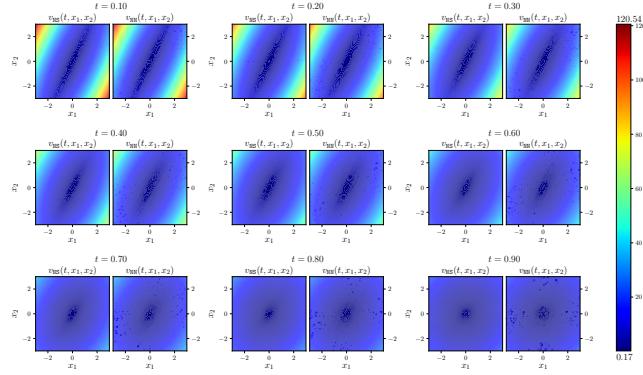


SCDAA Coursework Report



Yuebo Yang*, Yu Wang†, Zheyuan Lu‡

April 11, 2024

The project source code can be found on GitHub
at: <https://github.com/Warren12138/SCDAA>

*s2601126

†s1973765

‡s2559490

1 Linear quadratic regulator

Exercise 1.1 Solving LQR using from Riccati ODE.

According to the requirement, we created a class named `LQR_solver` for solving Linear quadratic regulator, which mainly contains 3 methods listed as follows:

1. `solve_riccati_ode`: a numerical solver for Riccati ODE. We offered two different numerical algorithm for solving the ODE for user to choose, which are 1) Euler method and 2) order 4 Runge-Kutta method. The corresponding code is presented below.

```
1 def solve_riccati_ode(self, time_grids):
2     ...
3     time_grids_in = torch.flip(time_grids, dims=[1])
4     S = self.R.clone()
5     repl = torch.ones(time_grids_in.shape)
6     rep_exd = repl.unsqueeze(-1).unsqueeze(-1)
7     S_exd = S.unsqueeze(0).unsqueeze(0)
8     S_repl = rep_exd*S_exd
9     dt = time_grids_in[:,1:]-time_grids_in[:,::1]
10
11    for i in range(dt.shape[1]):
12
13        if self.method == "euler":
14            S_repl[:,i+1] = self.euler_step(S_repl[:,i], dt[:,i])
15        elif self.method == "rk4":
16            S_repl[:,i+1] = self.rk4_step(S_repl[:,i], dt[:,i])
17        else:
18            raise ValueError("Unsupported method")
19
20    return torch.flip(S_repl, dims=[1])
21
22 def euler_step(self, S_in, dt_in):
23     dS_in = -2 * self.H.T @ S_in + S_in @ self.M @ torch.inverse(self.D) @ self.M.T @
24         S_in - self.C
25     dt_resized = dt_in[:, None, None]
26
27     return S_in + dS_in * dt_resized
28
29 def rk4_step(self, S_in, dt_in):
30     def riccati_derivative(S):
31         return -2 * self.H.T @ S_in + S_in @ self.M @ torch.inverse(self.D) @ self.M.T
32             @ S_in - self.C
33     dt_resized = dt_in[:, None, None]
34     k1 = riccati_derivative(S_in)
35     k2 = riccati_derivative(S_in + 0.5 * k1 * dt_resized)
36     k3 = riccati_derivative(S_in + 0.5 * k2 * dt_resized)
37     k4 = riccati_derivative(S_in + k3 * dt_resized)
38
39     return S_in + (k1 + 2*k2 + 2*k3 + k4)*dt_resized/6
```

2. `value_function`: computation of the value function $v(t, x)$, for which we developed two strategies to calculate $S(t)$, an essential component in determining $v(t, x)$. Our

preferred method involves computing a uniformly discrete $S(t)$ across the entire time horizon $[0, T]$. Subsequently, we employed cubic spline interpolation of $S(t)$ to efficiently compute $v(t, x)$ for each distinct (t, x) within a single batch of inputs. This approach guarantees a consistent level of numerical error across computations. The corresponding code is presented below.

```

42     time_grid_after_t1 = time_grid_for_intpl[t1_index:]
43     dt_t1 = time_grid_after_t1[0] - t_batch[i]
44     dt_after_t1 = time_grid_after_t1[1:]-time_grid_after_t1[:-1]
45     traces_after_t1 = traces[t1_index:]
46     traces_after_t1_for_int = torch.tensor(0.5,dtype = torch.double) *
47         (traces_after_t1[1:] + traces_after_t1[:-1])
48     int_t1 = torch.tensor(0.5,dtype = torch.double) *
49         (torch.from_numpy(trace_cubic(t_batch_np[i])).to(dtype = torch.double)
50          + traces_after_t1[0])
51     intgl = (dt_t1*int_t1).squeeze() + (dt_after_t1.view(1,
52         -1)@traces_after_t1_for_int.squeeze(1))
53
54     xTSx[i,0]+= intgl.squeeze()
55
56     v_tx = xTSx.squeeze()
57
58     ...
59
60
61     return v_tx

```

3. `markov_control`: a computation of Markov control function $a(t, x)$, which shares the same strategies and structure of coding as `value_function`. The corresponding code is presented below.

```

1 def markov_control(self, t_batch, x_batch, sol_method = 'interpolation'):
2
3     if sol_method == 'interpolation':
4
5         N_step = 2*self.N_step
6
7         if not (t_batch.dim() == 1 and torch.all((t_batch >= 0) & (t_batch <= 1))):
8             raise TypeError("t_batch should be a 1D tensor in which every entry is in
9                             [0,1].")
10            else:
11                if not (x_batch.dim() == 3 and x_batch.size()[0] == len(t_batch) and
12                    x_batch.size()[1] == 1 and x_batch.size()[2] == self.H.size()[0]):
13                    raise TypeError("x_batch should have shape (%d, 1,
14                                     %d)." %(len(t_batch),self.H.size(2)))
15
16                time_grid = torch.stack([torch.linspace(0, self.T, N_step, dtype =
17                                         torch.double) for i in [0]])
18
19                S_tensor_tensor = self.solve_riccati_ode(time_grid)
20
21                index_s_1 = torch.searchsorted(time_grid[0,:], torch.min(t_batch), right=True
22                                              - 1
23                time_grid_for_intpl = time_grid[0,index_s_1:]
24                S_tensor_tensor_for_intpl = S_tensor_tensor[0,index_s_1:]
25
26                time_grid_for_intpl_np = time_grid_for_intpl.numpy()
27                S_tensor_tensor_for_intpl_np = S_tensor_tensor_for_intpl.numpy()
28                t_batch_np = t_batch.numpy()
29
30
31
32
33
34
35
36
37
38
39
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
129
130
131
132
133
134
135
136
137
138
139
139
140
141
142
143
144
145
146
147
148
149
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
167
168
169
169
170
171
172
173
174
175
176
177
177
178
179
179
180
181
182
183
184
185
186
186
187
188
188
189
189
190
191
192
193
194
195
196
196
197
198
198
199
199
200
201
202
203
203
204
204
205
205
206
206
207
207
208
208
209
209
210
210
211
211
212
212
213
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
```

```

25     S_c_spl = CubicSpline(time_grid_for_intpl_np, S_tensor_tensor_for_intpl_np)
26
27     def S_intpl(t_batch_np_in):
28         return torch.from_numpy(S_c_spl(t_batch_np_in))
29
30     S_t_s = S_intpl(t_batch_np)
31     x_batch_T = x_batch.transpose(1, 2)
32
33     MT = self.M.T
34     D_inv = torch.inverse(self.D)
35     x = torch.transpose(x_batch, dim0 = 2, dim1 = 1)
36     a_tx = - D_inv @ MT @ S_t_s @ x_batch_T
37     a_tx = torch.transpose(a_tx, dim0 = 1, dim1 = 2).squeeze()
38
39     ...
40
41     return a_tx.unsqueeze(1)

```

Exercise 1.2 LQR MC check.

