
3 Appendix B: Code

```
1  # %% [markdown]
2  # # Q1.1.1
3
4  # %%
5  # Import the necessary libraries
6  import numpy as np
7
8  import matplotlib.pyplot as plt
9  import matplotlib.patches as patches
10 from matplotlib.ticker import FormatStrFormatter
11
12 import scipy.stats as stats
13 from scipy.io import loadmat
14 from scipy.optimize import minimize
15 from scipy.special import expit, logit
16
17 # %%
18 # set hyperparameters for common voxels and slices we will be using
19 im_slice = 71
20 vox_i = 91
21 vox_j = 64
22 dim_i = 145
23 dim_j = 174
24
25 # %%
26 # Load in the diffusion MRI data set and calculate settings for each component image
27 dwis = loadmat('data.mat')['dwis']
28 dwis = np.double(dwis)
29 dwis = dwis.transpose((3, 0, 1, 2))
30 [Dc, Dx, Dy, Dz] = dwis.shape
31
32 qhat = np.loadtxt("bvecs", delimiter = " ").T
33 bvals = 1000 * np.sum(qhat * qhat, axis=1)
34
35 # %%
36 # Solve for x in  $\log(A) = Gx$  - where x has all unknowns
37 x = np.zeros((dim_i, dim_j, 7))
38 quadratic_matrix = -bvals * np.array([qhat[:,0]**2, 2*qhat[:,0]*qhat[:,1], 2*qhat[:,0]*qhat[:,2], qhat[:,1]**2, 2*qhat[:,1]*qhat[:,2], qhat[:,2]**2])
39 G = np.concatenate([np.ones((108,1)), quadratic_matrix.T], axis=1)
40
41 # for each voxel calculate the solution
42 for i in range(dim_i):
43     for j in range(dim_j):
44         A = dwis[:,i,j,im_slice]
45         if np.min(A) > 0:
46             W = np.diag(A**2)
47             invmap = np.linalg.pinv(G.T @ W @ G) @ G.T @ W
48             x[i,j,:] = invmap @ np.log(A)
49
50 # calculate the DT for each voxel
51 D = np.zeros((dim_i,dim_j,3,3))
52 for i in range(dim_i):
53     for j in range(dim_j):
54         Dxx = x[i,j,1]
55         Dxy = x[i,j,2]
56         Dxz = x[i,j,3]
57         Dyy = x[i,j,4]
58         Dyz = x[i,j,5]
59         Dzz = x[i,j,6]
60         D[i,j] = np.array(
```

```

61         [[Dxx, Dxy, Dxz],
62          [Dxy, Dyy, Dyz],
63          [Dxz, Dyz, Dzz]]
64     )
65
66
67
68     # %%
69     # Plot the model estimate against the measure signal at voxel 92x65
70
71     A_est = np.exp(G @ np.squeeze(x[vox_i,vox_j,:]).reshape((-1,1)))
72     A_exact = np.squeeze(dwis[:,vox_i,vox_j,im_slice])
73     # Create grid of subplots
74     plt.scatter(np.arange(108), A_exact, marker='o', color='b', label='Observations')
75     plt.scatter(np.arange(108), A_est, marker='x', color='r', label='Model Estimate')
76     plt.legend()
77     plt.title(f'DT Model - at voxel {vox_i+1}x{vox_j+1}')
78     plt.show()
79
80     # %%
81     # Calculate mean diffusivity across the image
82     mean_D = np.zeros((dim_i,dim_j))
83
84     for i in range(dim_i):
85         for j in range(dim_j):
86             mean_D[i,j] = np.trace(D[i,j]) / 3
87
88     # Calculate FA
89     FA = np.zeros((dim_i,dim_j))
90     eig_val_D = np.zeros(((dim_i,dim_j,3)))
91     eig_vec_D = np.zeros((dim_i,dim_j,3,3))
92
93     for i in range(dim_i):
94         for j in range(dim_j):
95             eig_val, eig_vec = np.linalg.eig(np.squeeze(D[i,j]))
96             eig_vec_D[i,j, :, :] = eig_vec
97             eig_val_D[i,j, :] = eig_val
98             if eig_val.sum() > 0:
99                 FA[i,j] = np.sqrt(1.5 * np.sum((eig_val - eig_val.mean())**2) / (eig_val**2).sum()))
100
101     # %%
102     # plot Mean Diffusivity
103     fig, axs = plt.subplots(1, figsize=(5, 5))
104
105     axs.imshow(np.flipud(mean_D.T), cmap='gray')
106     axs.set_title('DT Model - Mean Diffusivity')
107
108     # Show the plot
109     plt.show()
110
111
112     # %%
113     # Plot the FA weighted with eigenvalues on RGB spectrum
114     FA_RGB = np.zeros((dim_i,dim_j,3))
115
116     for i in range(dim_i):
117         for j in range(dim_j):
118             principal_eig_idx = np.argmax(np.abs(eig_val_D[i,j]))
119             if eig_val_D[i,j,principal_eig_idx] > 0:
120                 eig_vec = eig_vec_D[i,j, :, principal_eig_idx]
121                 FA_RGB[i,j, :] = FA[i,j] * np.abs(np.array([eig_vec[0], eig_vec[1], eig_vec[2]]))
122
123     # normalise RGB values

```

```

124 FA_RGB = FA_RGB / np.max(FA_RGB)
125
126 # print to screen FA_RGB
127 fig, axs = plt.subplots(1, 2)
128 fig.suptitle('DT Model\nFractional Anisotropy')
129
130 axs[0].imshow(np.flipud(FA.T), cmap='gray')
131
132 axs[1].imshow(np.flipud(np.transpose(FA_RGB, (1,0,2))))
133
134 plt.tight_layout()
135 np.transpose
136
137
138 # %% [markdown]
139 # # Q1.1.2
140 # ### Ball and Stick Model
141
142 # %%
143 def ball_stick(x):
144
145     # Extract the parameters
146     # diff: diffusion
147     # f: fraction of signal contributed by diffusion tensor along fiber direction theta, phi
148     S0, diff, f, theta, phi = x
149
150     # Fiber direction
151     fibdir = np.array([
152         np.cos(phi) * np.sin(theta),
153         np.sin(phi) * np.sin(theta),
154         np.cos(theta),
155     ])
156
157     # creates a 2D array of fibdir stacked ontop of each other len(bvals) times
158     # so now has the dimensions [len(bvals)x3]
159     tile = np.tile(fibdir, (len(bvals), 1))
160     fibdotgrad = np.sum(qhat * tile, axis=1)
161
162     S = S0 * (f * np.exp(-bvals * diff * (fibdotgrad**2)) + (1-f) * np.exp(-bvals * diff))
163     return S
164
165
166 def BallStickSSD(x, voxel):
167     S = ball_stick(x)
168     # Compute sum of square differences
169     return np.sum((voxel - S) ** 2)
170
171 # %%
172 # Use minimize for non-linear estimation of the ball-and-stick parameters
173 # The first starting point finds a spurious local min; the second
174 # a more reasonable min.
175 avox = dwis[:,vox_i,vox_j,im_slice]
176 #startx = np.array([3500, -5e-6, 120, 0, 0])
177 #startx = np.array([4200, 4e-4, 0.25, 0, 0])
178 # start given by moodle note:
179 startx = np.array([3300, 1.0e-03, 4.5e-01, 1.0, 1.0])
180 results = minimize(
181     fun=BallStickSSD,
182     x0=startx,
183     args=(avox,),
184 )
185
186 results

