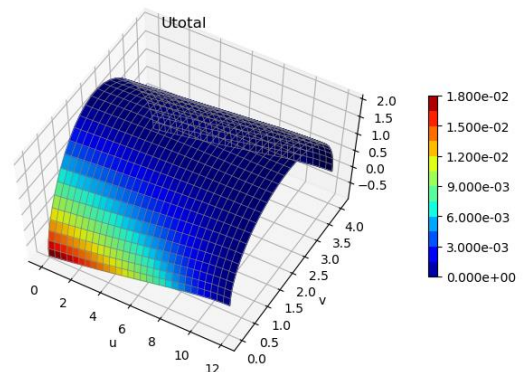
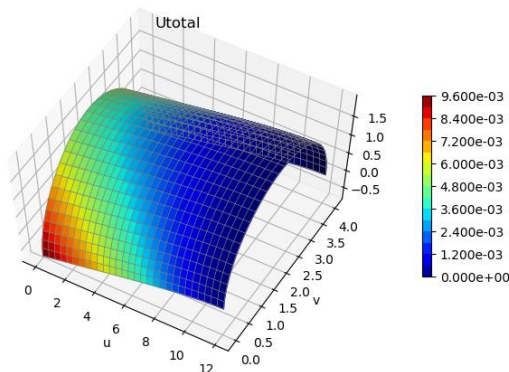
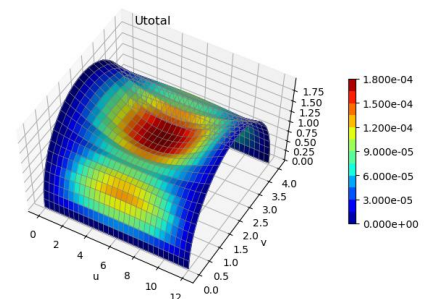
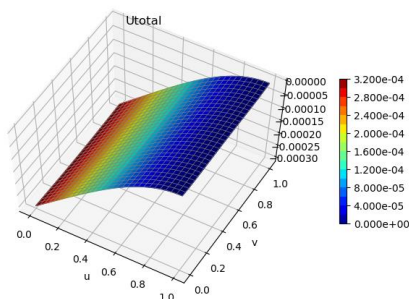
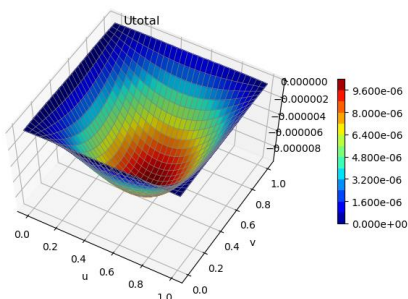


Finite difference analysis of shell structures

Chulong Li



Student:
Project duration
Assessment committee

Chulong Li
Dec. 3, 2020 – Oct. 31, 2021
Dr. ir. P.C.J. Hoogenboom
Prof. dr. ir. M.A.N. Hendriks
Dr. ir. C. Kasbergen
Dr. ir. F.P. van der Meer

TU Delft
TU Delft

Summary

The finite element method is widely used in modelling shell structures. However, the finite element method does not solve the shell differential equations because shell finite elements are derived from solid elements. Currently, there is not software available for solving the shell differential equations. For plates, a novel finite difference method has been recently explored by the author (Li, 2020). This Python algorithm does not solve the fourth-order plate differential equation directly. Instead, it solves eleven first-order differential equations simultaneously. The advantage of the method is in the boundary conditions; no edge or corner molecules are involved. Can this plate algorithm can be extended to shell structures? The general shell differential equations (Sanders-Koiter equations) have never been solved by the finite difference method. If possible, this would provide an independent way of checking shell finite element results.

The objective of this project was to develop and test a finite difference algorithm called *shell code* that can solve the 21 Sanders-Koiter equations. The idea was to use first-order finite-difference approximation only because this gives a simple discretization and modern computers may be able to handle the large number of equations.

To this end, a 1200-line Python program has been built. In the process many versions of *shell code* were considered, including

- 1) Two programming languages (Python and R)
- 2) Three interpolations for approximating gradients (three-point and five-point with two end slopes)
- 3) Determined and over-determined systems of equations (square and rectangular matrices)
- 4) Four solvers for the systems of equations

Important constraints are required memory and computation time and they were recorded for each test. Five shell models with various geometries, loads, and boundary conditions have been analyzed. The results of these model tests (displacement, bending moment, and shear force) were compared to finite element results. Discussions on the comparisons have shown that almost all versions produced incorrect results and the most important factor for affecting results is the solving method.

The version that works well has the following features; five-point interpolation with zero end slope, rectangular matrix, solver `lm.fit.sparse` (R). Approximately 80% of the shell code results match the finite element results with a deviation less than 5% (see Test BLM1-5). The deviation may be removed in the near future by a finer grid on a powerful computer or by applying an advanced solver. The main conclusion is that it is not only theoretically possible but also practically possible to solve the Sanders-Koiter equations by the finite difference method.

Content

1. Introduction.....	1
1.1 Problem Statement.....	1
1.2 Objective.....	1
1.3 Approach.....	2
1.4 Research workflow.....	4
2. Literature review.....	5
2.1 A short review on the development of plate and shell theories.....	5
2.2 Sanders-Koiter equations.....	6
2.3 A short review on the finite different method.....	9
2.4 A short overview of application of finite difference method on shell theory.....	10
2.5 Sparse linear system: overdetermined and determined, its storage and solving methods.....	11
3. Shell code and tests.....	19
3.1 Work flow of code.....	19
3.2 Tested models.....	20
3.3 Differentiation approximated by FDM.....	21
3.4 Formation of matrix [M].....	22
3.5 Loading steps and plotting steps.....	24
3.6 Formation of square matrix [M].....	25
3.7 Model tests of rectangular and square matrix.....	30
3.8 Solver tests.....	31
3.9 Five-point difference approximation and their tests.....	32
4. Results.....	34
4.1 Finite element solution.....	34
4.2 Rectangular matrix test results.....	37
4.3 Square matrix test results.....	37
4.4 Solver test results.....	38
4.5 Five-point difference approximation test results.....	39
4.6 Overall comparison between shell code results.....	40
4.7 Comparison between Rectangular matrix test and Square matrix test results.....	47
4.8 Comparison between pinv solver and <i>lm.fit.sparse</i> solver test results.....	48
4.9 Comparison between five-point difference approximation and two-point difference approximation test results.....	48
Discussion.....	50
5.1 Accuracy, reliability, and efficiency.....	50
5.2 Matrix quality.....	59
5.3 Possible factors affecting shell code results.....	61
5.4 Simplicity of code structure.....	67
6. Conclusion.....	69
7. Recommendation.....	70
8. Reference list.....	71
9. Appendix.....	73
Rectangular matrix test plots.....	73
Square matrix test plots.....	76
Square matrix test plots.....	80
Solver test results.....	91
Five-point difference approximation test plots.....	102
Matrix quality check.....	117
Discussion on number of iterations.....	118
Discussion on unit system.....	118

List of figures

Figure 1: Result plots of uniformly loaded two-way slab 30*30 nodes (Li, 2020)	1
Figure 2: First order derivative by finite difference method.....	9
Figure 3: Compressed Sparse Row format (index pointers, indices, and data)	13
Figure 4:Compressed Sparse Column format (index pointers, indices, and data).....	13
Figure 5: Row permutation p_r discovering banded structure in the matrix A. (c-d) Row permutation p_r while solving vertical concatenation of two matrices	17
Figure 6: Work flow of code.....	19
Figure 7: Add equations to [M].....	24
Figure 8: Non-zero value distribution in [M] and [f].....	24
Figure 9: Distribution of assigned equations in the rectangular matrix	25
Figure 10: Modified distribution of assigned equations by central node method.....	26
Figure 11: Modified distribution of assigned equations by undefined node method.....	28
Figure 12: Modified distribution of assigned equations by undefined node method.....	29
Figure 13: Work flow of solver tests.....	31
Figure 14: Model 1 displacement, bending moment and shear force finite element results by SCIA Engineer	34
Figure 15: Model 2 displacement, bending moment and shear force finite element results by SCIA Engineer	34
Figure 16:Model 3 displacement, bending moment and shear force finite element results by SCIA Engineer	35
Figure 17: Model 4 displacement, bending moment and shear force finite element results by SCIA Engineer	36
Figure 18: Model 5 displacement, bending moment and shear force finite element results by SCIA Engineer	36
120Figure 19: Comparison of overall test results for model 1.....	43
Figure 20: Comparison of overall test results for model 2	43
Figure 21: Comparison of overall test results for model 3	44
Figure 22: Comparison of overall test results for model 4	44
Figure 23: Comparison test results for model 5	44
Figure 24: Comparison of uz results for model 1	44
Figure 25: Comparison of uz results for model 2	44
Figure 26: Comparison of uz results for model 3	44
Figure 27: Comparison of uz results for model 4	44
Figure 28: Comparison of uz results for model 5	45
Figure 29: Comparison of mxx results for model 1	45
Figure 30: Comparison of mxx results for model 2	45
Figure 31: Comparison of mxx results for model 3	45
Figure 32: Comparison of mxx results for model 4	45
Figure 33: Comparison of mxx results for model 5	45
Figure 34: Comparison of vx results for model 1	46
Figure 35: Comparison of vx results for model 2	46
Figure 36: Comparison of vx results for model 3	46
Figure 37: Comparison of vx results for model 4	46
Figure 38: Comparison of vx results for model 5	46
Figure 47:Actual deformed shape of models by shell code results (Test LM1-5).....	51
Figure 48:Actual deformed shape of models by finite element solution (SCIA Engineer 19).....	51
Figure 49: 3D surface and projections of uz plots from Test R1-5.....	52
Figure 50: 3D surface and projections of uz plots from Test LM1-5	52
Figure 51:Bending moment mxx edge results by shell code (Test LM1-5, m=n=50).....	53
Figure 52:Bending moment mxx edge results by finite element solution (SCIA Engineer 19)	54

Figure 53: Shear force vx edge results by shell code (Test LM1-5, m=n=50)	54
Figure 54: Shear force vx edge results by finite element solution (SCIA Engineer 19)	55
Figure 55: Plots of top 100 maximum and minimum values from Test LM1-5 (m=n=30)	56
Figure 56: Plots of top 100 maximum and minimum values from Test R1-5 (m=n=30)	57
Figure 57: Sparsity of matrices	59
Figure 58: Condition number of rectangular matrices	60
Figure 59: Condition number of square matrices	60
Figure 60: Percentage of rank number (%) of rectangular matrices (A and $[A b]$)	61
Figure 61: Percentage of rank number (%) of square matrices (A and $[A b]$)	61
Figure 62: Deviation of Test R1 results by increasing number of iterations (m=n=20)	62
Figure 63: Deviation of Test R3 results by increasing number of iterations (m=n=20)	62
Figure 64: Deviation of Test SE1 results by increasing number of iterations (m=n=20)	62
Figure 65: Deviation of Test SE3 results by increasing number of iterations (m=n=20)	62
Figure 66: Deviation of model 1 results with different unit systems	64
Figure 67: Deviation of model 2 results with different unit systems	64
Figure 68: Deviation of model 3 results with different unit systems	64
Figure 69: Deviation of model 4 results with different unit systems	64
Figure 70: Deviation of model 5 results with different unit systems	65
Figure 71: Condition number of rectangular matrices with different unit systems (m=n=15)	65
Figure 72: Percentage of rank number (%) of rectangular matrices (A and $[A b]$) with different unit systems (m=n=15)	65
Figure 73: Condition number of rectangular matrices with different approximation method (m=n=15)	66
Figure 74: Percentage of rank number (%) of rectangular matrices (A and $[A b]$) with different approximation method (m=n=15)	66
Figure 75: Tests on stopping tolerances in lsqr solver for Test R1 (m=n=30)	67
Figure 76: Tests on stopping tolerances in lsqr solver for Test R3 (m=n=30)	67
Figure 77: Tests on stopping tolerances in lsqr solver for Test SE1 (m=n=30)	67
Figure 78: Tests on stopping tolerances in lsqr solver for Test SE3 (m=n=30)	67
Figure 79: Test R1 displacement uz results (m=n=20, 30, 50)	73
Figure 80: Test R1 bending moment mxx results (m=n=20, 30, 50)	73
Figure 81: Test R1 shear force vx results (m=n=20, 30, 50)	73
Figure 82: Test R2 displacement uz results (m=n=20, 30, 50)	73
Figure 83: Test R2 bending moment mxx results (m=n=20, 30, 50)	74
Figure 84: Test R2 shear force vx results (m=n=20, 30, 50)	74
Figure 85: Test R3 displacement uz results (m=n=20, 30, 50)	74
Figure 86: Test R3 bending moment mxx results (m=n=20, 30, 50)	74
Figure 87: Test R3 shear force vx results (m=n=20, 30, 50)	75
Figure 88: Test R4 displacement uz results (m=n=20, 30, 50)	75
Figure 89: Test R4 bending moment mxx results (m=n=20, 30, 50)	75
Figure 90: Test R4 shear force vx results (m=n=20, 30, 50)	75
Figure 91: Test R5 displacement uz results (m=n=20, 30, 50)	76
Figure 92: Test R5 bending moment mxx results (m=n=20, 30, 50)	76
Figure 93: Test R5 shear force vx results (m=n=20, 30, 50)	76
Figure 94: Test SC1 displacement uz results (m=n= 30, 50)	76
Figure 95: Test SC1 bending moment mxx results (m=n=30, 50)	77
Figure 96: Test SC1 shear force vx results (m=n=30, 50)	77
Figure 97: Test SC2 displacement uz results (m=n=30, 50)	77
Figure 98: Test SC2 bending moment mxx results (m=n=30, 50)	77
Figure 99: Test SC2 shear force vx results (m=n= 30, 50)	78
Figure 100: Test SC3 displacement uz results (m=n=20, 30, 50)	78

Figure 101: Test SC3 bending moment m_{xx} results (m=n=20, 30, 50).....	78
Figure 102: Test SC3 shear force v_x results (m=n=20, 30, 50).....	78
Figure 103: Test SC4 displacement u_z results (m=n=20, 30, 50).....	79
Figure 104: Test SC4 bending moment m_{xx} results (m=n=20, 30, 50).....	79
Figure 105: Test SC4 shear force v_x results (m=n=20, 30, 50).....	79
Figure 106: Test SC5 displacement u_z results (m=n=20, 30, 50).....	79
Figure 107: Test SC5 bending moment m_{xx} results (m=n=20, 30, 50).....	80
Figure 108: Test SC5 shear force v_x results (m=n=20, 30, 50).....	80
Figure 109: Test SC1 displacement u_z results (m=n= 30, 50).....	80
Figure 110: Test SC1 bending moment m_{xx} results (m=n=30, 50).....	80
Figure 111: Test SC1 shear force v_x results (m=n=30, 50).....	81
Figure 112: Test SC2 displacement u_z results (m=n=30, 50).....	81
Figure 113: Test SC2 bending moment m_{xx} results (m=n=30, 50).....	81
Figure 114: Test SC2 shear force v_x results (m=n= 30, 50).....	81
Figure 115: Test SC3 displacement u_z results (m=n=20, 30, 50).....	82
Figure 116: Test SC3 bending moment m_{xx} results (m=n=20, 30, 50).....	82
Figure 117: Test SC3 shear force v_x results (m=n=20, 30, 50).....	82
Figure 118: Test SC4 displacement u_z results (m=n=20, 30, 50).....	82
Figure 119: Test SC4 bending moment m_{xx} results (m=n=20, 30, 50).....	83
Figure 120: Test SC4 shear force v_x results (m=n=20, 30, 50).....	83
Figure 121: Test SC5 displacement u_z results (m=n=20, 30, 50).....	83
Figure 122: Test SC5 bending moment m_{xx} results (m=n=20, 30, 50).....	83
Figure 123: Test SC5 shear force v_x results (m=n=20, 30, 50).....	84
Figure 124: Test SU1 displacement u_z results (m=n=20, 30, 50).....	84
Figure 125: Test SU1 bending moment m_{xx} results (m=n=20, 30, 50).....	84
Figure 126: Test SU1 shear force v_x results (m=n=20, 30, 50).....	84
Figure 127: Test SU2 displacement u_z results (m=n=20, 30, 50).....	85
Figure 128: Test SU2 bending moment m_{xx} results (m=n=20, 30, 50).....	85
Figure 129: Test SU2 shear force v_x results (m=n=20, 30, 50).....	85
Figure 130: Test SU3 displacement u_z results (m=n= 30, 50).....	85
Figure 131: Test SU3 bending moment m_{xx} results (m=n=30, 50).....	86
Figure 132: Test SU3 shear force v_x results (m=n=30, 50).....	86
Figure 133: Test SU4 displacement u_z results (m=n=30, 50).....	86
Figure 134: Test SU4 bending moment m_{xx} results (m=n=30, 50).....	86
Figure 135: Test SU4 shear force v_x results (m=n= 30, 50).....	87
Figure 136: Test SU5 displacement u_z results (m=n=30, 50).....	87
Figure 137: Test SU5 bending moment m_{xx} results (m=n=30, 50).....	87
Figure 138: Test SU5 shear force v_x results (m=n= 30, 50).....	87
Figure 139: Test SU1 displacement u_z results (m=n=20, 30, 50).....	88
Figure 140: Test SU1 bending moment m_{xx} results (m=n=20, 30, 50).....	88
Figure 141: Test SE1 shear force v_x results (m=n=20, 30, 50).....	88
Figure 142: Test SE2 displacement u_z results (m=n=20, 30, 50).....	88
Figure 143: Test SE2 bending moment m_{xx} results (m=n=20, 30, 50).....	89
Figure 144: Test SE2 shear force v_x results (m=n=20, 30, 50).....	89
Figure 145: Test SE3 displacement u_z results (m=n= 30, 50).....	89
Figure 146: Test SE3 bending moment m_{xx} results (m=n=30, 50).....	89
Figure 147: Test SE3 shear force v_x results (m=n=30, 50).....	90
Figure 148: Test SE4 displacement u_z results (m=n=30, 50).....	90
Figure 149: Test SE4 bending moment m_{xx} results (m=n=30, 50).....	90
Figure 150: Test SE4 shear force v_x results (m=n= 30, 50).....	90
Figure 151: Test SE5 displacement u_z results (m=n=30, 50).....	91
Figure 152: Test SE5 bending moment m_{xx} results (m=n=30, 50).....	91

Figure 153: Test SE5 shear force vx results (m=n= 30, 50)	91
Figure 154: Test P1 displacement uz results (m=n=10, 20)	91
Figure 155: Test P1 bending moment mxx results (m=n=10, 20)	92
Figure 156: Test P1 shear force vx results (m=n=10, 20).....	92
Figure 157: Test P2 displacement uz results (m=n=10, 20)	92
Figure 158: Test P2 bending moment mxx results (m=n=10, 20)	92
Figure 159: Test P2 shear force vx results (m=n=10, 20).....	93
Figure 160: Test P3 displacement uz results (m=n=10, 20)	93
Figure 161: Test P3 bending moment mxx results (m=n=10, 20)	93
Figure 162: Test P3 shear force vx results (m=n=10, 20).....	93
Figure 163: Test P4 displacement uz results (m=n=10, 20)	93
Figure 164: Test P4 bending moment mxx results (m=n=10, 20)	94
Figure 165: Test P4 shear force vx results (m=n=10, 20).....	94
Figure 166: Test P5 displacement uz results (m=n=10, 20, 30)	94
Figure 167: Test P5 bending moment mxx results (m=n=10, 20)	94
Figure 168: Test P5 shear force vx results (m=n=10, 20).....	95
Figure 169: Test LM1 displacement uz results (m=n= 20, 30, 50).....	95
Figure 170: Test LM1 bending moment mxx results (m=n= 20, 30, 50)	95
Figure 171: Test LM1 shear force vx results (m=n=20, 30, 50).....	95
Figure 172: Test LM2 displacement uz results (m=n=20, 30, 50).....	96
Figure 173: Test LM2 bending moment mxx results (m=n=20, 30, 50)	96
Figure 174: Test LM2 shear force vx results (m=n=20, 30, 50).....	96
Figure 175: Test LM3 displacement uz results (m=n=30, 50).....	96
Figure 176: Test LM3 bending moment mxx results (m=n=30, 50)	97
Figure 177: Test LM3 bending moment vx results (m=n=30, 50).....	97
Figure 178: Test LM4 displacement uz results (m=n=30, 50).....	97
Figure 179: Test LM4 bending moment mxx results (m=n=30, 50)	97
Figure 180: Test LM4 bending moment vx results (m=n=30, 50).....	98
Figure 181: Test LM5 displacement uz results (m=n=30, 50).....	98
Figure 182: Test LM5 bending moment mxx results (m=n=30, 50)	98
Figure 183: Test LM5 shear force vx results (m=n=30, 50).....	98
Figure 184: Test SLM1 displacement uz results (m=n=20, 30, 50)	99
Figure 185: Test SLM1 bending moment mxx results (m=n=20, 30, 50)	99
Figure 186: Test SLM1 shear force vx results (m=n=20, 30, 50).....	99
Figure 187: Test SLM2 displacement uz results (m=n=20, 30, 50)	99
Figure 188: Test SLM2 bending moment mxx results (m=n=20, 30, 50)	100
Figure 189: Test SLM2 shear force vx results (m=n=20, 30, 50).....	100
Figure 190: Test SLM3 displacement uz results (m=n=30, 50)	100
Figure 191: Test SLM3 bending moment mxx results (m=n=30, 50)	100
Figure 192: Test SLM3 bending moment vx results (m=n=30, 50)	101
Figure 193: Test SLM4 displacement uz results (m=n=30, 50)	101
Figure 194: Test SLM4 bending moment mxx results (m=n=30, 50)	101
Figure 195: Test SLM 4 bending moment vx results (m=n=30, 50)	101
Figure 196: Test SLM5 displacement uz results (m=n=30, 50)	102
Figure 197: Test SLM5 bending moment mxx results (m=n=30, 50)	102
Figure 198: Test LM5 shear force vx results (m=n=30, 50).....	102
Figure 199: Test ALM1 displacement uz results (m=n= 20, 30, 50).....	102
Figure 200: Test ALM1 bending moment mxx results (m=n= 20, 30, 50).....	103
Figure 201: Test ALM1 shear force vx results (m=n=20, 30, 50)	103
Figure 202: Test ALM2 displacement uz results (m=n=20, 30, 50).....	103
Figure 203: Test ALM2 bending moment mxx results (m=n=20, 30, 50).....	103
Figure 204: Test ALM2 shear force vx results (m=n=20, 30, 50).....	104

Figure 205: Test ALM3 displacement uz results (m=n=30, 50).....	104
Figure 206: Test ALM3 bending moment mxx results (m=n=30, 50).....	104
Figure 207: Test ALM3 bending moment vx results (m=n=30, 50).....	104
Figure 208: Test ALM4 displacement uz results (m=n=30, 50).....	105
Figure 209: Test ALM4 bending moment mxx results (m=n=30, 50).....	105
Figure 210: Test ALM4 bending moment vx results (m=n=30, 50).....	105
Figure 211: Test ALM5 displacement uz results (m=n=30, 50).....	105
Figure 212: Test ALM5 bending moment mxx results (m=n=30, 50).....	106
Figure 213: Test ALM5 shear force vx results (m=n=30, 50).....	106
Figure 214: Test BLM1 displacement uz results (m=n= 20, 30, 50).....	106
Figure 215: Test BLM1 bending moment mxx results (m=n= 20, 30, 50).....	106
Figure 216: Test BLM1 shear force vx results (m=n=20, 30, 50).....	107
Figure 217: Test BLM2 displacement uz results (m=n=20, 30, 50).....	107
Figure 218: Test BLM2 bending moment mxx results (m=n=20, 30, 50).....	107
Figure 219: Test BLM2 shear force vx results (m=n=20, 30, 50).....	107
Figure 220: Test BLM3 displacement uz results (m=n=30, 50).....	108
Figure 221: Test BLM3 bending moment mxx results (m=n=30, 50).....	108
Figure 222: Test BLM3 bending moment vx results (m=n=30, 50).....	108
Figure 223: Test BLM4 displacement uz results (m=n=30, 50).....	108
Figure 224: Test BLM4 bending moment mxx results (m=n=30, 50).....	109
Figure 225: Test BLM4 bending moment vx results (m=n=30, 50).....	109
Figure 226: Test BLM5 displacement uz results (m=n=30, 50).....	109
Figure 227: Test BLM5 bending moment mxx results (m=n=30, 50).....	109
Figure 228: Test BLM5 shear force vx results (m=n=30, 50).....	110
Figure 229: Test AR1 displacement uz results (m=n= 20, 30, 50).....	110
Figure 230: Test AR1 bending moment mxx results (m=n= 20, 30, 50).....	110
Figure 231: Test AR1 shear force vx results (m=n=20, 30, 50).....	110
Figure 232: Test AR2 displacement uz results (m=n=20, 30, 50).....	111
Figure 233: Test AR2 bending moment mxx results (m=n=20, 30, 50).....	111
Figure 234: Test AR2 shear force vx results (m=n=20, 30, 50).....	111
Figure 235: Test AR3 displacement uz results (m=n=30, 50).....	111
Figure 236: Test AR3 bending moment mxx results (m=n=30, 50).....	112
Figure 237: Test AR3 bending moment vx results (m=n=30, 50).....	112
Figure 238: Test AR4 displacement uz results (m=n=30, 50).....	112
Figure 239: Test AR4 bending moment mxx results (m=n=30, 50).....	112
Figure 240: Test AR4 bending moment vx results (m=n=30, 50).....	113
Figure 241: Test AR5 displacement uz results (m=n=30, 50).....	113
Figure 242: Test AR5 bending moment mxx results (m=n=30, 50).....	113
Figure 243: Test AR5 shear force vx results (m=n=30, 50).....	113
Figure 244: Test BR1 displacement uz results (m=n= 20, 30, 50).....	114
Figure 245: Test BR1 bending moment mxx results (m=n= 20, 30, 50).....	114
Figure 246: Test BR1 shear force vx results (m=n=20, 30, 50).....	114
Figure 247: Test BR2 displacement uz results (m=n=20, 30, 50).....	114
Figure 248: Test BR2 bending moment mxx results (m=n=20, 30, 50).....	115
Figure 249: Test BR2 shear force vx results (m=n=20, 30, 50).....	115
Figure 250: Test BR3 displacement uz results (m=n=30, 50).....	115
Figure 251: Test BR3 bending moment mxx results (m=n=30, 50).....	115
Figure 252: Test BR3 bending moment vx results (m=n=30, 50).....	116
Figure 253: Test BR4 displacement uz results (m=n=30, 50).....	116
Figure 254: Test BR4 bending moment mxx results (m=n=30, 50).....	116
Figure 255: Test BR4 bending moment vx results (m=n=30, 50).....	116
Figure 256: Test BR5 displacement uz results (m=n=30, 50).....	117

Figure 257: Test BR5 bending moment m_{xx} results ($m=n=30, 50$)	117
Figure 258: Test BR5 shear force v_x results ($m=n=30, 50$).....	117

List of tables

Table 1: Sanders-Koiter equations (Hoogenboom , 2021).....	7
Table 2: Boundary conditions for an edge in the x direction and the y axis pointing outwards.....	8
Table 3: Boundary conditions for an edge in the x direction and the y axis pointing inwards.....	8
Table 4: Boundary conditions for an edge in the y direction and the x axis pointing outwards.....	8
Table 5: Boundary conditions for an edge in the y direction and the x axis pointing inwards.....	8
Table 6: Geometry and material parameters of models	20
Table 7: Pinned edges and Cantilever boundary conditions	20
Table 8: Boundary equations	21
Table 9: Model configuration.....	21
Table 10: Number of assigned equations in rectangular [M].....	25
Table 11: Number of assigned equations in square [M] by central node method.....	27
Table 12: Number of assigned equations in square [M] by undefined node method	28
Table 13: Replaced S-K equations and boundary equations correlation	30
Table 14: Number of assigned equations in square [M] by undefined node method	30
Table 15: Additional boundary equations	30
Table 16: Rectangular and square matrix test configuration	30
Table 17a: Solver test configuration of rectangular matrices	31
Table 17b: Solver tests of square matrices (The matrices are made square by the equation replacement method)	32
Table 18: Absolute maximum value of finite element results for model1-5	36
Table 19: Absolute maximum value of test R1-5 plots.....	37
Table 20: Absolute maximum value of test SC1-5 plots	37
Table 21: Absolute maximum value of test SU1-5 plots	37
Table 22: Absolute maximum value of test SE1-5 plots.....	37
Table 23: Absolute maximum value of test P1-5.....	38
Table 24: Absolute maximum value of test LM1-5 plots	38
Table 25: Absolute maximum value of test SLM1-5 plots	38
Table 26: Absolute maximum value of test AR1-5 plots.....	39
Table 27: Absolute maximum value of test BR1-5 plots.....	39
Table 28: Absolute maximum value of test ALM1-5 plots	39
Table 29: Absolute maximum value of test BLM1-5 plots	39
Table 31: Deviation of Test R1-5 results (green:14%, blue: 42%, orange: 44%)	40
Table 32: Deviation of Test SU1-5 results (green:12%, blue: 44%, orange: 44%).....	40
Table 33: Deviation of Test SC1-5 results (green:7%, blue: 40%, orange: 53%)	40
Table 34: Deviation Test SE1-5 results (green:12%, blue: 44%, orange: 44%).....	41
Table 35: Deviation of Test P1-5 (green:33%, blue: 67%, orange: 0%)	41
Table 37: Deviation of Test LM1-5 results (green:61%, blue: 39%, orange: 0%)	41
Table 36: Deviation of Test SLM1-5 (green:53%, blue: 42%, orange: 5.7%)	41
Table 36: Deviation of Test ALM1-5 (green:12%, blue: 44%, orange: 44%).....	42
Table 36: Deviation of Test BLM1-5 (green:78%, blue: 19%, orange: 3%).....	42
Table 40: Summary of deviation of test results	42
Table 37: Comparison between lsqr solver results	47
Table 38: Comparison between pinv solver and lm.fit.sparse solver results	48
Table 39: Comparison between five-point and two-point difference approximation results (lsqr solver)	48
Table 40: Comparison between five-point and two-point difference approximation results (lm.fit.sparse solver).....	49

Table 41: Comparison between bending moment m_{xx} edge results	52
Table 42: Comparison between shear force v_x edge results	53
Table 43: Deviation of Test LM1-5 overall bending moment & shear force results ($m=n=50$).....	53
Table 44: Memory usage and time by finite element software (SCIA Engineer 19).....	58
Table 45: Memory usage and time by different solver in shell code	58
Table 46: Model parameters by different unit systems	63
Table 47: Sparsity of rectangular matrices.....	117
Table 48: Sparsity of square matrices	117
Table 49: Condition number of rectangular matrices	117
Table 50: Condition number of square matrices	118
Table 51: Rank number of rectangular matrices	118
Table 52: Rank number of square matrices.....	118
Table 53: Deviation of Test R1 results by increasing number of iterations ($m=n=20$)	118
Table 54: Deviation of Test R3 results by increasing number of iterations ($m=n=20$)	118
Table 55: Deviation of Test SE1 results by increasing number of iterations ($m=n=20$)	118
Table 56: Deviation of Test SE3 results by increasing number of iterations ($m=n=20$)	118
Table 57: Deviation of Test R1-5 results by new unit systems (N, mm).....	118
Table 58: Deviation of Test LM1-5 results by new unit systems (N, mm)	119
Table 59: Deviation of Test R1-5 results by new unit systems (KN, 10m).....	119
Table 60: Deviation of Test LM1-5 results by new unit systems (KN, 10m).....	119

1. Introduction

1.1 Problem Statement

The finite element method is the industry standard for analysing shell structures due to its generality and sophistication. In popular commercial finite element software, the most used shell element type has been derived from a solid, therefore, the shell differential equations (for example, Sanders-Koiter equations) have not been used. In fact, currently, there is not a method available for solving the shell differential equations. Solutions to those equations could be used to perform independent checks of finite element results. It can be expected that the finite difference results will be the same as the finite element results, however, there might be theoretically interesting differences, for example in edge stresses. This can provide insight into both the Sanders-Koiter equations and the applied finite elements. Although the finite element method is a mature method with a long history of application, it is always good to try and falsify theories. For this purpose, a direct way to solve the shell differential equations is required.

The simplest way to solve differential equations is the finite difference method. This method has a long history of application. For example, for plate problems the finite difference method was applied long before the finite element method (Figure 1). The finite difference method was already used in hand calculations before the development of electronic computers (Thomé, 2001). However, its application to shell theory was always considered impractical. In shell theory many higher-order differentiations occur and the grid is curvilinear, which means that the discretized form of those equations is large and different for every grid point. Nonetheless, there must be a practical, even simple manner to apply the finite difference method to shell theory.

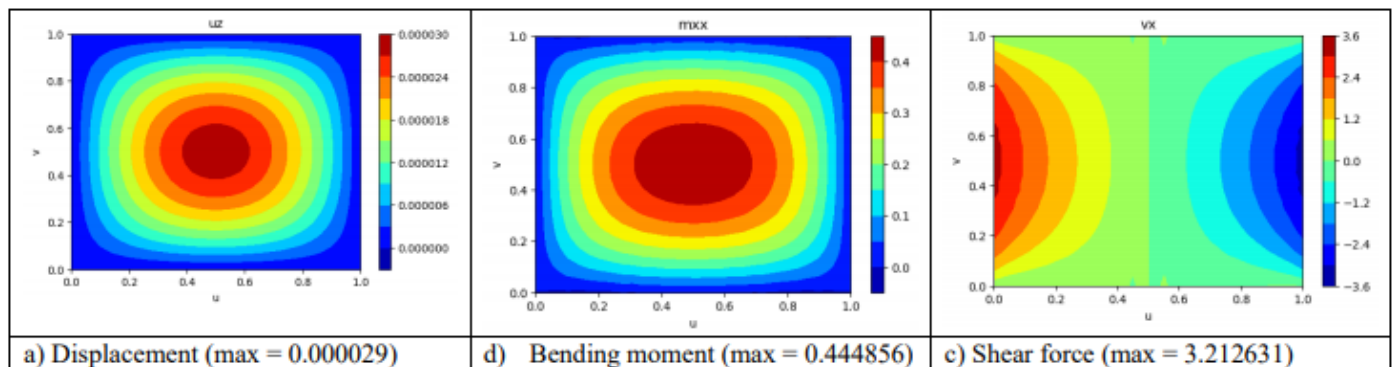


Figure 1: Result plots of uniformly loaded two-way slab 30*30 nodes (Li, 2020)

One simple finite difference method is applied in a Python algorithm, called *plate code*, that has been recently developed by the author (Li, 2020). The method is simple because only first-order derivatives are used instead of the common forth-order derivatives. The advantage is that few simple molecules need to be implemented for the various boundary conditions. The disadvantage is that the matrix constructed is very large, which gives memory capacity problems and is time-consuming. This method was shown to work well, however, there was still an unsolved challenge: The number of discretized equations exceeds the number of unknown, which leads to a rectangular matrix, which is solved in a least square approximation. The least square approximation may cut off peaks in the solution, for example it may cut of moment peaks or membrane force peaks. It should be possible to remove some of the equations and make the matrix square without it becoming singular.

1.2 Objective

Based on the previously explored method of solving plates, the objective of this research is 1) to develop a practical finite difference algorithm for solving the Sanders-Koiter equations for any shell model for available orthogonal parameterization and 2) understand how the algorithm results can be affected by various factors.

This shell code should have the following key features:

1. Use the finite difference method to approximate the Sanders-Koiter equations
2. Results (extreme values, contour plots) agree with finite element solutions or analytical solutions
3. Universally applicable to shell models with different geometries, loads, and boundary conditions
4. Practical computation time and memory usage
5. Square matrix to avoid computing over-determined systems
6. Easy to modify in case of future changes in the Sanders-Koiter equations or the boundary conditions

1.3 Approach

To develop an algorithm with the above key features, the development process of this code is mainly divided into the following steps. These steps describe the internal logic of the *shell code* and the work method experienced by the author. The methodology and encountered difficulties mentioned below are summarized from daily testing and coding the *shell code* program. At later stage of *shell code* development, many plots and much data were obtained. To understand how the algorithm results can be affected by various factors, a number of tests were organized and their results were compared. Those results and comparisons were evidence of the feasibility of this algorithm. Based on those results, errors in the code were spotted and corrected. Meanwhile, new methods and new concepts were tried to improve the performance of the algorithm.

a) Build the code

The first step to develop this new shell code is to extrapolate the verified method from the plate code (Li, 2020). Many ideas to develop this algorithm have been verified in the previous plate code including ideas on how to add model equations, define boundary conditions, and correctly approximate differentiation with the finite difference method. It is worth mentioning that the fundamental concept of solving Sanders-Koiter equations by finite difference method is from an algorithm developed by Dr. Hoogenboom which was not successful yet at early development. The number of implemented equations in shell code is nearly twice that used in plate code and they have more components involved. In shell code, the model body requires 21 Sanders-Koiter (S-K) equations (plates 11) and every edge requires 4 boundary equations (plates 2). As the most fundamental part of shell code, correctly adding equations for every node on the grid of the model is the first challenge to be solved. Meanwhile, if a square matrix is required for testing, the specific method of replacing model equations should be studied. After finishing constructing the matrix, a proper type of solver should be selected to solve the system which may directly determine the quality of results. During this phase, most time was spend on the mathematical interpretation of finite difference method and the S-K equations and how to implement them in Python coding. The challenging part was on how to use programming to realize the mathematical concepts and structural mechanics concepts.

b) Test the code

In order to prove the general universality of shell code, this new algorithm should be able to solve different shell model problems with various material properties, geometry shapes, boundary conditions, and load cases. For this reason, a number of tests was set up for testing shell code with different shell model problems to prove the shell code can correctly convert models into matrix systems and solve them. Meanwhile, tests were also organized to investigate other potential factors which might affect the code results like the number of nodes, type of solvers, and type of matrices. If the shell code could work properly, plots of displacement, shear force, bending moment, and other results were generated and collected after each test. Other information like the setup of tests, spent time, and memory usage of the shell code were also recorded. A large group of extreme values from every generated plot was collected and analyzed.

c) Validate the results

The above test results were used to prove three properties of code results: accuracy, reliability, and efficiency. To prove accuracy, test results were used to compare with external finite element results. Those finite element results were obtained from a popular commercial finite element software (SCIA Engineer) where the same shell models were analysed. The difference between extreme values of displacement, shear force, bending

moment results from shell code and finite element software were calculated. They are listed and categorized in terms of the type of model, the number of nodes, type of matrix, and types of solver in order to show the accuracy of shell code results and extent of participation of each factor in affecting the accuracy. To prove the reliability of results, the collected extreme values of displacement, shear force and bending moment were plot and reviewed for any possible spike in the trend toward an infinite value (singularity). If such a case occurred, this extreme value should be considered as a computational error and be excluded from comparing with finite element results. To prove efficiency, spent time and memory usage for each running test were collected and listed in terms of the type of model, the number of nodes, type of matrix, and types of the solver. By comparing them with each other, factors that affect the efficiency of code were identified. Spent time and memory usage were also compared with those of a popular commercial finite element software to show whether the shell code is efficient for practical use.

d) Improve the code

After obtaining results from the initial version of the workable shell code, the performance of shell code, including accuracy, reliability, and efficiency mentioned above, were improved if they were not in an acceptable range. Clearly, this is a common issue at the initial stage of code development. The first step is to check with the basic setting for shell model parameters or the adding process of S-K equations for model body and boundary condition to ensure that the purposed model is correctly described. By iterating this step for different tests which have various setting and requirement, those basic error related to describing model were revised. Then the next step is to select a proper solver for solving the matrix system. The selection of the solver is based on the properties and formation process of input matrices. If a square matrix is not required, then the constructed matrices will be rectangular matrices, an overdetermined linear sparse system. If the square matrix is needed, this square matrix is still a linear sparse system, but possibly a singular matrix. Further research needs to be done to understand the characteristics of different solvers in order to select a suitable solver for each test. The third step is to optimize the application of the S-K equations by applying additional definitions for parts of the model. It is possible that the application of the S-K equations might be limited by the mathematic property of the finite difference method or other factors. Therefore, additional equations were used as theoretic reinforcement for correcting a potential error in edge or corner behaviour. For example, the free corners or edges of models require additional equations for defining boundary conditions. Additionally, new finite difference methods were tried since the performance of shell code is directly related to the finite difference method applied. At the early stages of code development, a two-point difference approximation was used to replace first derivative in Sanders-Koiter equations. To investigate the effect of computing truncation errors arising in this process, a five-point difference approximation was also tested and its results were compared to that of the previous method.

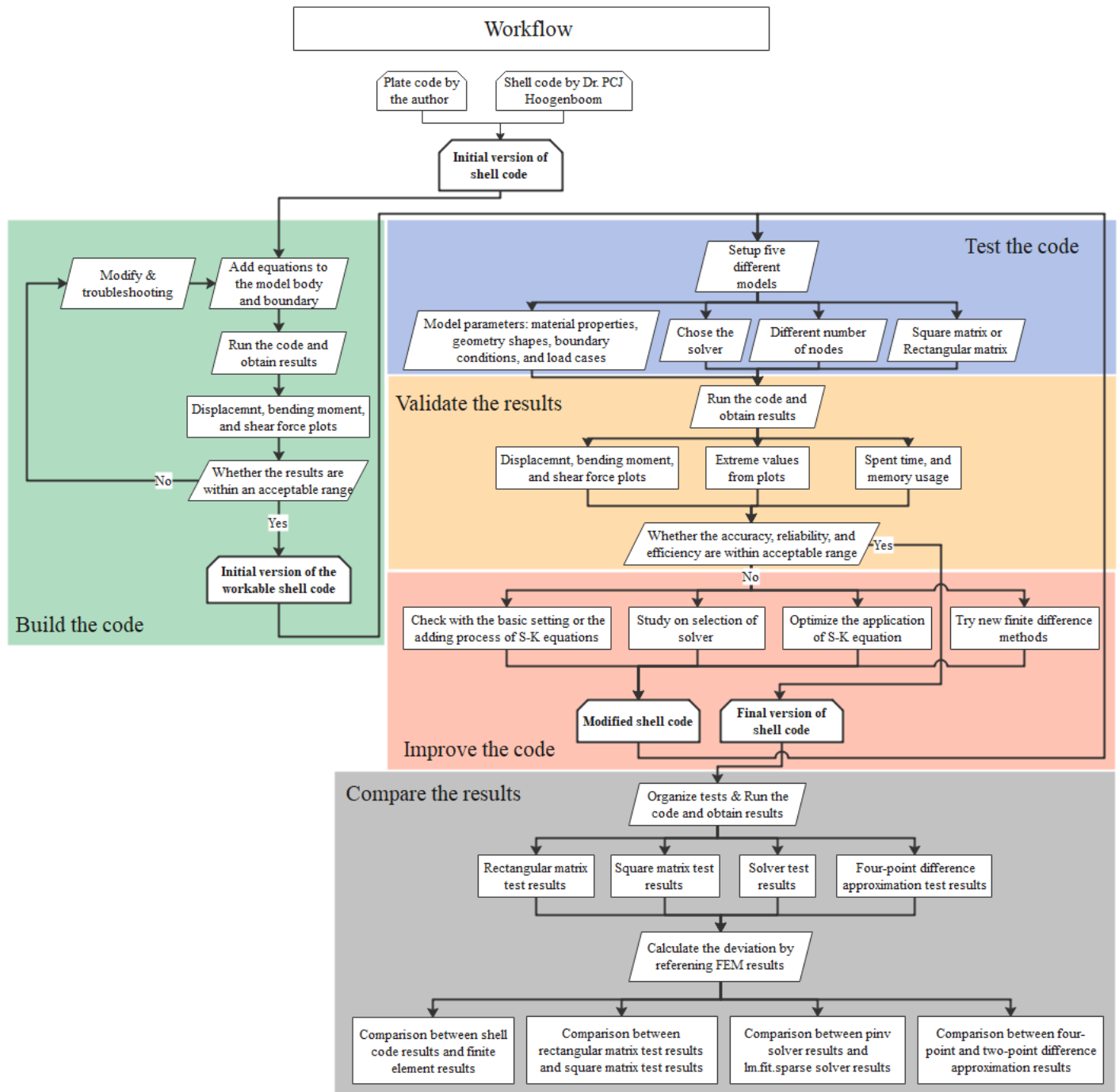
e) Compare the results

After going through the above development process, the shell code produced massive data and plots. In order to display them in an orderly manner, the main results are categorized into four test groups: rectangular matrix tests, square matrix tests, solver tests, and five-point difference approximation tests. For each group of tests, five shell models with various material properties, geometry shapes, boundary conditions, and load cases are used. Meanwhile, at least two levels of the number of nodes were used for each model. In the rectangular matrix tests, the matrix type is rectangular matrix and the selected solver is *lsmr* which can solve a linear model by least square estimation (Fong and Saunders, 2011). In square matrix tests, there are three types of square matrices generated by different methods and the selected solver was the same as that of rectangular matrix tests. In solver tests, rectangular matrix and square matrix were both used and two types of the solver (*pinv*, *lm.fit.sparse*) were used. The *lm.fit.sparse* solver can solve the system as a sparse linear fitting problem. The *pinv* solver aims to provide approximated inversion of matrices by SVD decomposition (Davis, Rajamanickam and Sid-Lakhdar, 2016). In the above tests, only two-point difference approximation was used. In five-point difference approximation tests, two types of five-point approximation and two types of solvers (*lsmr*, *lm.fit.sparse*) were used. To show those results more properly, the first step is to calculate the deviation of the obtained extreme values by using finite element results as the reference values. The following comparisons were made: between shell code results and finite element results, between rectangular matrix test and square matrix test results, between *pinv* solver and *lm.fit.sparse* solver results, and between five-point and two-point difference approximation results. By reviewing those comparisons, it was found how those factors

could affect shell code results. Additionally, the calculated deviation of shell code results was also categorized by the type of model, to investigate how the model setting parameters could affect shell code results.

1.4 Research workflow

As shown in the above section, the shell code development can be divided into five sections: build the code, test the code, validate the results, improve the code, and compare the results. In the workflow, those sections are repeated a number of times until the shell code satisfies the purposed key features. The below flow chart shows how the work has been done for the shell code development.



2. Literature review

This section aims to provide more information on the theoretical knowledge applied to this algorithm. The below content includes a short review of general plate and shell theories and a short description of the development of the S-K equations. Those equations are bounding rules for the behavior of the shell model which can provide an argument when discussing abnormality among the generated shell code results. There is also a short review on the finite difference method and its application. Those equations are the mathematical tools used to approximate first-order derivatives that appear in the S-K equations. The inherited computational error of the finite difference method could be important for shell code result analysis. The theoretical knowledge on types of matrix systems is also included. Some explanation on major methods of solving sparse overdetermined linear system through programming language is also included. That information can be used to determine the proper solver for different types of matrices and predict the solver capacity in order to select the best solver.

2.1 A short review on the development of plate and shell theories

The term shell refers to a physical body bounded by two curved surfaces whose distance is smaller than other dimensions. The distance between the surfaces is called the thickness of the shell (t). The imaginative surface that divide the thickness into equal halves is called middle surface. Such geometry structure makes the shell structure has excellent performance in terms of strength / weight ratio and bearing efficiency.

The study on shell started can be dated back to the free vibration analysis of plate problems performed by Euler (1766). Plate can be viewed as the shell without the curvature brought by curved surfaces. Then J. Bernoulli (1789) presented a plate model in an attempt to theoretically explain those results. In his model, plates were described as the combination of mutually perpendicular Euler–Bernoulli beams at right angles. Furthermore, French mathematician Germain (1826) developed a plate differential equation which mathematically describe the deformation of plate. The missing term for warping behaviour was added later by her reviewer Lagrange (1828). The completed form of this equation is the well-known Germain–Lagrange equation:

$$\frac{\partial^4 w}{\partial x^4} + 2 \frac{\partial^4 w}{\partial x^2 \partial y^2} + \frac{\partial^4 w}{\partial y^4} = \frac{q}{D}$$

where D is the flexural rigidity of the plate, w is the deflection of the plate, h is the plate's thickness, and q is the uniform distributed load.

This equation as the governing differential equation for deflections can also be formulated based on general equations of theory of elasticity. Cauchy (1828) and Poisson (1828) were the first to do so. The theory of bending of plates was improved by Navier (1823), who considered flexural rigidity of the plate D in the above equation as a function of the plate thickness. Later, Kirchhoff published an important thesis on the theory of thin plates where he introduced physical meaning into the theory of plates by the famous “Kirchhoff's hypotheses” (1850). Kirchhoff's hypotheses are a series of fundamental assumptions used in thin plate bending theory in which the deflection of a plate is assumed to be small, linear, and elastic. Restated assumptions (Ventsel and Krauthammer, 2001) are list as below:

- ✧ Elastic, homogenous, and isotropic material.
- ✧ Initially flat plate.
- ✧ Small vertical deflection of the midplane compared with the thickness of the plate.

- ✧ “Needle hypotheses”: The normal lines of the middle plane remain straight and normal to the middle surface during the deformation. Thickness remains constant. Negligible vertical shear strains (γ_{xy} , γ_{yz}) and normal strain (ϵ_z).
- ✧ Negligible normal stress σ_z
- ✧ Middle surface remains unstrained

Timoshenko also made a profound contribution to the plate bending theory and application of it by providing solutions of circular plates considering large deflections (1915) and formulation of elastic stability problems (1913). Moreover, he and Woinowsky-Krieger published a fundamental monograph (1959) which provided several solutions to various plate bending problems.

The above-mentioned classical plate theory mainly focused on the describing bending and twisting behavior of plate models while shell models are usually deformed in another way. It is because the shell has an additional characteristic than a plate –curvature. It makes the deformation of the shell model is predominantly induced by in-plane stressing. Depending on their curvatures, shells can be categorized as cylindrical (non-circular and circular), conical, spherical, ellipsoidal, paraboloidal, toroidal, and hyperbolic paraboloidal model. The most available commercial finite element software is merely capable to solve shell model problems without considering larger deformation and inelastic behaviour. In order to avoid the difficulties in solving 3D shell models, alternative 2D shell theories are to be more commonly used where shell problems are usually reduced to the study of deformations of the middle surface. It is a practical and efficient way to solve shell model problems as long as the above hypotheses were satisfied. However, those simplified elasticity equations reduce the accuracy of analysis results since some degrees of freedom of the model are omitted.

The first successful approximated shell theory was developed by Love (1892). He took constitutive relations from Kirchhoff’s hypotheses with his own assumption of small deflection and thinness of a shell to simplify the strain–displacement relationships in shell models. This theory, called Kirchhoff–Love shell theory, is a first-order approximation method to solve shell models. However, it is not a perfect theory which have several inconsistencies. After that, many other first-order approximation theories were later developed based on it. One of those later theories is Reissner’s linear theory of thin shells (1941). By taking the equilibrium equations, strain–displacement relations, and stress resultants expression from the three-dimensional theory of elasticity, the inconsistencies in Love’s theory were eliminated in Reissner’s theory. And Sanders develop his own first-order approximation shell theory from the principle of virtual work equation and it also successfully removed inconsistencies in Love’s theory.

2.2 Sanders-Koiter equations

The Sanders-Koiter equations have been individually developed by Sanders (Sanders, 1963) and Koiter (1966) as a refined nonlinear theory of shells. The accuracy of the Sanders-Koiter theory for calculating larger vibration amplitudes has been proved (Amabili, 2003).

Symbols

E	Young’s modulus
t	Shell thickness
α_x, α_y	Lamé parameters
k_x, k_y	In plane curvature of parameter lines
k_{xx}, k_{yy}, k_{xy}	Curvature tensor
m_{xx}, m_{yy}, m_{xy}	Moment tensor
$n_{xx}, n_{yy}, n_{xy}, n_{yx}$	Membrane force tensor

p_x, p_y, p_z	Distributed load
v_x, v_y, v_z	Out of plane shear forces
$\varepsilon_{xx}, \varepsilon_{yy}, \gamma_{xy}$	Strain tensor of the middle surface
$\varphi_x, \varphi_y, \varphi_z$	Rotation of a pin perpendicular to the surface
$\kappa_{yy}, \kappa_{xx}, \rho_{xy}$	Curvature deformation tensor
ν	Poisson's ratio
u, v	Curvilinear coordinates
u_x, u_y, u_z	Displacements
x, y, z	Local Cartesian coordinates

Table 1: Sanders-Koiter equations (Hoogenboom, 2021)

Equilibrium equations	$k_{xx}n_{xx} + k_{xy}(n_{xy} + n_{yx}) + k_{yy}n_{yy} + \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + k_y v_x + k_x v_y + p_z = 0$	1
	$\left(\frac{n_{xx}}{\partial x} + \frac{n_{yx}}{\partial y}\right) + k_y(n_{xx} - n_{yy}) + k_x(n_{xy} + n_{yx}) - k_{xx}v_x - k_{xy}v_y + p_x = 0$	2
	$\left(\frac{n_{yy}}{\partial y} + \frac{n_{xy}}{\partial x}\right) + k_x(n_{yy} - n_{xx}) + k_y(n_{xy} + n_{yx}) - k_{yy}v_y - k_{xy}v_x + p_y = 0$	3
	$\left(\frac{m_{xx}}{\partial x} + \frac{m_{xy}}{\partial y}\right) + k_y(m_{xx} - m_{yy}) + 2k_x m_{xy} - q_x = 0$	4
	$\left(\frac{m_{yy}}{\partial y} + \frac{m_{xy}}{\partial x}\right) + k_x(m_{yy} - m_{xx}) + 2k_y m_{xy} - v_y = 0$	5
	$k_{xy}(m_{xx} - m_{yy}) - (k_{xx} - k_{yy})m_{xy} + n_{xy} - n_{yx} = 0$	6
Constitutive equations	$n_{xx} = \frac{Et}{1-\nu^2}(\varepsilon_{xx} + \nu\varepsilon_{yy})$	7
	$n_{yy} = \frac{Et}{1-\nu^2}(\varepsilon_{yy} + \nu\varepsilon_{xx})$	8
	$\frac{n_{xy} + n_{yx}}{2} = \frac{Et}{2(1+\nu)}\gamma_{xy}$	9
	$m_{xx} = \frac{Et^3}{12(1-\nu^2)}(\kappa_{xx} + \nu\kappa_{yy})$	10
	$m_{yy} = \frac{Et^3}{12(1-\nu^2)}(\kappa_{yy} + \nu\kappa_{xx})$	11
	$m_{xy} = \frac{Et^3}{24(1+\nu)}\rho_{xy}$	12
Kinematic equations	$\varepsilon_{xx} = \frac{\partial u_x}{\partial x} - k_{xx}u_z + k_x u_y$	13
	$\varepsilon_{yy} = \frac{\partial u_y}{\partial y} - k_{yy}u_z + k_y u_x$	14
	$\gamma_{xy} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} - 2k_{xy}u_z - k_x u_x - k_y u_y$	15

	$\phi_x = -\frac{\partial u_z}{\partial x} - k_{xx}u_x - k_{xy}u_y$	16
	$\phi_y = -\frac{\partial u_z}{\partial y} - k_{yy}u_y - k_{xy}u_x$	17
	$\phi_z = \frac{1}{2} \left(-\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} - k_x u_x + k_y u_y \right)$	18
	$\kappa_{xx} = \frac{\partial \phi_x}{\partial x} - k_{xy}\phi_z + k_x\phi_y$	19
	$\kappa_{yy} = \frac{\partial \phi_y}{\partial y} + k_{xy}\phi_z + k_y\phi_x$	20
	$\rho_{xy} = \frac{\partial \phi_x}{\partial y} + \frac{\partial \phi_y}{\partial x} + (k_{xx} - k_{yy})\phi_z - k_x\phi_x - k_y\phi_y$	21

Boundary conditions for shell are defined as follows:

Table 2: Boundary conditions for an edge in the x direction and the y axis pointing outwards

Type Kinematic (K)	or	Type Dynamic (D)	
Impose displacement u_x		apply line load $q_x = n_{yx} - k_{xx}V$	BC1
Impose displacement u_y		apply line load $q_y = n_{yy} - k_{xy}V$	BC2
Impose displacement u_z		apply line load $q_z = v_y + \frac{\partial V}{\partial x}$	BC3
Impose displacement $-\phi_y$		apply line load $-m_{yy}$	BC4

Table 3: Boundary conditions for an edge in the x direction and the y axis pointing inwards

Type Kinematic (K)	or	Type Dynamic (D)	
Impose displacement u_x		apply line load $q_x = -n_{yx} + k_{xx}V$	BC5
Impose displacement u_y		apply line load $q_y = -n_{yy} + k_{xy}V$	BC6
Impose displacement u_z		apply line load $q_z = -v_y - \frac{\partial V}{\partial x}$	BC7
Impose displacement $-\phi_y$		apply line load m_{yy}	BC8

Table 4: Boundary conditions for an edge in the y direction and the x axis pointing outwards

Type Kinematic (K)	or	Type Dynamic (D)	
Impose displacement u_x		apply line load $q_x = n_{xx} - k_{xy}V$	BC9
Impose displacement u_y		apply line load $q_y = n_{xy} - k_{yy}V$	BC10
Impose displacement u_z		apply line load $q_z = v_x + \frac{\partial V}{\partial y}$	BC11
Impose displacement ϕ_x		apply line load m_{xx}	BC12

Table 5: Boundary conditions for an edge in the y direction and the x axis pointing inwards

Type Kinematic (K)	or	Type Dynamic (D)	
Impose displacement u_x		apply line load $q_x = -n_{xx} + k_{xy}V$	BC13

Impose displacement u_y	or	apply line load $q_y = -n_{xy} + k_{yy}V$	BC14
Impose displacement u_z	or	apply line load $q_z = -v_x - \frac{\partial V}{\partial y}$	BC15
Impose displacement φ_x	or	apply line load $-m_{xx}$	BC16

2.3A short review on the finite different method

The idea of the finite difference method is studying the continuous process by applying mathematical discretization. By dividing the process into a finite number of sufficiently small parts, the differential equations are approximated by a large number of linear equations. The results of derivative over a continuous domain can be approximated as the summation of a weight function multiplied with results of discrete points.

The general procedure of applying different finite difference schemes for the numerical solution of partial differential equation is outlined as below:

- ✧ Convert the continuous process variables into a discrete set of points.
- ✧ Approximate partial derivatives using finite difference approximation.
- ✧ Solve the resulting finite difference equations.

By definition, the first order derivative can also be calculated as below. The first order derivative of a one-dimensional, continuous function $f(x)$ is calculated based on values of adjacent points

$$f' = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} = \lim_{h \rightarrow 0} \frac{f(x) - f(x-h)}{h} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

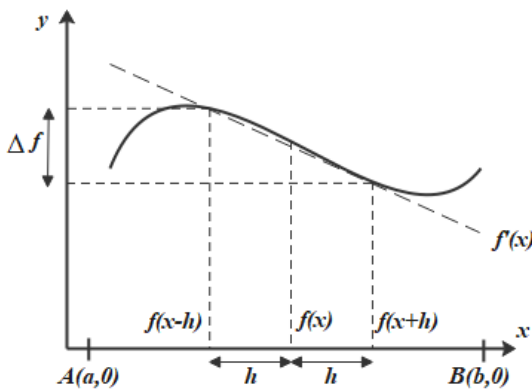


Figure 2: First order derivative by finite difference method

The key point of the finite difference method is approximating the derivatives. For example, the domain of variable x of the continuous function $f(x)$ is interval AB . The interval AB starts at point $A(a, 0)$ and ends at $B(b, 0)$ which is divided into equal intervals $\Delta x = h$. h is the finite increment of the variable x . Assume the

$f(x)$ is linear continuous function with expression of $f(x) = a + b \cdot x + c \cdot x^2$.

The first order derivative of $f(x)$ is given by below calculation:

$$\left\{ \begin{array}{l} f_{i-1} = f(x-h) \\ f_i = f(x) \\ f_{i+1} = f(x+h) \end{array} \right\} \Rightarrow \left\{ b = -\frac{f_{i-1} - f_{i+1}}{2h} \right\} \Rightarrow \frac{\partial f(x)}{\partial x} = b = \frac{f_{i-1} - f_{i+1}}{2h}$$

In practice, it is called the expression for a central difference approximation of $(\partial f(x)/\partial x)$. It is also referred to as the two-point derivative approximation since the calculation is based on the values of two adjacent points. If the two adjacent points are on the one side, it is called the one-sided finite difference approximation which is calculated as below:

$$\left\{ \begin{array}{l} f_i = f(x) \\ f_{i+1} = f(x+h) \\ f_{i+2} = f(x+2h) \end{array} \right\} \Rightarrow \left\{ b = \frac{3 \cdot f_i - 4 \cdot f_{i+1} + f_{i+2}}{2h} \right\} \Rightarrow \frac{\partial f(x)}{\partial x} = b = \frac{3 \cdot f_i - 4 \cdot f_{i+1} + f_{i+2}}{2h}$$

$$\left\{ \begin{array}{l} f_i = f(x) \\ f_{i-1} = f(x-h) \\ f_{i-2} = f(x-2h) \end{array} \right\} \Rightarrow \left\{ b = -\frac{3 \cdot f_i - 4 \cdot f_{i-1} + f_{i-2}}{2h} \right\} \Rightarrow \frac{\partial f(x)}{\partial x} = b = -\frac{3 \cdot f_i - 4 \cdot f_{i-1} + f_{i-2}}{2h}$$

Assume $f(x)$ is linear continuous function with expression $f(x) = a + b \cdot x + c \cdot x^2 + d \cdot x^3 + e \cdot x^4$. The first order derivative of $f(x)$ is given by the following calculation:

$$\left\{ \begin{array}{l} f_{i-2} = f(x-2h) \\ f_{i-1} = f(x-h) \\ f_i = f(x) \\ f_{i+1} = f(x+h) \\ f_{i+2} = f(x+2h) \end{array} \right\} \Rightarrow \left\{ b = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12h} \right\} \Rightarrow \frac{\partial f(x)}{\partial x} = b = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12h}$$

It is the five-point derivative approximation since the calculation is based on the values of four adjacent points.

2.4A short overview of application of finite difference method on shell theory

Analytical methods to solve plate and shell problems are limited to relatively simple geometries, load cases, and boundary conditions. Exact solutions to plate and shell problems are difficult, or even impossible to find. Even if it can be solved, those analytical solutions are usually expressed in terms of infinite trigonometric series. Therefore, the finite element method is widely used. An alternative computational method is the finite difference method. (Rudolph Szilard 1974).

Some general concepts of the finite different method are listed in the following:

1. *Finite difference mesh*: the reference network that covers the middle surface of the shell in the shape of a rectangular or triangular grid.
2. *Finite difference operator*: the finite difference equations that replaces governing differential equations of shell theory at mesh point.

The finite difference operator is also used to formulate the boundary conditions. Applying the finite difference method to shell models creates a set of linear algebraic equations for every node within the model. By solving the linear system of equations, the numerical results of nodal displacement, shear force, bending moment, and many other important results can be obtained. In the early stage of application, it was mostly used to obtain the deformation results. It was achieved by solving biharmonic equations for each node where the fourth-order differential equations are approximated by finite difference method. However, solving a system of equations involved with four-order approximation usually results in excessive time on formulating equation molecules for various boundary conditions and geometry shapes. One way to save time is to replace biharmonic equations with lower-order equations. Marcus (1932) has split the biharmonic equation into three equations: two second-order deflection equations and one normal moment equation. By solving them, good displacement results were obtained for plates with simple supported edges.

Another alternative approximation was proposed by Reddy and Gera (1979) in which the fourth-order differential equation is replaced with three second-order differential equations, as shown in Equation 1. For various boundary conditions, those equations can perform bending moment analysis for rectangular plates with conventional finite-difference molecules.

$$\begin{aligned}
 M_x &= -D \left(\frac{\partial^2 w}{\partial x^2} + \nu \frac{\partial^2 w}{\partial y^2} \right) & M_x &= -D \left(\frac{\partial \beta_x}{\partial x} + \nu \frac{\partial \beta_y}{\partial y} \right) & Q_x &= -\frac{\pi^2}{12} Gh \left(\frac{\partial w}{\partial x} - \beta_x \right) \\
 M_y &= -D \left(\frac{\partial^2 w}{\partial x^2} + \nu \frac{\partial^2 w}{\partial y^2} \right) & M_y &= -D \left(\frac{\partial \beta_y}{\partial y} + \nu \frac{\partial \beta_x}{\partial x} \right) & Q_y &= -\frac{\pi^2}{12} Gh \left(\frac{\partial w}{\partial y} - \beta_y \right) \\
 -P &= \frac{\partial^2 M_x}{\partial x^2} - 2D(1-\nu) \frac{\partial^2 w}{\partial x^2 \partial y^2} + \frac{\partial^2 M_y}{\partial x^2} & M_{xy} &= -(1-\nu) \frac{D}{2} \left(\frac{\partial \beta_y}{\partial x} + \frac{\partial \beta_x}{\partial y} \right) \\
 & & \beta_x &= \frac{\partial w}{\partial x} - \gamma_{zx} & \beta_y &= \frac{\partial w}{\partial y} - \gamma_{zy}
 \end{aligned}$$

Equation 1: Three second-order differential equations used in finite difference approximation

Equation 2: Six first-order differential equations used in finite difference approximation ($h =$ plate thickness)

Assadi-Lamouki and Krauthammer (1989) develop a method to solve plate vibration problem by solving the finite difference approximation of six first-order equations (see in Equation 2). Their research has shown that the obtained vibration study results were in good quality compared to results obtained by classical plate theory. As mentioned above, the finite difference method replaces the governing equations with a set of algebraic equations. Then computers are used to find the solution to the algebraic equations. Advantages are:

- a) It is a straightforward method to be understood and applied;
- b) It is a universal method that can be applied to various problems; and
- c) This method can be relatively easily implemented.

Disadvantages are:

- d) It requires a certain time and mathematical knowledge to find the proper finite difference operators;
- e) To perform the analysis efficiently, this method is usually achieved through computer programming. It requires more work to produce a program allowing complete automation of the procedure;
- f) The parameter matrix of approximated algebraic equations is asymmetric, sparse, and highly possible to be overdetermined, causing difficulties in finding its numerical solution unless using a least square solver;
- g) It may have serious difficulties in applying finite difference method into complicated geometry rather than the regular square and rectangular shape.

2.5 Sparse linear system: overdetermined and determined, its storage and solving methods

The key to applying the finite difference method to shell problems is to solve the matrix of the approximating system of linear algebraic equations. From previous experience and practice, the constructed matrix of shell or plate models is usually overdetermined. 21 S-K equations are assigned to mn nodes of the model. To define the boundary conditions, an additional number of 4 equations are assigned to the $2(m+n)$ edge nodes. The resulting matrix is a rectangular matrix having $21mn$ columns and $21mn+8(m+n)$ rows. Alternatively, a square matrix is constructed by replacing 4 equations of the 21 equations at every edge and corner node. Consequently, two programs were developed; one for overdetermined and one for determined systems (rectangular and square matrices).

There are 21 dependent variables in the 21 S-K equations. One equation only contains 4 or 5 dependent variables on average. Depending on what interpolation is used (two-point or five-point), the interpolation at one point only involves 2 or 4 adjacent points. The factors of discretized dependent variable are distributed in

lines parallel to the diagonal of the matrix. Almost all entries in the matrix are zero, which is convenient for computation. Matrices with this property are said to be sparse. Therefore, the constructed determined and overdetermined systems are sparse.

It is important to have efficient storage of, and operations on large sparse matrices.

1. Storage

A sparse matrix is a matrix that has a value of zero for most elements. In a sparse matrix, the ratio of number of non-zero elements to the size is considered as less than 0.5. If a sparse matrix is stored as a normal dense matrix, most information stored will be zero elements which is highly inefficient. In order to perform faster operations and use less memory, there are several different storage methods developed that only store those non-zero elements, and the zero elements are left unspecified. Since the revolutionary development of computers, many new alternatives have been developed. Many new algorithms and a number of new software packages are designed for efficiently finding the solution of different sparse symmetric systems. However, one important factor limiting the efficiency of solvers is the memory usage of the matrix. The author has constructed a dense matrix for a shell model of 50*50 nodes which consumed 4 GB of RAM from a laptop having 8 BG of RAM. Since the operation needed for solving a dense system increasing with the cube of the matrix size, the computational time for solving this system is more than an hour when using a solver involving factorization of the system matrix. The Python SciPy sparse package provides below implementations:

a) Coordinate Matrix & Dictionary of Keys Matrix

The simplest sparse matrix format is the Coordinate (COO) format. In this format, three subarrays are used to store the element values and their coordinate positions. In doing so, the saved memory consumption can be substantial for a large matrix. Managing the subarrays create overhead which can become negligible as the dataset grows. It has to be noticed that the dataset should be sufficiently sparse enough otherwise the several subarrays created by COO format might consume more memory.

Dictionary of Keys (DOK) format performs similar operations like COO except it stores element values and their coordinates as key-value pairs in dictionaries. The built-in functionality come with dictionaries provide convenience in constructing and updating matrices. In doing so, the key is hashed as the hash indicates for looking up corresponding values. It means a constant lookup time for identifying values at any given location.

b) Compressed Sparse Matrices (CSR, CSC)

Three subarrays are stored in the compressed sparse matrices format: index pointers, indices, and data. The (start, stop) slice of indices are recorded as adjacent pair of number in index pointers. The location of those adjacent pairs in index pointers indicates the column number (if Compressed Sparse Row format (CSR) is used) or row number (if Compressed Sparse Column format (CSC) is used). Then the left row number (if CSR is used) or column number (if CSC is used) is determined by the numbers in the slice of indices. By doing so, the location of each value in data array in the original matrix can be determined.

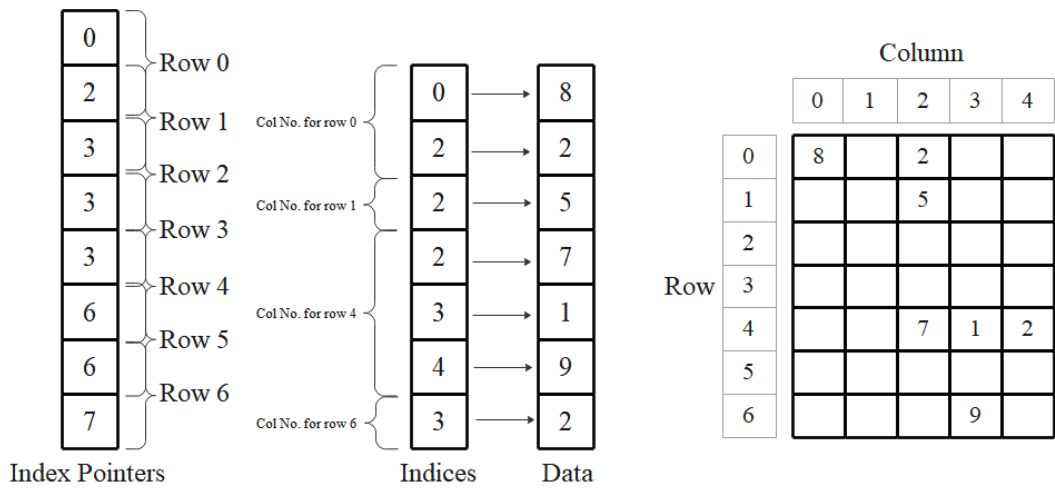


Figure 3: Compressed Sparse Row format (index pointers, indices, and data)

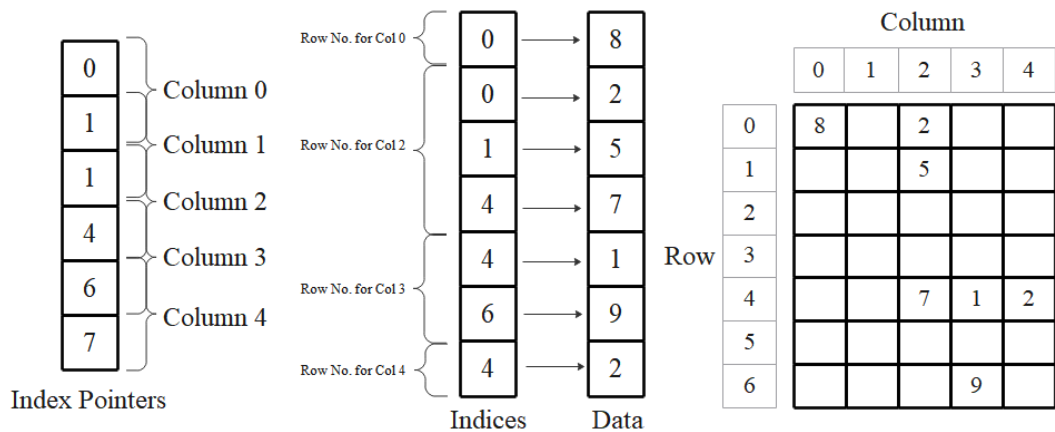


Figure 4: Compressed Sparse Column format (index pointers, indices, and data)

2. Solve

Solving large sparse linear systems has been a concerning issue for years. It lies at the heart of many problems in computational science and engineering. Particularly when encountering discretizing continuous problems, it is common that the constructed system is sparse and large. The direct method for solving a sparse linear system $A \cdot x = b$ involves the explicit factorization of system matrix A , such as Gaussian elimination, into the product of lower and upper triangular matrices L and U . In the most case, a permutation of system matrix A is used $PAQ = LU$ where permutations P and Q are chosen to preserve sparsity and maintain stability. Cholesky factorization $U = L^T$ is used if system matrix A is symmetric. Forward elimination followed by backward substitution completes the solution process for each given right-hand side b . The direct methods are the general and robust solution for many sparse linear systems. Meanwhile, the constructed system matrix A in this shell code was a determined or overdetermined sparse matrix and stored in CSR format. Below are listed the solving methods which can deal with such systems matrices in an efficient way.

a) Linear regression:

In linear regression, the response of a system is assumed to be the linear combination of the predictors. From the lecture from NYU (Fernandez-Granda, 2016), the linear regression model can be described as below: the linear regression model is parametrized by the intercept β_0 and weight vector $\underline{\beta}$. For each value of response vector \underline{y} , the corresponding values of the predictors are $X_{i1}, X_{i2}, X_{i3}, \dots, X_{ip}$ where p is the number of predictors. And the response vector \underline{y} contains n number of responses. The linear model is given by:

$$y_i = \beta_0 + \sum_{j=1}^p X_{ij}\beta_j, 1 < i < n$$

$$\underline{y} = \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} \beta_0 + \underline{\underline{X}}\underline{\underline{\beta}} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} \approx \begin{bmatrix} 1 \\ 1 \\ \dots \\ 1 \end{bmatrix} \beta_0 + \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1p} \\ X_{21} & X_{22} & \dots & X_{2p} \\ & & \dots & \\ X_{n1} & X_{n2} & \dots & X_{np} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \dots \\ \beta_p \end{bmatrix}$$

The linear regression model is usually used in statistics aiming to predict the values of response accurately for new values of the predictors. The accuracy of prediction of response can be improved as more data points are included. The weight vector $\underline{\underline{\beta}}$ is updated during this process. The sparse linear system $A \cdot x = b$ can also be viewed as a linear regression model where the intercept $\beta_0 = 0$ as an inverse problem aiming to determine weight vector $\underline{\underline{\beta}}$.

For $p \geq n$, in the condition of full-rank $\underline{\underline{X}}$, the solution of least-square problems is weight estimate $\underline{\underline{\beta}}_{ls}$ and it is given by:

$$\underline{\underline{\beta}}_{ls} = \left(\underline{\underline{X}}^T \underline{\underline{X}} \right)^{-1} \underline{\underline{X}}^T \underline{y}$$

For $n = p$, which means the number of data points to fit the linear model that is the same as the number of parameters, the exact solution of weight vector $\underline{\underline{\beta}}$ can be found. However, this solution usually does not reflect the actual relation between \underline{y} and $\underline{\underline{X}}$ since the model is too flexible with respect to the number of available data. This phenomenon is called overfitting. As discussed above, the system matrices in the shell code can be both determined and overdetermined, which means $n \geq p$. Linear regression is a suitable method for a linear model when the number of data points n is large than the p is the number of predictors.

In this way, the problem of solving a sparse linear system $A \cdot x = b$ can be restated as the estimation of the weight vector $\underline{\underline{\beta}}$ from the linear regression model $\underline{y} = \underline{\underline{X}}\underline{\underline{\beta}}$ where the predictors $\underline{\underline{X}}$ and response \underline{y} are known.

Currently the most popular method to estimate the weight vector $\underline{\underline{\beta}}$ is to minimize the sum of the squares of the fitting error on the training set, which is called least-squares estimation.

$$\text{Minimize } \left\| \underline{y} - \underline{\underline{X}}\underline{\underline{\beta}} \right\|_2$$

b) Least-squares estimation: *lsmr* solver from Python Package ‘*scipy.sparse.linalg*’

The Python Package ‘*scipy.sparse.linalg*’ provides an iterative solver ‘*lsmr*’ for least-squares estimation. *lsmr* can solve the system of linear equations $A \cdot x = b$. A is a sparse or dense matrix of dimension $m \times n$ where all cases are allowed: $m = n$, $m > n$, or $m < n$. b is a vector of length m . This solver is developed based on the iterative method *LSMR* proposed by Fond and Saunders(2011).. *LMSR* is a numerical method for computing a solution x of linear least square problem $\|Ax - b\|_2$. Compared to the well-known method *LSQR* (Paige and Saunders, 1982), *LMSR* is also based on Golub-Kahan bidiagonalization of A . It has the property of reducing $\|r_k\|$ and $\|Ar_k\|$ monotonically, where $r_k = b - Ax_k$ is the residual for the approximate solution

x_k . Hence, although LSQR and *LSMR* ultimately converge to similar points, it is safer to use *LSMR* in situations where the solver must be terminated early.