Utilizing the LQR solver from [Exercise 1.1](#), we employed Monte Carlo simulation to compare the outcomes with those obtained in [Exercise 1.1](#). Specifically, we varied the number of Monte Carlo samples and adjusted the number of time steps to explore how these changes affect the simulation results.

Initialization

We set the required matrices to values as follows:

$$H = \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, \quad M = \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, \quad C = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix},$$

$$D = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, \quad R = \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma = \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}.$$

This part in our Python code is presented as follows, where the shapes of the vectors are adjusted due to the convenience of matrix multiplication.

```

1 H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = torch.double)
2 M = torch.tensor([[0.5, 0.7], [0.3, 1.0]], dtype = torch.double)
3 sigma = torch.tensor([[0.08], [0.11]]], dtype = torch.double)
4 C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = torch.double)
5 D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = torch.double)
6 R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = torch.double)
7 T = torch.tensor(1.0, dtype = torch.double)

```

Then, we first planned to do this comparison on a wider range of the variable space, which is

$$\begin{aligned} t &\in \{0.1, 0.2, \dots, 0.9\}, \\ x_1 &\in \{0.1, 0.2, \dots, 0.9\}, \\ x_2 &\in \{0.1, 0.2, \dots, 0.9\}, \end{aligned}$$

i.e.,

$$\begin{aligned} \{(t, x)^i, i = 1, \dots, N\} &= \{0.1, 0.2, \dots, 0.9\} \times \{0.1, 0.2, \dots, 0.9\}^2, \\ N &= 9 \times 9 \times 9. \end{aligned}$$

Finally, due to the limitation of our computational resources, we chose a much smaller one, which is

$$\begin{aligned} t &\in \{0.1\}, \\ x_1 &\in \{0.5\}, \\ x_2 &\in \{0.5\}, \end{aligned}$$

i.e.,

$$\begin{aligned} \{(t, x)^i, i = 1\} &= \{0.1\} \times \{0.5\}^2, \\ N &= 1. \end{aligned}$$

Setting for Monte Carlo Simulation

In order to analyze the two parameters' impact on the accuracy of Monte Carlo simulation, we divided our calculation tasks into two parts:

1. Fix sampling size (`FSS = int(1e5)`), and variate time step number (`FSS_VTSN = [int(x) for x in [1e0, 1e1, 5e1, 1e2, 5e2, 1e3, 5e3]]`)
2. Fix time step number (`FTSN = int(5e3)`), and variate sampling size (`FTSN_VSS = [int(x) for x in [1e1, 5e1, 1e2, 5e2, 1e3, 5e3, 1e4, 5e4, 1e5]]`)

For all computation tasks, we crafted a shell script named `run_MCs.sh`, which is executed in the terminal. This script invokes `Exercise1_2_parallel_MC.py`, our custom Python script designed for conducting the Monte Carlo simulation's main process. It performs the simulation through parallel computation and ultimately generates the corresponding data files storing the $J^\alpha(t, x)$. These files are then organized into subfolders, each named according to different parameter settings.

One of our group members developed both the explicit scheme, utilizing a `for` loop approach, and the implicit scheme, through direct computation of a massive sparse matrix's inverse. For the implicit scheme, we further optimized the process by decomposing the large sparse matrix into relatively smaller blocks to facilitate the computation of its inverse.

The corresponding codes of the two schemes is presented below.

1. Explicit scheme:

```

1 def MonteCarloSampler(iteration, params):
2     #Explicit scheme
3     C = params['C']
4     D = params['D']
5     N = params['N']
6     R = params['R']
7     S = params['S']
8     X0 = params['X0']
9     H = params['H']
10    dt = params['dt']
11    multX = params['multX']
12    multa = params['multa']
13    sig = params['sig']
14
15    X_0_N = X0
16
17    for i in range(N):
18
19        X_next = ((torch.eye(2) + dt[i] * (H + multX@S[i])) @ X_0_N[i:]).transpose(1,2)
20            + sig * torch.sqrt(dt[i]) * torch.randn(1)).transpose(1,2)
21        X_0_N = torch.cat((X_0_N, X_next), dim=0)
22
23    alp = multa @ X_0_N.transpose(1,2)
24
25    int_ = X_0_N @ C @ X_0_N.transpose(1,2) + alp.transpose(1,2) @ D @ alp
26    J = X_0_N[-1] @ R @ X_0_N[-1].T + torch.tensor(0.5)*dt @
27        ((int_.squeeze(1)[1:]+int_.squeeze(1)[:1]))
28
29    return J

```

2. Implicit scheme:

```

1 def implicit(it,params):
2
3     #Implicit code by Yuebo Yang Mar.25.2024
4
5     device = params['device']
6     AA = params['AA'].to(device)
7
8     C = params['C'].to(device)
9     D = params['D'].to(device)
10    N = params['N']
11    R = params['R'].to(device)
12
13    Step_limit = params['Step_limit']
14    X0 = params['X0'].to(device)
15    dt = params['dt'].to(device)
16    multa = params['multa'].to(device)
17    sig = params['sig'].to(device)
18
19    b = torch.cat((X0.squeeze(),sig.squeeze().repeat(N-1) *
20                  torch.sqrt(dt).repeat_interleave(len(X0.squeeze())))*

```

```

19         torch.randn(len(sig.squeeze().repeat(N-1)), dtype=torch.float32, device =
20             device)))
21
22     if N//Step_limit >= 1:
23
24         X_0_N = torch.clone(X0.squeeze().repeat(N))
25
26         for i in range(N//Step_limit):
27
28             b_in_batch = torch.clone(b[i*Step_limit*2:(i+1)*Step_limit*2])
29
30             if i == 0:
31                 _X_0 = torch.zeros_like(b_in_batch[:2], dtype = torch.float32, device =
32                     device)
33             else:
34                 _X_0 = torch.clone(X_0_N[i*(Step_limit)*2-2:i*(Step_limit)*2])
35
36             b_in_batch[:2] = b_in_batch[:2]+_X_0
37
38             AA_in_batch = torch.clone(AA[i * Step_limit * 2 : (i+1) * Step_limit * 2,
39                             i * Step_limit * 2 : (i+1) * Step_limit * 2])
40
41             X_0_N[i * Step_limit * 2 : (i+1) * Step_limit * 2] =
42                 (torch.inverse(AA_in_batch) @ b_in_batch)
43
44             if (N%Step_limit != 0):
45                 _X_0 = torch.clone(X_0_N[-(N % Step_limit + 1) * 2 : - (N % Step_limit) *
46                     2])
47
48             #final_section
49
50             b_fin = torch.clone(b[-(N % Step_limit) * 2 : ])
51             b_fin[:2] = torch.clone(b_fin[:2]+_X_0)
52             AA_fin = AA[-(N%Step_limit)*2:,-(N%Step_limit)*2:]
53
54             X_0_N[-(N%Step_limit)*2:] = (torch.inverse(AA_fin)@b_fin)
55
56             X_0_N = X_0_N.reshape(N,1,2)
57
58     else:
59
60         X_0_N = (torch.inverse(AA)@b).reshape(N,1,2)
61
62         alpha = multa@X_0_N.transpose(1,2)
63
64         int_ = X_0_N@C@X_0_N.transpose(1,2) + alpha.transpose(1,2)@D@alpha
65
66         J = X_0_N[-1]@R@X_0_N[-1].T + torch.tensor(0.5, dtype=torch.float32, device =
67             device)*dt@((int_.squeeze(1)[1:]+int_.squeeze(1)[:-1]))
68
69         return J.to('cpu')

```

Analysis and Conclusion

Upon completing all simulations, we compiled the data across various settings and graphed the squares of the differences between them and the numerical solution.