```

```

187
188 # %%
189 # Use the fitted parameters to get estimated values
190 A_est = ball_stick(results['x'])
191 A_exact = np.squeeze(dwis[:,vox_i,vox_j,im_slice])
192
193 # Find the mean and std of the errors
194 error_dist = stats.describe(A_est - A_exact)
195 error_mean = error_dist[2]
196 error_var = error_dist[3]
197 print(f"error mean: {error_mean:.1f}")
198 print(f"error std: {np.sqrt(error_var):.1f}")
199
200 # Use the given noise std to calculate the expected SSD
201 ss_expected = 108 * 200**2
202 ss_exact = results['fun']
203 print(f"expected SS: {ss_expected:.1f}, calculated SS: {ss_exact:.1f}, diff: {(ss_expected - ss_exact):.1f}")
204
205 # Create grid of subplots to compare
206 plt.scatter(np.arange(108), A_exact, marker='o', color='b', label='Observations')
207 plt.scatter(np.arange(108), A_est, marker='x', color='r', label='Model Estimate')
208 plt.legend()
209 plt.title(f'Ball and Stick at {vox_i+1}x{vox_j+1}\noptimize without constraints')
210 plt.show()
211
212 # %% [markdown]
213 # # Q1.1.3
214
215 # %%
216 # We are now constraining the parameters ( $S_0$ ,  $\text{diff} > 0$ ,  $f$  in  $(0,1)$ ,  $\theta$ ,  $\phi$  in  $(0,2\pi)$ )
217 # We do this by transforming  $x$  to be squared, or  $\text{expit}()$  and scaled to ensure they are in the
218 # correct domain. Because the transformation happens after the optimizer guesses  $x_{t+1}$  we
219 # have to transform the optimized solution to get the correct fitted parameters
220
221 # Given  $S_0 \cdot 0.5$ ,  $\text{diff} \cdot 0.5$ ,  $\text{logit}(f)$ ,  $\text{logit}(\theta/\pi)$  and  $\text{logit}(\phi/2\pi)$  - we transform back to domain we want it
222 def transform(x):
223     return [x[0]**2, x[1]**2, expit(x[2]), expit(x[3])*np.pi, expit(x[4])*2*np.pi]
224
225 # Given  $S_0$ ,  $\text{diff}$ ,  $f$ ,  $\theta$ , and  $\phi$ , we inverse transform it to the unconstrained domain
226 def transform_inv(x):
227     return [x[0]**0.5, x[1]**0.5, logit(x[2]), logit(x[3]/(np.pi)), logit(x[4]/(2*np.pi))]
228
229 def BallStickSSD_constrained(x, voxel):
230     # given  $x$  that is unconstrained, we transform it to our wanted domain
231     S = ball_stick(transform(x))
232     # Compute sum of square differences
233     return np.sum((voxel - S) ** 2)
234
235
236 # %%
237 # Use the transform to find the parameters constrained
238 # note:  $\text{startx}$  is in our constrained domain, so we have to inverse transform it to be in
239 # the unconstrained domain
240 results = minimize(
241     fun=BallStickSSD_constrained,
242     x0=transform_inv(startx),
243     args=(avox,),
244 )
245
246 results
247
248 # With constraints the fitting works and we get sensible results ( $S_0$ ,  $\text{diff} > 0$ , etc.)
249 # and plotting against observed values we get much better results

```

```

250
251 # %%
252 # Use the fitted parameters to get estimated values
253 print(f"found fitted x = {transform(results['x'])}")
254 #A_est = ball_stick(transform(results['x']))
255 #A_exact = np.squeeze(dwis[:,91,64,71])
256 A_est = ball_stick(transform(results['x']))
257 A_exact = np.squeeze(dwis[:,vox_i,vox_j,im_slice])
258
259 # Find the mean and std of the errors
260 error_dist = stats.describe(A_est - A_exact)
261 error_mean = error_dist[2]
262 error_var = error_dist[3]
263 print(f"error mean: {error_mean}")
264 print(f"error std: {np.sqrt(error_var)}")
265
266 # Use the standard deviation to calculate the estimated Sum of Squares diff
267 ss_expected = 108 * 200*200
268 ss_exact = results['fun']
269 print(f"estimated SS: {ss_expected}, calculated SS: {ss_exact}, diff: {ss_expected - ss_exact}")
270
271 # Create grid of subplots to compare
272 plt.scatter(np.arange(108), A_exact, marker='o', color='b', label='Observations')
273 plt.scatter(np.arange(108), A_est, marker='x', color='r', label='Model Estimate')
274 plt.legend()
275 plt.title(f'Observations vs Ball and Stick Model at voxel {vox_i+1}x{vox_j+1}\noptimized with constraints')
276 plt.show()
277
278 # %% [markdown]
279 # Sum of squares has significantly reduced because now the model is fitting the observed data much better
280
281 # %% [markdown]
282 # # Q1.1.4
283
284 # %%
285 # for the same voxel run multiple times to try and find the best minimum
286 def BallStickSSD_constrained_findSSDmin(max_iter, startx, avox):
287     # given parameters of a single avox, run max_iter times and find converged SSD each time
288     # return all found solutions and SSD values
289
290     noise_std = startx / 5
291     num_parameters = startx.size
292     X_single_voxel = np.zeros((max_iter, num_parameters))
293     X_SSD = np.zeros(max_iter)
294
295     for i in range(max_iter):
296         # find some noise, add to the start, and transform and inverse it to make sure the
297         # perturbed start is a realistic start
298         noise = np.random.normal(loc=np.zeros(num_parameters), scale=noise_std)
299         x_i = startx + noise
300         x_i = transform_inv(transform(x_i))
301         results = minimize(
302             fun=BallStickSSD_constrained,
303             x0=transform_inv(x_i),
304             args=(avox,),
305         )
306         X_single_voxel[i,:] = results['x']
307         SSD_result = results['fun']
308         if np.isnan(SSD_result):
309             SSD_result = np.inf
310         X_SSD[i] = SSD_result
311
312     return X_single_voxel, X_SSD

