ΠΑΝΟΣ ΛΕΛΑΚΗΣ

1083712

4ο ΕΤΟΣ

Ισοστάθμιση ακουστικών: μία αυτοματοποιημένη προσέγγιση

Περίληψη

Το θέμα της εργασίας είναι η ανάπτυξη και εφαρμογή μίας μεθόδου για την αυτοματοποίηση της διαδικασίας της ισοστάθμισης ακουστικών, με στόχο τη βελτίωση της ακουστικής τους απόκρισης σύμφωνα με μία προκαθορισμένη καμπύλη στόχο. Η διαδικασία αυτή περιλαμβάνει την προσαρμογή της απόκρισης συχνότητας των ακουστικών μέσω φίλτρων, ώστε να επιτυγχάνεται μια πιο φυσική και ισορροπημένη ηχητική εμπειρία.

Στο πλαίσιο της εργασίας, μελετήθηκαν και υλοποιήθηκαν με κώδικα υπάρχουσες τεχνικές ισοστάθμισης ακουστικών με χρήση φίλτρων Finite Impulse Response και Second Order Sections. Για την αυτοματοποίηση των βημάτων της ισοστάθμισης, σχεδιάστηκε και υλοποιήθηκε κατάλληλος αλγόριθμος που αξιοποιεί τις υπάρχουσες μεθόδους. Η μεθοδολογία που ακολουθήθηκε περιλαμβάνει τη μέτρηση της αρχικής απόκρισης των ακουστικών, την σύγκρισή της με την επιθυμητή απόκριση, την εφαρμογή του αλγορίθμου ισοστάθμισης και την αξιολόγηση των αποτελεσμάτων μέσω συγκριτικής ανάλυσης με άλλες μεθόδους.

Τα αποτελέσματα τόσο για κάθε στάδιο της ανάλυσης όσο και για κάθε τεχνική, παρουσιάζονται μέσω γραφικών απεικονίσεων των καμπυλών απόκρισης πριν και μετά την ισοστάθμιση, ενώ αναλύονται και τα σφάλματα που προκύπτουν από τις διαφορετικές παραμέτρους που χρησιμοποιήθηκαν. Η αναφορά καταλήγει με συμπεράσματα σχετικά με την αποτελεσματικότητα της κάθε μεθόδου και του αλγορίθμου που αναπτύχθηκε και σημειώνει τυχόν περιπτώσεις της λογικής του που χρήζουν βελτίωσης.

Α. ΕΙΣΑΓΩΓΗ

Η ισοστάθμιση ακουστικών (headphone equalization) αποτελεί μία από τις κυριότερες ασχολίες των επαγγελματιών στον χώρο της επεξεργασίας ήχου. Πρόκειται για την διαδικασία με την οποία με εφαρμογή φίλτρων, η συμπεριφορά ενός ακουστικού προσαρμόζεται έτσι ώστε να προσεγγίζει ένα σύνολο επιθυμητών εντάσεων για συγκεκριμένες συχνότητες. Η συμπεριφορά ή αλλιώς απόκριση του ακουστικού, από τη μία, απεικονίζεται στον χώρο της συχνότητας με τον μετασχηματισμό Fourier ο οποίος δείχνει με τι ένταση, δηλαδή πόσο δυνατά, αποδίδεται η κάθε συχνότητα (headphone response). Οι επιθυμητές εντάσεις, από την άλλη, είναι η καμπύλη-στόχος (target response) που επιδιώκεται η απόκριση του ακουστικού να προσεγγίσει.

Οι λόγοι για τους οποίους μπορεί να είναι αναγκαία η ισοστάθμιση ενός ακουστικού ποικίλουν. Συνηθέστερες αιτίες συνιστούν οι κατασκευαστικές ατέλειες και σφάλματα, όπως επίσης και η χρήση φτηνών υλικών. Πέρα από αυτούς τους λόγους, όμως, μπορεί ο χρήστης λόγω προσωπικών προτιμήσεων να επιθυμεί να ρυθμίσει διαφορετικά το ακουστικό.

Για να επιτευχθεί η καμπύλη-στόχος, οι επαγγελματίες σήμερα εφαρμόζουν σε headphone responses φίλτρα κατάλληλου τύπου και χαρακτηριστικών, βασιζόμενοι κυρίως στην εμπειρία τους και στην ακουστική τους ικανότητα. Η παρούσα εργασία παρουσιάζει πώς η διαδικασία αυτή μπορεί να αυτοματοποιηθεί και να πραγματοποιείται πλέον εξ ολοκλήρου από έναν υπολογιστή.

Από την εκφώνηση της εργασίας δόθηκαν συγκεκριμένα tasks, τα οποία οδήγησαν βήμα-βήμα στην ανάπτυξη του ολοκληρωμένου

προγράμματος που εκτελεί equalization σε δεδομένα headphone responses, έτσι ώστε να προσεγγίζουν συγκεκριμένα target responses. Για τον σκοπό της εργασίας, αναπτύχθηκε κώδικας σε Python, ο οποίος πληροί τα ζητούμενα του κάθε task.

Η μελέτη βιβλιογραφικών πηγών κρίθηκε απαραίτητη τόσο για την θεωρητική κατανόηση του προβλήματος, όσο και για την υλοποίηση της λύσης του με κώδικα. Αρχικά, για την κατανόηση του θέματος χρησιμοποιήθηκαν πηγές από το διαδίκτυο [1, 2]. Επιπλέον θεωρητικά θέματα αναζητήθηκαν κατά τη διάρκεια συγγραφής του κώδικα. Για την ανάπτυξη του προγράμματος, μελετήθηκε το εργαλείο AutoEQ [3, 4], το οποίο μέχρι σήμερα αποτελεί την πιο επιτυχημένη δημοσιευμένη υλοποίηση του αυτόματου headphone equalization. Επίσης, χρησιμοποιήθηκαν εκτενώς οι συναρτήσεις για επεξεργασία σημάτων της βιβλιοθήκης SciPy.signal [5]. Ο αλγόριθμος που υλοποιήθηκε για την αυτοματοποίηση της διαδικασίας βασίστηκε στον αλγόριθμο του AutoEQ [6, 7]. Περαιτέρω πηγές που χρησιμοποιήθηκαν αποκλειστικά για την συγγραφή του κώδικα, για παράδειγμα για την επίλυση κάποιου error, αναφέρονται στα αντίστοιχα σημεία στον κώδικα.

II. ΑΝΑΠΤΥΞΗ ΘΕΜΑΤΟΣ

Στην παρούσα ενότητα αναλύεται η μεθοδολογία που ακολουθήθηκε για την επίλυση του προβλήματος. Ειδικότερα, αναφέρονται τα ζητούμενα των tasks της εκφώνησης, η χρήση του μετασχηματισμού Fourier, η εύρεση της επιθυμητής απόκρισης των φίλτρων και η μέτρηση του σφάλματος. Όσον αφορά τους τρόπους με τους οποίους πραγματοποιείται το equalization, εξηγείται η χρήση τόσο των Finite Impulse Response (FIR) όσο και των Second Order Sections (SOS) φίλτρων. Τέλος, παρουσιάζεται ο αλγόριθμος που αναπτύχθηκε, ο οποίος αξιοποιεί τις παραπάνω μεθόδους για την αυτοματοποίηση της διαδικασίας.