The evident convergence behavior observed in both fig. 1.1 and fig. 1.2 underscores the accuracy of our Monte Carlo simulation code.

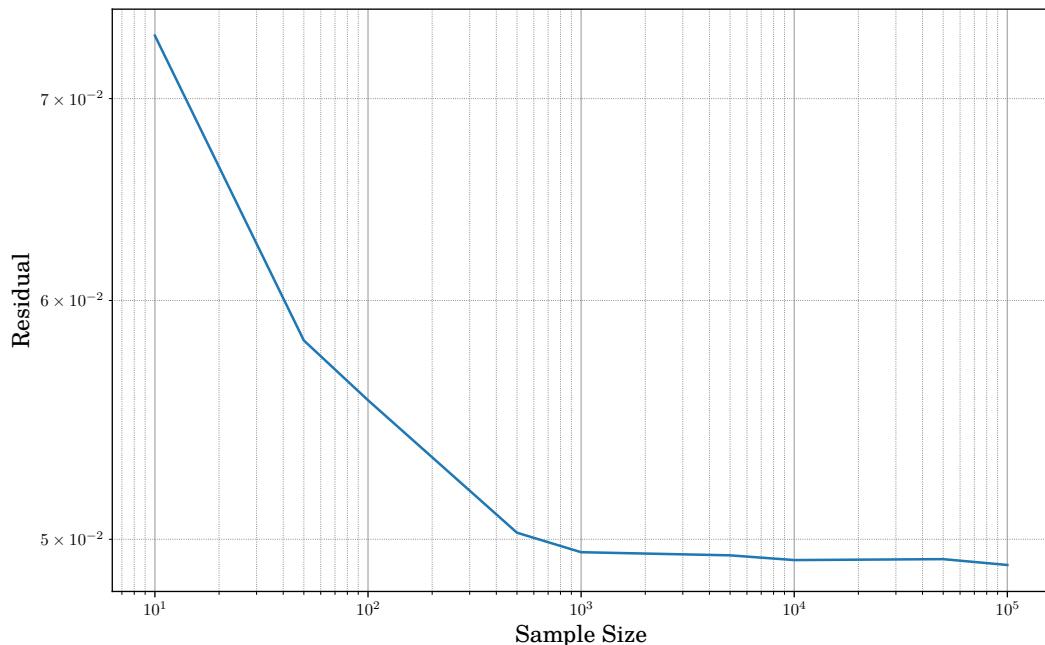


Figure 1.1: Residual convergence over increasing sample size

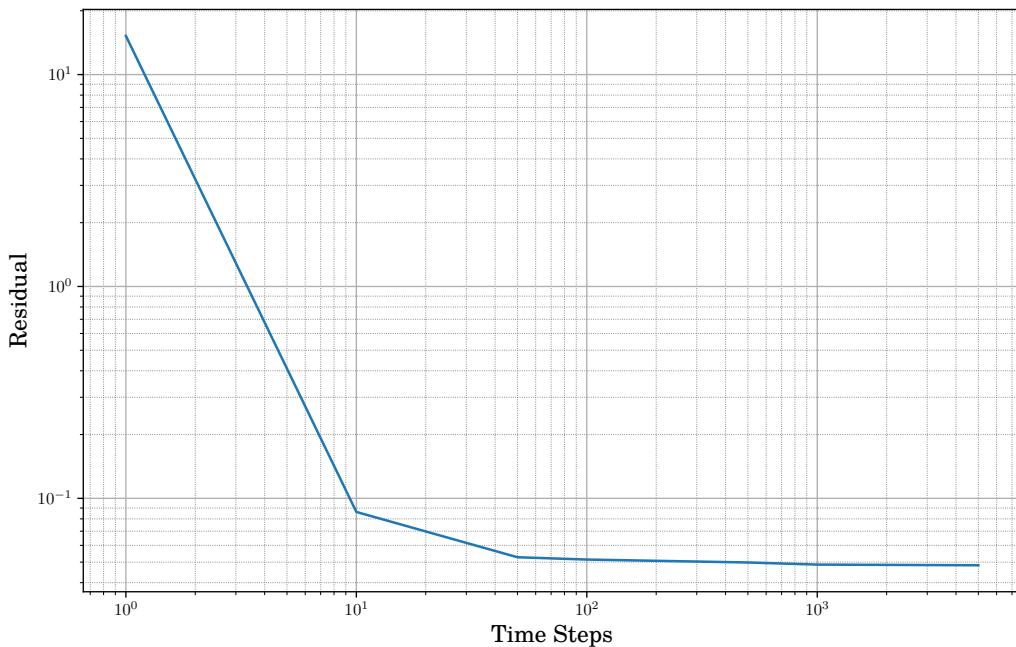


Figure 1.2: Residual convergence over increasing time steps

2 Supervised learning, checking the NNs are good enough

Exercise 2.1 Supervised learning of value function v .

Exercise 2.2 Supervised learning of Markov control a .

Initialization

To approximate the value function, which has been explicitly calculated in **Exercise 1.1**, we employed a neural network model as an estimation tool. This necessitates an initial configuration for the implementation.

$$H = \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, \quad M = \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, \quad C = \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix},$$
$$D = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, \quad R = \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, \quad \sigma = \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}.$$

```
1 H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = torch.double)
2 M = torch.tensor([[0.5,0.7], [0.3,1.0]], dtype = torch.double)
3 sigma = torch.tensor([[[0.8],[1.1]]], dtype = torch.double)
4 C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = torch.double)
5 D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = torch.double)
6 R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = torch.double)
7 T = torch.tensor(1.0, dtype = torch.double)
```

Framework

We trained a neural network model for **Exercise 2.1** in conjunction with the optimizer `torch.optim.Adam` to identify network parameters that yield an accurate approximation of the value functions.

```
1 class ValueFunctionNN(nn.Module):
2     def __init__(self):
3         # Inherit properties from the parent class nn.Module
4         super(ValueFunctionNN, self).__init__()
5         # First layer takes 3 inputs (time t and spatial variables x1 and x2) and
6         # outputs 100 features
7         self.layer1 = nn.Linear(3, 100)
8         # Second and third layers are fully connected layers with 100 neurons each
9         self.layer2 = nn.Linear(100, 100)
10        self.layer3 = nn.Linear(100, 100)
11        # ReLU (Rectified Linear Unit) activation function for introducing
12        # non-linearity
13        self.relu = nn.ReLU()
14        # Output layer: reduces the dimension from 100 to 1, representing the value
15        # function
```

```

13     self.output = nn.Linear(100, 1)
14
15     def forward(self, x):
16         # Defines the forward pass of the neural network
17         # Applies ReLU activation function after each layer except for the output layer
18         x = self.relu(self.layer1(x)) # Input passes through the first layer and ReLU
19         x = self.relu(self.layer2(x)) # Then, it passes through the second layer and
20             ReLU
21         x = self.relu(self.layer3(x)) # Finally, it passes through the third layer and
22             ReLU
23     return self.output(x) # The output layer produces the final value function
24         approximation

```

Upon addressing the computational challenges, we executed the model training employing the following strategic approach:

- i. Set `iteration` = 5. In each `iteration` we set `epochs` = 1000.
- ii. In each `epoch`, we choose a `sample_size` = N to sample the required space $(t, x) \in [0, T] \times [-3, 3]^2$. After we get one batch of this sample points $\{(t, x)^i, i = 1, \dots, N\}$, we can calculate the corresponding $\{(u(t, x))^i, i = 1, \dots, N\}$, and their gradients as well as Hessian matrices.
- iii. Following the expression of $R(\theta)$, we can construct the `loss` function for the training.
- iv. During the training process, we employ `torch.optim.Adam` as our optimizer with an initial `learning_rate` = 0.001.