```

```

313
314
315 def find_prob_finding_SSD_globalmin(startx, avox, max_iter=100, eps=1e-1):
316     # given a voxel, and a starting position. Optimize to solve for x with 95% confidence the global minima has been found
317
318     X_single_voxel, X_SSD = BallStickSSD_constrained_findSSDmin(max_iter, startx=startx, avox=avox)
319
320     min_SSD = np.min(X_SSD)
321     min_SSD_count = np.isclose(X_SSD, min_SSD, eps).sum()
322     p = min_SSD_count / X_SSD.shape[0]
323     print(f"min_SSD: {min_SSD}, found min {min_SSD_count} times, prob_global_min = {p}")
324     return p
325
326 # given prob of finding global min, return how many times we have to run optimisation
327 # to have 95% chance of finding it
328 def find_N_for_95percent_global_min(p):
329     return int(np.ceil(np.log(0.05) / np.log(1-p)))
330
331 # %%
332 # Check a number of different voxels for the number of times we have to run optimisation
333 # to have a 95% prob of finding the global min
334 voxel_idxxs = np.array([[vox_i, vox_j], [60,60], [55,55], [80,60], [85,55], [85, 72], [70, 70]])
335 N_array = np.zeros(shape=voxel_idxxs.shape[0])
336
337 for i, voxel_idx in enumerate(voxel_idxxs):
338     p = find_prob_finding_SSD_globalmin(startx, dwis[:, voxel_idx[0], voxel_idx[1], im_slice], max_iter=100)
339     N_array[i] = find_N_for_95percent_global_min(p)
340
341 print(f"Found Ns: {N_array}")
342 N_global_min = int(N_array.max())
343 print(f"Will use the max N found in our small sample: max_N = {N_global_min}")
344
345 # %% [markdown]
346 # # Q1.1.5
347
348 # %%
349 # use the found max N found across checked voxels to find the global min at each voxel in image slice
350
351 # Note: this function takes a while so the solution has been run once and saved down
352 # change is_skip to FALSE to re-calculate from scratch
353 is_skip = True
354 file_path = "X_optimize_each_voxel_with_same_N.npy"
355
356 if not is_skip:
357     X = np.zeros((dim_i, dim_j, startx.size))
358     X = np.load(file_path)
359
360     for i in range(dim_i):
361         for j in range(dim_j):
362             A = dwis[:, i, j, im_slice]
363             if np.min(A) > 0:
364                 x_single_voxel, x_SSD = BallStickSSD_constrained_findSSDmin(N_global_min, startx, dwis[:, i, j, im_slice])
365                 min_idx = np.argmin(x_SSD)
366                 X[i, j, :] = transform(x_single_voxel[min_idx, :])
367             # save each row to a file
368             np.save(file_path, X)
369             print(f"row {i} complete")
370 else:
371     X = np.load(file_path)
372
373
374
375 # %%

```

```

376 # Calculate RESNORM across the image slice
377
378 RESNORM = np.zeros(shape=(dim_i, dim_j))
379
380 for i in range(dim_i):
381     for j in range(dim_j):
382         RESNORM[i,j] = BallStickSSD(X[i,j,:], voxel=dwis[:,i,j, im_slice])
383
384 # %%
385 (RESNORM > 4e7).sum()
386
387 # %%
388 # plot the S0, d, f, and the RESNORM, and fibre direction of n
389
390 S0 = X[:, :, 0]
391 d_raw = X[:, :, 1]
392 d_processed = np.where(d_raw > 3, 0, d_raw)
393 f = X[:, :, 2]
394 theta = X[:, :, 3]
395 phi = X[:, :, 4]
396
397 n_zplane_x = np.sin(theta) * np.cos(phi) * f
398 n_zplane_y = - np.sin(theta) * np.sin(phi) * f
399
400 n_yplane_x = np.sin(theta) * np.sin(phi) * f
401 n_yplane_y = np.cos(theta) * f
402
403
404 # plot FA
405 # Create a 2x2 grid of subplots
406 fig, axs = plt.subplots(2,2, figsize=(10, 10))
407 fig.suptitle('Mapped parameters\nUsing Transform Optimised method with Ball & Stick')
408
409 axs[0,0].imshow(np.flipud(S0.T), cmap='gray')
410 axs[0,0].set_title('S0')
411
412 axs[0,1].imshow(np.flipud(f.T), cmap='gray')
413 axs[0,1].set_title('f')
414
415 axs[1,0].imshow(np.flipud(d_raw.T), cmap='gray', vmax=0.004)
416 axs[1,0].set_title('diffusivity\nmapped between (0, 0.004)')
417
418 axs[1,1].imshow(np.flipud(RESNORM.T), cmap='gray', vmax=1.5e7)
419 axs[1,1].set_title('RESNORM\nmapped between (0, 1.5e7)')
420
421 # Show the plot
422 plt.tight_layout()
423 plt.show()
424
425 # %%
426
427 fig, axs = plt.subplots(1, figsize=(8, 8))
428
429 axs.quiver(n_zplane_x.T, n_zplane_y.T)
430 axs.set_title('n\nprojected onto the x-z plane')
431 axs.set_aspect('equal')
432
433 plt.show()
434
435 # %% [markdown]
436 # There are many outliers in the found parameters so there are some found parameters that don't have found solutions. This can
437
438 # %% [markdown]

```

```

439 # # Q1.2.1
440
441 # %%
442
443
444 # %%
445 # Classical Bootstrapping method
446 # Sample with replacement T times to get A_t sampled data set. Each data set solve for parameters
447 # Plot the found parameters on a histogram and keep the middle 95%. Calculate sigma for the estimate
448 def calssical_bootstrap_find_parameters(vox_i, vox_j, im_slice, T=300):
449     N_data = dwis.shape[0]
450     # create indexes for T different iteration, each iteration have N samples indexes
451     sampled_idx = np.random.randint(N_data, size=(T,N_data))
452     bootstrap_parameters = np.zeros(shape=(T, 5))
453
454     for t in range(T):
455         A_t = dwis[sampled_idx[t], vox_i, vox_j, im_slice]
456         if np.min(A_t > 0):
457             x_single_voxel, x_SSD = BallStickSSD_constrained_findSSDmin(N_global_min, startx, A_t)
458             min_idx = np.argmin(x_SSD)
459             bootstrap_parameters[t,:] = transform(x_single_voxel[min_idx,:])
460
461     return bootstrap_parameters
462
463
464 # %%
465 # method to plot the histogram of bootstrap parameters given data and axes
466 def plot_histogram_sigma_95percent(axes, data, title, shade_colour='grey', shade_alpha=0.4, sigma_line_colour='red', mean_colour='red'):
467     T = data.size
468     data_std = np.std(data)
469     data_sorted_idx = np.argsort(data)
470     data_95_idx = [data_sorted_idx[int(T * 0.025)], data_sorted_idx[int(T * 0.975)]]
471     data_95_range = [data[data_95_idx[0]], data[data_95_idx[1]]]
472     data_2sigma_range = [data.mean() - 2*data_std, data.mean() + 2*data_std]
473
474     # plot histograms of data with shaded 95% region, and line showing 2 sigma range
475     height_values, _, _ = axes.hist(data)
476     axes.axvspan(xmin=data_95_range[0], xmax=data_95_range[1], facecolor=shade_colour, alpha=shade_alpha, label='95% confidence')
477     axes.plot(data_2sigma_range, [height_values.max()/2,height_values.max()/2], marker='|', c=sigma_line_colour, label='2 sigma')
478     axes.scatter(data.mean(), height_values.max()/2, marker='x', c=mean_colour, label='parameter mean')
479     if is_legend():
480         axes.legend()
481     axes.set_title(title)
482
483     return data_95_range, data_2sigma_range
484
485 # %%
486 # use classical bootstrapping method to find a range of parameters and plot on a histogram
487
488 vox_is = np.array([vox_i, 80, 60, 85])
489 vox_js = np.array([vox_j, 60, 60, 72])
490
491 num_vox = vox_is.size
492
493 bootstrap_2sigma_range = np.zeros(shape=(num_vox,2))
494 bootstrap_95_range = np.zeros(shape=(num_vox,2))
495
496 fig, axes = plt.subplots(num_vox,3, figsize=(10, 10))
497 fig.suptitle(f'Classical Bootstrap')
498
499 print('Bootstrap parameter ranges\n')
500 for vox in range(num_vox):
501     if vox==0:

```

```

502         is_legend=True
503     else:
504         is_legend=False
505     bootstrap_parameters = calssical_bootstrap_find_parameters(vox_is[vox], vox_js[vox], im_slice=im_slice, T=200)
506     bootstrap_95_range[0,:], bootstrap_2sigma_range[0,:] = plot_histogram_sigma_95percent(axes[vox, 0], bootstrap_parameters[:,
507     bootstrap_95_range[1,:], bootstrap_2sigma_range[1,:] = plot_histogram_sigma_95percent(axes[vox, 1], bootstrap_parameters[:,
508     bootstrap_95_range[2,:], bootstrap_2sigma_range[2,:] = plot_histogram_sigma_95percent(axes[vox, 2], bootstrap_parameters[:,
509
510     # print the ranges for each parameter:
511     print(f'Voxel: {vox_is[vox]+1}x{vox_js[vox]+1}')
512     print(f"S0: mean = {bootstrap_parameters.mean(axis=0)[0]}, 95% confidence = {bootstrap_95_range[0]}, 2sigma range = {bootstrap_2sigma_range[0]}")
513     print(f"Diffusivity: mean = {bootstrap_parameters.mean(axis=0)[1]}, 95% confidence = {bootstrap_95_range[1]}, 2sigma range = {bootstrap_2sigma_range[1]}")
514     print(f"f: mean = {bootstrap_parameters.mean(axis=0)[2]}, 95% confidence = {bootstrap_95_range[2]}, 2sigma range = {bootstrap_2sigma_range[2]}")
515
516 plt.tight_layout()
517
518 # %% [markdown]
519 # The first two plots for S0 and d match fairly well between 2 sigma range and the diffusivity which gives evidence that the di
520
521 # %% [markdown]
522 # # Q1.2.2
523 # MCMC
524 #
525 # for t in range(T):
526 #     y is sampled from dist Q
527 #     calculate alpha(x_{t-1}, y)
528 #     if alpha > U(0,1):
529 #         x_t = y
530 #     else
531 #         x_t = x_{t-1}
532 #     remove burn in
533 #     take every stride-th element as sample
534 #     return sample_dist
535
536 # %%
537 # given a ndarray of points, and a ndarray of standard deviations for each dimension
538 # return a point sampled from a gaussian located at the point with std
539 def q_sample_from_dist(x, param_std):
540     # add noise, then transform and inverse then inverse to ensure we remain within our domain
541     return transform(transform_inv(x + np.random.randn(x.size) * param_std))
542
543 # given parameters x and y, calculate the probability of p(A/y) / p(A/x)
544 # ie. how likely are we to sample y relative to x
545 # note: this is assuming the q distribution is symmetrical
546 def alpha_prob_ratio(y, x, data, noise_std):
547     x_SSD = BallStickSSD(x, data)
548     y_SSD = BallStickSSD(y, data)
549     return np.exp((1 / (2 * noise_std**2)) * (x_SSD - y_SSD)) * (np.sin(y[3]) / np.sin(x[3]))
550
551 # given burn_in, number of samples to throw away after burn (stride), and other parameters
552 # return the samples sequence from the distribution p(x/A)
553 def MCMC(data, x0=startx, burn_in=100, stride=10, sample_length=100, param_std=startx/5, noise_std=200):
554     param_num = 5
555     raw_sequence_length = burn_in + stride * sample_length
556     raw_sequence = np.zeros(shape=(raw_sequence_length, param_num))
557     accepted = np.zeros(raw_sequence_length)
558
559     # initialise parameters
560     raw_sequence[0,:] = x0
561
562     for t in range(1, raw_sequence_length):
563         x = raw_sequence[t-1,:]
564         y = q_sample_from_dist(x, param_std)

```

```

565     alpha = alpha_prob_ratio(y, x, data, noise_std=noise_std)
566     if alpha > np.random.rand():
567         raw_sequence[t] = y
568         accepted[t] = 1
569     else:
570         raw_sequence[t] = x
571
572     acceptance_after_burn = accepted[burn_in:].sum()/(raw_sequence_length-burn_in)
573     print(f"MCMC Complete: Total sequence length {raw_sequence_length}\nraw acceptance rate of {100*accepted.sum()/raw_sequence_length}%")
574     print(f"after burn in acceptance rate of {100*acceptance_after_burn:.0f}%")
575
576     after_burn_sequence = raw_sequence[burn_in:]
577     final_sequence_idx = np.arange(sample_length) * stride
578     return after_burn_sequence[final_sequence_idx], acceptance_after_burn
579
580
581 # %%
582 burn_in = 2000
583 stride = 5
584 sample_length = 2000
585
586 # 82% total
587 #param_std=np.array([1e1, 1e-6, 1e-3, 1e-2, 1e-2])
588 # 50% individual
589 #param_std=np.array([5e1, 4e-5, 2e-2, 6e-2, 8e-2])
590 # 70% individual
591 param_std=np.array([3e1, 1.5e-5, 1e-2, 2e-2, 3e-2])
592
593 data = dwis[:, vox_i, vox_j, im_slice]
594 MCMC_sequence, acceptance_rate = MCMC(data, startx, burn_in=burn_in, stride=stride, sample_length=sample_length, param_std=param_std)
595
596 # %%
597 fig, axs = plt.subplots(3, 2, figsize=(10, 10))
598 fig.suptitle(f'MCMC - at voxel:{vox_i+1}x{vox_j+1}\nburn in: {burn_in}, keep every {stride}th sample, kept sequence {sample_length}')
599
600 MCMC_S0 = MCMC_sequence[:,0]
601 MCMC_d = MCMC_sequence[:,1]
602 MCMC_f = MCMC_sequence[:,2]
603
604 burn_in_colour = 'grey'
605 shade_alpha = 0.4
606
607 axs[0,0].plot(MCMC_S0)
608 axs[0,0].set_title('MCMC S0')
609 axs[1,0].plot(MCMC_d)
610 axs[1,0].set_title('MCMC d')
611 axs[2,0].plot(MCMC_f)
612 axs[2,0].set_title('MCMC f')
613
614 MCMC_S0_95_range, MCMC_S0_2sigma_range = plot_histogram_sigma_95percent(axs[0, 1], MCMC_S0, 'S0', is_legend=True)
615 MCMC_d_95_range, MCMC_d_2sigma_range = plot_histogram_sigma_95percent(axs[1, 1], MCMC_d, 'Diffusivity')
616 MCMC_f_95_range, MCMC_f_2sigma_range = plot_histogram_sigma_95percent(axs[2, 1], MCMC_f, 'f')
617
618 axs[1,1].xaxis.set_major_formatter(FormatStrFormatter('%.2e'))
619 axs[1,1].tick_params(axis='x', labelsz=7)
620
621 print('MCMC parameter ranges\n')
622 print(f'Voxel: {vox_i+1}x{vox_j+1}')
623 print(f"S0: mean = {MCMC_S0.mean()}, 95% confidence = {MCMC_S0_95_range}, 2sigma range = {MCMC_S0_2sigma_range}")
624 print(f"Diffusivity: mean = {MCMC_d.mean()}, 95% confidence = {MCMC_d_95_range}, 2sigma range = {MCMC_d_2sigma_range}")
625 print(f"f: mean = {MCMC_f.mean()}, 95% confidence = {MCMC_f_95_range}, 2sigma range = {MCMC_f_2sigma_range}\n")
626
627 plt.tight_layout()