The *LSMR* algorithm contains following major steps:

1. **The Golub-Kahan process:** It is an iterative procedure for transforming A and b into upper-bidiagonal form $\beta_1 e_1$ and B_k . It is equivalent to what would be generated by the symmetric Lanczos process with $A^T A$ and $A^T b$
2. **Using Golub-Kahan to solve the normal equation:** In order to find the solution x_k of the equation $A^T A \cdot x = A^T b$, the subproblem is to choose y_k to minimize $\|Ar_k\|$ at each stage. It is given by:

$$\min_{y_k} \|Ar_k\| = \min_{y_k} \left\| \bar{\beta}_1 e_1 - \begin{pmatrix} B_k^T B_k \\ \bar{\beta}_{k+1} e_k^T \end{pmatrix} y_k \right\|$$

3. **Two QR factorizations:** Convert the subproblem into: $\min_{y_k} \|Ar_k\| = \min_{t_k} \left\| \begin{pmatrix} z_k \\ \zeta_{k+1} \end{pmatrix} - \begin{pmatrix} \bar{R}_k \\ 0 \end{pmatrix} t_k \right\|$
4. **The heart of *LSMR* algorithm:** Through matrix rotations and substitutions, the recurrence for the approximated solution x_k is given by:

$$x_k = x_{k-1} + \left(\frac{\zeta_k}{\rho_k \bar{\rho}_k} \right) \bar{h}_k, \quad \bar{h}_k = h_k - \left(\frac{\bar{\theta}_k \rho_k}{\rho_{k-1} \bar{\rho}_{k-1}} \right) \bar{h}_{k-1}, \quad h_{k+1} = v_{k+1} - \left(\frac{\theta_{k+1}}{\rho_k} \right) h_k$$

x_k, \bar{h}_k, h_k are updated for each iteration until the stopping rules are satisfied

5. **Stopping rules:** $\|r_k\|, \|A^T r_k\|, \|x_k\|$, and estimates of $\|A\|$ and $\text{cond}(A)$ are used. All quantities are computed at each iteration for checking the stopping rules. The practical stopping criteria includes three rules:

S1: Stop if $\|r_k\| \leq BTOL \|b\| + ATOL \|A\| \|x_k\|$

S2: Stop if $\|A^T r_k\| \leq ATOL \|A\| \|x_k\|$

S3: Stop if $\text{cond}(A) \geq CONLIM$

ATOL and *BTOL* are cv that can be set by the user. S1 is applied when the $A \cdot x = b$ system is consistent. S2 is applied when the $A \cdot x = b$ system is inconsistent. If the both stop tolerances are $1e-6$, then the iteration will stop when the final residual $\|r_k\|$ is accurate to about 6 digits. Those stop tolerances are the estimates of the relative error in entries of A and b , allowing for uncertainty in the system. This prevents the algorithm from doing unnecessary calculation beyond the uncertainty of the input data.

CONLIM is the user-set limit for the condition number of A . The stop rules S3 means the algorithm terminates if the an estimate of $\text{cond}(A)$ exceeds *CONLIM*. S3 can be applied to any system, consistent or inconsistent. The $\text{cond}(A)$, the condition number for inversion of A , is used to measure how sensitive the inversion of matrix is to changes or errors in the input.

c) Sparse QR factorization:

As discussed above, the linear regression method can deal with a linear model when the number of data points n is larger than the number of predictors p . However, in many applications, the number of data might not be sufficient enough. Although the system matrices in the shell code gives $n \geq p$, difference ratio between n and p is $\frac{n-p}{n} = \frac{N}{M+N}$ where M is the total number of S-K equations and N is total number of boundary equations. Meanwhile, $M = 21 \times \text{Number of body nodes}$ and $N = 4 \times \text{Number of edge nodes}$. It means the difference ratio actually becomes smaller for a model with larger number of nodes. In most cases, the number of data points n is roughly equal to the number of predictors p . The linear regression may not able to solve the system accurately. The overdetermined problem is obviously expected when fitting such model through linear regression. Due to the sparsity of the system, not all predictors are involved for each data point. The S-K equation contains 4 or 5 quantities on average, which means only around 20% of predictors are involved for each data point. So, in terms of the linear regression problem, the matrix of predictors still is very sparse.

In the least-squares estimation of linear regression problem, QR decomposition is usually used to convert the given matrix A into orthonormal matrix Q and upper triangular matrix R and $A = QR$. For a rectangular matrix A with size of $M \times N$, Q has size of $M \times M$ and R has size of $M \times N$. By introduce the $A = QR$ into system equation $A \cdot x = b$, it gives below equation.

$$A \cdot x = b \rightarrow A^T A \cdot x = A^T b \xrightarrow{QR} R^{-T} A^T b = Q^T b$$

It is notable that the matrix R in the QR decomposition is a Cholesky factor of $A^T A$. The $A^T A$ has a condition number which is the square of A . On the other hand, the orthonormal matrix Q will be very dense in general since A is large and sparse. It means $A^T A$ and Q usually cannot be computed explicitly which can adversely affect numerical precision and robustness. So, it may not be advantageous to directly use the QR decomposition for large linear sparse systems. Consequently, new steps are needed to provide more accurate calculation, which is called the sparse QR factorization.

A sparse QR factorization usually contains following steps:

1. Permute the columns or rows of A so that the Cholesky factor of $A^T A$ (or the matrix R , which has the same structure) remains sparse.
2. Compute a QR decomposition based on the permuted A to obtain matrix R
3. Solve $Rx = Q^T b$ where $Q^T b = R^{-T} A^T b$

One solver with such features is the ***lm.fit.sparse* solver** from R Package 'MatrixModels'. It is a basic computing engine for sparse linear least squares regression. This solver receives a sparse overdetermined matrix as input and returns a vector of approximated solution where the sparse QR factorization method can be used. However, due to the lack of information in the user manual and source code, the mathematical details of this solver remain unclear. Previous research done by Svoboda, Cashman, and Fitzgibbon (2018) introduces an open-source suite of solvers QRkit that can perform sparse QR factorization for common sparsity patterns. Since QRkit solvers share similar features with *lm.fit.sparse* solver, it can help to understand the how the actual *lm.fit.sparse* solver might work.

The general strategy of QRkit solvers is to express matrix A as some combination of smaller matrices $A_{1..K}$. $A_{1..K}$ are divided based on shape and sparsity patterns to store and compute the QR factorization more efficiently. Then those smaller matrices are processed through different methods leading to easier QR factorizations. The final result of factorization of A is the combination of factorization of smaller matrices. One of those process methods is row and column permutations which is are used in the most of sparse QR factorization solver. If the sparsity pattern of A is not obvious enough to be categorized, applying row and

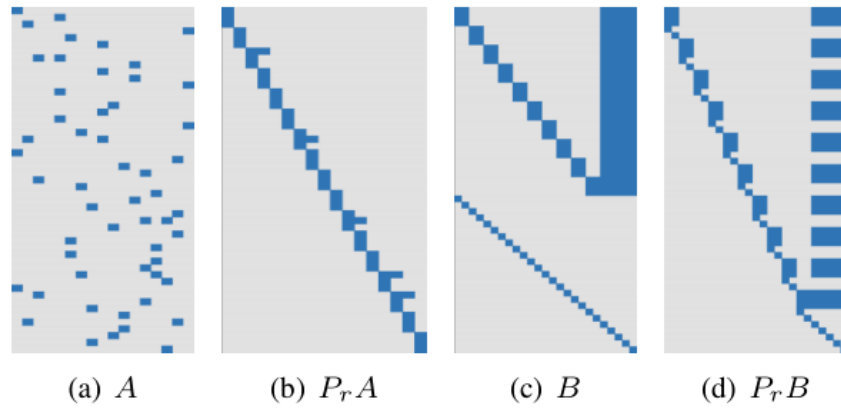
column permutations is a suitable method which can convert an overdetermined and sparse A into a block diagonal/banded matrix. The row and column permutation P_r, P_c would reorder the rows or columns of A in order to create an A' with 'As-Banded-As-Possible' sparsity pattern.

$$A' = P_r A, A' = A P_c$$

In practice, both permutations are used in order to create A' with row-banded structure and reduce the fill-in of the QR decomposition at the same time.

$$A' = P_r A P_c$$

Figure 5: Row permutation p_r discovering banded structure in the matrix A . (c-d) Row permutation p_r while solving vertical concatenation of two matrices



d) Moore-Penrose inverse

The Moore–Penrose inverse, variously known as the generalized inverse, pseudoinverse, or Moore–Penrose inverse, is common $m \times n$ method to find the solution of linear equations that cannot be computed through least square method. For all matrices A whose entries are real or complex numbers, their Moore–Penrose inverse A^+ is defined and unique. This matrix was independently defined by Moore in 1920 and Penrose (1955)

A Moore–Penrose inverse satisfy all of the following four criteria:

$$AA^+A = A, A^+AA^+ = A^+, (AA^+)^* = AA^+, (A^+A)^* = A^+A$$

where A^* denotes the conjugate transpose for matrix A .

If the inverse of A^*A exists, then: $A^+ = (A^*A)^{-1}A^*$ and $A^+A = I_n$

If the inverse of AA^* exists, then: $A^+ = A^*(AA^*)^{-1}$ and $AA^+ = I_m$

However, it is common that above conditions are not satisfied, meaning those inversions have zero or many solutions. In this case, the approximated pseudoinverse can still be found with the help of single value decomposition (SVD). Since $A^+A = I_n$ is impossible, the problem now is to find the $A^+A \approx I_n$ by minimizing

$\|A^+A - I_n\|_2$. The SVD provides the following solution:

$$A^+ = VD^+U^T$$

where U , D , V respectively the left singular vectors, the singular values and the right singular vectors of A . In SVD, the matrix A is a factorization of form UDV^* which is not unique. Since the singular values D is rectangular diagonal matrix with on-negative real numbers on the diagonal, the pseudoinverse D^+ can be calculated by taking the reciprocal of non-zero values of D .

One significant impact of applying SVD is the high computational cost during the decomposition of the matrix. According to Trefethen (1997), the first step of SVD is reducing the matrix into a bidiagonal matrix. The second step is to compute the SVD of the bidiagonal matrix through an iterative method with set-up certain precision. The overall computational cost is about $O(mn^2)$ floating-point operations (flops). With particular methods (Householder reflections, QR algorithm), the overall cost ranges from $2mn^2$ to $4mn^2$ flops. It is several times higher than the normal matrix multiplication.

The **pinv** solver from the Python numpy package is able to perform the above calculation.

3. Shell code and tests

This section provides the general mathematic details and programming structure of the shell code. The basic workflow of the code is introduced. Some important steps like the formation of matrices, construction of square matrices, and loading and plotting are explained specifically. In order to prove the general universality of the shell code, two shell models with various material properties, geometry shapes, boundary conditions were set up. A number of tests were organized to investigate influential factors including number of nodes, types of solvers, different methods of keeping the matrix square, and different difference approximation methods.

3.1 Work flow of code

This code is designed to deal with simple shell model problems by solving Sanders-Koiter equations at a nodal level where the differentiation in equations is replaced by the finite difference method. With proper input parameters, several sparse matrices are constructed to form an overdetermined sparse linear system $[M]*[u] = [f]$ where $[M]$ represents the stiffness matrix of the modelled shell structure, $[u]$ represents the vector of deformations, forces and moments at nodal level, and $[f]$ represents the vector of applied forces or prescribed deformation at nodal level. Vector $[u]$ is solved and reconstructed to generate 21 plots of nodal results of u_x , u_y , u_z , ϵ_{xx} , ϵ_{yy} , γ_{xy} , φ_x , φ_y , φ_z , κ_{xx} , κ_{yy} , ρ_{xy} , n_{xx} , n_{yy} , n_{xy} , n_{yx} , v_x , v_y , m_{xx} , m_{yy} , and m_{xy} .

The work flow of code can be summarized and illustrated as below:

- Step 1:** Inputs of parameters to determine material property, geometry of models and boundary condition type
- Step 2:** Define differential equations in x and y direction by finite difference methods
- Step 3:** Create empty matrices for $[M]$, $[u]$ and $[f]$
- Step 4:** Add Sanders-Koiter equations to overdetermined sparse matrix $[M]$ while add load components to $[f]$
- Step 5:** Defined the boundary conditions by adding components to $[M]$ and $[f]$
- Step 6:** Solve $[u]$ from $[M]*[u] = [f]$ by several different solvers
- Step 7:** Postprocessing and display results of solved $[u]$

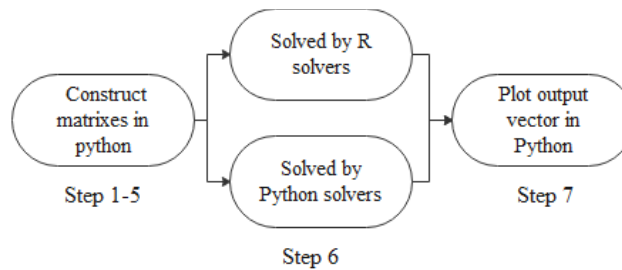


Figure 6: Work flow of code

In step 6, several different solvers from the R language and Python language are tried and compared in order to obtained satisfactory accuracy. The comparison between different solver results is shown in below section of solver performance analysis with solver computation time and memory usage.

By comparing shell code results with the analytical solution and finite element solution, one set of results with highest accuracy is selected as main results to be shown in below section of code results.

Meanwhile, for an overdetermined system like $[M]*[u] = [f]$, $[u]$ cannot be solved exactly and an approximated $[M]$ is used in the calculation. To find out which Sanders-Koiter equations are approximated and the extent of that approximation, some equations are left out in order to construct $[M]$ as a square matrix.

The specific details of replacing equations with boundary conditions are shown in section of formation of square matrix (Section 3.3.6, page 27).

3.2 Tested models

To fully demonstrate the capability of the shell code, two shapes were tested (Table 6) with various boundary conditions (Table 7) and various numbers of nodes. Table 8 specifies the boundary condition equations. The configuration of tested five tested models are summarized in Table 9. There are two load cases involved: uniformly vertical load and uniformly normal load. They both have the magnitude of $p = -10 \text{ kN/m}^2$. The direction of uniform vertical load is aligned with the global vertical axis. The direction of uniformly normal load is perpendicular to the middle surface of shell.

Table 6: Geometry and material parameters of models

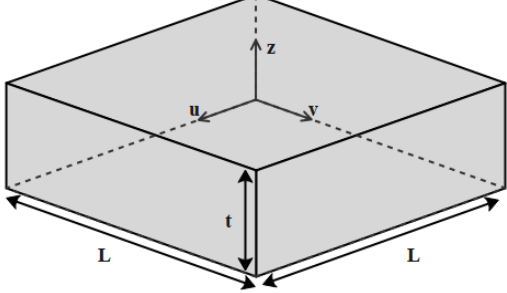
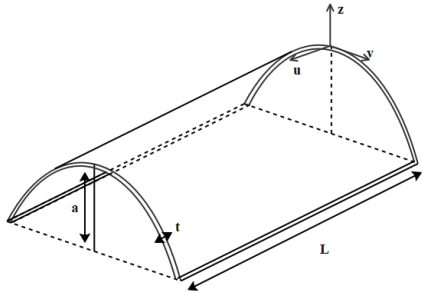
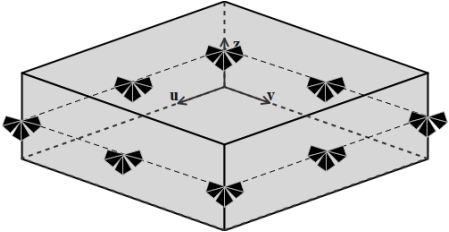
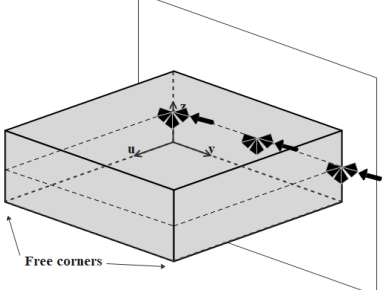
Flat square shape	Canopy shape
	
<p>Length $l = 1m$, thickness $t = 0.06m$ Young's modulus $E = 21 \cdot 10^7 \text{ kN/m}^2$ Poisson's ratio $\nu = 0.3$ Number of nodes in x direction: m Number of nodes in y direction: n Shell curvature: $k_{xx} = k_{yy} = k_{xy} = 0$ Lamé parameters: $\alpha_x = \frac{l}{m-1}, \alpha_y = \frac{l}{n-1}$ In plane curvature: $k_x = k_y = 0$</p>	<p>Length $l = 12m$, radius $a = 2m$, thickness $t = 0.06m$ Young's modulus $E = 21 \cdot 10^7 \text{ kN/m}^2$ Poisson's ratio $\nu = 0.15$ Number of nodes in x direction: m Number of nodes in y direction: n Shell curvature: $k_{xx} = k_{xy} = 0, k_{yy} = -\frac{1}{a}$ Lamé parameters: $\alpha_x = \frac{l}{m-1}, \alpha_y = \frac{\pi \cdot a}{n-1}$ In plane curvature: $k_x = k_y = 0$</p>

Table 7: Pinned edges and Cantilever boundary conditions

	Pinned edges: Fixed translation at all edges	Cantilever: Fixed translation and rotation at one edge
Flat square shape		

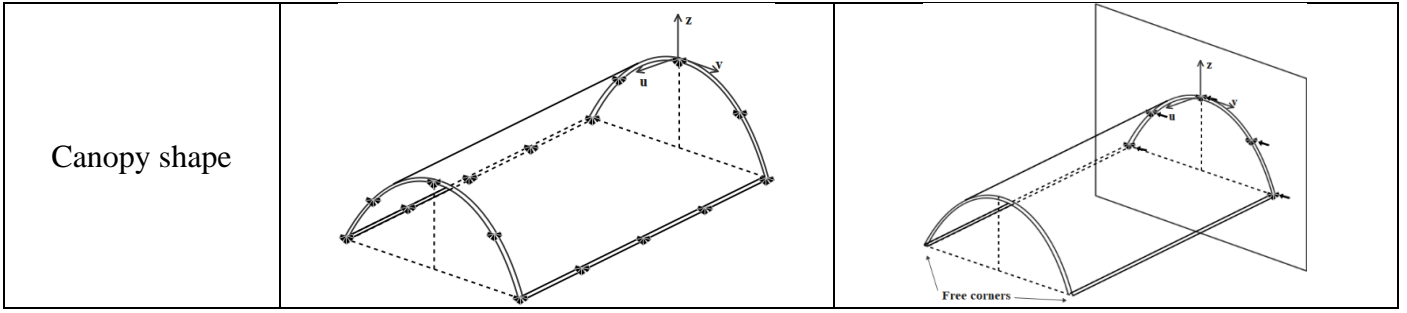


Table 8: Boundary equations

	Pinned edges: Fixed translation at all edges	Cantilever: Fixed translation and rotation at one edge
Edge in the x direction and the y axis pointing outwards	$u_x = u_y = u_z = 0$ $-m_{yy} = 0$	$n_{yx} - k_{xx}m_{xy} = 0, n_{yy} - k_{xy}m_{xy} = 0$ $v_y + \frac{\partial m_{xy}}{\partial x} = 0, -m_{yy} = 0$
Edge in the x direction and the y axis pointing inwards	$u_x = u_y = u_z = 0$ $m_{yy} = 0$	$-n_{yx} + k_{xx}m_{xy} = 0, -n_{yy} + k_{xy}m_{xy} = 0$ $-v_y - \frac{\partial m_{xy}}{\partial x} = 0, m_{yy} = 0$
Edge in the y direction and the x axis pointing outwards	$u_x = u_y = u_z = 0$ $-m_{xx} = 0$	$u_x = u_y = u_z = 0$ $\varphi_x = 0$
Edge in the y direction and the x axis pointing inwards	$u_x = u_y = u_z = 0$ $m_{xx} = 0$	$n_{xx} - k_{xy}m_{xy} = 0, n_{xy} - k_{yy}m_{xy} = 0$ $v_x + \frac{\partial m_{xy}}{\partial y} = 0, m_{xx} = 0$
Free corners	/	$\frac{n_{xy} + n_{yx}}{2} = 0, m_{xy} = 0$

Table 9: Model configuration

No.	Shape	Boundary conditions	Load case	Number of nodes
Model 1	Flat square shape	Pinned edges	Uniformly vertical load	$m = n = 20, 30, 50$
Model 2		Cantilever		
Model 3	Canopy shape	Pinned edges	Uniformly vertical load	$m = n = 30, 50$
Model 4		Cantilever	Uniformly vertical load	
Model 5			Uniformly normal load	

3.3 Differentiation approximated by FDM

$$k_{xx}n_{xx} + k_{xy}(n_{xy} + n_{yx}) + k_{yy}n_{yy} + \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + k_y q_x + k_x q_y + p_z = 0$$

① ②

Sanders-Koiter equation 1

.....
 1: Dx (vx, 1.0)
 2: Dy (vy, 1.0)

Differentiation

In order to add Sanders-Koiter equations to matrix [M] at nodal level, differentiation of that equation is required to be translated into a discrete form where the finite difference method takes place. The first order

derivative of $f(x)$ is given as $\frac{\partial f(x)}{\partial x} = \frac{f_{i-1} - f_{i+1}}{2\alpha_x}$ and the one-sided finite differences of first order derivative is $\frac{\partial f(x)}{\partial x} = -\frac{3 \cdot f_i - 4 \cdot f_{i+1} + f_{i+2}}{2\alpha_x}$. The differentiation of one quantity (α_x, α_y) in the shell code is approximated by predefined finite difference functions ($D_x(k, g), D_y(k, g)$) where k represents the quantity to be differentiated while g works as positive/negative sign of the value ($g = -1/1$).

Central difference:	$D_x(k, g) = \frac{g}{2\alpha_x} (f_{i-1} - f_{i+1})$	$D_y(k, g) = \frac{g}{2\alpha_y} (f_{j+1} - f_{j-1})$
Forward difference:	$D_x(k, g) = \frac{g}{2\alpha_x} (3 \cdot f_i - 4 \cdot f_{i+1} + f_{i+2})$	$D_y(k, g) = \frac{g}{2\alpha_y} (3 \cdot f_j - 4 \cdot f_{j+1} + f_{j+2})$
Backward difference	$D_x(k, g) = -\frac{g}{2\alpha_x} (3 \cdot f_i - 4 \cdot f_{i-1} + f_{i-2})$	$D_y(k, g) = -\frac{g}{2\alpha_y} (3 \cdot f_j - 4 \cdot f_{j-1} + f_{j-2})$

<pre>def Dlx(k,g): if i==0: M[row,k*m*n+j*m+i+2]=-1*g/(2*alphax(i/(m-1),j/(n-1))) M[row,k*m*n+j*m+i+1]= 4*g/(2*alphax(i/(m-1),j/(n-1))) M[row,k*m*n+j*m+i]=-3*g/(2*alphax(i/(m-1),j/(n-1))) elif i==m-1: M[row,k*m*n+j*m+i]= 3*g/(2*alphax(i/(m-1),j/(n-1))) M[row,k*m*n+j*m+i-1]=-4*g/(2*alphax(i/(m-1),j/(n-1))) M[row,k*m*n+j*m+i-2]= 1*g/(2*alphax(i/(m-1),j/(n-1))) else: M[row,k*m*n+j*m+i+1]= 1*g/(2*alphax(i/(m-1),j/(n-1))) M[row,k*m*n+j*m+i-1]=-1*g/(2*alphax(i/(m-1),j/(n-1))) return</pre>	<pre>Location: (at left edge) k3 = k*m*n+j*m+i+2 k2 = k*m*n+j*m+i+1 k1 = k*m*n+j*m+i (at right edge) k3 = k*m*n+j*m+i k2 = k*m*n+j*m+i-1 k1 = k*m*n+j*m+i-2 (inside grids) k2 = k*m*n+j*m+i+1 k1 = k*m*n+j*m+i-1</pre>
---	--

Code 1: Finite difference approximation in x direction

3.4 Formation of matrix [M]

The size of matrix [M] depends on the number of nodes and applied boundary conditions. The number of columns is $21 \cdot mn$ and number of rows is $21 \cdot mn + 8 \cdot m + 8 \cdot n$ for pinned edge boundary condition where $8 \cdot m + 8 \cdot n$ is from four boundary equations for each edge node. If free corners exist, the additional boundary equations are added at free corner nodes. Each free corner node requires two additional boundary equations. For cantilever boundary condition where two free corners exist, the number of rows is $21 \cdot mn + 8 \cdot m + 8 \cdot n + 4$.

As defined in Sanders-Koiter equations, 21 unknown quantities are assigned to each node. which are $u_x, u_y, u_z, \epsilon_{xx}, \epsilon_{yy}, \gamma_{xy}, \Phi_x, \Phi_y, \Phi_z, \kappa_{xx}, \kappa_{yy}, \rho_{xy}, n_{xx}, n_{yy}, n_{yx}, v_x, v_y, m_{xx}, m_{yy}, m_{xy}$. During the code tests, they are assigned with an integral value from 0 to 21 indicating their proposed location in a solved [u] in order to extract results accordingly.

For example, Sanders-Koiter equation 1 is added to the matrix [M] by the adding process code. For adding one value to matrix, the process can be described as $M[\text{Row number}, K \cdot m \cdot n + j \cdot m + i] = \text{Parameter of unkown}$. In this expression, K is the assigned integral value from 0 to 21 representing the location in the solved [u] vector. j is the coordinate of the node in v direction and i is the coordinate of the node in u direction.

$$k_{xx}n_{xx} + k_{xy}(n_{xy} + n_{yx}) + k_{yy}n_{yy} + \frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + k_y q_x + k_x q_y + p_z = 0$$

Sanders-Koiter equation 1

- 1: $M[\text{row}, \text{nxx} \cdot m \cdot n + j \cdot m + i] = k_{xx}(i / (m-1), j / (n-1))$
- 2: $M[\text{row}, \text{nxy} \cdot m \cdot n + j \cdot m + i] = k_{xy}(i / (m-1), j / (n-1))$
- 3: $M[\text{row}, \text{nyx} \cdot m \cdot n + j \cdot m + i] = k_{xy}(i / (m-1), j / (n-1))$

Adding process

The adding process contains two loops (Loop ① & Loop ②). In Loop ①, adding process is repeated for every node at one line along u-direction. Loop ② repeats Loop ① until all lines are fulfilled, so that unknown quantities are added to every node. In each adding process, the parameters of unknowns are added into matrices as their location in [M] is altered with changing i and j in the loops.

```

row=-1
for j in range(n): # Add Sanders-Koiter equation 1 to the matrix
    for i in range(m):
        row=row+1
        M[row, nxx*m*n+j*m+i]=kxx(i/(m-1), j/(n-1))
        M[row, nxy*m*n+j*m+i]=kxy(i/(m-1), j/(n-1))
        M[row, nyx*m*n+j*m+i]=kxy(i/(m-1), j/(n-1))
        M[row, nyy*m*n+j*m+i]=kyy(i/(m-1), j/(n-1))
        Dx(vx, 1.0)
        Dy(vy, 1.0)
        M[row, vx*m*n+j*m+i]=ky(i/(m-1), j/(n-1))
        M[row, vy*m*n+j*m+i]=kx(i/(m-1), j/(n-1))
        f[row]=-pz(i/(m-1), j/(n-1))

```

Code 2: Add Sanders-Koiter equation 1 to the matrix [M]

The column number of starting points for unknown quantity K is $K \cdot m \cdot n$. For each adding process in Loop ①, the column number and row number increased accordingly for m times. Then Loop ② repeats Loop ① for n times. Meanwhile for every Loop ①, the row number is incremented by one. By adding one equation, $m \cdot n$ rows of matrix have been generated.

The below figure shows the pattern of non-zero values in matrix [M] during this adding procedure where the values are diagonally distributed. The same adding procedure is also utilized for adding boundary conditions at edges and corners. While adding values to [M], the force vector [f] is also filled with applied force components.

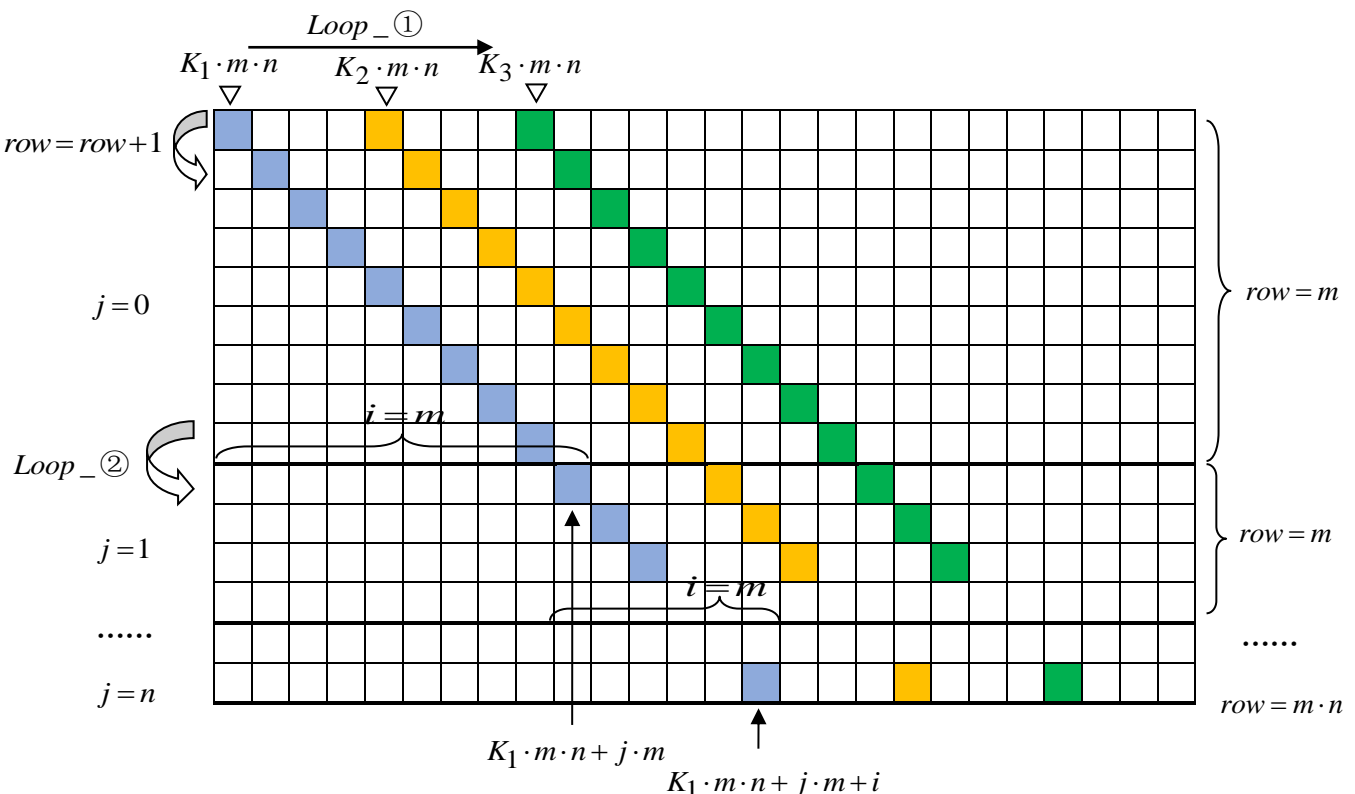


Figure 7: Add equations to [M]

After all equations and boundary conditions have been added to [M], the distribution of non-zero values in matrix [M] is simplified as below. The completely assembled matrix [M] is a rectangular sparse matrix where values are in parallel to the diagonal line of the matrix.

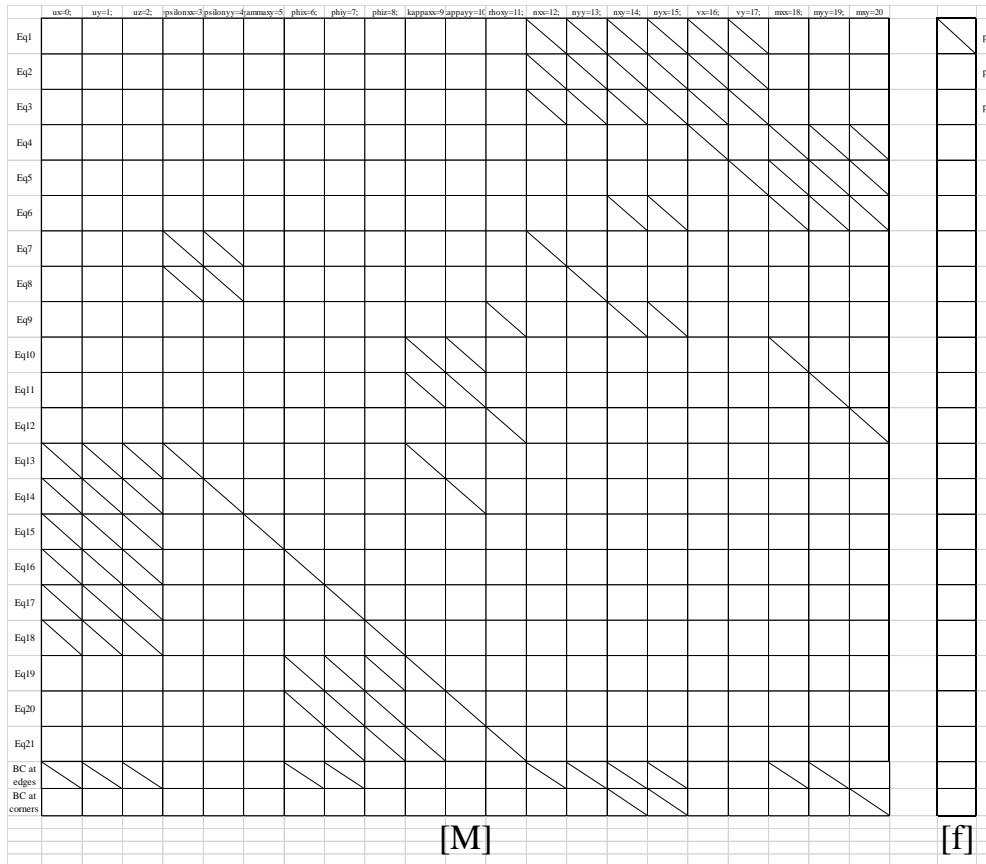


Figure 8: Non-zero value distribution in [M] and [f]

3.5 Loading steps and plotting steps

Sanders-Koiter equations give the vertical displacement u_z which is in the normal direction of the mid surface of the shell. For the non-flat geometry like the canopy shape used in model 3-5, the direction of u_z for those models does not comply with the global z-axis. In order to apply vertical load and plot the vertical displacement in global z-axis, different definition is required at loading step and plotting step of shell code. The total displacement u_{xyz} is calculated as $u_{xyz} = \sqrt{u_x^2 + u_y^2 + u_z^2}$ by those calculated u_x , u_y , and u_z results.

<p>Uniformly vertical load for model 1& 2:</p> <pre>def px(u, v): return 0 def py(u, v): return 0 def pz(u, v): return p #p = 10 kN</pre>	<p>Uniformly vertical load for model 3-5:</p> <pre>def px(u, v): return 0 def py(u, v): return p/(math.sin(v*math.pi)*math.tan(v*math.pi)+math.cos(v* math.pi)) def pz(u, v): return p*math.tan(v*math.pi)/(math.sin(v*math.pi)*math.tan(v* math.pi)+math.cos(v*math.pi)) #p = 10 kN</pre>
	<p>Normal load for model 3-5:</p> <pre>def px(u, v): return 0 def py(u, v):</pre>

```

return 0
def pz(u, v):
return p #p = 10 kN

```

Code 3: loading step for different models

```

# x[j][i], y[j][i], z[j][i] are the displacement results solved by S-K equations
if modelv == 1: # when geometry shape is canopy shape
for j in range(n):
for i in range(m):
UX[j][i] = x[j][i]
UY[j][i] = y[j][i]*math.sin(j/(n-1)*math.pi)+z[j][i]*math.cos(j/(n-1)*math.pi)
UZ[j][i] = y[j][i]*math.cos(j/(n-1)*math.pi)+z[j][i]*math.sin(j/(n-1)*math.pi)
UXYZ[j][i] = math.sqrt(x[j][i]**2+y[j][i]**2+z[j][i]**2)
elif modelv == 2: # when geometry shape is flat square
for j in range(n):
for i in range(m):
UX[j][i] = x[j][i]
UY[j][i] = y[j][i]
UZ[j][i] = z[j][i]
UXYZ[j][i] = math.sqrt(x[j][i]**2+y[j][i]**2+z[j][i]**2)

```

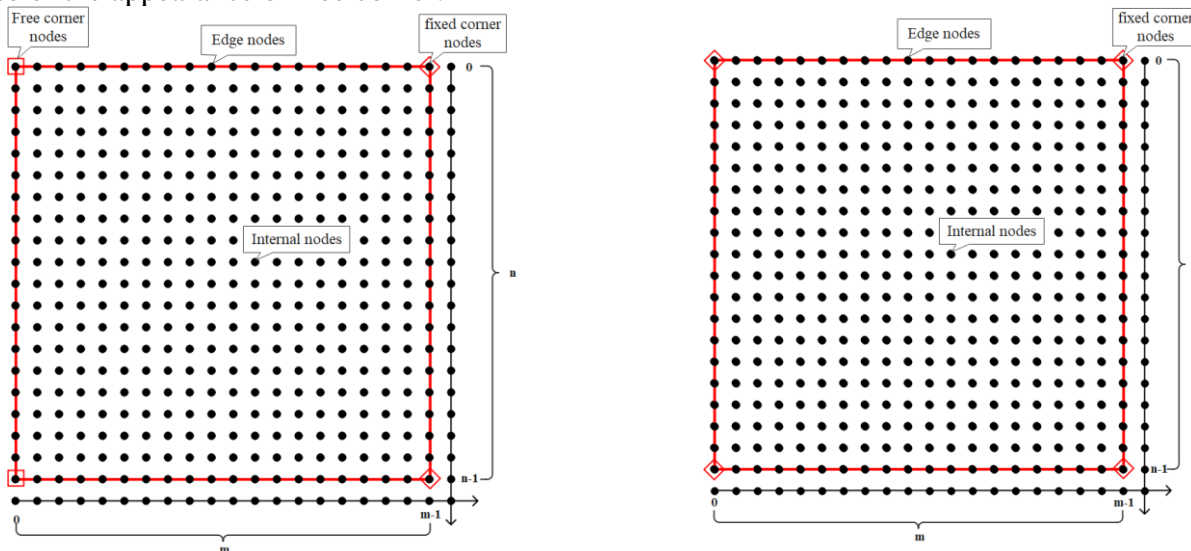
Code 4: plot the vertical displacement in global z-axis for different models

3.6 Formation of square matrix [M]

As shown in above section, the constructed matrix is in rectangular shape. It requires the solver of shell code should able to solve an overdetermined sparse system. Only an approximated solution can be found for such system. It is possible to find a way to make the matrix square by replacing boundary condition equations at edge nodes with S-K equations at internal nodes. Solving such square matrix might produce results with higher accuracy compared to the solution of rectangular matrix. Below sections show how the equations are distributed for nodes and show three different ways of making the matrix square.

a) Distribution of assigned equations in rectangular matrix

As shown above, the matrix [M] is a rectangular sparse matrix because additional boundary condition equations are defined for edge and corner nodes. In order to make a square matrix, some of the equations at edges and corners need to be removed. The distribution of assigned equations for the nodes can be different in case of the appearance of free corner.



a) Cantilever

b) Pinned edges

Figure 9: Distribution of assigned equations in the rectangular matrix

Table 10: Number of assigned equations in rectangular [M]

Node type	Cantilever: Number of nodes	Pinned edges: Number of nodes	N_{SK}	N_{BE}	N_{ABE}
-----------	--------------------------------	----------------------------------	----------	----------	-----------

Edge nodes	$2 \cdot m + 2 \cdot n - 8$	$2 \cdot m + 2 \cdot n - 8$	21	4	/
Free corner nodes	2	/	21	8	2
Fixed corner nodes	2	4	21	8	/
Internal nodes	$(m-2) \cdot (n-2)$	$(m-2) \cdot (n-2)$	21	/	/
(Total number of nodes)	$m \cdot n$	$m \cdot n$			

(N_{SK} : Number of defined Sanders-Koiter equations per node,
 N_{BE} : Number of defined regular boundary equations per node
 N_{ABE} : Number of defined additional boundary equations per node)

In addition, on the cantilever fixed edge corner node only impose $u_x = u_y = u_x = \text{phix} = 0$. The reason is that this node is fixed and not free at the same time. The adjacent node on the free edge is really free

The total number of equations are calculated for the two boundary conditions which determines the number of rows in rectangular matrix [M].

Cantilever: $(21+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (21+8+2) \cdot 2 + (21+8) \cdot 2 + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn + 8 \cdot m + 8 \cdot n + 4$

Pinned edges: $(21+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (21+8) \cdot 4 + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn + 8 \cdot m + 8 \cdot n$

b) Central node method

As boundary equations are essential for constraining shell behavior, some Sanders-Koiter equations at internal nodes have to be taken out. One way of doing that is only add the full Sanders-Koiter equations to a small group of central nodes ($4 < i < m-5$, $4 < j < n-5$). Except the internal nodes marked below, only 20 Sanders-Koiter equations are added for the rest nodes.

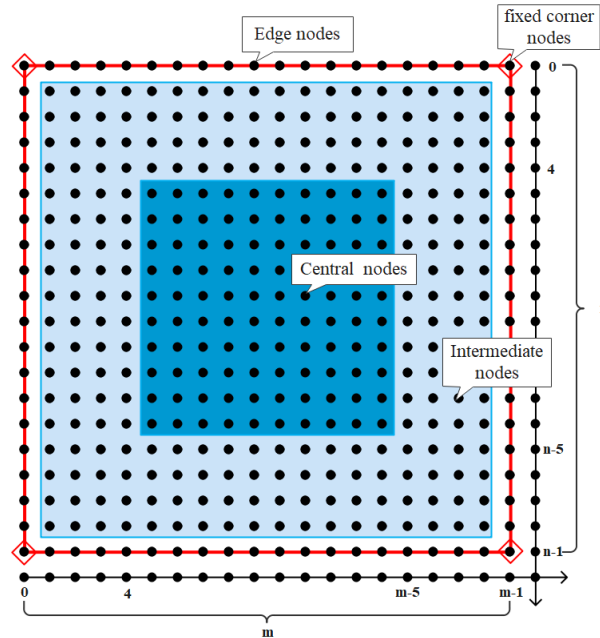


Figure 10: Modified distribution of assigned equations by central node method

Node type	Cantilever: Number of nodes	Pinned edges: Number of nodes	N_{SK}	N_{BE}	N_{ABE}
-----------	-----------------------------	-------------------------------	----------	----------	-----------

Edge nodes	$2 \cdot m + 2 \cdot n - 8$	$2 \cdot m + 2 \cdot n - 8$	20	4	/
Free corner nodes	2	/	20	8	2
Fixed corner nodes	2	4	20	8	/
Intermediate nodes	$2 \cdot (4 \cdot (m-6)) + 2 \cdot (4 \cdot (n-6))$	$2 \cdot (4 \cdot (m-6)) + 2 \cdot (4 \cdot (n-6))$	20		
Central nodes	$(m-10) \cdot (n-10)$	$(m-10) \cdot (n-10)$	21	/	/
(Total number of nodes)	$m \cdot n$	$m \cdot n$			

Table 11: Number of assigned equations in square [M] by central node method

The total number of equations are calculated for the two boundary conditions which determines the number of rows in rectangular matrix [M]:

Cantilever: $(20+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (20+8+2) \cdot 2 + (20+8) \cdot 2 + 20 \cdot [2 \cdot (3 \cdot (m-6)) + 2 \cdot (3 \cdot (n-6))] + 21 \cdot (m-10) \cdot (n-10) = 21 \cdot mn - 2 \cdot m - 2 \cdot n + 104$

Pinned edges: $(20+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (20+8) \cdot 4 + 20 \cdot [2 \cdot (3 \cdot (m-6)) + 2 \cdot (3 \cdot (n-6))] + 21 \cdot (m-10) \cdot (n-10) = 21 \cdot mn - 2 \cdot m - 2 \cdot n + 100$

In fact, by removing one Sanders-Koiter equation for intermediate nodes, when $m=n>25$, the previous overdetermined matrix becomes underdetermined. The spare rows are filled with zeros in order to make the matrix square. If $m=n \leq 25$, the row number is larger than the column number making an overdetermined matrix. So that in following model tests, the model 1 and model 2 with 20×20 node are not tested.

```

for j in range(n): # Add Sanders-Koiter equation 21 to the matrix -----
    for i in range(m):
        BCs() # adding boundary conditions at edges and corners

        if i < m-5 and i > 4 and j < n-5 and j > 4:
            row=row+1
            M[row, rhoxy*m*n+j*m+i]=-1
            Dly(phi_x, 1.0)
            Dlx(phi_y, 1.0)
            M[row, phi_z*m*n+j*m+i]=kxx(i/(m-1), j/(n-1))-kyy(i/(m-1), j/(n-1))
            M[row, phi_x*m*n+j*m+i] +=-kx(i/(m-1), j/(n-1))
            M[row, phi_y*m*n+j*m+i] +=-ky(i/(m-1), j/(n-1))

```

Code 5: Adding Sanders-Koiter equation 21 only to central nodes

c) Undefined node method

Another way is simply not adding one of the Sanders-Koiter equations for all nodes. The adding process stops when the number of remained nodes is equal to the number of required boundary equations. In this way, there is a part of the internal nodes only having 20 Sanders-Koiter equations which is called an undefined internal node as shown below.

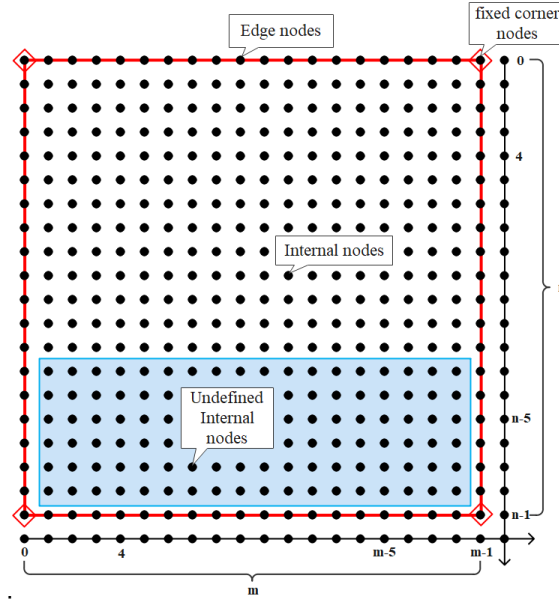


Figure 11: Modified distribution of assigned equations by undefined node method

Table 12: Number of assigned equations in square $[M]$ by undefined node method

Node type	Cantilever: Number of nodes	Pinned edges: Number of nodes	N_{SK}	N_{BE}	N_{ABE}
Edge nodes	$2 \cdot m + 2 \cdot n - 8$	$2 \cdot m + 2 \cdot n - 8$	21	4	/
Free corner nodes	2	/	21	8	2
Fixed corner nodes	2	4	21	8	/
Undefined internal nodes	$8 \cdot m + 8 \cdot n + 4$	$8 \cdot m + 8 \cdot n$	20		
Internal nodes	$(m-2) \cdot (n-2) - (8 \cdot m + 8 \cdot n + 4)$	$(m-2) \cdot (n-2) - (8 \cdot m + 8 \cdot n)$	21	/	/
(Total number of nodes)	$m \cdot n$	$m \cdot n$			

Under such configuration, the calculated total number of equations is equal to the number of columns in both two boundary conditions.

$$\text{Cantilever: } (21+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (21+8+2) \cdot 2 + (21+8) \cdot 2 + 20 \cdot [(m-2) \cdot (n-2) - (8 \cdot m + 8 \cdot n + 4)] + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn$$

$$\text{Pinned edges: } (21+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (21+8) \cdot 4 + 20 \cdot [(m-2) \cdot (n-2) - (8 \cdot m + 8 \cdot n + 4)] + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn$$

```

for j in range(n): # Add Sanders-Koiter equation 21 to the matrix -----
    for i in range(m):
        row = row + 1
        M[row, rhoxy*m*n+j*m+i]=-1
        Dly(phi_x, 1.0)
        Dlx(phi_y, 1.0)
        M[row, phi_z*m*n+j*m+i]=kxx(i/(m-1), j/(n-1))-kyy(i/(m-1), j/(n-1))
        M[row, phi_x*m*n+j*m+i] +=-kx(i/(m-1), j/(n-1))
        M[row, phi_y*m*n+j*m+i] +=-ky(i/(m-1), j/(n-1))
        if row >= B-(C)-1: # B = 21*(m)*(n); C = 8*n+8*m+4 or 8*n+8*m+
            break
    else:
        continue
    if row >= B-(C)-1:
        break

for j in range(n): # adding boundary conditons -----
    for i in range(m):
        BCs()

```

Code 6: Adding Sanders-Koiter equation 21 to nodes except the undefined nodes

d) Equation replacement method

The third method is replacing specific Sanders-Koiter equations at only edge nodes. The selection of replaced Sanders-Koiter equations follows the types of quantities in defined boundary condition equations. Additionally, several Sanders-Koiter equations at corners are also replaced based on the types of quantities in defined corner boundary condition equations. The number of replaced Sanders-Koiter equations is equal to the number of added boundary equations at edges and corners. The correlation between replaced Sanders-Koiter equations and added boundary equations is listed in below.

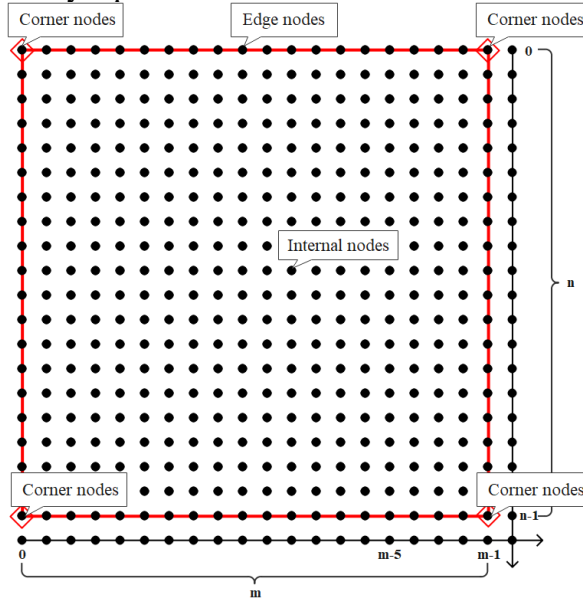


Figure 12: Modified distribution of assigned equations by undefined node method

Added boundary equations	Replaced Sanders-Koiter equations	Added boundary equations	Replaced Sanders-Koiter equations
$u_z = 0$	S-K equation 1	$n_{xy} - k_{yy}m_{xy} = 0$	S-K equation 9
$u_x = 0$	S-K equation 2	$n_{yx} - k_{xx}m_{xy} = 0$	S-K equation 9
$u_y = 0$	S-K equation 3	$m_{xx} = 0$	S-K equation 10
$v_x + \frac{\partial m_{xy}}{\partial y} = 0$	S-K equation 4	$m_{yy} = 0$	S-K equation 11
$v_y + \frac{\partial m_{xy}}{\partial x} = 0$	S-K equation 5	$m_{xy} = 0$	S-K equation 12
$n_{xx} - k_{xy}m_{xy} = 0$	S-K equation 7	$\phi_x = 0$	S-K equation 16

$n_{yy} - k_{xy}m_{xy}$	S-K equation 8	$\phi_y = 0$	S-K equation 17
-------------------------	----------------	--------------	-----------------

Table 13: Replaced S-K equations and boundary equations correlation

For pinned edges boundary condition, as shown in boundary equation table (Table 8), five different boundary equations are involved: $u_z = 0$, $u_x = 0$, $u_y = 0$, $m_{xx} = 0$ and $m_{yy} = 0$. The corresponding replaced Sanders-Koiter equations are equation 1 to 3 and 10 to 11. For cantilever boundary condition, the replaced Sanders-Koiter equations include equation 1 to 11 and 16. Based on boundary equations, four corresponding Sanders-Koiter equations are replaced for each edge node. For corner nodes, the Sanders-Koiter equations are removed without adding boundary equations to avoid repeated definition of boundary condition. The required boundary condition equations are defined by adding additional boundary equations.

Table 14: Number of assigned equations in square [M] by undefined node method

Node type	Cantilever: Number of nodes	Pinned edges: Number of nodes	N_{SK}	N_{BE}	N_{ABE}
Edge nodes	$2 \cdot m + 2 \cdot n - 8$	$2 \cdot m + 2 \cdot n - 8$	17	4	/
Free corner nodes	2	/	17	0	4
Fixed corner nodes	2	4	17	0	4
Internal nodes	$(m-2) \cdot (n-2)$	$(m-2) \cdot (n-2)$	21	/	/
(Total number of nodes)	$m \cdot n$	$m \cdot n$			

The total number of equations are calculated for the two boundary conditions which determines the number of rows in rectangular matrix [M]:

$$\begin{aligned} \text{Cantilever:} & (17+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (17+4) \cdot 2 + (17+4) \cdot 2 + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn \\ \text{Pinned edges:} & (17+4) \cdot (2 \cdot m + 2 \cdot n - 8) + (17+4) \cdot 4 + 21 \cdot (m-2) \cdot (n-2) = 21 \cdot mn \end{aligned}$$

Table 15: Additional boundary equations

	Pinned edges	Cantilever
Free corners	/	$\frac{n_{xy} + n_{yx}}{2} = 0, m_{xy} = 0$ $m_{xx} = 0, m_{yy} = 0$
Fixed corners	$u_z = 0, u_x = 0, u_y = 0,$ $m_{xx} = 0$ or $m_{yy} = 0$	$u_z = 0, u_x = 0, u_y = 0,$ $\phi_x = 0$ or $\phi_y = 0$

3.7 Model tests of rectangular and square matrix

For the purpose of finding the approximation of results during the sparse matrix solving, both rectangular and square matrices are constructed for all models. The constructed systems are solved with a least square solver (Python solver: from `scipy.sparse.linalg` import `lsqr`) for further comparison. The test configuration is listed below.

Table 16: Rectangular and square matrix test configuration

(Least square method solver)	Model 1	Model 2	Model 3	Model 4	Model 5
Rectangular matrix	Test R1	Test R2	Test R3	Test R4	Test R5
Square matrix	Test SC1	Test SC2	Test SC3	Test SC4	Test SC5

(central node method)					
Square matrix (undefined node method)	Test SU1	Test SU2	Test SU3	Test SU4	Test SU5
Square matrix (equation replacement method)	Test SE1	Test SE2	Test SE3	Test SE4	Test SE5

3.8 Solver tests

The performances of three solvers have been compared

- 1) Python least square method (rectangular and square matrices)
- 2) Python singular value decomposition (rectangular)
- 3) R linear sparse model fitting (rectangular and square)

Table 17 and 18 show the extra computations that have been performed.

By various initial inputs of geometry, boundary conditions, load components and node number, sparse matrices $[M]$ and $[f]$ are constructed and saved in files. To evaluate the capacity of various solvers, all models are solved with other solvers than the least square method solver. The test configuration is list as below.

Table 17: Solver test configuration of rectangular matrices

	Model 1	Model 2	Model 3	Model 4	Model 5
Python solver: from numpy.linalg import pinv (Singular value decomposition)	Test P1	Test P2	Test P3	Test P4	Test P5
R solver: MatrixModels::lm.fit.sparse (Linear sparse model fitting)	Test LM1	Test LM2	Test LM3	Test LM4	Test LM5

The sparse rectangular matrix $[M]$ is constructed in the form of the Python dok matrix (dictionary of keys based sparse matrix). The completely assembled matrix $[M]$ is converted into coordinate format and saved as npz. file whose size is determined by the number of nodes. The size of the saved $[M]$ is 175 kb for 30×30 node model and 450 kb for 50×50 node model. If a R solver is applied, the $[M]$ have to be reconstructed into a readable R sparse matrix, while sparse matrix $[f]$ is constructed in the form of a Python array and saved as npy. file. The size of saved $[f]$ is 154 kb for 30×30 node model and 420 kb for a 50×50 node model. Then the calculated $[u]$ array is saved as npy. file and results are extracted from it to generate the 21 plots (Fig. 17). The above process can be summarized as the flow chart below. Beside the generated plots, the computational time and memory usage during the execution process is also recorded.

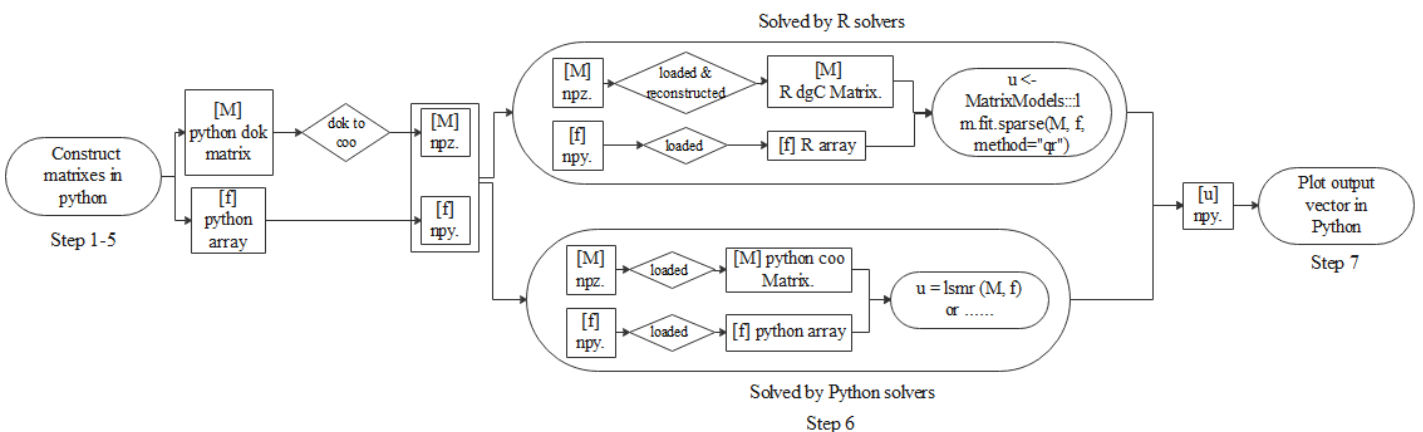


Figure 13: Work flow of solver tests

For the constructed sparse square matrix [M] by equation replacement method, the R script solver of linear model fitting is used to be compared with the results of least square method solver.

Table 18: Solver tests of square matrices (The matrices are made square by the equation replacement method)

(Square matrix, Equation replacement method)	Model 1	Model 2	Model 3	Model 4	Model 5
R solver: MatrixModels:::lm.fit.sparse (Linear sparse model fitting)	Test SLM1	Test SLM2	Test SLM3	Test SLM4	Test SLM5

3.9 Five-point difference approximation and their tests

In the above tests, three-point interpolation is used for the first derivative expression in the Sanders-Koiter equations. When performing a finite number of steps to approximate a process with infinitely many steps, discrepancies arise between the approximation and actual expression. By expanding the function in a Taylor series around the point where the finite difference formula approximates the derivative, the truncation error can be calculated.

	Truncation error
Central difference:	$R = \frac{1}{24} f'''(t) \Delta t^2 + O(\Delta t^4)$
Forward difference:	$R = \frac{1}{3} f'''(t) \Delta t^2 + O(\Delta t^3)$
Backward difference:	$R = -\frac{1}{3} f'''(t) \Delta t^2 + O(\Delta t^3)$

The dominating term for small Δt is $\frac{1}{24} f'''(t) \Delta t^2$, $\frac{1}{3} f'''(t) \Delta t^2$ and $-\frac{1}{3} f'''(t) \Delta t^2$ in above expressions which are proportional to Δt^2 and the truncation error for two-point difference approximations is of second order in Δt . Thus, three-point difference approximation is a second-order accurate discretization of the derivative. Such error might play an essential role in evaluating the accuracy of calculation results of shell code. In order to investigate the effect of truncation error, two set of difference approximations with higher level accuracy were used in several tests. They both are five-point interpolations but with different difference coefficients as listed below.

Five-point difference approximation type A:

Central difference:	$D_x(k, g) = \frac{3g}{4\alpha_x} (f_{i-1} - f_{i+1}) + \frac{g}{8\alpha_x} (-f_{i-2} + f_{i+2})$	$D_y(k, g) = \frac{g}{\alpha_y} (\dots f_j \dots)$
Forward difference:	$D_x(k, g) = \frac{g}{\alpha_x} \left(-\frac{1}{56} f_i + \frac{3}{28} f_{i+1} - \frac{3}{7} f_{i+2} + \frac{45}{28} f_{i+3} - \frac{71}{56} f_{i+4} \right)$	$D_y(k, g) = \frac{g}{\alpha_y} (\dots f_j \dots)$
	$D_x(k, g) = \frac{g}{\alpha_x} \left(-\frac{13}{28} f_{i-1} - \frac{3}{14} f_i + \frac{6}{7} f_{i+1} - \frac{3}{14} f_{i+2} + \frac{1}{28} f_{i+3} \right)$	$D_y(k, g) = \frac{g}{\alpha_y} (\dots f_j \dots)$
Backward difference:	$D_x(k, g) = -\frac{g}{\alpha_x} \left(-\frac{1}{56} f_i + \frac{3}{28} f_{i-1} - \frac{3}{7} f_{i-2} + \frac{45}{28} f_{i-3} - \frac{71}{56} f_{i-4} \right)$	$D_y(k, g) = \frac{g}{\alpha_y} (\dots f_j \dots)$
	$D_x(k, g) = -\frac{g}{\alpha_x} \left(-\frac{13}{28} f_{i+1} - \frac{3}{14} f_i + \frac{6}{7} f_{i-1} - \frac{3}{14} f_{i-2} + \frac{1}{28} f_{i-3} \right)$	$D_y(k, g) = \frac{g}{\alpha_y} (\dots f_j \dots)$

Five-point difference approximation type B:

Central difference:	$D_x(k, g) = \frac{8g}{12\alpha_x}(f_{i-1} - f_{i+1}) + \frac{g}{12\alpha_x}(-f_{i-2} + f_{i+2})$	$D_y(k, g) = \frac{g}{\alpha_y}(\dots\dots f_j \dots\dots)$
Forward difference:	$D_x(k, g) = \frac{g}{12\alpha_x}(-25f_i + 48f_{i+1} - 36f_{i+2} + 16f_{i+3} - 3f_{i+4})$	$D_y(k, g) = \frac{g}{\alpha_y}(\dots\dots f_j \dots\dots)$
	$D_x(k, g) = \frac{g}{12\alpha_x}(-3f_{i-1} - 10f_i + 18f_{i+1} - 6f_{i+2} + f_{i+3})$	$D_y(k, g) = \frac{g}{\alpha_y}(\dots\dots f_j \dots\dots)$
Backward difference:	$D_x(k, g) = -\frac{g}{12\alpha_x}(-25f_i + 48f_{i-1} - 36f_{i-2} + 16f_{i-3} - 3f_{i-4})$	$D_y(k, g) = \frac{g}{\alpha_y}(\dots\dots f_j \dots\dots)$
	$D_x(k, g) = -\frac{g}{12\alpha_x}(-3f_{i+1} - 10f_i + 18f_{i-1} - 6f_{i-2} + f_{i-3})$	$D_y(k, g) = \frac{g}{\alpha_y}(\dots\dots f_j \dots\dots)$

New tests with five-point difference approximations are proposed and their test configuration is in the list below:

(Rectangular matrix)	Solver type	Model 1	Model 2	Model 3	Model 4	Model 5
Type A approximation	Least square method solver	Test AR1	Test AR2	Test AR3	Test AR4	Test AR5
	Linear sparse model fitting	Test ALM1	Test ALM2	Test ALM3	Test ALM4	Test ALM5
Type B approximation	Least square method solver	Test BR1	Test BR2	Test BR3	Test BR4	Test BR5
	Linear sparse model fitting	Test BLM1	Test BLM2	Test BLM3	Test BLM4	Test BLM5

4. Results

This section presents the test results obtained mainly as plots and extreme values, and compares them. Other information like solver capacity and singularities are also shown. For each run of the code, 21 plots were generated. In order to prevent excessive work, only three plots (displacement u_z , bending moment m_{xx} , and shear force v_x) and their extreme values were analyzed. Below sections only show the extreme values and the plots are listed in appendix (Page 82). The first shown are the plots generated by finite element software, which work as a reference for calculating the errors of shell code results. Subsequently presented are results from rectangular tests, square tests, solver tests, and five-point difference approximation tests. These results are compared. In addition, several 3D plots show the overall deformation of the structures and distortion at boundaries more intuitively. At the end of this section is a summary of major findings.

4.1 Finite element solution

The defined five models are also tested in finite element software SCIA Engineer (version 19.1.3). The generated displacement, bending moment and shear force plots are shown in below. The averaged element size is 0.05 m for all models so that the number of nodes is 20×20 for model 1 & 2 and 240×125 for model 3, 4 & 5.

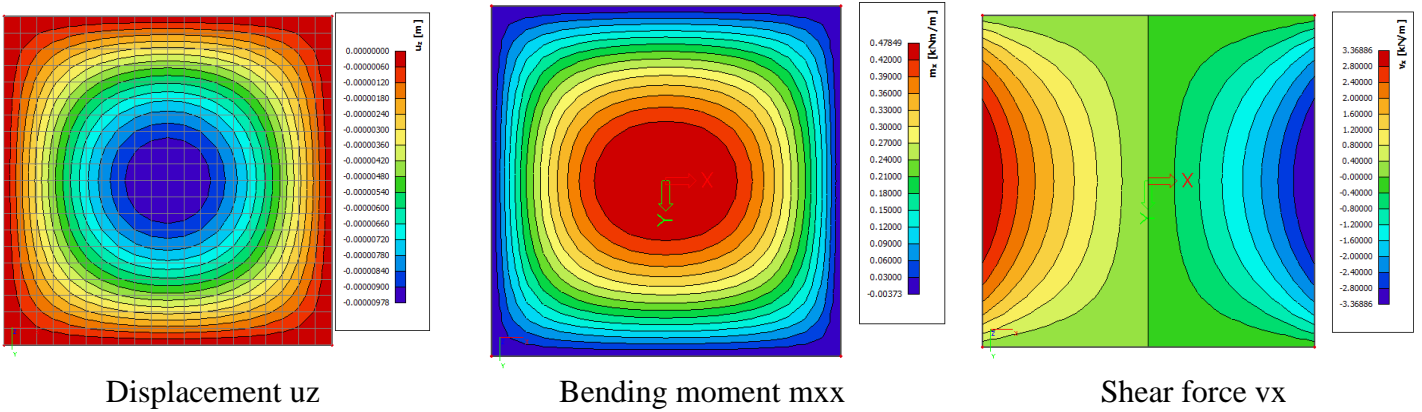


Figure 14: Model 1 displacement, bending moment and shear force finite element results by SCIA Engineer

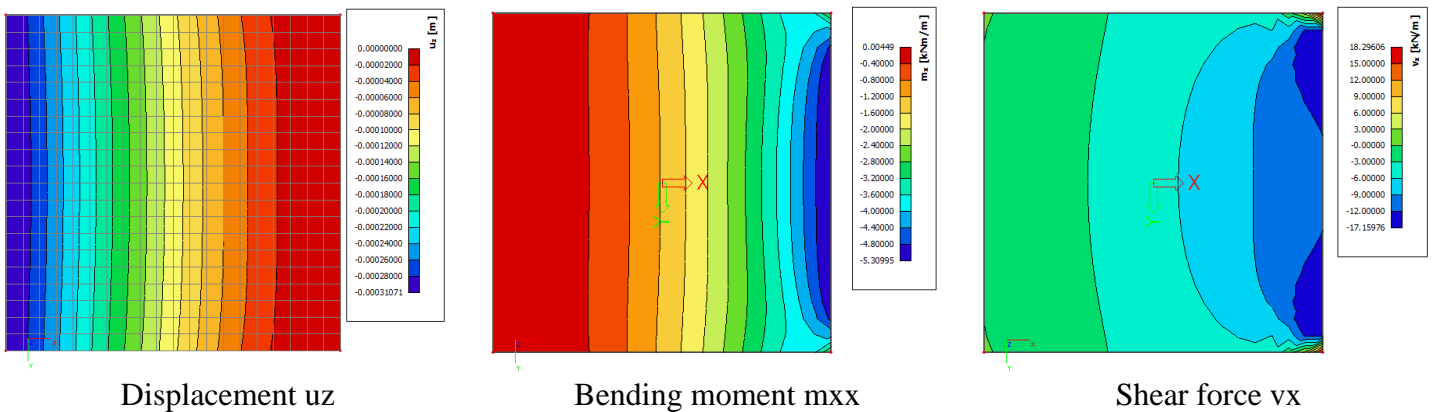
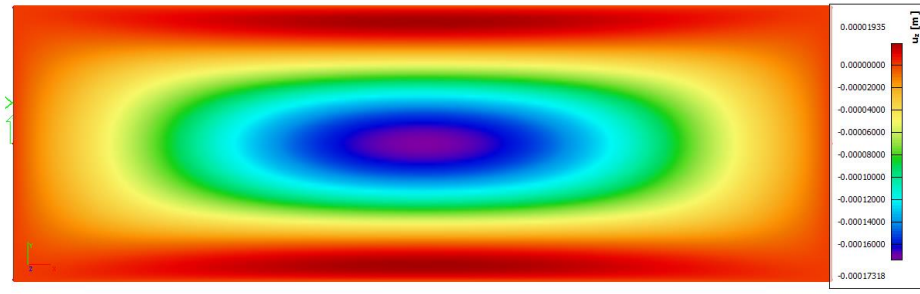
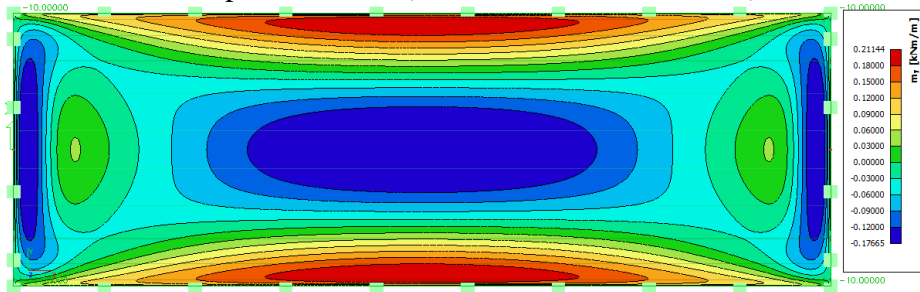


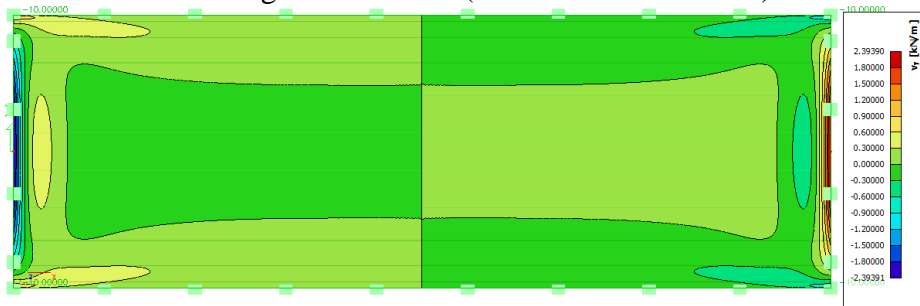
Figure 15: Model 2 displacement, bending moment and shear force finite element results by SCIA Engineer



Displacement uz (abs max value: 0.0001935)

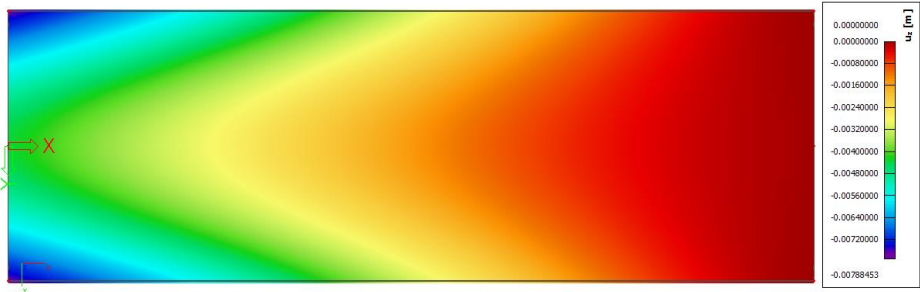


Bending moment mxx (abs max value: 0.2114)

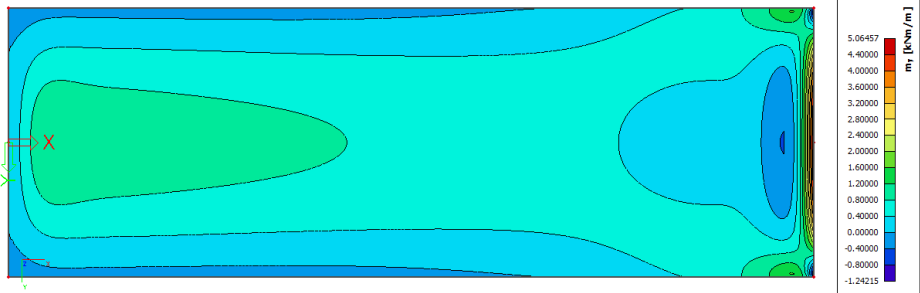


Shear force vx (abs max value: 2.393)

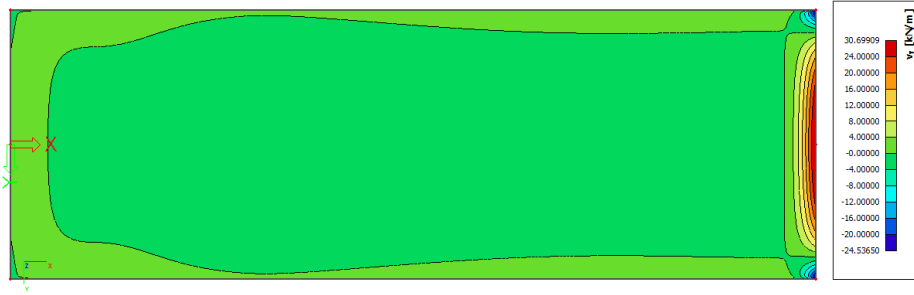
Figure 16: Model 3 displacement, bending moment and shear force finite element results by SCIA Engineer



Displacement uz (abs max value: 0.007884)

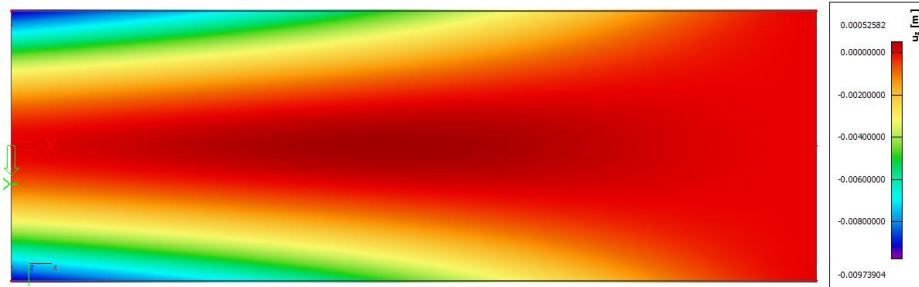


Bending moment mxx (abs max value: 5.0646)

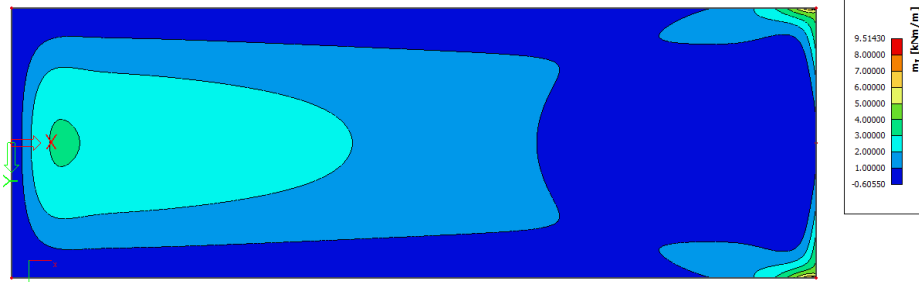


Shear force vx (abs max value: 30.699)

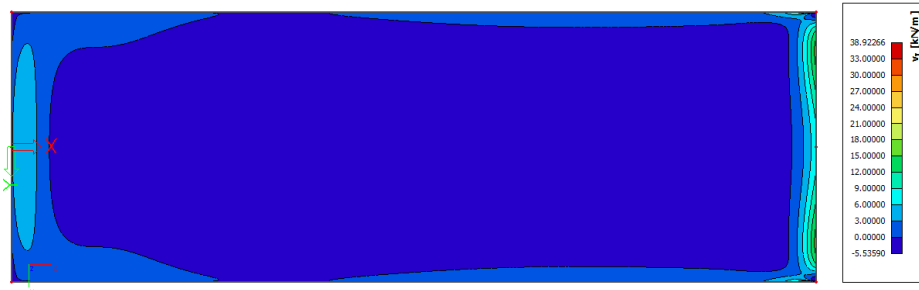
Figure 17: Model 4 displacement, bending moment and shear force finite element results by SCIA Engineer



Displacement uz (abs max value: 0.009739)



Bending moment mxx (abs max value: 9.5143)



Shear force vx (abs max value: 38.9266)

Figure 18: Model 5 displacement, bending moment and shear force finite element results by SCIA Engineer

Table 19: Absolute maximum value of finite element results for model1-5

	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement uz (m)	$9.78 \cdot 10^{-6}$	$3.107 \cdot 10^{-4}$	$1.935 \cdot 10^{-4}$	$7.884 \cdot 10^{-3}$	$9.734 \cdot 10^{-3}$
Bending moment mxx (kNm/m)	0.478	5.310	0.2114	5.0646	9.5143
Shear force vx (kN/m)	3.369	18.296	2.393	30.699	38.9366

4.2 Rectangular matrix test results

(Please find plots in Appendix)

Table 20: Absolute maximum value of test R1-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	$7.333 \cdot 10^{-6}$	$2.896 \cdot 10^{-4}$			
	30*30	$7.006 \cdot 10^{-6}$	$2.525 \cdot 10^{-4}$	$3.062 \cdot 10^{-3}$	0.166	0.101
	50*50	$6.728 \cdot 10^{-6}$	$1.746 \cdot 10^{-4}$	$3.281 \cdot 10^{-3}$	0.182	0.116
Bending moment (kNm/m)	20*20	0.39	7.42			
	30*30	0.39	7.29	5.71	290.74	179.39
	50*50	0.39	8.73	5.77	299.47	189.15
Shear force (kN/m)	20*20	3.28	4.45			
	30*30	3.32	4.52	15.15	95.51	58.49
	50*50	3.32	4.61	15.13	97.50	60.62

4.3 Square matrix test results

(Please find plots in Appendix)

Table 21: Absolute maximum value of test SCI-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20					
	30*30	6.063E-6	2.594E-04	2.526E-03	1.398	1.085
	50*50	6.146E-6	1.779E-04	2.970E-03	1.720	0.885
Bending moment (kNm/m)	20*20					
	30*30	0.252	8.238	5.691	537.988	340.874
	50*50	0.246	7.55	5.759	478.973	304.628
Shear force (kN/m)	20*20					
	30*30	3.316	4.162	15.154	119.880	74.463
	50*50	3.342	4.049	15.137	112.732	70.404

Table 22: Absolute maximum value of test SUI-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	5.915E-06	3.699E-04			
	30*30	5.644E-06	3.317E-04	2.681E-03	1.727	1.087
	50*50	5.714E-06	1.997E-04	2.948E-03	1.406	0.888
Bending moment (kNm)	20*20	0.261	8.094			
	30*30	0.252	8.224	5.692	538.000	340.879
	50*50	0.246	7.551	5.759	478.976	304.630
Shear force (kN/m)	20*20	3.277	4.229			
	30*30	3.316	4.162	15.154	119.876	74.461
	50*50	3.342	4.049	15.137	112.731	70.403

Table 23: Absolute maximum value of test SE1-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	6.715E-06	3.215E-05			
	30*30	6.517E-06	2.731E-05	3.152E-03	1.449	0.915
	50*50	6.413E-06	1.847E-04	3.348E-03	1.148	0.728
	20*20	0.269	7.693			

Bending moment (kNm/m)	30*30	0.258	7.791	5.655	501.722	310.424
	50*50	0.249	7.128	5.642	490.586	293.100
Shear force (kN/m)	20*20	3.068	4.105			
	30*30	3.164	4.058	15.767	135.343	83.841
	50*50	3.252	4.639	15.707	129.465	80.819

4.4 Solver test results

(Please find plots in Appendix)

Table 24: Absolute maximum value of test P1-5

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	10*10	9.741E-06	3.172E-04	1.307E-04	0.0103	0.0131
	20*20	9.704E-06	3.140E-04	1.647E-04	7.828	0.0144
Bending moment (kNm/m)	10*10	0.460	5.436	0.111	4.967	4.901
	20*20	0.473	5.691	0.182	1.624	5.192
Shear force (kN/m)	10*10	3.299	13.503	0.127	4.262	5.595
	20*20	3.368	15.854	0.458	1.278	4.162

Table 25: Absolute maximum value of test LMI-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	9.707E-06	3.142E-04			
	30*30	9.704E-6	3.134E-04	1.660E-4	7.703E-03	9.375E-03
	50*50	9.770E-06	3.126E-04	1.703E-04	7.776E-03	9.329E-03
Bending moment (kNm/m)	20*20	0.473	5.696			
	30*30	0.476	5.732	0.201	1.598	7.300
	50*50	0.478	5.519	0.208	2.590	8.927
Shear force (kN/m)	20*20	3.367	15.866			
	30*30	3.325	16.042	0.407	1.809	8.181
	50*50	3.363	15.881	0.578	8.887	13.311

Table 26: Absolute maximum value of test SLM1-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	9.707E-06	3.107E-04			
	30*30	9.748E-06	3.107E-04	1.685E-04	7.847E-03	9.671E-03
	50*50	9.764E-06	3.107E-04	1.715E-04	7.882E-03	9.559E-03
Bending moment (kNm/m)	20*20	0.472	5.311			
	30*30	0.476	5.312	0.201	1.676	7.080
	50*50	0.478	5.312	0.208	3.060	8.377
Shear force (kN/m)	20*20	3.355	15.502			
	30*30	3.368	37.308	0.836	4.040	21.910
	50*50	3.373	85.032	1.278	12.221	27.159

4.5 Five-point difference approximation test results

(Please find plots in Appendix)

Table 27: Absolute maximum value of test ARI-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	6.92E-06	1.62E-04	0.0026	0.1275	0.0798
	30*30	6.56E-06	2.22E-04	0.0031	0.1460	0.0916
	50*50	6.24E-06	2.43E-04	0.0027	0.1854	0.1229
Bending moment (kNm/m)	20*20	0.4008	5.6398	6.2022	270.1302	171.0242
	30*30	0.3903	6.5045	6.0320	281.4246	176.5635
	50*50	0.3815	7.8378	5.7824	297.1835	190.9475
Shear force (kN/m)	20*20	3.3058	4.2638	14.5238	103.3915	64.3321
	30*30	3.3283	4.4615	14.7720	102.5811	63.6840
	50*50	3.3459	4.5138	14.9345	102.8183	64.5399

Table 28: Absolute maximum value of test BRI-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	6.86E-06	5.62E-05	0.002	0.158	0.101
	30*30	6.52E-06	5.70E-05	0.003	0.225	0.145
	50*50	6.22E-06	5.46E-05	0.003	0.271	0.173
Bending moment (kNm/m)	20*20	0.397	3.965	5.863	229.598	145.720
	30*30	0.389	4.481	5.939	247.476	157.932
	50*50	0.380	4.919	5.830	253.871	161.883
Shear force (kN/m)	20*20	3.297	4.175	14.995	98.603	61.354
	30*30	3.323	5.106	15.102	100.866	63.118
	50*50	3.344	7.108	15.085	100.285	62.853

Table 29: Absolute maximum value of test ALM1-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	9.926E-06	3.119E-04	1.795E-04		
	30*30	9.660E-06	3.117E-04	1.412E-04	8.24E-03	9.39E-03
	50*50	9.671E-06	3.123E-04	1.536E-04	7.89E-03	9.59E-03
Bending moment (kNm/m)	20*20	0.4837	5.3274	1.0208		
	30*30	0.4757	5.4997	0.6447	3.4571	5.9238
	50*50	0.4757	5.5936	0.4981	2.9374	7.3001
Shear force (kN/m)	20*20	3.2730	13.0253	4.8254		
	30*30	3.0708	13.8312	3.7759	7.3638	6.0894
	50*50	3.0451	14.5493	3.4545	5.2990	8.1810

Table 30: Absolute maximum value of test BLM1-5 plots

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	9.72E-06	3.12E-04			
	30*30	9.75E-06	3.11E-04	1.71E-04	7.79E-03	9.48E-03
	50*50	9.77E-06	3.11E-04	1.73E-04	7.81E-03	9.51E-03
Bending moment (kNm/m)	20*20	0.4767	5.5872			
	30*30	0.4779	5.6588	0.2087	3.6551	9.6528
	50*50	0.4785	5.5902	0.2112	5.0066	9.2010
Shear force	20*20	3.3681	19.6723			

(kN/m)	30*30	3.3785	27.1999	0.8588	16.9374	17.1115
	50*50	3.3755	52.9049	2.4201	29.3033	14.4334

4.6 Overall comparison between shell code results

To show the accuracy of shell code results, the absolute maximum values of the above plots are compared to absolute maximum values obtained by the finite element method. The deviation is calculated as (shell code result - finite element result)/ finite element result. The deviation is converted into percentage form if it is in range of (-1 ,1). Those results are listed in the below tables and their absolute values are labeled with different colors where green represents less than 10%, blue represents ranged from 10% to 100%. and orange represents over 100%. Overall considered, the results of Test BLM1-5 show the best accuracy where green results take account for 78%. In the results of tests using *lsmr* solver (Test R1-5, Test SU1-5, Test SC1-5 and Test SE1-5), the accuracy of results of model 1 and model 2 is relatively good. The results of rest models are heavily overestimated.