Subsequently, we proceeded to employ the previously configured neural network model to undertake the data training process.

```

1 # Initializes the model for approximating the value function
2 model_value = ValueFunctionNN().double()
3
4 # Sets up the optimizer with a learning rate of 0.001
5 # Adam optimizer is used for its adaptive learning rate properties
6 optimizer_value = torch.optim.Adam(model_value.parameters(), lr=0.001)
7
8 # Defines the loss function as Mean Squared Error (MSE) to measure
9 # the difference between the predicted and true values
10 criterion_value = nn.MSELoss()
11
12 # Initializes a list to store the loss for each epoch, useful for visualizing the
13     # training process
14 epoch_losses = []
15
16 # Specifies the batch size and number of epochs for training
17 batch_size = 5
18 epochs = 1000
19
20 # Training loop over the specified number of batches
21 for batch in range(batch_size):
22     print(f'Batch {batch+1}/{batch_size}'+ '\n')

```

```

23 # Generates new data for training
24 t_data, x_data, v_data = new_data(10000, 'value')
25
26 # Prepares the input by concatenating time and spatial data
27 inputs = torch.cat((t_data.unsqueeze(-1), x_data.squeeze(1)), dim=1)
28
29 # Creates a DataLoader for batch processing
30 dataset = TensorDataset(inputs, v_data)
31 dataloader = DataLoader(dataset, batch_size=512, shuffle=True)
32
33 # Iterates over each epoch for training
34 for epoch in range(epochs):
35     model_value.train() # Sets the model to training mode
36     total_loss = 0
37
38     # Iterates over each mini-batch
39     for batch_idx, (data, target) in enumerate(dataloader):
40         optimizer_value.zero_grad() # Clears the gradients of all optimized tensors
41         output = model_value(data) # Forward pass: computes the model's output
42         loss = criterion_value(output, target.unsqueeze(1)) # Computes the loss
43         loss.backward() # Backward pass: computes the gradient of the loss w.r.t.
44             # the parameters
45         optimizer_value.step() # Updates the parameters
46         total_loss += loss.item() # Aggregates the loss
47
48     # Stores the average loss for this epoch
49     epoch_losses.append(total_loss / len(dataloader))
50
51     # Prints loss information at specific epochs
52     if epoch == 0 or (epoch + 1) % 100 == 0:
53         print(f'Epoch {epoch+1}/{epochs} \t Loss: {total_loss / len(dataloader)}')
54
55 print('\n')

```

In this section, we have only demonstrated the framework and training process code for the value function aspect. The setup and training for the Markov control are analogous.

Analysis and Conclusion

Figure 2.1 illustrates a discernible decline in the training loss for the value function approximation as the training progresses, indicating continuous convergence across successive

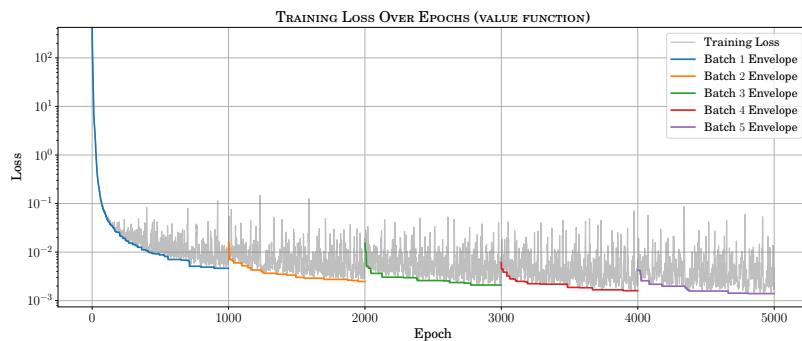


Figure 2.1: Training loss over epochs (value function)

training batches.

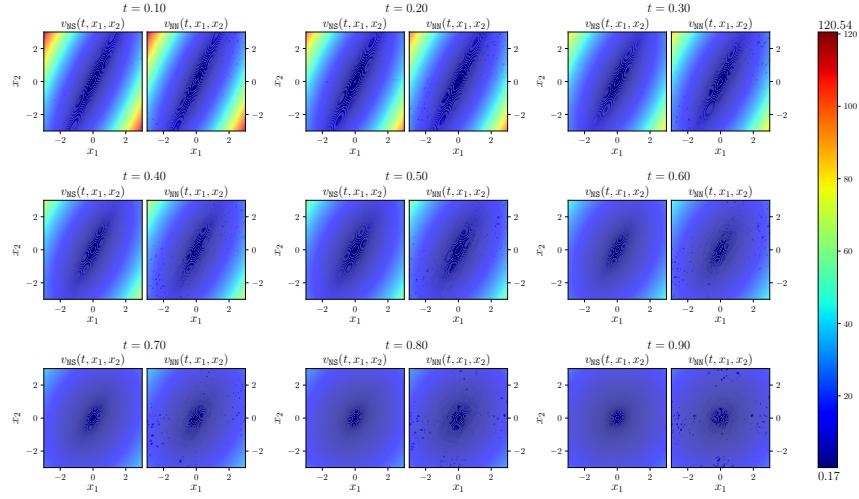


Figure 2.2: Visual comparison between v_{NS} and v_{NN} , $t \in [0.1, \dots, 0.9]$

We created a comparative visualization of the value function in fig. 2.2, illustrating the neural network approximation against numerical methods across time which is $t \in \{0.1, \dots, 0.9\}$ and a two-dimensional spatial scale. From the graph, it is evident that the two approaches yield closely aligned results.

Similarly to the previous model for the value function, after employing the same training setup, we obtain the training loss plot for the Markov control model:

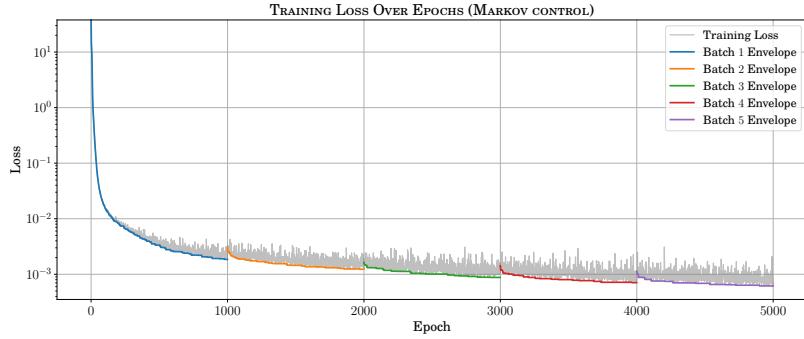


Figure 2.3: Training Loss over epochs (Markov control)

Observing fig. 2.3 in comparison to fig. 2.1, it is evident that the training loss for the Markov control model converges swiftly to a significantly lower value, resulting in a tidier curve that continues to decrease across subsequent batches. This suggests that, in theory, enhancing the neural network architecture with additional layers and extending the training duration could potentially lead the training loss to approach zero. Nonetheless, constrained by available computational resources such as CPUs or GPUs, this represents the extent of optimization achievable under current conditions.