```

```

628
629 # %%
630 # Compare the Bootstrap output to MCMC output
631 MCMC_x = np.array([MCMC_S0.mean(),
632                   MCMC_d.mean(),
633                   MCMC_f.mean(),
634                   MCMC_sequence[:,3].mean(),
635                   MCMC_sequence[:,4].mean()])
636 bootstrap_x = np.array([bootstrap_parameters.mean(axis=0)[0],
637                        bootstrap_parameters.mean(axis=0)[1],
638                        bootstrap_parameters.mean(axis=0)[2],
639                        bootstrap_parameters.mean(axis=0)[3],
640                        bootstrap_parameters.mean(axis=0)[4]])
641
642 MCMC_est = ball_stick(MCMC_x)
643 bootstrap_est = ball_stick(bootstrap_x)
644 A_exact = np.squeeze(dwis[:,vox_i,vox_j,im_slice])
645
646 MCMC_SSD = BallStickSSD(MCMC_x, dwis[:,vox_i,vox_j,im_slice])
647 bootstrap_SSD = BallStickSSD(bootstrap_x, dwis[:,vox_i,vox_j,im_slice])
648
649 # Create grid of subplots to compare
650 plt.scatter(np.arange(108), A_exact, marker='o', color='b', label='Observations')
651 plt.scatter(np.arange(108), MCMC_est, marker='x', color='r', label='MCMC')
652 plt.scatter(np.arange(108), bootstrap_est, marker='x', color='g', label='Bootstrap')
653 plt.legend()
654 plt.title(f'MCMC vs Bootstrap fit at voxel {vox_i+1}x{vox_j+1}')
655 plt.show()
656
657
658
659 # %% [markdown]
660 # # Q1.3.1
661
662 # %%
663 # load in normalised data
664 # D.shape = 3612x6
665 # D has headers: vox1, vox2, vox3, vox4, vox5, vox6
666 D = np.genfromtxt('isbi2015_data_normalised.txt', skip_header=1, dtype=float, encoding='utf-8')
667
668 # load in protocol
669 # A.shape = 3612x7
670 # A has headers: dir-x, dir-y, dir-z, |G|, DELTA, delta, TE
671 A = np.genfromtxt('isbi2015_protocol.txt', skip_header=1, dtype=float, encoding='utf-8')
672
673 # dir-x, dir-y, dir-z
674 grad_dirs = A[:,0:3] # mT/mm
675 G = A[:,3] # mT/mm
676 delta = A[:,4] # s x10-6
677 smallldel = A[:,5] # ms (10x-3)
678 TE = A[:,6] # s x10-6
679
680 GAMMA = 2.675987e8
681
682 bvals = ((GAMMA * smallldel * G)**2) * (delta - (smallldel / 3))
683 # convert bvals to s/mm^2 from s/m^2
684 bvals = bvals / (10**6)
685 qhat = grad_dirs
686
687 num_vox = D.shape[1]
688 noise_std = 0.04
689
690 # %%

```

```

691 # given a model find the min SSD
692 def model SSD_constrained_findSSDmin(model_constrained SSD, transform, transform_inv, max_iter, startx, avox, method='BFGS'):
693     # given parameters of a single avox, run max_iter times and find converged SSD each time
694     # return best found solution and SSD value
695
696     noise_std = np.abs(startx / 5)
697     num_parameters = startx.size
698     X_single_voxel = np.zeros((max_iter, num_parameters))
699     X_SSD = np.zeros(max_iter)
700
701     for i in range(max_iter):
702         # find some noise, add to the start, and transform and inverse it to make sure the
703         # perturbed start is a realistic start
704         noise = np.random.normal(loc=np.zeros(num_parameters), scale=noise_std)
705         x_i = startx + noise
706         x_i = transform_inv(transform(x_i))
707         results = minimize(
708             fun=model_constrained SSD,
709             x0=transform_inv(x_i),
710             method=method,
711             args=(avox,),
712         )
713         X_single_voxel[i,:] = results['x']
714         SSD_result = results['fun']
715         if np.isnan(SSD_result):
716             SSD_result = np.inf
717         X_SSD[i] = SSD_result
718
719     min_idx = np.argmin(X_SSD)
720     x = X_single_voxel[min_idx]
721     min_SSD = X_SSD[min_idx]
722     min_SSD_count = np.isclose(X_SSD, min_SSD, 1e-1).sum()
723
724     return x, min_SSD, min_SSD_count
725
726 # %%
727 # given a model plot the results against observed signal
728 def model_plot_results(model, X, voxels, RESNORMs, model_name):
729
730     # plot the results
731     col_num = 2
732     row_num = 3
733     data_num = voxels.shape[0]
734
735     fig, axs = plt.subplots(col_num, row_num, figsize=(10, 10))
736     fig.suptitle(model_name)
737
738     vox = 0
739     for col in range(2):
740         for row in range(3):
741             A_est = model(X[vox])
742             A_exact = np.squeeze(voxels[:,vox])
743
744             # Create grid of subplots to compare
745             axs[col, row].scatter(np.arange(data_num), A_exact, marker='o', color='b', label='Observations')
746             axs[col, row].scatter(np.arange(data_num), A_est, marker='x', color='r', label='Model Estimate')
747             axs[col, row].set_title(f'Voxel {vox + 1}\nRESNORM: {RESNORMs[vox]:.1f}')
748             axs[col, row].legend()
749
750             vox = vox+1
751
752     plt.show()
753