A. Tasks

Στην εκφώνηση της εργασίας δόθηκαν συγκεκριμένα tasks τα οποία οδήγησαν στην σταδιακή ανάπτυξη του ολοκληρωμένου προγράμματος που πραγματοποιεί equalization. Συγκεκριμένα, ζητήθηκε:

1. Να παρασταθούν γραφικά τα headphone και target responses, καθώς και να μετρηθεί το Mean Squared Error (MSE) και το Root Mean Squared Error (RMSE).
2. Να δημιουργηθεί ένα φίλτρο που να επιτυγχάνει για όλα τα headphone responses τα targets στο πεδίο της συχνότητας.
3. Να δημιουργηθεί ένα FIR φίλτρο που να επιτυγχάνει τα targets μέσω συνέλιξης (convolution) στο πεδίο του χρόνου.
4. Να συγκριθούν τα αποτελέσματα μετά από μεταβολή του πλήθους των συντελεστών (coefficients) του FIR φίλτρου.
5. Να κατασκευαστεί συνάρτηση που να εφαρμόζει παραμετρική ισοστάθμιση (parametric equalization) έχοντας ως δεδομένα τις κεντρικές συχνότητες (center frequencies Wo) στις οποίες πρόκειται να εφαρμοστεί, καθώς και τους συντελεστές ποιότητας (quality factors Q) που καθορίζουν την καμπυλότητα στις αντίστοιχες συχνότητες. Η συνάρτηση ζητήθηκε να υλοποιεί τα φίλτρα σε μορφή SOS.
6. Να χρησιμοποιηθεί η συνάρτηση parametric equalizer που δημιουργήθηκε, ώστε να επιτευχθούν τα targets μέσω συνέλιξης.
7. Να υλοποιηθεί αυτόματος parametric equalizer που να ανιχνεύει τις συχνότητες στις οποίες απαιτείται εφαρμογή φίλτρων, να αποφασίζει για τον τύπο τους, να υπολογίζει τα χαρακτηριστικά τους και να τα υλοποιεί αξιοποιώντας την συνάρτηση που κατασκευάστηκε. Το πρόγραμμα ζητείται να λαμβάνει ως είσοδο το headphone response και το target και να επιστρέφει τα φίλτρα που είναι αναγκαίο να εφαρμοστούν, χωρίς να απαιτείται αλληλεπίδραση με τον χρήστη σε οποιοδήποτε στάδιο της διαδικασίας.

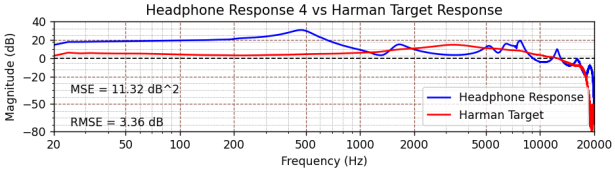
Για τον σκοπό της εργασίας, δόθηκαν από τους καθηγητές του μαθήματος μετρήσεις για 7 διαφορετικά ακουστικά, τα οποία ζητήθηκαν να

ισοσταθμιστούν. Επίσης, δόθηκαν το Flat και το Harman target ως καμπύλες-στόχοι, για τα οποία έγινε προσπάθεια να προσεγγιστούν.

B. Μετασχηματισμός Fourier και μέτρηση σφάλματος

Ο μετασχηματισμός Fourier (Fourier Transform - FT) είναι ένα από τα κυριότερα εργαλεία για το headphone equalization. Πιο συγκεκριμένα, μετασχηματίζοντας ένα σύνολο μετρήσεων για ένα ακουστικό στο πεδίο της συχνότητας, είναι δυνατό να κατανοηθεί η συμπεριφορά του, δηλαδή να παρατηρηθεί με πόση ένταση αυτό αποδίδει την κάθε συχνότητα.

Στο Σχ. 1 παρουσιάζονται ενδεικτικά το headphone response για το 4ο από τα δεδομένα ακουστικά και το Harman target response.



Σχ. 1: Η αναπαράσταση του headphone response για ένα από τα δεδομένα ακουστικά και του Harman target, στο πεδίο της συχνότητας.

Οι τιμές των MSE και RMSE αποτελούν δείκτες σφάλματος και εκφράζουν την διαφορά (σφάλμα) μεταξύ headphone και target response. Σύμφωνα με τις πηγές [8, 9], μεγαλύτερη τιμή στην ένδειξη του MSE υποδεικνύει μεγαλύτερο σφάλμα, δηλαδή μεγαλύτερη απόσταση μεταξύ των δύο καμπυλών.

Τα MSE και RMSE υπολογίζονται από τους τύπους

$$MSE = Μέσος\ όρος\ του\ [(ένταση\ headpone\ response) - (ένταση\ target\ response)]^2$$

$$RMSE = \sqrt{MSE}$$

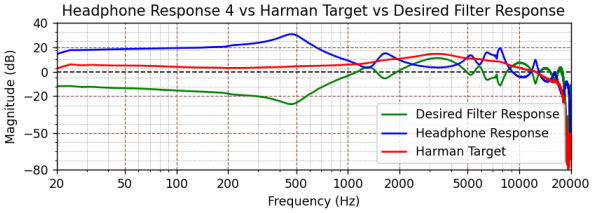
Από τις μετρήσεις των δεδομένων ακουστικών, παρατηρείται ότι το MSE κυμαίνεται από 0.5 dB^2 μέχρι και 12 dB^2. Στόχος του headphone equalization είναι η μείωση του σφάλματος σε τιμές κοντά στο μηδέν.

C. Επιθυμητή απόκριση φίλτρων

Για να επιτευχθεί το target, είναι αναγκαία η εφαρμογή φίλτρων στο headphone response. Μάλιστα, λόγω της πολυπλοκότητας των καμπυλών, απαιτείται χρήση πολλαπλών φίλτρων, διαφορετικού τύπου και χαρακτηριστικών το καθένα. Εάν θεωρηθεί η παράλληλη (cascaded) εφαρμογή τους στο headphone response ως μία συνάρτηση μεταφοράς (transfer function) και συμβολιστεί με H(f), εάν συμβολιστεί με X(f) η είσοδος δηλαδή το headphone response και με Y(f) η επιθυμητή έξοδος δηλαδή το target response, τότε για την ιδανική H(f) ισχύει:

$$Y(f) = H(f) \cdot X(f) \Longrightarrow H(f) = \frac{Y(f)}{X(f)}$$

Στο Σχ. 2 παρουσιάζεται ενδεικτικά η επιθυμητή απόκριση των φίλτρων (desired filter response) για το 4ο από τα δεδομένα ακουστικά με στόχο το Harman target.



Σχ. 2: Η αναπαράσταση του desired filter response για ένα από τα δεδομένα ακουστικά ώστε να επιτευχθεί το Harman target.