Table 31: Deviation of Test R1-5 results (green:14%, blue: 42%, orange: 44%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-30.04%	-14.58%			
	30*30	-28.36%	-3.55%	12.121	20.012	9.417
	50*50	-33.35%	6.06%	13.024	22.039	10.913
Bending moment (kNm/m)	20*20	-17.58%	39.83%			
	30*30	-19.01%	37.32%	26.000	56.406	17.855
	50*50	-19.01%	64.46%	26.315	58.131	18.881
Shear force (kN/m)	20*20	-2.75%	-75.67%			
	30*30	-1.57%	-75.31%	5.332	2.11	50.23%
	50*50	-1.59%	-74.81%	5.324	2.18	55.69%

Table 32: Deviation of Test SU1-5 results (green:12%, blue: 44%, orange: 44%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-34.09%	19.05%			
	30*30	-38.07%	6.76%	12.855	218.051	110.670
	50*50	-37.47%	-35.73%	14.235	177.336	90.227
Bending moment (kNm)	20*20	-20.89%	52.43%			
	30*30	-47.19%	54.88%	25.925	105.228	34.828
	50*50	-21.02%	42.20%	26.242	93.573	31.018
Shear force (kN/m)	20*20	-7.91%	-76.89%			
	30*30	-1.57%	-77.25%	5.333	2.90	91.24%
	50*50	-1.66%	-77.87%	5.326	2.67	80.81%

Table 33: Deviation of Test SC1-5 results (green:7%, blue: 40%, orange: 53%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20					
	30*30	-38.07%	-93.98%	12.818	176.321	110.465
	50*50	-37.47%	-43.62%	14.349	217.163	89.918
Bending moment (kNm/m)	20*20					
	30*30	-47.19%	54.87%	25.948	105.225	34.828
	50*50	-21.02%	42.21%	26.242	93.573	31.018
Shear force (kN/m)	20*20					
	30*30	-1.57%	-77.25%	5.333	2.91	91.24%
	50*50	-1.66%	-77.87%	5.326	2.67	80.82%

Table 34: Deviation Test SE1-5 results (green:12%, blue: 44%, orange: 44%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-31.34%	3.48%			
	30*30	-33.36%	-12.10%	15.289	182.790	93.000
	50*50	-34.43%	-40.55%	16.302	144.611	73.789
Bending moment (kNm)	20*20	-43.72%	44.88%			
	30*30	-46.03%	46.72%	25.750	98.064	31.627
	50*50	-47.91%	34.24%	25.689	95.866	29.806
Shear force (kN/m)	20*20	-8.93%	-77.56%			
	30*30	-6.08%	-77.82%	5.589	3.41	115.33%
	50*50	-3.47%	-74.64%	5.564	3.22	107.57%

Table 35: Deviation of Test P1-5 (green:33%, blue: 67%, orange: 0%)

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	10*10	-0.40%	-98.98%	-32.45%	30.64%	34.58%
	20*20	-0.78%	-98.99%	-14.88%	-0.71%	47.94%
Bending moment (kNm)	10*10	-3.77%	2.37%	-47.49%	-1.93%	-48.49%
	20*20	-1.05%	7.18%	-13.91%	-67.93%	-45.43%
Shear force (kN/m)	10*10	-2.08%	-26.20%	-94.69%	-86.12%	-85.63%
	20*20	-0.03%	-13.35%	-80.86%	-95.84%	-89.31%

Table 36: Deviation of Test LM1-5 results (green:61%, blue: 39%, orange: 0%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-0.75%	1.13%			
	30*30	-0.43%	0.87%	-14.26%	-0.60%	-1.45%
	50*50	-0.10%	0.61%	-11.98%	-0.96%	-1.98%
Bending moment (kNm)	20*20	-1.05%	7.27%			
	30*30	-0.42%	7.95%	-5.15%	-68.46%	-23.27%
	50*50	0.00%	3.94%	-1.73%	-48.85%	-6.17%
Shear force (kN/m)	20*20	-0.06%	-13.28%			
	30*30	-1.31%	-12.32%	-83.17%	-94.11%	-78.99%
	50*50	-0.18%	-13.20%	-76.14%	-71.05%	-65.81%

Table 37: Deviation of Test SLM1-5 (green:53%, blue: 42%, orange: 5.7%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-0.75%	0.00%			
	30*30	-0.33%	0.00%	-14.21%	-0.47%	-0.65%
	50*50	-0.16%	0.00%	-11.99%	-0.03%	-1.80%
Bending moment (kNm)	20*20	-1.26%	0.02%			
	30*30	-0.42%	0.04%	-4.92%	-66.91%	-25.59%
	50*50	0.00%	0.04%	-1.61%	-39.58%	-11.95%
Shear force (kN/m)	20*20	-0.42%	-15.27%			
	30*30	-0.03%	103.91%	-65.06%	-89.62%	-43.73%
	50*50	0.12%	364.76%	-46.59%	-68.61%	-30.25%

Table 38: Deviation of Test ALM1-5 (green:12%, blue: 44%, orange: 44%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	1.5%	0.4%			
	30*30	-1.2%	0.3%	-27.04%	4.49%	-3.58%
	50*50	-1.1%	0.5%	-20.63%	0.12%	-1.45%
Bending moment (kNm)	20*20	1.2%	0.3%			
	30*30	-0.5%	3.6%	2.05	-31.74%	-37.74%
	50*50	-0.5%	5.3%	1.36	-42.00%	-23.27%
Shear force (kN/m)	20*20	-2.8%	-28.8%			
	30*30	-8.9%	-24.4%	0.58	-76.01%	-84.4%
	50*50	-9.6%	-20.5%	0.44	-82.74%	-79.0%

Table 39: Deviation of Test BLM1-5 (green:78%, blue: 19%, orange: 3%)

	Node number	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-0.6%	0.3%			
	30*30	-0.3%	0.3%	-11.83%	-1.18%	-2.56%
	50*50	-0.1%	0.1%	-10.84%	-1.00%	-2.33%
Bending moment (kNm)	20*20	-0.3%	5.2%			
	30*30	0.0%	6.6%	-1.28%	-27.83%	1.46%
	50*50	0.1%	5.3%	-0.09%	-1.15%	-3.29%
Shear force (kN/m)	20*20	0.0%	7.5%			
	30*30	0.3%	48.7%	-64.11%	-44.83%	-56.1%
	50*50	0.2%	1.89	1.13%	-4.55%	-62.9%

Table 40: Summary of deviation of test results

	Green ($\leq 10\%$)	Blue ($\leq 100\%$)	Orange ($> 100\%$)
Test R1-5	13.89%	41.67%	44.44%
Test SC1-5	6.67%	40.00%	53.33%
Test SU1-5	11.11%	44.44%	44.44%
Test SE1-5	11.11%	38.89%	50.00%
Test P1-5	33.33%	66.67%	0.00%
Test LM1-5	61.11%	38.89%	0.00%
Test SLM1-5	58.33%	36.11%	5.56%
Test AR1-5	11.11%	44.44%	44.44%
Test ALM1-5	52.78%	41.67%	5.56%
Test BLM1-5	77.78%	19.44%	2.78%

It can be observed that the accuracy of shell code results is strongly related to model types and test configurations. In order to show their influence more comprehensively, the deviation of shell code results was reorganized into model result plots categorized by its test type. In this way, it can be shown more clearly that how the shell results can be influenced by those important factors including the number of nodes, type of solver, type of matrix, and type of approximation. In the below plots, the deviation of all test results is shown in one figure for every model.

The type of solver is the most important factor on the accuracy. The *lm.fit.sparse* solver (Test LM, SLM, ALM, BLM) generally gives far more accurate results than *lsmr* solver (Test, R, AR, BR, SC, SU, SE). As shown in the summary table (Table 40), the green results in *lm.fit.sparse* solver tests take up over 50% while the percentage of green results in *lsmr* solver tests is less than 15%. As shown in Figure 24 to Figure 28, under *lm.fit.sparse* solver the most results of model 1 and model 2 have excellent accuracy around 1% while only shear force results of model 2 have deviation over 10%. The rest *lsmr* solver results are mostly orange where

the deviation of many results is even over 1000%. For model 3 to 5, in fact there are only a few green results no matter which solver is used.

Other factors like the type of matrices (Rectangular or square), difference approximation (five-point or two-point) play a less significant role in the accuracy. The different matrix types (square: Test SC, SU, SE or rectangular: Test R, AR, BR) and different finite difference approximation methods (two-point: Test R, LM or five-point: Test: AR, BR, ALM, BLM) did not cause not significant difference. In the most cases, the use of square matrix and new approximation method has made the shell code give less accurate results. As shown in summary table (Table 40), shell code give less green results when use square matrix and new approximation method. However, the Test BLM is an exception with the highest rate of green results.

Another important factor is number of nodes and its influence differs for different types of models and types of solvers. For models 1 to 3, generally the deviation of displacement and bending moment results is increased for higher number of nodes when *lsmr* solver is used. If *pinv* solver and *lm.fit.sparse* solver is used, a higher number of nodes bring lower deviation. For model 4 to 5, not matter which solver is used, higher number of nodes often means lower deviation.

By comparing absolute maximum values of plots to finite element results, the accuracy of shell code results can be found to be related to the type of solver, number of nodes, and type of model. The type of solver is the most important factor on the accuracy. The accuracy is also various between different types of results. Other factors like the type of matrices (Rectangular or square), difference approximation (five-point or two-point) does not play a significant role in the accuracy.

Accuracy by type of solver: R solver: *lm.fit.sparse* > Python solver: *pinv* > Python solver: *lsmr*

Accuracy by models: model 1 ≈ model 2 > model 3 ≈ model 4 ≈ model 5

Accuracy by type of matrices: square matrix ≈ rectangular matrix

Accuracy by type of difference approximation: five-point approximation > two- point approximation (only in a limited cases)

Accuracy by type of result: displacement result > shear force, bending moment result

Accuracy by number of nodes: 50*50 > 30*30 > 20*20

a) Overall test results per model

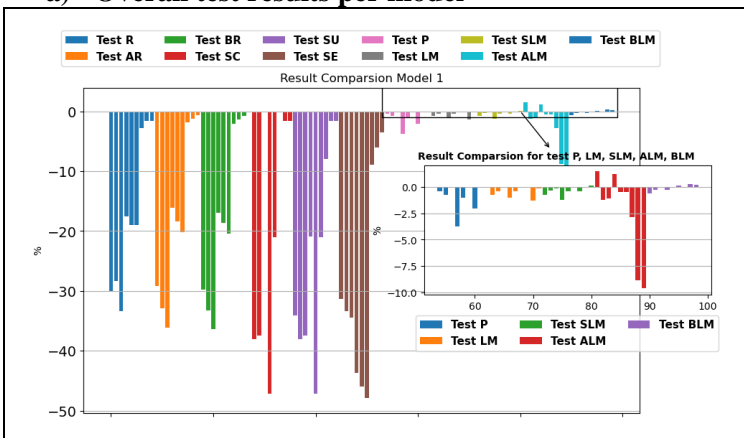


Figure 19: Comparison of overall test results for model 1

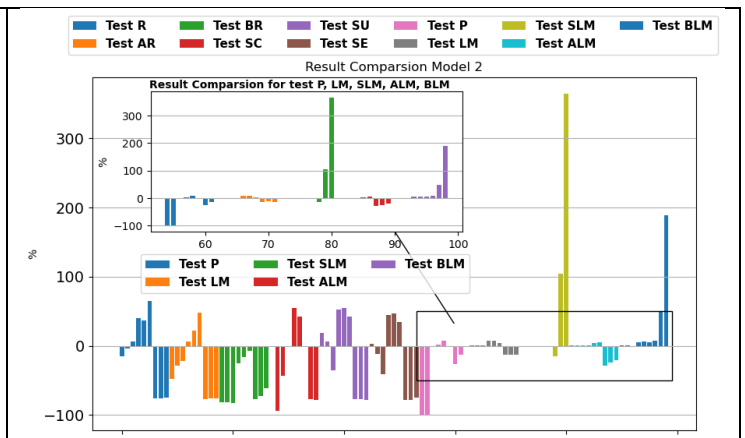


Figure 20: Comparison of overall test results for model 2

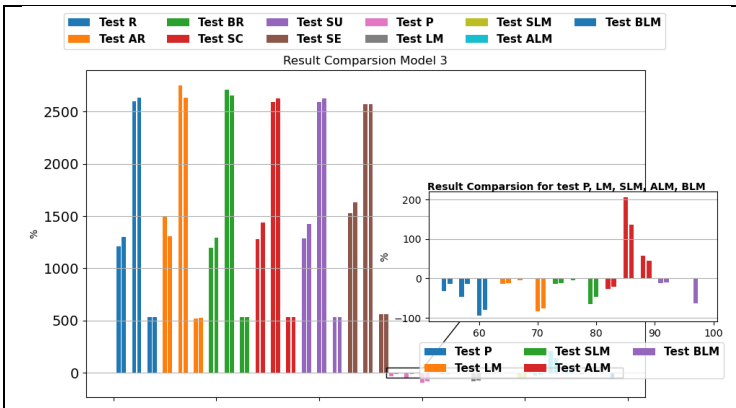


Figure 21: Comparison of overall test results for model 3

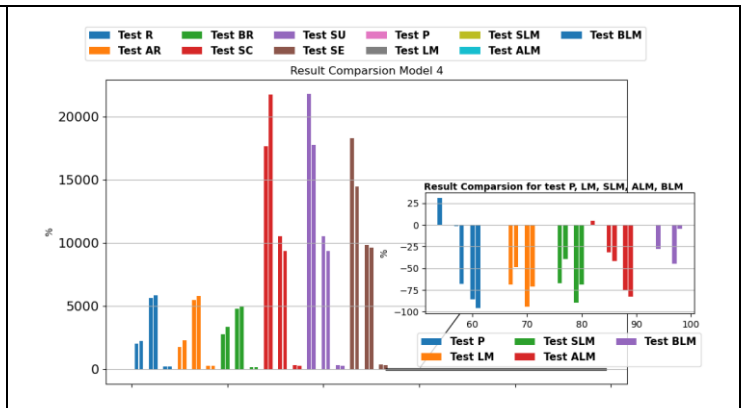


Figure 22: Comparison of overall test results for model 4

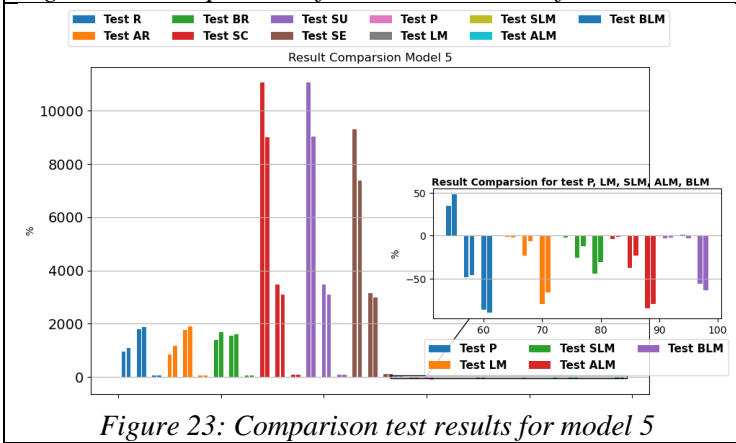
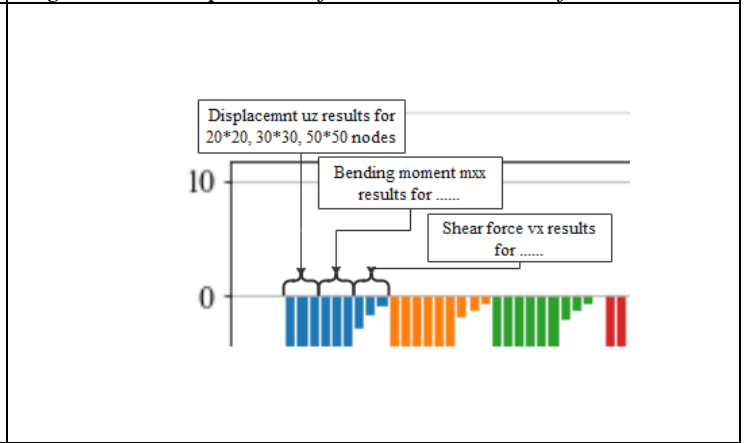


Figure 23: Comparison test results for model 5



b) Displacement uz test results per model

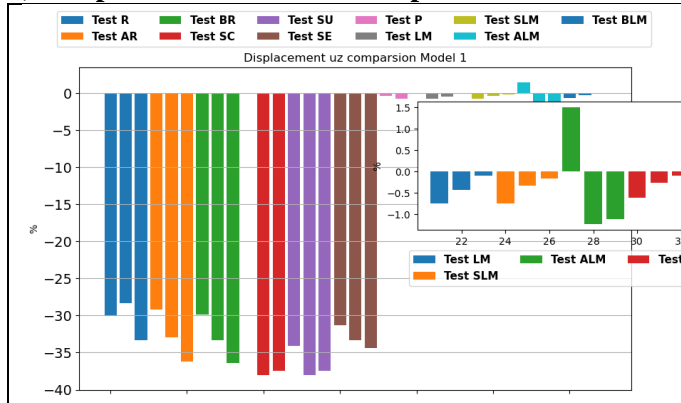


Figure 24: Comparison of uz results for model 1

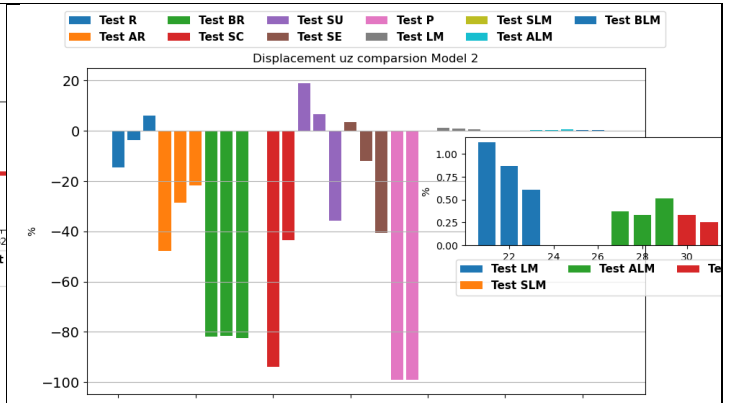


Figure 25: Comparison of uz results for model 2

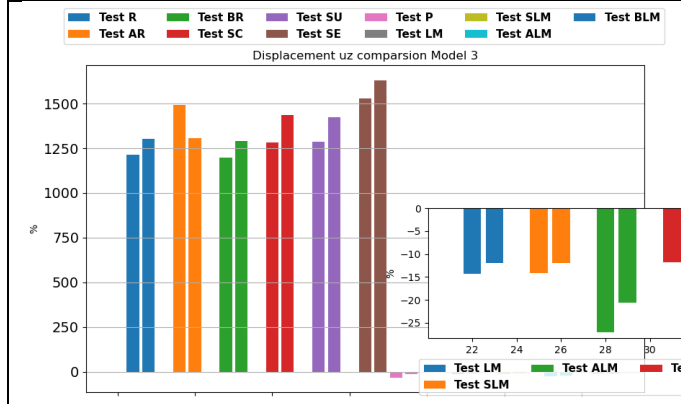


Figure 26: Comparison of uz results for model 3

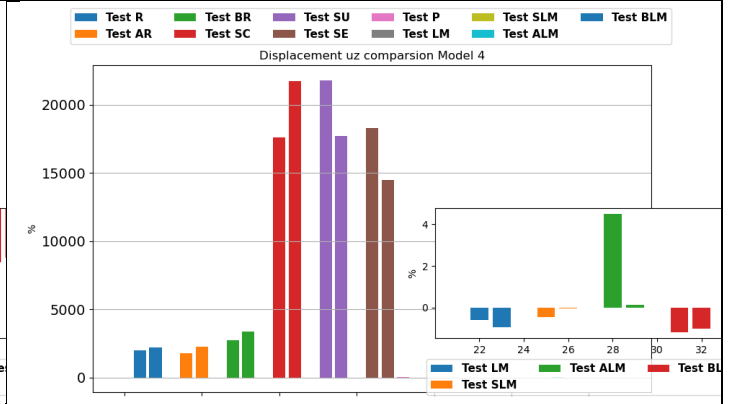


Figure 27: Comparison of uz results for model 4

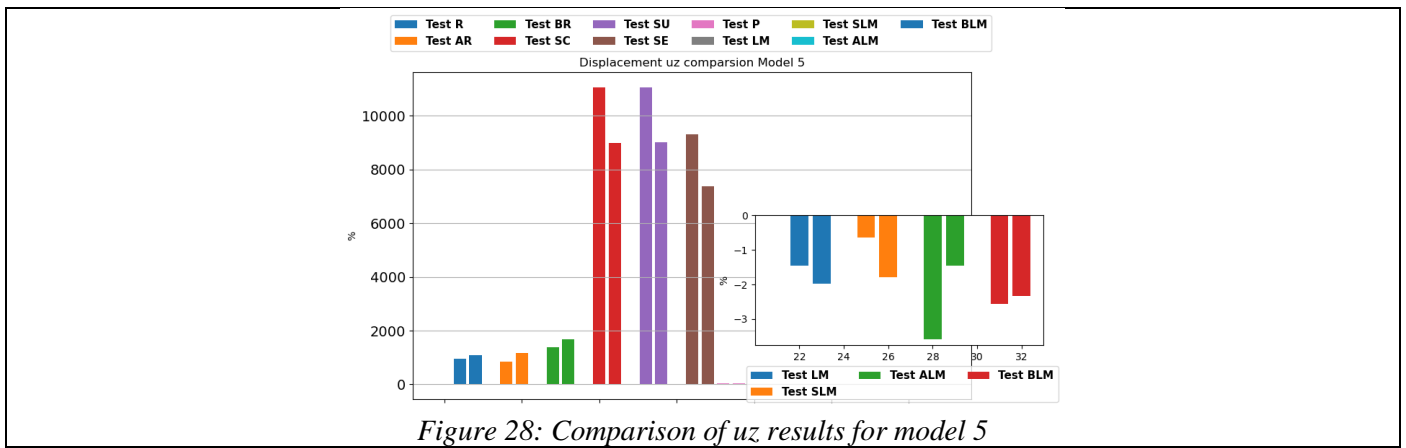


Figure 28: Comparison of uz results for model 5

c) Bending moment mxx results per model

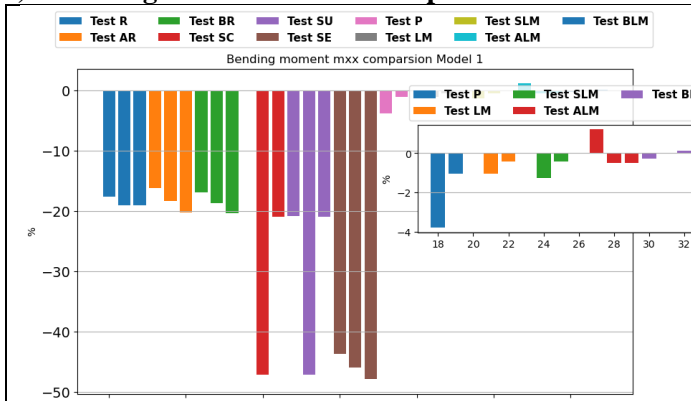


Figure 29: Comparison of mxx results for model 1

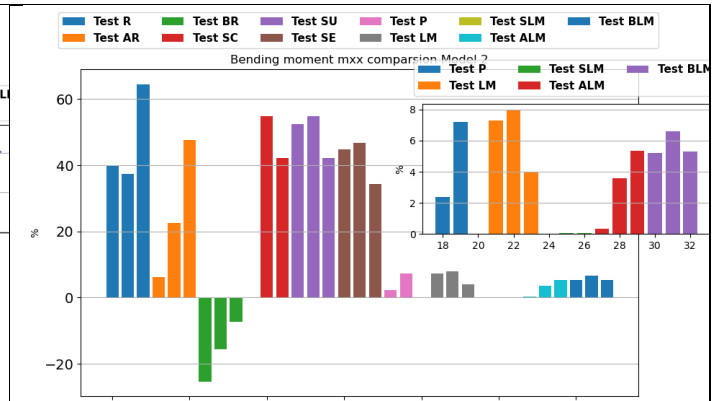


Figure 30: Comparison of mxx results for model 2

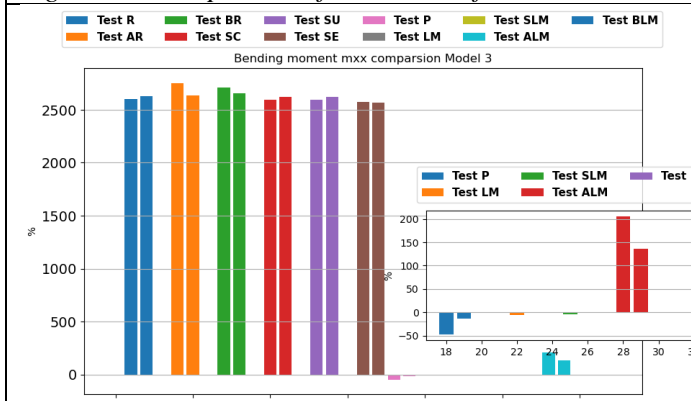


Figure 31: Comparison of mxx results for model 3

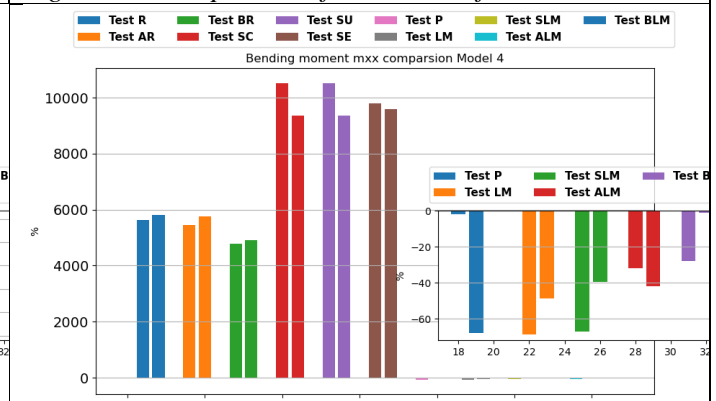


Figure 32: Comparison of mxx results for model 4

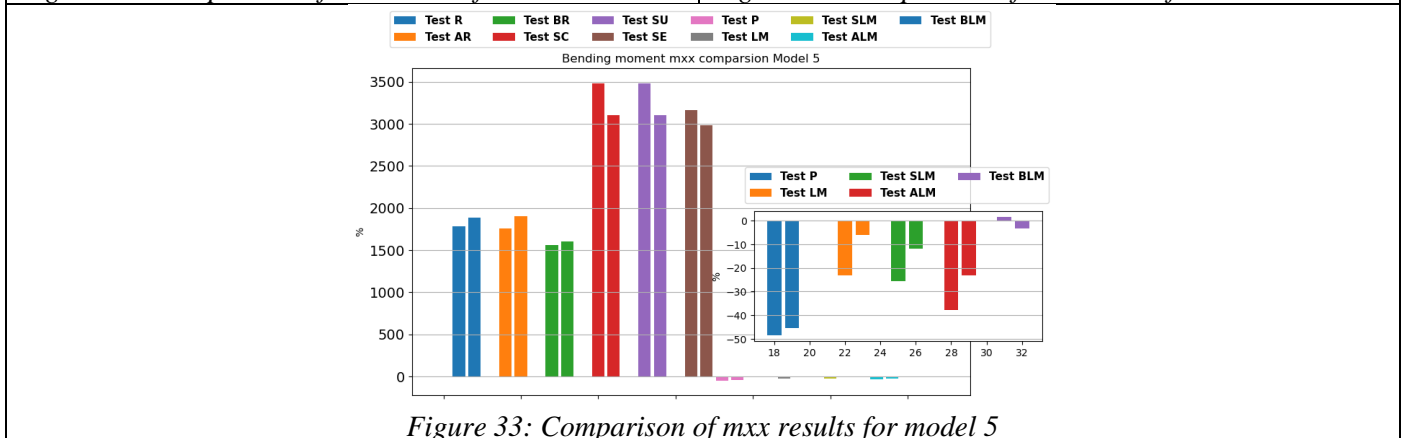


Figure 33: Comparison of mxx results for model 5

d) Shear force vx test result per model

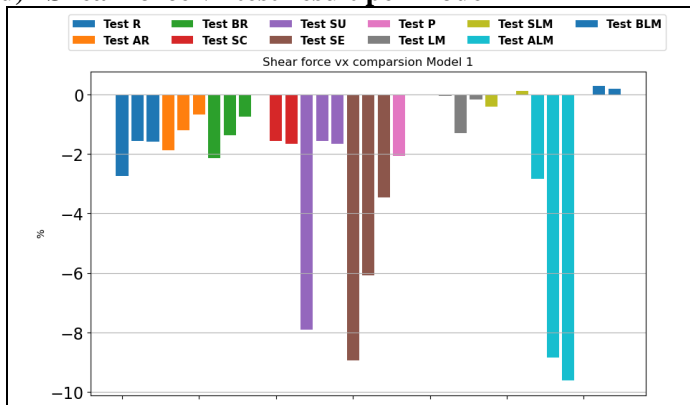


Figure 34: Comparison of vx results for model 1

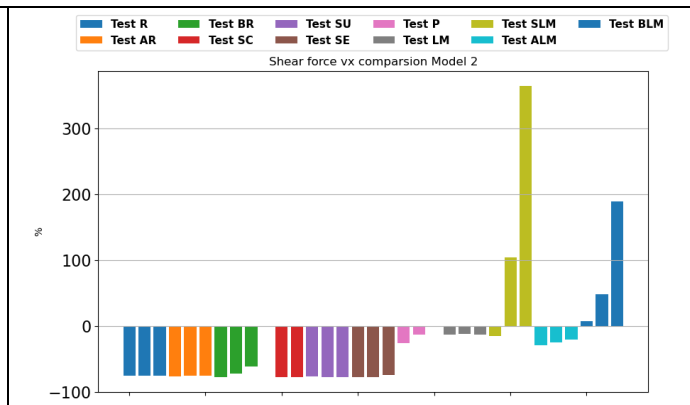


Figure 35: Comparison of vx results for model 2

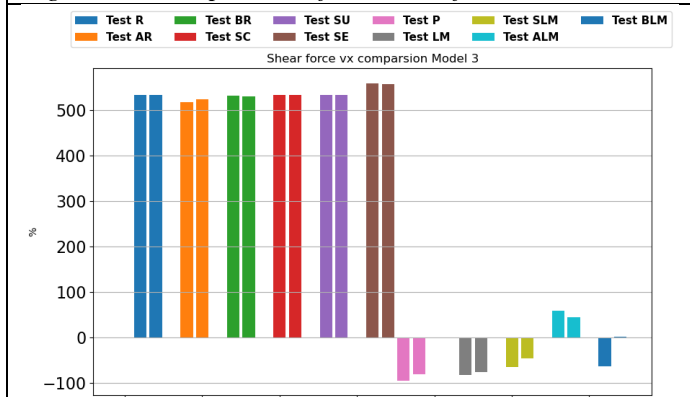


Figure 36: Comparison of vx results for model 3

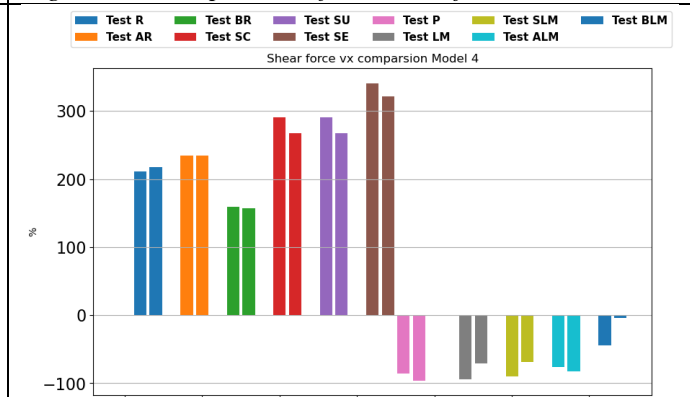


Figure 37: Comparison of vx results for model 4

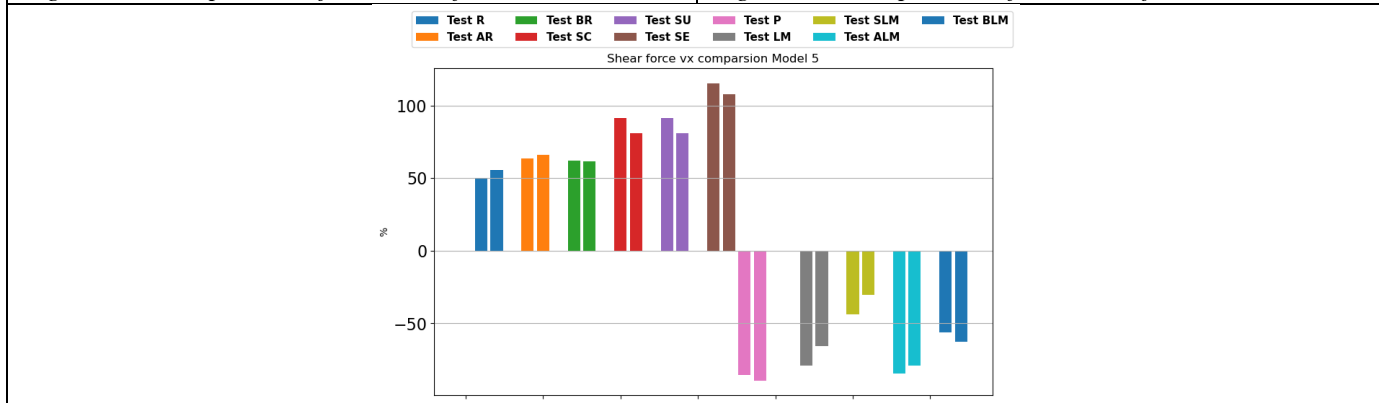


Figure 38: Comparison of vx results for model 5

4.7 Comparison between Rectangular matrix test and Square matrix test results

The results of tests using *lsmr* solver (Test R1-5, Test SU1-5, Test SC1-5 and Test SE1-5) are reorganized and listed in below table. As shown in below table, different methods of constructing square matrix and number of nodes did not play a significant role on affecting the accuracy of results. Especially for shear force and bending moment results, many of them remain unchanged regardless of type of matrix or number of nodes.

Table 41: Comparison between *lsmr* solver results

	Node number	Test	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	R	-30.0%	-14.6%			
		SU	-34.1%	19.1%			
		SC					
		SE	-31.3%	3.5%			
	30*30	R	-28.4%	-3.5%	12.12	20.01	9.42
		SU	-38.1%	6.8%	12.86	218.05	110.67
		SC	-38.1%	-94.0%	12.82	176.32	110.46
		SE	-33.4%	-12.1%	15.29	182.79	93.00
	50*50	R	-33.4%	6.1%	15.96	176.45	110.46
		SU	-37.5%	-35.7%	14.24	177.34	90.23
		SC	-37.5%	-43.6%	14.35	217.16	89.92
		SE	-34.4%	-40.6%	16.30	144.61	73.79
Bending moment (kNm/m)	20*20	R	-45.4%	52.4%			
		SU	-45.4%	52.4%			
		SC					
		SE	-43.7%	44.9%			
	30*30	R	-47.1%	54.9%	26.25	105.23	34.83
		SU	-47.3%	54.9%	25.93	105.23	34.83
		SC	-47.3%	55.1%	25.92	105.23	34.83
		SE	-46.0%	46.7%	25.75	98.06	31.63
	50*50	R	-48.5%	42.2%	25.93	93.57	31.02
		SU	-48.5%	42.2%	26.24	93.57	31.02
		SC	-48.5%	42.2%	26.24	93.57	31.02
		SE	-47.9%	34.2%	25.69	95.87	29.81
Shear force (kN)	20*20	R	-2.7%	-76.9%			
		SU	-2.7%	-76.9%			
		SC					
		SE	-8.9%	-77.6%			
	30*30	R	-1.6%	-77.3%	5.33	2.08	91.24%
		SU	-1.6%	-77.3%	5.33	2.08	91.24%
		SC	-1.6%	-77.3%	5.33	2.08	91.24%
		SE	-6.1%	-77.8%	5.59	2.48	1.15
	50*50	R	-0.8%	-77.9%	5.33	1.90	80.81%
		SU	-0.8%	-77.9%	5.33	1.90	80.8%
		SC	-0.8%	-77.9%	5.33	1.90	80.8%
		SE	-3.5%	-74.6%	5.56	2.33	1.08

4.8 Comparison between pinv solver and *lm.fit.sparse* solver test results

The results of tests using pinv solver and *lm.fit.sparse* solver (Test P1-5, Test LM1-5 and Test SLM1-5) are reorganized and listed in below table. For model 1, *pinv* solver and *lm.fit.sparse* solver can both provide excellent results where the most deviation is around 1%. For model 2, both solver are bad in terms of providing shear force results. Only *lm.fit.sparse* solver can provide accurate displacement results. For rest of models, the both solvers cannot provide accurate results in the most cases except displacement results of model 4 and model 5 and bending moment results of model 3.

Table 42: Comparison between pinv solver and *lm.fit.sparse* solver results

	Node number	Test	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	10*10	P	-0.4%	-98.98%	-32.45%	30.64%	34.58%
	20*20	P	-0.8%	-98.99%	-14.88%	-0.71%	47.94%
		LM	-0.7%	1.13%			
		SLM	-0.7%	0.00%			
	30*30	LM	-0.4%	0.87%	-14.26%	-0.60%	-1.45%
		SLM	-0.3%	0.00%	-14.21%	-0.47%	-0.65%
	50*50	LM	-0.1%	0.61%	-11.98%	-0.96%	-1.98%
		SLM	-0.2%	0.00%	-11.99%	-0.03%	-1.80%
Bending moment (kNm/m)	10*10	P	-3.8%	2.37%	-47.49%	-1.93%	-48.49%
	20*20	P	-1.0%	7.18%	-13.91%	-67.93%	-45.43%
		LM	-1.0%	7.27%			
		SLM	-1.3%	0.02%			
	30*30	LM	-0.4%	7.95%	-5.15%	-68.46%	-23.27%
		SLM	-0.4%	0.04%	-4.92%	-66.91%	-25.59%
	50*50	LM	0.0%	3.94%	-1.73%	-48.85%	-6.17%
		SLM	0.0%	0.04%	-1.61%	-39.58%	-11.95%
Shear force (kN)	10*10	P	-2.1%	-26.20%	-94.69%	-86.12%	-85.63%
	20*20	P	0.0%	-13.35%	-80.86%	-95.84%	-89.31%
		LM	-0.1%	-13.28%			
		SLM	-0.4%	-15.27%			
	30*30	LM	-1.3%	-12.32%	-83.17%	-94.11%	-78.99%
		SLM	0.0%	1.04	-65.06%	-89.62%	-43.73%
	50*50	LM	-0.2%	-13.20%	-76.14%	-71.05%	-65.81%
		SLM	0.1%	3.65	-46.59%	-68.61%	-30.25%

4.9 Comparison between five-point difference approximation and two-point difference approximation test results

The results of tests using five-point difference approximation (Test AR1-5, Test ALM1-5, Test BR1-5, and Test BLM1-5) and two-point difference approximation (Test R1-5 and Test LM1-5,) are reorganized and listed in below table. A few bending results provided by *lsmr* solver were improved where their deviation was dramatically reduced (Test AR2 & Test BR2). In the contrast, under *lm.fit.sparse* solver some bending moment results have a larger deviation (Test ALM3 & Test BLM2). For rest of results, the expected improvement on the accuracy by five-point approximation are not clear. There is no significant impact can be found for the most results.

Table 43: Comparison between five-point and two-point difference approximation results (*lsmr* solver)

	Number of nodes	Test	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	R	-30.0%	-14.6%			
		AR	-29.2%	-47.8%			

	30*30	BR	-29.8%	-81.9%				
		R	-28.4%	-3.5%	12.12	20.01	9.42	
		AR	-33.0%	-28.5%	14.91	17.52	8.41	
	50*50	BR	-33.3%	-81.7%	11.98	27.52	13.92	
		R	-33.4%	6.1%	15.96	176.45	110.46	
		AR	-36.2%	-21.9%	13.07	22.52	11.62	
	Bending moment (kNm/m)	20*20	BR	-36.4%	-82.4%	12.90	33.40	16.82
			R	-45.4%	52.4%			
			AR	-16.2%	6.2%			
30*30		BR	-16.9%	-25.3%				
		R	-47.1%	54.9%	26.25	105.23	34.83	
		AR	-18.3%	22.5%	27.53	54.57	17.56	
50*50		BR	-18.7%	-15.6%	27.09	47.86	15.60	
		R	-48.5%	42.2%	25.93	93.57	31.02	
		AR	-20.2%	47.6%	26.35	57.68	19.07	
Shear force (kN)	20*20	BR	-20.4%	-7.4%	26.58	49.13	16.01	
		R	-2.7%	-76.9%				
		AR	-1.9%	-76.7%				
	30*30	BR	-2.1%	-77.2%				
		R	-1.6%	-77.3%	5.33	2.08	91.24%	
		AR	-1.2%	-75.6%	5.17	2.34	63.56%	
	50*50	BR	-1.4%	-72.1%	5.31	1.59	62.11%	
		R	-0.8%	-77.9%	5.33	1.90	80.81%	
		AR	-0.7%	-75.3%	5.24	2.35	65.76%	
		BR	-0.8%	-61.2%	5.30	1.58	61.42%	

Table 40: Comparison between five-point and two-point difference approximation results (lm.fit.sparse solver)

	Number of nodes	Test	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	LM	-0.7%	1.1%			
		ALM	1.5%	0.4%			
		BLM	-0.6%	0.3%			
	30*30	LM	-0.1%	0.6%	-11.98%	-0.96%	-1.98%
		ALM	-1.2%	0.3%	-27.04%	4.49%	-3.58%
		BLM	-0.3%	0.3%	-11.83%	-1.18%	-2.56%
	50*50	LM	-0.1%	0.6%	-11.98%	-0.96%	-1.98%
		ALM	-1.1%	0.5%	-20.63%	0.12%	-1.45%
		BLM	-0.1%	0.1%	-10.84%	-1.00%	-2.33%
Bending moment (kNm/m)	20*20	LM	-1.0%	7.3%			
		ALM	1.2%	0.3%			
		BLM	-0.3%	5.2%			
	30*30	LM	-0.4%	7.9%	-5.15%	-68.46%	-23.27%
		ALM	-0.5%	3.6%	2.05	-31.74%	-37.74%
		BLM	0.0%	6.6%	-1.28%	-27.83%	1.46%
	50*50	LM	0.0%	3.9%	-1.73%	-48.85%	-6.17%
		ALM	-0.5%	5.3%	1.36	-42.00%	-23.27%
		BLM	0.1%	5.3%	-0.09%	-1.15%	-3.29%
Shear force (kN)	20*20	LM	-0.1%	-13.3%			
		ALM	1.2%	0.3%			
		BLM	0.0%	7.5%			
	30*30	LM	-1.3%	-12.3%	-82.99%	-59.66%	-83.56%
		ALM	-8.9%	-24.4%	57.79%	-76.01%	-84.36%
		BLM	0.3%	48.7%	-64.11%	-44.83%	-56.05%
	50*50	LM	-0.2%	-13.2%	-76.14%	-71.05%	-65.81%
		ALM	-9.6%	-20.5%	44.36%	-82.74%	-78.99%
		BLM	0.2%	1.89	1.13%	-4.55%	-62.93%

Discussion

This section aims to discuss whether the objective of this project is fulfilled by analyzing the obtained results. First, the accuracy, reliability, and efficiency of shell code are validated by reviewing the results and comparison. The sparsity, condition number, and rank number are calculated for checking the quality of generated matrices. Then the factors affecting shell code results and the mechanism behind them are also shown. There are five possible factors affecting shell code results have been discussed, include the number of nodes, type of matrix, type of solvers, difference approximation methods, number of iterations. The simplicity of code structure is also discussed by showing the coding process of how to simply apply the new shell theory equation. In the end, the final discussion is made on the possible improvement and further research.

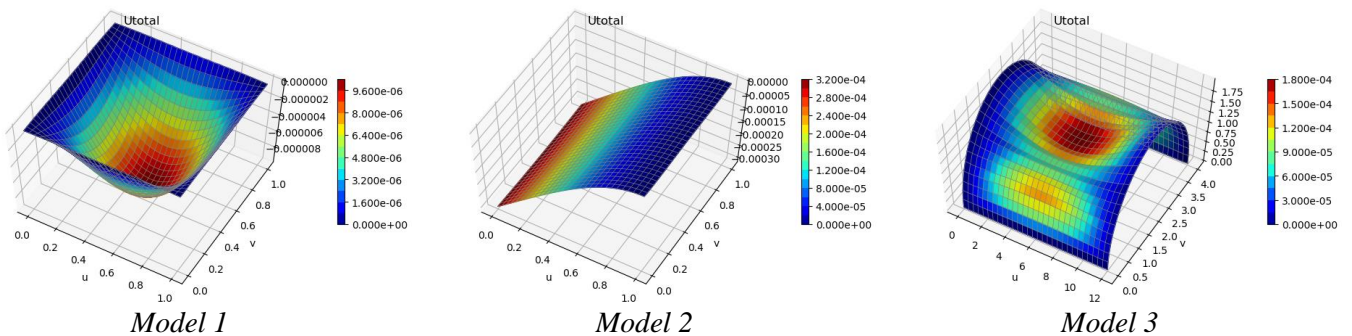
5.1 Accuracy, reliability, and efficiency

a) Accuracy

The shell code results should bear the three characteristics: accuracy, reliability, and efficiency. Accuracy means compared to the general finite element results the local and global deviation of shell code results should be in a reasonable range. Specifically, it requires shell code not only can produce a good estimation of extreme values in a local area but also can correctly describe the overall behaviour of the tested models globally. So that discussion on the accuracy is not only on the deviation of numerical values, but also on the distortion of contour lines, and edges results.

As mentioned in section 4.6, the overall comparison clearly indicated that the deviation of shell code results is large for most cases. Only for model 1&2 when using *lm.fit.sparse* solver, the shell code can produce results with a small deviation less than 15%. Especially the displacement results, in this case, are generally equal to finite element results (deviation is less than 1%). For model 3-5 when using *lm.fit.sparse* solver, the displacement results have a small deviation less than 15%. However, for bending moment and shear force results, the deviation of extreme values is up to 90%. In some cases, using *lsmr* solver, the maximum deviation can even reach 22000%.

To better illustrate the shell code displacement results, actual deformed shapes of models are generated based on the displacement results (u_x, u_y, u_z). The displacement results from Test LM1-5 are used to add to the undeformed geometry shapes to show actual deformed shape. The color map of shapes is determined by the value of total displacement u_{xyz} . From the general perspective of observing deformation, the shell code and finite element software give quite similar results in terms of extreme values and general deformed shape. The actual deformed shapes by shell code results share some important key features with the finite element results (Figure 40). For example, the boundary behaviors under various definitions including fixed edges, pin edges and free edges are correctly described. The concentration of displacement at free nodes and distortion of mesh grid are also shown in both shell code and finite element results.



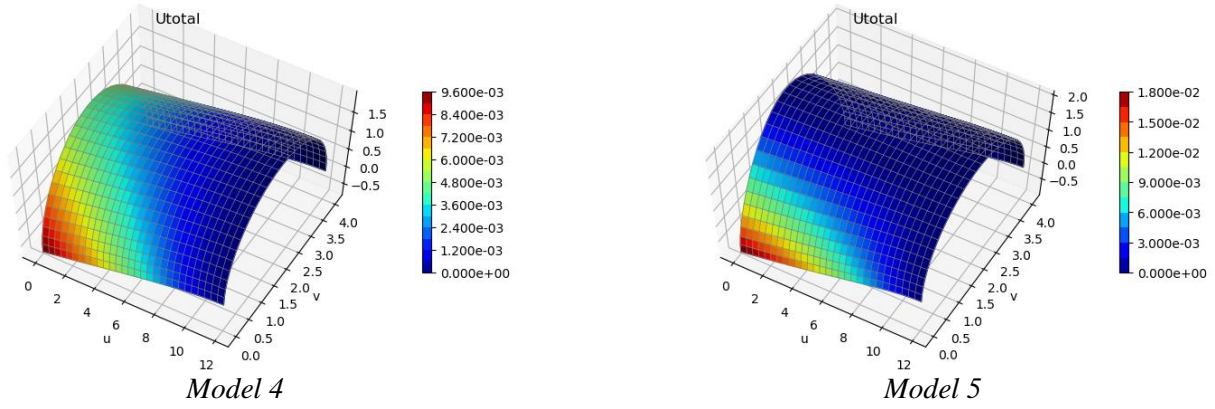


Figure 39: Actual deformed shape of models by shell code results (Test LM1-5)

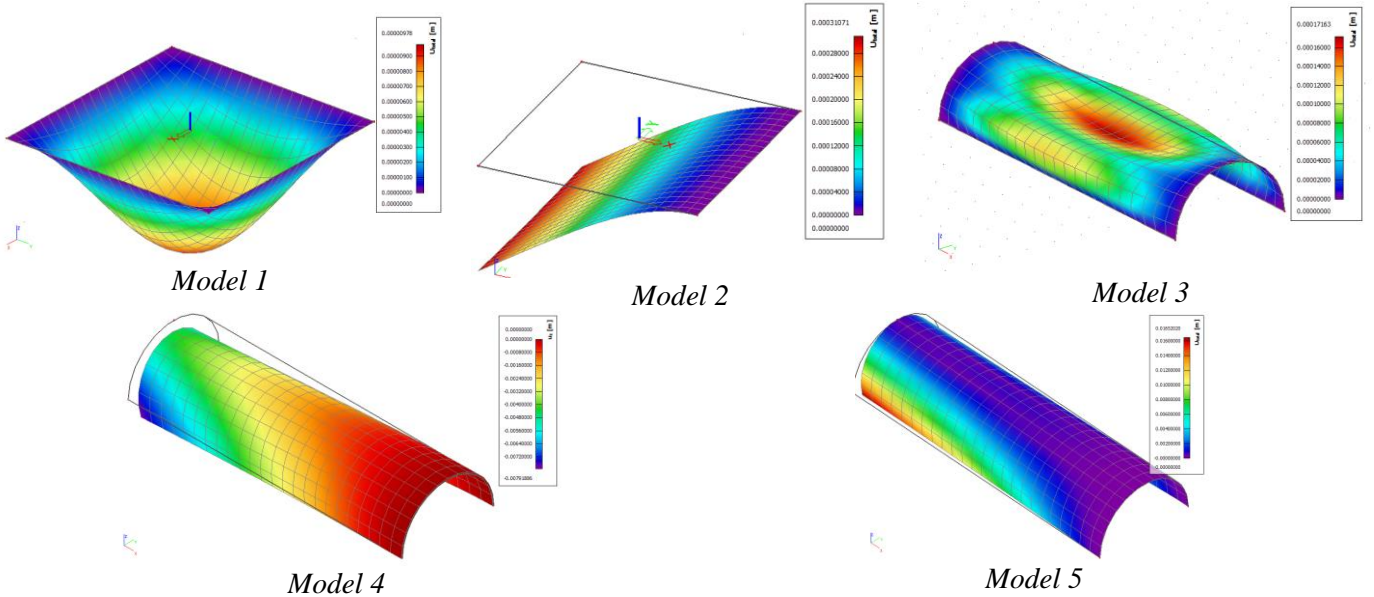


Figure 40: Actual deformed shape of models by finite element solution (SCIA Engineer 19)

The distortion of contour lines is common among the results when using *lsmr* solver (Test R1-5, Test SC1-5, Test SU1-5, and Test SE1-5). At the boundaries of those models, large displacement existed at corners and edges where the predefined boundary conditions do not allow. As shown in those plots the displacement contour lines are usually jagged alongside the free edges of model. For better illustration of that, the u_z plots from Test R1-5 are reconstructed into 3D surfaces with projection in horizontal directions (Figure 49) as a representative of them. The results of other tests might have a few differences in the extreme values but they share similar distorted boundary behavior. From those horizontal projections, it is clearly be observed that larger displacement existed at pinned and fixed edges where no displacement should appear. Those edges lines are distorted into the shape of sine waves. The crests of sine waves are usually local at mid of edge lines. At the corners of model 1, the displacement is even below the permitted edge lines. And for the free edges of model 4 and model 5, the edge line should be in sine wave shape instead of the current flat line. The above features can be more obvious in the comparison of the results of Test LM1-5 (Figure 49). The projections of u_z plot from Test LM 1-5 show that the defined boundary conditions are clearly and correctly expressed at edges when using *lm.fit.sparse* solver. And the *lsmr* solver did not achieve this.

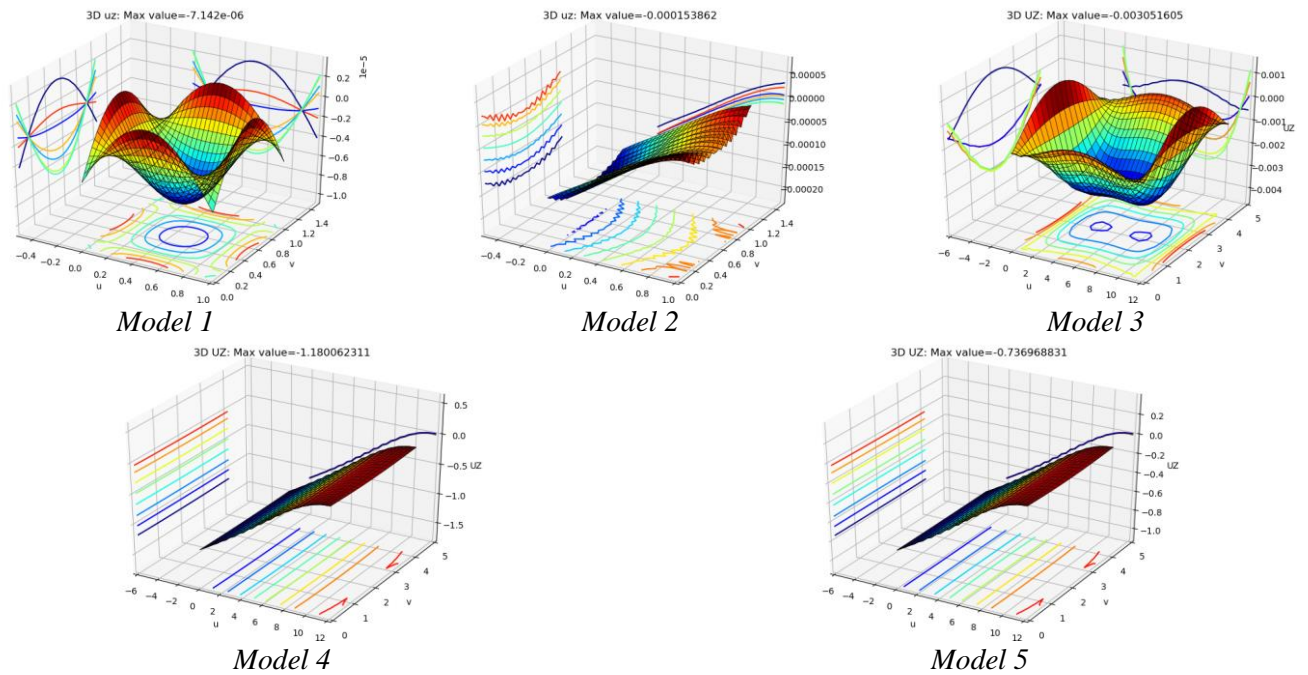


Figure 49: 3D surface and projections of uz plots from Test R1-5

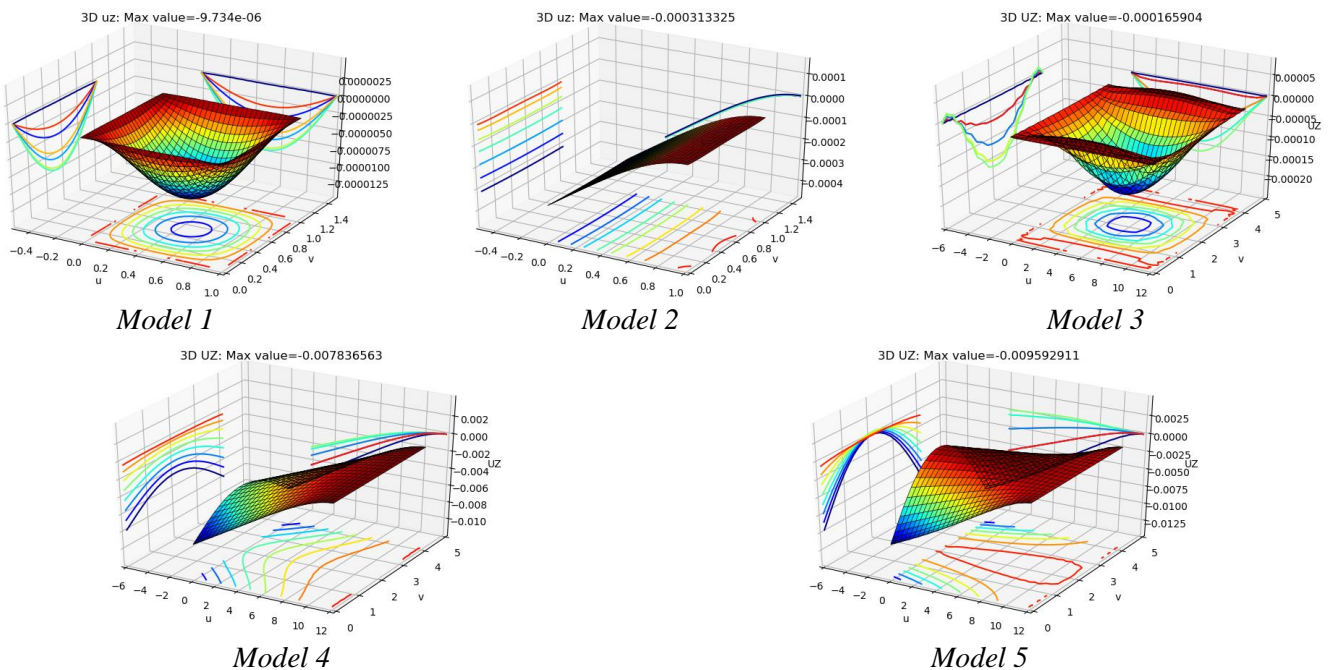


Figure 42: 3D surface and projections of uz plots from Test LM1-5

To better illustrate the edge results of bending moment and shear force, bending moment diagram and shear force diagram is plot alongside the edges of models (Figure 43, Figure 45). The corresponding finite element edge results are also shown (Figure 44, Figure 46). The extreme values from those plots are collected in below:

Table 45: Comparison between bending moment m_{xx} edge results

		Model 1	Model 2	Model 3	Model 4	Model 5	
Bending moment m_{xx} edge results (kNm/m)	(Max)	Shell code	0.0034	5.519	0.00396	2.59	8.927
		Finite element solution	0.0007	6.311	0.0459	5.044	9.531
	(Min)	Shell code	-0.00587	-0.0178	-0.0012	-0.221	-0.642
		Finite element solution	-0.00027	-0.08152	-0.00224	-1.237	-0.6037

Table 46: Comparison between shear force vx edge results

			Model 1	Model 2	Model 3	Model 4	Model 5
Shear force vx edge results (kN/m)	(Max)	Shell code	3.3631	15.905	0.571	8.886	13.311
		Finite element solution	3.376	274.24	2.328	30.297	30.911
	(Min)	Shell code	-3.3631	-11.409	-0.571	-0.594	-1.704
		Finite element solution	-3.376	-17.915	-2.328	-23.351	-4.17

Table 47: Deviation of Test LM1-5 overall bending moment & shear force results (m=n=50)

	Model 1	Model 2	Model 3	Model 4	Model 5
Bending moment	0.00%	3.94%	-1.61%	-20.53%	-19.54%
Shear force	-0.18%	-13.20%	-75.85%	-92.02%	-75.11%

The above comparison indicates that the shell code results basically do not comply with the finite element results in terms of extreme values except the shear force edge results of model 1. Overall considered, the shell code has underestimated the edge results numerically. For instance, the shell code gives the maximum shear force at the edges of model 2 as 15.9 kN/m which is dramatically lower than finite element results of 274.24 kN/m. Such underestimation also occurred for the entire models as shown in Table 43 but to a smaller extent. Meanwhile, the shell code bending moment and shear force results are more likely concentrated at corners. For example, the finite element results are generally smoothly varied alongside edges except for bending moment results of model 1 and shear force results of model 2. The concentration at corners is almost shown on every model in the shell code results. And also the shell code edge results is jagged for most models.