```

```

754 # %% [markdown]
755 # Find solution for Ball & Stick model and plot for each voxel
756
757 # %%
758 D.shape
759
760 # %%
761 # solve for x in each voxel
762 num_param = 5
763 max_iter = 100
764 BS_startx = np.array([1, 5.0e-03, 8e-01, 1.0e+00, 1.0e+00])
765
766 BS_results = np.zeros(shape=(num_vox, num_param))
767 BS_SSD_results = np.zeros(shape=(num_vox))
768
769 for vox in range(num_vox):
770     x, min_SSD, min_SSD_count = model_SSD_constrained_findSSDmin(BallStickSSD_constrained, transform, transform_inv, max_iter,
771     BS_results[vox] = transform(x)
772     BS_SSD_results[vox] = min_SSD
773     print(f'N = {find_N_for_95percent_global_min(min_SSD_count / max_iter)}, p = {min_SSD_count / max_iter}, RESNORM = {min_SS
774
775 # %%
776 model_plot_results(ball_stick, BS_results, D, BS_SSD_results, 'Ball & Stick')
777
778 # %% [markdown]
779 # # Q1.3.2
780
781 # %%
782 # Define the different models
783 # NOTE: qhat and bvals have to be globally set before calling any of these models
784
785 # Diffusion Tensor
786 def DT_model(x):
787     S0, Dxx, Dxy, Dxz, Dyy, Dyz, Dzz = x
788
789     Diff = np.array(
790         [[Dxx, Dxy, Dxz],
791          [Dxy, Dyy, Dyz],
792          [Dxz, Dyz, Dzz]])
793
794     S = S0 * np.exp(-bvals * (qhat @ Diff @ qhat.T).diagonal())
795     return S
796
797
798 def zeppelin_stick_model(x):
799     # eig_val_1 >= eig_val_2 > 0
800     # eig_val_1 is assumed to be the same size as diffusivity
801     # eig_val_1 is in the same direction as the fibre direction (n_hat)
802     # eig_val_1 = eig_val_2 + eig_val_diff
803     # eigenvalues are setup this way so we can easily constrain the inputs
804     S0, eig_val_2, eig_val_diff, f, theta, phi = x
805
806     eig_val_1 = eig_val_2 + eig_val_diff
807
808     fibdir = np.array([
809         np.cos(phi) * np.sin(theta),
810         np.sin(phi) * np.sin(theta),
811         np.cos(theta),
812     ])
813
814     # creates a 2D array of fibdir stacked ontop of each other len(bvals) times
815     # so now has the dimensions [len(bvals)x3]
816     tile = np.tile(fibdir, (len(bvals), 1))

```

```

817     fibdotgrad = np.sum(qhat * tile, axis=1)
818
819     # intra-cellular signal
820     # note: largest eigen value is assumed to be the same as the diffusivity
821     # so we can substitute it in here
822     S_i = np.exp(-bvals * eig_val_1 * (fibdotgrad**2))
823
824     # extra-cellular signal
825     S_e = np.exp(-bvals * (eig_val_2 + (eig_val_1 - eig_val_2) * (fibdotgrad**2)))
826
827     # total signal S
828     S = S0 * (f * S_i + (1-f) * S_e)
829
830     return S
831
832
833 def zeppelin_stick_tur_model(x):
834     # same as Zeppelin_stick_model but with eig_val_2 = (1-f)*eig_val_1
835     S0, eig_val_2, f, theta, phi = x
836     eig_val_1 = eig_val_2 / (1-f)
837
838     y = [S0, eig_val_1, eig_val_2, f, theta, phi]
839
840     return zeppelin_stick_model(y)
841
842
843 def model_SSD(model, x, voxel):
844     # given a model calculate the modelled signals and return the sum of square difference
845     # compared to the observed signals
846     S = model(x)
847     return np.sum((voxel - S) ** 2)
848
849 # %%
850 # We are now constraining the parameters (S0, diff > 0, f in (0,1), theta, phi in (0,2pi))
851 # We do this by transforming x to be squared, or expit() and scaled to ensure they are in the
852 # correct domain. Because the transformation happens after the optimizer guesses x_t+1 we
853 # have to transform the optimized solution to get the correct fitted parameters
854
855
856 def DT_transform_inv(x):
857     # Given x we transform it to what we want to optimize to constrain it
858     # x = S0, Dxx, Dxy, Dxz, Dyy, Dyx, Dzz
859     Dxx = x[1]
860     Dxy = x[2]
861     Dxz = x[3]
862     Dyy = x[4]
863     Dyx = x[5]
864     Dzz = x[6]
865
866     # Use the cholesky decomposition to constrain D = L @ L.T
867     D = np.array(
868         [[Dxx, Dxy, Dxz],
869          [Dxy, Dyy, Dyx],
870          [Dxz, Dyx, Dzz]])
871
872     L = np.linalg.cholesky(D)
873     Lxx = L[0,0]
874     Lxy = L[0,1]
875     Lxz = L[0,2]
876     Lyy = L[1,1]
877     Lyx = L[1,2]
878     Lzz = L[2,2]
879

```

```

880     return [x[0]**0.5, Lxx, Lxy, Lxz, Lyy, Lyz, Lzz]
881
882
883 def DT_transform(x):
884     # Given transformed x return parameters we are looking for
885     # x = abs(S0)**0.5, Lxx, Lxy, Lxz, Lyy, Lyz, Lzz
886     # where L is the cholesky decomposition D = L @ L.T
887     Lxx = x[1]
888     Lxy = x[2]
889     Lxz = x[3]
890     Lyy = x[4]
891     Lyz = x[5]
892     Lzz = x[6]
893
894     L = np.array(
895         [[Lxx, 0, 0],
896          [Lxy, Lyy, 0],
897          [Lxz, Lyz, Lzz]])
898
899     D = L @ L.T
900     Dxx = D[0,0]
901     Dxy = D[0,1]
902     Dxz = D[0,2]
903     Dyy = D[1,1]
904     Dyz = D[1,2]
905     Dzz = D[2,2]
906
907     return [x[0]**2, Dxx, Dxy, Dxz, Dyy, Dyz, Dzz]
908
909
910 def DT_constrained_SSD(x, voxel):
911     S = DT_model(DT_transform(x))
912     # Compute sum of square differences
913     return np.sum((voxel - S) ** 2)
914
915
916 # %%
917 # Transformation functions for Zeppelin Stick model
918
919 def zeppelin_stick_transform_inv(x):
920     # Given x we transform it to what we want to optimize to constrain it
921     # x = S0, eig_val_2, eig_val_diff, f, theta, phi
922     return [x[0]**0.5, x[1]**0.5, x[2]**0.5, logit(x[3]), logit(x[4]/np.pi), logit(x[5]/(2*np.pi))]
923
924
925 def zeppelin_stick_transform(x):
926     # Given transformed x return parameters we are looking for
927     # x = abs(S0)**0.5, abs(eig_val_2)**0.5, abs(eig_val_diff)**0.5, expit(f), expit(theta), expit(phi)
928     return [x[0]**2, x[1]**2, x[2]**2, expit(x[3]), expit(x[4])*np.pi, expit(x[5])*2*np.pi]
929
930
931 def zeppelin_stick_constrained_SSD(x, voxel):
932     S = zeppelin_stick_model(zeppelin_stick_transform(x))
933     # Compute sum of square differences
934     return np.sum((voxel - S) ** 2)
935
936
937 # %%
938 # Transformation functions for Zeppelin Stick Turtuosity model
939
940 def zeppelin_stick_tur_transform_inv(x):
941     # Given x we transform it to what we want to optimize to constrain it
942     # x = S0, eig_val_2, f, theta, phi