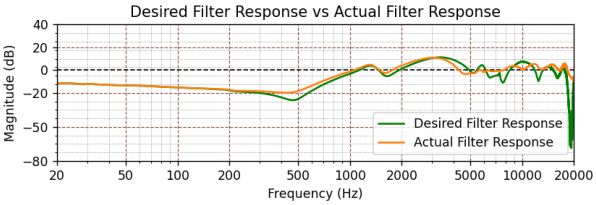
Η ιδανική αυτή απόκριση των φίλτρων δεν είναι πάντα υλοποιήσιμη λόγω της πολυπλοκότητας των καμπυλών της, για αυτό και το σφάλμα δεν μηδενίζεται ποτέ πραγματικά. Επομένως, σκοπός του equalization είναι η προσέγγισή της.

D. Φίλτρα FIR

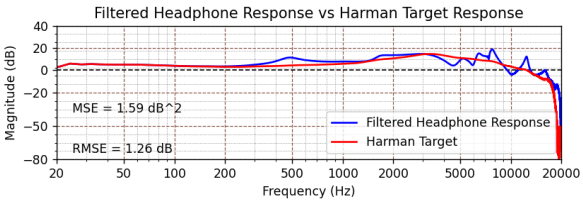
Ένας τρόπος προσέγγισης της καμπύλης της επιθυμητής απόκρισης των φίλτρων είναι η μέθοδος frequency sampling for FIR filter design [10]. Σύμφωνα με τη θεωρία, η εφαρμογή του αντίστροφου μετασχηματισμού Fourier (Inverse Fourier Transform - IFT) στην καμπύλη αυτή επιστρέφει τους συντελεστές στον αριθμητή (b coefficients) της συνάρτησης μεταφοράς. Οι ίδιοι συντελεστές ύστερα χρησιμοποιούνται για την υλοποίηση ενός φίλτρου FIR, το οποίο τελικά θα

έχει απόκριση συχνότητας που θα προσεγγίζει την επιθυμητή.

Στο Σχ. 3 φαίνεται η προσπάθεια προσέγγισης της επιθυμητής καμπύλης και στο Σχ. 4 παρουσιάζεται το αποτέλεσμα της εφαρμογής του παραγόμενου FIR φίλτρου για το 4ο headphone response με στόχο το Harman target.



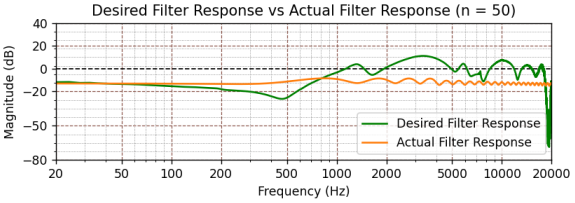
Σχ. 3: Η προσέγγιση της επιθυμητής καμπύλης απόκρισης με φίλτρο FIR μέσω της μεθόδου frequency sampling.



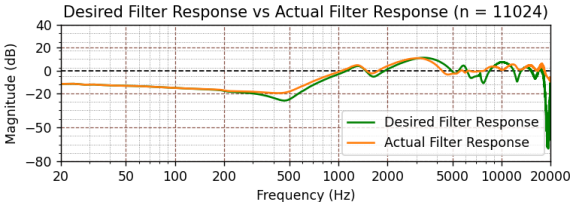
Σχ. 4: Τα αποτελέσματα μετά από φιλτράρισμα του headphone response με το παραγόμενο FIR φίλτρο. Σημειώνεται ότι το αρχικό MSE ήταν 11.32 dB^2.

Είναι φανερό ότι η προσέγγιση είναι επιτυχημένη στο συγκεκριμένο παράδειγμα και το σφάλμα έχει μειωθεί σημαντικά. Ωστόσο, μετά από εφαρμογή της μεθόδου στις διάφορες μετρήσεις των ακουστικών για το Flat και το Harman target, παρατηρήθηκε ότι η βελτίωση δεν είναι πάντα εγγυημένη, καθώς σημειώθηκε αύξηση του σφάλματος σε συγκεκριμένες περιπτώσεις.

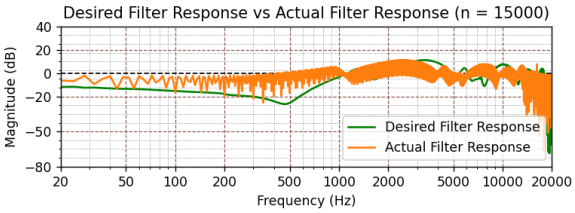
Ένα μέγεθος που επηρεάζει άμεσα τα παραγόμενα αποτελέσματα είναι το πλήθος των συντελεστών που υπολογίζονται από τον IFT. Σε περίπτωση που ο χρήστης δεν δώσει συγκεκριμένο πλήθος συντελεστών, η Python χρησιμοποιεί το διπλάσιο του μήκους του FT της καμπύλης ως default πλήθος και από τα αποτελέσματα φαίνεται ότι αυτό είναι το βέλτιστο. Θεωρητικά, όσο μεγαλύτερος είναι ο αριθμός των συντελεστών, τόσο πιο ακριβής είναι η προσέγγιση. Παρόλα αυτά, αυξάνοντας το πλήθος τους πάνω από το βέλτιστο, παρατηρείται το φαινόμενο Gibbs [11], με αποτέλεσμα να εμφανίζονται παρασιτικές ανεπιθύμητες καμπύλες. Τα Σχ. 5-7 συνοψίζουν όλες τις παραπάνω περιπτώσεις για το 4ο headphone response με στόχο το Harman target.



Σχ. 5: Χρήση μικρού αριθμού συντελεστών (underfitting) για τον σχεδιασμό του FIR φίλτρου που προσεγγίζει την επιθυμητή απόκριση.



Σχ. 6: Χρήση του βέλτιστου αριθμού συντελεστών (balanced fitting) για τον σχεδιασμό του FIR φίλτρου που προσεγγίζει την επιθυμητή απόκριση.



Σχ. 7: Χρήση μεγάλου αριθμού συντελεστών (overfitting) για τον σχεδιασμό του FIR φίλτρου που προσεγγίζει την επιθυμητή απόκριση.

E. Φίλτρα SOS

Ο κύριος λόγος για τον οποίο τα FIR φίλτρα αποτυγχάνουν να προσεγγίσουν την καμπύλη επιθυμητής απόκρισης με ακρίβεια είναι επειδή στην συνάρτηση μεταφοράς τους περιέχουν συντελεστές μόνο στον αριθμητή (b coefficients) και θεωρούν τον παρονομαστή ίσο με τη μονάδα. Ως αποτέλεσμα, χάνονται οι ιδιομορφίες της καμπύλης, όπως για παράδειγμα καμπάνες και απότομες καμπύλες. Την λύση για το πρόβλημα αυτό αποτελούν τα Infinite Impulse Response (IIR) φίλτρα, που σε αντίθεση με τα FIR έχουν συντελεστές τόσο στον αριθμητή (b) όσο και στον παρονομαστή (a coefficients).