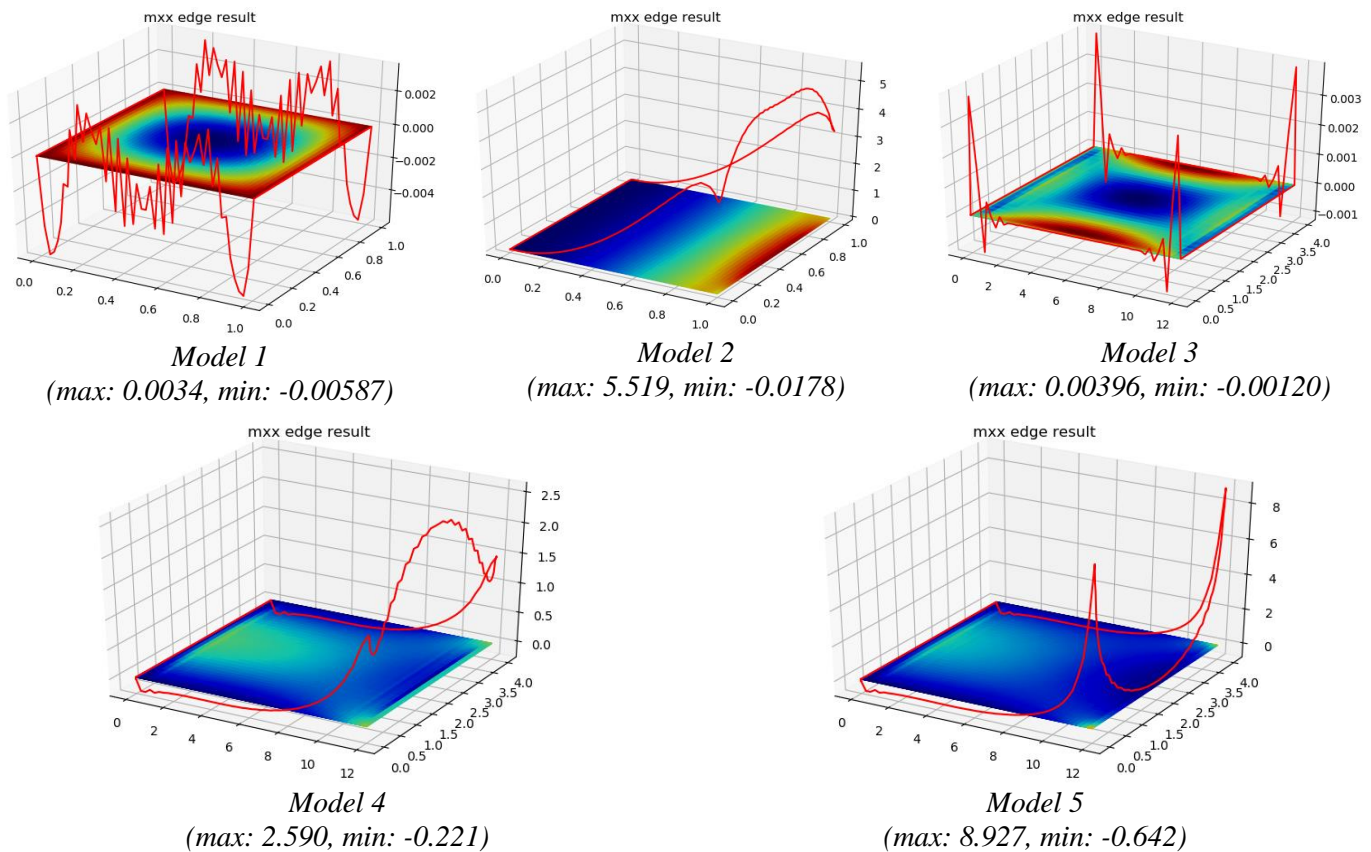


Figure 43: Bending moment m_{xx} edge results by shell code (Test LM1-5, $m=n=50$)

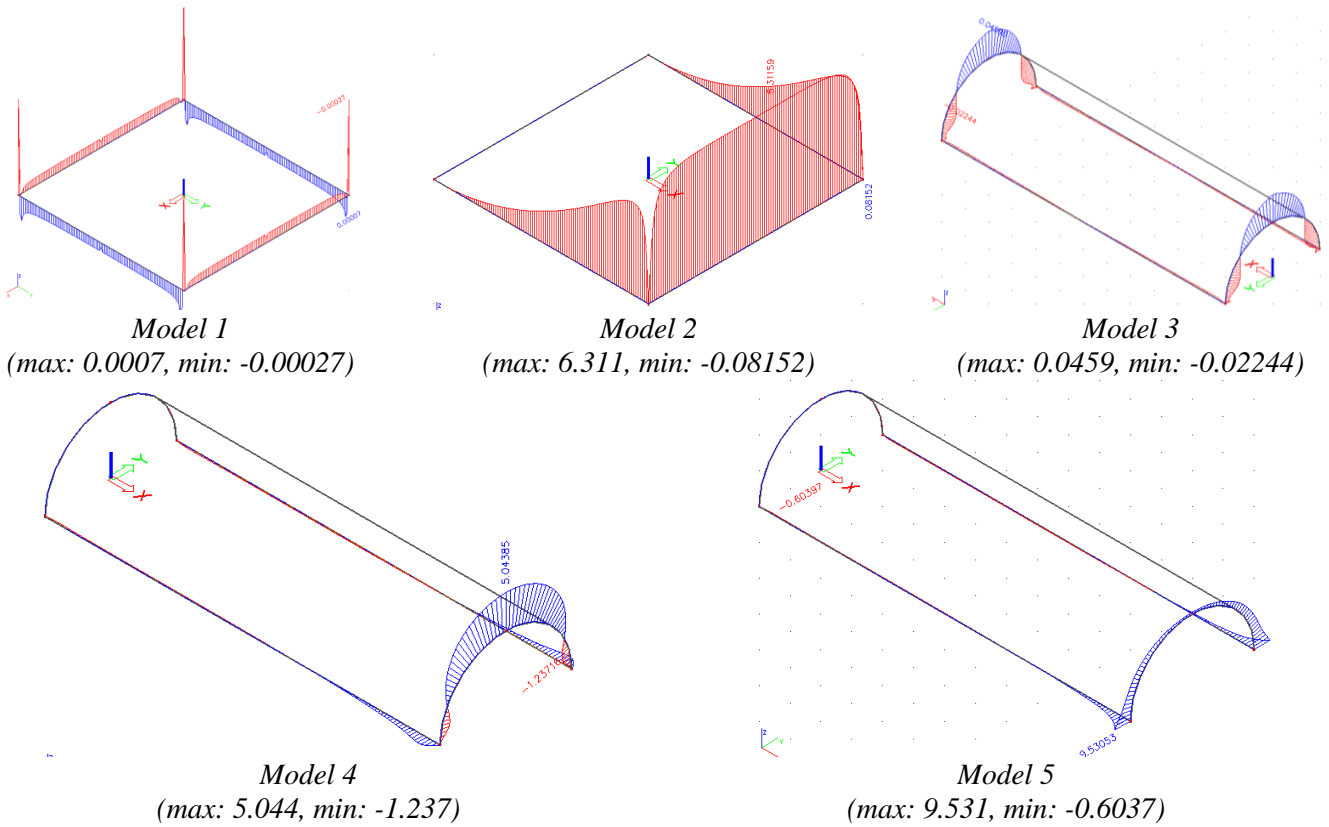


Figure 44: Bending moment m_{xx} edge results by finite element solution (SCIA Engineer 19)

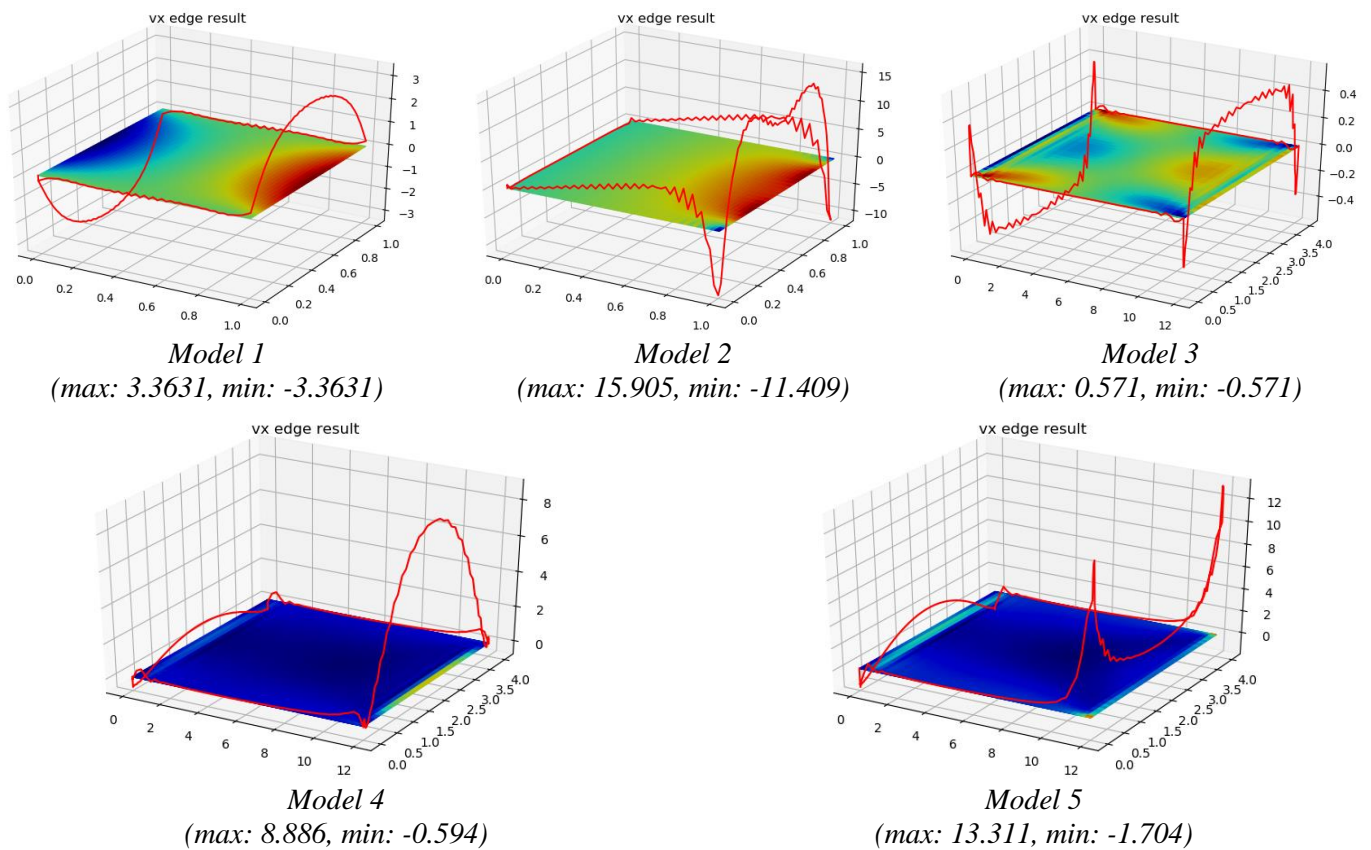


Figure 45: Shear force v_x edge results by shell code (Test LM1-5, $m=n=50$)

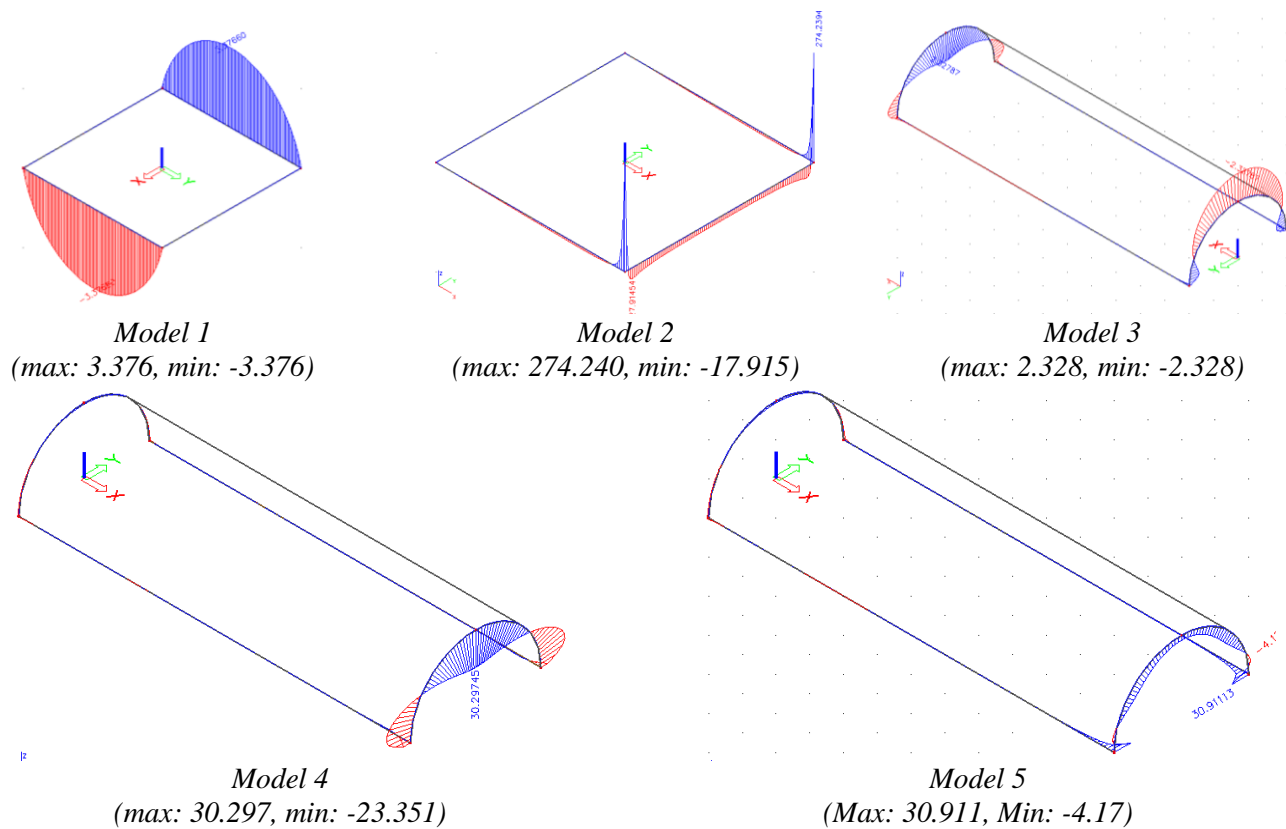
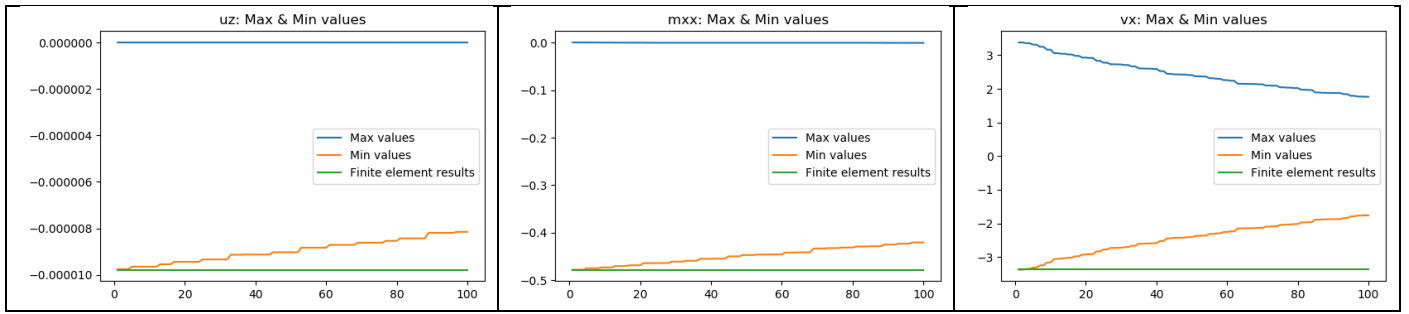


Figure 46: Shear force vx edge results by finite element solution (SCIA Engineer 19)

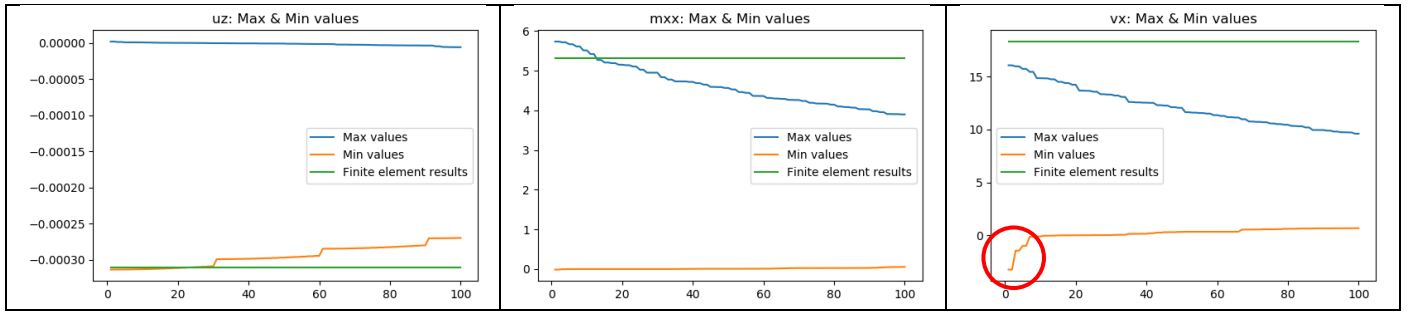
In conclusion, the shell code can only produce accurate displacement results (deviation is less than 15%) when using *lm.fit.sparse* solver. The distortion of contour lines is common among the results when using *lsmr* solver. In terms of edge results, the most of them is incorrect as they are significantly underestimated.

b) Reliability

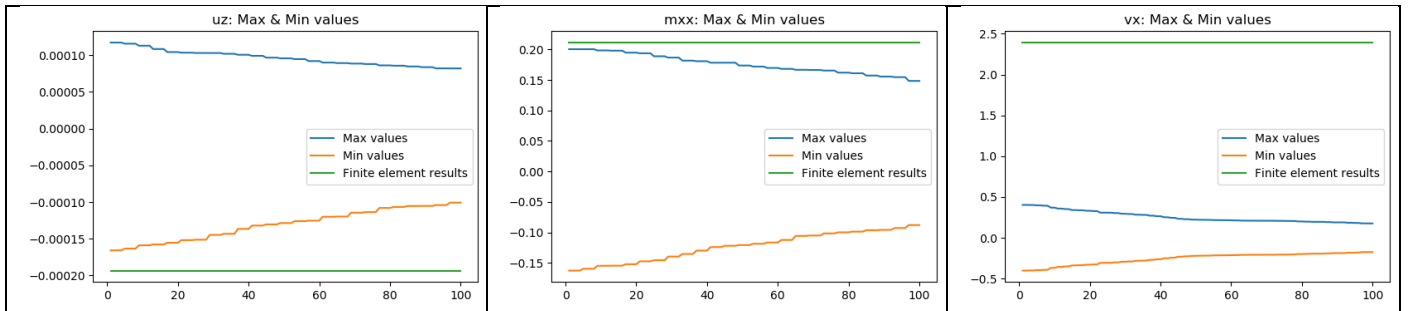
The reliability means obtained results are examined through various methods to avoid possible numerical error as much as possible. For example, a singularity check has been made to ensure those extreme values were not collected at those unrealistic peaks. In finite element models, it is possible that a few extreme values of finite difference models tend toward an infinite value as the result of a potential computational error. Such a phenomenon is also likely to occur in finite difference models. The singularities should be excluded from normal extreme values otherwise they might compromise the validity of the comparison of extreme values in the above section. To do that, the top 100 maximum values and minimum values from Test LM1-5 and Test R1-5 are collected and plot in below (Figure 47 & Figure 48) to show whether there is any spike in the distribution of extreme values. The absolute maximum values from finite element results of Table 18 are also plot as a comparison. Generally speak, there is no such spike showing extreme values tend toward an infinite value. Among those plots, there are three major spikes in terms of varying extreme values: shear force results of Test LM2, bending moment results of Test LM4 and bending moment results of Test R2. Compare to the overall variation of extreme values among the 100 nodes, those spikes indicate a rapid variation in values for a small number of nodes. However, the extent of such variation is far from the trend toward an infinite value. In the summary, there is no singularity has been found in those results.



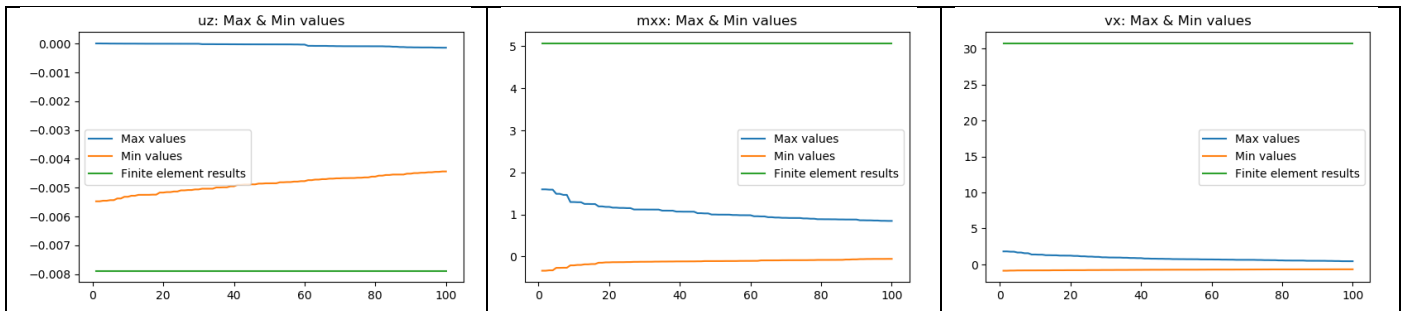
a) Test LM1



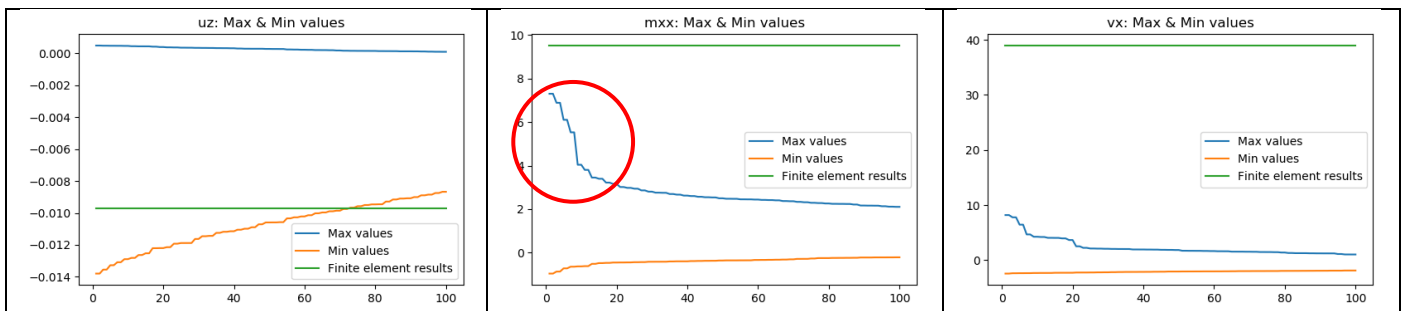
b) Test LM2



c) Test LM3

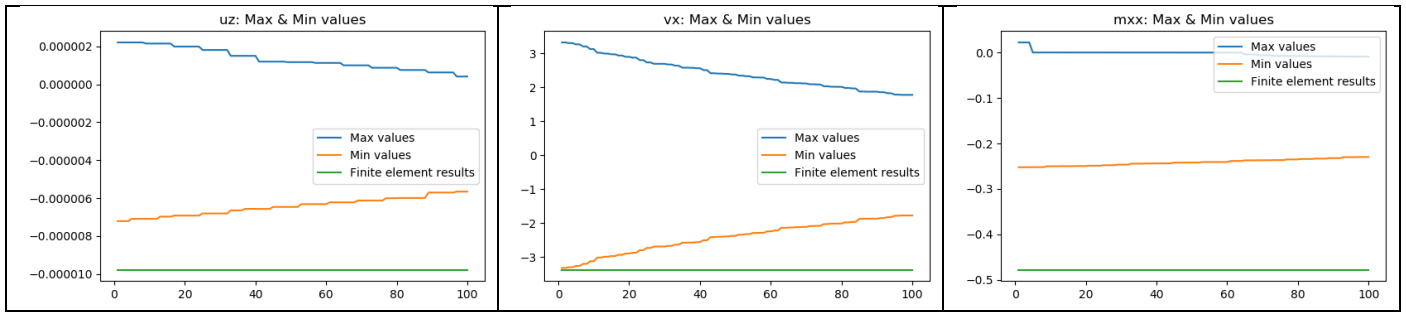


d) Test LM4

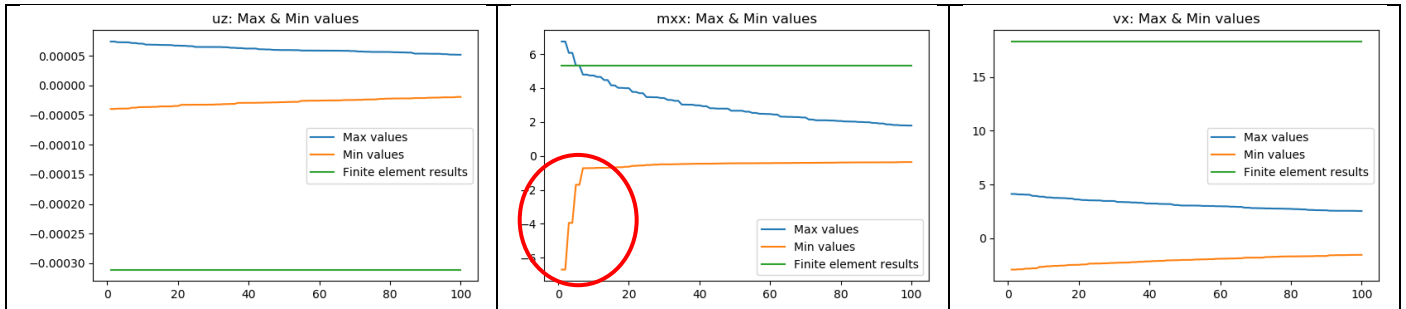


e) Test LM5

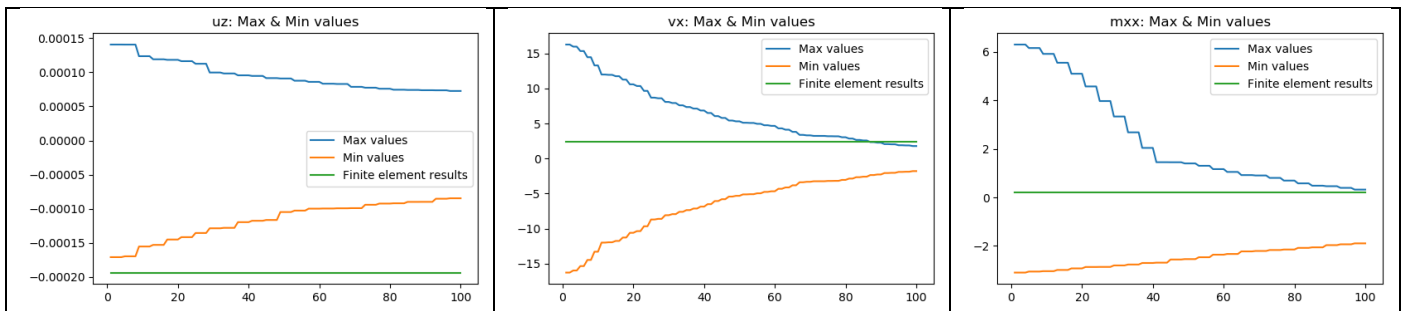
Figure 47: Plots of top 100 maximum and minimum values from Test LM1-5 ($m=n=30$)



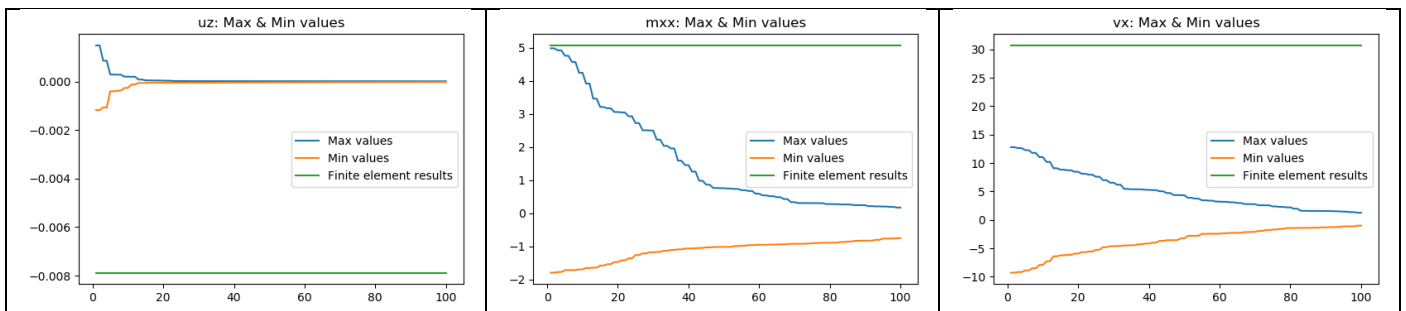
a) Test R1



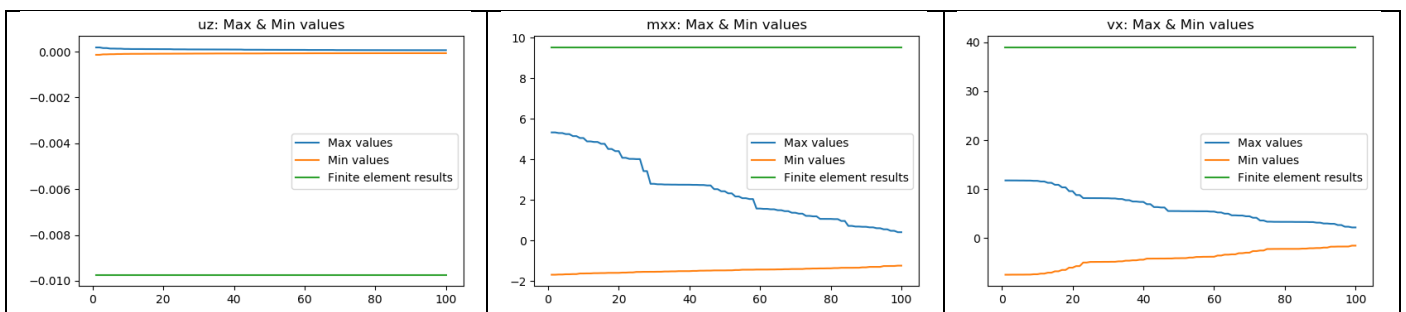
b) Test R2



c) Test R3



d) Test R4



e) Test R5

Figure 48: Plots of top 100 maximum and minimum values from Test R1-5 ($m=n=30$)

c) Efficiency

The efficiency means shell code can produce results in a fast and economic way. It means the time spending and memory usage of shell code is competitive when compared to some popular commercial finite element software. Here SCIA Engineer 19 is selected to solve those model problems. The model data used in SCIA is the same as the shell code model configuration (Table 9). The computational time and memory usage of computation activity of SCIA was recorded as below. Since lack of a direct method for measuring the running time of SCIA, the time was measured in approximated way manually. It is noticeable that the computational time and memory usage of this finite element software is only related to the number of nodes.

Table 48: Memory usage and time by finite element software (SCIA Engineer 19)

Number of nodes	20*20		30*30		50*50	
	Memory usage	Time	Memory usage	Time	Memory usage	Time
Model 1-5	231MB	1s	241MB	4s	274MB	6s

The memory usage of shell code and spent time at the solving step were recorded and listed below. overall considered, the shell code uses less memory but more time. The spent time of shell code is significantly higher than that of SCIA and it increases dramatically as the number of nodes increases. The highest memory usage of shell code is only nearly half of finite element software. The memory usage of shell code with *lsmr* solver is increased by 23% when as the number of nodes increased from 20*20 to 50*50. And there is no significant difference appeared in memory usage at the solving step when a different input matrix is used. The memory usage at the solving step seems only related to number of nodes regardless of type of matrices and solvers. However, the spent time is significantly higher as the input matrix of the square sparse matrix is used in most scenarios. The input square matrices generated by the undefined node method and central node method lead to generally higher time spending for all five models (Test SU, SC). On the contrary, the results of Test SE and Test SLM indicate that spent time at the solving step is sensitive in a model-oriented way. When test models are flat plate square (Test SE1, 2 & Test SLM1, 2), the spent time can be only around 25% of that of canopy models (Test SE3-5 & Test SLM3-5).

Table 49: Memory usage and time by different solver in shell code

Solver type	Number of nodes Type of input matrix [M]	20*20		30*30		50*50	
		Memory usage	Time	Memory usage	Time	Memory usage	Time
Python solver: <i>lsmr</i>	Rectangular sparse matrix	98.8672 MB	0.897s	104.0859 MB	2.154s	120.9961 MB	10.105s
	Square sparse matrix (Test SU, SC)	98.8008 MB	5.046s	103.5820 MB	11.410s	120.6719 MB	72.991s
	Square sparse matrix (Test SE1, 2)	97.5430 MB	3.733s	102.4844 MB	8.115s	118.7852 MB	22.093s
	Square sparse matrix (Test SE3)	/	/	102.6445 MB	17.1068s	119.7070 MB	31.699s
	Square sparse matrix (Test SE4)	/	/	102.8359 MB	10.130s	119.6328 MB	89.229s
	Square sparse matrix (Test SE5)	/	/	102.8359 MB	13.279s	119.7578 MB	97.949s
R solver: <i>lm.fit.sparse</i>	Rectangular sparse matrix	147.3MB	1.63s	148.9MB	4.19s	153.9MB	24.68s
	Square sparse matrix (Test SLM 1)	111.9023 MB	4.863s	115.2969 MB	5.568s	115.3047 MB	8.106s
	Square sparse matrix (Test SLM 2)	111.8125 MB	6.365s	115.5742 MB	5.763s	141.3125 MB	8.293s
	Square sparse matrix (Test SLM 3)	/	/	114.6328 MB	8.460s	135.4336 MB	27.723s
	Square sparse matrix (Test SLM 3)	/	/	114.6328 MB	8.460s	135.4336 MB	27.723s

	Square sparse matrix (Test SLM 4)	/	/	113.6367 MB	8.502s	135.7070 MB	26.590s
	Square sparse matrix (Test SLM 5)	/	/	118.3633 MB	8.988s	130.5742 MB	28.346s
		10*10		20*20			
Python solver: pinv	Rectangular matrix in Python array form	131.375 MB	7.412s	148.2656 MB	453.638s		

5.2 Matrix quality

a) Sparsity

In order to show the quality of matrices generated by shell code, the rectangular matrices and square matrices produced by equation replacement method were checked. The first property checked is sparsity which is defined as the fraction of zero elements in a matrix. According to the calculation, the sparsity of rectangular matrices is basic same to square matrices. So that below Figure 57 only shows how the sparsity is varied with different number of nodes for each model. As the number of nodes increased, sparsity rises slowly from lowest value of 99.99675% to highest value of 99.99951%. Since the non-zero elements are generally distributed on the local diagonal lines of matrix, the number of zero elements increase in a faster way with larger matrix size. Despite the different boundary conditions of each model, the sparsity is the same for the models that use same geometry shape (Flat square shape: Model 1&2, Canopy shape: Model 3-5). As shown in below figure, the sparsity of model 1 and model 2 is generally equal. Models 3-5 also share the same sparsity.

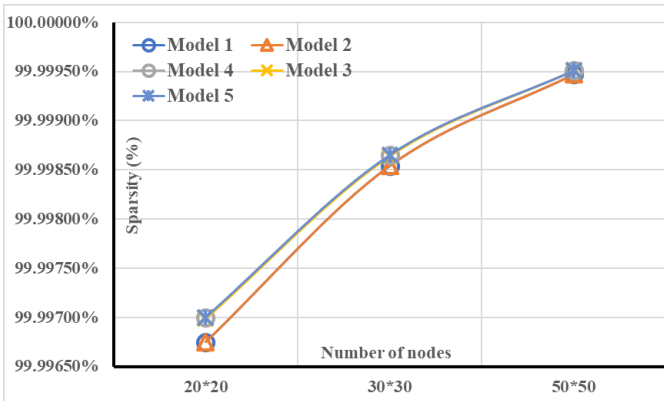


Figure 57: Sparsity of matrices

b) Condition number

The second property checked is the condition number for inversion. For matrix A , the condition number is $\kappa(A) = \|A\| \|A^{-1}\|$. For a system of linear equations $A \cdot x = b$, the condition number of matrix A can be viewed as a relative error magnification factor (Cline *et al.*, 1979). The error in solution x is $\frac{\|y-x\|}{\|x\|}$ which satisfies the below expression:

$$\frac{\|y-x\|}{\|x\|} \leq \varepsilon \|A\| \|A^{-1}\| \rightarrow \frac{\|y-x\|}{\|x\|} \leq \varepsilon \cdot \kappa(A) \quad \left((A+E)y = b, \varepsilon = \frac{\|E\|}{\|A\|} \right)$$

where ε is the relative error in A , y is the perturbed solution of the linear system, E represents the perturbation of matrix A .

The upper bound of error in solution x is determined by the condition number and relative error of matrix A . The same upper bound was also found if perturbation exists in vector b :

$$\frac{\|y-x\|}{\|x\|} \leq \varepsilon \|A\| \|A^{-1}\| \rightarrow \frac{\|y-x\|}{\|x\|} \leq \varepsilon \cdot \kappa(A) \quad \left(Ay = b + e, \varepsilon \geq \frac{\|e\|}{\|b\|} \right)$$

where ε is the relative error in b , y is the perturbed solution of the linear system, e represents the perturbation of vector b .

The derivation of above relation depends on inequality $\|b\| \leq \|A\| \|x\|$. However, when $\kappa(A)$ is large, this relation is very weak for almost all b . In summary, the upper bound of error in solution x is mainly determined by the error in matrix A and the condition number of A . A higher condition number can magnify the error in matrix A so that error in solution x is more likely higher. It is noticeable that the precondition of this conclusion is that the linear system is solved by matrix inversion. Although only part of test results was obtained through matrix inversion, the condition number can be still used to measure the how sensitive the solution is to the perturbations in matrix A .

The condition number of rectangular matrices used in Test R1-5 and square matrices Test SE1-5 was obtained and plot in below figures (Figure 58, Figure 59). Due to limited capacity of author's PC, number of nodes of models in condition number calculation was lower than actual model used in tests. The data of those figures can be found in Annex (Table 49, Table 50). For all matrices, the condition number increases with the number of nodes of the model. The condition number of model 4 is nearly equal to that of model 5 regardless of types of matrices. For rectangular matrix, condition number of models: model 1 > model 2 > model 3 > model 4 = model 5. For square matrices, condition number of models: model 3 > model 1 > model 2 > model 4 = model 5. The lowest condition number is 6.46e+15 and the highest condition number is 8.69E+16. Regardless of type of matrix, the condition number is much larger than 1. Those matrices can be said as ill-conditioned and they are almost singular. Finding the inverse of such matrices or solution of such linear systems is highly possible to end up with large numerical errors.

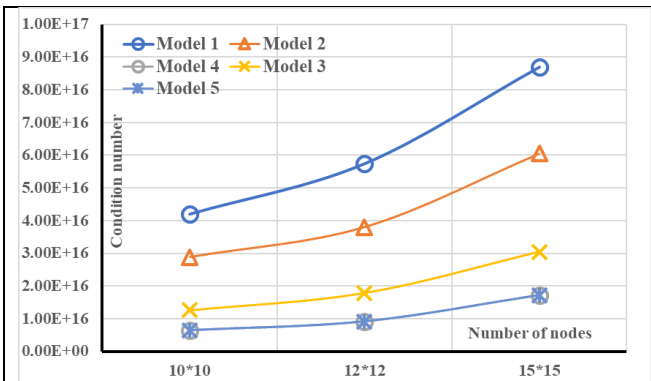


Figure 50: Condition number of rectangular matrices

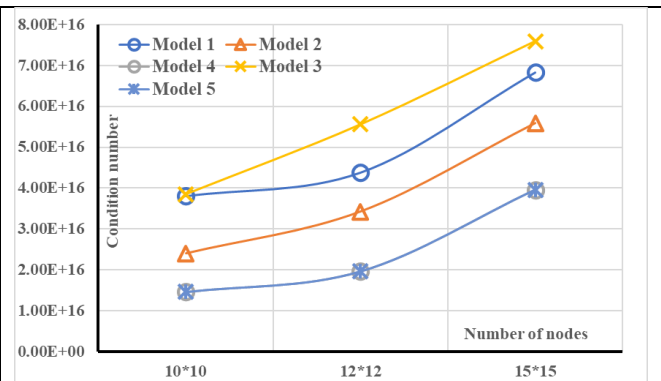
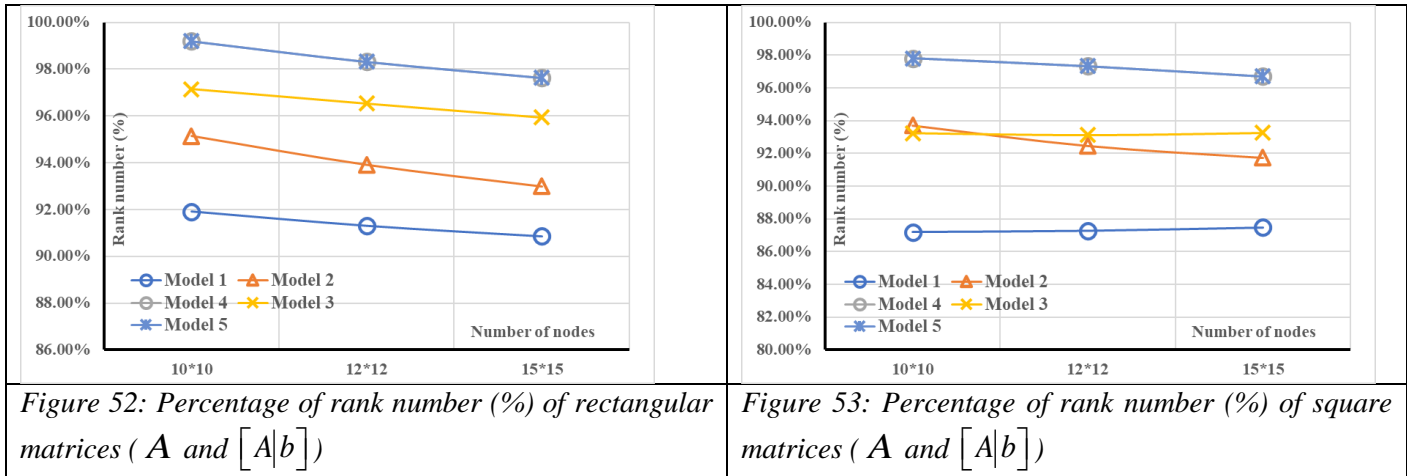


Figure 51: Condition number of square matrices

c) Rank number

Another property checked is the rank number of matrix A . The rank of matrix is equal to length of a longest linearly independent list of columns (or rows) of matrix (Johnson and Horn, 1985). For a system of linear equations $A \cdot x = b$, if the rank of A is equal to the rank of augmented matrix $[A|b]$, the system is consistent which means there is at least one solution for x . The rank number of rectangular matrices used in Test R1-5 and square matrices Test SE1-5 was obtained. The rank number as a percentage of the total number of rows of solution x is plot in below figures (Figure 60, Figure 61). The data of those figures can be found in Annex (Table 51, Table 52). The rank of augmented matrix $[A|b]$ for those tests was also obtained. It has been found that rank of A is equal to the rank of augmented matrix $[A|b]$ for all tests.

As shown in below figures, the rank number of matrix is reduced with higher number of nodes. For rectangular matrices, percentage of rank number of models: model 5 = model 4 > model 3 > model 2 > model 1. For square matrices, percentage of rank number of models: model 5 = model 4 > model 3 > model 2 > model 1. Regardless of type of matrix, percentage of rank number is less 100% for all models which means those linear systems solved shell code has infinitely many solutions.



5.3 Possible factors affecting shell code results

a) Type of matrix, difference approximation method, type of solver

In the above result sections (4.7, 4.8, 4.9), several comparisons have been made on numerical results in order to investigate three factors specifically including the type of matrix, difference approximation methods, and type of solver. Those are factors previously assumed to play an important role in affecting shell code results. However, those numerical result comparisons indicate that not every factor can significantly affect the results.

For example, the comparison between Rectangular matrix test and Square matrix test result shows that many results remain near equal regardless of the type of matrix, as shown in Figure 39 and Figure 40. No matter what method was used to produce square matrix, the most of square matrix results remained nearly unchanged. Meanwhile, the expected improvement on the result accuracy brought by square matrix was only observed on few displacement results with a small extent. In the most cases, rectangular matrix gives better results but also in a small degree.

In the comparison of results of five-point and two-point approximation methods (section: 4.9), the expected improvement on the result accuracy by five-point approximation depends on model type. For model 2, type A five-point approximation methods given the best bending moment results under *lsmr* solver (Figure 43). For model 4 & 5, the deviation of displacement and bending moment results was significantly reduced by applying five-point approximations under *lsmr* solver (Figure 44). In those results, Type B approximation provided more reduction on deviation. For the rest results under *lsmr* solver and the most results under *lm.fit.sparse* solver, regardless of the type of approximation method, shell code has given nearly identical value. The expected improvement on the result accuracy has been only found for limited cases. Specially for results under *lm.fit.sparse* solver, five-point approximation even dramatically increase the deviation. It indicates that the truncation error reduced by five-point approximation method did not play a significant role in affecting the accuracy of shell code results.

The results of *pinv* solver and *lm.fit.sparse* solver have shown better numerical accuracy than the *lsmr* solver in almost all cases (see 4.6). In the comparison between results of *pinv* and *lm.fit.sparse* solver, the *pinv* solver gives better displacement result and bending moment only for model 4 of 20*20 nodes. In rest cases, *pinv* solver and *lm.fit.sparse* solver gives results in similar level of accuracy.

b) Number of nodes, number of iterations

The effect of other factors including number of nodes can be found in the overall comparison (section 4.6). In finite element software, higher number of nodes generally brings more accurate results. The improvement on accuracy can be found in many results (Figure 27, Figure 28, Figure 32, Figure 33, Figure 34, Figure 37, Figure 38). They are displacement and bending moment results of model 4&5 and shear force results of model 1&5. However, the opposite has been found in many displacement and bending moment results of model 1&2 (Figure 24, Figure 25, Figure 26, Figure 29). As shown in those figures, the deviation of shell code results was increased as the number of nodes grows. While another factor that can be found in the overall comparison is the model setting. It is obvious that the results of models 1&2 (flat square shape) are less deviated on average compared to the results of models 3-5 (canopy shape). The highest deviation of results of models 1&2 is less than 400% while many results of models 3-5 have deviation over 1000%.

Another factor is the number of iterations for *lsmr* solver. It is equal to the number of columns of matrix A for the linear system $A \cdot x = b$ in default. If the number of nodes is 20×20 , the number of iterations for *lsmr* solver is $21 \cdot mn = 21 \cdot 20 \cdot 20 = 8400$. In order to investigate the effect of this factor, Tests R1, R3, SE1, SE3 with 20×20 nodes were recalculated by manually changing number of iterations ranged from $8400 \cdot 0.01$ to $8400 \cdot 100$. The deviation of those results is shown in below figures (Figure 62, Figure 63, Figure 64, Figure 65). The data of those figures can be found in Annex (Table 53, Table 54, Table 55, Table 56). Those data clearly shows that *lsmr* did not give the most accurate results under default setting. For model 1 (Test R1, Test SE1), the deviation of results can be reduced to less than 1% as the number of iterations increases by 100 times. For model 3 (Test R3, Test SE3), the deviation of results is dramatically lower with increasing number of iterations. However, the deviation of displacement results is still around 200% at $8400 \cdot 100$ iterations. A higher number of iterations can help to improve the numerical accuracy but to a different extent for different models.

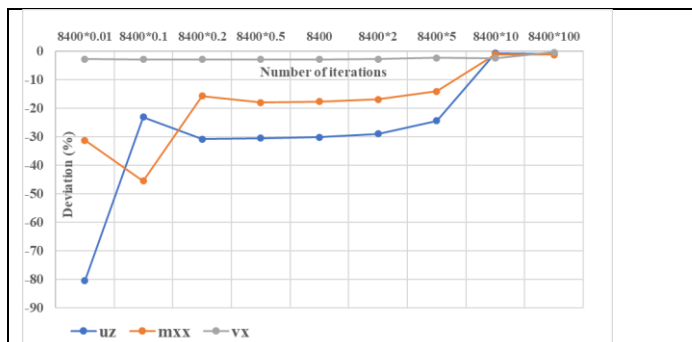


Figure 54: Deviation of Test R1 results by increasing number of iterations ($m=n=20$)

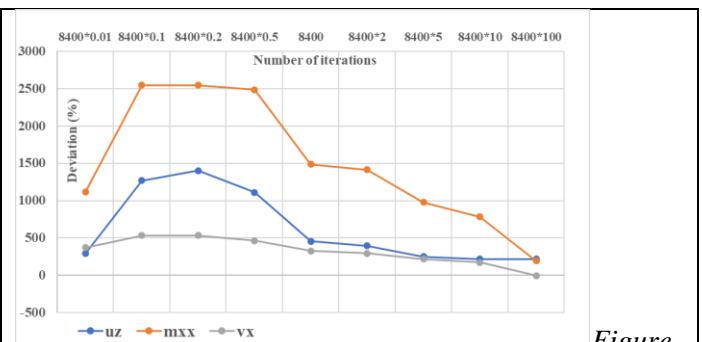


Figure 55: Deviation of Test R3 results by increasing number of iterations ($m=n=20$)

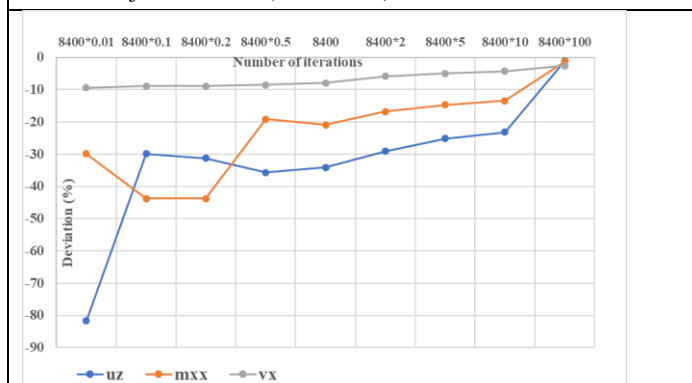


Figure 56: Deviation of Test SE1 results by increasing number of iterations ($m=n=20$)

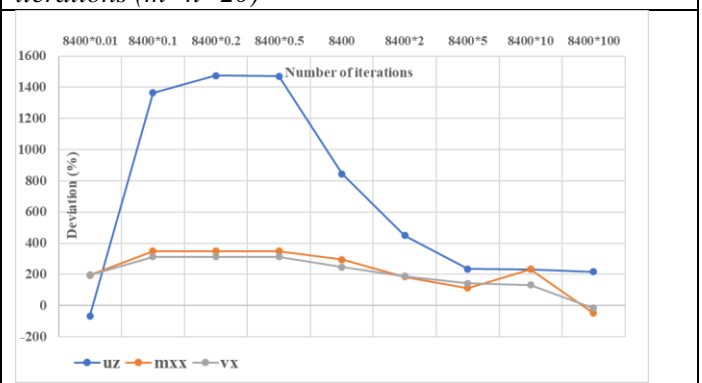


Figure 57: Deviation of Test SE3 results by increasing number of iterations ($m=n=20$)

c) Unit systems

In shell code, the unit system is determined when set up model parameters. Meanwhile, different unit system can lead a difference in several orders of magnitude for the numerical values of those parameters, thus affecting the numerical values of the matrix elements. For the models in pervious tests, the unit of force is kilo Newton (kN) and the unit of length is meter (m). In engineering practices, another unit system is widely used which is Newton for force (N) and millimeter for length (mm). In order to investigate the effects of value of matrix elements on the calculation, this unit system with another new unit system was used for Test R1-5 and Test LM 1-5. The new unit system uses 10 meters (10m) for length and kilo Newton (kN) for force. It is not designed for practical use but merely for changing the value of matrix elements in a different way. The values of model parameters with different unit system are shown in below:

Table 46: Model parameters by different unit systems

	Model 1-2	Model 3-5	All models
(kN, m) unit system (Default)	$l = 1m, t = 0.06m$	$l = 12m, t = 0.06m, a = 2m$	$p = 10 \text{ kN} / m^2,$ $E = 21 \times 10^7 \text{ kN} / m^2$ $E \cdot t = 1.26 \times 10^7 \text{ kN} / m$ $E \cdot t^3 = 4.536 \times 10^4 \text{ kNm}$
(N, mm) unit system	$l = 1000mm, t = 60mm$	$l = 12000mm, t = 60mm,$ $a = 2000mm$	$p = 10 \times 10^{-3} \text{ N} / mm^2,$ $E = 21 \times 10^4 \text{ N} / mm^2$ $E \cdot t = 1.26 \times 10^7 \text{ N} / mm$ $E \cdot t^3 = 4.536 \times 10^{10} \text{ Nmm}$
(kN, 10m) unit system	$l = 0.1 \text{ 10m}, t = 0.006 \text{ 10m}$	$l = 1.2 \text{ 10m}, t = 0.006 \text{ 10m},$ $a = 0.2 \text{ 10m}$	$p = 10 \times 10^2 \text{ kN} / 10m^2$ $E = 21 \times 10^9 \text{ kN} / 10m^2$ $E \cdot t = 1.26 \times 10^8 \text{ kN} / 10m$ $E \cdot t^3 = 4.536 \times 10^{-3} \text{ kN10m}$
For all unit system	Shell curvature: $k_{xx} = k_{yy} = k_{xy} = 0$ Lamé parameters: $\alpha_x = \frac{l}{m-1}, \alpha_y = \frac{l}{n-1}$ In plane curvature: $k_x = k_y = 0$	Shell curvature: $k_{xx} = k_{xy} = 0, k_{yy} = -\frac{1}{a}$ Lamé parameters: $\alpha_x = \frac{l}{m-1}, \alpha_y = \frac{\pi \cdot a}{n-1}$ In plane curvature: $k_x = k_y = 0$	

As shown in shell code sections (3.4, 3.5) the value of matrix elements is directly determined by value of model parameters (length l , thickness t , radius a , Young's modulus E , load p). When adding S-K equations, if there are components of first order derivative, the values of matrix element is determined by the finite difference approximation where the Lamé parameters α_x and α_y are involved. Those elements take up about 70% of total. Higher numerical value of length l leads higher numerical value of Lamé parameters which means smaller values of matrix elements for the approximated derivative (see section 3.3). For some components involving shell curvature higher numerical value of radius a leads to lower values of shell curvature k_{yy} which also contributes lower values of matrix elements. When adding constitutive equations (see Table 1), the value of matrix elements is determined by $E \cdot t$ and $E \cdot t^3$ where Young's modulus E and thickness t are both involved. For other elements that are not involved with $E \cdot t$, $E \cdot t^3$, shell curvature k_{yy} and finite difference approximation, their values are remain unchanged regardless of unit systems.

As shown in above table, compared the default (kN, m) unit system, the (N, mm) unit system brings higher values of length, thickness t and radius a and lower values of Young's modulus E and load p . It means under

(N, mm) unit system the elements of matrix [M] has lower value when they are produced by finite difference approximation. The matrix [M] under (kN, 10m) unit system is the exact opposite where those elements have higher value. While for the elements produced during adding constitutive equations, only a few of them has higher values under (N, mm) unit system since $E \cdot t$ is numerically equal to pervious one but $E \cdot t^3$ is dramatically higher. Under (kN, m) unit system, higher $E \cdot t$ and lower $E \cdot t^3$ makes some of these elements have higher values and some have lower values. In summary, applying (N, mm) unit system can lead to the majority of elements of matrix [M] has lower value and a few elements have higher values. Applying (kN, 10m) unit system means the opposite where majority of elements has higher value while a few elements have lower values.

The deviation of Test R1-5 and Test LM1-5 results under new unit systems was calculated and compared results with default unit system (see Figure 66, Figure 67, Figure 68, Figure 69, Figure 70). The data of new test results can be found in Annex (Table 57, Table 58, Table 59, Table 60). In order to display the results in details, only deviation between -100% and 100% is shown in those plots. Overall considered, no matter what unit system is used, deviation of most of results is not reduced but increased. The deviation of Test LM1-5 results under new unit systems is considerably higher compared those under default unit system. The highest deviation of Test LM1-5 under default unit system was below 100% but many new unit system results were over that. It is also true for Test R1-5 where deviation of the most of new unit system results were over 100%. The only exception is the results of Test R1 under (kN, 10m) unit system (see Figure 66). The deviation of those results was lower than the default unit system results. In conclusion, no matter it makes matrix elements have higher or lower values, applying new unit systems only cause more deviation in the most cases.

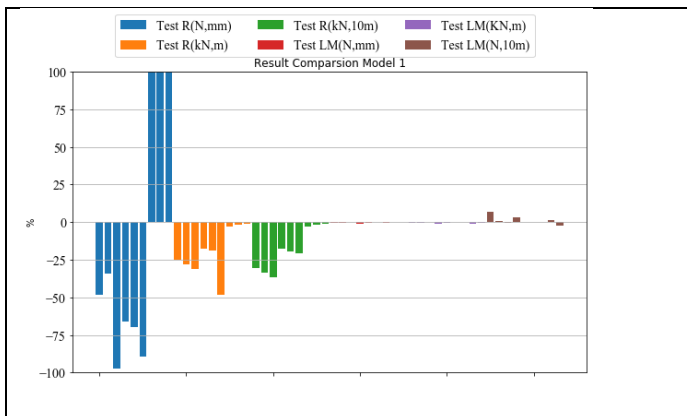


Figure 58: Deviation of model 1 results with different unit systems

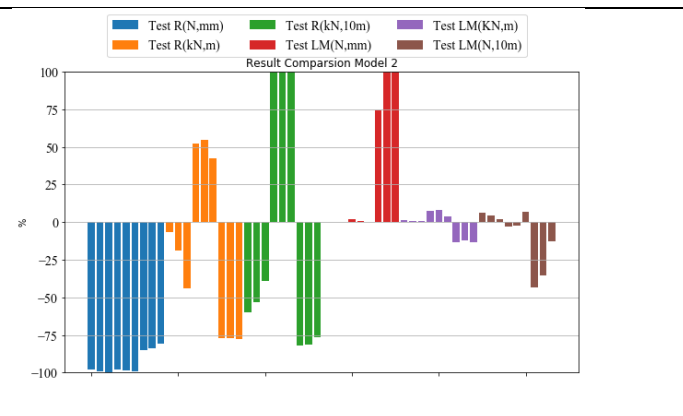


Figure 59: Deviation of model 2 results with different unit systems

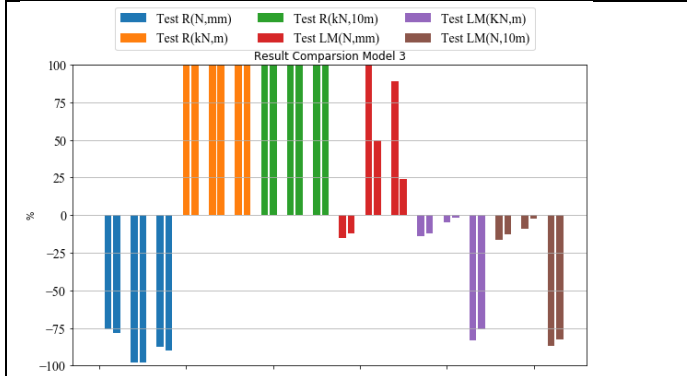


Figure 60: Deviation of model 3 results with different unit systems

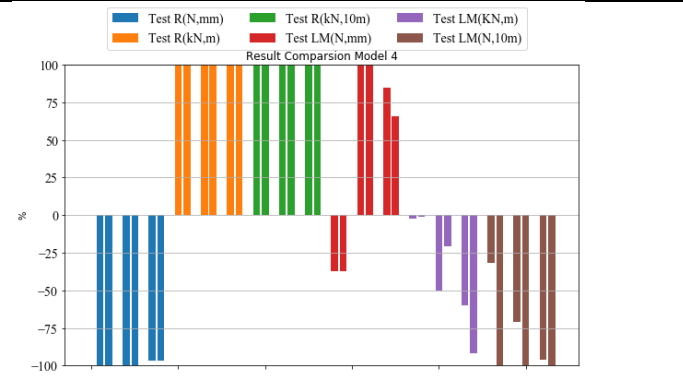
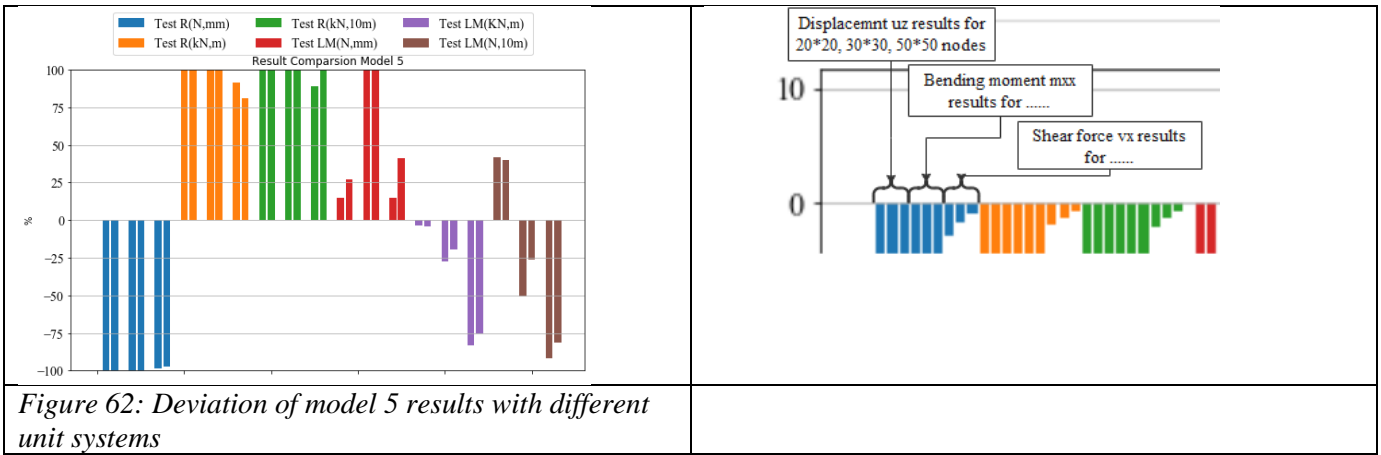
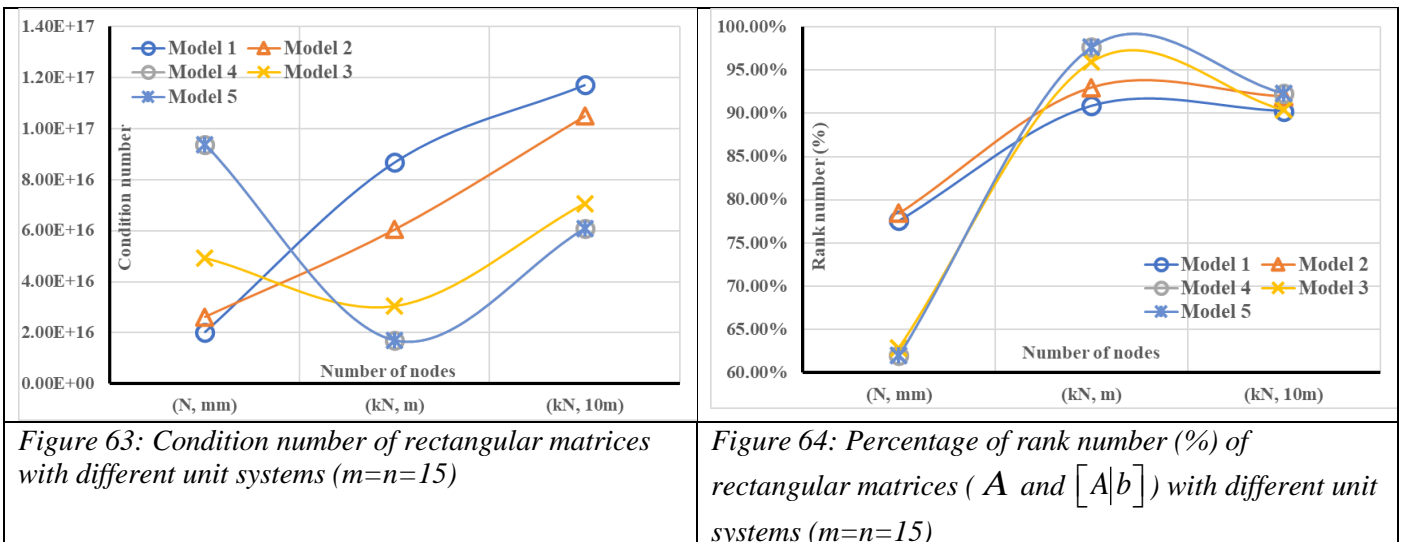


Figure 61: Deviation of model 4 results with different unit systems



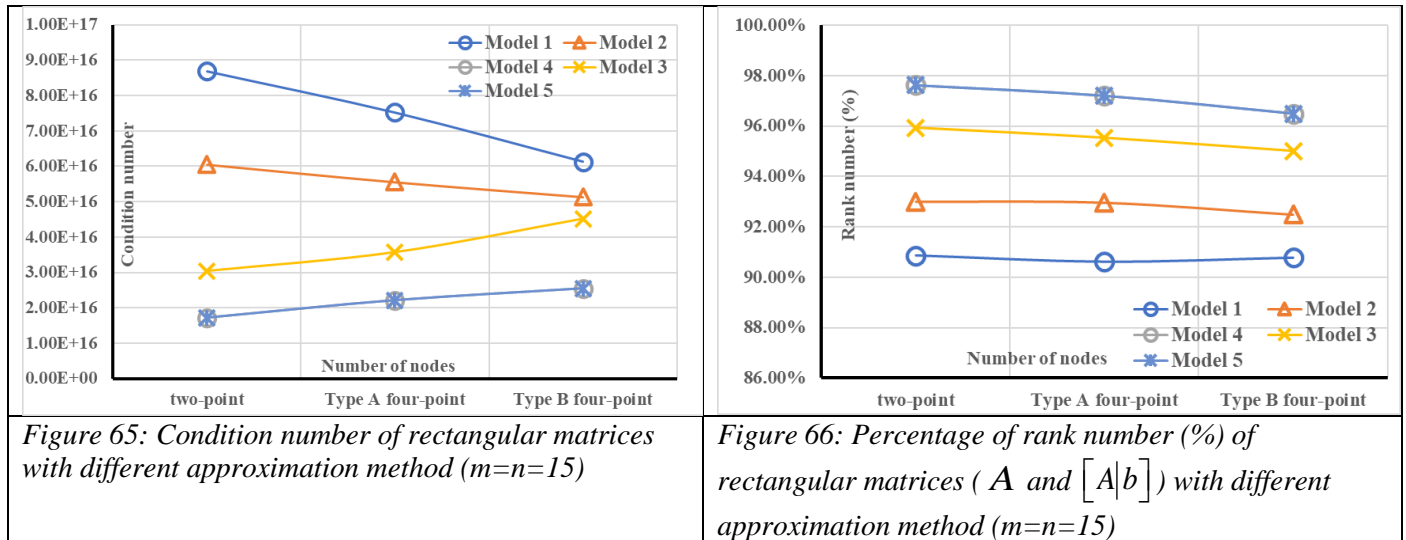
d) How do these factors work?

The factors discussed in above sections can be categorized into two types: factors that alter the matrix (number of nodes, difference approximated method, unit system), factors that alter the solving process (type of solver, number of iterations). The number of nodes directly changes the size of the matrix, which dramatically altering the quality of matrices. As shown in matrix quality section 5.2, sparsity, condition number, and rank number of generated matrices show a strong correlation with the number of nodes. Regardless of type of model and type of matrix, higher number of nodes increases the sparsity and condition number while rank number was reduced. It indicates that the approximated solution of linear systems with higher number of nodes is more possible to having large numerical errors, which has been reflected in shell code results. When applying different the unit systems, the value of the most matrix elements has been reduced or magnified exponentially. The altered matrix shows different properties (see Figure 71, Figure 72). Matrices were even more rank deficient if a new unit system was used. Meanwhile the condition number of matrices was higher with new unit systems. The only exception occurred for model 1 & 2 where their condition number was lower when using (N, mm) unit system. It could explain why only deviation of some model 1 and model 2 results was lower under (N, mm) unit system while other results deviation was higher with new unit system.



Applying new difference approximation method also altered the matrices whose properties is shown in below figures (Figure 73, Figure 74). The condition number of matrices with five-point approximation increased for model 3-5 while decreased for model 1 & 2. These increases and decreases occur on a relatively small scale. The rank number with five-point approximation has varied but also on a relatively small scale. Mathematically, the inherent numerical error brought by two-point approximation is larger than that of five-point approximation. If such error plays an important role in affecting deviation of results, the properties of matrices are expected to be significantly altered by applying a new five-point approximation. However, such alteration

has not been found. It indicates that the finite difference approximation method does not have a significant effect on the shell code results.



The most profound difference on shell code results was caused by the type of solver. However, due to the lack of information on *lm.fit.sparse* solver, the specific details on how this solver has such an advantage still remains unclear. The possible reason might relate to the sparse QR factorization in *lm.fit.sparse* solver and the stopping rules in *lsmr* solver. As illustrated in 2.5 section, applying QR factorization to system matrix is a necessary step for both *lm.fit.sparse* solver and *lsmr* solver. In normal QR factorization the system equation $A \cdot x = b$ is converted to $R^{-T} A^T b = Q^T b$ where Q is orthonormal matrix and R is upper triangular matrix and $A = Q \cdot R$. The orthonormal matrix Q of an overdetermined sparse system cannot be found explicitly which can adversely affect numerical precision of results. The computational error in this process is influenced by the sparsity and size of the matrix. So that factorization of smaller matrices could have less computational error than the factorization of entire matrix. The general strategy for sparse QR factorization is dividing A into numbers of smaller matrices $A_1 \dots A_K$. The factorization of those smaller matrices is computed individually to the form the final result of factorization of A instead of applying factorization directly to a large sparse overdetermined system as *lsmr* solver. So that compared to *lsmr* and *pinv* solver, *lm.fit.sparse* solver produces results with much less deviation in overall. Like the number of iterations, the stopping rule of *lsmr* solver could also adversely affect results. Three stopping rules mentioned in 2.5 section show that iteration will stop when the residual of the final iteration is smaller than the stopping tolerance $ATOL$ and $BTOL$. The default stopping tolerance used in tests is $ATOL = BTOL = 1 \times 10^{-12}$ which means relative error in A and b is assumingly equal to this value. In order to show the effect of this factor, a series of new stopping tolerances were used for Test R1&3 and Test SE1&3 and results are shown in below figures (Figure 75, Figure 76, Figure 77, Figure 78). It shows that default stopping tolerance does not guarantee a lower deviation for every result. The deviation of displacement results even increases with lower stopping tolerance used. For model 3, the deviation of shear force and bending moment results also grows as the stopping tolerance decreases. The lowest deviation of results occurred at some point below default stopping tolerance and it is different for each type of results. It indicates that the current default stopping tolerance is not the best setting to obtain the accurate results. In conclusion, the sparse QR factorization reduces computational error in *lm.fit.sparse* solver results and improper stop tolerance setting enlarge the error in *lsmr* solver results.

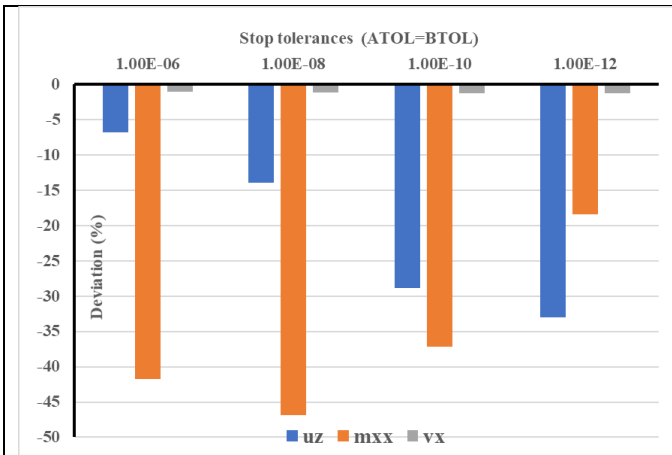


Figure 67: Tests on stopping tolerances in lsmr solver for Test R1 (m=n=30)

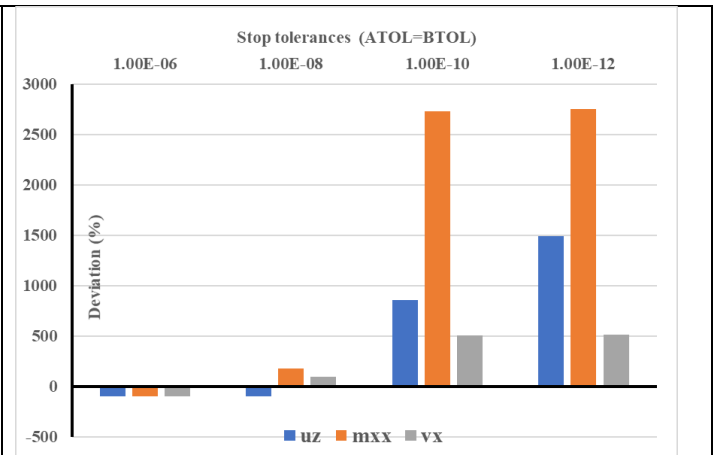


Figure 68: Tests on stopping tolerances in lsmr solver for Test R3 (m=n=30)

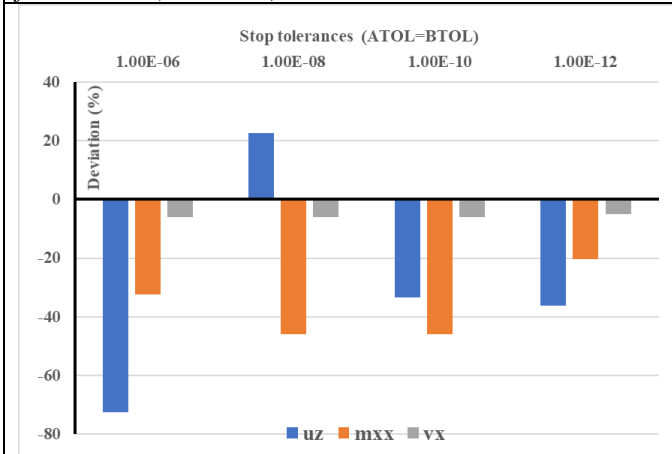


Figure 69: Tests on stopping tolerances in lsmr solver for Test SE1 (m=n=30)

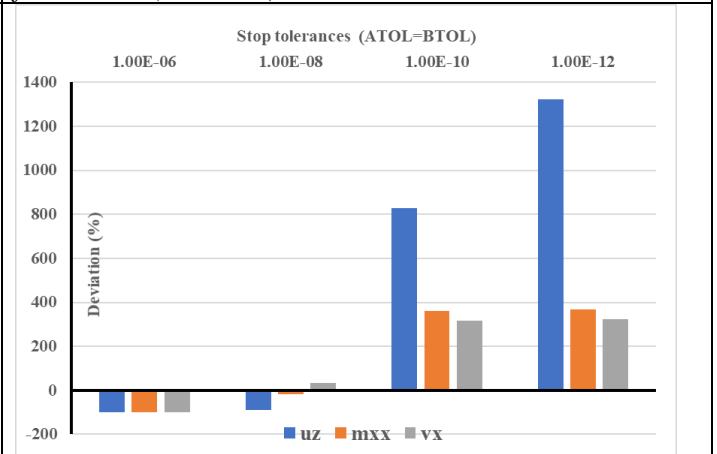


Figure 70: Tests on stopping tolerances in lsmr solver for Test SE3 (m=n=30)

5.4 Simplicity of code structure

One of the objectives of this project is the being able to easily modify the code structure in order to change applied shell theory. In common finite element software, applying a different theory generally means switching the element type and user defined element usually is not allowed. Some sophisticate finite element software can do that but require coding and mathematic knowledge. Compared to them, the advantage of shell code is that it allows users to apply a new theory to models in a much straightforward manner.

In shell code, modifying shell theory equations can be achieved by directly deleting or adding code lines. For example, if torsion behaviour of models is not considered while S-K equations is still wanted, it can be directly achieved by muting the lines involving shear strain γ_{xy} in the equation adding process. As shown in section 3.4, when adding equations to the models, each component of equation is added individually and of course can be removed individually.

<p>S-K equation 9 (remove γ_{xy}):</p> $\frac{n_{xy} + n_{yx}}{2} = \frac{Et}{2(1+\nu)} \gamma_{xy}$ $\Rightarrow \frac{n_{xy} + n_{yx}}{2} = 0$	<pre> for j in range(n): # Add Sanders-Koiter equation 9 to the matrix ----- for i in range(m): row=row+1 M[row,nxy*m*n+j*m+i]=-0.5 M[row,nyx*m*n+j*m+i]=-0.5 # M[row,gammaxy*m*n+j*m+i]=k #muted </pre>
--	--

S-K equation 15 (remove γ_{xy}):

$$\cancel{\gamma_{xy}} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} - 2k_{xy}u_z - k_x u_x - k_y u_y$$

$$\Rightarrow 0 = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} - 2k_{xy}u_z - k_x u_x - k_y u_y$$

```

for j in range(n): # Add Sanders-Koiter equation 15 to
the matrix -----
    for i in range(m):
        row=row+1
        # M[row,gammaxy*m*n+j*m+i]=-1 #muted
        Dly(ux,1.0)
        Dlx(uy,1.0)
        M[row,uz*m*n+j*m+i]=-2*kxy(i/(m-1),j/(n-1))
        M[row,ux*m*n+j*m+i] +=-kx(i/(m-1),j/(n-1))
        M[row,uy*m*n+j*m+i] +=-ky(i/(m-1),j/(n-1))

```

The whole equation or entire set of equations can be modified like this. It is possible to replace the entire S-K equations with another set of user-defined equations. Users are free to adopt their own theory to solve model problems in the shell code.

5.5 Plate code and shell code

In the previously developed plate code (Li, 2020), three plate models have been tested whose results were usually 10 to 20% lower than its analytical solution. The main conclusion was that it is possible to solve plate equations by only applying first-order finite-difference approximation. The assumed computation error in those results was suspected to be related to the programming error and solving method.

6. Conclusion

The goal of this project was to develop a workable algorithm that can solve shell model problems through the finite difference method and understand how the results can be affected. The finite difference results were compared to finite element solutions. The main conclusion is that the selected version of the algorithm performs a relatively good analyses of the considered shell models. It produces displacement, shear force and bending moment results for both the flat square shape and the canopy shape with various boundary conditions and load conditions. This algorithm uses the *lm.fit.sparse* solver, an over-determined system of equations (rectangular matrix) and a five-point interpolation (version B). About 80% of shell code results match the finite element results with a deviation less than 5%.

Many factors have little influence on the computation results (displacement, membrane forces, moments, and shear forces). Examples of these factors are the mathematic property of the matrix (determined or overdetermined), the finite difference approximation method, and the unit system. The effect of those factors has been investigated with sparsity, conditional number, and rank number of the matrices. Those matrix quality checks have shown the non-uniform way of change in conditional number and rank number of matrices when those factors were monotonically altered. There is not a clear relationship between those factors and deviations of shell code results from corresponding tests. This means that obtaining the best results by altering those factors could a very case-sensitive and time-consuming task.

Since in most cases the constructed matrices were ill-conditioned, the standard solver *numpy.linalg.solve* (Python) was not used. Three solvers have been implemented and compared; *numpy.linalg.lstsq* (Python), *numpy.linalg.pinv* (Python), *lm.fit.sparse* (R). The latter is able to compute the 50×50 grid, however, this solver is not easy to implement. Unfortunately, the results strongly depend on settings in the solving steps such as the number of iterations and stop tolerances (see pp 61, 66). With the help of sparse QR factorization, the best shell code results were produced by the *lm.fit.sparse* solver. Some *numpy.linalg.pinv* solver results can reach a similar level of numerical accuracy but it is more expensive in terms of time and memory usage.

The boundary conditions have been implemented by adding equations or by replacing equations. The former leads to an over determined system the latter to a determined system of equations. There are no remarkable differences in computation results between these systems.

The discretization has been implemented by three-point interpolation and by five-point interpolation. The five-point interpolation type B produces very promising results (see pp. 33, 113). Sometimes an hourglass mode occurs as a slight undulation on the results (see pp. 75, 77, 80). The undulation changes with more grid points but does not disappear. The undulation is strongest for determined systems.

Through numbers of tests and discussions, one thing has been proved is that the selected version of shell code can solve shell model problems by solving Sanders-Koiter equations with finite difference method. The current best configuration of algorithm is using rectangular matrix which means describing boundary conditions by adding equations instead of replacing them, and using a five-point interpolation for discretization. Many previously assumed important factors for affecting shell code results were actually less significant. During the process of exploring the possible reasons for numerical error in generated results, the most tests are about changing the inputs of computation and they did not affect shell code significantly. However, the most vital difference in results occurred under different computation methods. It is reasonable to assume that there should be a more powerful mathematic tool for solving an overdetermined linear sparse system. Maybe under further research of developing mathematic tools, finite difference method could be a more promising and practical method for solving model problems.

7. Recommendation

As discussed in the above sections, the solving method should be a key part of further improvement of the algorithm. The correct solver settings are essential for the quality of results. In the current shell code, matrix quality checks and solver settings were manually set up in previous tests. It is possible to build an iterative process so that the solver can automatically alter its setting. The deviation of shell code results will be checked and work as a reference for altering solver settings in the next iteration until deviation results is within an acceptable range. It could help to find the best solver settings for every case efficiently.

Another possible improvement is to find a new iterative solver, which can apply a vector of weights in the fitting process. The stop rules of current iterative solvers used in shell code (*lsmr* and *lm.fit.sparse*) all consider the residual r_k for the approximate solution x_k . Once it meets the preset stop tolerance, the solver will stop and give results. This residual is calculated as $r_k = b - Ax_k$ where all the rows of system matrix A are equally involved. However, rows for equations describing boundary conditions are only accounted for less than 1% of the whole matrix. This means the boundary conditions might be not satisfied as the residual is low enough to meet the stop rules since the rest rows for S-K equations describing model body dominates the residual r_k . This might explain why distortion of results has been usually found on boundaries of model. This assumed vector of weights could manually increase the weights of boundary condition equations in calculating the residual so that when residual meets the stop rules the boundary conditions are satisfied.

8. Reference list

- Amabili, M. (2003) ‘A comparison of shell theories for large-amplitude vibrations of circular cylindrical shells: Lagrangian approach’, *Journal of Sound and Vibration TA - TT -*, 264(5), pp. 1091–1125. doi: 10.1016/S0022-460X(02)01385-8 LK - <https://tudelft.on.worldcat.org/oclc/4924705977>.
- Assadi-Lamouki, A. and Krauthammer, T. (1989) ‘An explicit finite difference approach for the Mindlin plate analysis’, *Computers & structures*. Elsevier, 31(4), pp. 487–494.
- Bernoulli, J., J. (1789) ‘Essai theorique sur les vibrations de plaques elastiques rectangularies et libers’, *Nova Acta Acad Petropolis*, 5, pp. 197–219.
- Cauchy, A.-L. (1828) ‘Sur l’equilibre et le mouvement d’une plaque solide’, *Exercises de Matematique*, 3(1828), pp. 328–355.
- Chladni, E. F. F. (1802) *Die akustik*. Breitkopf & Härtel.
- Cline, A. K. *et al.* (1979) ‘An estimate for the condition number of a matrix’, *SIAM Journal on Numerical Analysis*. SIAM, 16(2), pp. 368–375.
- Davis, T. A., Rajamanickam, S. and Sid-Lakhdar, W. M. (2016) ‘A survey of direct methods for sparse linear systems.’, *Acta Numer.*, 25, pp. 383–566.
- Euler, L. (1766) ‘De motu vibratorio tympanorum’, *Novi commentarii academiae scientiarum Petropolitanae*, pp. 243–260.
- Fernandez-Granda, C. (2016) *DS-GA 1013 / MATH-GA 2824 Mathematical Tools for Data Science*, New York University. Available at: <https://cde.nyu.edu/math-tools/>.
- Fong, D. C.-L. and Saunders, M. (2011) ‘LSMR: An iterative algorithm for sparse least-squares problems’, *SIAM Journal on Scientific Computing*. SIAM, 33(5), pp. 2950–2971.
- Germain, S. (1826) *Remarques sur la nature, les bornes et l’étendue de la question des surfaces élastiques, et équation générale de ces surfaces*. Huzard-Courcier.
- Johnson, C. R. and Horn, R. A. (1985) *Matrix analysis*. Cambridge university press Cambridge.
- Kirchhoff, G. (1850) *Über das Gleichgewicht und die Bewegung einer elastischen Scheibe*.
- Koiter, W. T. and Delft University of Technology, D. of M. E. L. of E. M. (1966) *Purpose and achievements of research in elastic stability.*, Report Department of Mechanical Engineering, Delft University of Technology ; 363 TA - TT -. Delft SE - 29 blz. ; .. cm.: Delft University of Technology. Available at: <https://tudelft.on.worldcat.org/oclc/841152853>.
- Lagrange, J. L. (1828) ‘Note communiquée aux Commissaires pour le prix de la surface élastique décembre 1811’, *Ann. Chimie Physique*, 39(149151), p. 1828.
- Li, C. (2020) *Finite difference analysis of plate structures*. Delft University of Technology. Available at: http://homepage.tudelft.nl/p3r3s/BSc_projects/eindrapport_chulong_li.pdf.
- Love, A. (1892) ‘A treatise on the mathematical theory of elasticity’.
- Marcus, H. (1932) *Die Theorie elastischer Gewebe und ihre Anwendung auf die Berechnung biegsamer Platten*. Springer.
- Navier, C. (1823) ‘Bulletin des Sciences de la Societe Philomathique de Paris’. Paris.
- O’Connor, D. (1985) ‘Report on the Dublin matrix theory conference, March 1984: An introduction to sparse matrices’, *Linear Algebra and its Applications*. North-Holland, 68, pp. 271–272.
- Paige, C. C. and Saunders, M. A. (1982) ‘LSQR: An algorithm for sparse linear equations and sparse least squares’, *ACM Transactions on Mathematical Software (TOMS)*. ACM New York, NY, USA, 8(1), pp. 43–71.
- Penrose, R. (1955) ‘A generalized inverse for matrices’, in *Mathematical proceedings of the Cambridge philosophical society*. Cambridge University Press, pp. 406–413.
- Poisson, S.-D. (1828) *Mémoire sur l’équilibre et le mouvement des corps élastiques*. F. Didot.
- Reddy, J. N. and Gera, R. (1979) ‘An improved finite-difference analysis of bending of thin rectangular elastic plates’, *Computers and Structures*, 10(3), pp. 431–438. doi: 10.1016/0045-7949(79)90018-X.
- Reissner, E. (1941) ‘A new derivation of the equations for the deformation of elastic shells’, *American Journal of Mathematics*. JSTOR, 63(1), pp. 177–184.
- Sanders, J. L. (1960) *An improved first-approximation theory for thin shells*. US Government Printing Office.
- Sanders, J. L. (1963) ‘Nonlinear theories for thin shells’, *Quarterly of Applied Mathematics*, 21(1), pp. 21–36. doi: 10.1090/qam/147023.