```

```

943     return [x[0]**0.5, x[1]**0.5, logit(x[2]), logit(x[3]/np.pi), logit(x[4]/(2*np.pi))]
944
945
946 def zeppelin_stick_tur_transform(x):
947     # Given transformed x return parameters we are looking for
948     # x = abs(SD)**0.5, abs(eig_val_2)**0.5, expit(f), expit(theta), expit(phi)
949     return [x[0]**2, x[1]**2, expit(x[2]), expit(x[3])*np.pi, expit(x[4])*2*np.pi]
950
951
952 def zeppelin_stick_tur_constrained_SSD(x, voxel):
953     S = zeppelin_stick_tur_model(zeppelin_stick_tur_transform(x))
954     # Compute sum of square differences
955     return np.sum((voxel - S) ** 2)
956
957
958 # %%
959 # Use least squares to get an estimate on D before using the minimise function
960
961 data_range = [0,1000]
962 data_num = data_range[1] - data_range[0]
963
964 # Solve for x in log(A) = Gx - where x has all unknowns
965 x = np.zeros(7)
966 quadratic_matrix = -bvals[data_range[0]:data_range[1]] * np.array([qhat[data_range[0]:data_range[1],0]**2, 2*qhat[data_range[0]:data_range[1],1], qhat[data_range[0]:data_range[1],2]**2])
967 G = np.concatenate([np.ones((data_range[1] - data_range[0],1)), quadratic_matrix.T], axis=1)
968
969
970 A = D[data_range[0]:data_range[1],0]
971 W = np.diag(A**2)
972 invmap = np.linalg.pinv(G.T @ W @ G) @ G.T @ W
973 x = invmap @ np.log(A)
974
975 Dxx = x[1]
976 Dxy = x[2]
977 Dxz = x[3]
978 Dyy = x[4]
979 Dyz = x[5]
980 Dzz = x[6]
981 DT = np.array(
982     [[Dxx, Dxy, Dxz],
983      [Dxy, Dyy, Dyz],
984      [Dxz, Dyz, Dzz]]
985 )
986
987 y = x.copy()
988 y[0] = np.exp(y[0])
989 print(f'solution found from MSE: {y}')
990
991 # %%
992 # Use the transform to find the parameters constrained
993 DT_startx = y
994
995 # solve for x in each voxel
996 num_param = DT_startx.size
997 max_iter = 10
998
999 DT_results = np.zeros(shape=(num_vox, num_param))
1000 DT_SSD_results = np.zeros(shape=(num_vox))
1001
1002 for vox in range(num_vox):
1003     x, min_SSD, min_SSD_count = model_SSD_constrained_findSSDmin(DT_constrained_SSD, DT_transform, DT_transform_inv, max_iter,
1004     DT_results[vox] = DT_transform(x)
1005     DT_SSD_results[vox] = min_SSD

```

```

1006         print(find_N_for_95percent_global_min(min_SSD_count / max_iter))
1007
1008     # %%
1009     model_plot_results(DT_model, DT_results, D, DT_SSD_results, 'Diffusion Tensor')
1010
1011     # %%
1012     # Use the transform to find the parameters constrained
1013     zep_stick_startx = np.array([1,1e-3,1-3,0.5,1,1])
1014
1015     # solve for x in each voxel
1016     num_param = zep_stick_startx.size
1017     max_iter = 60
1018
1019     zep_stick_results = np.zeros(shape=(num_vox, num_param))
1020     zep_stick_SSD_results = np.zeros(shape=(num_vox))
1021
1022     for vox in range(num_vox):
1023         x, min_SSD, min_SSD_count = model_SSD_constrained_findSSDmin(zeppelin_stick_constrained_SSD, zeppelin_stick_transform, zep
1024         zep_stick_results[vox] = zeppelin_stick_transform(x)
1025         zep_stick_SSD_results[vox] = min_SSD
1026         print(find_N_for_95percent_global_min(min_SSD_count / max_iter))
1027
1028     # %%
1029     model_plot_results(zeppelin_stick_model, zep_stick_results, D, zep_stick_SSD_results, 'Zeppelin & Stick')
1030
1031     # %%
1032     zep_stick_results
1033
1034     # %%
1035     # Use the transform to find the parameters constrained
1036     zep_stick_tur_startx = np.array([1,5e-4,0.5,0.1,1])
1037
1038     # solve for x in each voxel
1039     num_param = zep_stick_tur_startx.size
1040     max_iter = 250
1041
1042     zep_stick_tur_results = np.zeros(shape=(num_vox, num_param))
1043     zep_stick_tur_SSD_results = np.zeros(shape=(num_vox))
1044
1045     for vox in range(num_vox):
1046
1047         x, min_SSD, min_SSD_count = model_SSD_constrained_findSSDmin(zeppelin_stick_tur_constrained_SSD, zeppelin_stick_tur_transf
1048         zep_stick_tur_results[vox] = zeppelin_stick_tur_transform(x)
1049         zep_stick_tur_SSD_results[vox] = min_SSD
1050         print(find_N_for_95percent_global_min(min_SSD_count / max_iter))
1051
1052     # %%
1053     model_plot_results(zeppelin_stick_tur_model, zep_stick_tur_results, D, zep_stick_tur_SSD_results, 'Zeppelin & Stick with Turtu
1054
1055     # %% [markdown]
1056     # # Q1.3.3
1057     #
1058
1059     # %%
1060     def log_like(A_est, A_exact, noise_std):
1061         var = noise_std**2
1062         diff_squared_sum = ((A_est - A_exact)**2).sum()
1063         log_like = ( -0.5 * np.log(2*np.pi*var) - (1/(2*var)) * diff_squared_sum )
1064         return log_like
1065
1066
1067     def AIC(A_est, A_exact, deg_freedom, noise_std):
1068         # we are calculating the noise directly so we don't have to increase N by 1