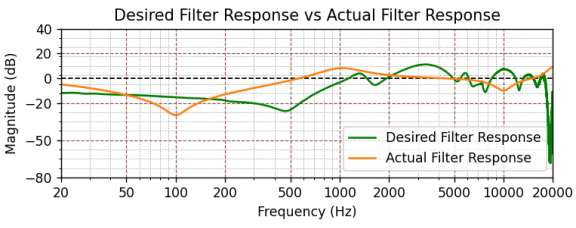
Για να προσεγγιστούν, όμως, οι ιδιομορφίες της καμπύλης, απαιτείται υλοποίηση IIR φίλτρων μεγάλης τάξης, που είναι ασταθή και ανακριβή. Για τον λόγο αυτό, τα IIR φίλτρα μπορούν να μετατραπούν σε Second Order Sections (SOS), τα οποία σύμφωνα με την θεωρία [12] προσφέρουν ακρίβεια και σταθερότητα, λόγω της χαμηλής τους τάξης. Έτσι, υλοποιήθηκε συνάρτηση που λαμβάνει ως είσοδο όλες τις αναγκαίες πληροφορίες για τα φίλτρο που πρόκειται να σχεδιαστεί και το υλοποιεί ως SOS. Συγκεκριμένα, τα χαρακτηριστικά που δέχεται ως είσοδο η συνάρτηση είναι:

- Οι κεντρικές συχνότητες (Wo) οι οποίες πρόκειται να επηρεαστούν από το φίλτρο.
- Οι συντελεστές ποιότητας (Q) για τις κεντρικές συχνότητες, οι οποίοι εκφράζουν το πόσο απότομη είναι η καμπύλη.
- Τα κέρδη-εντάσεις (gain), που υποδεικνύουν την αυξομείωση σε dB της έντασης της κάθε κεντρικής συχνότητας.
- Ο τύπος του φίλτρου σε κάθε κεντρική συχνότητα, όπως για παράδειγμα lowpass, notch και άλλα.

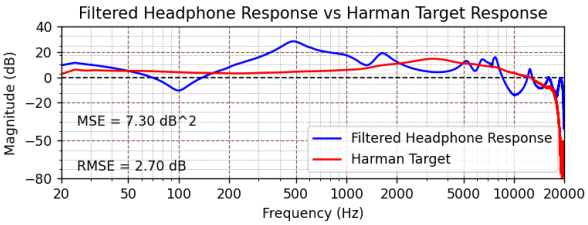
Στα Σχ. 8, 9 παρουσιάζεται ενδεικτικά ένα παράδειγμα χρήσης της συνάρτησης για το 4ο headphone response με στόχο το Harman target, εφαρμόζοντας το φιλτράρισμα του πίνακα Ι:

Πίνακας Ι. Φιλτράρισμα παραδείγματος

Τύπος	Wo (kHz)	Q	Gain (dB)
bell	0.1	0.8	-30
bell	1	1	10
bell	10	2	-10
highshelf	18	1	10



Σχ. 8: Η απόκριση του φίλτρου του παραδείγματος.



Σχ. 9: Το αποτέλεσμα της εφαρμογής του φίλτρου του παραδείγματος. Σημειώνεται ότι το αρχικό MSE ήταν 11.32 dB^2.

F. Αλγόριθμος για automatic equalization

Ο κύριος σκοπός της εργασίας είναι η ανάπτυξη προγράμματος που να εκτελεί την διαδικασία του equalization αυτόματα. Αυτό σημαίνει ότι το πρόγραμμα θα πρέπει χωρίς να αλληλοεπιδράσει ο χρήστης με αυτό να ανιχνεύει σε ποιες συχνότητες απαιτείται εφαρμογή φίλτρου και να υπολογίζει τα χαρακτηριστικά του, όπως ενεργεί και ο αλγόριθμος του AutoEQ [6, 7].

Αρχικά, για την ανίχνευση των συχνοτήτων που χρήζουν επεξεργασίας, ο αλγόριθμος αξιοποιεί την καμπύλη επιθυμητής απόκρισης για το διάστημα από 20 Hz έως 20 kHz, που είναι και τα όρια της ανθρώπινης ακοής. Με χρήση της συνάρτησης find_peaks της SciPy.signal, εντοπίζει τα θετικά τοπικά μέγιστα και τα αρνητικά τοπικά ελάχιστα και εξετάζει αν τα σημεία αυτά αποτελούν σημεία καμπής για καμπάνες, δηλαδή αν είναι οι κορυφές τους. Ο λόγος πίσω από αυτό τον έλεγχο είναι ότι μελετώντας τις καμπύλες των επιθυμητών αποκρίσεων φίλτρων, παρατηρείται έντονα η ύπαρξη πολλαπλών τμημάτων που έχουν μορφή καμπάνας.

Για την εξακρίβωση της καμπανοειδούς μορφής των μελετώμενων τμημάτων εξετάζεται το 3 dB bandwidth [13]. Εάν είναι δυνατόν αυτό να ευρεθεί, δηλαδή αν η ένταση συνεχίζει να μειώνεται πριν την αριστερή και μετά την δεξιά συχνότητα, τότε η καμπάνα θεωρείται έγκυρη. Αν δεν μπορεί να ευρεθεί η συχνότητα στην οποία ξεκινά ή τελειώνει το bandwidth, τότε το αντίστοιχο τμήμα της καμπύλης απορρίπτεται και δεν αντιμετωπίζεται καθώς κάτι τέτοιο θα απαιτούσε πιο λεπτομερή ανάλυση και χρήση πολύπλοκων φίλτρων.

Το επόμενο βήμα του αλγορίθμου είναι ο έλεγχος για επικαλυπτόμενες (overlapping) καμπάνες για καθεμία από τις έγκυρες. Αυτό γίνεται ελέγχοντας την συχνότητα στην οποία αρχίζει και τελειώνει η καθεμία. Αν μία καμπάνα αρχίζει εντός του bandwidth μίας άλλης, τότε θεωρείται ότι αυτές επικαλύπτονται και δεν μπορούν να αντιμετωπιστούν η καθεμία ξεχωριστά. Έτσι, όταν εντοπίζεται επικάλυψη, απορρίπτεται η καμπάνα με το μικρότερο τοπικό ακρότατο, δηλαδή με το μικρότερο gain.

Εάν αντιμετωπίζονταν όλες οι καμπάνες συμπεριλαμβανομένων και των επικαλυπτόμενων, τότε στα επόμενα στάδια του αλγορίθμου θα εφαρμοζόταν υπερβολικά μεγάλο πλήθος φίλτρων, τα οποία θα είχαν γειτονικές κεντρικές συχνότητες πολύ κοντά η μία στην άλλη. Ως αποτέλεσμα, τα φίλτρα θα αλληλοεπηρεαζόντουσαν, δημιουργώντας έτσι παρασιτικές καμπύλες και μειώνοντας την ακρίβεια. Κρατώντας μόνο τις καμπάνες από αυτές που επικαλύπτονται με το υψηλότερο σε μέτρο gain, εξασφαλίζεται ότι αντιμετωπίζονται τα τμήματα που χρήζουν σημαντικότερης επεξεργασίας και ότι το αποτέλεσμα προστατεύεται από τις παραπάνω παρενέργειες.