3 Deep Galerkin approximation for a linear PDE

Exercise 3.1 Deep galerkin, Linear PDE.

Initialization

To solve the linear PDE

$$\begin{aligned} \partial_t u + \frac{1}{2}\text{tr}(\sigma\sigma^\top\partial_{xx}u) + (\partial_x u)^\top Hx + (\partial_x u)^\top M\alpha + x^\top Cx + \alpha^\top Dx = 0 & \quad \text{on } [0, T) \times \mathbb{R}^2, \\ u(T, x) = x^\top Rx & \quad \text{on } \mathbb{R}^2, \end{aligned}$$

where $\alpha = (1, 1)^\top$, we set the matrices to values as follows:

$$\begin{aligned} H &= \begin{bmatrix} 1.2 & 0.8 \\ -0.6 & 0.9 \end{bmatrix}, & M &= \begin{bmatrix} 0.5 & 0.7 \\ 0.3 & 1.0 \end{bmatrix}, & C &= \begin{bmatrix} 1.6 & 0 \\ 0 & 1.1 \end{bmatrix}, \\ D &= \begin{bmatrix} 0.5 & 0 \\ 0 & 0.7 \end{bmatrix}, & R &= \begin{bmatrix} 0.9 & 0 \\ 0 & 1 \end{bmatrix}, & \sigma &= \begin{bmatrix} 0.08 \\ 0.11 \end{bmatrix}. \end{aligned}$$

Together with α and time horizon $T = 1$, this section in our Python code is presented as follows, where the shapes of the vectors are adjusted due to the convenience of matrix multiplication.

```
1 Proj_dtype = torch.double
2 Proj_device = 'cpu'
3
4 H = torch.tensor([[1.2, 0.8], [-0.6, 0.9]], dtype = Proj_dtype, device = Proj_device)
5 M = torch.tensor([[0.5, 0.7], [0.3, 1.0]], dtype = Proj_dtype, device = Proj_device)
6 C = torch.tensor([[1.6, 0.0], [0.0, 1.1]], dtype = Proj_dtype, device = Proj_device)
7 D = torch.tensor([[0.5, 0.0], [0.0, 0.7]], dtype = Proj_dtype, device = Proj_device)
8 R = torch.tensor([[0.9, 0.0], [0.0, 1.0]], dtype = Proj_dtype, device = Proj_device)
9 T = torch.tensor(1.0, dtype = Proj_dtype, device = Proj_device)
10 sigma = torch.tensor([[0.08],[0.11]]], dtype = Proj_dtype, device = Proj_device)
11 alpha = torch.tensor([[1.0],[1.0]]], dtype = Proj_dtype, device = Proj_device)
```

Priliminary Work

We primarily opted for the simplest architecture, a Fully Connected Neural Network (FCNN), as the foundation for our model, the corresponding code is in the Jupyter Notebook file `Exercise3_1_primary.ipynb`.

Here is a code presentation of our primary NN structure.

```
1 class DGMNN(nn.Module):
2
3     def __init__(self):
4         super(DGMNN, self).__init__()
5         # Define three linear layers with 100 neurons each
```

```

6     self.layer1 = nn.Linear(3, 100) # Input layer (takes 3-dimensional input)
7     self.layer2 = nn.Linear(100, 100) # Hidden layer
8     self.layer3 = nn.Linear(100, 100) # Hidden layer
9     # Define activation functions
10    self.tanh = nn.Tanh() # Tangent hyperbolic activation function
11    self.relu = nn.ReLU() # Rectified Linear Unit activation function
12    # Define the output layer
13    self.output = nn.Linear(100, 1) # Output layer (produces 1-dimensional output)
14
15    def forward(self, x):
16        # Forward pass through the network
17        x = self.relu(self.layer1(x)) # Activation after first layer
18        x = self.tanh(self.layer2(x)) # Activation after second layer
19        x = self.relu(self.layer3(x)) # Activation after third layer
20
21        return self.output(x) # Return the output of the network

```

In this exercise, the aspect we found more challenging than the others was computing the Hessian matrix within the framework of PyTorch. This difficulty arises because directly applying the `torch.autograd.grad` method to $\partial_x u$ yields only the diagonal entries of the Hessian matrices. Such an outcome does not meet our requirements, as we need the full Hessian matrix for its multiplication. Therefore, we had to resort to calculating it through a loop-based approach. Here is the code for this crucial step.

```

1 def get_hessian(grad, x):
2
3     Hessian = torch.tensor([], device=Proj_device) # Initialize empty tensor for
4         # storing Hessian matrices
5
6     # Iterate over each element in x to compute second derivatives
7     for i in range(len(x)):
8         hessian = torch.tensor([], device=Proj_device) # Temporary tensor for storing
9             # Hessian of the current element
10            # Compute second derivative for each component of the gradient
11            for j in range(len(grad[i])):
12                # Compute the second derivative using autograd.grad
13                u_xxi = torch.autograd.grad(
14                    grad[i][j],
15                    x,
16                    grad_outputs=torch.ones_like(grad[i][j]),
17                    retain_graph=True,
18                    create_graph=True,
19                    allow_unused=True
20                )[0]
21
22                # Concatenate the second derivative to the temporary Hessian tensor
23                hessian = torch.cat((hessian, u_xxi[i].unsqueeze(0)))
24            # Concatenate the computed Hessian for the current element to the final
25            # Hessian tensor
26            Hessian = torch.cat((Hessian, hessian.unsqueeze(0)), dim=0)
27
28    return Hessian

```

After solving all the computation problems, we performed the model training with our

training strategy as follows:

- i. Set `iteration` = 5. In each `iteration` we set `epochs` = 40.
- ii. In each `epoch`, we choose a `sample_size` = N to sample the required space $(t, x) \in [0, T] \times [-3, 3]^2$. After we get one batch of this sample points $\{(t, x)^i, i = 1, \dots, N\}$, we can calculate the corresponding $\{(u(t, x))^i, i = 1, \dots, N\}$, and their gradients as well as Hessian matrices.
- iii. Following the expression of $R(\theta)$, we can construct the `loss` function for the training.
- iv. During the training process, we employ `torch.optim.Adam` as our optimizer with an initial `learning_rate` = 0.001, and we enhance it with an `ExponentialLR` learning rate scheduler, which is configured with a decay factor of $\gamma = 0.9$.

Priliminary Analysis and Conclusion

We studied the convergence of the model with different N . For $N \in \{100, 1000, 5000\}$, the training records are presented below. In each graph, we marked an approximate region (green) to which they ultimately converge.

From fig. 3.1 we can observe that with the utilization of this type of random sampling set for training, the model's performance does not achieve steady convergence when the sample size is relatively small, as exemplified by a size of 100. It becomes essential to increase the sample size to a sufficiently large scale to ensure stable convergence.

Nevertheless, the ultimate capacity of these models to align with the results obtained from the Monte Carlo method remains significantly inadequate. This observation is supported by the subsequent graph, where we compare the values generated by each method.