- Sanders Jr, J. L. (1967) 'On the shell equations in complex form'.
- Svoboda, J., Cashman, T. and Fitzgibbon, A. (2018) 'QRkit: Sparse, Composable QR Decompositions for Efficient and Stable Solutions to Problems in Computer Vision', in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, pp. 1263–1272.
- Szilard, R. T. A.-T. T.- (1974) 'Theories and Applications of Plate Analysis : Classical Numerical and Engineering Methods'. Hoboken: Wiley [Imprint]. Available at:
<http://onlinelibrary.wiley.com/book/10.1002/9780470172872>.
- Thomé, V. (2001) 'From finite differences to finite elements. A short history of numerical analysis of partial differential equations', *Journal of Computational and Applied Mathematics*, 128(1–2), pp. 1–54. doi: 10.1016/S0377-0427(00)00507-0.
- Timoshenko, S. (1913) *Sur la stabilité des systèmes élastiques*. A. Dumas.
- Timoshenko, S. P. (1915) 'On large deflections of circular plates', *Mem Inst Ways Commun*, 89.
- Timoshenko, S. P. and Woinowsky-Krieger, S. (1959) *Theory of plates and shells*. McGraw-hill.
- Trefethen, L. N. and Bau III, D. (1997) *Numerical linear algebra*. Siam.
- Ventsel, E. and Krauthammer, T. T. A.-T. T.- (2001) 'Thin plates and shells : theory, analysis, and applications'. New York: Marcel Dekker. doi: 10.1201/9780203908723 LK -
<https://tudelft.on.worldcat.org/oclc/54351771>.

9. Appendix

Rectangular matrix test plots

Test R1, three-point interpolation, flat square, vertical loading, rectangular matrix

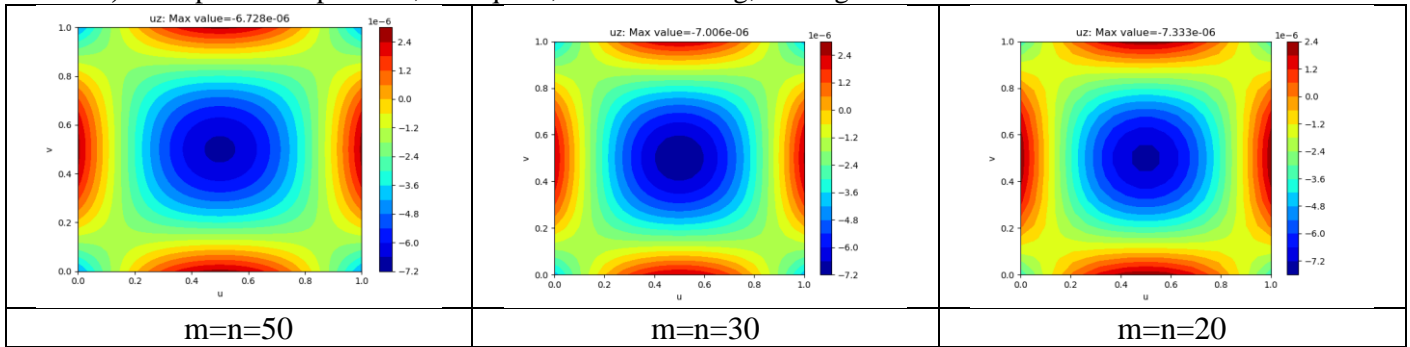


Figure 71: Test R1 displacement uz results ($m=n=20, 30, 50$)

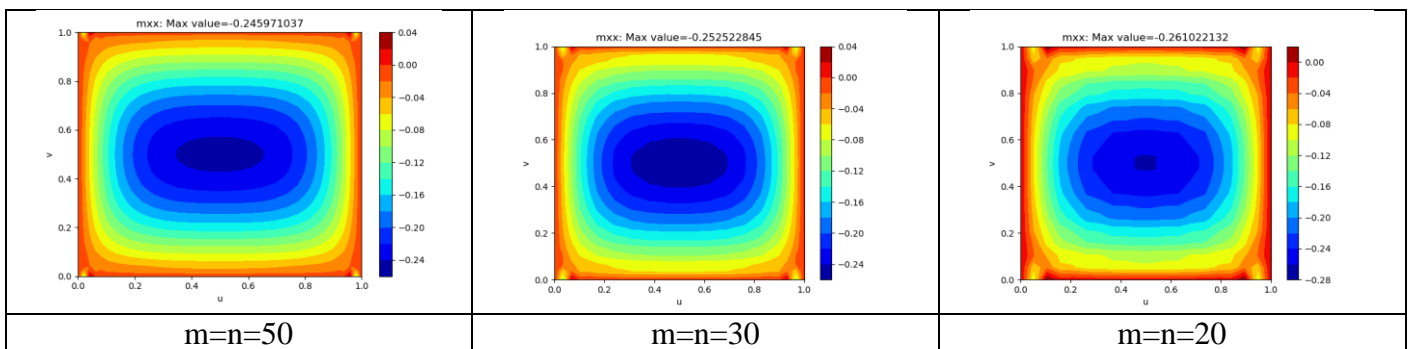


Figure 72: Test R1 bending moment mxx results ($m=n=20, 30, 50$)

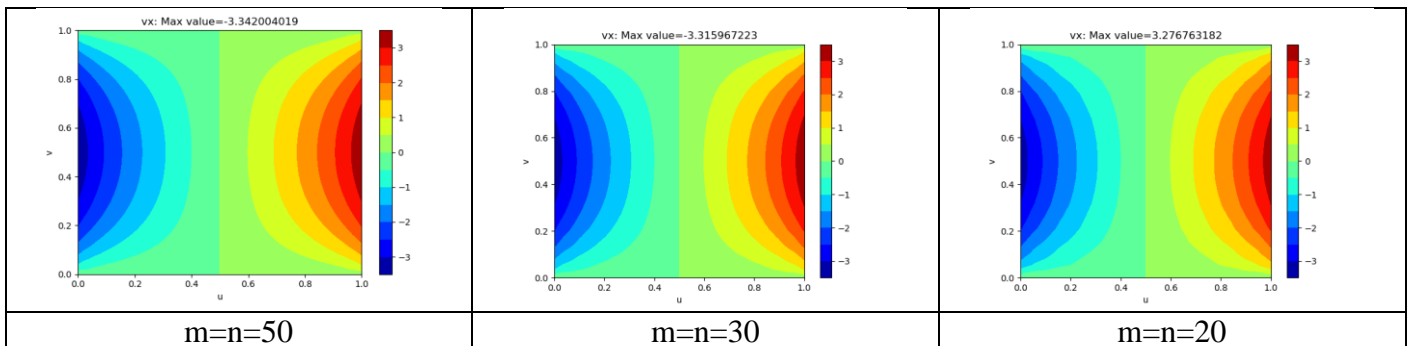


Figure 73: Test R1 shear force vx results ($m=n=20, 30, 50$)

Test R2, three-point interpolation, flat square, vertical loading, rectangular matrix

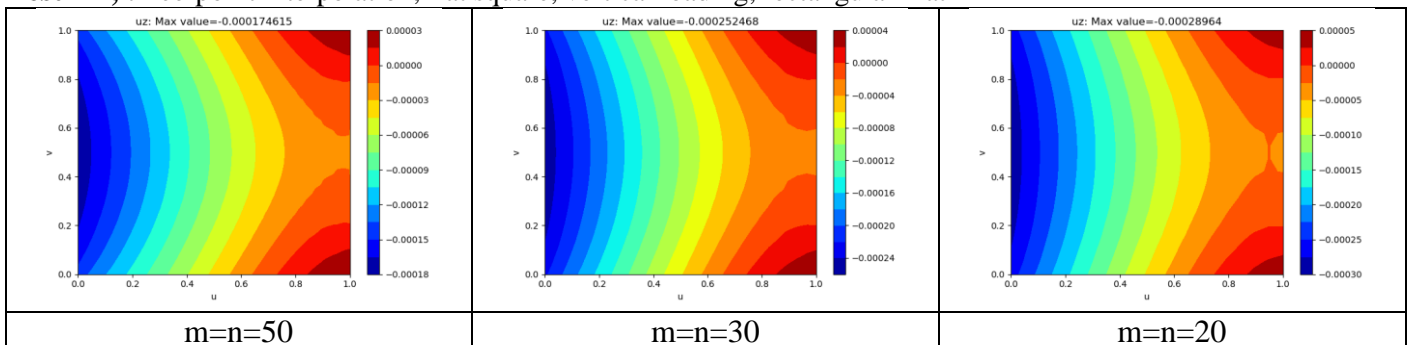


Figure 74: Test R2 displacement uz results ($m=n=20, 30, 50$)

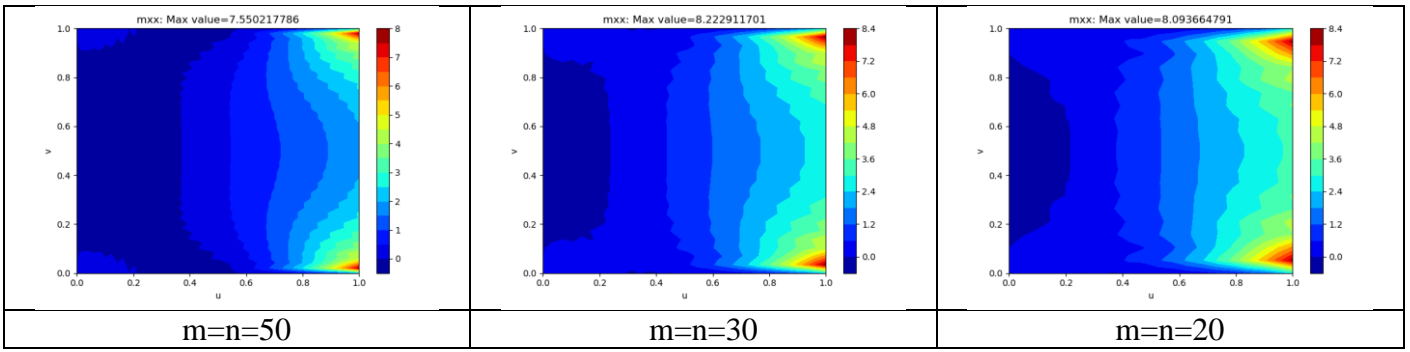


Figure 75: Test R2 bending moment m_{xx} results ($m=n=20, 30, 50$)

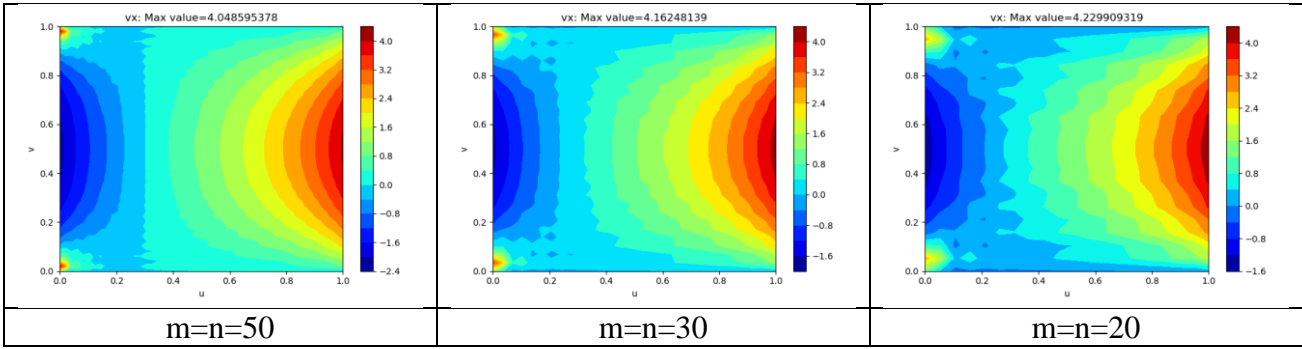


Figure 76: Test R2 shear force v_x results ($m=n=20, 30, 50$)

a) Test R3, three-point interpolation, canopy, vertical loading, rectangular matrix

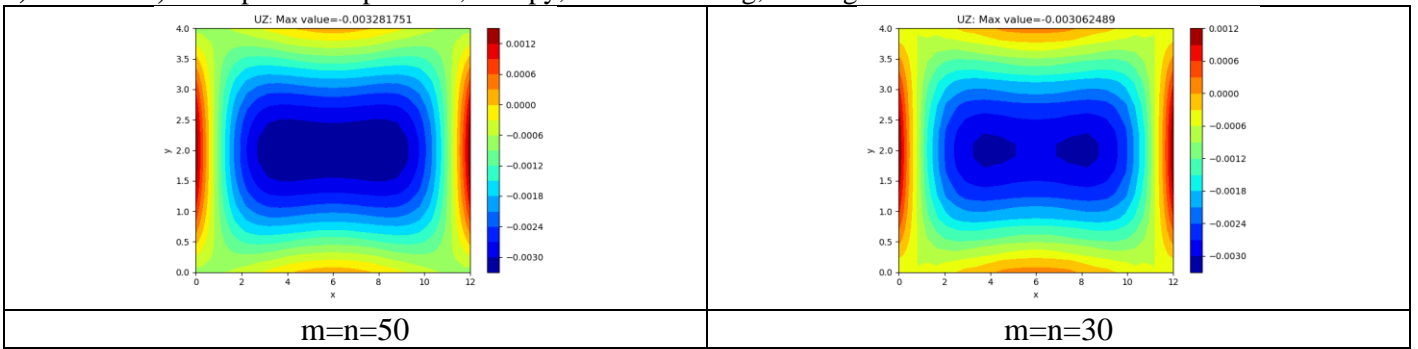


Figure 77: Test R3 displacement u_z results ($m=n=20, 30, 50$)

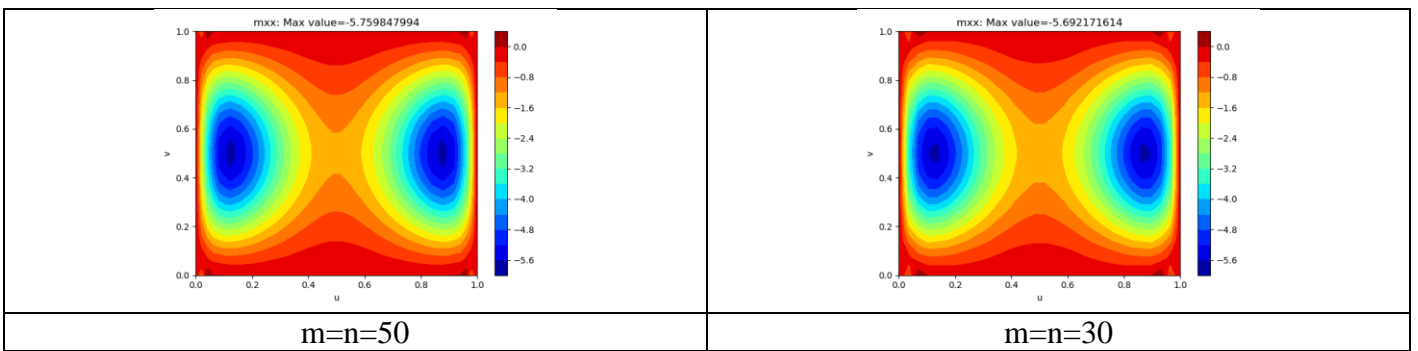


Figure 78: Test R3 bending moment m_{xx} results ($m=n=20, 30, 50$)

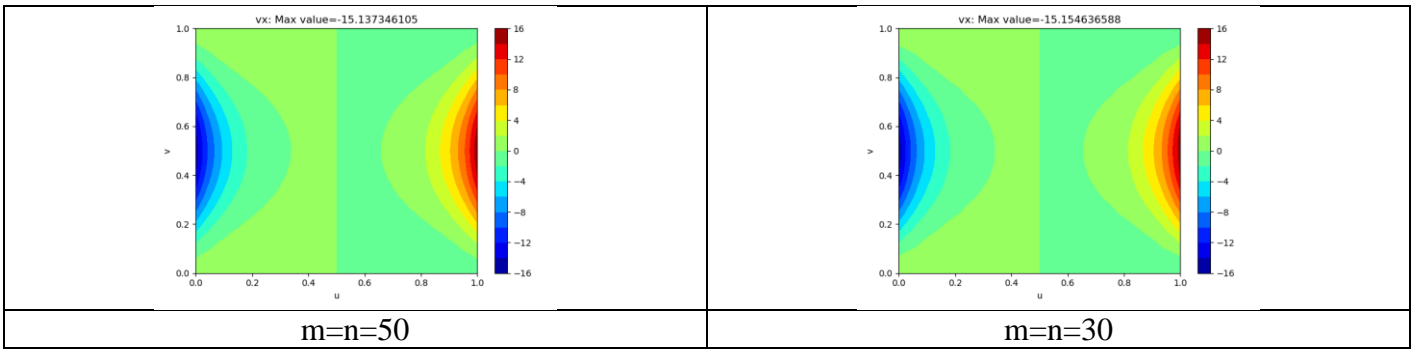


Figure 79: Test R3 shear force v_x results ($m=n=20, 30, 50$)

b) Test R4, three-point interpolation, canopy, vertical loading, rectangular matrix

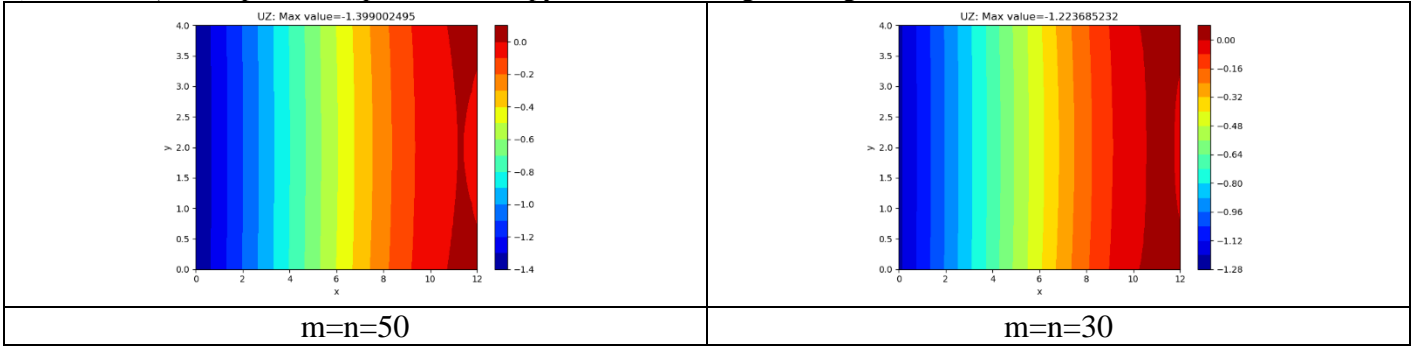


Figure 80: Test R4 displacement u_z results ($m=n=20, 30, 50$)

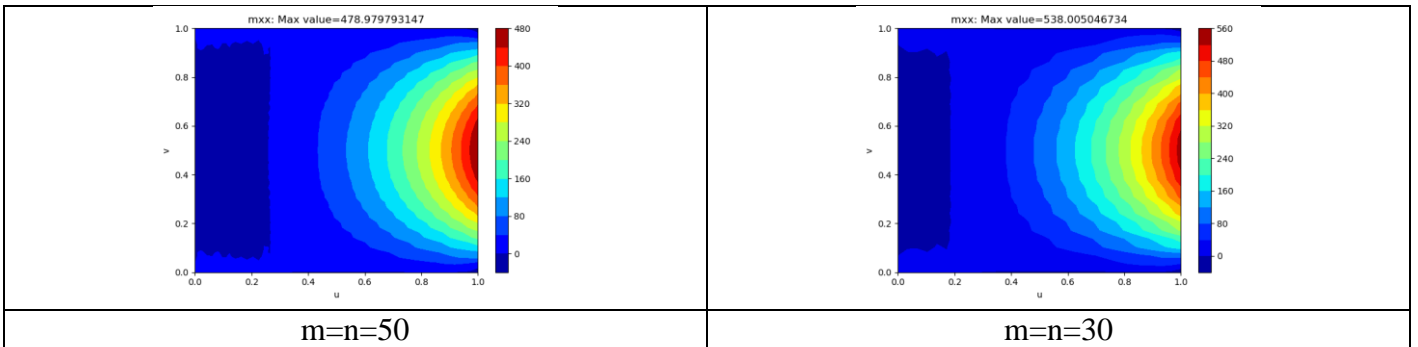


Figure 81: Test R4 bending moment m_{xx} results ($m=n=20, 30, 50$)



Figure 82: Test R4 shear force v_x results ($m=n=20, 30, 50$)

c) R5, three-point interpolation, canopy, normal loading, rectangular matrix

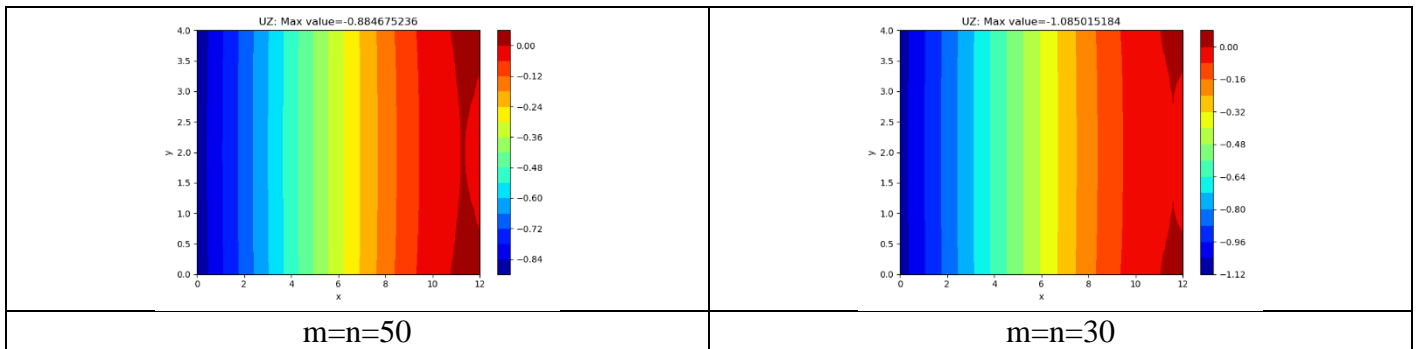


Figure 83: Test R5 displacement uz results ($m=n=20, 30, 50$)



Figure 84: Test R5 bending moment mxx results ($m=n=20, 30, 50$)

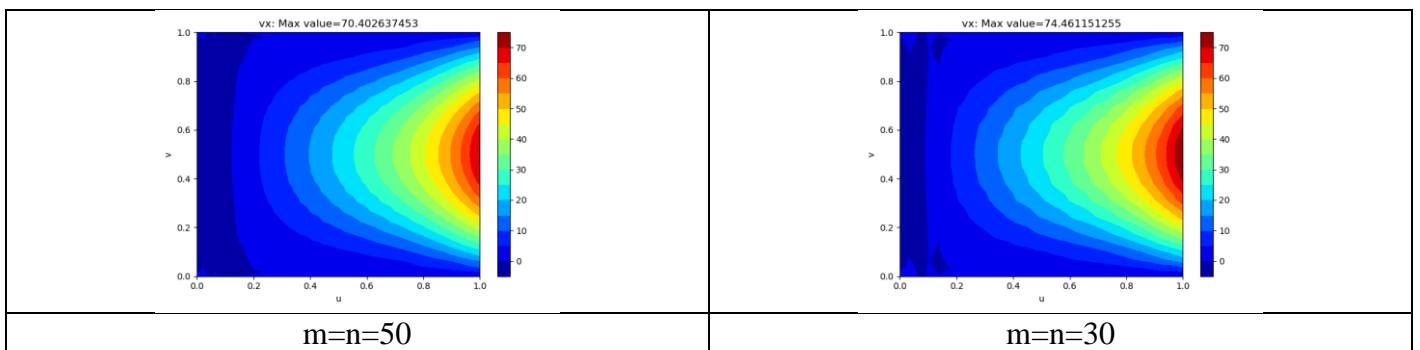


Figure 85: Test R5 shear force vx results ($m=n=20, 30, 50$)

Square matrix test plots

Test SC1, three point interpolation, flat square, vertical loading, square matrix

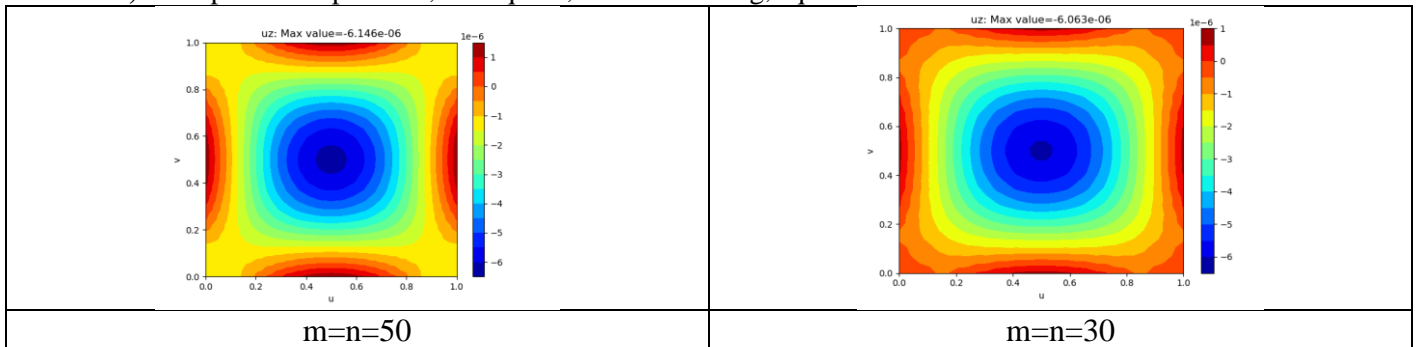


Figure 86: Test SC1 displacement uz results ($m=n= 30, 50$)

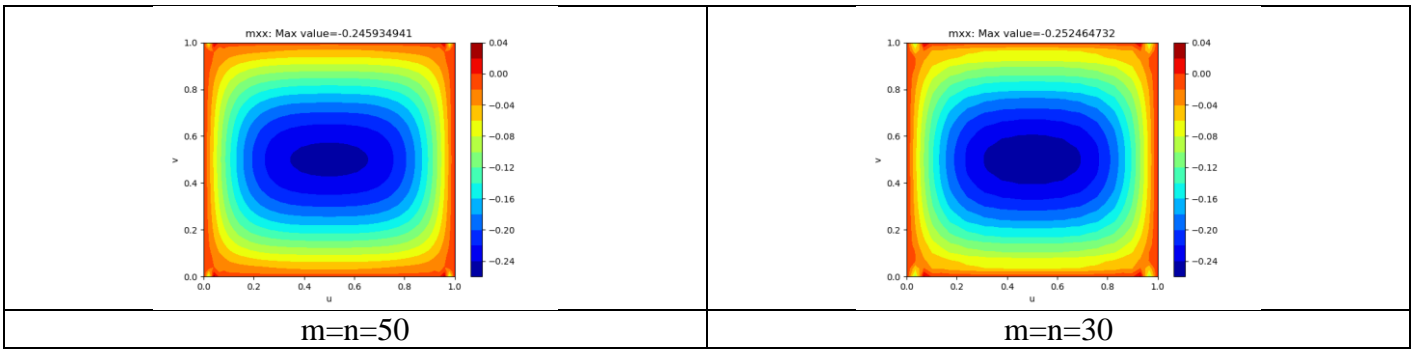


Figure 87: Test SC1 bending moment m_{xx} results ($m=n=30, 50$)

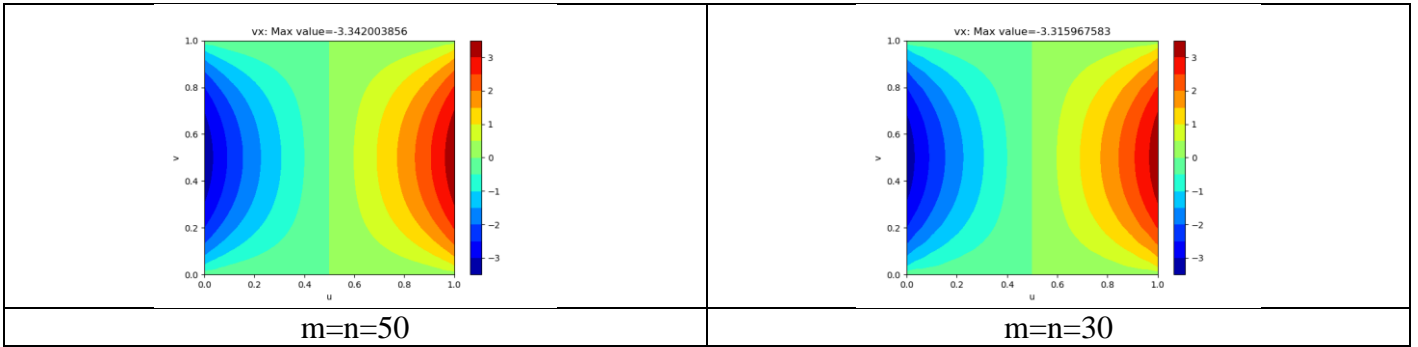


Figure 88: Test SC1 shear force v_x results ($m=n=30, 50$)

Test SC2, three point interpolation, flat square, vertical loading, square matrix

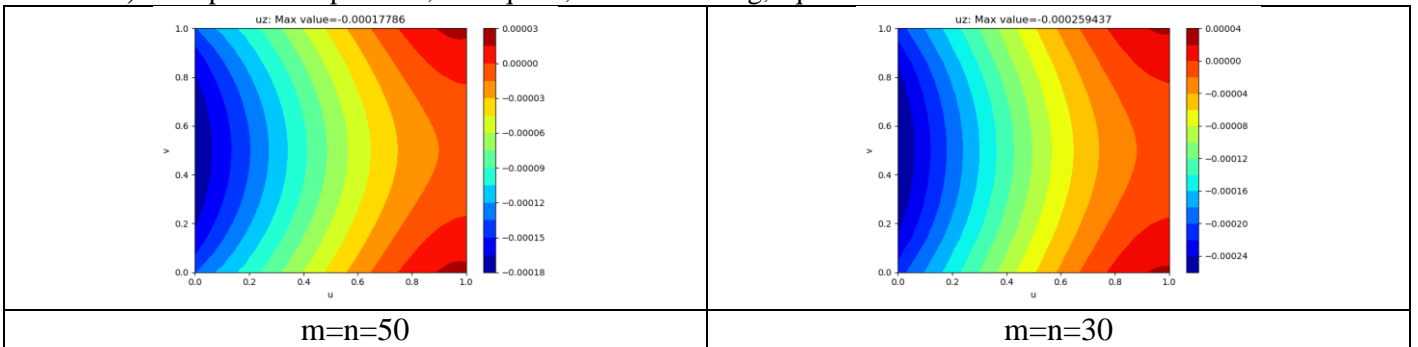


Figure 89: Test SC2 displacement u_z results ($m=n=30, 50$)

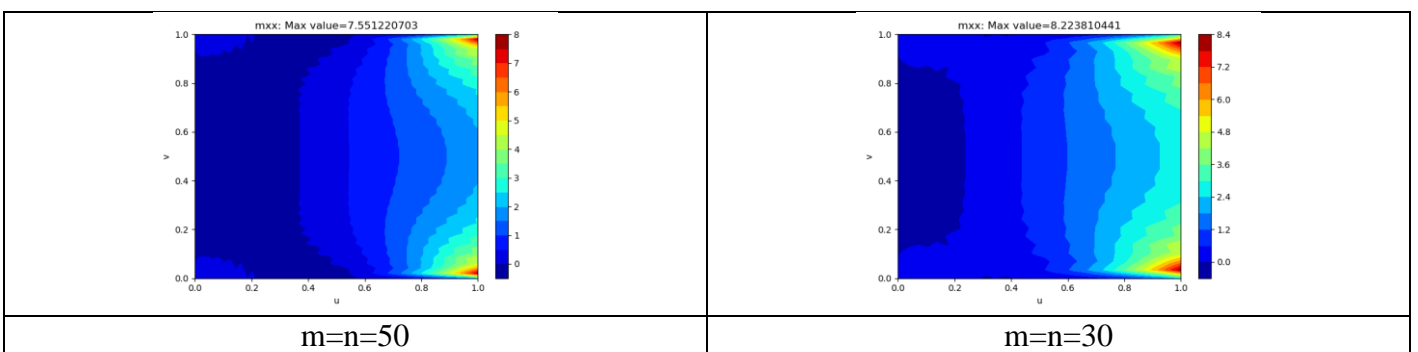


Figure 90: Test SC2 bending moment m_{xx} results ($m=n=30, 50$)

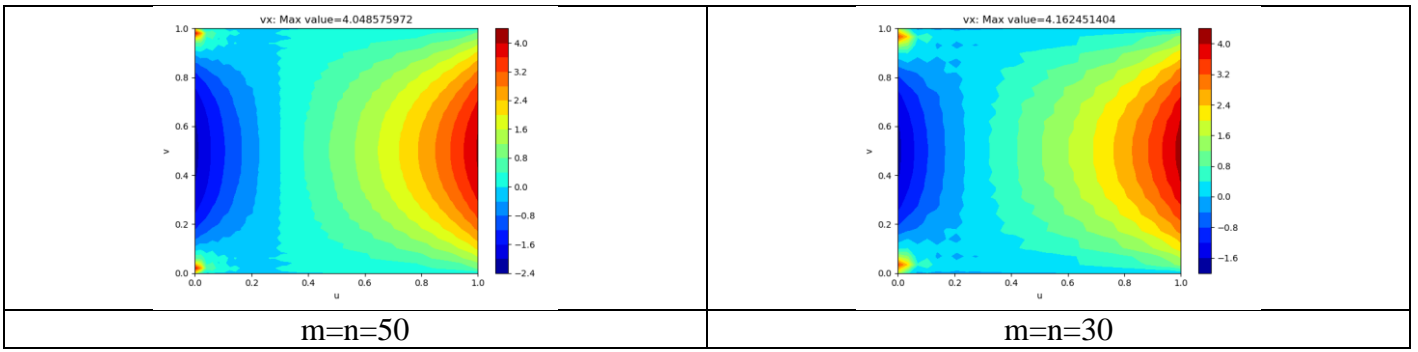


Figure 91: Test SC2 shear force vx results ($m=n=30, 50$)

Test SC3, three-point interpolation, canopy, vertical loading, square matrix

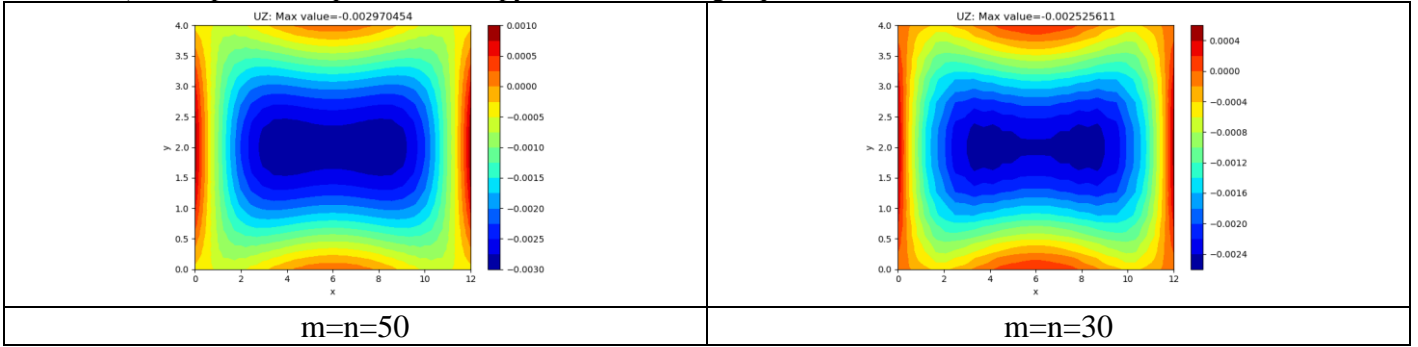


Figure 92: Test SC3 displacement uz results ($m=n=20, 30, 50$)

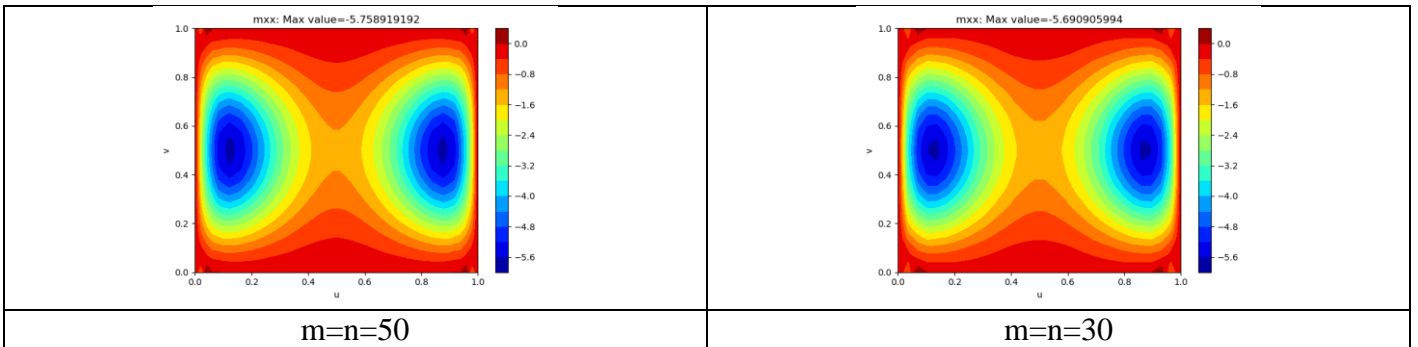


Figure 93: Test SC3 bending moment mxx results ($m=n=20, 30, 50$)

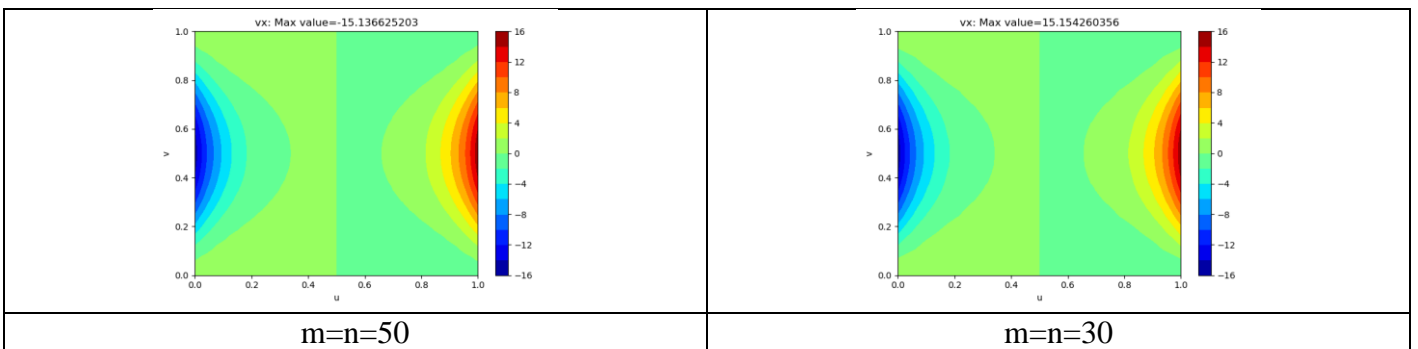


Figure 94: Test SC3 shear force vx results ($m=n=20, 30, 50$)

Test SC4, three-point interpolation, canopy, vertical loading, square matrix

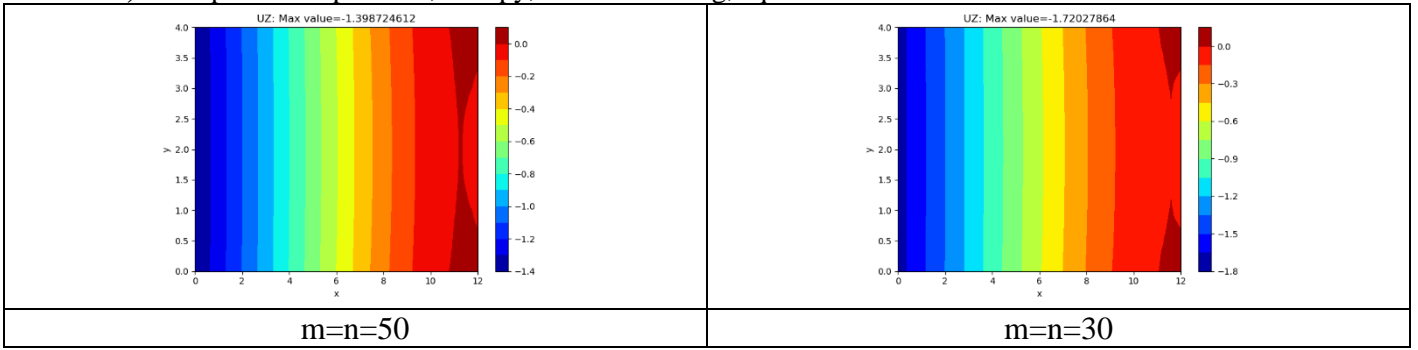


Figure 95: Test SC4 displacement uz results (m=n=20, 30, 50)

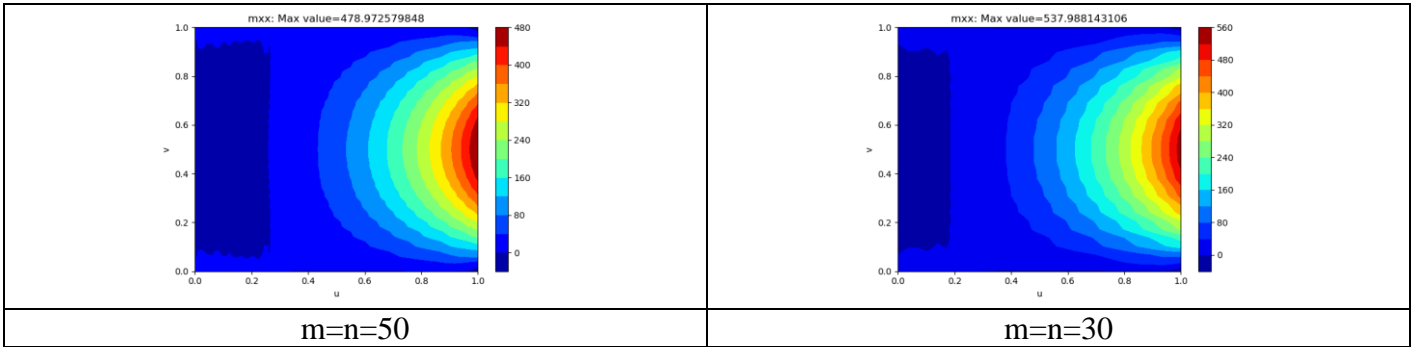


Figure 96: Test SC4 bending moment mxx results (m=n=20, 30, 50)

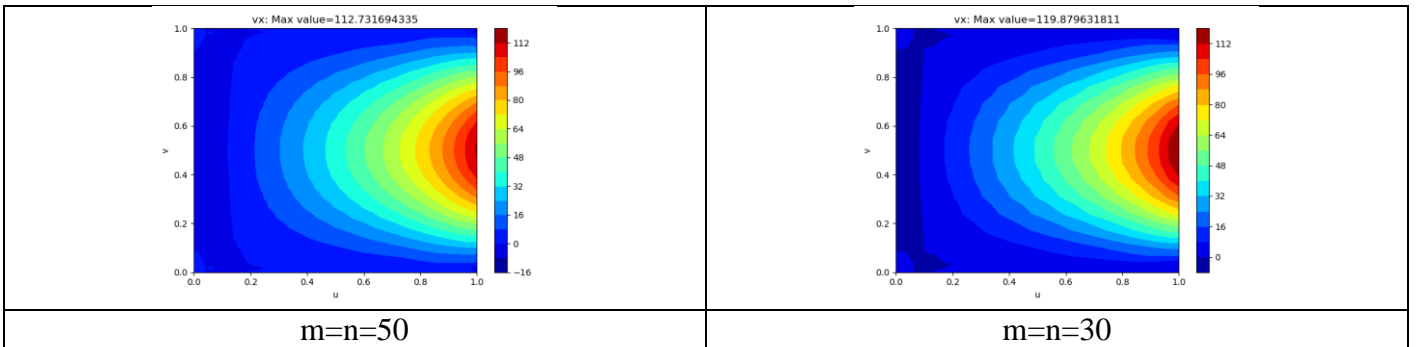


Figure 97: Test SC4 shear force vx results (m=n=20, 30, 50)

Test SC5, three-point interpolation, canopy, normal loading, square matrix

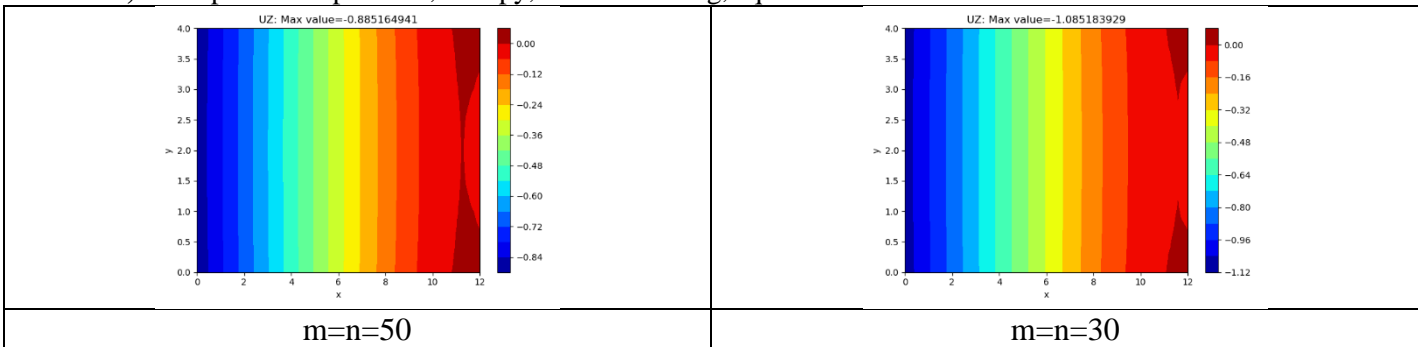


Figure 98: Test SC5 displacement uz results (m=n=20, 30, 50)

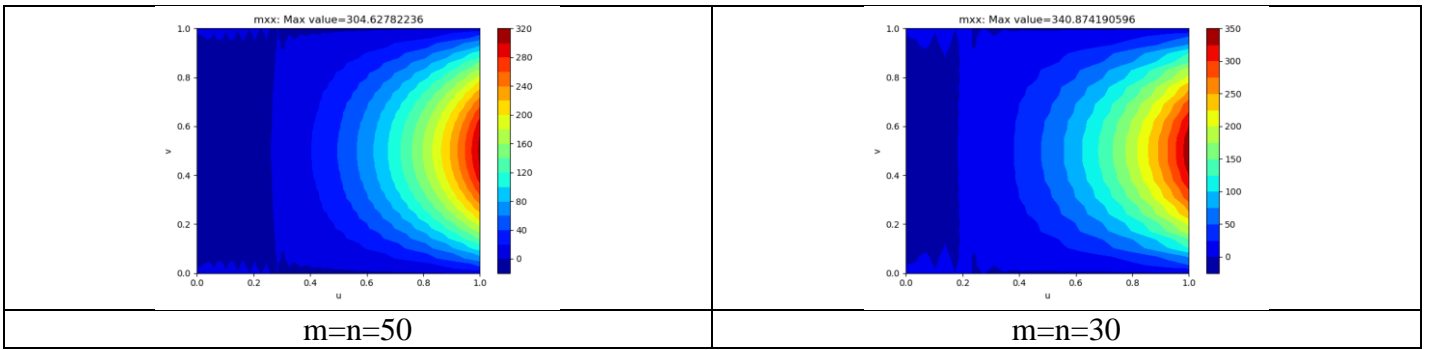


Figure 99: Test SC5 bending moment m_{xx} results ($m=n=20, 30, 50$)

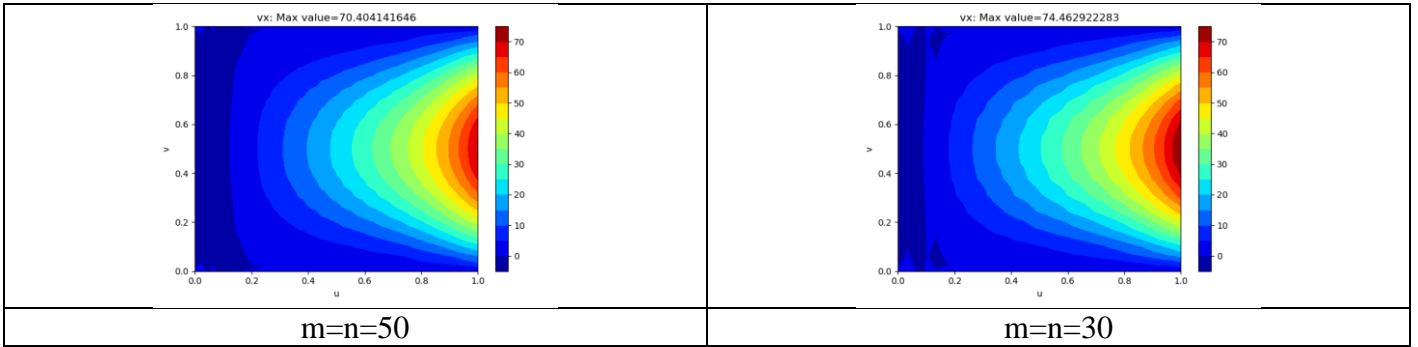


Figure 100: Test SC5 shear force v_x results ($m=n=20, 30, 50$)

Square matrix test plots

Test SC1, three-point interpolation, flat square, vertical loading, square matrix

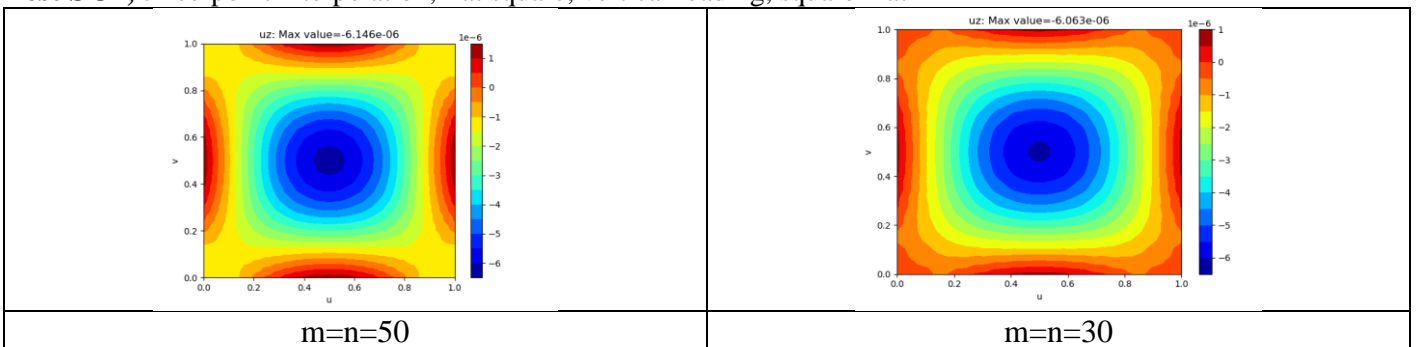


Figure 101: Test SC1 displacement u_z results ($m=n= 30, 50$)

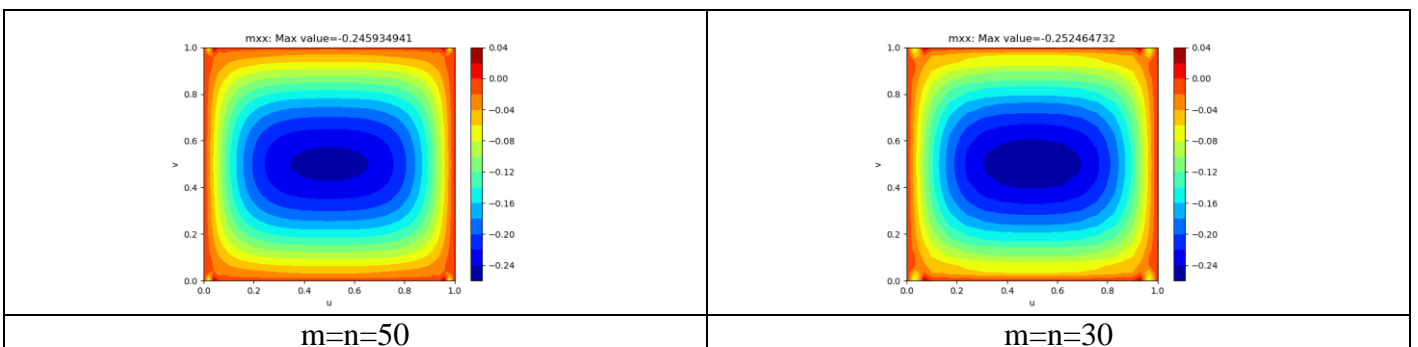


Figure 102: Test SC1 bending moment m_{xx} results ($m=n=30, 50$)

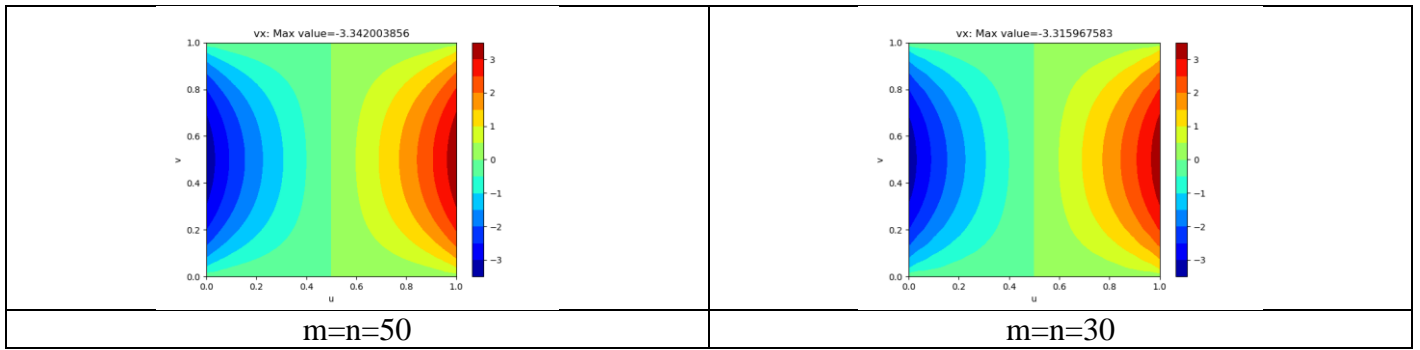


Figure 103: Test SC1 shear force v_x results ($m=n=30, 50$)

Test SC2, three-point interpolation, flat square, vertical loading, square matrix

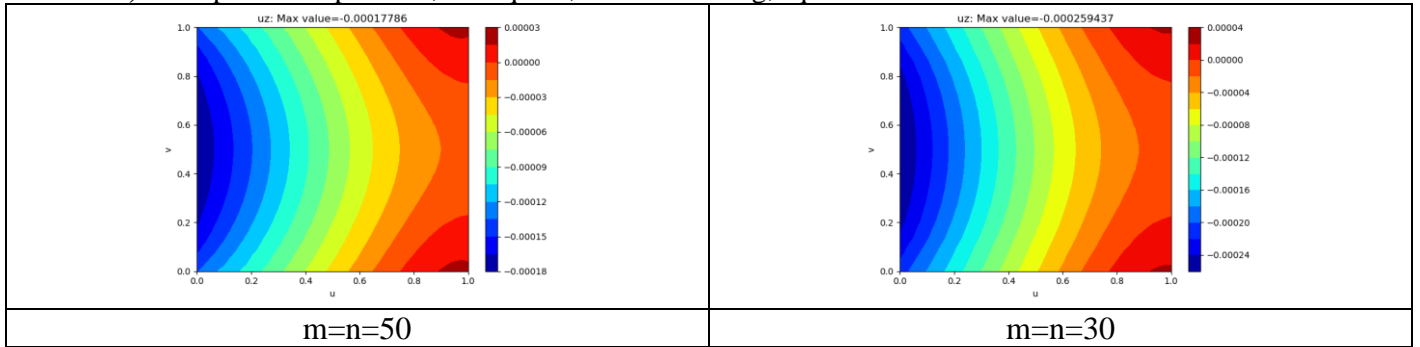


Figure 104: Test SC2 displacement u_z results ($m=n=30, 50$)

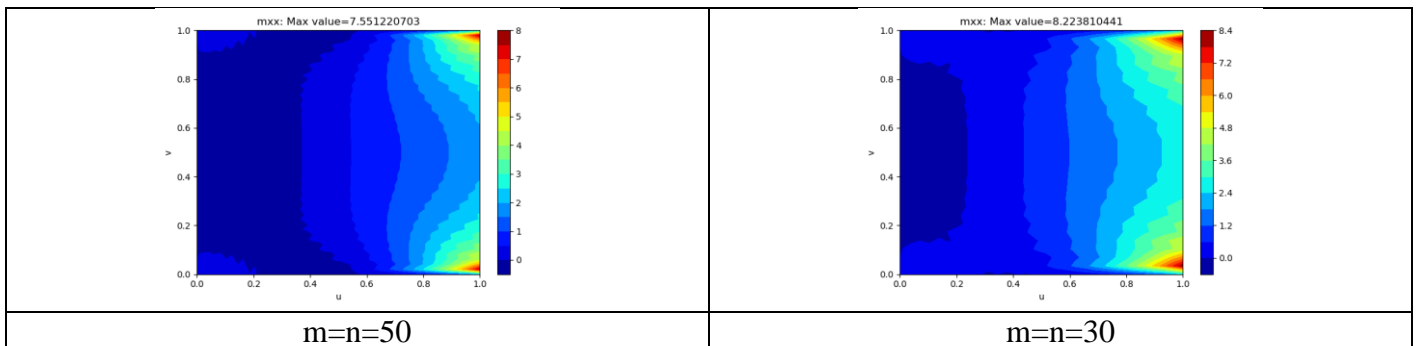


Figure 105: Test SC2 bending moment m_{xx} results ($m=n=30, 50$)



Figure 106: Test SC2 shear force v_x results ($m=n= 30, 50$)

Test SC3, three-point interpolation, canopy, vertical loading, square matrix

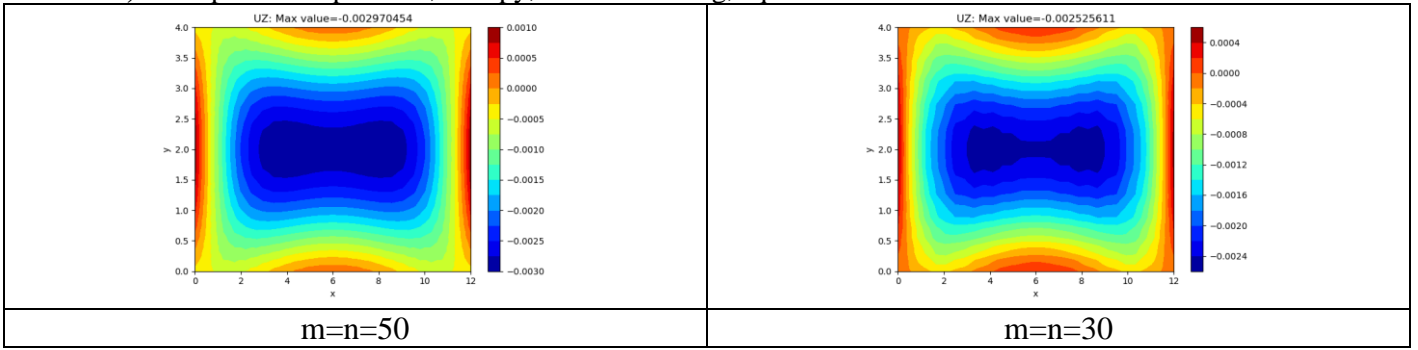


Figure 107: Test SC3 displacement uz results ($m=n=20, 30, 50$)

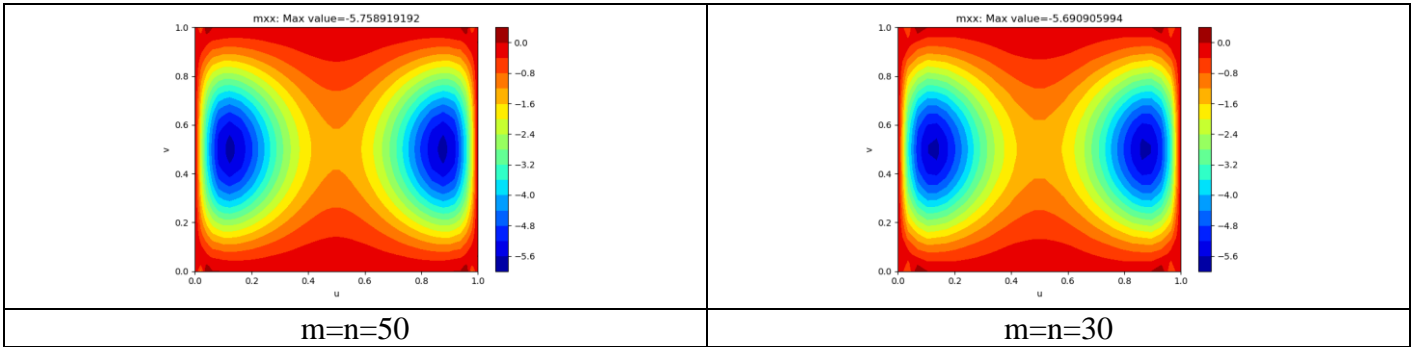


Figure 108: Test SC3 bending moment mxx results ($m=n=20, 30, 50$)

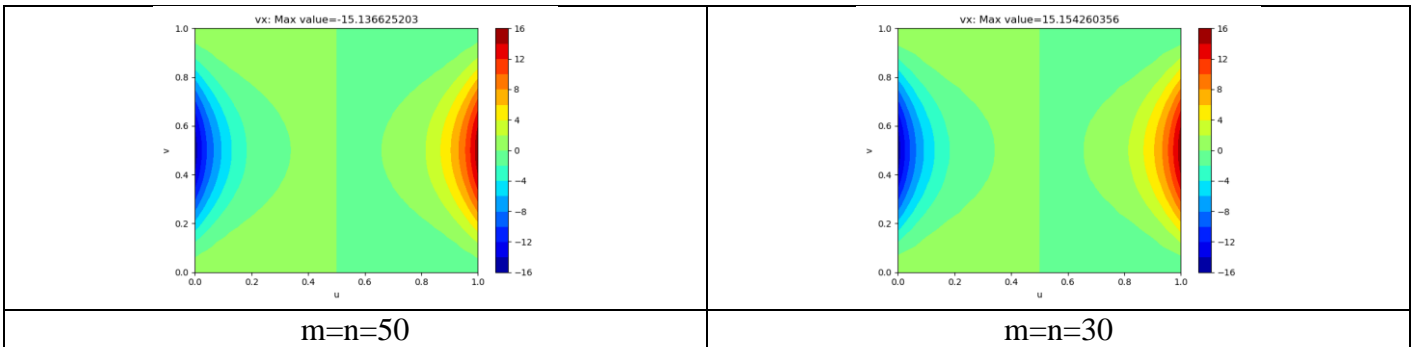


Figure 109: Test SC3 shear force vx results ($m=n=20, 30, 50$)

Test SC4, three-point interpolation, canopy, vertical loading, square matrix

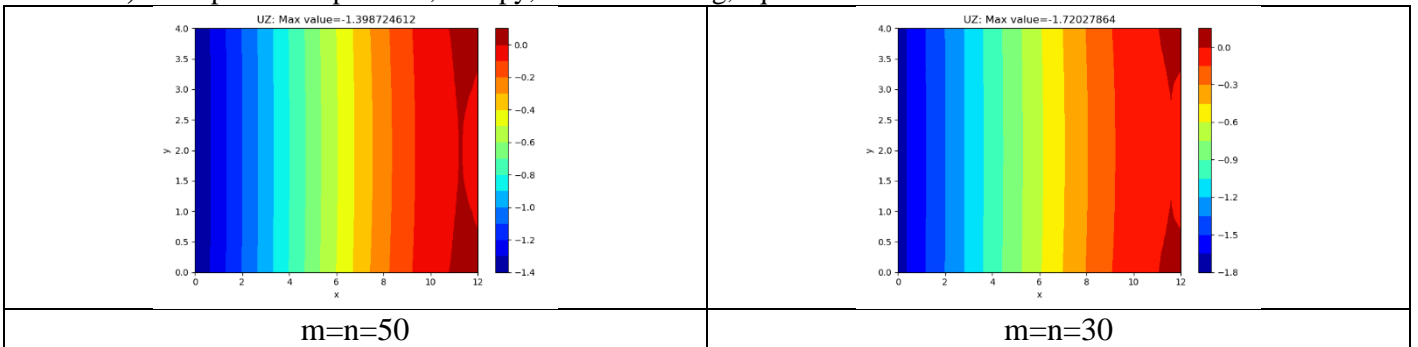


Figure 110: Test SC4 displacement uz results ($m=n=20, 30, 50$)

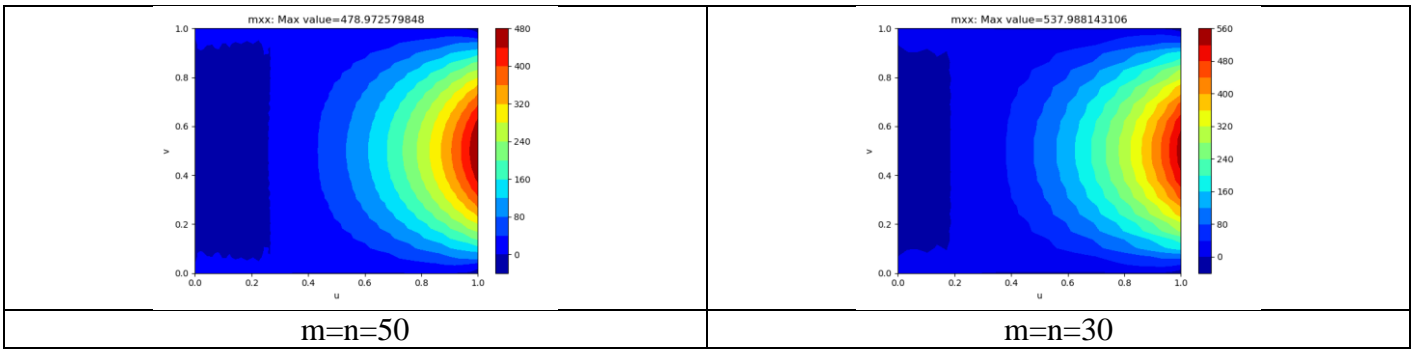


Figure 111: Test SC4 bending moment m_{xx} results ($m=n=20, 30, 50$)

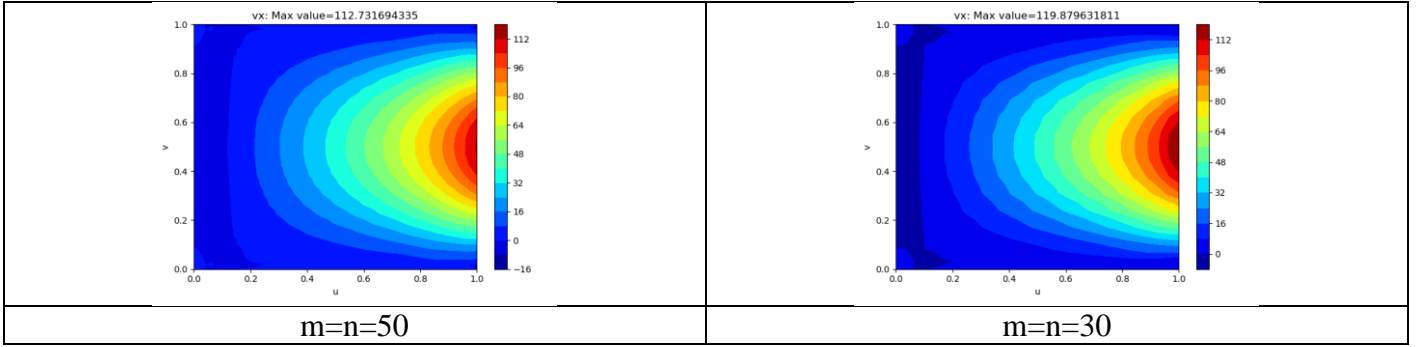


Figure 112: Test SC4 shear force v_x results ($m=n=20, 30, 50$)

Test SC5, three-point interpolation, canopy, normal loading, square matrix

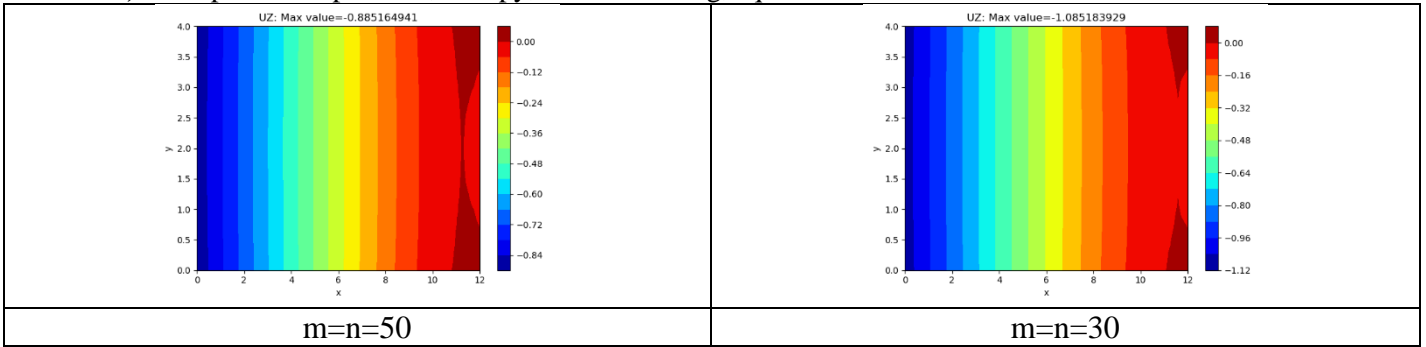


Figure 113: Test SC5 displacement u_z results ($m=n=20, 30, 50$)

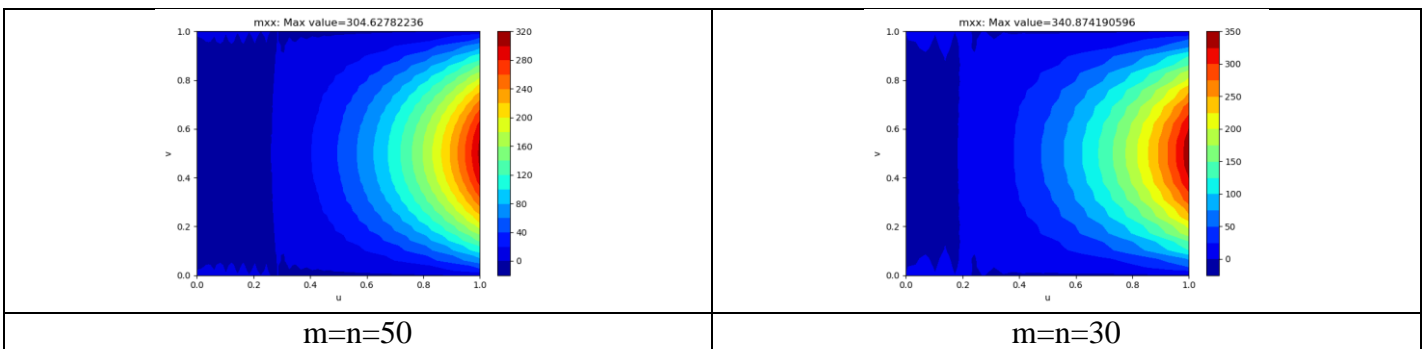


Figure 114: Test SC5 bending moment m_{xx} results ($m=n=20, 30, 50$)

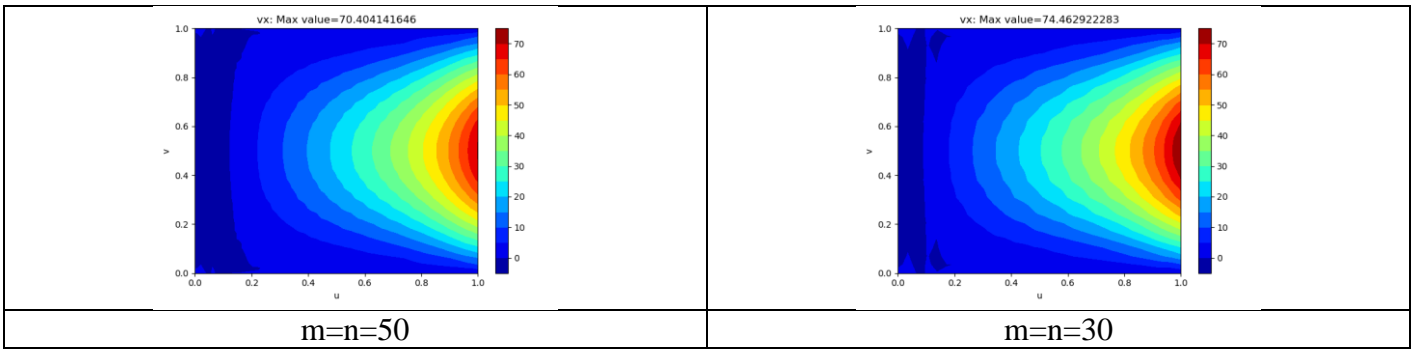


Figure 115: Test SC5 shear force v_x results ($m=n=20, 30, 50$)

Test SU1, three-point interpolation, canopy, vertical loading, square matrix

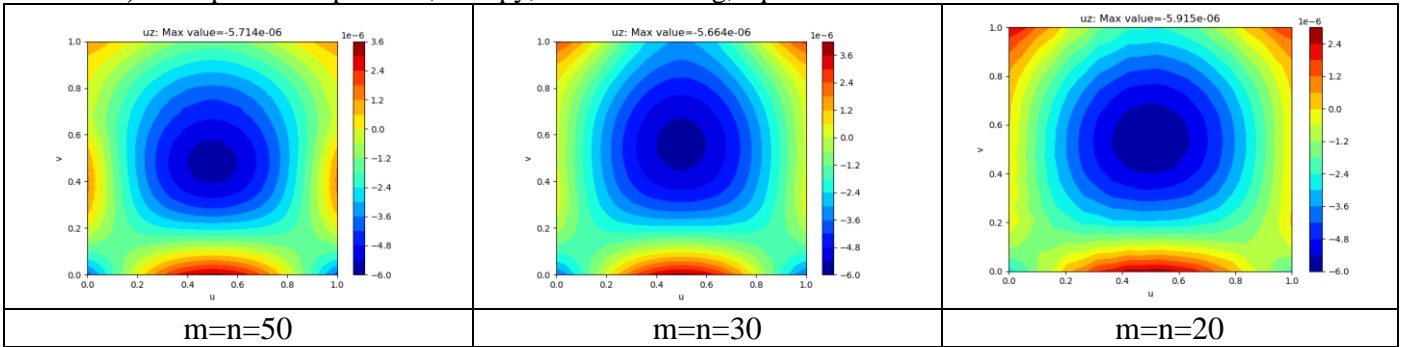


Figure 116: Test SU1 displacement u_z results ($m=n=20, 30, 50$)

