C:\Users\Public\Documents\GBDP Ansys\Results\Upper_Clevicse_Mag.csv"
11
12 # --- 3. Hook into the ANSYS tree ---
13 analysis = ExtAPI.DataModel.Project.Model.Analyses[0]
14 solution = analysis.Solution
15
16 disp = ExtAPI.DataModel.GetObjectsByName("Sweep_Displacement")[0]
17 force_probe = ExtAPI.DataModel.GetObjectsByName("Sweep_Reaction")[0]
18
19 # --- 4. The Sweep Loop ---
20 ExtAPI.Log.WriteMessage("Starting parameter sweep...")
21
22 with open(file_path, "w") as f:
23     f.write("X,Y,Z,Fx,Fy,Fz,Status,Message\n")
24
25     for x in x_steps:
26         for y in y_steps:
27             for z in z_steps:
28                 error_info = "None"
29                 solve_status = "Success"
30                 fx, fy, fz = 0.0, 0.0, 0.0 # Defaults
31
32                 try:
33                     # 1. Update Displacements
34                     disp.XComponent.Output.SetDiscreteValue(0, Quantity(x, "mm"))
35                     disp.YComponent.Output.SetDiscreteValue(0, Quantity(y, "mm"))
36                     disp.ZComponent.Output.SetDiscreteValue(0, Quantity(z, "mm"))
37
38                     ExtAPI.Application.Messages.Clear()
39                     analysis.Solve(True)
40
41                     # --- REVERSED LOGIC: More intuitive to read ---
42                     # Check if the status IS actually "Done"
43                     if str(analysis.Solution.Status) == "Done":
44                         # SUCCESS: Read the results
45                         force_probe.EvaluateAllResults()
46                         fx = force_probe.XAxis.Value
47                         fy = force_probe.YAxis.Value
48                         fz = force_probe.ZAxis.Value
49                     else:
50                         # FAILURE: Something went wrong in the solver
51                         solve_status = "FAILED"
52                         error_info = "Solver did not reach 'Done' status."
53

```

```

54         # Grab the specific error from the message pane
55         app_messages = ExtAPI.Application.Messages
56         if app_messages and app_messages.Count > 0:
57             for msg in app_messages:
58                 if str(msg.Severity) == "Error":
59                     error_info = msg.DisplayString.replace(",", " ",
60                                     "\n").replace("\n", " ")
61                     break
62
63         except Exception as e:
64             solve_status = "SCRIPT_ERROR"
65             error_info = str(e).replace(",", " |")
66
67         # --- THE CRITICAL PART: Writing to the file ---
68         # This MUST be inside the loop, but outside the try/except/if/else
69         csv_line = "{},{},{},{},{},{},{},{},{},{}\n".format(x, y, z, fx, fy, fz,
            solve_status, error_info)
        f.write(csv_line)

```