Για τις έγκυρες καμπάνες, επομένως, υπολογίζονται τα εξής χαρακτηριστικά, τα οποία και χρησιμοποιούνται ίδια για την υλοποίηση φίλτρων καμπάνας:

- Η κεντρική συχνότητα (Wo), που είναι η συχνότητα στην οποία εμφανίζεται το τοπικό μέγιστο ή ελάχιστο.
- Το gain, που είναι η τιμή (ένταση) του τοπικού μεγίστου ή ελαχίστου.
- Ο συντελεστής ποιότητας (Q), που υπολογίζεται σύμφωνα με την θεωρία [14] από τον τύπο:

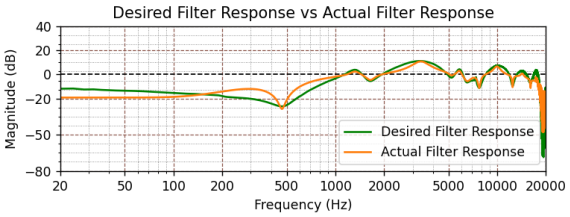
$$Q = \frac{center\ frequency\ (Wo)}{3\ dB\ bandwidth}$$

Ο αλγόριθμος, εκτός από τα καμπανοειδή τμήματα, αντιμετωπίζει και τις καμπύλες στα άκρα του διαστήματος, δηλαδή κοντά στα 20 Hz και στα 20 kHz. Από τη μελέτη των επιθυμητών αποκρίσεων φίλτρων παρατηρούνται στα άκρα του διαστήματος καμπύλες όμοιες με αυτές των αποκρίσεων lowpass και highpass φίλτρων. Ο αλγόριθμος εντοπίζει τις κεντρικές συχνότητες του πρώτου και το τελευταίου φίλτρου καμπάνας, δηλαδή στη μικρότερη και στην υψηλότερη συχνότητα, έστω f_A και f_B αντίστοιχα. Για το διάστημα από 20 Hz έως f_A ο αλγόριθμος εφαρμόζει ένα lowshelf φίλτρο, ενώ για το διάστημα από f_B έως 20 kHz εφαρμόζει ένα highshelf φίλτρο.

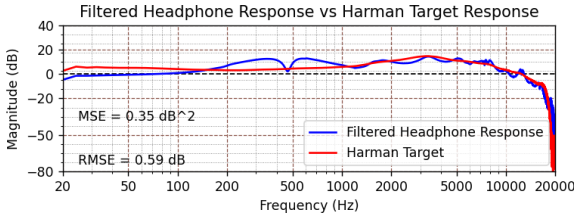
Τα δύο διαστήματα θεωρούνται ως ευθείες και για το καθένα υπολογίζεται η κλίση του μεταξύ των ακρών του, της οποίας το συνημίτονο χρησιμοποιείται ως shelf slope (S) για το φίλτρο, που εκφράζει πόσο απότομη είναι η καμπύλη του. Η κλίση μετατρέπεται σε συνημίτονο λόγω του περιορισμού του S να έχει τιμές από 0 έως 1. Τα χαρακτηριστικά των lowshelf και highshelf φίλτρων που χρησιμοποιούνται είναι:

- Η κεντρική συχνότητα (Wo), που είναι το μέσον του κάθε διαστήματος.
- Το gain, που είναι ο μέσος όρος μεταξύ των εντάσεων στα άκρα του κάθε διαστήματος.
- Το shelf slope (S), όπως αυτό υπολογίστηκε.

Στα Σχ. 10, 11 παρουσιάζεται η εφαρμογή του αλγορίθμου στο 4ο headphone response, με στόχο το Harman target.



Σχ. 10: Η απόκριση φίλτρου έπειτα από εφαρμογή του αλγορίθμου.



Σχ. 11: Το αποτέλεσμα της εφαρμογής του αλγορίθμου στο headphone response του παραδείγματος. Το αρχικό MSE ήταν 11.32 dB^2.

Επίσης, στο terminal τυπώνεται το πλήθος των φίλτρων που απαιτούνται για το συγκεκριμένο αποτέλεσμα, που είναι 54. Εκτός από αυτά,

εμφανίζεται και η λίστα με όλα τα φίλτρα που υλοποιούνται και τα χαρακτηριστικά τους.

Από τα Σχ. 10, 11 είναι φανερό ότι ο αλγόριθμος προσεγγίζει την επιθυμητή καμπύλη απόκρισης των φίλτρων με επιτυχία. Ωστόσο, στις υψηλές συχνότητες κοντά στα 20 kHz, όπου η επιθυμητή καμπύλη παρουσιάζει απότομες και σχεδόν κάθετες αλλαγές στην έντασή της, ο αλγόριθμος δεν την ακολουθά με ακρίβεια, λόγω ύπαρξης μικρών και στενών καμπανών, των οποίων τα χαρακτηριστικά υπολογίζονται με δυσκολία. Από την λίστα με όλα τα φίλτρα φαίνεται ότι η συντριπτική πλειοψηφία τους, συγκεκριμένα τα 37 από τα 54, έχουν κεντρική συχνότητα από 17.5 kHz και πάνω. Τέλος, σημειώνεται μείωση του σφάλματος (MSE) κατά 96.9% .

Οι παραπάνω παρατηρήσεις αποδεικνύουν ότι ο αλγόριθμος προσεγγίζει την επιθυμητή καμπύλη με ακρίβεια. Τα σφάλματα στις υψηλές συχνότητες όμως χρήζουν ειδικής αντιμετώπισης λόγω της ιδιομορφίας της επιθυμητής απόκρισης σε αυτές, και δημιουργούν χώρο για βελτίωση του αλγορίθμου.

III. ΣΥΝΟΨΗ

Η υλοποίηση του προγράμματος για την αυτόματη ισοστάθμιση ακουστικών συνδύασε την χρήση ενός συνόλου γνώσεων και μεθόδων που αφορούν την επεξεργασία ήχου και ανάλυση δεδομένων στο πεδίο της συχνότητας.

Παρ’ ότι περιορισμένη σε εύρος, η εργασία έδωσε μία πολύ χρήσιμη πρώτη επαφή τόσο με τα προτερήματα της αυτοματοποίησης της διαδικασίας της ισοστάθμισης, όσο και με τα εμπόδια που προκύπτουν κατά την προσπάθεια της αυτοματοποίησης. Η ανάγκη εύρεσης και υλοποίησης από το μηδέν ενός αλγορίθμου που να αυτοματοποιεί μία διαδικασία που μέχρι και σήμερα οι επαγγελματίες πραγματοποιούν χειροκίνητα βάσει εμπειρίας, ήταν με διαφορά το πιο ενδιαφέρον αλλά και απαιτητικό μέρος της εργασίας. Επιπλέον, ήταν ιδιαίτερα χρήσιμο το γεγονός ότι απαιτήθηκε κάποια έρευνα σε ένα σύνολο πόρων (κυρίως στο διαδίκτυο) για να βρεθεί η πληροφορία σχετικά με τον τρόπο που πραγματοποιείται η ισοστάθμιση, καθώς και για να ξεπεραστούν τα προβλήματα που εμφανίστηκαν κατά τη διάρκεια της ανάπτυξης.