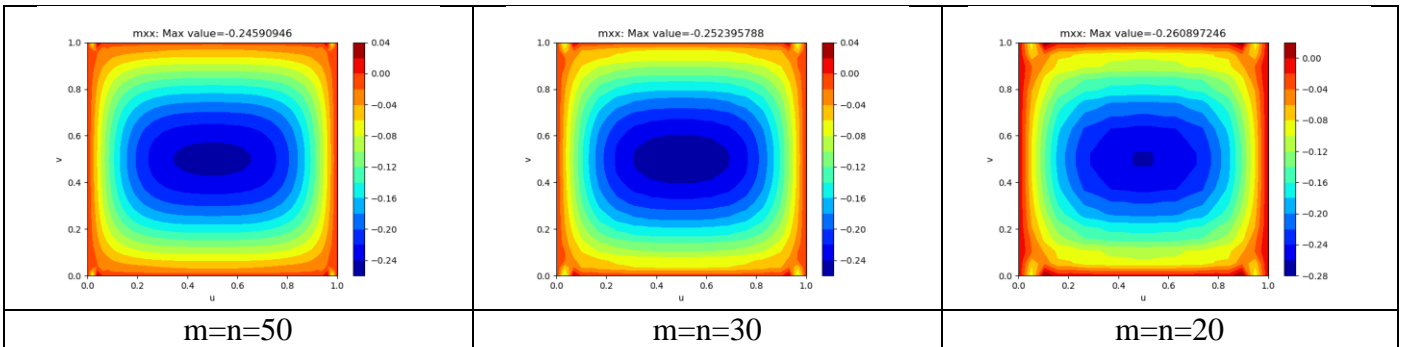


Figure 117: Test SU1 bending moment m_{xx} results ($m=n=20, 30, 50$)

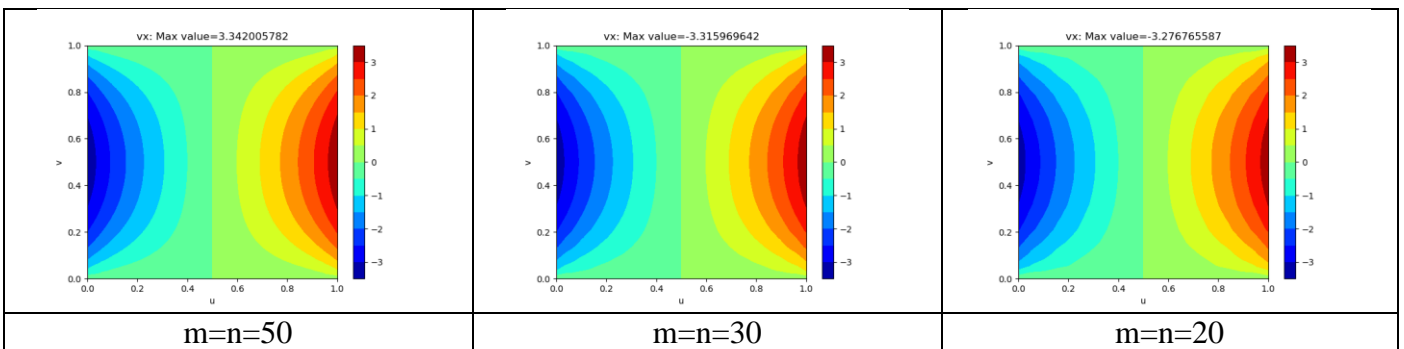


Figure 118: Test SU1 shear force v_x results ($m=n=20, 30, 50$)

Test SU2, three-point interpolation, flat square, vertical loading, square matrix

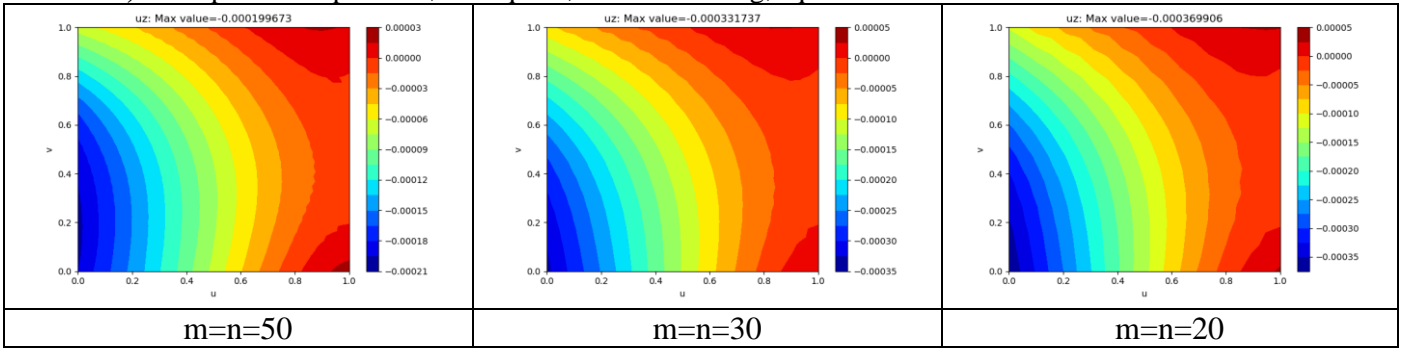


Figure 119: Test SU2 displacement u_z results ($m=n=20, 30, 50$)

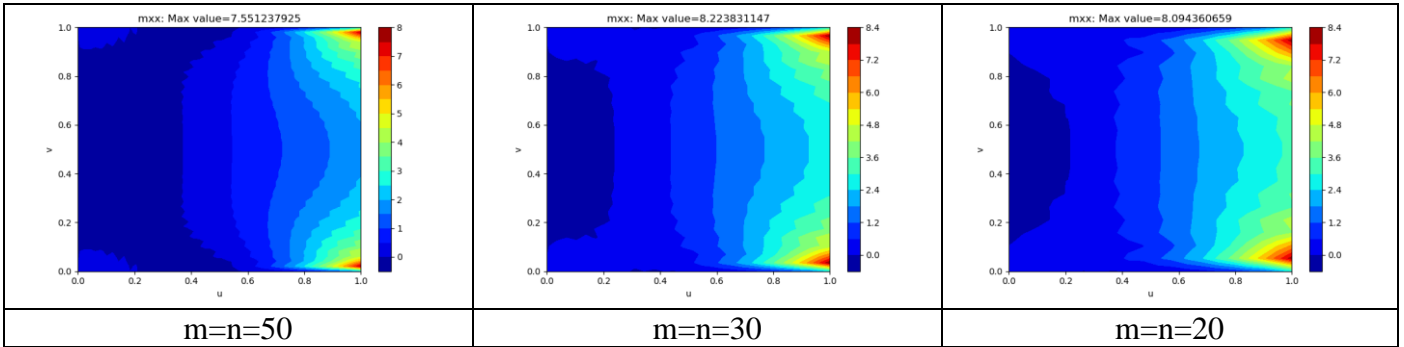


Figure 120: Test SU2 bending moment m_{xx} results ($m=n=20, 30, 50$)

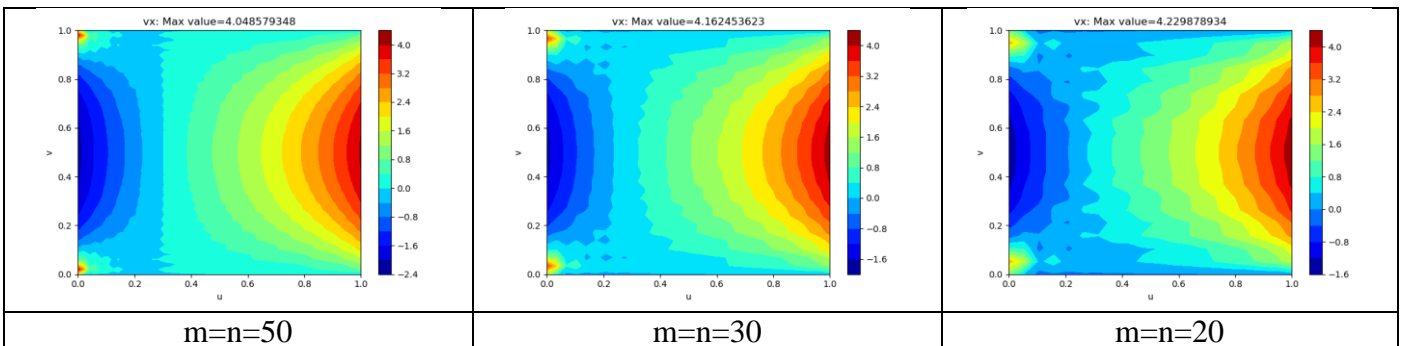


Figure 121: Test SU2 shear force v_x results ($m=n=20, 30, 50$)

Test SU3, three-point interpolation, canopy, vertical loading, square matrix

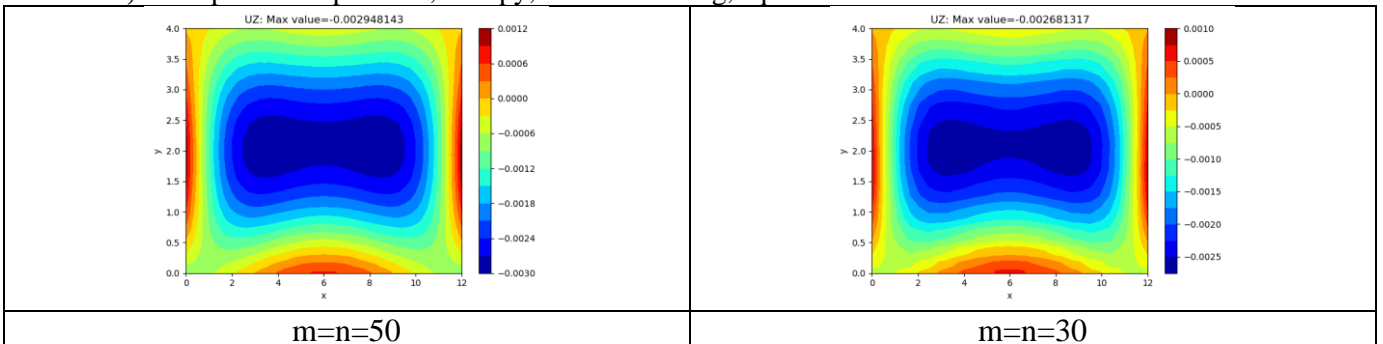


Figure 122: Test SU3 displacement u_z results ($m=n=30, 50$)

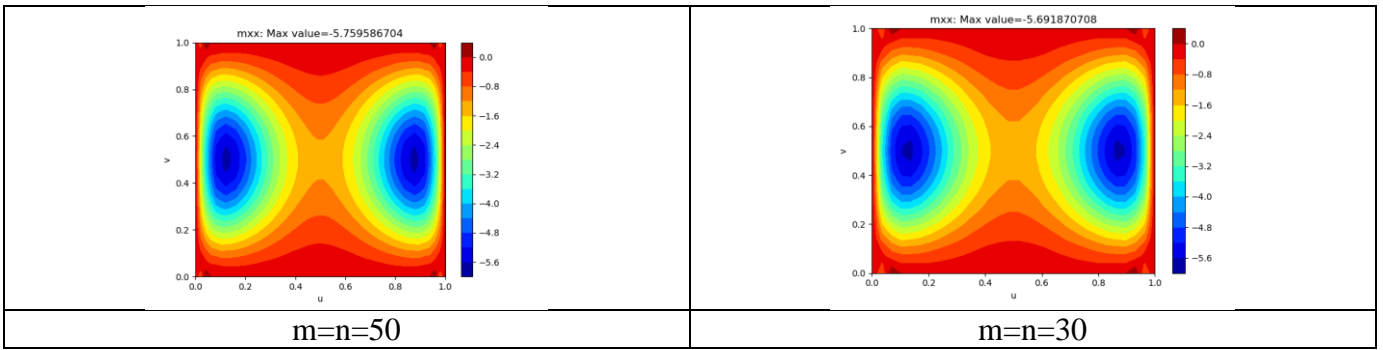


Figure 123: Test SU3 bending moment m_{xx} results ($m=n=30, 50$)

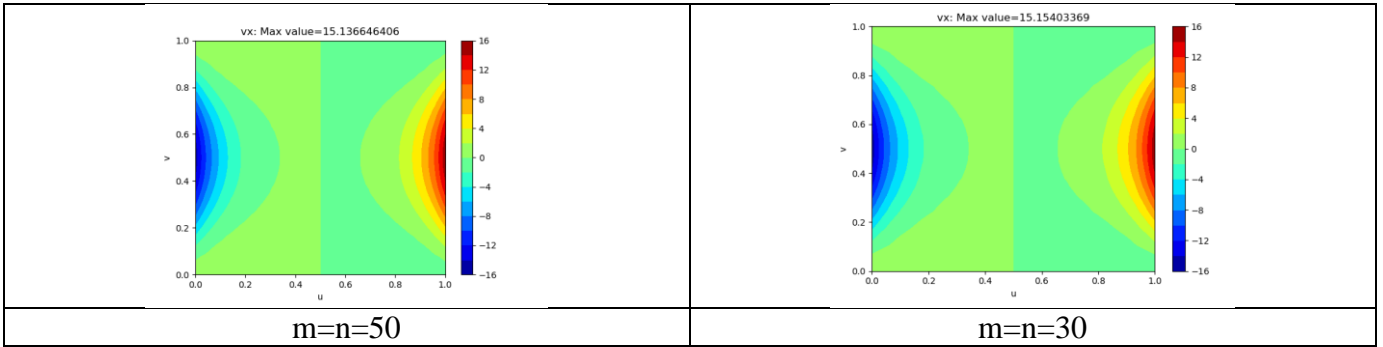


Figure 124: Test SU3 shear force v_x results ($m=n=30, 50$)

Test SU4, three-point interpolation, canopy, vertical loading, square matrix

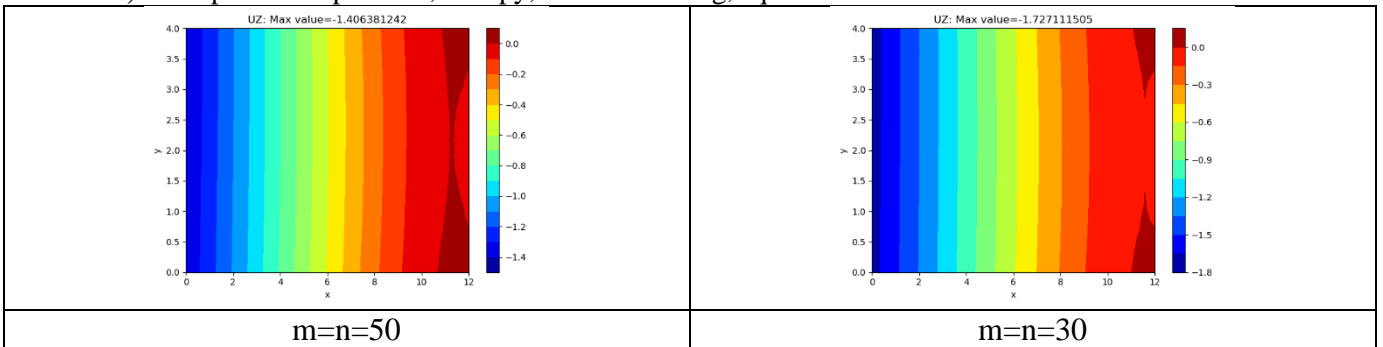


Figure 125: Test SU4 displacement u_z results ($m=n=30, 50$)

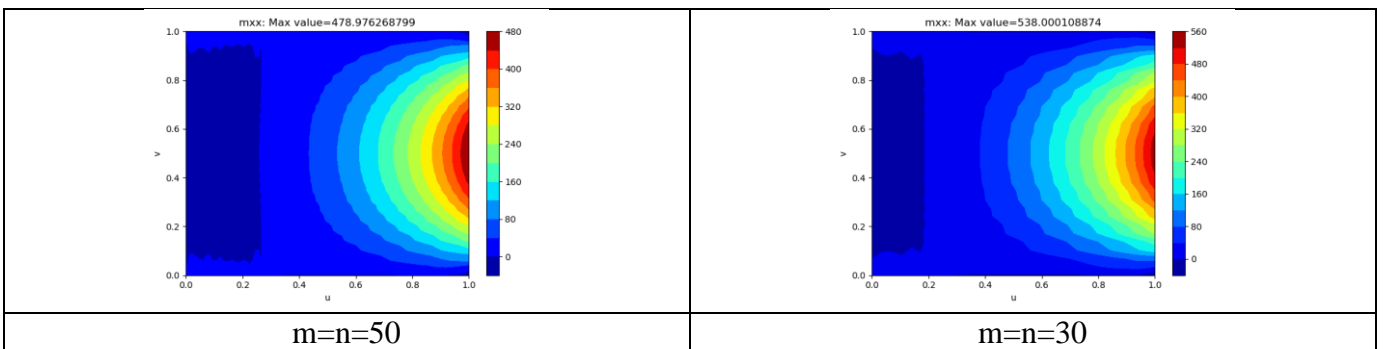


Figure 126: Test SU4 bending moment m_{xx} results ($m=n=30, 50$)

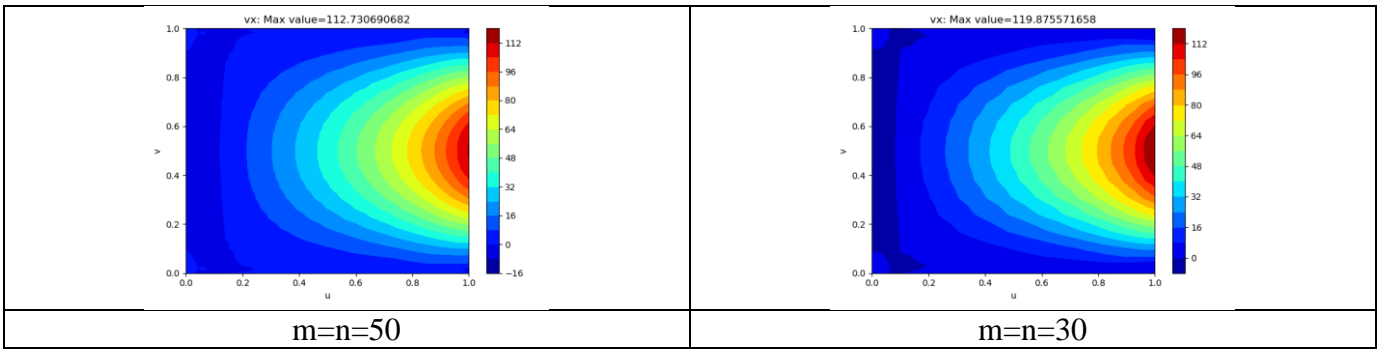


Figure 127: Test SU4 shear force vx results ($m=n=30, 50$)

Test SU5, three-point interpolation, canopy, normal loading, square matrix

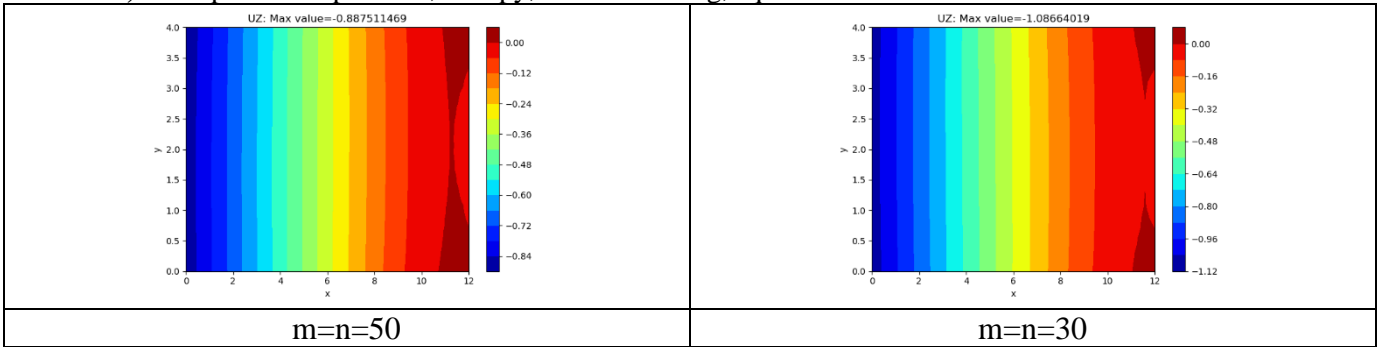


Figure 128: Test SU5 displacement uz results ($m=n=30, 50$)

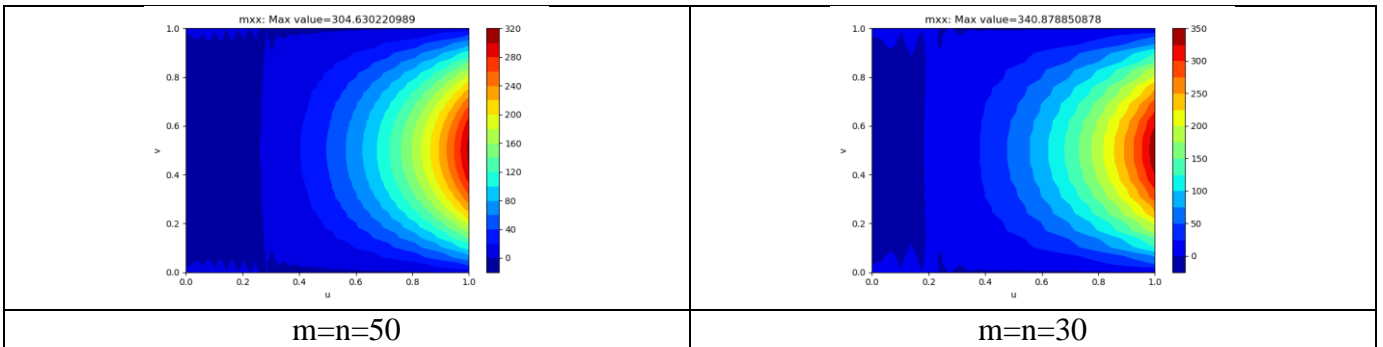


Figure 129: Test SU5 bending moment mxx results ($m=n=30, 50$)

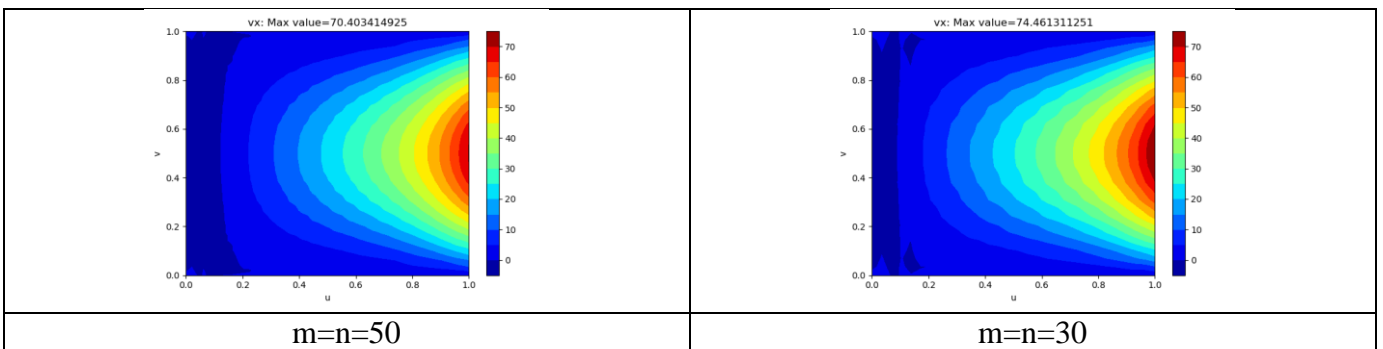


Figure 130: Test SU5 shear force vx results ($m=n=30, 50$)

Test SE1, three-point interpolation, flat square, vertical loading, square matrix

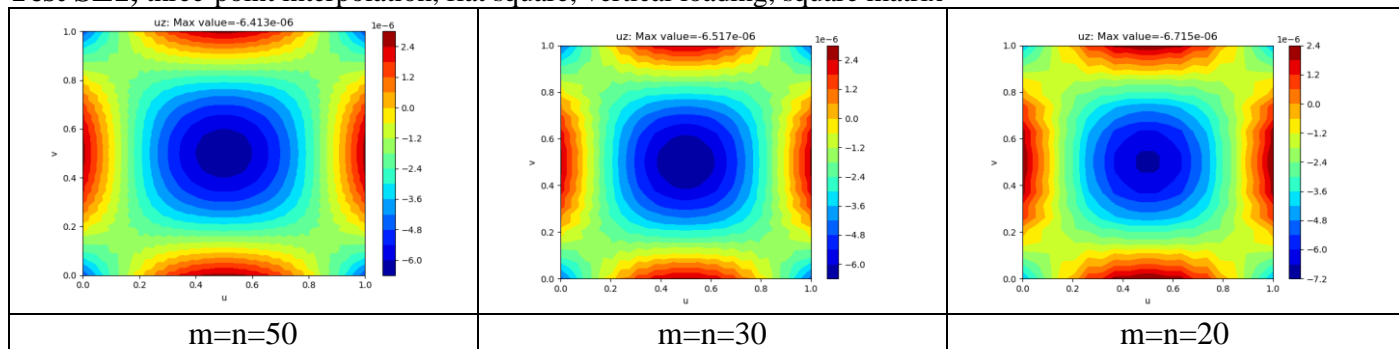


Figure 131: Test SU1 displacement uz results ($m=n=20, 30, 50$)

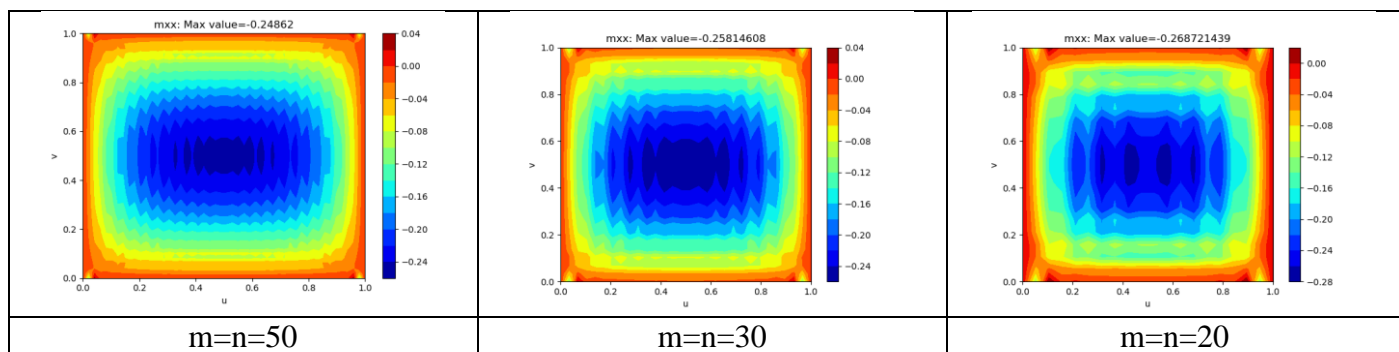


Figure 132: Test SU1 bending moment mxx results ($m=n=20, 30, 50$)

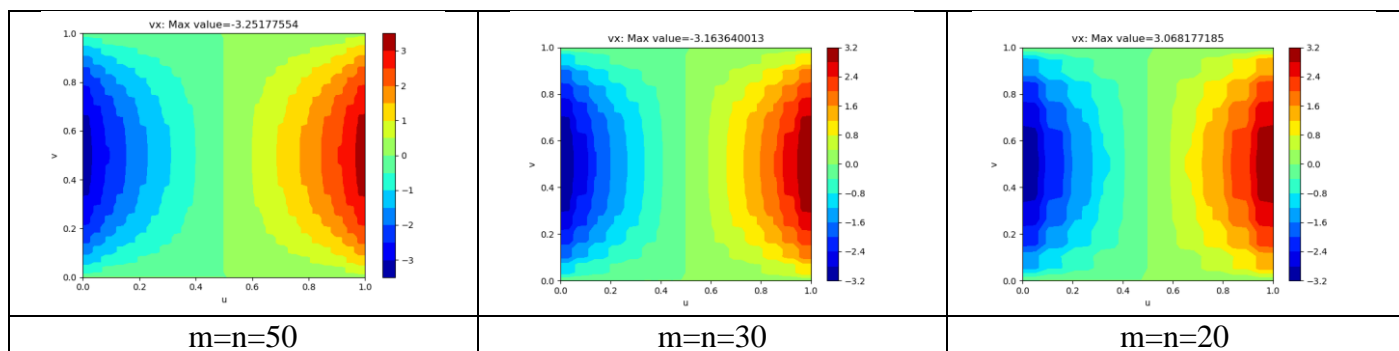


Figure 133: Test SE1 shear force vx results ($m=n=20, 30, 50$)

Test SE2, three-point interpolation, flat square, vertical loading, square matrix

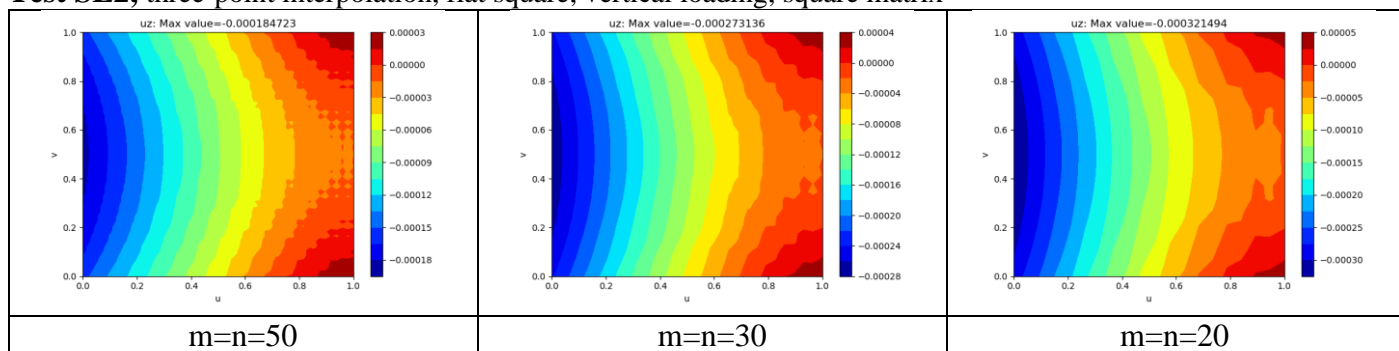


Figure 134: Test SE2 displacement uz results ($m=n=20, 30, 50$)

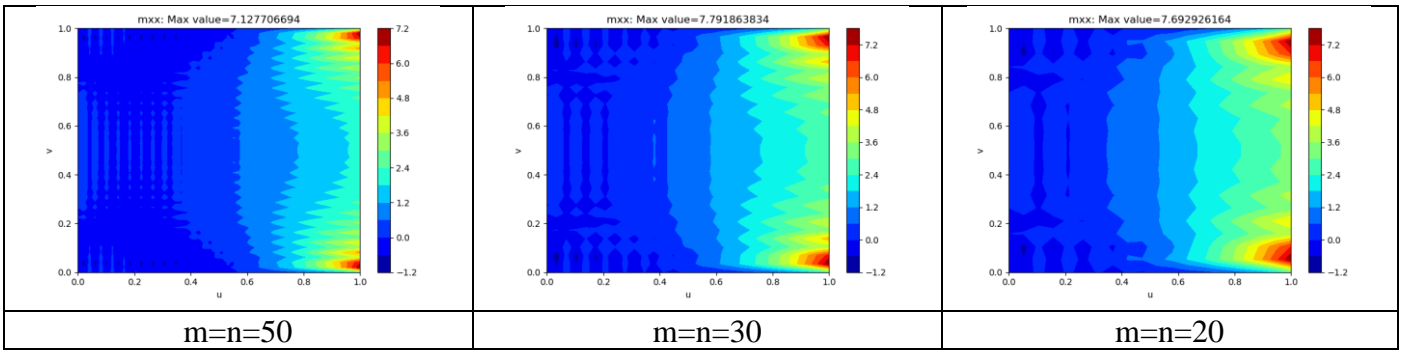


Figure 135: Test SE2 bending moment m_{xx} results ($m=n=20, 30, 50$)

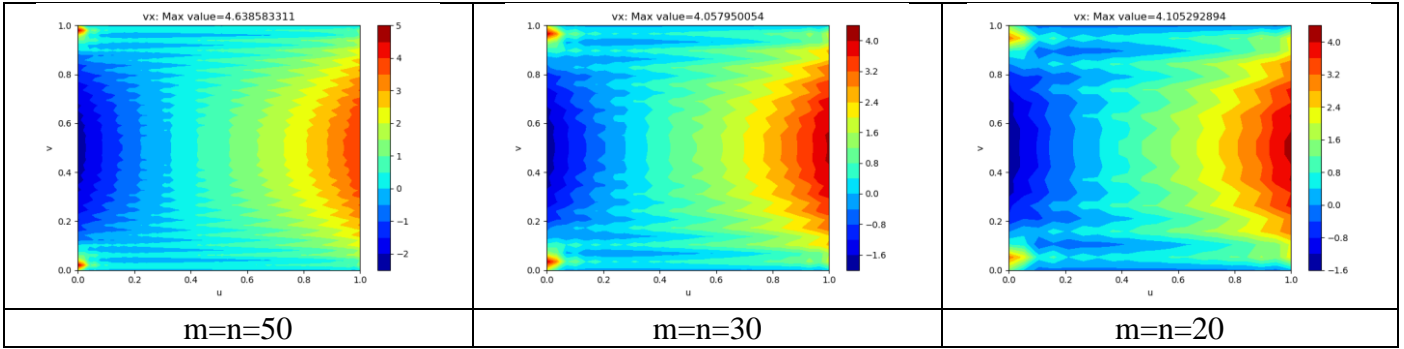


Figure 136: Test SE2 shear force v_x results ($m=n=20, 30, 50$)

Test SE3, three-point interpolation, canopy, vertical loading, square matrix

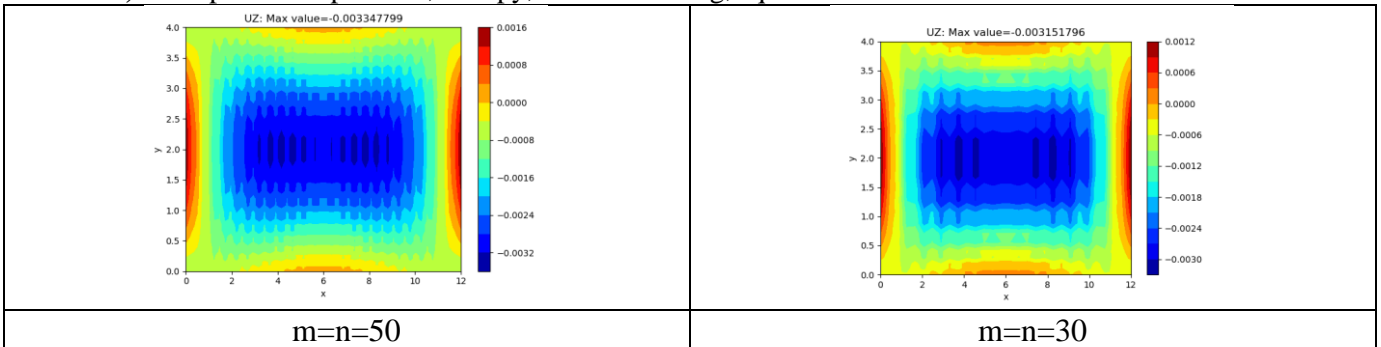


Figure 137: Test SE3 displacement u_z results ($m=n=30, 50$)

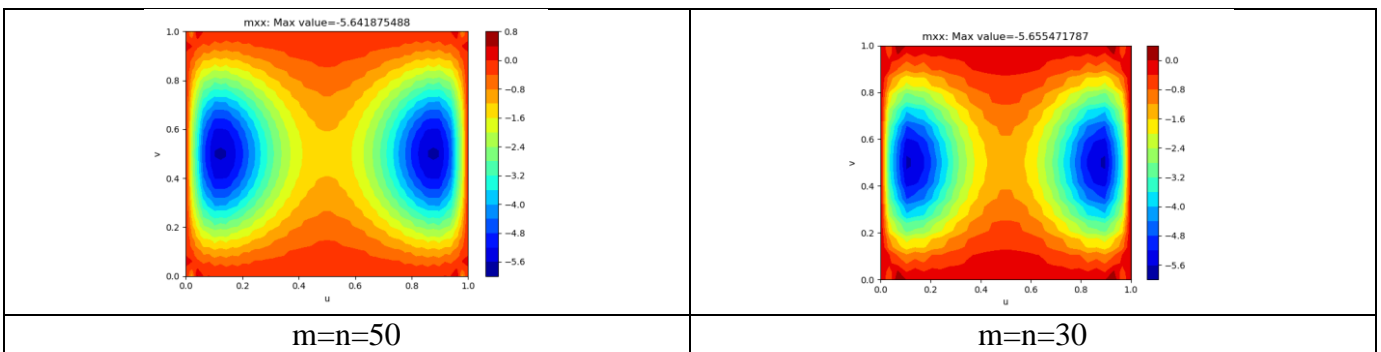


Figure 138: Test SE3 bending moment m_{xx} results ($m=n=30, 50$)

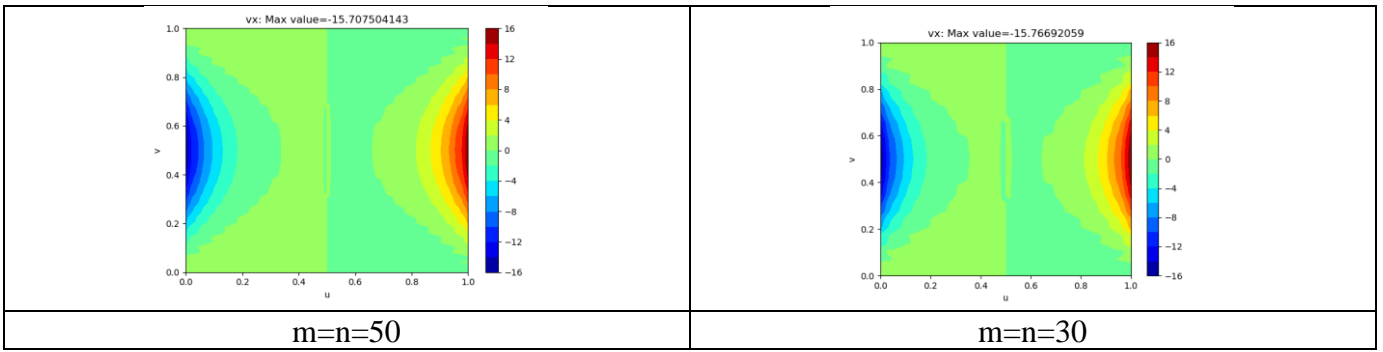


Figure 139: Test SE3 shear force v_x results ($m=n=30, 50$)

Test SE4, three-point interpolation, canopy, vertical loading, square matrix

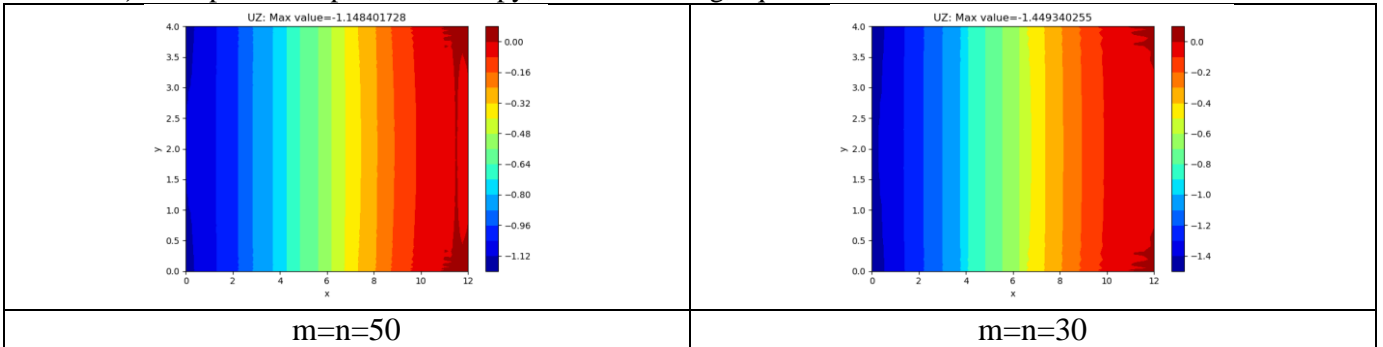


Figure 140: Test SE4 displacement u_z results ($m=n=30, 50$)

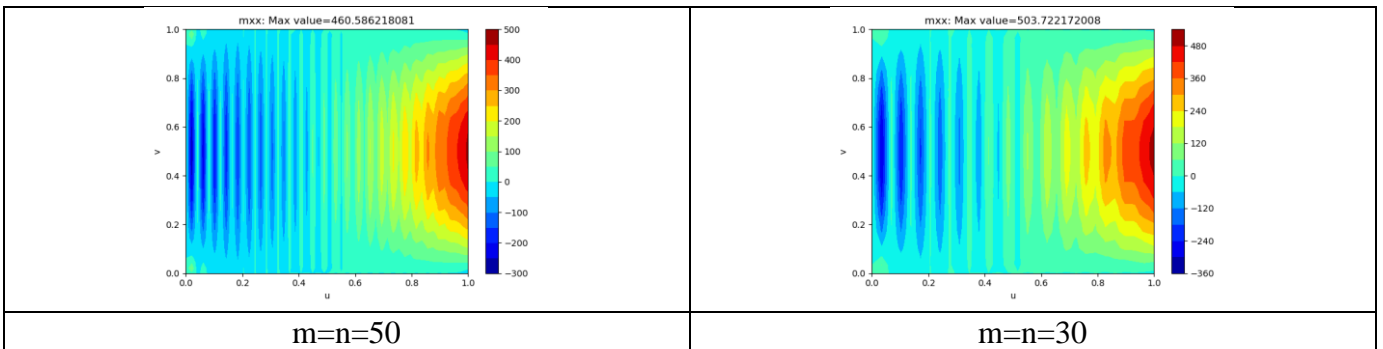


Figure 141: Test SE4 bending moment m_{xx} results ($m=n=30, 50$)



Figure 142: Test SE4 shear force v_x results ($m=n= 30, 50$)

Test SE5, three-point interpolation, canopy, normal loading, square matrix

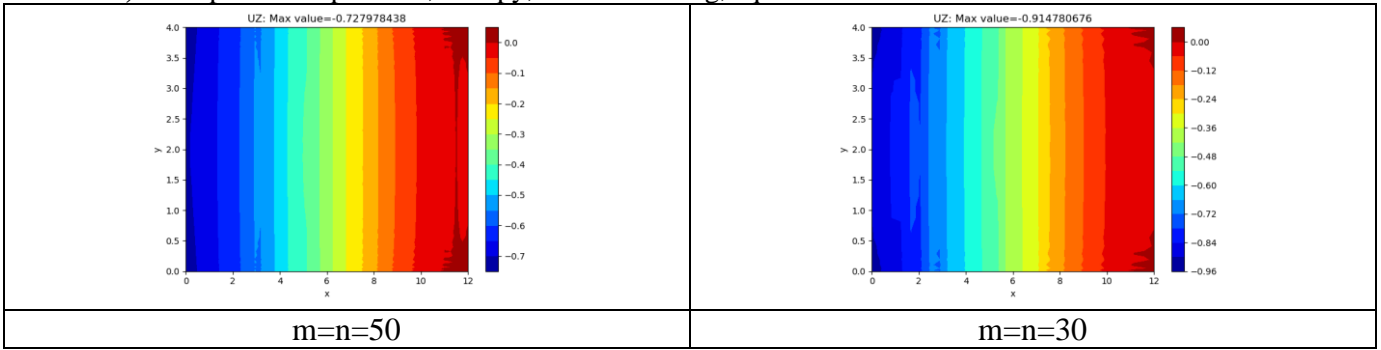


Figure 143: Test SE5 displacement uz results ($m=n=30, 50$)

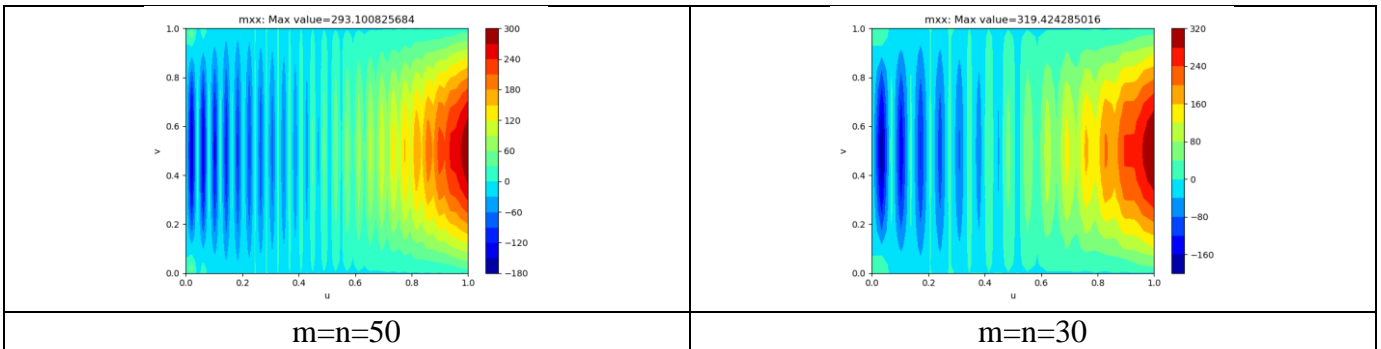


Figure 144: Test SE5 bending moment mxx results ($m=n=30, 50$)

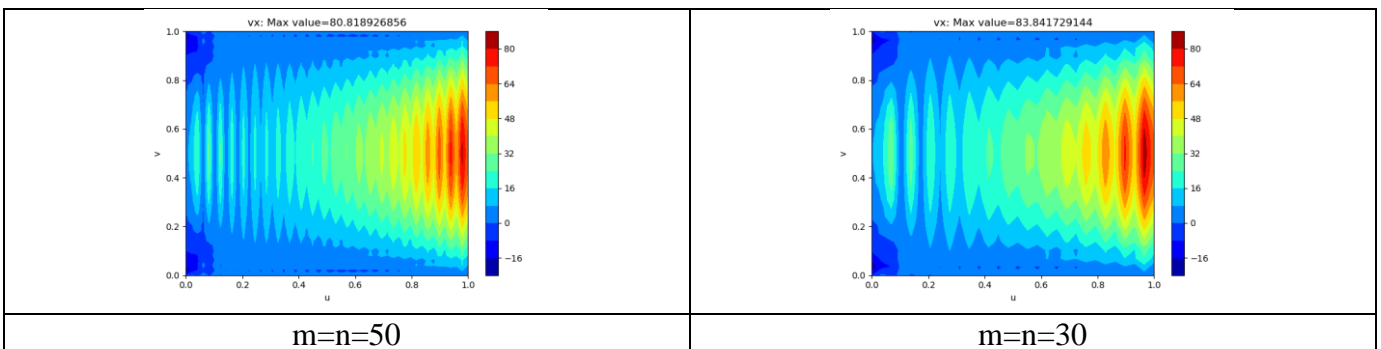


Figure 145: Test SE5 shear force vx results ($m=n= 30, 50$)

Solver test results

Test P1, three-point interpolation, flat square, vertical loading, rectangular matrix

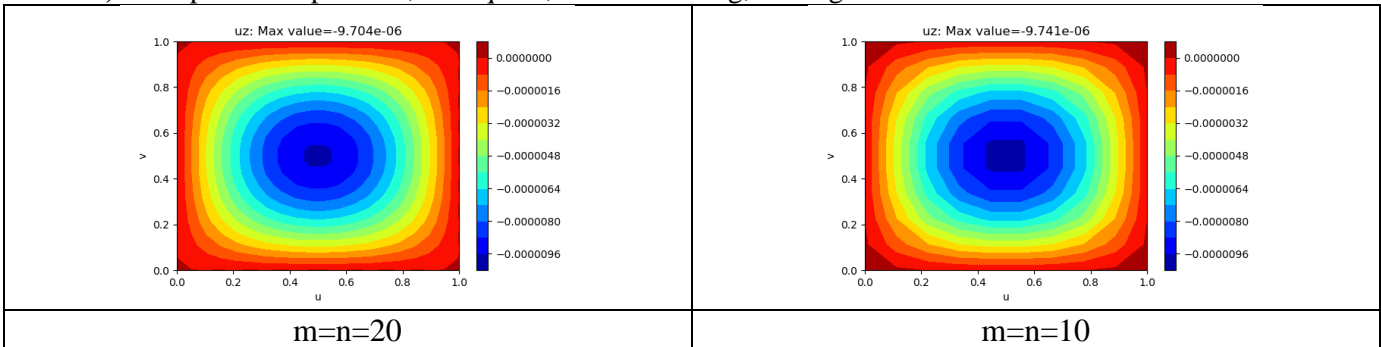


Figure 146: Test P1 displacement uz results ($m=n=10, 20$)

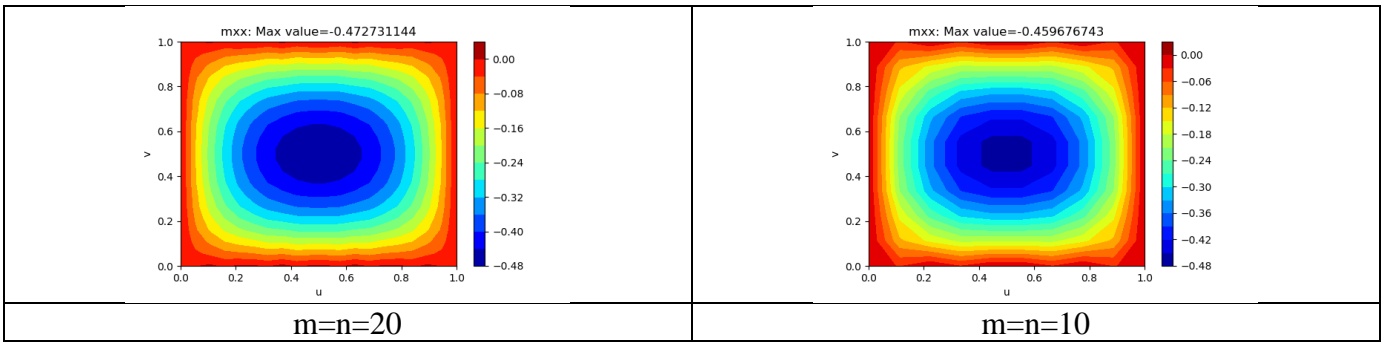


Figure 147: Test P1 bending moment m_{xx} results ($m=n=10, 20$)

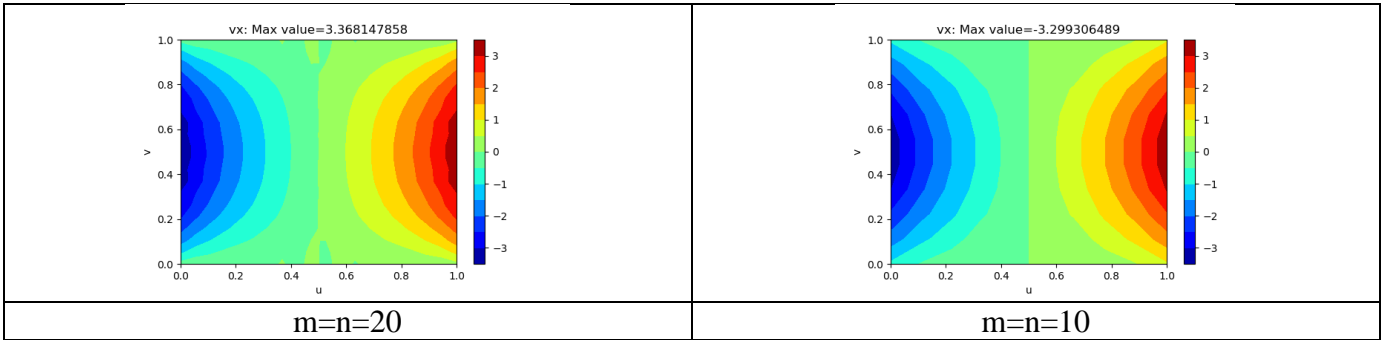


Figure 148: Test P1 shear force v_x results ($m=n=10, 20$)

Test P2, three-point interpolation, flat square, vertical loading, rectangular matrix

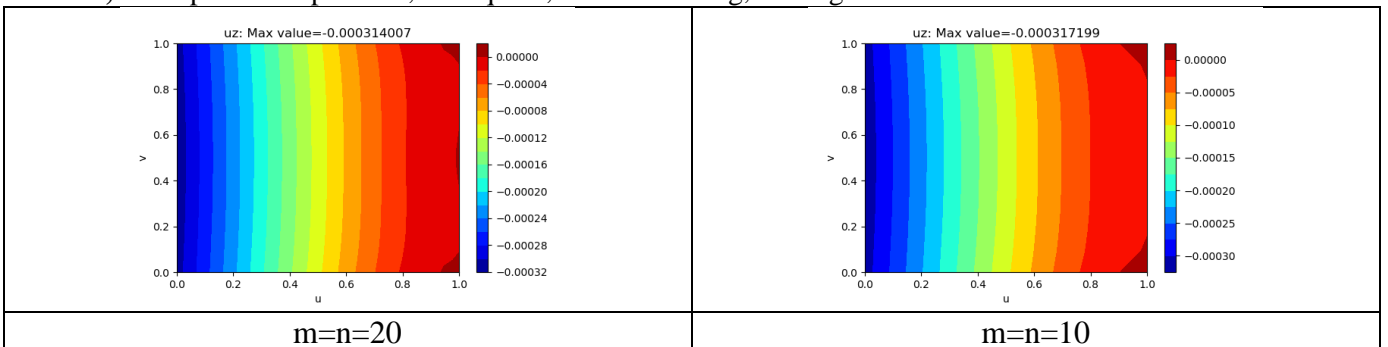


Figure 149: Test P2 displacement u_z results ($m=n=10, 20$)

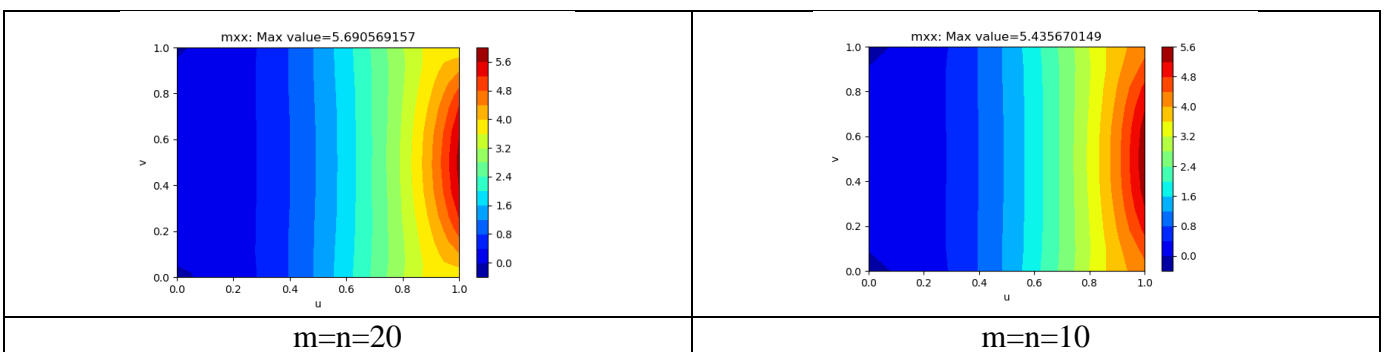
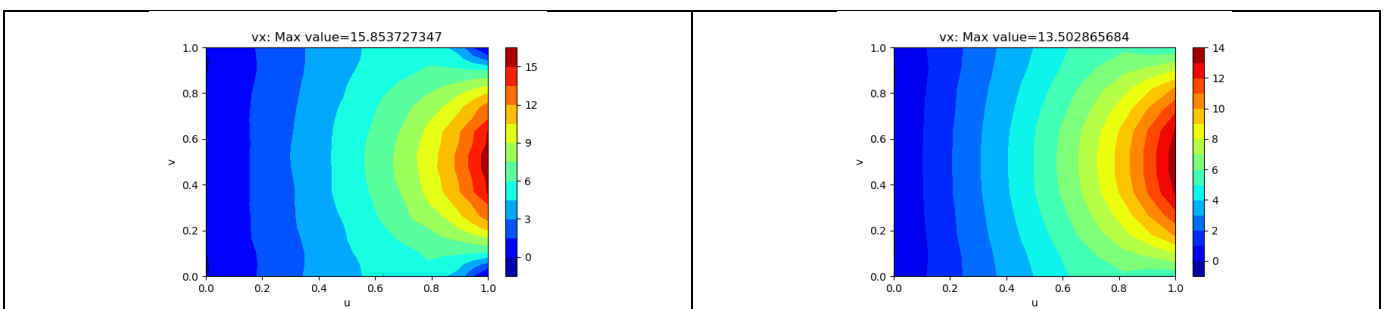


Figure 150: Test P2 bending moment m_{xx} results ($m=n=10, 20$)



$m=n=20$	$m=n=10$
----------	----------

Figure 151: Test P2 shear force vx results ($m=n=10, 20$)

Test P3, three-point interpolation, canopy, vertical loading, rectangular matrix

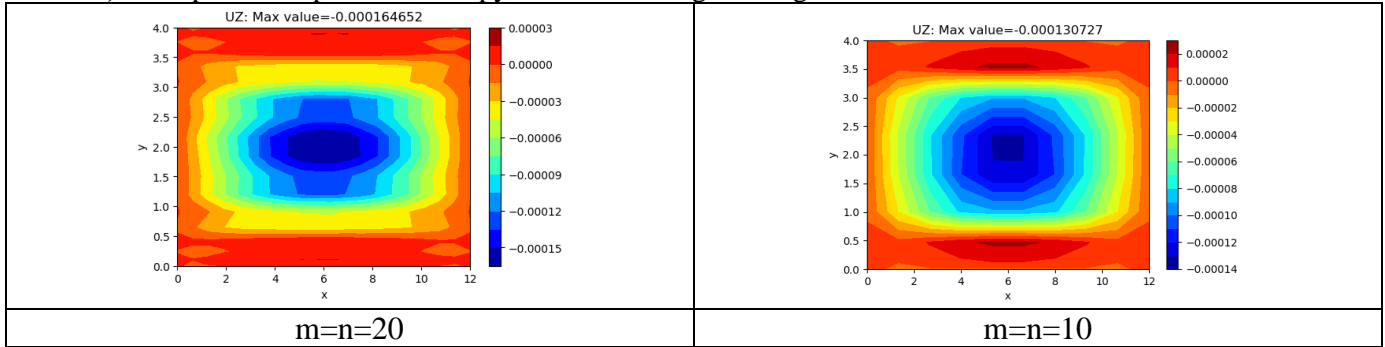


Figure 152: Test P3 displacement uz results ($m=n=10, 20$)

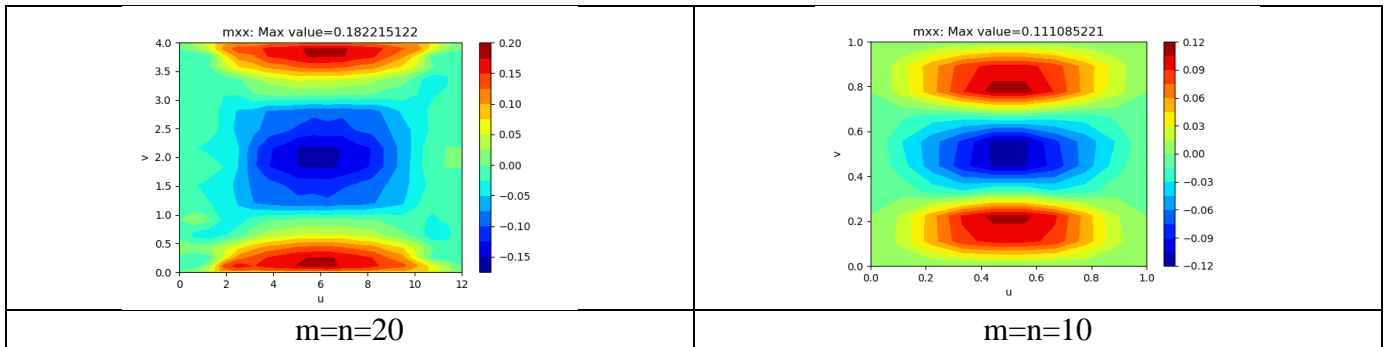


Figure 153: Test P3 bending moment mxx results ($m=n=10, 20$)

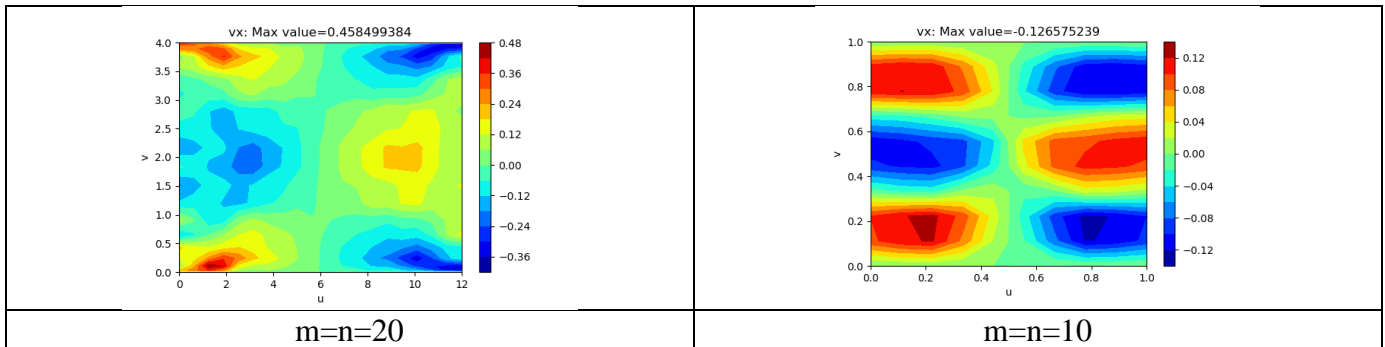


Figure 154: Test P3 shear force vx results ($m=n=10, 20$)

Test P4, three-point interpolation, canopy, vertical loading, rectangular matrix

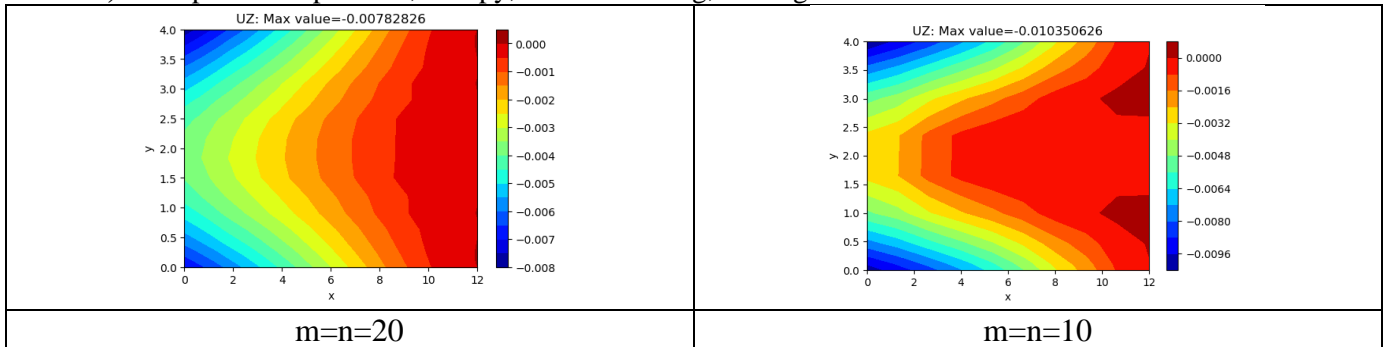


Figure 155: Test P4 displacement uz results ($m=n=10, 20$)

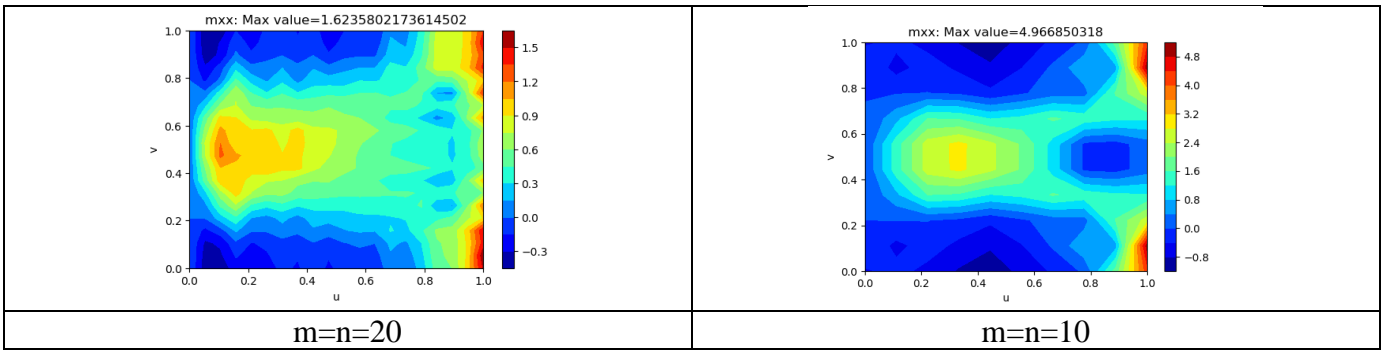


Figure 156: Test P4 bending moment m_{xx} results ($m=n=10, 20$)

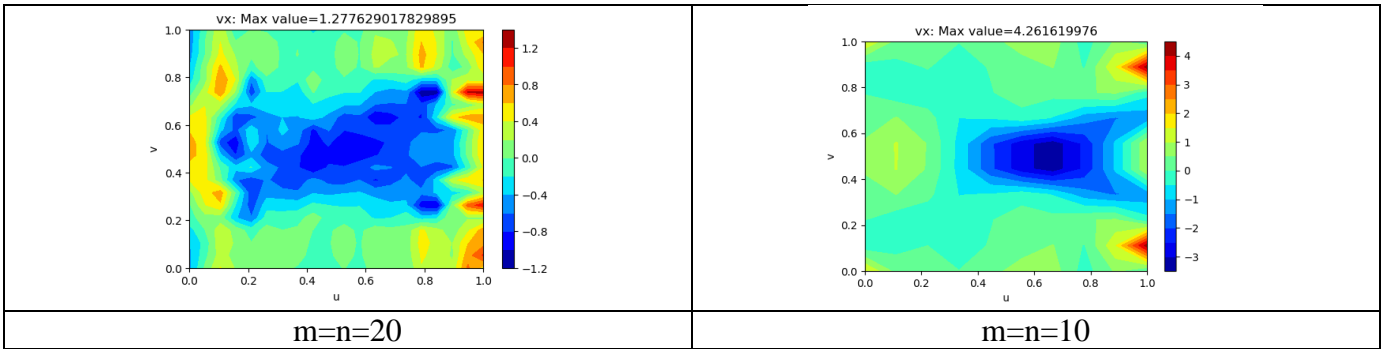


Figure 157: Test P4 shear force v_x results ($m=n=10, 20$)

Test P5, three-point interpolation, canopy, normal loading, rectangular matrix



Figure 158: Test P5 displacement u_z results ($m=n=10, 20, 30$)

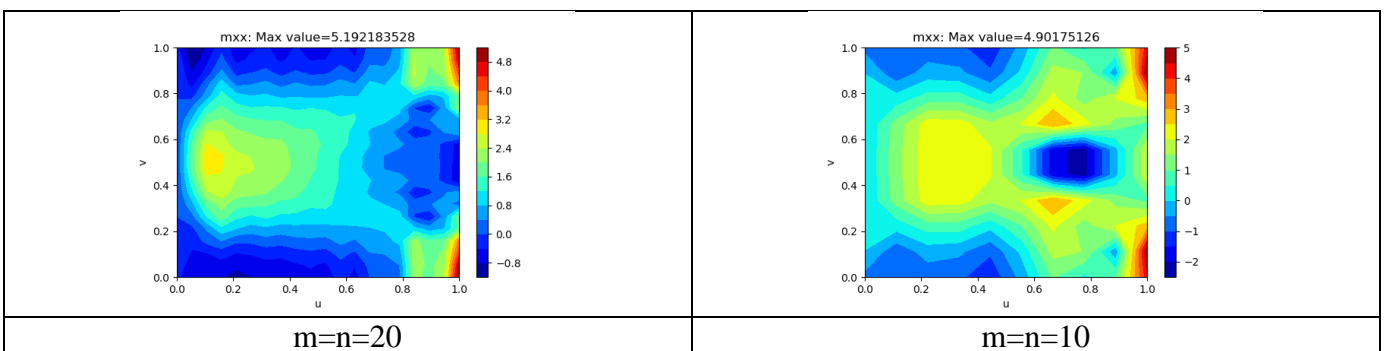


Figure 159: Test P5 bending moment m_{xx} results ($m=n=10, 20$)

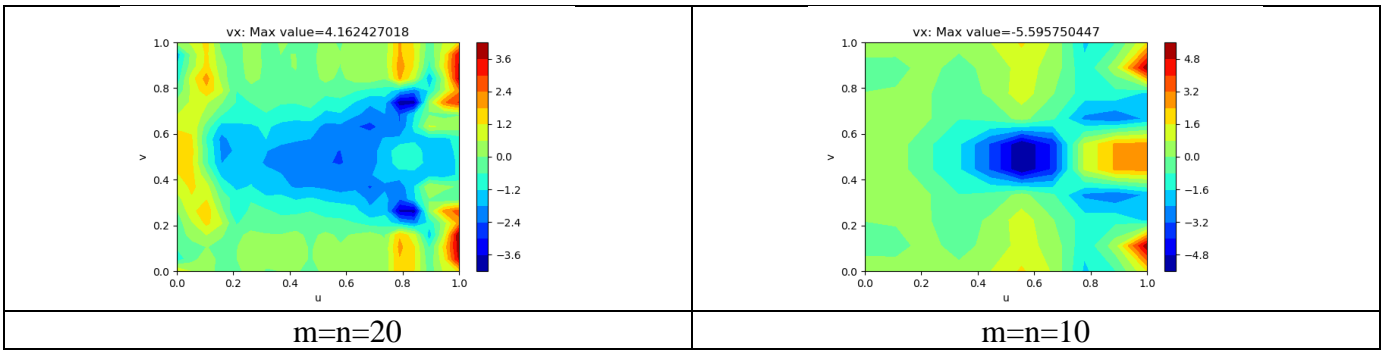


Figure 160: Test P5 shear force v_x results ($m=n=10, 20$)

Test LM1, three-point interpolation, flat square, vertical loading, rectangular matrix

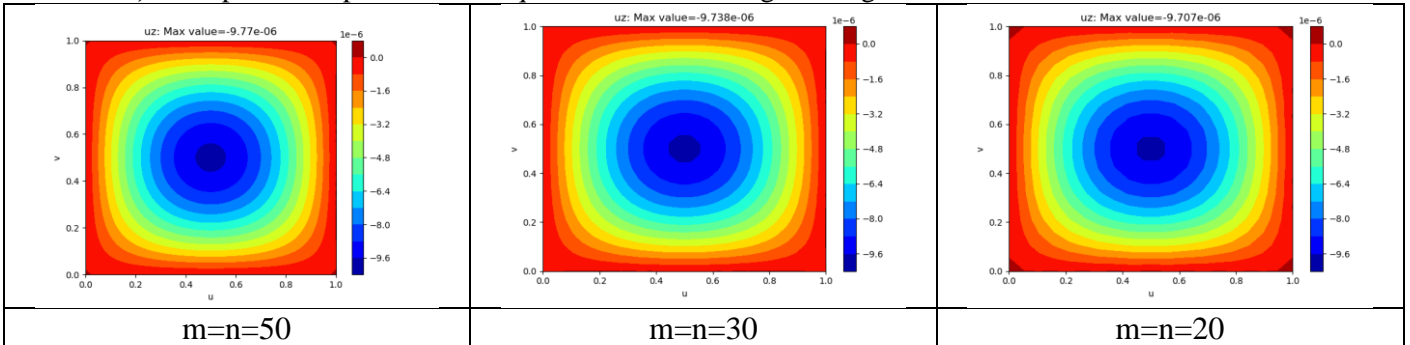


Figure 161: Test LM1 displacement u_z results ($m=n= 20, 30, 50$)

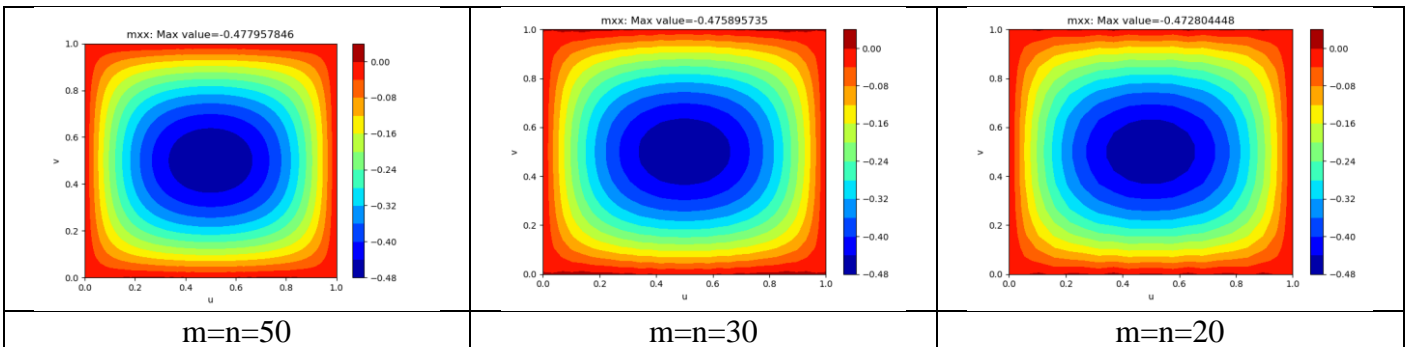


Figure 162: Test LM1 bending moment m_{xx} results ($m=n= 20, 30, 50$)

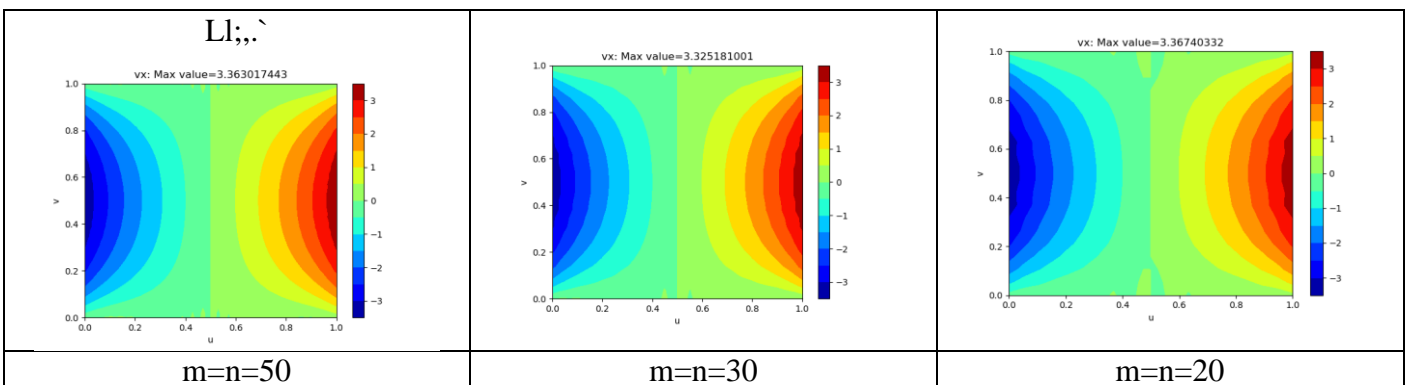


Figure 163: Test LM1 shear force v_x results ($m=n=20, 30, 50$)

Test LM2, three-point interpolation, flat square, vertical loading, rectangular matrix

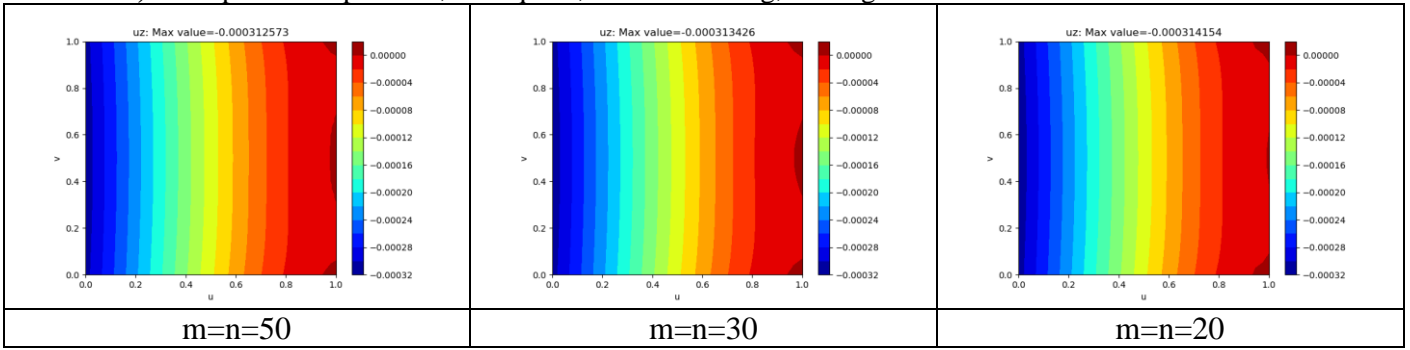


Figure 164: Test LM2 displacement uz results ($m=n=20, 30, 50$)