The comparison study is structured as follows:

- i. After the model has been trained, we choose a set of input $\{(t, x)^i, i = 1, \dots, N\}$ for both the trained model and the Monte Carlo simulation. In detail,

$$\begin{aligned} t &\in \{0.1, 0.2, \dots, 0.9\}, \\ x_1 &\in \{-3, 0, 3\}, \\ x_2 &\in \{-3, 0, 3\}, \end{aligned}$$

i.e.,

$$\begin{aligned} \{(t, x)^i, i = 1, \dots, N\} &= \{0.1, 0.2, \dots, 0.9\} \times \{-3, 0, 3\}^2, \\ N &= 9 \times 3 \times 3. \end{aligned}$$

- ii. For the trained neural network model, we simply turn it to the evaluation mode and then use this set as input to generate the corresponding $\{(u(t, x))^i, i = 1, \dots, N\}_{\text{NN}}$.
- iii. For the Monte Carlo simulation, we use the program build in **Exercise 1.2**, then set `time_step_number` = 5000, `sample_size` = 10000, and instead of using the S from the theoretical numerical solution to compute control vector, we use $\alpha = (1, 1)^\top$ this time. Denote the solution as $\{(u(t, x))^i, i = 1, \dots, N\}_{\text{MC}}$.

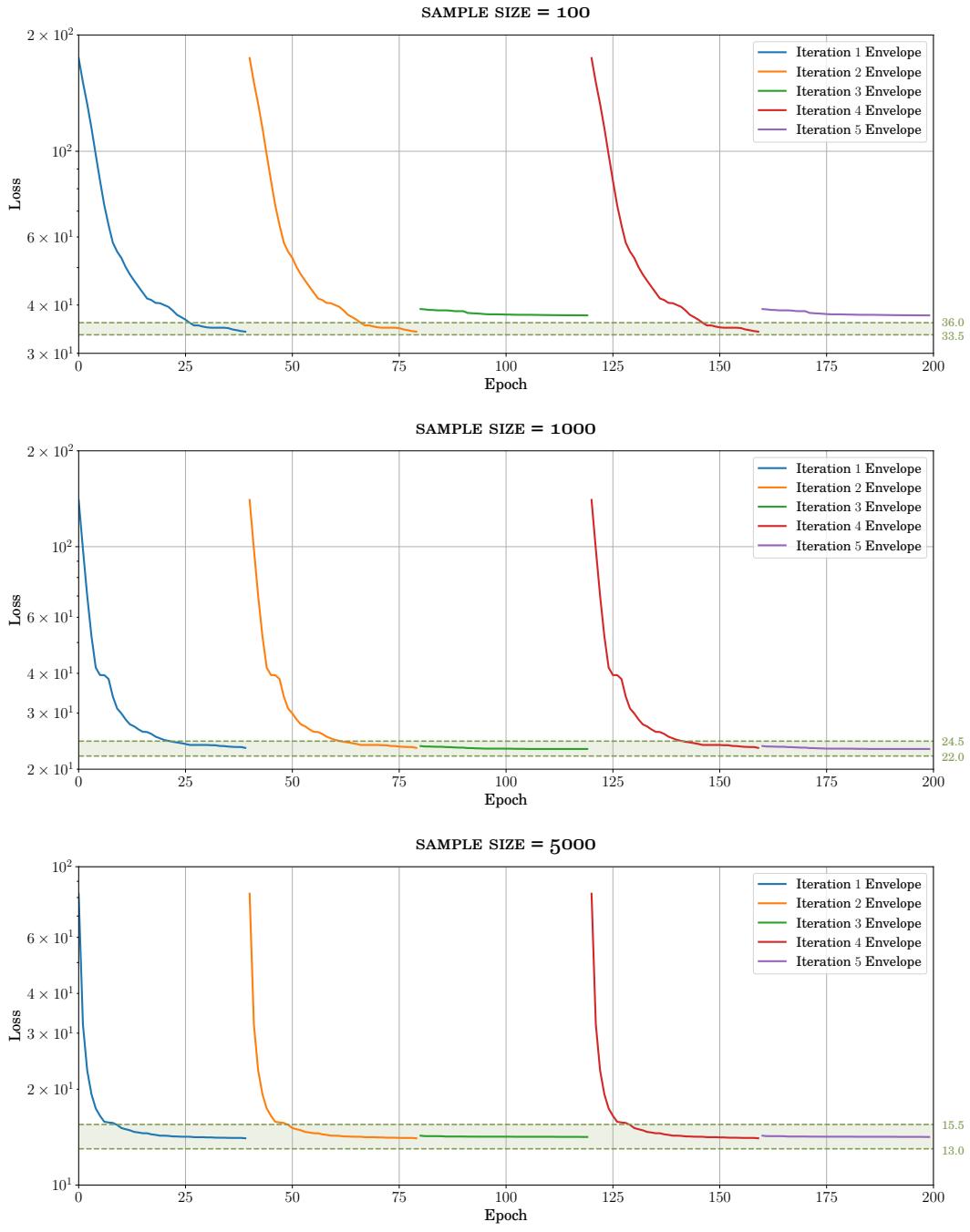


Figure 3.1: Training loss over epochs

- iv. After obtaining both sets of results, we plotted them for comparison. Due to the excessively large error, we opted not to plot the error directly. Instead, we represented both $\{(u(t, x))^i, i = 1, \dots, N\}_{\text{NN}}$ and $\{(u(t, x))^i, i = 1, \dots, N\}_{\text{MC}}$ as 1-Dimensional functions of the index i . This approach guarantees a one-to-one correspondence between the two sets of results, facilitating a clearer comparison.

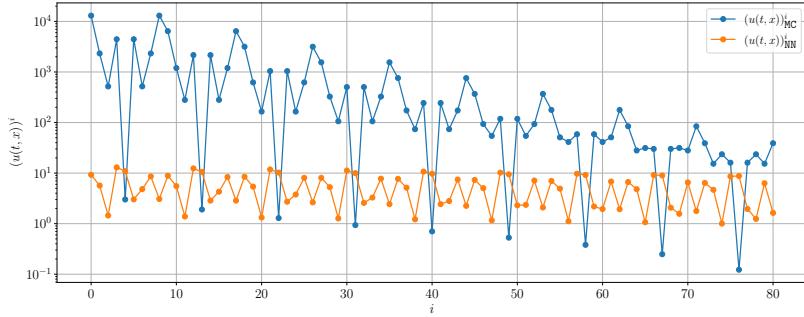


Figure 3.2: Comparison between $(u(t, x))_{\text{NN}}$ and $(u(t, x))_{\text{MC}}$

Figure 3.2 is the plot of the comparison between the two results, from which we can conclude that

1. The plots revealed a clear periodic pattern, attributed to the indices being organized sequentially according to time and spatial positioning.
2. From a numerical perspective, the fitting accuracy of the Neural Network (NN) model is profoundly unsatisfactory.

Further Research

To enhance the performance of the model employed in the Deep Galerkin Method (DGM), we delved into the relevant literature and identified a particularly pertinent study by Justin Sirignano and Konstantinos Spiliopoulos [1]. In their paper, the authors propose that the model's structure plays a pivotal role in accurately capturing the "sharp turns" exhibited by the function at boundaries (terminal conditions). This aspect is vital not only for fidelity in representation but also for training efficiency.