Τέλος, το πρόγραμμα και ο αλγόριθμος σχεδιάστηκαν με απλές προδιαγραφές και λογική, καθώς δόθηκε περισσότερη έμφαση σε θέματα κατανόησης της θεωρίας πίσω από τις διάφορες τεχνικές επεξεργασίας ήχου και των εργαλείων επεξεργασίας στο πεδίο της συχνότητας με την Python. Παρόλα αυτά, τα αποτελέσματα, όπως παρουσιάστηκαν παραπάνω, υποδεικνύουν επιτυχημένη εφαρμογή του αλγορίθμου και των τεχνικών, αλλά παράλληλα αφήνουν και χώρο για περαιτέρω βελτίωση της λογικής του αλγορίθμου.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1]. https://en.wikipedia.org/wiki/Equalization_(audio)

[2]. https://www.soundguys.com/how-to-equalize-fine-tune-your-listening-experience-16410/

[3]. https://autoeq.app/

[4]. https://github.com/jaakkopasanen/AutoEq

[5]. https://docs.scipy.org/doc/scipy/reference/signal.html

[6]. https://github.com/jaakkopasanen/AutoEq/wiki/How-Does-AutoEq-Work%3F

[7]. https://patents.google.com/patent/EP1843636B1/en

[8]. https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=120742201

[9]. https://en.wikipedia.org/w/index.php?title=Root-mean-square_deviation&oldid=1204670839

[10]. Pushpavathi.K.P, B.Kanmani, Frequency Sampling method of FIR Filter design: A comparative study, 2018

[11]. https://en.wikipedia.org/wiki/Gibbs_phenomenon

[12]. https://en.wikipedia.org/wiki/Digital_biquad_filter

[13]. https://www.quora.com/What-is-meant-by-3-dB-bandwidth

[14]. https://ecstudiosystems.com/discover/textbooks/basic-electronics/filters/quality-factor-and-bandwidth/

ΠΑΡΑΡΤΗΜΑ Ι

Όπως ζητήθηκε στην εκφώνηση της εργασίας, στο παράρτημα παρατίθεται ο κώδικας που αναπτύχθηκε, καθώς και οδηγίες εγκατάστασης και χρήσης του.

Οδηγίες εγκατάστασης:

Το πρόγραμμα δημιουργήθηκε με χρήση της Python 3.10.10 . Επίσης, χρησιμοποιούνται οι βιβλιοθήκες:

- matplotlib==3.8.3
- numpy==2.0.1
- scipy==1.14.0

Οι παραπάνω βιβλιοθήκες είναι απαραίτητο να εγκατασταθούν ώστε να τρέξει ο κώδικας. Αφότου ο χρήστης εγκαταστήσει την παραπάνω έκδοση της Python μέσω της επίσημης ιστοσελίδας (https://www.python.org/downloads/release/python-31010/), μπορεί να εκτελέσει στο command line την παρακάτω εντολή ώστε να εγκαταστήσει τις βιβλιοθήκες:

pip install -r /path/to/requirements.txt

Το requirements.txt είναι ένα αρχείο που περιέχεται μαζί με τον κώδικα και περιέχει τις βιβλιοθήκες που χρησιμοποιούνται. Η χρήση διαφορετικών εκδόσεων είτε για την Python είτε για τις βιβλιοθήκες δεν εγγυάται την σωστή εκτέλεση του προγράμματος.

Μόλις εγκατασταθούν η Python και οι βιβλιοθήκες, ο χρήστης έχει την δυνατότητα να εκτελέσει το πρόγραμμα.

Οδηγίες χρήσης προγράμματος:

Η εκτέλεση του προγράμματος μπορεί να γίνει είτε από command line ή με την χρήση εφαρμογής ανάπτυξης κώδικα.

Για το κάθε task της εργασίας, δημιουργήθηκε ένα ξεχωριστό αρχείο κώδικα. Τα αρχεία δεν αλληλεξαρτώνται και δεν απαιτούν άλλα αρχεία παρά τα headphone responses και τα targets, τα οποία αποθηκεύονται στους ομώνυμους φακέλους headphone_responses και targets. Οι φάκελοι αυτοί είναι αναγκαίο να βρίσκονται στο ίδιο directory με τα αρχεία κώδικα. Σε περίπτωση που ο χρήστης επιθυμεί να χρησιμοποιήσει δικά του headphone ή target responses, δηλαδή διαφορετικά από τα δεδομένα της εργασίας, θα πρέπει:

- Τα αρχεία των headphone responses να είναι τύπου .wav και να έχουν ονομασία hpxy.wav, όπου xy ο αύξων αριθμός τους (2 ψηφία), ξεκινώντας από το 01. Για παράδειγμα, αν ο χρήστης έχει 3 headphone responses, θα πρέπει να τα ονομάσει hp01.wav, hp02.wav και hp03.wav .
- Τα αρχεία των targets να είναι και αυτά τύπου .wav . Δεν υπάρχει περιορισμός στην ονομασία, όμως ο χρήστης θα πρέπει να αντικαταστήσει στις εντολές που διαβάζουν τα αρχεία των targets τις default ονομασίες αυτών με τις ονομασίες των δικών του αρχείων. Για παράδειγμα, αν ο χρήστης έχει ένα target με όνομα target.wav, θα πρέπει να τοποθετήσει στην εντολή την αντίστοιχη ονομασία: wavfile.read('targets/target.wav') .
- Να αλλάξει τις τιμές των μεταβλητών hpfilesnum και targetsnum με τον αριθμό των αρχείων headphone responses και των αρχείων targets αντίστοιχα.

Δεν απαιτείται η αλληλεπίδραση του χρήστη με το πρόγραμμα σε κανένα στάδιο της διαδικασίας. Αυτό σημαίνει ότι μόλις ο χρήστης ακολουθήσει τα παραπάνω βήματα για να επεξεργαστεί δικά του headphone responses και targets αν το επιθυμεί, μπορεί να τρέξει το πρόγραμμα χωρίς να χρειαστεί να εκτελέσει αυτός την οποιαδήποτε ενέργεια. Μοναδική εξαίρεση αποτελεί το πρόγραμμα για το task 4, στο οποίο ο χρήστης μέσω του terminal καλείται να δώσει το πλήθος των συντελεστών των FIR φίλτρων που υλοποιούνται. Για την συγκεκριμένη περίπτωση, τυπώνονται σχετικές οδηγίες στο terminal, που αφορούν τις έγκυρες τιμές.