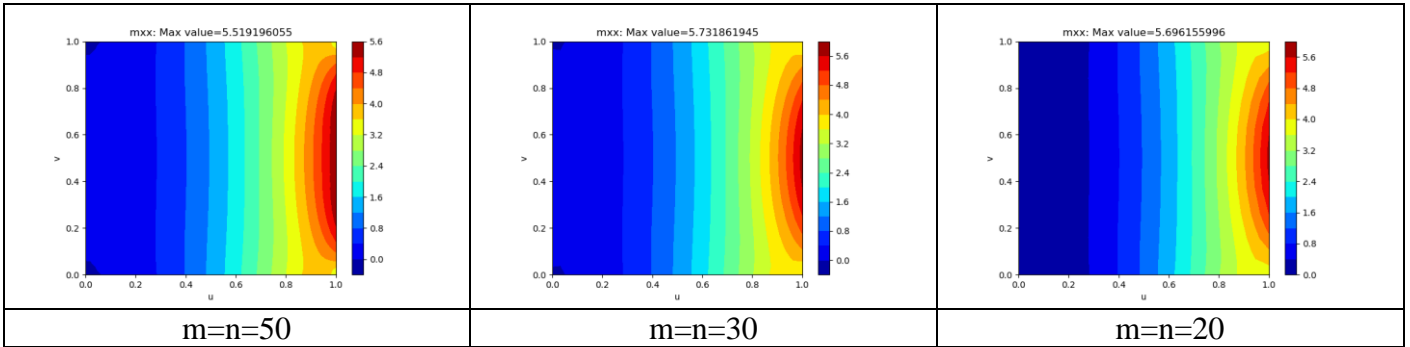


Figure 165: Test LM2 bending moment mxx results ($m=n=20, 30, 50$)

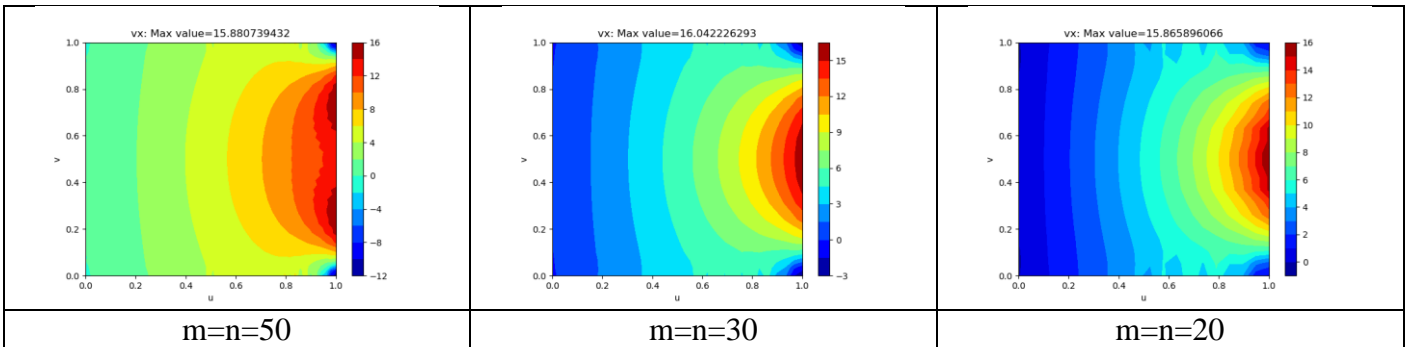


Figure 166: Test LM2 shear force vx results ($m=n=20, 30, 50$)

Test LM3, three-point interpolation, canopy, vertical loading, rectangular matrix

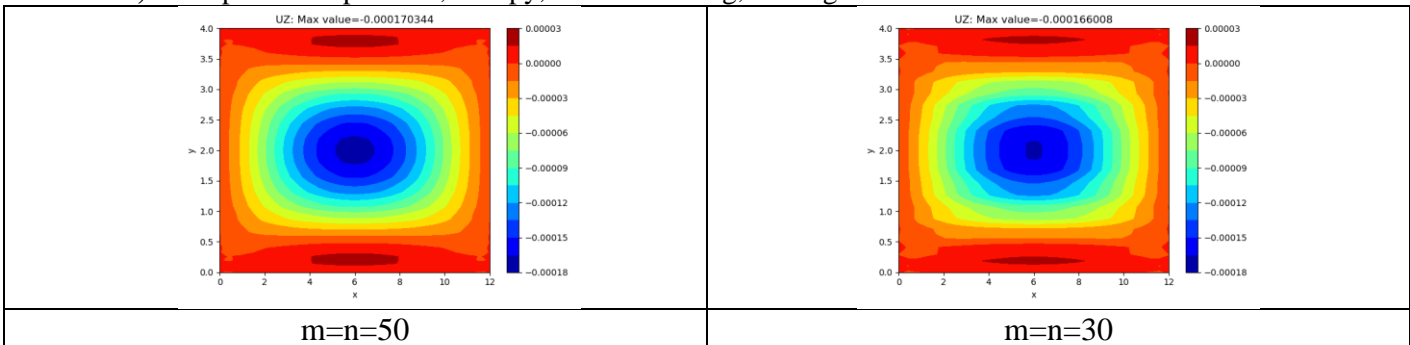


Figure 167: Test LM3 displacement uz results ($m=n=30, 50$)

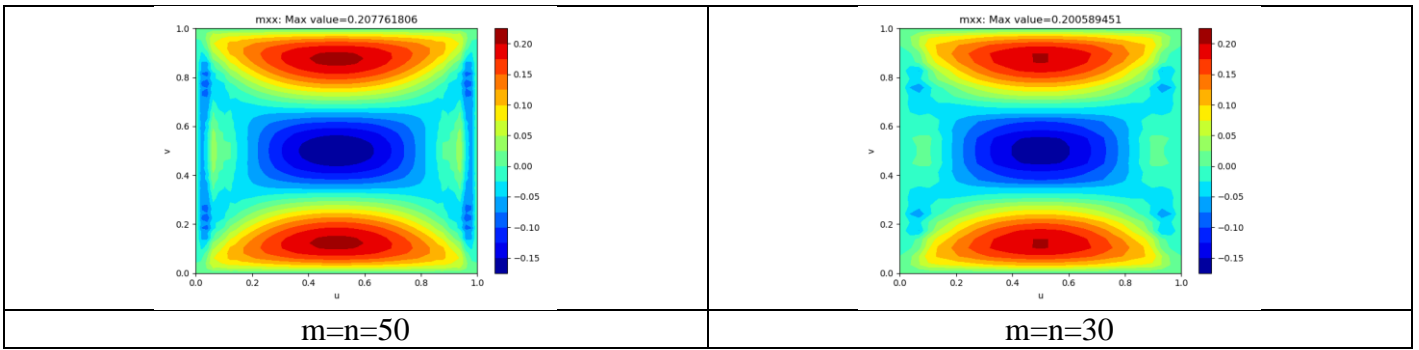


Figure 168: Test LM3 bending moment m_{xx} results ($m=n=30, 50$)

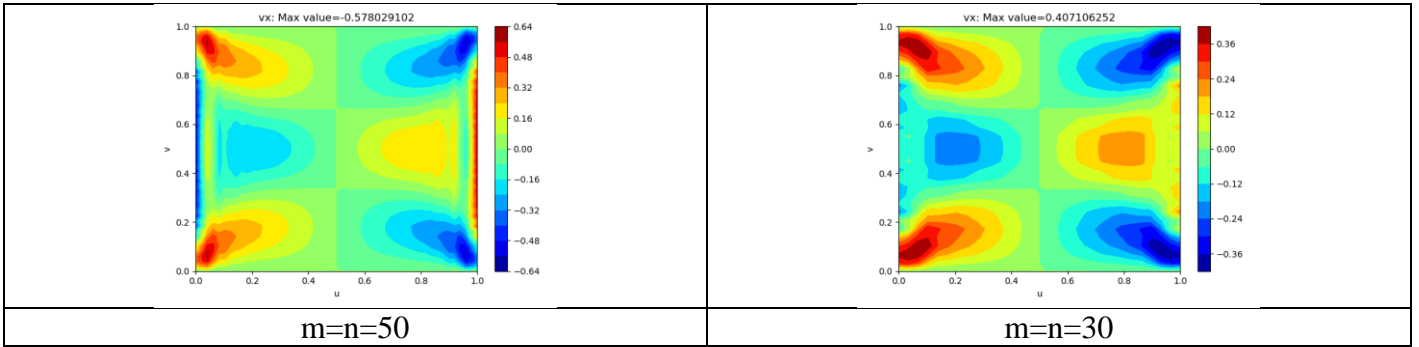


Figure 169: Test LM3 bending moment v_x results ($m=n=30, 50$)

Test LM4, three-point interpolation, canopy, vertical loading, rectangular matrix

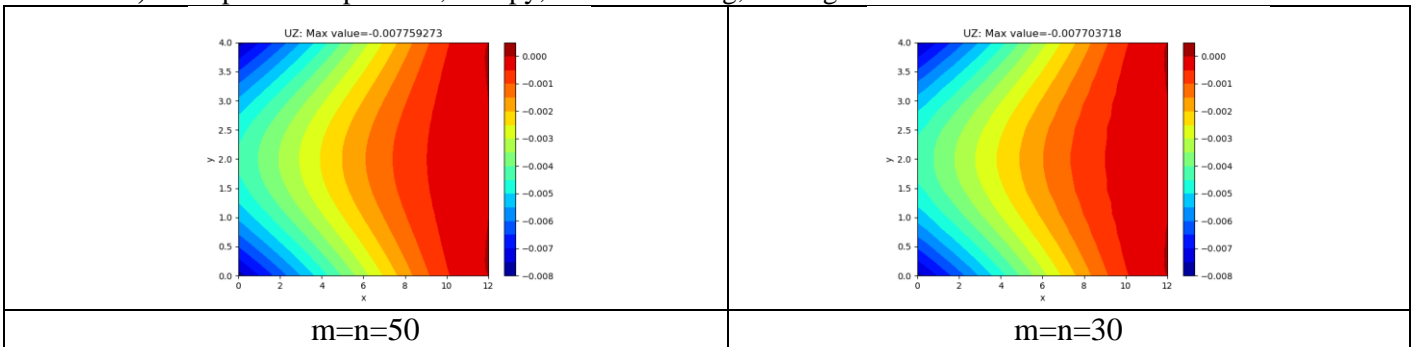


Figure 170: Test LM4 displacement u_z results ($m=n=30, 50$)

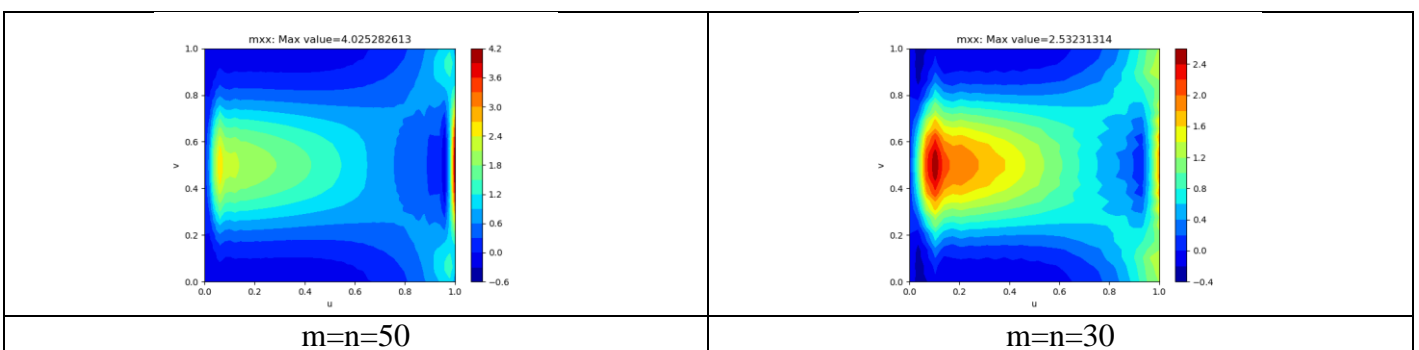


Figure 171: Test LM4 bending moment m_{xx} results ($m=n=30, 50$)

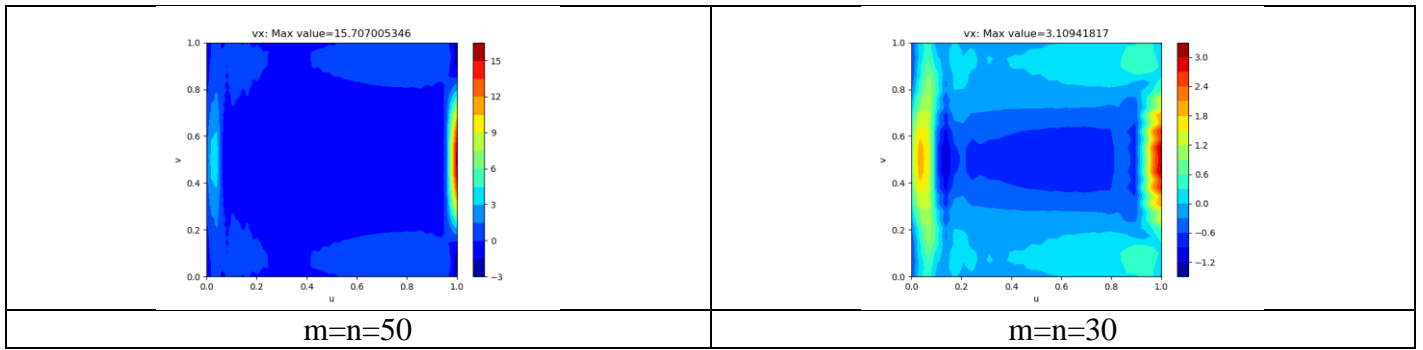


Figure 172: Test LM4 bending moment v_x results ($m=n=30, 50$)

Test LM5, three-point interpolation, canopy, normal loading, rectangular matrix

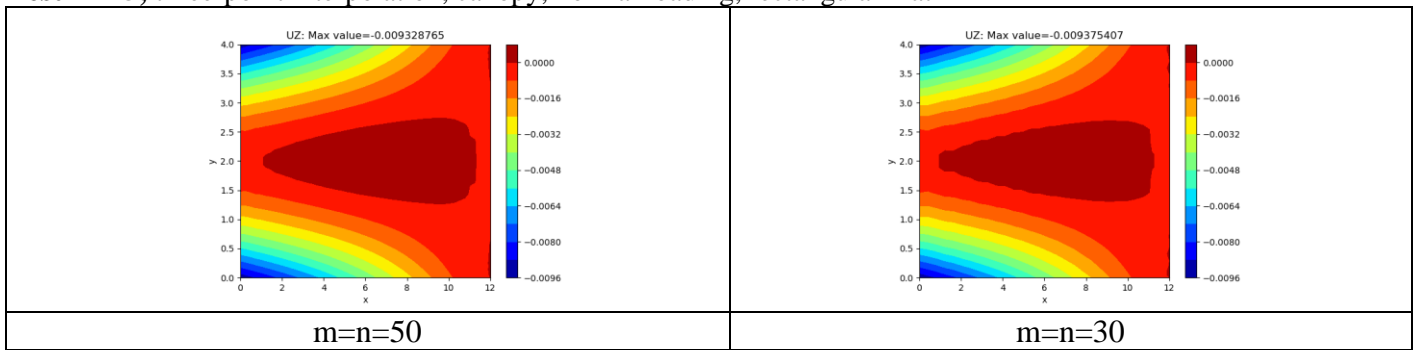


Figure 173: Test LM5 displacement u_z results ($m=n=30, 50$)

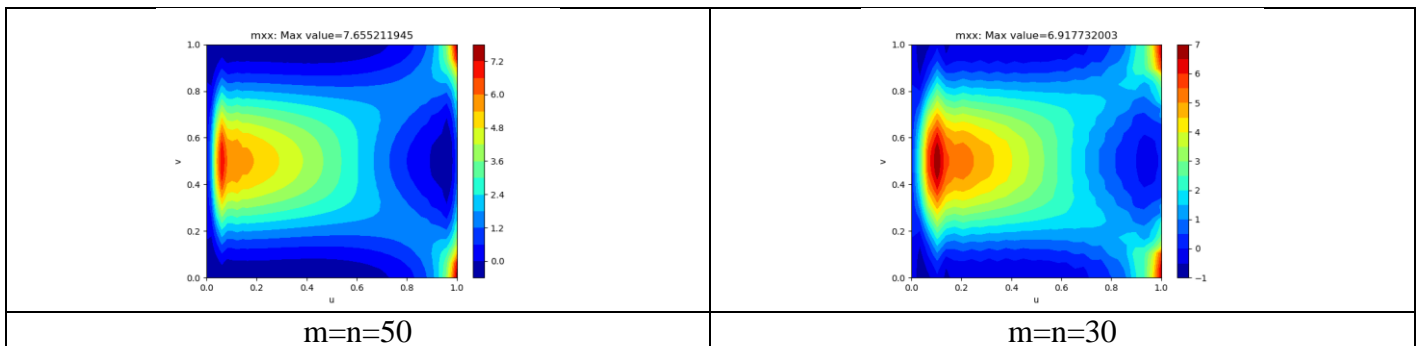


Figure 174: Test LM5 bending moment m_{xx} results ($m=n=30, 50$)

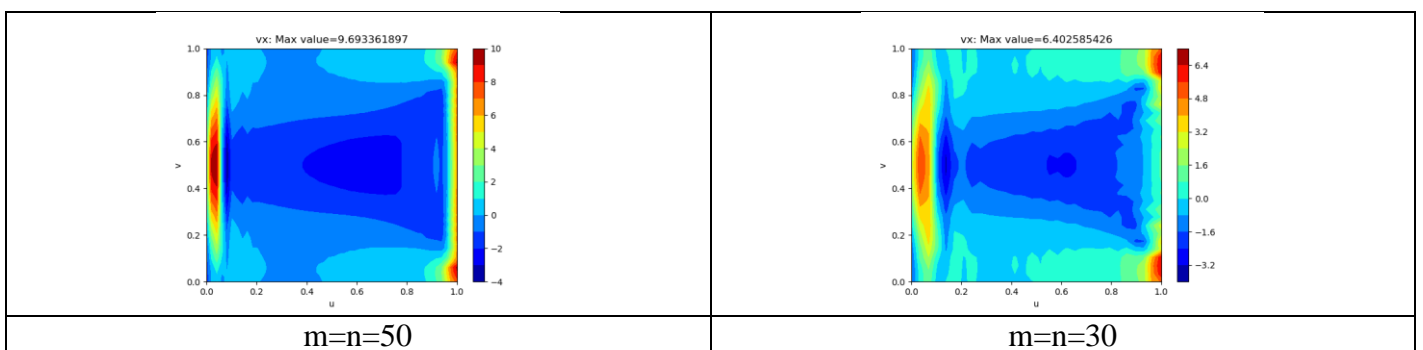


Figure 175: Test LM5 shear force v_x results ($m=n=30, 50$)

Test SLM1, three-point interpolation, flat square, vertical loading, square matrix

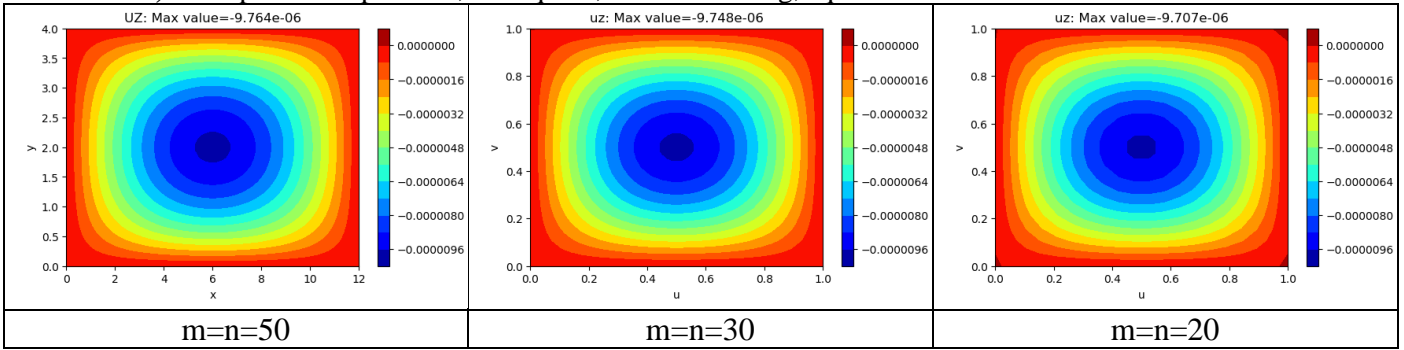


Figure 176: Test SLM1 displacement uz results (m=n=20, 30, 50)

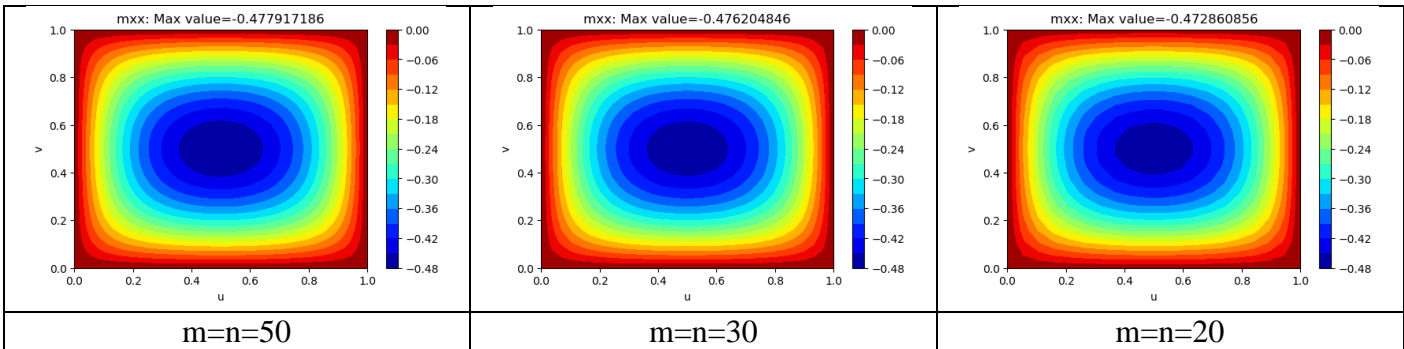


Figure 177: Test SLM1 bending moment mxx results (m=n=20, 30, 50)

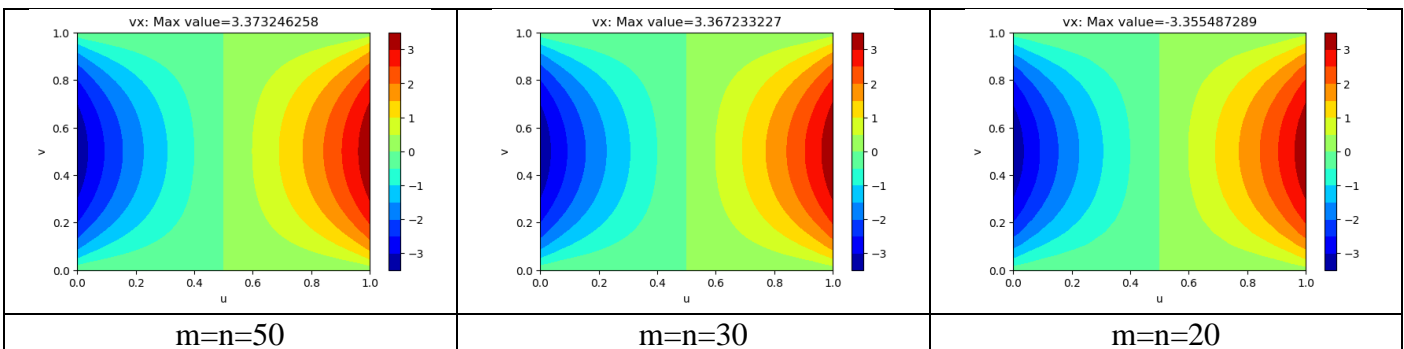


Figure 178: Test SLM1 shear force vx results (m=n=20, 30, 50)

Test SLM2, three-point interpolation, flat square, vertical loading, square matrix

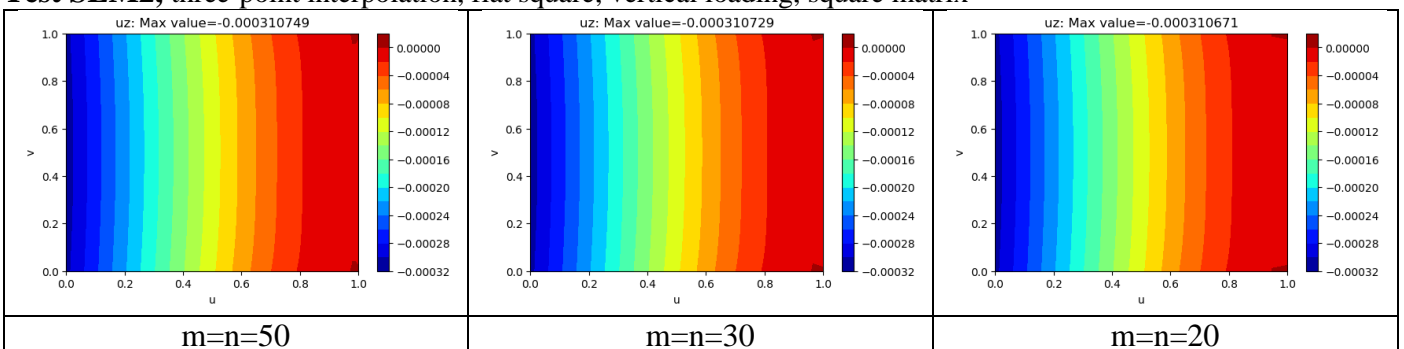


Figure 179: Test SLM2 displacement uz results (m=n=20, 30, 50)

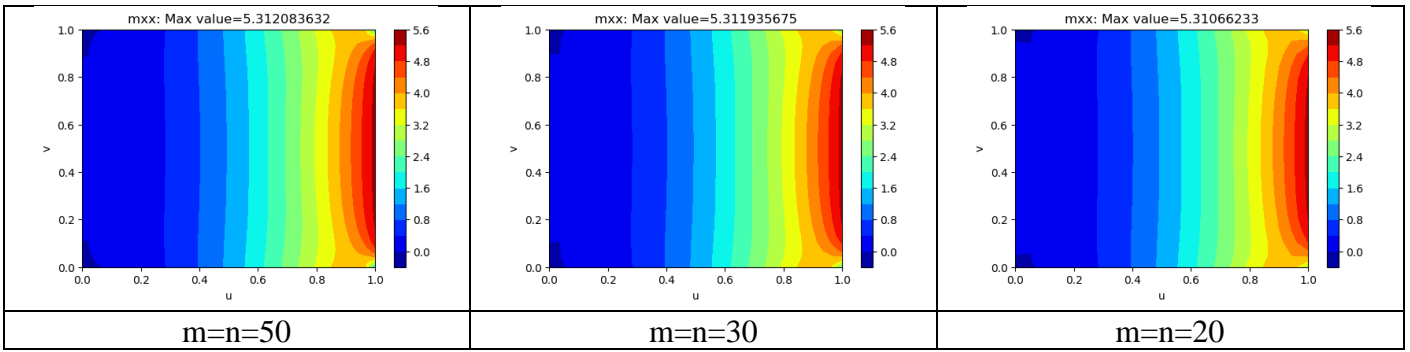


Figure 180: Test SLM2 bending moment m_{xx} results ($m=n=20, 30, 50$)

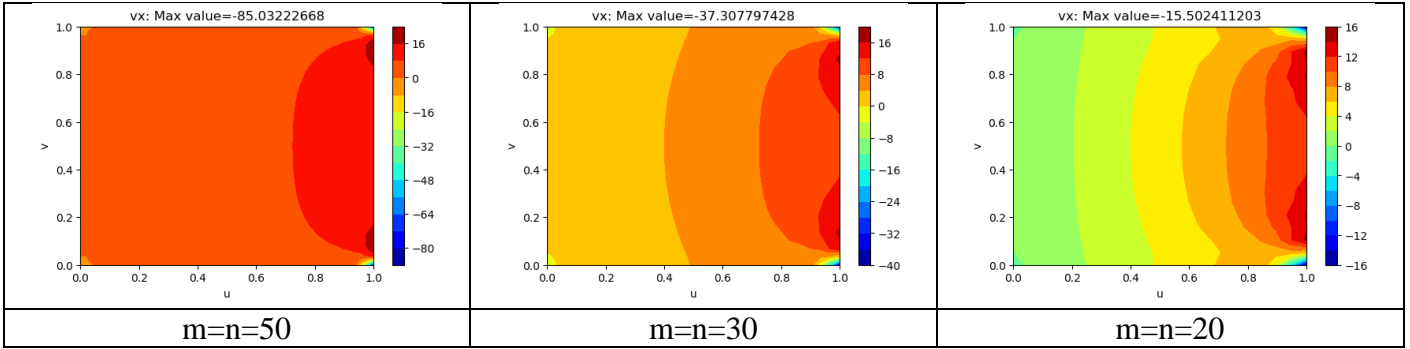


Figure 181: Test SLM2 shear force v_x results ($m=n=20, 30, 50$)

Test SLM3, three-point interpolation, canopy, vertical loading, square matrix

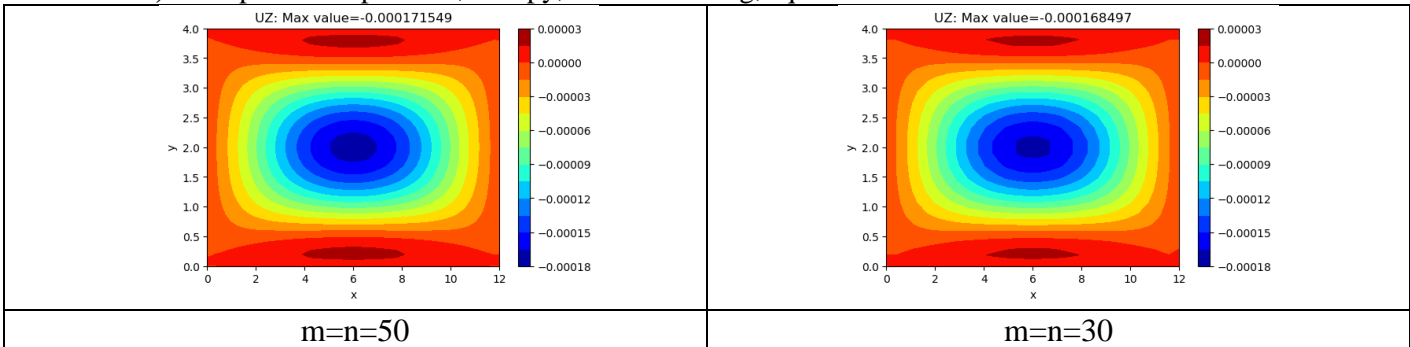


Figure 182: Test SLM3 displacement u_z results ($m=n=30, 50$)

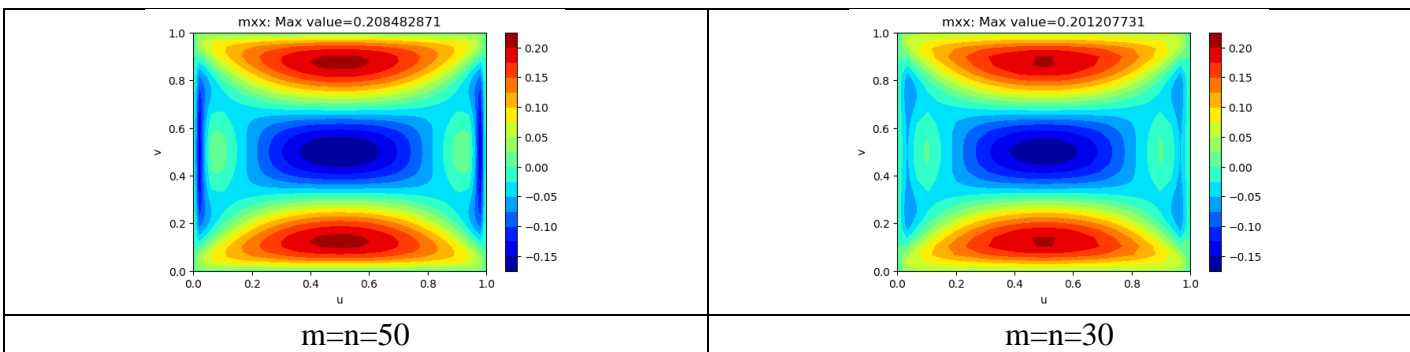


Figure 183: Test SLM3 bending moment m_{xx} results ($m=n=30, 50$)

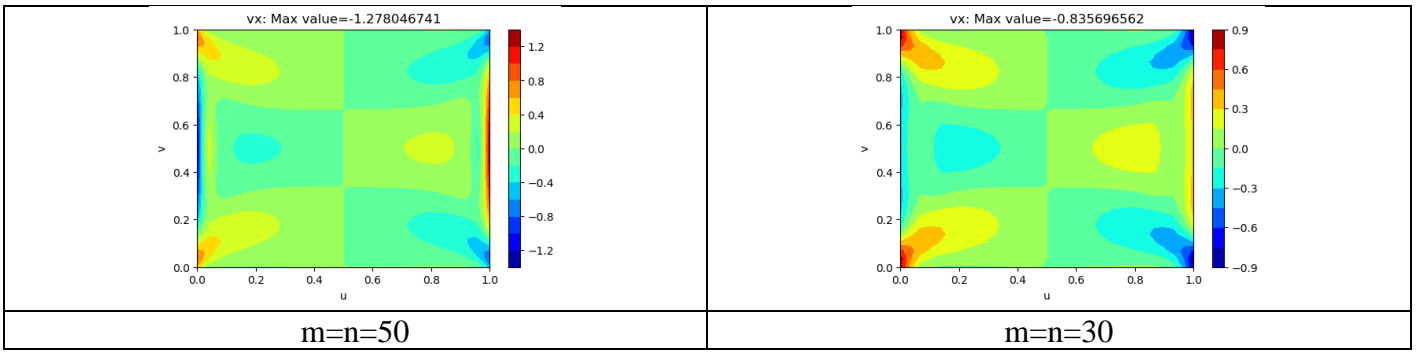


Figure 184: Test SLM3 bending moment vx results ($m=n=30, 50$)

Test SLM4, three-point interpolation, canopy, vertical loading, square matrix

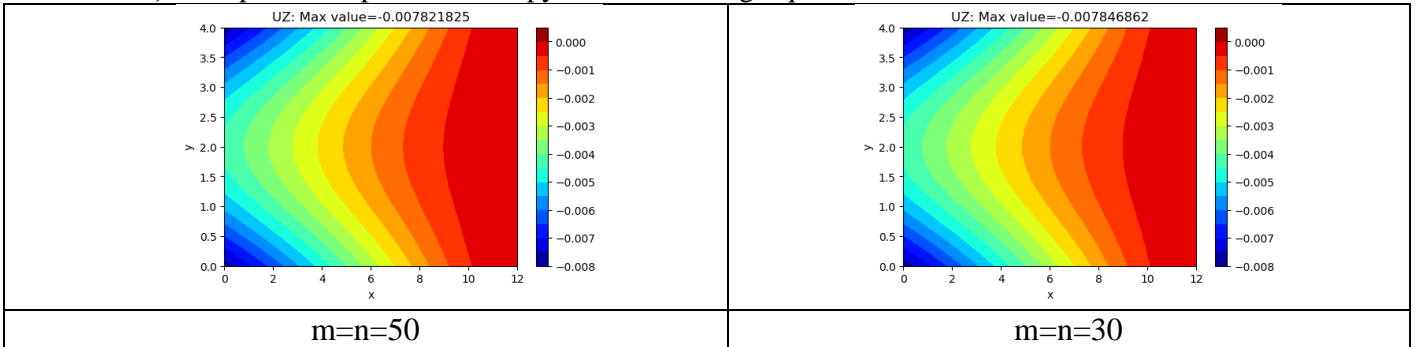


Figure 185: Test SLM4 displacement uz results ($m=n=30, 50$)

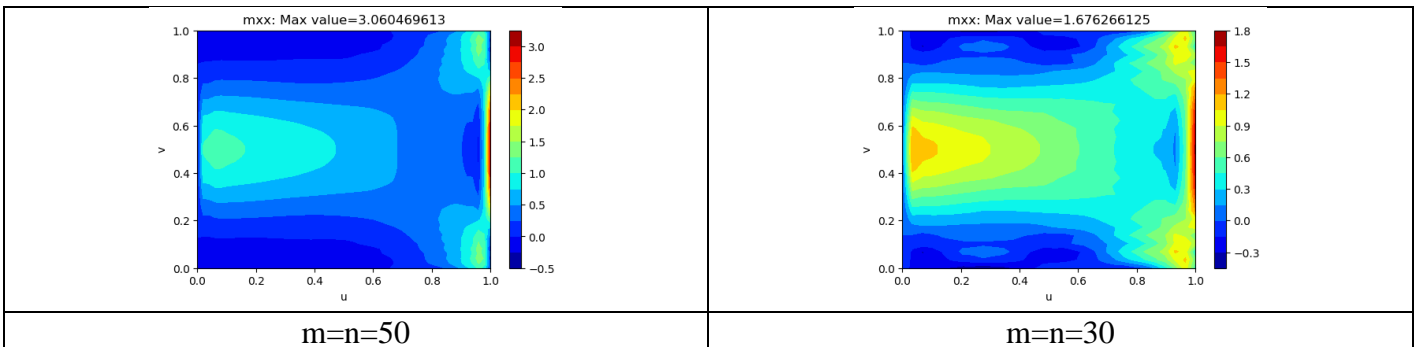


Figure 186: Test SLM4 bending moment mxx results ($m=n=30, 50$)

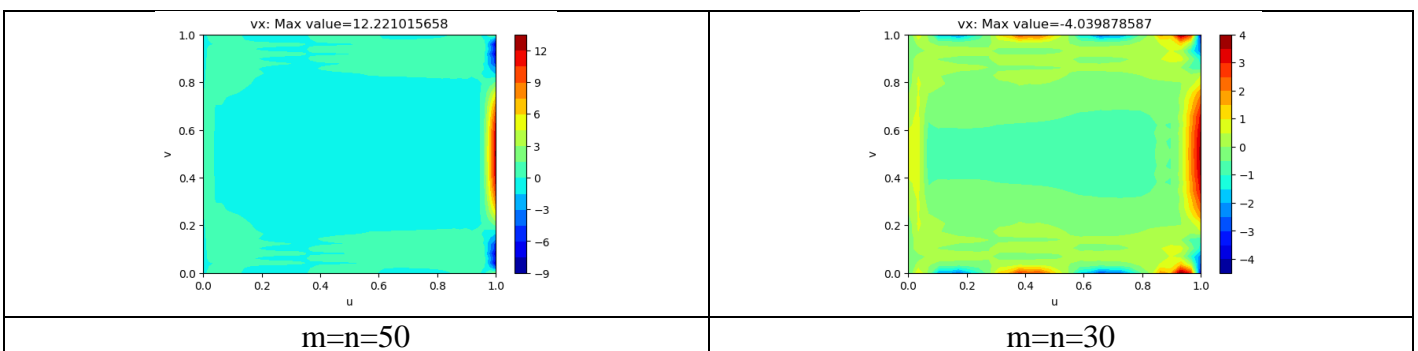


Figure 187: Test SLM 4 bending moment vx results ($m=n=30, 50$)

Test SLM5, three-point interpolation, canopy, normal loading, square matrix

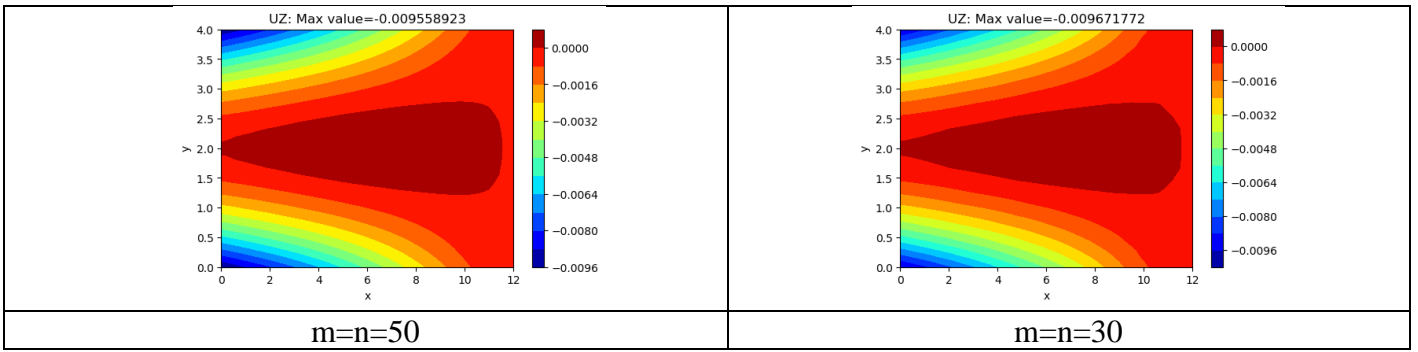


Figure 188: Test SLM5 displacement uz results ($m=n=30, 50$)

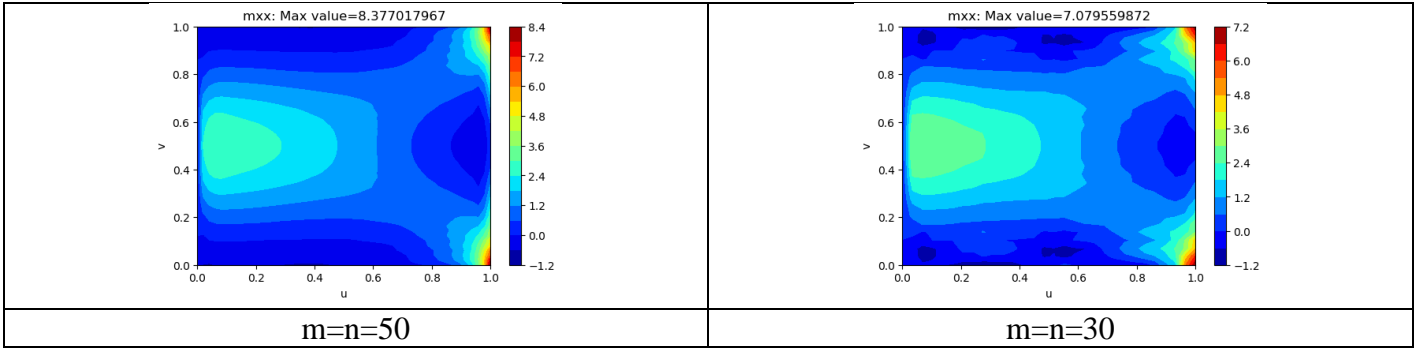


Figure 189: Test SLM5 bending moment mxx results ($m=n=30, 50$)

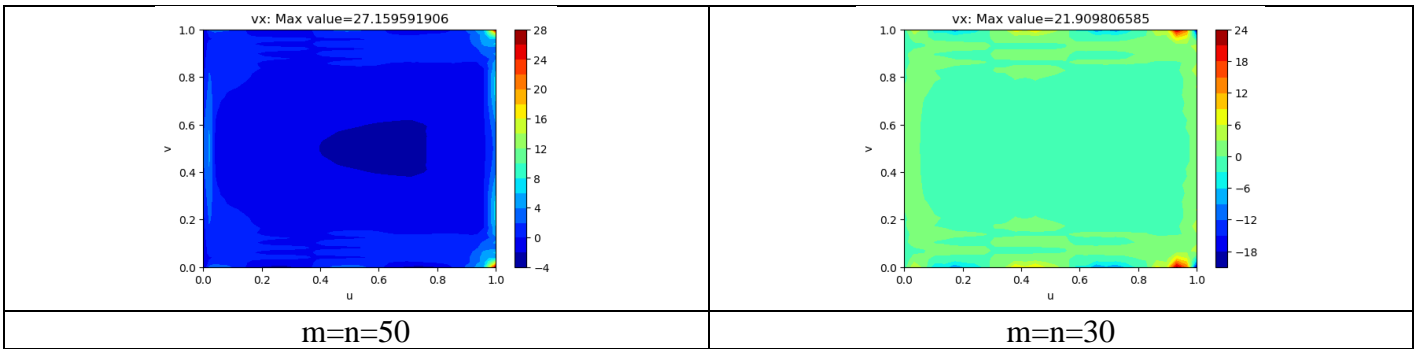


Figure 190: Test LM5 shear force vx results ($m=n=30, 50$)

Five-point difference approximation test plots

Test ALM1, five point interpolation, type A, flat square, vertical loading, rectangular matrix

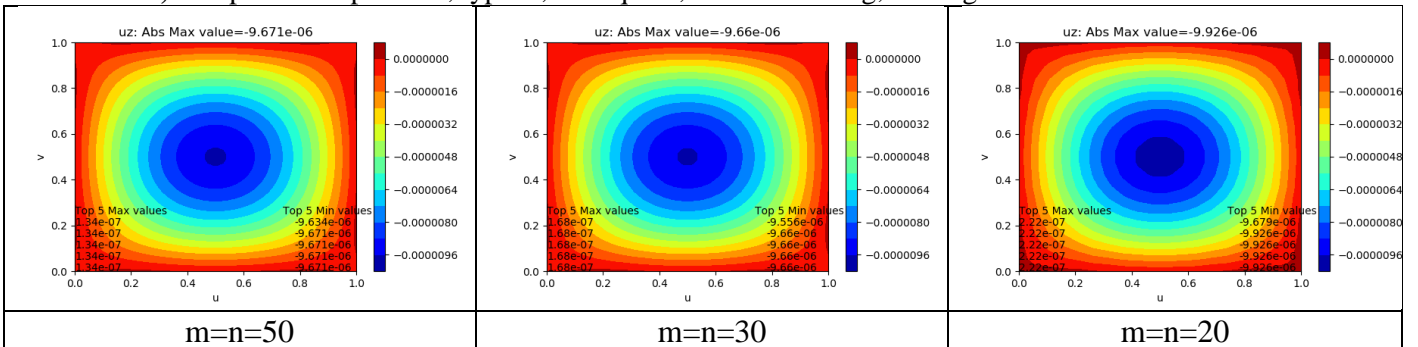


Figure 191: Test ALM1 displacement uz results ($m=n= 20, 30, 50$)

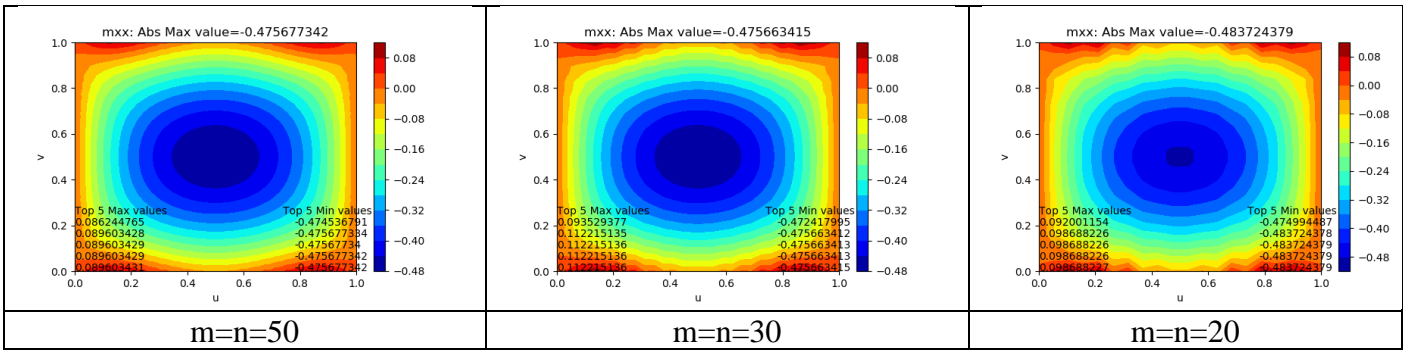


Figure 192: Test ALM1 bending moment m_{xx} results ($m=n=20, 30, 50$)

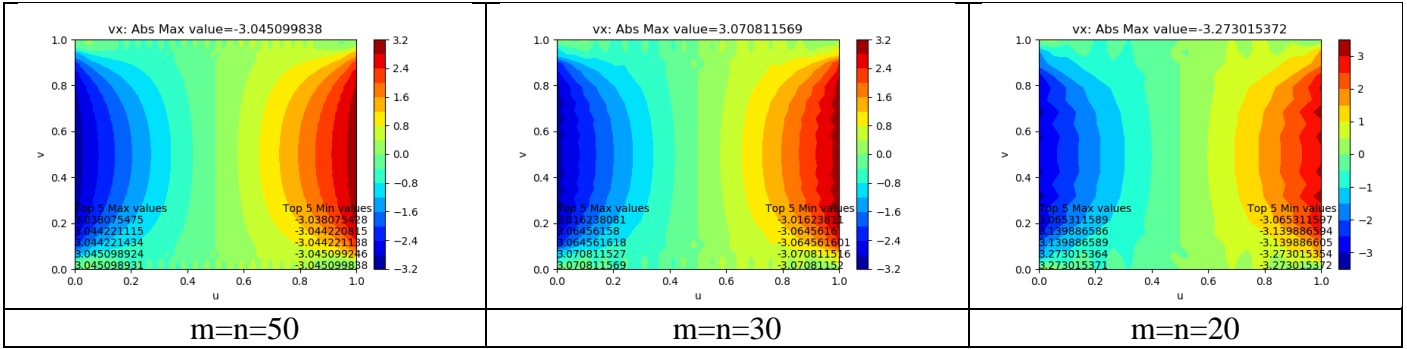


Figure 193: Test ALM1 shear force v_x results ($m=n=20, 30, 50$)

Test ALM2, five point interpolation, type A, flat square, vertical loading, rectangular matrix

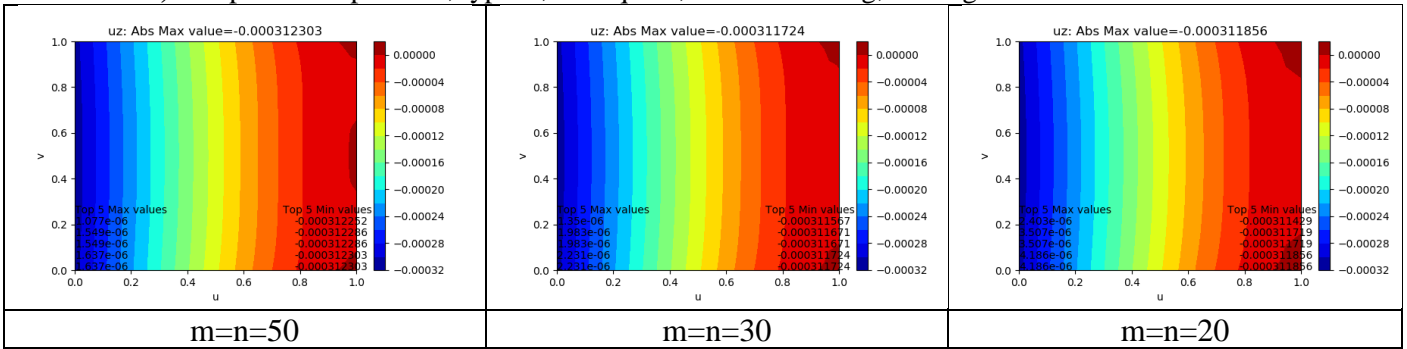


Figure 194: Test ALM2 displacement u_z results ($m=n=20, 30, 50$)

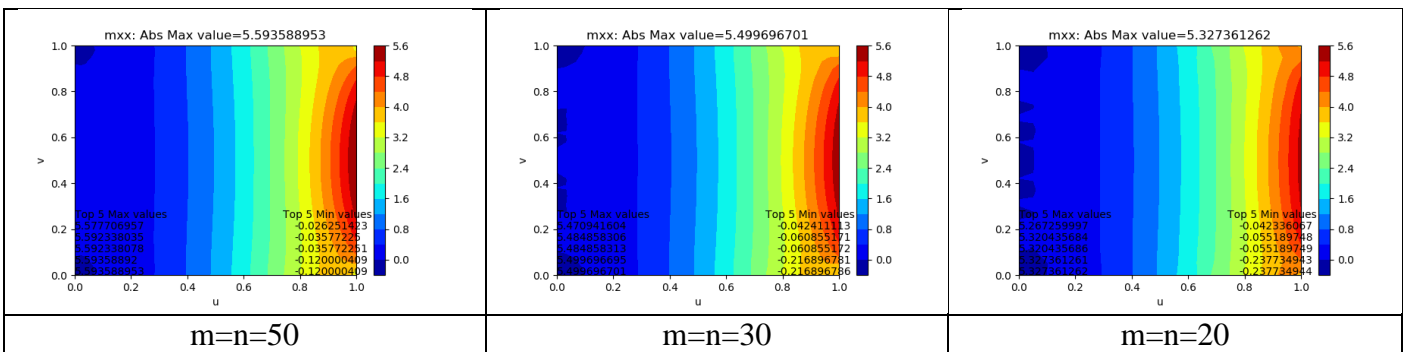


Figure 195: Test ALM2 bending moment m_{xx} results ($m=n=20, 30, 50$)

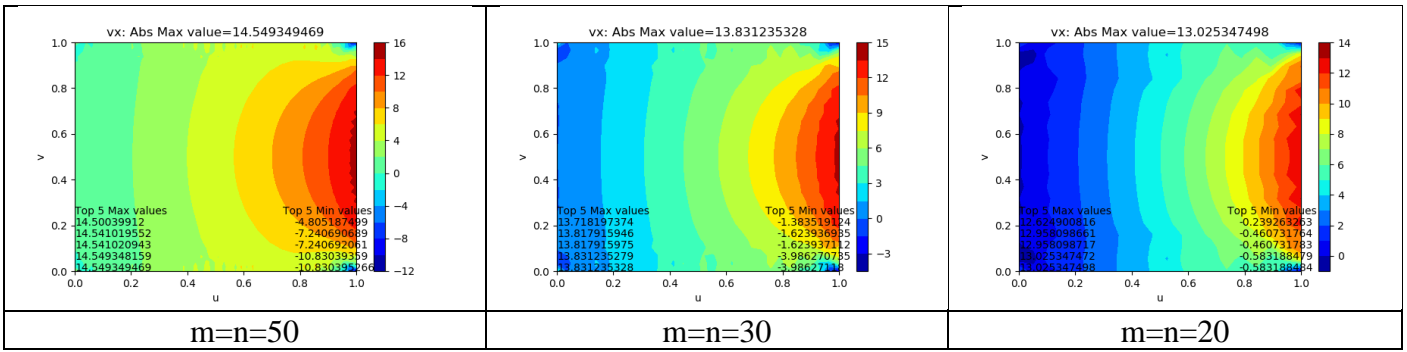


Figure 196: Test ALM2 shear force v_x results ($m=n=20, 30, 50$)

Test ALM3, five point interpolation, type A, canopy, vertical loading, rectangular matrix

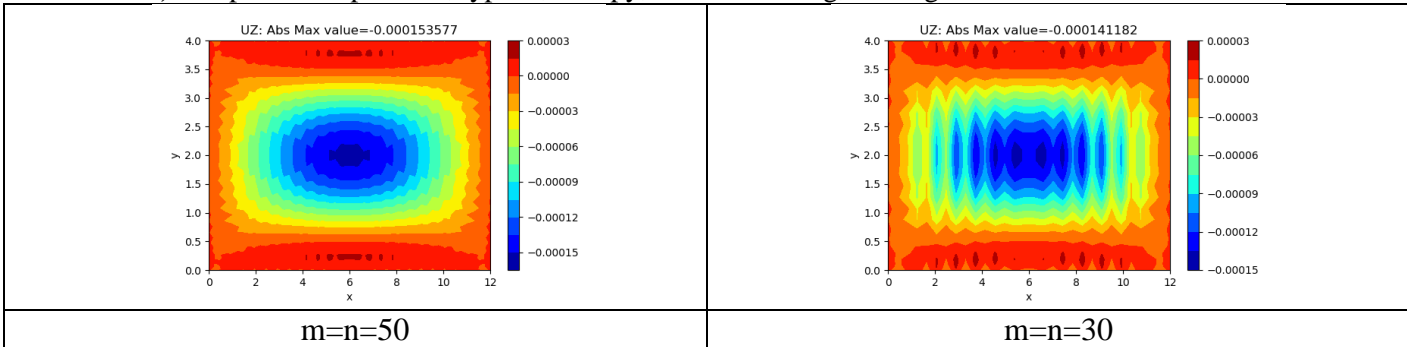


Figure 197: Test ALM3 displacement u_z results ($m=n=30, 50$)

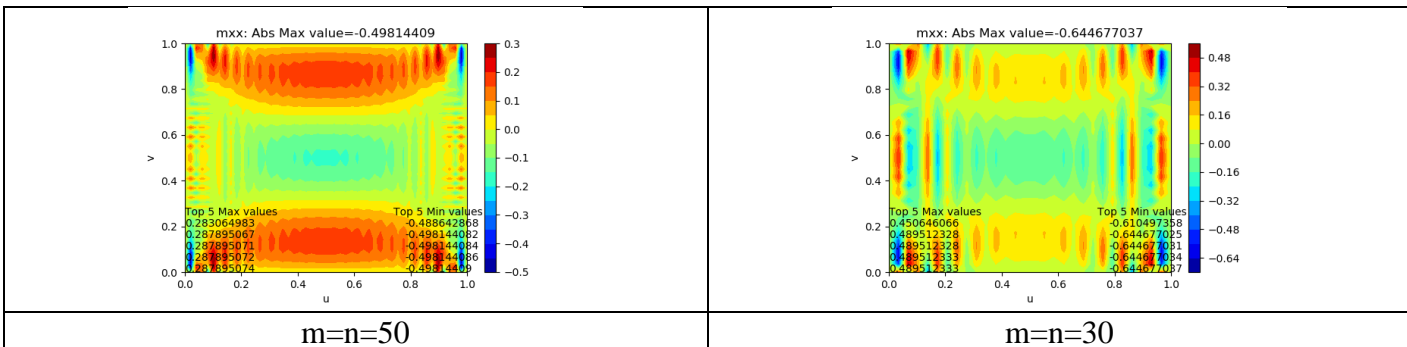


Figure 198: Test ALM3 bending moment m_{xx} results ($m=n=30, 50$)

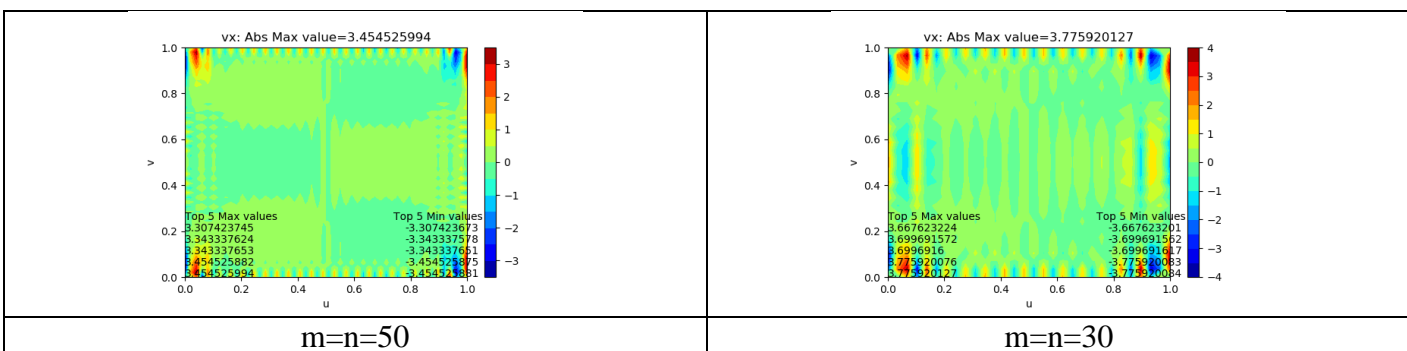


Figure 199: Test ALM3 bending moment v_x results ($m=n=30, 50$)

Test ALM4, five point interpolation, type A, canopy, vertical loading, rectangular matrix

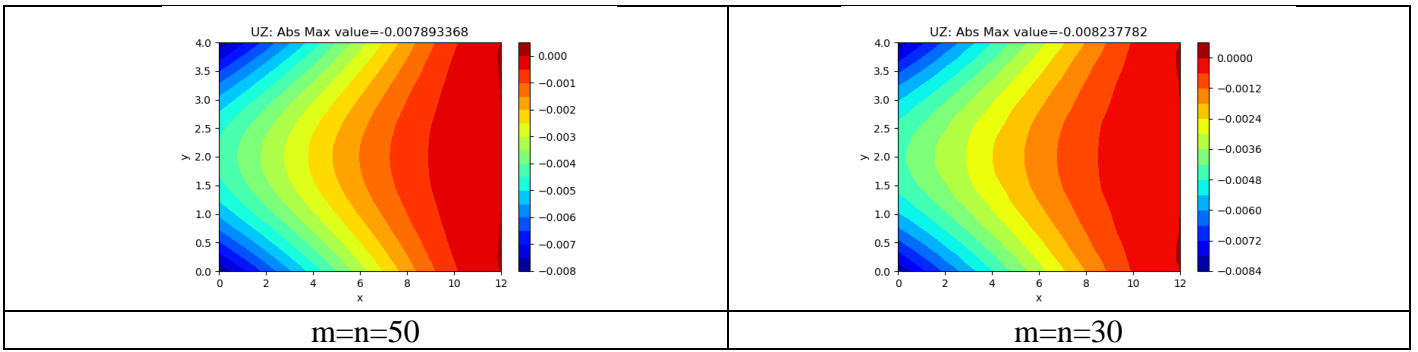


Figure 200: Test ALM4 displacement uz results ($m=n=30, 50$)

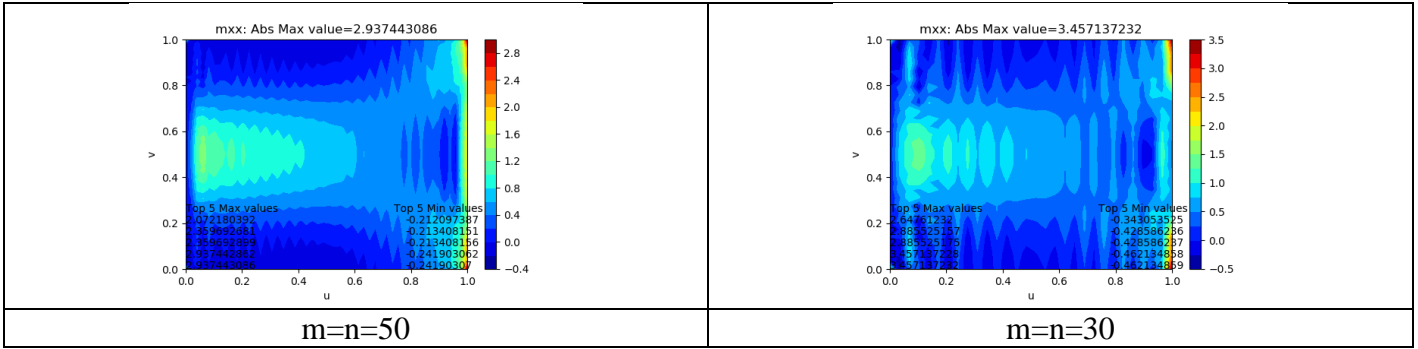


Figure 201: Test ALM4 bending moment mxx results ($m=n=30, 50$)

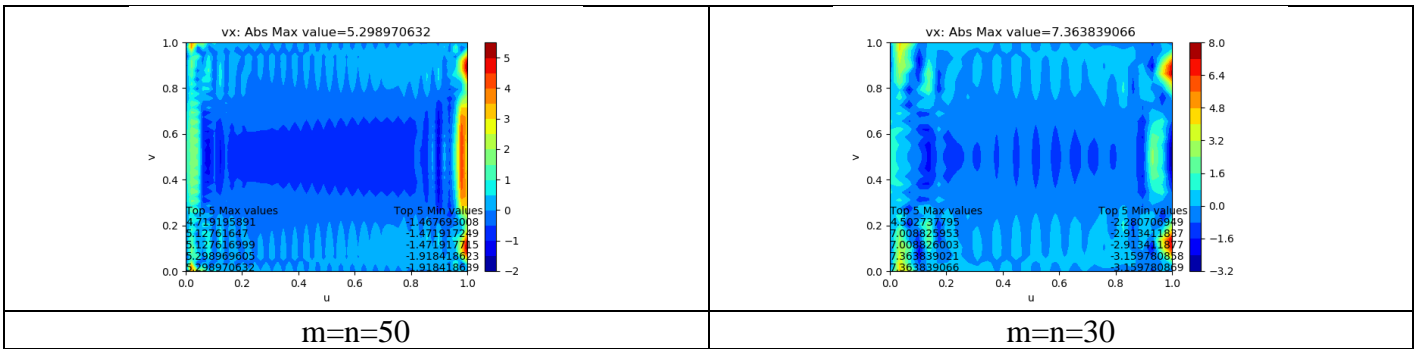


Figure 202: Test ALM4 bending moment vx results ($m=n=30, 50$)

Test ALM5, five point interpolation, type A, canopy, normal loading, rectangular matrix

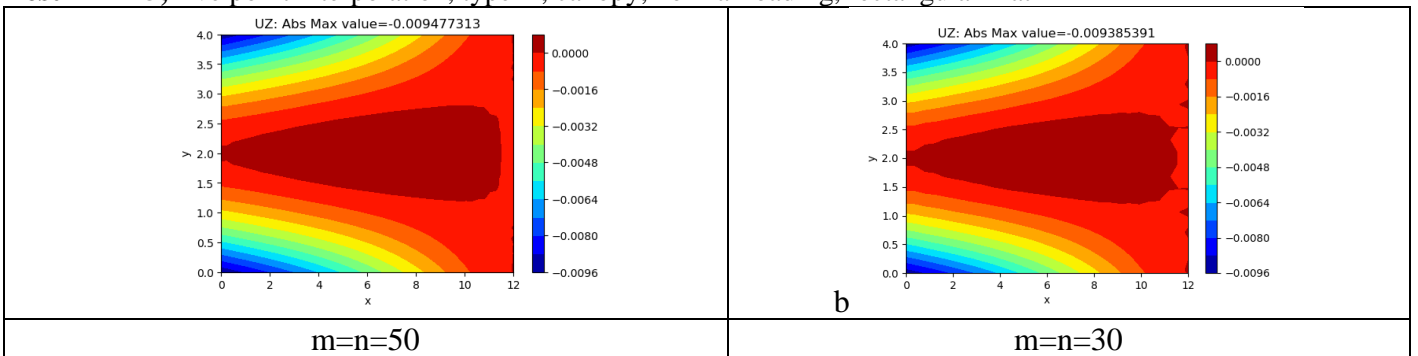


Figure 203: Test ALM5 displacement uz results ($m=n=30, 50$)

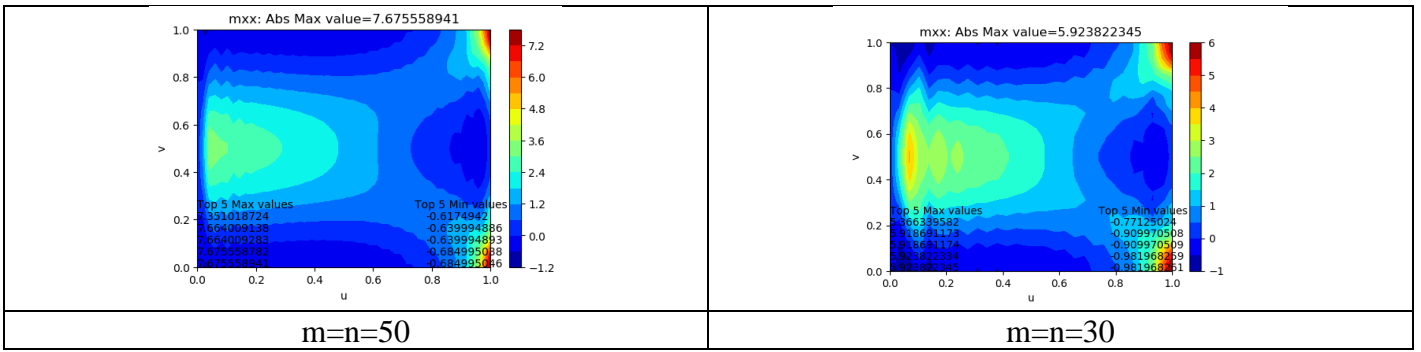


Figure 204: Test ALM5 bending moment m_{xx} results ($m=n=30, 50$)

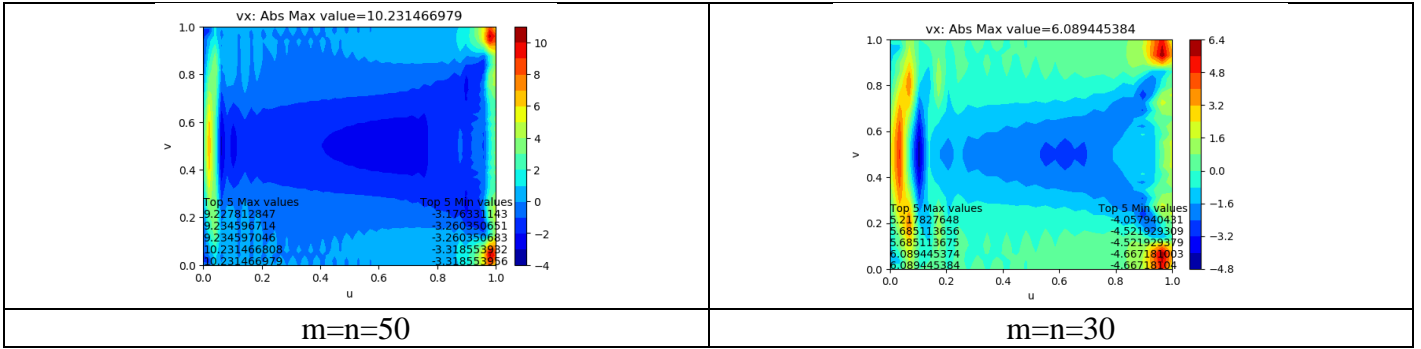


Figure 205: Test ALM5 shear force v_x results ($m=n=30, 50$)

Test BLM1, five point interpolation, type B, flat square, vertical loading, rectangular matrix

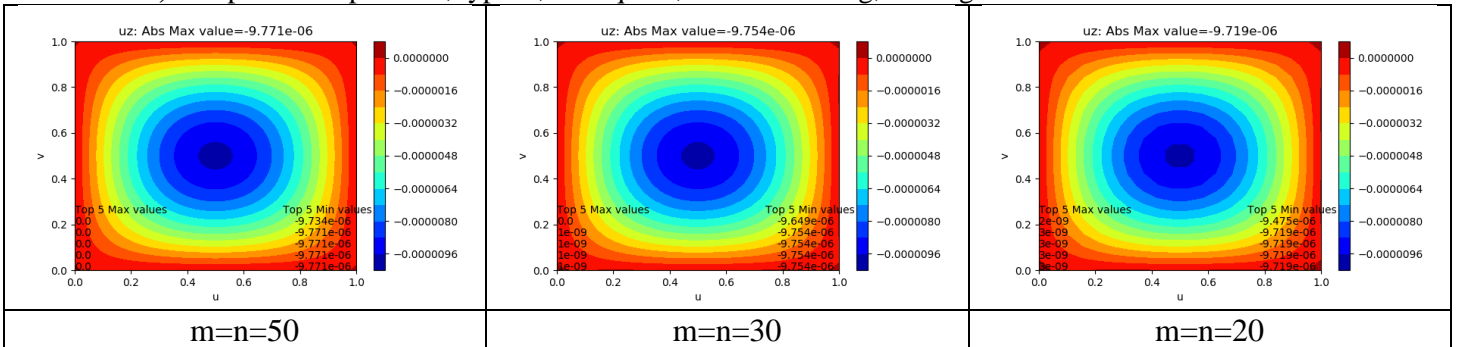


Figure 206: Test BLM1 displacement u_z results ($m=n= 20, 30, 50$)

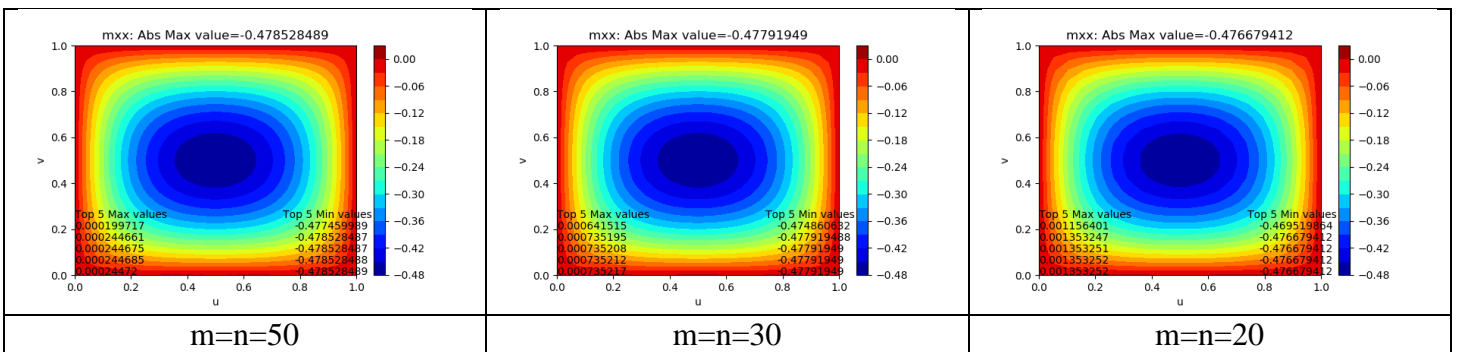


Figure 207: Test BLM1 bending moment m_{xx} results ($m=n= 20, 30, 50$)

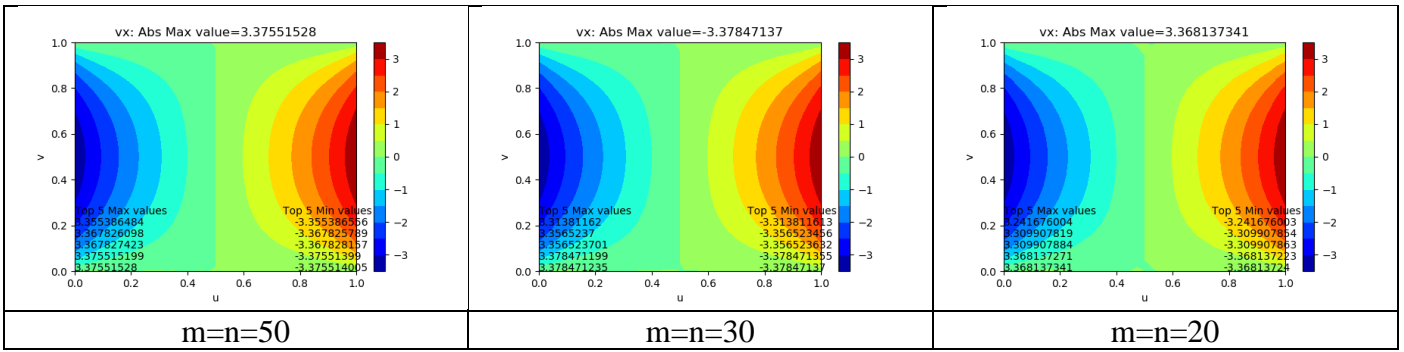


Figure 208: Test BLM1 shear force v_x results ($m=n=20, 30, 50$)

Test BLM2, five point interpolation, type B, flat square, vertical loading, rectangular matrix

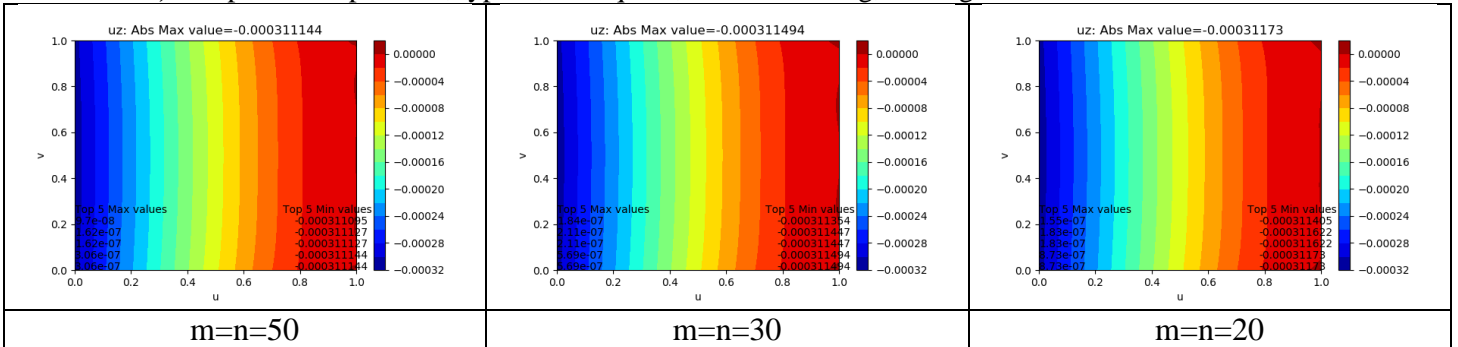


Figure 209: Test BLM2 displacement u_z results ($m=n=20, 30, 50$)