Consequently, we adopted their research approach by implementing the network structure they proposed in [1, Equation 4.2]. The relevant code implementation is detailed below.

```

1 class DGMhiddenlayerYYBver(nn.Module):
2
3     # From the original paper of Justin's, presented by Yuebo Yang Apr.9th 2024
4
5     def __init__(self, input_f, output_f, activation = 'tanh'):
6
7         super(DGMhiddenlayerYYBver, self).__init__()
8
9         self.input_f = input_f
10        self.output_f = output_f

```

```

11     # Params
12
13     # Zl's
14
15     self.Uzl = nn.Parameter(torch.Tensor(output_f, input_f))
16     self.Wzl = nn.Parameter(torch.Tensor(output_f, output_f))
17     self.bzl = nn.Parameter(torch.Tensor(output_f))
18
19     # Gl's
20
21     self.Ugl = nn.Parameter(torch.Tensor(output_f, input_f))
22     self.Wgl = nn.Parameter(torch.Tensor(output_f, output_f))
23     self.bgl = nn.Parameter(torch.Tensor(output_f))
24
25     # Rl's
26
27     self.Url = nn.Parameter(torch.Tensor(output_f, input_f))
28     self.Wrl = nn.Parameter(torch.Tensor(output_f, output_f))
29     self.brl = nn.Parameter(torch.Tensor(output_f))
30
31     # Hl's
32
33     self.Uhl = nn.Parameter(torch.Tensor(output_f, input_f))
34     self.Whl = nn.Parameter(torch.Tensor(output_f, output_f))
35     self.bhl = nn.Parameter(torch.Tensor(output_f))
36
37
38     if activation == 'tanh':
39         self.activation = torch.tanh
40     else:
41         self.activation = None
42
43     self.init_method = 'normal' # or 'uniform'
44
45     self._initialize_params()
46
47 def _initialize_params(self):
48
49     if self.init_method == 'uniform':
50         for param in self.parameters():
51             if param.dim() > 1:
52                 init.xavier_uniform_(param)
53             else:
54                 init.constant_(param, 0)
55
56     if self.init_method == 'normal':
57         for param in self.parameters():
58             if param.dim() > 1:
59                 init.xavier_normal_(param)
60             else:
61                 init.constant_(param, 0)
62
63 def forward(self, x, S1, Sl):
64
65     Zl = self.activation(torch.mm(x, self.Uzl.t())+ torch.mm(Sl, self.Wzl.t()) + self.bzl)

```

```

67
68     Gl = self.activation(torch.mm(x, self.Ugl.t())+ torch.mm(S1, self.Wgl.t()) +
69         self.bgl)
70
71     Rl = self.activation(torch.mm(x, self.Url.t())+ torch.mm(Sl, self.Wrl.t()) +
72         self.brl)
73
74     Hl = self.activation(torch.mm(x, self.Uhl.t())+ torch.mm(torch.mul(Sl,Rl),
75         self.Whl.t()) + self.bhl)
76
77     Sl_1 = torch.mul((1-Gl),Hl) + torch.mul(Zl,Sl)
78
79     return Sl_1
80
81
82
83 class DGMNN_YYBver(nn.Module):
84
85     # From the original paper of Justin's, presented by Yuebo Yang Apr.9th 2024
86
87     def __init__(self, init_method = 'uniform'):
88         super(DGMNN_YYBver, self).__init__()
89
90         self.nodenum = 50
91
92         self.layer1 = DGMhiddenlayerYYBver(3, self.nodenum)
93         self.layer2 = DGMhiddenlayerYYBver(3, self.nodenum)
94         self.layer3 = DGMhiddenlayerYYBver(3, self.nodenum)
95
96         self.tanh = nn.Tanh()
97
98         # Params
99
100        # S1's
101
102        self.W1 = nn.Parameter(torch.Tensor(self.nodenum, 3))
103        self.b1 = nn.Parameter(torch.Tensor(self.nodenum))
104
105        # Output's
106
107        self.W = nn.Parameter(torch.Tensor(1, self.nodenum))
108        self.b = nn.Parameter(torch.Tensor(1))
109
110        self.activation = torch.tanh
111
112        self.init_method = 'normal' # or 'uniform'
113
114        self._initialize_params()
115
116    def _initialize_params(self):
117
118        if self.init_method == 'uniform':
119            for param in self.parameters():
120                if param.dim() > 1:
121                    init.xavier_uniform_(param)
122                else:
123                    init.constant_(param, 0)

```

```

121     if self.init_method == 'normal':
122         for param in self.parameters():
123             if param.dim() > 1:
124                 init.xavier_normal_(param)
125             else:
126                 init.constant_(param, 0)
127
128
129     def forward(self, x):
130
131         S_1 = self.activation(torch.mm(x, self.W1.t()) + self.b1)
132         # l=1
133         S_2 = self.layer1(x,S_1,S_1)
134         # l=2
135         S_3 = self.layer2(x,S_1,S_2)
136         # l=3
137         S_4 = self.layer3(x,S_1,S_3)
138
139         output = torch.mm(S_4, self.W.t()) + self.b
140
141         return output

```

However, the authors outlined their training strategy towards the latter part of [1, Section 4.2], revealing a computational demand of 100,000 iterations with 5,000 points per iteration, where each iteration processes batches of 1,000 across GPU nodes. This level of computational investment significantly surpasses what we can feasibly achieve in this coursework. Despite conducting several training runs using this architecture, the improvements were minimal, limited by the computational resources at our disposal.

4 Policy iteration with DGM

Exercise 4.1 PIA with DGM, Linear PDE.

Initialization

To use the Deep Galerkin Method and to solve the same linear PDE listed in **Exercise 3.1**, the class `DGMNN_YYBver` in **Further Research** is introduced.

$$\partial_t u + \frac{1}{2} \text{tr}(\sigma \sigma^\top \partial_{xx} u) + (\partial_x u)^\top H x + (\partial_x u)^\top M \alpha + x^\top C x + \alpha^\top D \alpha = 0 \quad \text{on } [0, T) \times \mathbb{R}^2,$$
$$u(T, x) = x^\top R x \quad \text{on } \mathbb{R}^2,$$

where the initial Markov control is $\alpha = (1, 1)^\top$.

Also, the H, M, C, D, R, T are initialized as in the previous sections.

Preparation

We use class `value_diff` to compute the later difference.

```
1  class value_diff:
2      def __init__(self, t_b, x_b):
3          self.items = []
4          self.t_b = t_b
5          self.x_b = x_b
6
7      def add_item(self, item):
8          self.items.insert(0, item)
9          if len(self.items) > 2:
10              self.items.pop()
11
12     def calculate_difference(self):
13         if len(self.items) == 2:
14             v_n = self.items[0](torch.cat((self.t_b.unsqueeze(1),
15                 self.x_b.squeeze(1)), dim=1)).squeeze()
16             v_n1 = self.items[1](torch.cat((self.t_b.unsqueeze(1),
17                 self.x_b.squeeze(1)), dim=1)).squeeze()
18             return (v_n1 - v_n).pow(2).mean()
19         else:
20             inf1 = float('inf')
21             return inf1
```

We include the method to train the model at first part, defining two main function we will use in our iteration called `value_update` and `control_update`. The following is the excerpt.