Κώδικας:

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε για κάθε task της εργασίας.

```python
# Panos Lelakis, 1083712, Headphone EQ, task 1

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward') # norm='forward' for 1/n scaling
    hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
    hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
    hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

    plt.figure(i + 1)

    # Plot headphone response
    plt.subplot(targetsnum + 1, 1, 1)
    plt.plot(hp_freq, hp_mag_db, color='b')
    plt.gca().set_xscale('log')
    plt.gca().xaxis.set_major_formatter(ScalarFormatter())
```

```
    plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
    plt.gca().yaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
    plt.xlim([20, 20000])
    plt.ylim([-80, 40])
    plt.title(f'Headphone Response {i + 1}')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude (dB)')
    plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
    plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
    plt.gca().get_ygridlines()[3].set_color('k')
    plt.gca().get_ygridlines()[3].set_linewidth(1)
    plt.gca().minorticks_on()
    # Source for changing color and linewidth for specific gridlines:
    # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-grid-
lines
    # Source for making all ticks visible using ScalarFormatter():
    #
https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log
_scale/

    # Plot target responses in frequency domain
    for j in range(targetsnum):
        # Calculate frequency response of target
        target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
        target_mag = np.abs(target_fft)/2
        target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
        target_mag_db = 20 * np.log10(target_mag) - scaling_db

        # Calculate MSE and RMSE
        mse = np.mean((hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Source for mse and rmse formulas:
        # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

        # Plot headphone response vs target response
        plt.subplot(targetsnum + 1, 1, j + 2)
        plt.plot(hp_freq, hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
```

```
            plt.ylim([-80, 40])
            legend = 'Flat Target' if j == 0 else 'Harman Target'
            plt.title(f'Headphone Response {i + 1} vs {legend} Response')
            plt.legend(['Headphone Response', legend], loc='lower right')
            plt.xlabel('Frequency (Hz)')
            plt.ylabel('Magnitude (dB)')
            plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
            plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
            plt.gca().get_ygridlines()[3].set_color('k')
            plt.gca().get_ygridlines()[3].set_linewidth(1)
            plt.gca().minorticks_on()
            # Display MSE and RMSE inside the plot
            plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
            plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
            # Source for displaying text inside the plot using transAxes:
            # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Adjust layout to prevent overlap
        plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

```python
# Panos Lelakis, 1083712, Headphone EQ, task 2

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import freqz
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward') # norm='forward' for 1/n scaling
    hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
    hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
    hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

    plt.figure(i + 1)

    # Plot target responses in frequency domain
    for j in range(targetsnum):
        # Calculate frequency response of target
        target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
```

```
        target_mag = np.abs(target_fft)/2
        target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
        target_mag_db = 20 * np.log10(target_mag) - scaling_db

        # Calculate initial MSE and RMSE
        mse = np.mean((hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Source for mse and rmse formulas:
        # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

        # Plot headphone response vs target response
        plt.subplot(3, targetsnum, j + 1)
        plt.plot(hp_freq, hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Headphone Response {i + 1} vs {legend} Response')
        plt.legend(['Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Display initial MSE and RMSE inside the plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
        # Source for changing color and linewidth for specific gridlines:
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-
grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        #
https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log
_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Calculate desired filter frequency response
```

```
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Design the FIR filter using the frequency sampling method. Source:
        # https://dsp.stackexchange.com/questions/67338/building-a-fir-filter-from-an-
arbitrary-frequency-magnitude-response-curve-eg
        filter_coefs = np.fft.irfft(desired_mag)

        # Calculate frequency response of actual filter
        w, h = freqz(filter_coefs, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Apply the filter to the headphone response in the frequency domain
        filtered_hp_fft = hp_fft * h_mag # Convolution in time domain = multiplication in
frequency domain

        # Scale and adjust filtered headphone response
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i] / 2, len(filtered_hp_mag_db))

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title('Desired Filter Response vs Actual Filter Response')
        plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
```

```
        # Plot filtered headphone response vs target response
        plt.subplot(3, targetsnum, 5 + j)
        plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Filtered Headphone Response vs {legend} Response')
        plt.legend(['Filtered Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Calculate new MSE and RMSE
        mse = np.mean((filtered_hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Display new MSE and RMSE inside the new plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)

    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

```python
# Panos Lelakis, 1083712, Headphone EQ, task 3

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import freqz, convolve
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward')  # norm='forward' for 1/n scaling
    hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
    hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
    hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

    plt.figure(i + 1)

    # Plot target responses in frequency domain
    for j in range(targetsnum):
        # Calculate frequency response of target
        target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
```

```
        target_mag = np.abs(target_fft)/2
        target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
        target_mag_db = 20 * np.log10(target_mag) - scaling_db

        # Calculate initial MSE and RMSE
        mse = np.mean((hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Source for mse and rmse formulas:
        # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

        # Plot headphone response vs target response
        plt.subplot(3, targetsnum, j + 1)
        plt.plot(hp_freq, hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Headphone Response {i + 1} vs {legend} Response')
        plt.legend(['Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Display initial MSE and RMSE inside the plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
        # Source for changing color and linewidth for specific gridlines:
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-
grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        #
https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log
_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Calculate desired filter frequency response
```

```
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Design the FIR filter using the frequency sampling method. Source:
        # https://dsp.stackexchange.com/questions/67338/building-a-fir-filter-from-an-
arbitrary-frequency-magnitude-response-curve-eg
        filter_coefs = np.fft.irfft(desired_mag)

        # Calculate frequency response of actual filter
        w, h = freqz(filter_coefs, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Apply the filter to the headphone response using convolution in the time domain
        filtered_hp = convolve(filter_coefs, hp[i])
        # scipy.signal.convolve is used because of being faster (instead of numpy, lfilter or
other). Source:
        # https://scipy-cookbook.readthedocs.io/items/ApplyFIRFilter.html

        # Calculate FFT of filtered headphone response + scale/adjust
        filtered_hp_fft = np.fft.rfft(filtered_hp, len(hp[i]), norm='forward')
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i]/2, len(filtered_hp_mag))

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title('Desired Filter Response vs Actual Filter Response')
        plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
```

```
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()

        # Plot filtered headphone response vs target response
        plt.subplot(3, targetsnum, 5 + j)
        plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Filtered Headphone Response vs {legend} Response')
        plt.legend(['Filtered Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Calculate new MSE and RMSE
        mse = np.mean((filtered_hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Display new MSE and RMSE inside the new plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)

    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

```python
# Panos Lelakis, 1083712, Headphone EQ, task 4

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import freqz
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Ask user to give number of coefficients for the FIR filter - just for testing
while True:
    try:
        coef_num = int(input('Enter number of coefficients for the FIR filter (enter 0 for
best - computed automatically): '))
        if coef_num >= 0:
            break
        else:
            print('Invalid input. Enter positive integer.')
            continue
    except ValueError:
        print('Invalid input. Enter positive integer.')
        continue
```

```
# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward')  # norm='forward' for 1/n scaling
    hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
    hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
    hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

    plt.figure(i + 1)

    # Plot target responses in frequency domain
    for j in range(targetsnum):
        # Calculate frequency response of target
        target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
        target_mag = np.abs(target_fft)/2
        target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
        target_mag_db = 20 * np.log10(target_mag) - scaling_db

        # Calculate initial MSE and RMSE
        mse = np.mean((hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Source for mse and rmse formulas:
        # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

        # Plot headphone response vs target response
        plt.subplot(3, targetsnum, j + 1)
        plt.plot(hp_freq, hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Headphone Response {i + 1} vs {legend} Response')
        plt.legend(['Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
```