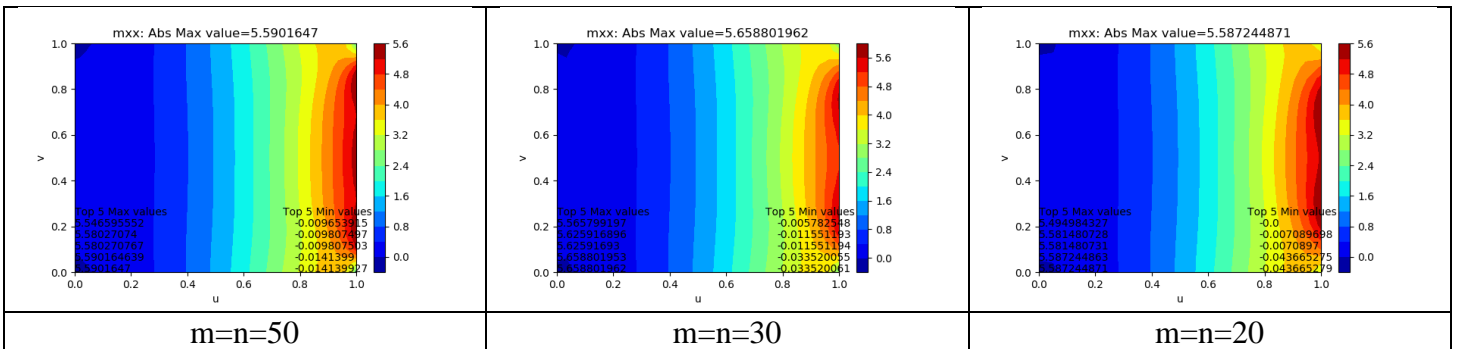


Figure 210: Test BLM2 bending moment m_{xx} results ($m=n=20, 30, 50$)

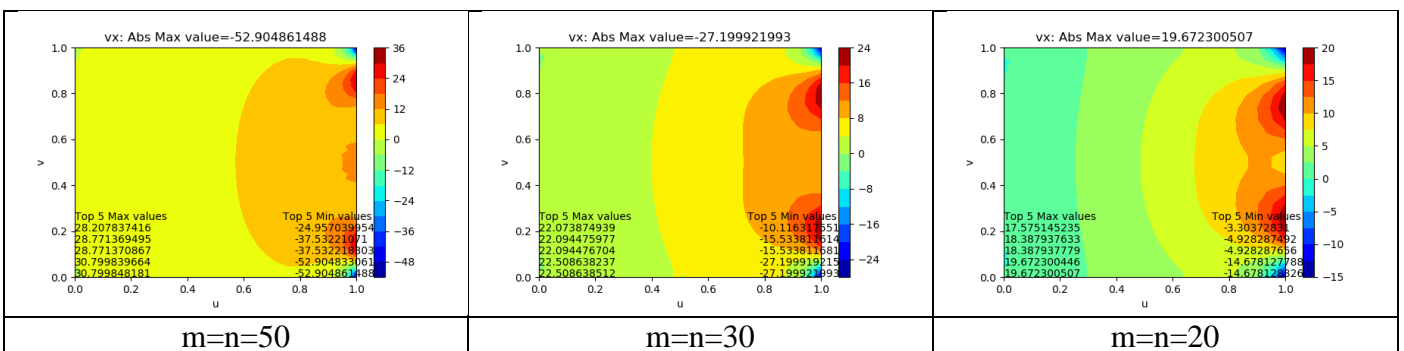


Figure 211: Test BLM2 shear force v_x results ($m=n=20, 30, 50$)

Test BLM3, five point interpolation, type B, canopy, vertical loading, rectangular matrix

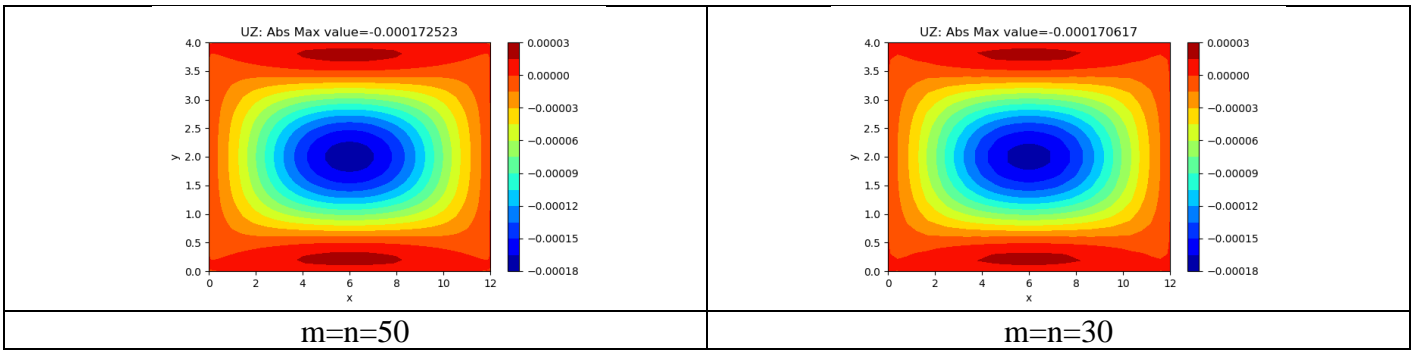


Figure 212: Test BLM3 displacement uz results ($m=n=30, 50$)

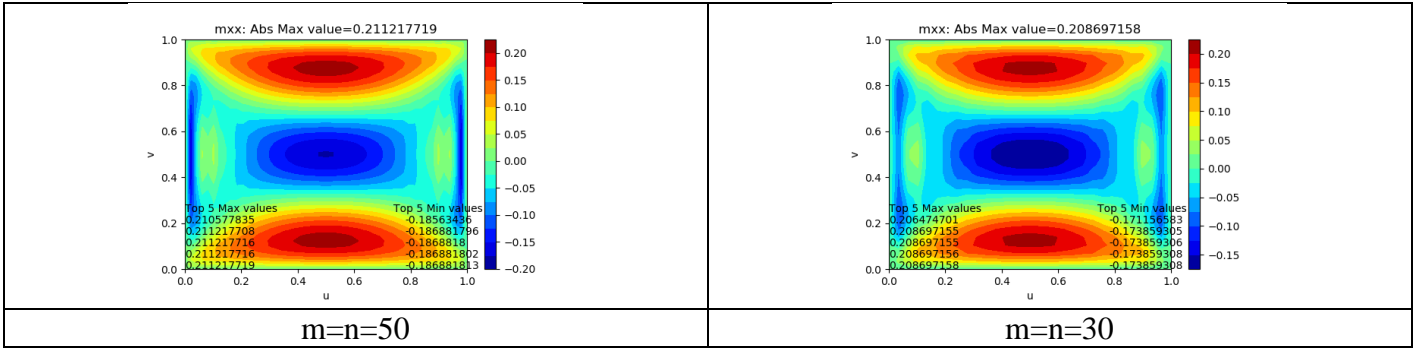


Figure 213: Test BLM3 bending moment mxx results ($m=n=30, 50$)

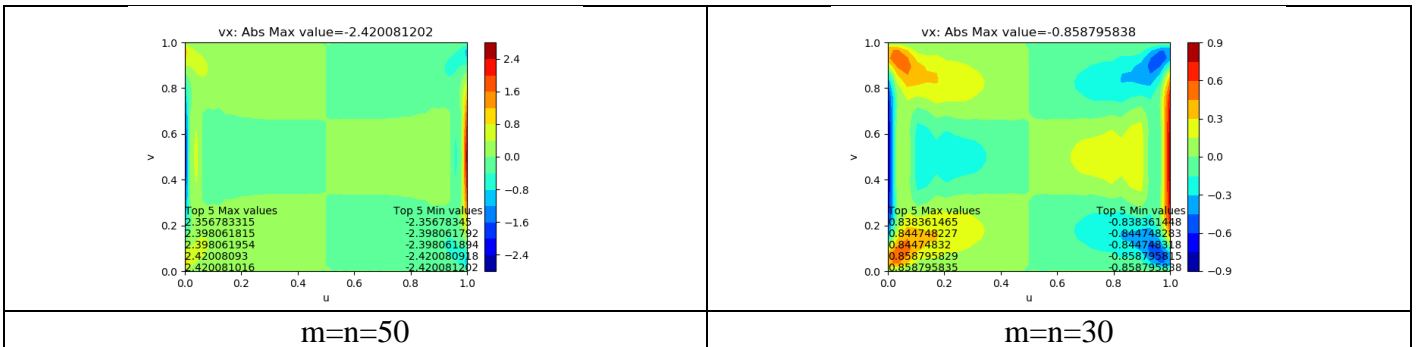


Figure 214: Test BLM3 bending moment vx results ($m=n=30, 50$)

Test BLM4, five point interpolation, type B, canopy, vertical loading, rectangular matrix

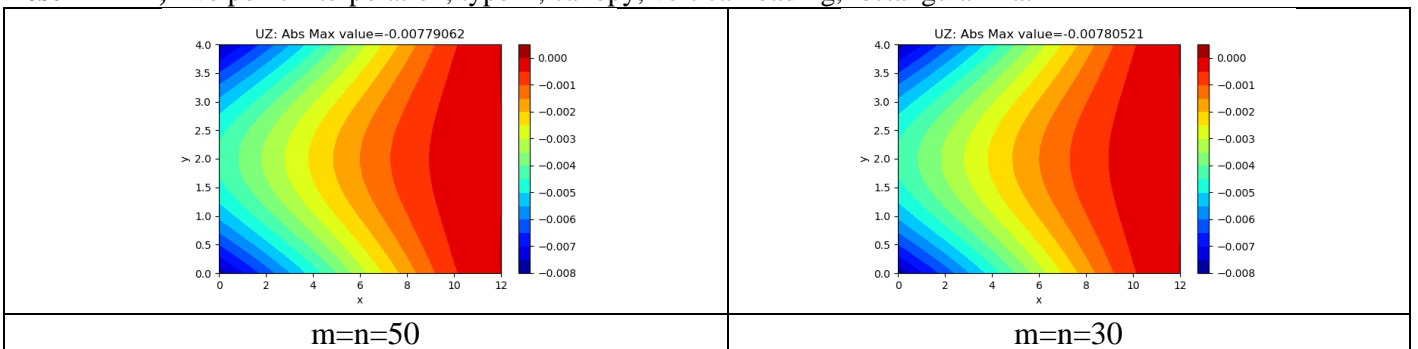


Figure 215: Test BLM4 displacement uz results ($m=n=30, 50$)

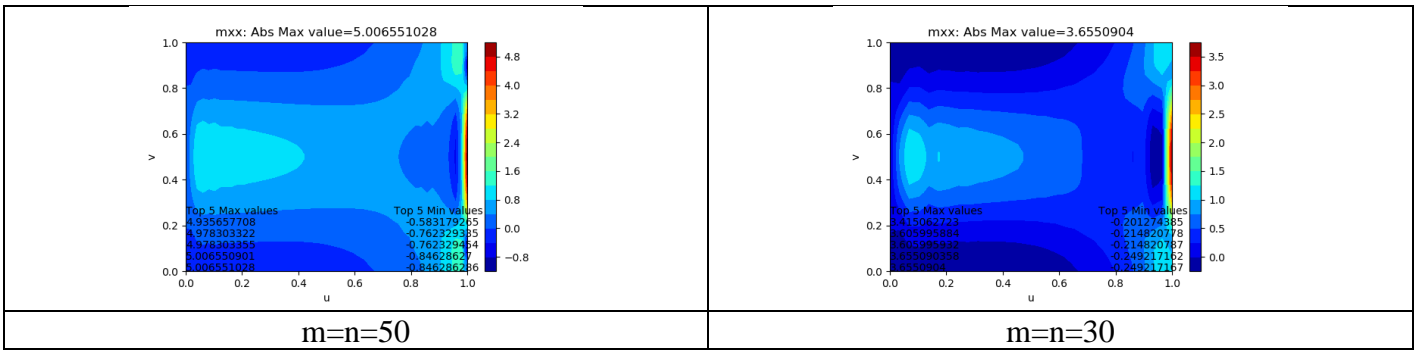


Figure 216: Test BLM4 bending moment m_{xx} results ($m=n=30, 50$)

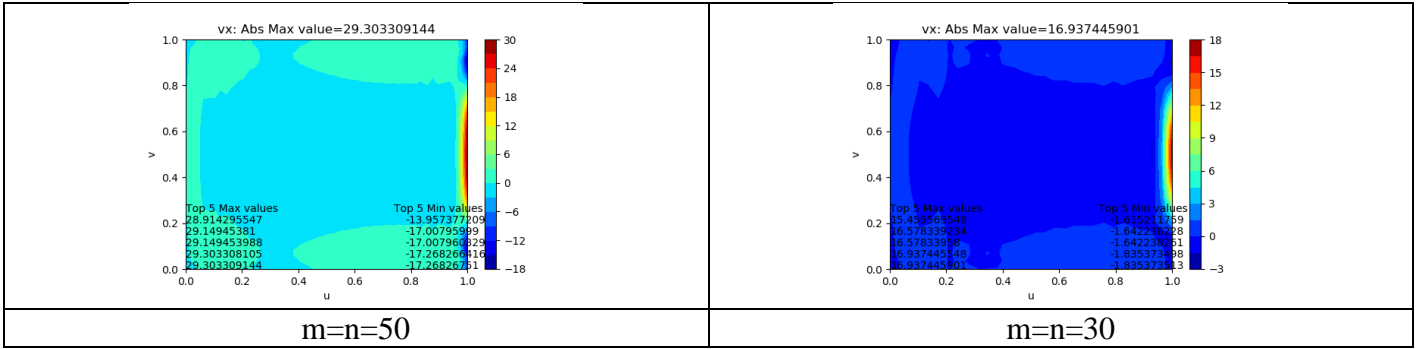


Figure 217: Test BLM4 bending moment v_x results ($m=n=30, 50$)

Test BLM5, five point interpolation, type B, canopy, normal loading, rectangular matrix

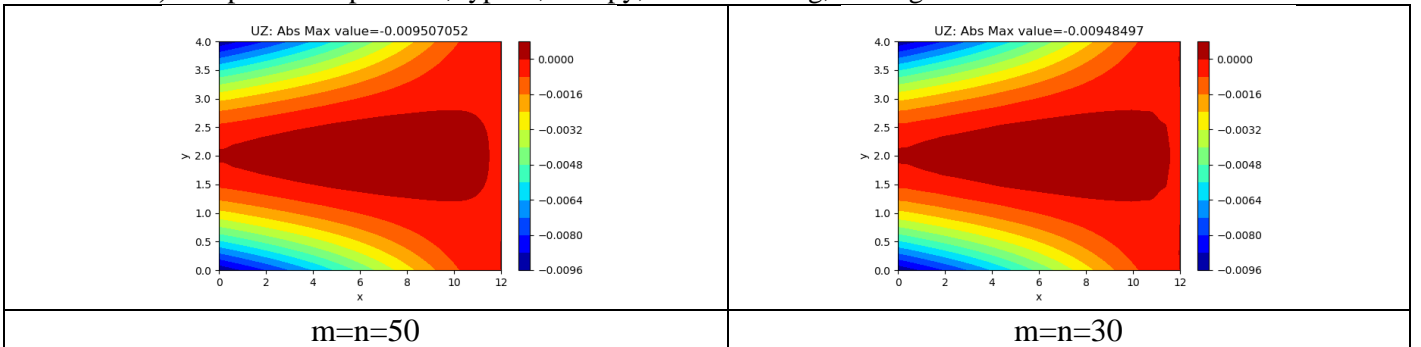


Figure 218: Test BLM5 displacement u_z results ($m=n=30, 50$)

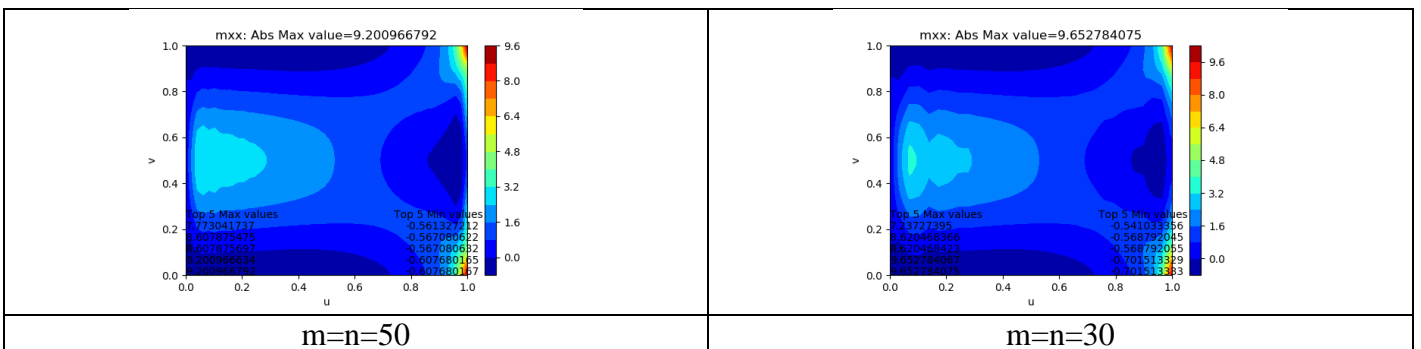


Figure 219: Test BLM5 bending moment m_{xx} results ($m=n=30, 50$)

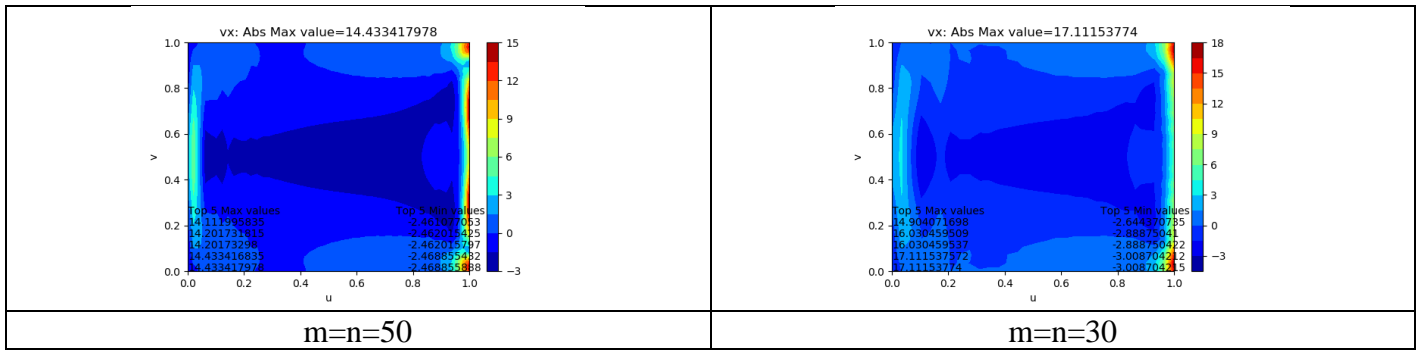


Figure 220: Test BLM5 shear force vx results (m=n=30, 50)

Test AR1, five point interpolation, type A, flat square, vertical loading, rectangular matrix

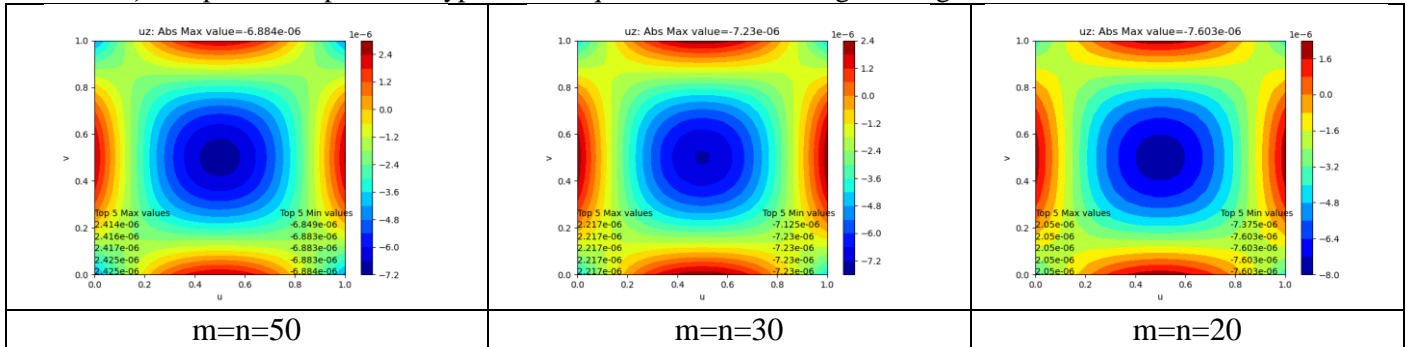


Figure 221: Test AR1 displacement uz results (m=n= 20, 30, 50)

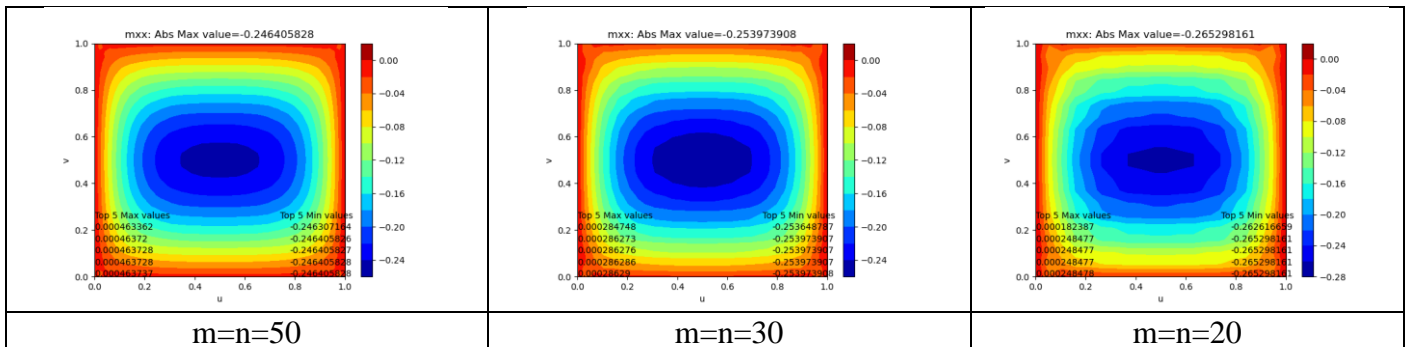


Figure 222: Test AR1 bending moment mxx results (m=n= 20, 30, 50)

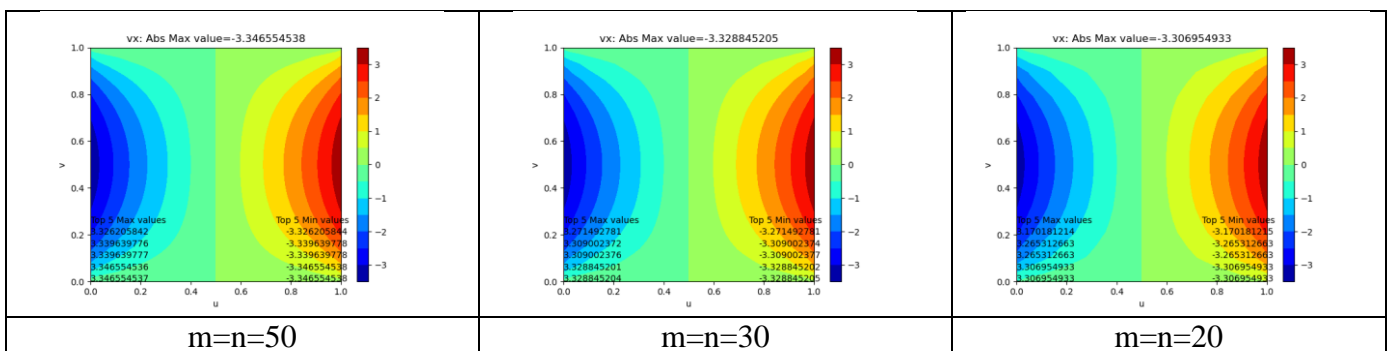


Figure 223: Test AR1 shear force vx results (m=n=20, 30, 50)

Test AR2, five point interpolation, type A, flat square, vertical loading, rectangular matrix

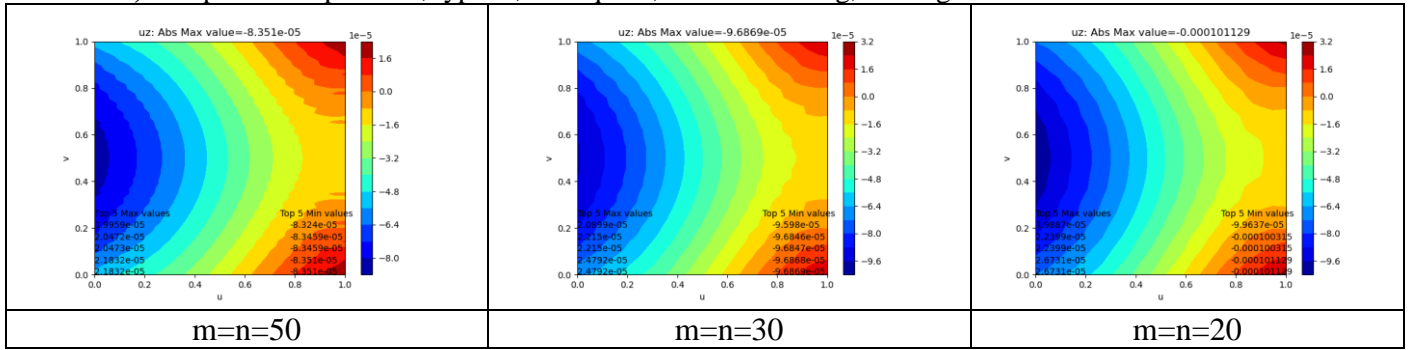


Figure 224: Test AR2 displacement uz results ($m=n=20, 30, 50$)

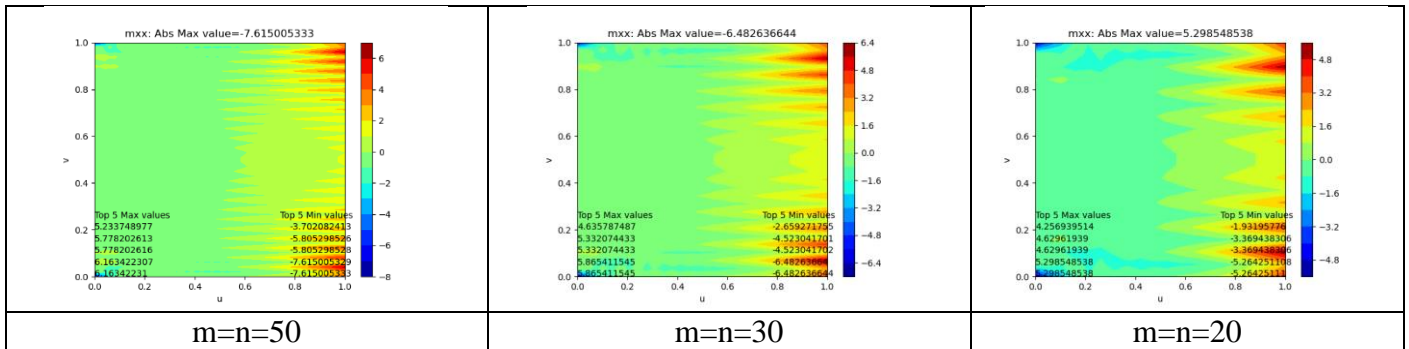


Figure 225: Test AR2 bending moment mxx results ($m=n=20, 30, 50$)

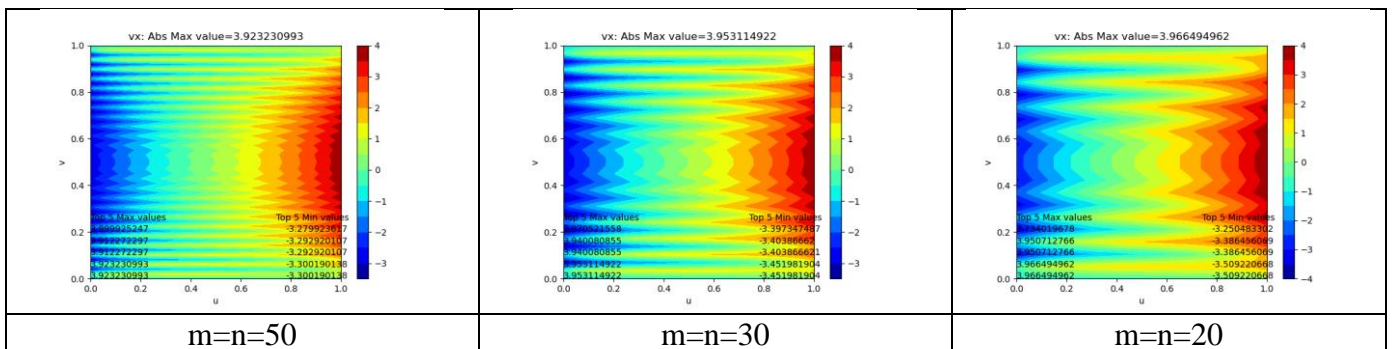


Figure 226: Test AR2 shear force vx results ($m=n=20, 30, 50$)

Test AR3, five point interpolation, type A, canopy, vertical loading, rectangular matrix

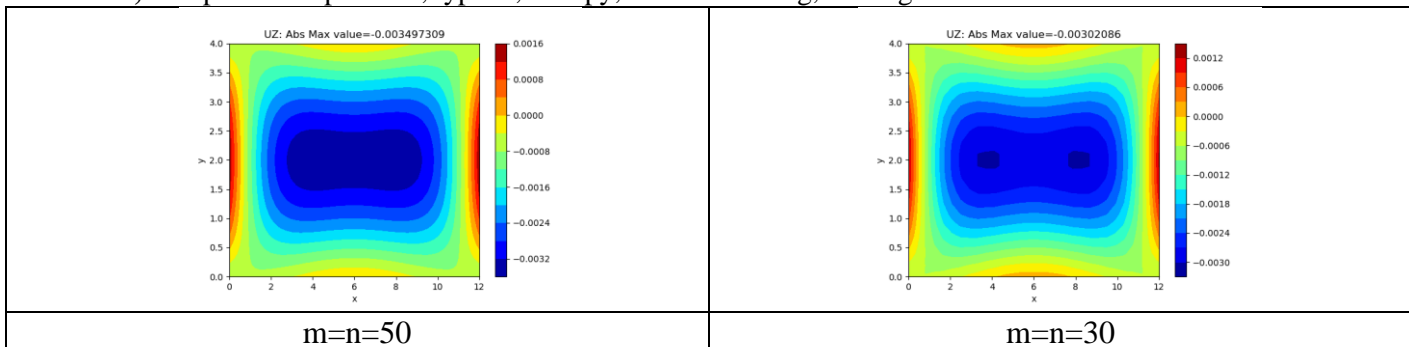


Figure 227: Test AR3 displacement uz results ($m=n=30, 50$)

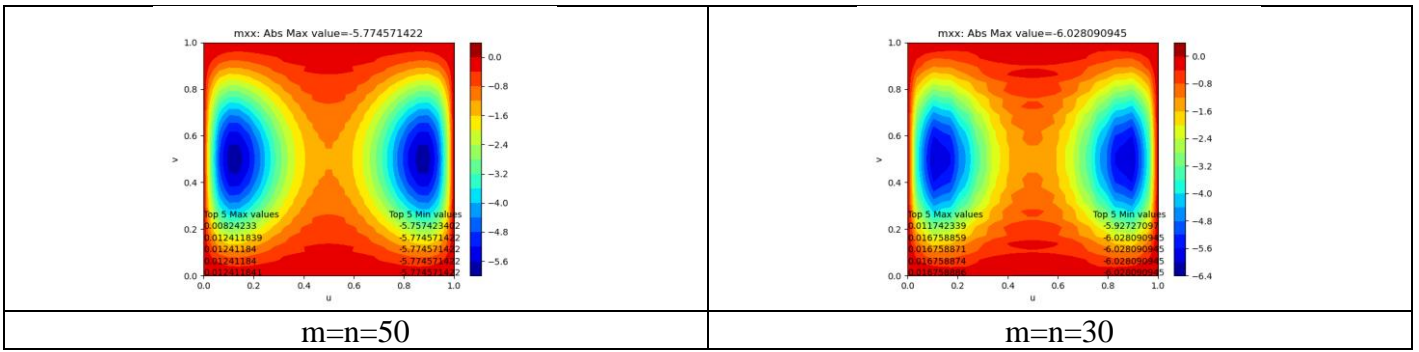


Figure 228: Test AR3 bending moment m_{xx} results ($m=n=30, 50$)

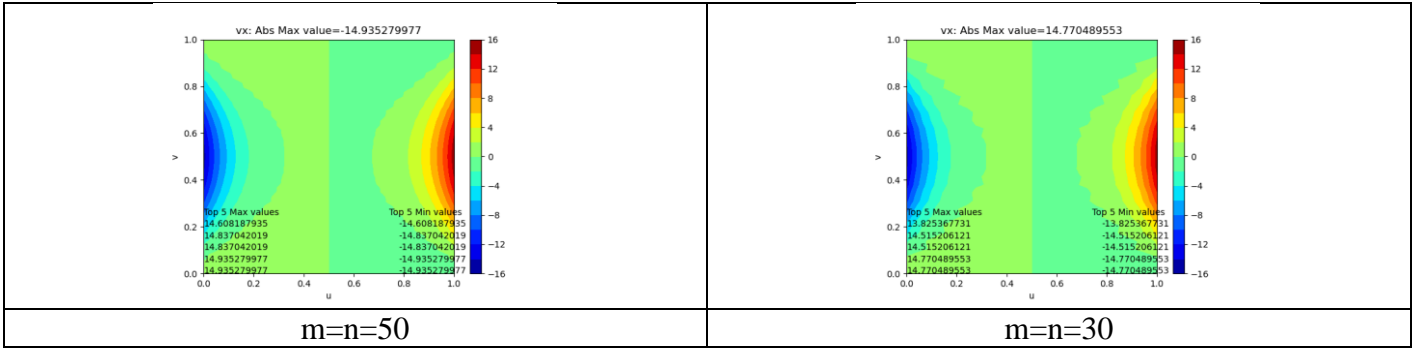


Figure 229: Test AR3 bending moment v_x results ($m=n=30, 50$)

Test AR4, five point interpolation, type A, canopy, vertical loading, rectangular matrix

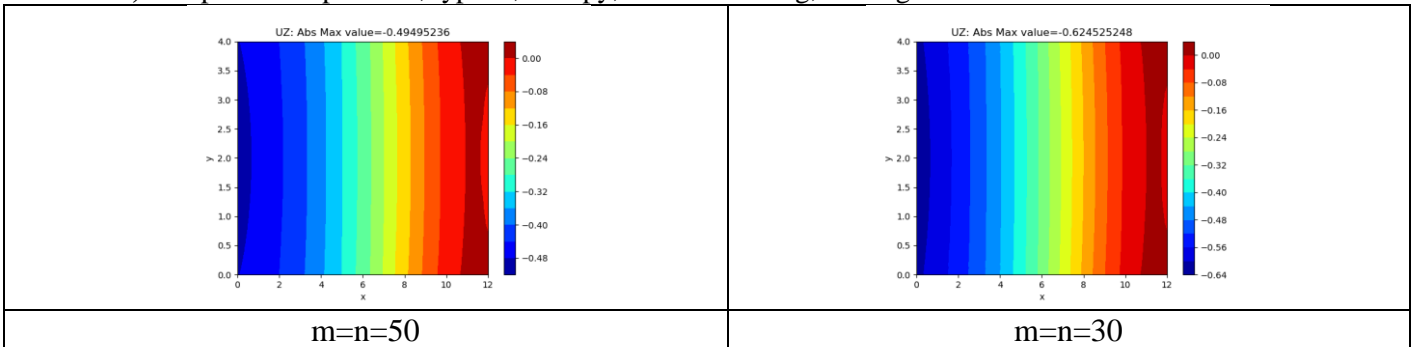


Figure 230: Test AR4 displacement u_z results ($m=n=30, 50$)

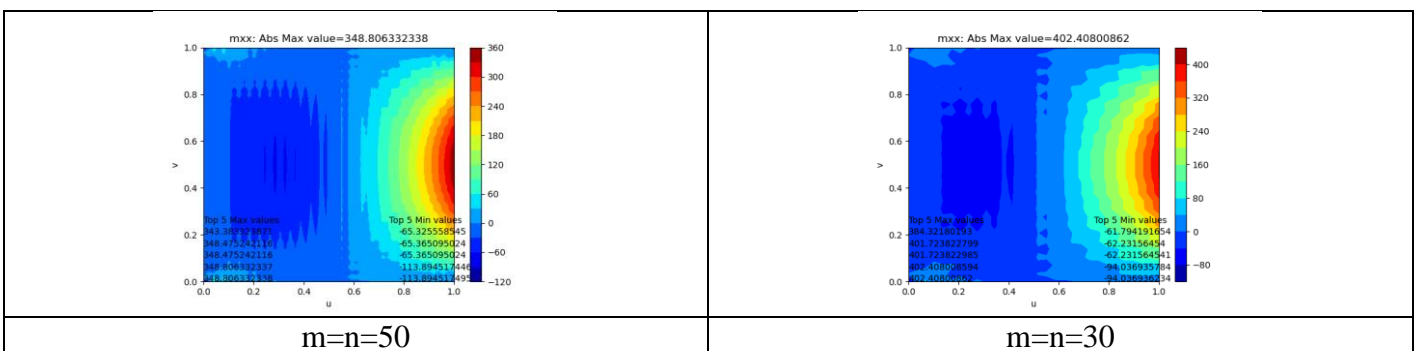


Figure 231: Test AR4 bending moment m_{xx} results ($m=n=30, 50$)

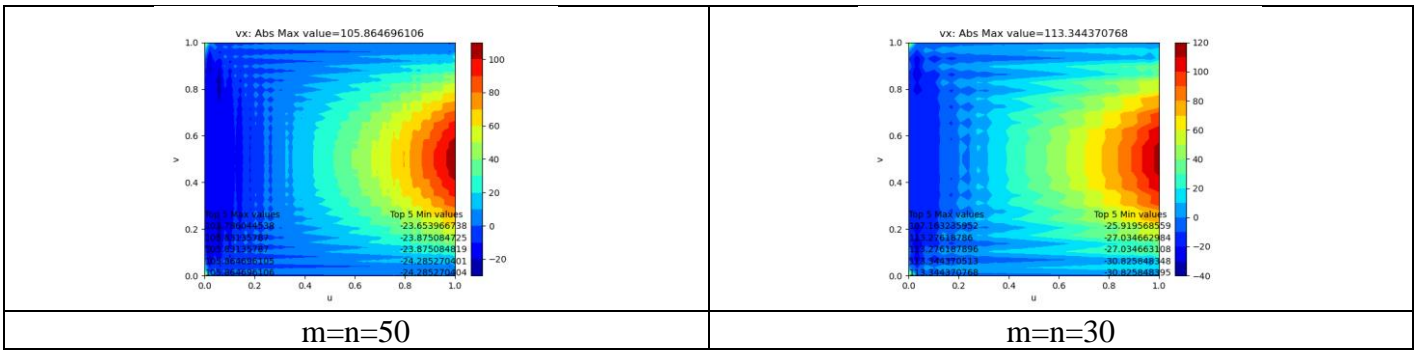


Figure 232: Test AR4 bending moment vx results ($m=n=30, 50$)

Test AR5, five point interpolation, type A, canopy, normal loading, rectangular matrix

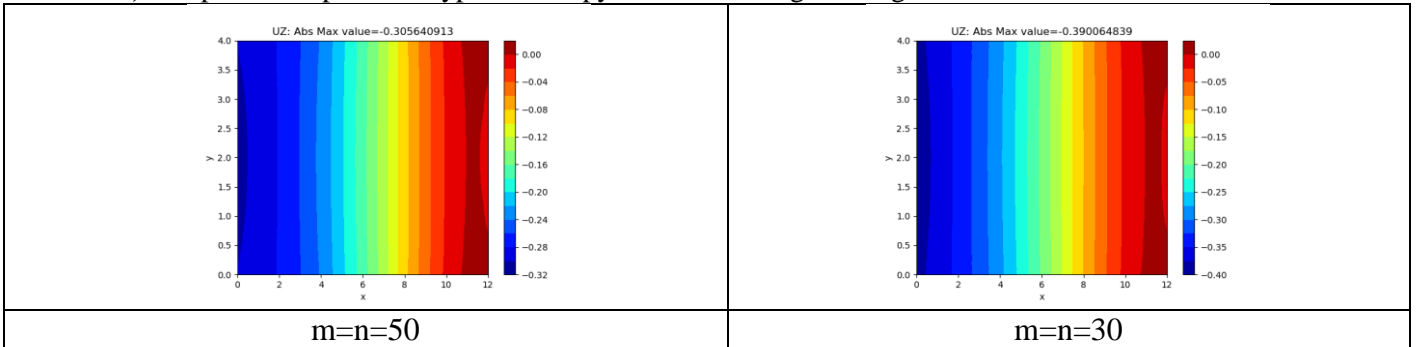


Figure 233: Test AR5 displacement uz results ($m=n=30, 50$)

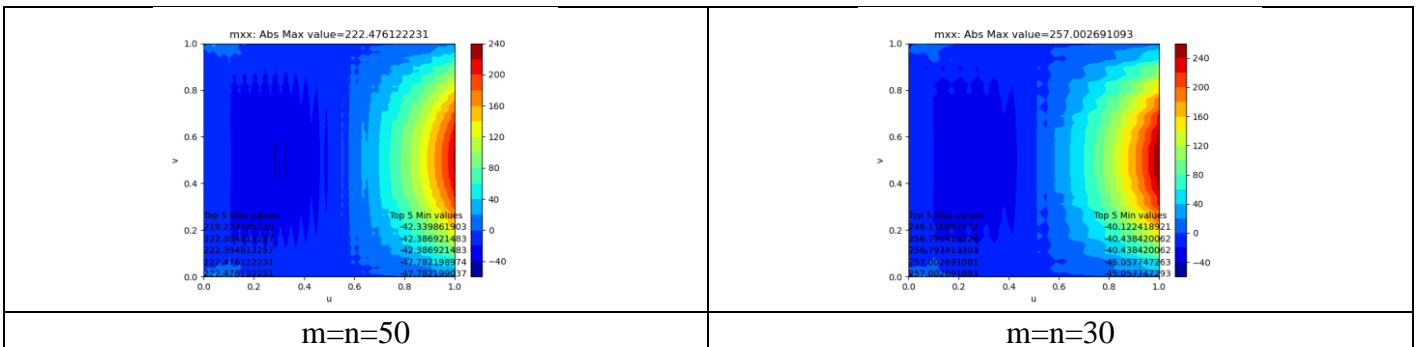


Figure 234: Test AR5 bending moment mxx results ($m=n=30, 50$)

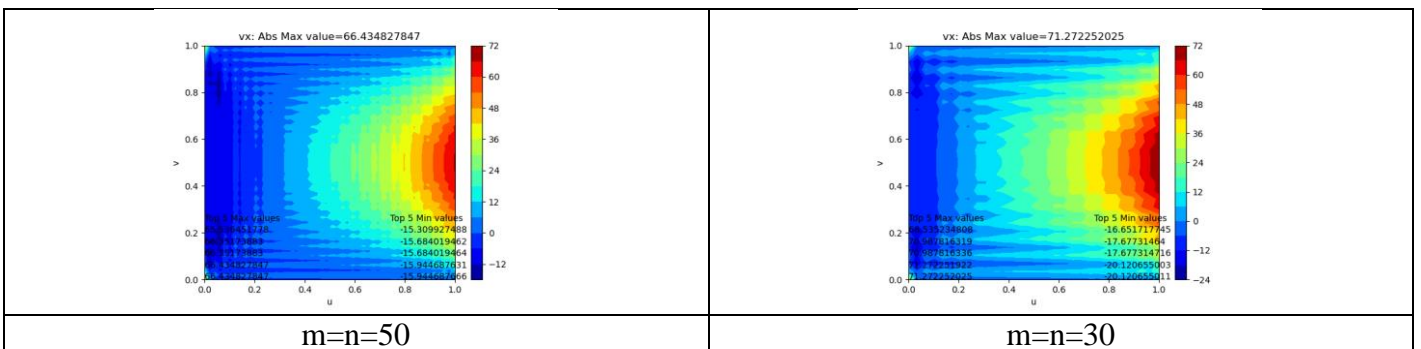


Figure 235: Test AR5 shear force vx results ($m=n=30, 50$)

Test BR1, five point interpolation, type B, flat square, vertical loading, rectangular matrix

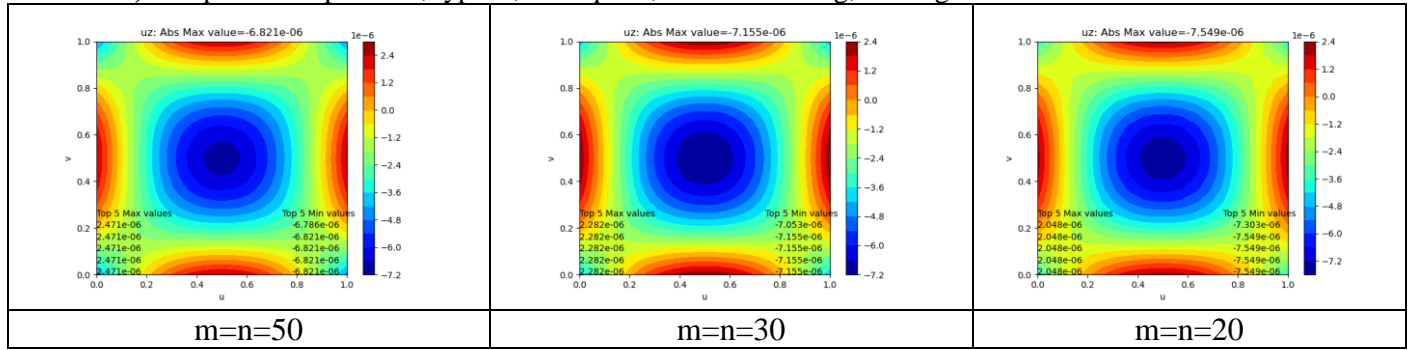


Figure 236: Test BR1 displacement uz results ($m=n= 20, 30, 50$)

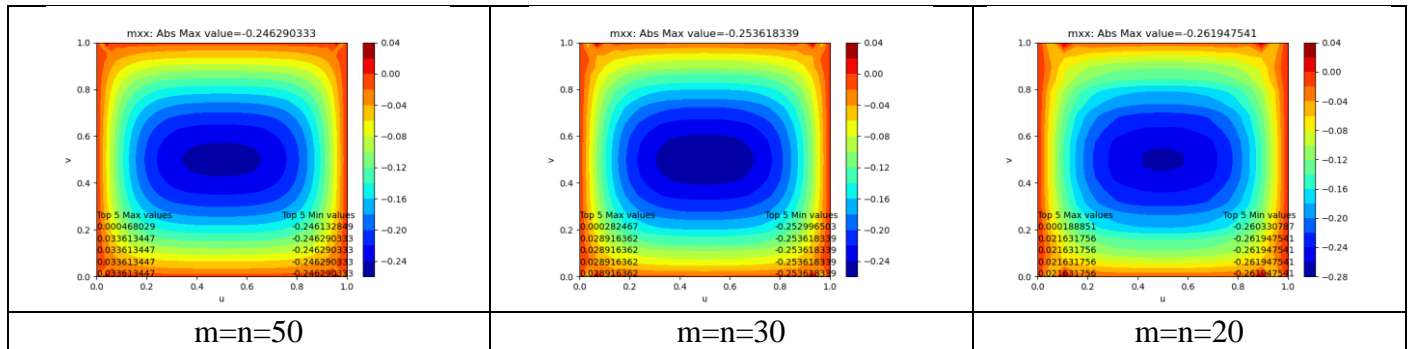


Figure 237: Test BR1 bending moment mxx results ($m=n= 20, 30, 50$)

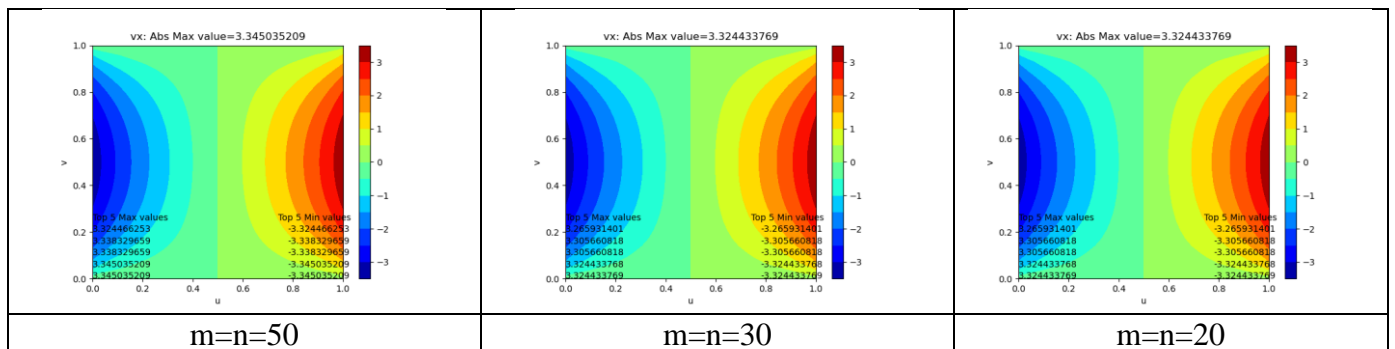


Figure 238: Test BR1 shear force vx results ($m=n=20, 30, 50$)

Test BR2, five point interpolation, type B, flat square, vertical loading, rectangular matrix

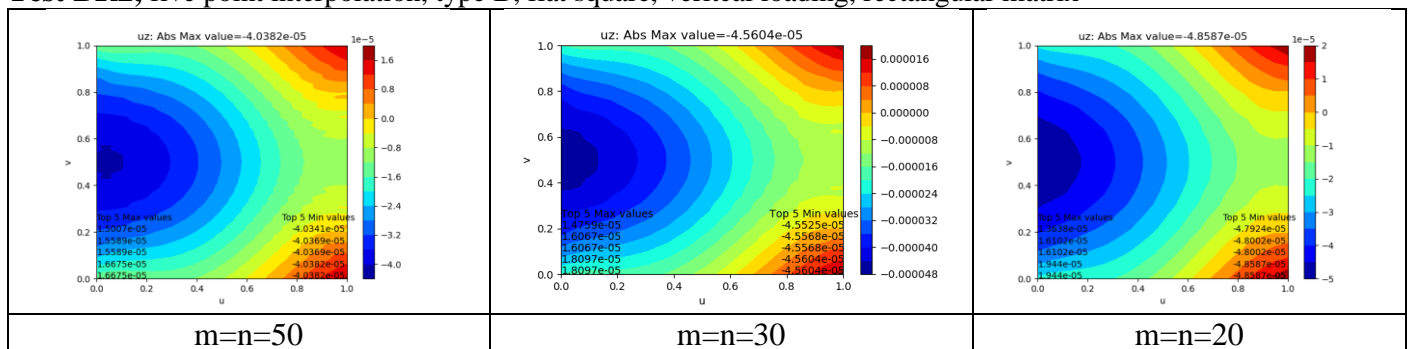


Figure 239: Test BR2 displacement uz results ($m=n=20, 30, 50$)

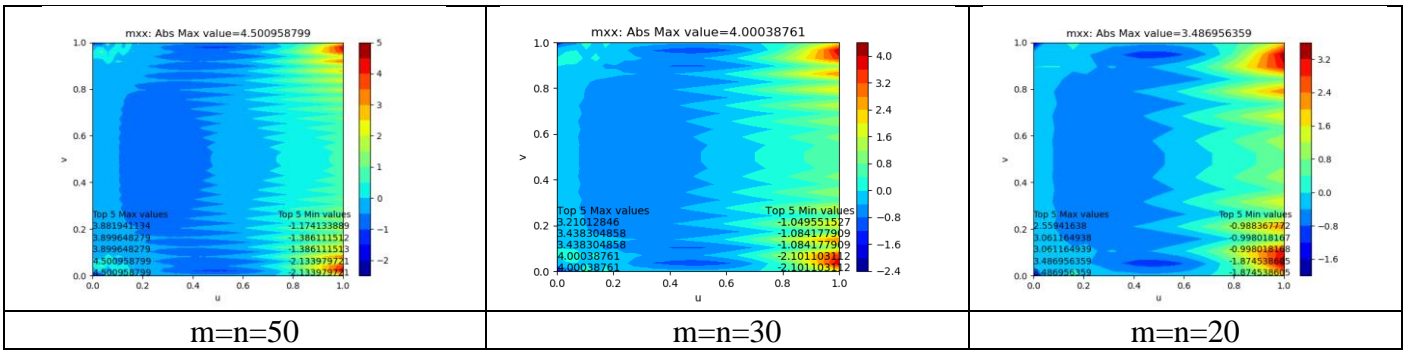


Figure 240: Test BR2 bending moment m_{xx} results ($m=n=20, 30, 50$)

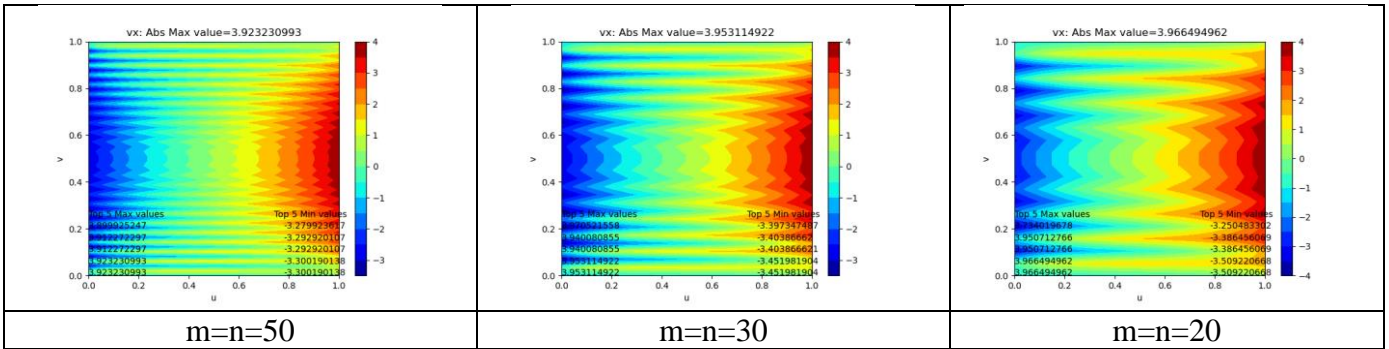


Figure 241: Test BR2 shear force v_x results ($m=n=20, 30, 50$)

Test BR3, five point interpolation, type B, canopy, normal loading, rectangular matrix

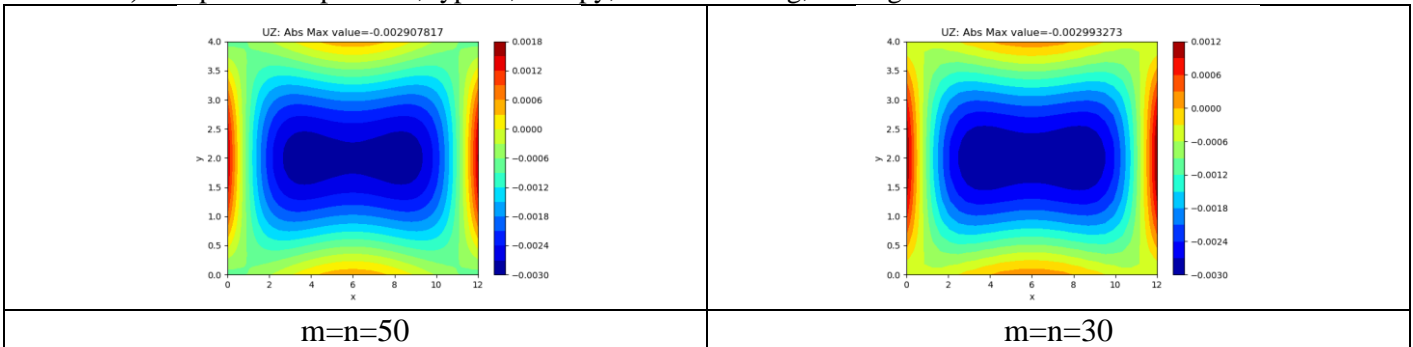


Figure 242: Test BR3 displacement u_z results ($m=n=30, 50$)

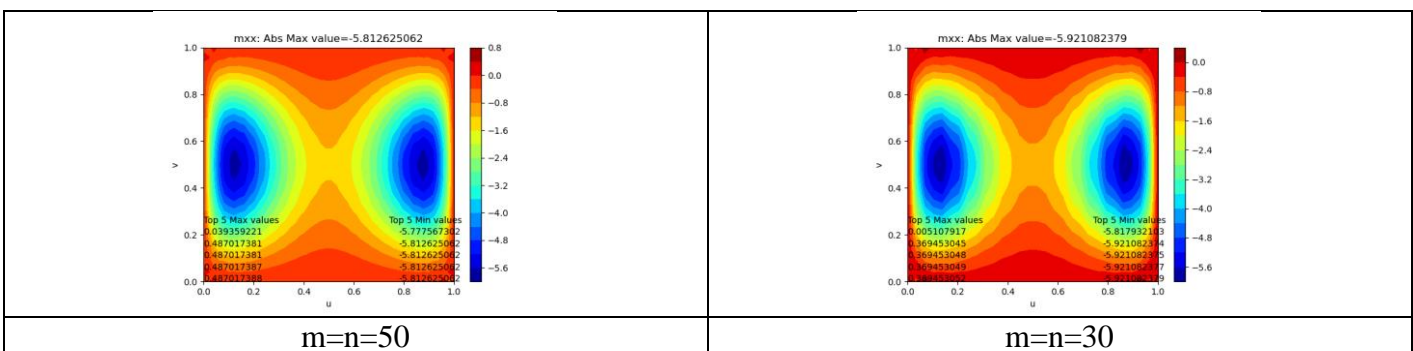


Figure 243: Test BR3 bending moment m_{xx} results ($m=n=30, 50$)

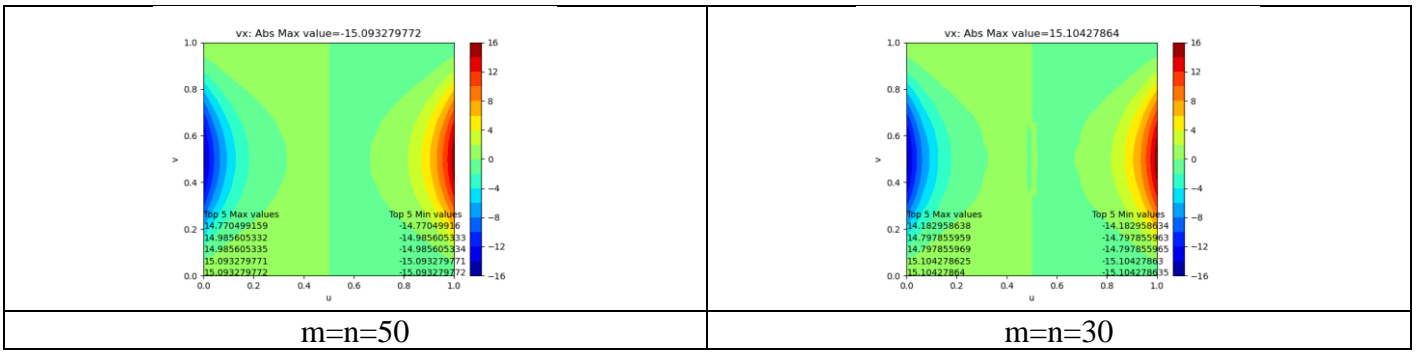


Figure 244: Test BR3 bending moment v_x results ($m=n=30, 50$)

Test BR4, five point interpolation, type B, canopy, vertical loading, rectangular matrix

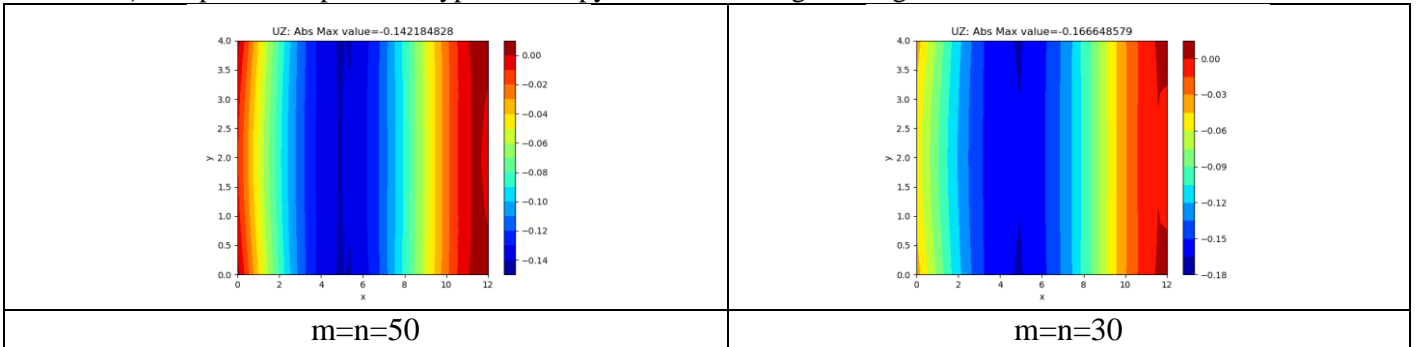


Figure 245: Test BR4 displacement u_z results ($m=n=30, 50$)

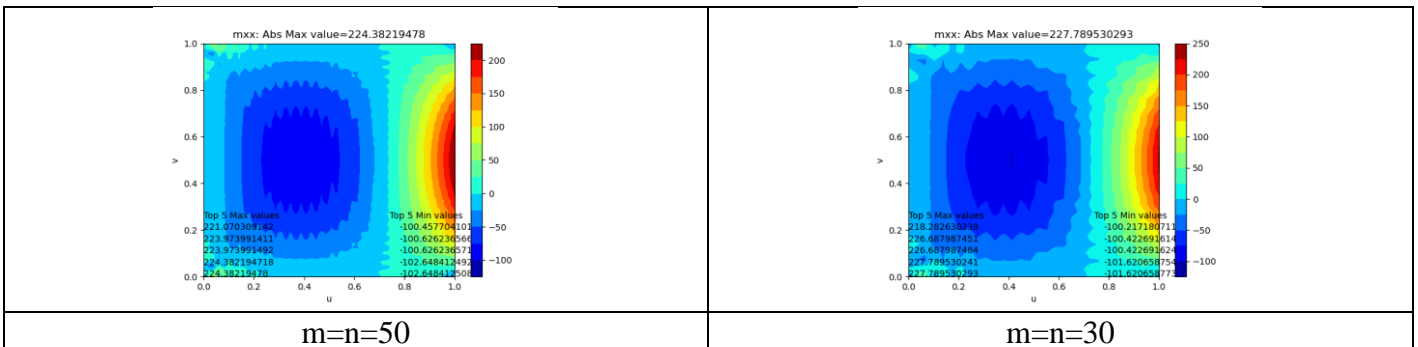


Figure 246: Test BR4 bending moment m_{xx} results ($m=n=30, 50$)

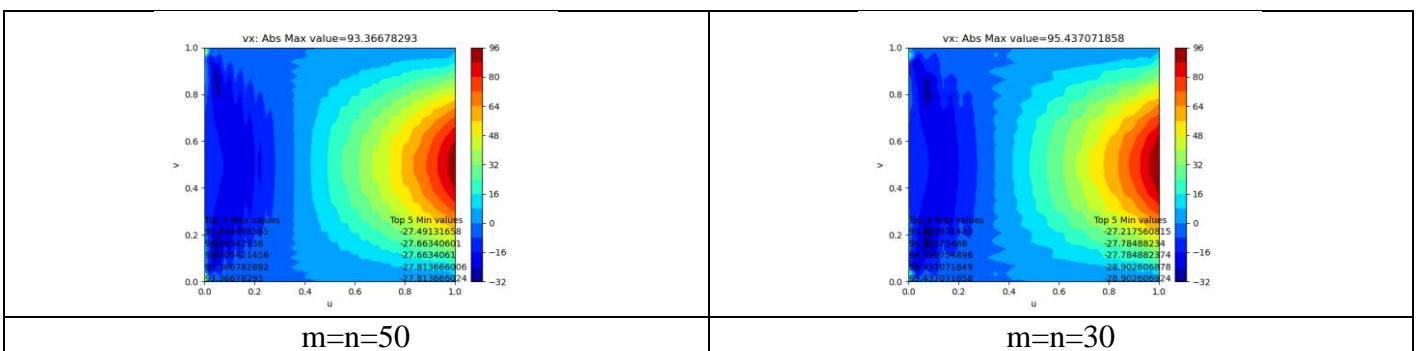


Figure 247: Test BR4 bending moment v_x results ($m=n=30, 50$)

Test BR5, five point interpolation, type B, canopy, normal loading, rectangular matrix

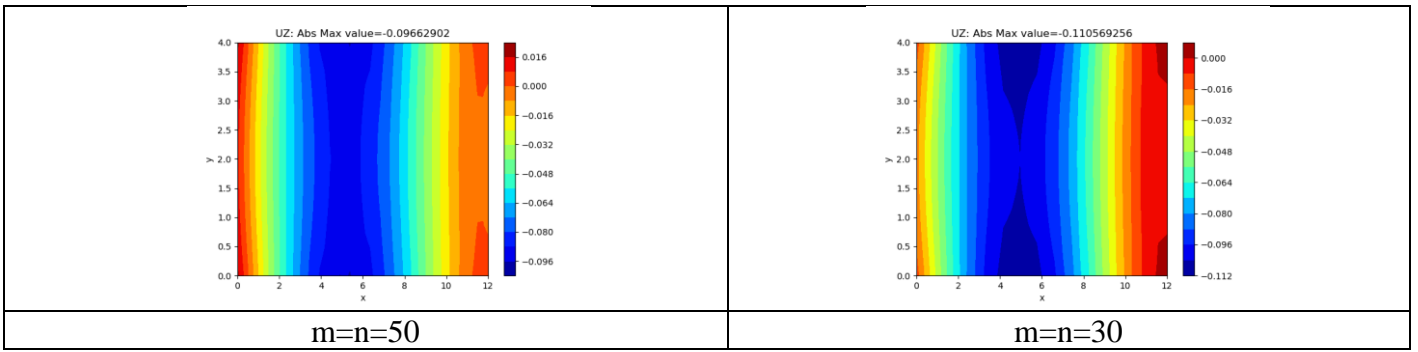


Figure 248: Test BR5 displacement uz results ($m=n=30, 50$)

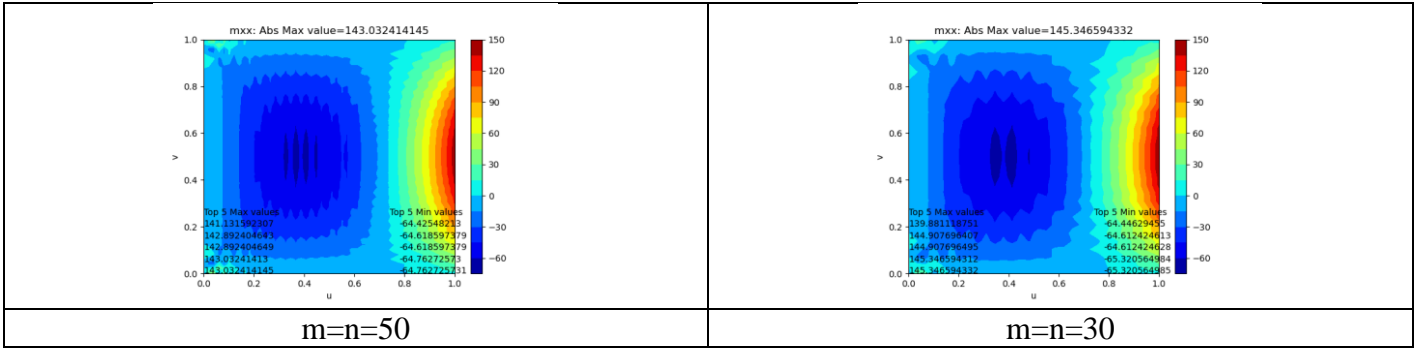


Figure 249: Test BR5 bending moment mxx results ($m=n=30, 50$)

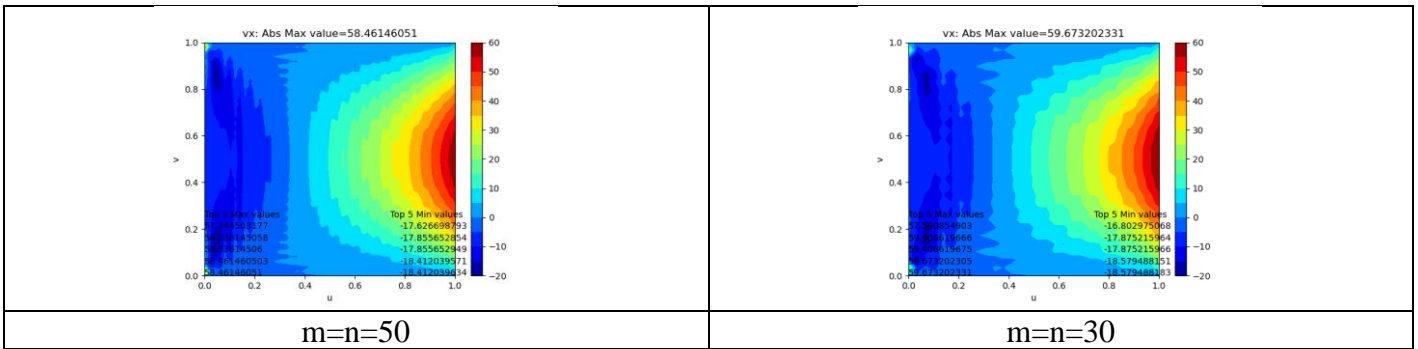


Figure 250: Test BR5 shear force vx results ($m=n=30, 50$)

Matrix quality check

Table 51: Sparsity of rectangular matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
20*20	400*8708	99.99675%	99.99675%	99.99699%	99.99700%	99.99700%
30*30	18900*19360	99.99854%	99.99854%	99.99864%	99.99865%	99.99865%
50*50	52500*53288	99.99947%	99.99947%	99.99951%	99.99951%	99.99951%

Table 52: Sparsity of square matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
20*20	400*8708	99.99675%	99.99675%	99.99699%	99.99700%	99.99700%
30*30	18900*19360	99.99854%	99.99854%	99.99864%	99.99865%	99.99865%
50*50	52500*53288	99.99947%	99.99947%	99.99951%	99.99951%	99.99951%

Table 53: Condition number of rectangular matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
10*10	2100*2240	4.197e+16	2.887e+16	1.256e+16	6.458e+15	6.458e+15

12*12	3024*3204	5.739e+16	3.804e+16	1.785e+16	9.142e+16	9.1428e+15
15*15	4725*4953	8.69E+16	6.05E+16	3.04E+16	1.71E+16	1.71E+16

Table 54: Condition number of square matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
10*10	2100*2100	3.809e+16	2.398e+16	3.851e+16	1.463e+16	1.463e+16
12*12	3024*3024	4.386e+16	3.424e+16	5.560e+16	1.966e+16	1.966e+16
15*15	4725*4725	6.83E+16	5.59E+16	7.60E+16	3.96E+16	3.96E+16

Table 55: Rank number of rectangular matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
10*10	2100*2240	1930	1998	2040	2083	2083
12*12	3024*3204	2761	2840	2919	2973	2973
15*15	4725*4953	4293	4394	4533	4613	4613

Table 56: Rank number of square matrices

Number of nodes	Matrix size	Model 1	Model 2	Model 3	Model 4	Model 5
10*10	2100*2100	1831	1968	1958	2054	2054
12*12	3024*3024	2639	2796	2816	2943	2943
15*15	4725*4725	4133	4334	4406	4569	4569

Discussion on number of iterations

Table 57: Deviation of Test R1 results by increasing number of iterations ($m=n=20$)

Number of iterations	8400*0.01	8400*0.1	8400*0.2	8400*0.5	8400	8400*2	8400*5	8400*10	8400*100
uz (%)	-80.43	-23.02	-30.74	-30.36	-30.04	-28.88	-24.33	-0.58	-0.77
mxx (%)	-31.15	-45.41	-15.59	-17.92	-17.58	-16.8	-13.93	-1.04	-1.1
vx (%)	-2.69	-2.74	-2.8	-2.79	-2.75	-2.65	-2.26	-2.3	-0.31

Table 58: Deviation of Test R3 results by increasing number of iterations ($m=n=20$)

Number of iterations	8400*0.01	8400*0.1	8400*0.2	8400*0.5	8400	8400*2	8400*5	8400*10	8400*100
uz (%)	291.42	1268.85	1402.11	1111.91	457.65	394.1	251.92	218.78	219.98
mxx (%)	1116.95	2544.8	2544.5	2484.44	1485.23	1413.49	978.81	785.59	194.43
vx (%)	372.75	531.91	531.88	465.89	328.5	294.73	219.74	174.21	-3.52

Table 59: Deviation of Test SE1 results by increasing number of iterations ($m=n=20$)

Number of iterations	8400*0.01	8400*0.1	8400*0.2	8400*0.5	8400	8400*2	8400*5	8400*10	8400*100
uz (%)	-81.72	-29.88	-31.3	-35.71	-34.09	-29.12	-25.2	-23.17	-1.09
mxx (%)	-29.86	-43.79	-43.79	-19.15	-20.89	-16.76	-14.68	-13.47	-1.28
vx (%)	-9.37	-8.93	-8.93	-8.46	-7.91	-5.89	-4.94	-4.23	-2.62

Table 60: Deviation of Test SE3 results by increasing number of iterations ($m=n=20$)

Number of iterations	8400*0.01	8400*0.1	8400*0.2	8400*0.5	8400	8400*2	8400*5	8400*10	8400*100
uz (%)	-66.48	1364.72	1473.82	1470.87	844.86	450.33	235.25	232.59	215.72
mxx (%)	195.46	349.78	349.54	349.54	295.72	185.47	112.73	232.59	-48.07
vx (%)	196.8	313.6	313.58	313.56	246.16	188.37	142.86	132.66	-16.04

Discussion on unit system

Table 61: Deviation of Test R1-5 results by new unit systems (N, mm)

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-48.51%	-98.15%			
	30*30	-34.39%	-99.37%	-75.95%	-99.67%	-99.87%
	50*50	-97.59%	-99.72%	-78.16%	-99.77%	-99.91%
Bending moment (kNm/m)	20*20	-66.27%	-97.97%			
	30*30	-69.74%	-98.50%	-97.83%	-99.63%	-99.92%
	50*50	-89.42%	-98.90%	-98.27%	-99.71%	-99.89%
Shear force (kN/m)	20*20	198.67%	-85.05%			
	30*30	375.13%	-84.11%	-87.47%	-96.60%	-98.67%
	50*50	146.14%	-81.11%	-90.00%	-96.99%	-97.57%

Table 62: Deviation of Test LMI-5 results by new unit systems (N, mm)

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-0.76%	-0.09%			
	30*30	-0.32%	-0.05%	-15.34%	-37.09%	14.76%
	50*50	-0.11%	-0.02%	-11.91%	-37.03%	27.10%
Bending moment (kNm/m)	20*20	-1.08%	1.69%			
	30*30	-0.38%	0.62%	270.58%	322.73%	168.03%
	50*50	-0.02%	0.11%	49.89%	228.76%	170.82%
Shear force (kN/m)	20*20	-0.41%	74.44%			
	30*30	-0.05%	256.73%	89.18%	84.83%	14.73%
	50*50	0.13%	639.12%	23.96%	65.69%	41.51%

Table 63: Deviation of Test R1-5 results by new unit systems (KN, 10m)

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	-30.32%	-60%			
	30*30	-33.74%	-53.09%	71950.46%	153287.15%	71051.63%
	50*50	-36.66%	-38.84%	71210.84%	125441%	67532.01%
Bending moment (kNm/m)	20*20	-17.58%	251.83%			
	30*30	-19.27%	315.03%	25888.83%	24437.23%	7744.13%
	50*50	-20.65%	398.61%	23656.46%	27326.24%	8840.20%
Shear force (kN/m)	20*20	-2.92%	-81.81%			
	30*30	-1.68%	-81.58%	766.57%	276.72%	89.10%
	50*50	-0.85%	-76.64%	781.53%	300.93%	103.48%

Table 64: Deviation of Test LMI-5 results by new unit systems (KN, 10m)

	Number of nodes	Model 1	Model 2	Model 3	Model 4	Model 5
Displacement (m)	20*20	6.77%	5.97%			
	30*30	0.56%	4.58%	-16.54%	-31.57%	41.60%
	50*50	-0.28%	1.69%	-12.91%	-32.08	40.15%
Bending moment (kNm/m)	20*20	2.97%	-2.73%			
	30*30	0.12%	-2.30%	-9.03%	-71.15%	-50.72%
	50*50	-0.12%	7.02%	-2.15%	-63.75	-26.17%
Shear force (kN/m)	20*20	0.04%	-43.57%			
	30*30	1.64%	-35.13%	-86.75%	-96.30%	-91.75%
	50*50	-2.30%	-12.91%	-82.70%	-92.23	-81.55%