The total residual is defined above as the sum of equation residual and boundary residual.

```
1  def value_update(H, M, sigma, alpha_i, C, D, R, T, iter_p):
```

```

3   model_DGM = DGMNN_YYBver(1).double()
4   optimizer_DGM = torch.optim.Adam(model_DGM.parameters(), lr=0.0001)
5   scheduler_DGM = lr_scheduler.ExponentialLR(optimizer_DGM, gamma=0.9)
6
7   ...
8
9       for batch_idx, (_t_data,_x_data) in enumerate(dataloader):
10          optimizer_DGM.zero_grad()
11          t_data = _t_data.clone().requires_grad_(True)
12          x_data = _x_data.clone().requires_grad_(True)
13          if iter_p == 0:
14              alpha = alpha_i
15          else:
16              alpha = alpha_i(torch.cat((t_data.unsqueeze(1),
17                  x_data),dim=1)).squeeze()
17          loss = total_residual(model_DGM, t_data, x_data, H, M, sigma, alpha,
18                  C, D, R, T)
19          loss.backward()
20          optimizer_DGM.step()
21          total_loss += loss.item()
22
23          avg_loss = total_loss / len(dataloader)
24          epoch_losses.append(avg_loss)
25
26          ...
27
28      return model_DGM

```

We will return an updated model of value function called model_DGM.
The Hamiltonian is defined in the following box.

```

1 def Hamiltonian(model_control, model_value, t, x, H, M, C, D):
2
3     input = torch.cat((t.unsqueeze(1), x),dim=1)
4
5     u = model_value(input)
6     a = model_control(input)
7
8     u_x = torch.autograd.grad(u, x, grad_outputs=torch.ones_like(u),
9                               create_graph=True, retain_graph=True)[0]
10
11    H_i = u_x.unsqueeze(1) @ H @ x.unsqueeze(1).transpose(1,2) + u_x.unsqueeze(1) @ M
12        @ a.unsqueeze(1).transpose(1,2) + x.unsqueeze(1) @ C @
13        x.unsqueeze(1).transpose(1,2) + a.unsqueeze(1) @ D @
14        a.unsqueeze(1).transpose(1,2)
15
16    # The true hamiltonian
17    H = H_i.mean()
18    return H

```

```

1 def control_update(H, M, C, D, T, model_value, iter_p):
2
3     control_DGM = DGMNN_YYBver(2).double()
4     optimizer_control = torch.optim.Adam(control_DGM.parameters(), lr=0.0001)
5     scheduler_control = lr_scheduler.ExponentialLR(optimizer_control, gamma=0.9)
6
7     ...
8
9     for epoch in range(epochs):
10
11         control_DGM.train()
12         total_loss = 0
13
14         for batch_idx, (_t_data, _x_data) in enumerate(dataloader):
15             optimizer_control.zero_grad()
16             t_data = _t_data.clone().requires_grad_(True)
17             x_data = _x_data.clone().requires_grad_(True)
18             loss = Hamiltonian(control_DGM, model_value, t_data, x_data, H, M,
19                 C, D)
20             loss.backward()
21             optimizer_control.step()
22             total_loss += loss.item()
23
24         avg_loss = total_loss / len(dataloader)
25         epoch_losses.append(avg_loss)
26
27     ...
28     return control_DGM

```

It will return an updated neural network model of Markov control.

Policy Iteration

In the main function, we run policy iteration by calling the functions defined above to approximate and update the value function and Markov controls.

After the initialization section,i.e., making a guess of the control $a_0 = \alpha = (1, 1)^\top$, the `while` loop continues until the difference of v_{n+1} and v_n is large and we ran the iteration no more than 5 times. In this setting, we set the limit to 0.01.

```

1 iter_p = 0
2 iter_max = 5
3 alpha = torch.tensor([[1., 1.]], dtype = Proj_dtype, device = Proj_device)
4
5 t_b, x_b = new_data(T, 10000)
6
7 v_diff = float('inf')
8 v_stack = value_diff(t_b, x_b)
9
10 while (v_diff > 0.01) and (iter_p < iter_max) :
11     print(f'Current policy iteration is {iter_p}/{iter_max}, and current value
          difference is {v_diff}.')

```

```

12     model_value = value_update(H, M, sigma, alpha, C, D, R, T, iter_p)
13     alpha = control_update(H, M, C, D, T, model_value, iter_p)
14     v_stack.add_item(model_value)
15     v_diff = v_stack.calculate_difference()
16     iter_p += 1
17
18     print(f'Policy iteration finished. Current iteration is {iter_p}/{iter_max}, and
          current value difference is {v_diff}.')

```

Analysis and Conclusion

The ultimate capacity of these models to align with the results obtained from the Deep Galerkin method remains significantly inadequate. This observation is supported by the subsequent graph, where we compared the values of numerical solution in **Exercise 1.1** and values generated by models.

- i. After the model has been trained, we chose the same set of input $\{(t, x)^i, i = 1, \dots, N\}$ for both the iteration and the numerical solution as stated in **Exercise 3.1**.
- ii. For the trained *neural network model*, we used them to generate the corresponding $\{(v(t, x))^i, i = 1, \dots, N\}_{\text{NN}}$ and $\{(a(t, x))^i, i = 1, \dots, N\}_{\text{NN}}$.
- iii. For the *numerical* value and control, we used the solver built in **Exercise 1.1**, then set `time_step_number = 10000`, `sample_size = 10000`, and denote the solution as $\{(v(t, x))^i, i = 1, \dots, N\}_{\text{NS}}$ and $\{(a(t, x))^i, i = 1, \dots, N\}_{\text{NS}}$.
- iv. After obtaining both sets of results, we plotted value and print controls for comparison. Due to the excessively large error, we opted not to plot the error directly. Instead, we represented both $\{(v(t, x))^i, i = 1, \dots, N\}_{\text{NN}}$ and $\{(v(t, x))^i, i = 1, \dots, N\}_{\text{NS}}$ as 1-Dimensional functions of the index i . This approach guarantees a one-to-one correspondence between the two sets of results, facilitating a clearer comparison.

The comparison study is structured as follows:

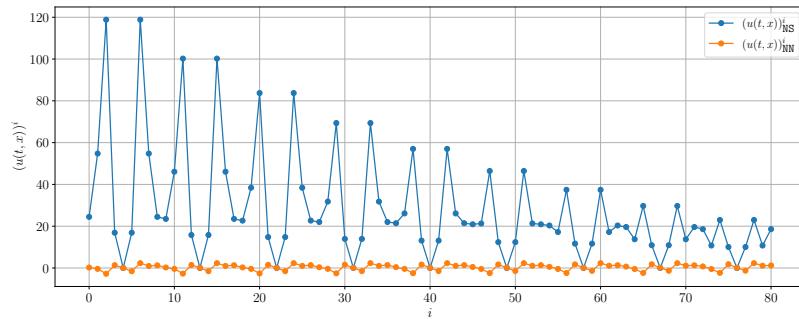


Figure 4.1: Comparison between $(v(t, x))_{\text{NN}}$ and $(v(t, x))_{\text{NS}}$

Figure 4.1 is the plot of the comparison between the two results, from which we can conclude that

Table 4.1: Comparison between $(a(t, x))_{\text{NS}}$ and $(a(t, x))_{\text{NN}}$

i	$(a(t, x))_{\text{NS}}^i$	$(a(t, x))_{\text{NN}}^i$
1	[6.4607, 9.9608]	[-5.5239e-01, 7.4345e-01]
2	[17.4128, 16.2514]	[1.2431e-01, 1.3142e-01]
3	[28.3648, 22.5420]	[-1.6899e+00, 2.4003e+00]
:	:	:
81	[-4.8322, -7.6592]	[1.7105e+00, 1.0442e+00]

1. From a numerical perspective, the convergence of ultimate value of the policy iteration with Deep Galerkin Method is profoundly unsatisfactory. So does the convergence of control.
2. This is mainly because of our inadequate training data point. This result could be improved by higher computing performance.

References

- [1] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.