```

```

1069     N = deg_freedom
1070     K = A_est.shape[0]
1071
1072     if K/N < 40:
1073         print('WARNING: K/N > 40 -> should use adjusted AIC')
1074
1075     LogL = log_like(A_est, A_exact, noise_std)
1076     AIC = 2*N - 2*LogL
1077     return AIC
1078
1079
1080 def BIC(A_est, A_exact, deg_freedom, noise_std):
1081     # we are calculating the noise directly so we don't have to increase N by 1
1082     N = deg_freedom
1083     K = A_est.shape[0]
1084
1085     LogL = log_like(A_est, A_exact, noise_std)
1086     BIC = N*np.log(K) - 2*LogL
1087     return BIC
1088
1089 # %%
1090 # calculate the AIC and BIC for each model
1091 data_num = D.shape[0]
1092 num_vox = D.shape[1]
1093
1094 BS_preds = np.zeros(shape=D.shape)
1095 DT_preds = np.zeros_like(BS_preds)
1096 zeppelin_stick_preds = np.zeros_like(BS_preds)
1097 zeppelin_stick_tur_preds = np.zeros_like(BS_preds)
1098
1099 for vox in range(num_vox):
1100     start_idx = vox*data_num
1101     end_idx = start_idx + data_num
1102
1103     BS_preds[:, vox] = ball_stick(BS_results[vox])
1104     DT_preds[:, vox] = DT_model(DT_results[vox])
1105     zeppelin_stick_preds[:, vox] = zeppelin_stick_model(zep_stick_results[vox])
1106     zeppelin_stick_tur_preds[:, vox] = zeppelin_stick_tur_model(zep_stick_tur_results[vox])
1107
1108 AICs = np.zeros(shape=(num_vox, 4))
1109 BICs = np.zeros(shape=(num_vox, 4))
1110
1111 for vox in range(num_vox):
1112     AICs[vox, 0] = AIC(BS_preds[:,vox], D[:,vox], deg_freedom=5, noise_std=0.04)
1113     AICs[vox, 1] = AIC(DT_preds[:,vox], D[:,vox], deg_freedom=7, noise_std=0.04)
1114     AICs[vox, 2] = AIC(zeppelin_stick_preds[:,vox], D[:,vox], deg_freedom=6, noise_std=0.04)
1115     AICs[vox, 3] = AIC(zeppelin_stick_tur_preds[:,vox], D[:,vox], deg_freedom=5, noise_std=0.04)
1116
1117     BICs[vox, 0] = BIC(BS_preds[:,vox], D[:,vox], deg_freedom=5, noise_std=0.04)
1118     BICs[vox, 1] = BIC(DT_preds[:,vox], D[:,vox], deg_freedom=7, noise_std=0.04)
1119     BICs[vox, 2] = BIC(zeppelin_stick_preds[:,vox], D[:,vox], deg_freedom=6, noise_std=0.04)
1120     BICs[vox, 3] = BIC(zeppelin_stick_tur_preds[:,vox], D[:,vox], deg_freedom=5, noise_std=0.04)
1121
1122 # %%
1123 models = {0:'BS', 1:'DT', 2:'zep_stick', 3:'zep_stick_tur'}
1124
1125 print(f'Model Ranking - Best to worst')
1126 for vox in range(num_vox):
1127     print(f'voxel: {vox}')
1128
1129     AIC_rank = np.argsort(AICs[vox])
1130     BIC_rank = np.argsort(BICs[vox])
1131

```

```

1132     AIC_text = 'AIC: '
1133     BIC_text = 'BIC: '
1134     for rank in range(4):
1135         str = f'{models[AIC_rank[rank]]}, '
1136         AIC_text = AIC_text + str
1137         str = f'{models[BIC_rank[rank]]}, '
1138         BIC_text = BIC_text + str
1139     print(AIC_text)
1140     print(BIC_text + '\n')
1141
1142
1143
1144     # %% [markdown]
1145     # # q1.3.4
1146
1147     # %%
1148     def B2S_model(x):
1149         # Behrens et al, 2003
1150         # Characterization and Propagation of Uncertainty in Diffusion-Weighted MR Imaging
1151         # https://doi.org/10.1002/mrm.10609
1152
1153         # Extract the parameters
1154         # diff: diffusion
1155         # f: fraction of signal contributed by diffusion tensor along fiber direction theta, phi
1156         S0, diff, f2, f_diff_ratio, theta1, phi1, theta2, phi2 = x
1157         f1 = f2 + (1 - f2) * f_diff_ratio
1158
1159         # Fiber direction
1160         fibdir1 = np.array([
1161             np.cos(phi1) * np.sin(theta1),
1162             np.sin(phi1) * np.sin(theta1),
1163             np.cos(theta1),
1164         ])
1165         fibdir2 = np.array([
1166             np.cos(phi2) * np.sin(theta2),
1167             np.sin(phi2) * np.sin(theta2),
1168             np.cos(theta2),
1169         ])
1170
1171         # creates a 2D array of fibdir stacked ontop of each other len(bvals) times
1172         # so now has the dimensions [len(bvals)x3]
1173         tile = np.tile(fibdir1, (len(bvals), 1))
1174         fibdotgrad1 = np.sum(qhat * tile, axis=1)
1175         tile = np.tile(fibdir2, (len(bvals), 1))
1176         fibdotgrad2 = np.sum(qhat * tile, axis=1)
1177
1178         # calculate intra and extra contributions to the model
1179         Si1 = np.exp(-bvals * diff * (fibdotgrad1**1))
1180         Si2 = np.exp(-bvals * diff * (fibdotgrad2**2))
1181         Se = np.exp(-bvals * diff)
1182
1183         S = S0 * (f1 * Si1 + f2 * Si2 + (1-f1-f2) * Se)
1184         return S
1185
1186
1187     def B2S_SSD(x, voxel):
1188         S = B2S_model(x)
1189         # Compute sum of square differences
1190         return np.sum((voxel - S) ** 2)
1191
1192
1193     # Given S0, diff, f2, f_diff_ratio, theta1, phi1, theta2, phi2 - we transform to constrain it
1194     # f1 = f2 + (1-f2) * f_diff_ratio

```

```

1195 # f1 + f2 < 1
1196 # 1 > f1 >= f2 >=0
1197 # f_diff_ratio in (0,1)
1198 def B2S_transform_inv(x):
1199
1200     return [x[0]**0.5, x[1]**0.5,
1201             logit(x[2]), logit(x[3]),
1202             logit(x[4]/np.pi), logit(x[5]/(2*np.pi)),
1203             logit(x[6]/np.pi), logit(x[7]/(2*np.pi))]
1204
1205 # Given transformed x return parameters we are looking for
1206 def B2S_transform(x):
1207
1208     return [x[0]**2, x[1]**2,
1209            expit(x[2]), expit(x[3]),
1210            expit(x[4])*np.pi, expit(x[5])*2*np.pi,
1211            expit(x[6])*np.pi, expit(x[7])*2*np.pi]
1212
1213 def B2S_constrained_SSD(x, voxel):
1214     S = B2S_model(B2S_transform(x))
1215     # Compute sum of square differences
1216     return np.sum((voxel - S) ** 2)
1217
1218 # %%
1219 # Use the transform to find the parameters constrained
1220 B2S_startx = np.array([1,1e-5,0.3, 0.7,1,2, 1, 1])
1221
1222 # solve for x in each voxel
1223 num_param = B2S_startx.size
1224 max_iter = 50
1225
1226 B2S_results = np.zeros(shape=(num_vox, num_param))
1227 B2S_SSD_results = np.zeros(shape=(num_vox))
1228
1229 for vox in range(num_vox):
1230     x, min_SSD, min_SSD_count = model_SSD_constrained_findSSDmin(B2S_constrained_SSD, B2S_transform, B2S_transform_inv, max_iter, B2S_startx)
1231     B2S_results[vox] = B2S_transform(x)
1232     B2S_SSD_results[vox] = min_SSD
1233     print(find_N_for_95percent_global_min(min_SSD_count / max_iter))
1234
1235 # %%
1236 model_plot_results(B2S_model, B2S_results, D, B2S_SSD_results, 'Ball & 2 Stick Model')
1237
1238

```