```
        # Display initial MSE and RMSE inside the plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
        # Source for changing color and linewidth for specific gridlines:
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-
grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        #
https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log
_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Calculate desired filter frequency response
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Design the FIR filter using the frequency sampling method. Source:
        # https://dsp.stackexchange.com/questions/67338/building-a-fir-filter-from-an-
arbitrary-frequency-magnitude-response-curve-eg
        if coef_num == 0:
            filter_coefs = np.fft.irfft(desired_mag)
        else:
            filter_coefs = np.fft.irfft(desired_mag, n=coef_num)

        # Calculate frequency response of actual filter
        w, h = freqz(filter_coefs, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Apply the filter to the headphone response in the frequency domain
        filtered_hp_fft = hp_fft * h_mag # Convolution in time domain = multiplication in
frequency domain

        # Scale and adjust filtered headphone response
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i] / 2, len(filtered_hp_mag_db))

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
```

```
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title(f'Desired Filter Response vs Actual Filter Response (n =
{len(filter_coefs)})')
        plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()

        # Plot filtered headphone response vs target response
        plt.subplot(3, targetsnum, 5 + j)
        plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Filtered Headphone Response vs {legend} Response')
        plt.legend(['Filtered Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Calculate new MSE and RMSE
        mse = np.mean((filtered_hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        # Display new MSE and RMSE inside the new plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
```

```
    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

```python
# Panos Lelakis, 1083712, Headphone EQ, task 5

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import sosfilt, sosfreqz, tf2sos
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Function to desgin SOS filter, based on the MATLAB designParamEQ function. Source:
# https://www.mathworks.com/help/audio/ug/parametric-equalizer-design.html
def design_parametric(Wo, Qo, gain, fs, filter_type):
    if not (len(Wo) == len(Qo) and len(Wo) == len(gain) and len(Wo) == len(filter_type)):
        raise ValueError('Error: all input arrays must have same length')

    filters = []

    for i in range(len(Wo)):
        wo = 2 * np.pi * Wo[i] / fs
        alpha = np.sin(wo) / (2 * Qo[i])
        A = 10 ** (gain[i] / 40)

        # Get coefficients depending on the filter type
```

```
        if filter_type[i] == 'notch':
            b0 = 1
            b1 = -2 * np.cos(wo)
            b2 = 1
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowpass':
            b0 = (1 - np.cos(wo)) / 2
            b1 = 1 - np.cos(wo)
            b2 = (1 - np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'highpass':
            b0 = (1 + np.cos(wo)) / 2
            b1 = -(1 + np.cos(wo))
            b2 = (1 + np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'bandpass':
            b0 = np.sin(wo) / 2
            b1 = 0
            b2 = -np.sin(wo) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'allpass':
            b0 = 1 - alpha
            b1 = -2 * np.cos(wo)
            b2 = 1 + alpha
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowshelf':
            # For shelving filters, alpha is different. In this case, the input Qo is S
(Slope)
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
```

```
            b1 = 2 * A * ((A - 1) - (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = -2 * ((A - 1) + (A + 1) * np.cos(wo))
            a2 = (A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'highshelf':
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
            b1 = -2 * A * ((A - 1) + (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = 2 * ((A - 1) - (A + 1) * np.cos(wo))
            a2 = (A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'bell': # Bell (peaking) filter
            b0 = 1 + alpha * A
            b1 = -2 * np.cos(wo)
            b2 = 1 - alpha * A
            a0 = 1 + alpha / A
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha / A

        else:
            raise ValueError('Unsupported filter type')

        b = [b0, b1, b2]
        a = [a0, a1, a2]
        # Normalize coefficients:
        b = np.array(b) / a0
        a = np.array(a) / a0

        # Formulas taken from AudioeqCookbook - source:
        # https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

        filters.append(tf2sos(b, a)) # tf2sos transforms b,a coefficients to SOS

    # Source for using vstack to stack vertically filters:
    # https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/
    return np.vstack(filters)

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward')  # norm='forward' for 1/n scaling
```

```
hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

plt.figure(i + 1)

# Plot target responses in frequency domain
for j in range(targetsnum):
    # Calculate frequency response of target
    target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
    target_mag = np.abs(target_fft)/2
    target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
    target_mag_db = 20 * np.log10(target_mag) - scaling_db

    # Calculate initial MSE and RMSE
    mse = np.mean((hp_mag - target_mag) ** 2)
    rmse = np.sqrt(mse)
    # Source for mse and rmse formulas:
    # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

    # Plot headphone response vs target response
    plt.subplot(3, targetsnum, j + 1)
    plt.plot(hp_freq, hp_mag_db, color='b')
    plt.plot(target_freq, target_mag_db, color='r')
    plt.gca().set_xscale('log')
    plt.gca().xaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
    plt.gca().yaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
    plt.xlim([20, 20000])
    plt.ylim([-80, 40])
    legend = 'Flat Target' if j == 0 else 'Harman Target'
    plt.title(f'Headphone Response {i + 1} vs {legend} Response')
    plt.legend(['Headphone Response', legend], loc='lower right')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude (dB)')
    plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
    plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
    plt.gca().get_ygridlines()[3].set_color('k')
    plt.gca().get_ygridlines()[3].set_linewidth(1)
    plt.gca().minorticks_on()
    plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
    plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
    # Source for changing color and linewidth for specific gridlines:
```

```
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        # https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Example usage of design_parametric function
        tmpsos = design_parametric([18000, 100, 1000, 10000], [1, 0.8, 1, 2], [10, -30, 10, -10], hp_fs[i], ['highshelf', 'bell', 'bell', 'bell'])

        # Apply the parametric equalizer to the headphone response
        filtered_hp = sosfilt(tmpsos, hp[i])

        # Calculate frequency response of filtered headphone
        filtered_hp_fft = np.fft.rfft(filtered_hp, len(filtered_hp), norm='forward')
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i] / 2, len(filtered_hp_mag_db))

        # Calculate desired filter frequency response
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Calculate frequency response of actual filter
        w, h = sosfreqz(tmpsos, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag + 1e-12)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title('Desired Filter Response vs Actual Filter Response')
```

```
        plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()

        # Plot filtered headphone response vs target response
        plt.subplot(3, targetsnum, 5 + j)
        plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Filtered Headphone Response vs {legend} Response')
        plt.legend(['Filtered Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        mse = np.mean((filtered_hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)

    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

```python
# Panos Lelakis, 1083712, Headphone EQ, task 6

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import sosfilt, sosfreqz, tf2sos
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Function to desgin SOS filter, based on the MATLAB designParamEQ function. Source:
# https://www.mathworks.com/help/audio/ug/parametric-equalizer-design.html
def design_parametric(Wo, Qo, gain, fs, filter_type):
    if not (len(Wo) == len(Qo) and len(Wo) == len(gain) and len(Wo) == len(filter_type)):
        raise ValueError('Error: all input arrays must have same length')

    filters = []

    for i in range(len(Wo)):
        wo = 2 * np.pi * Wo[i] / fs
        alpha = np.sin(wo) / (2 * Qo[i])
        A = 10 ** (gain[i] / 40)

        # Get coefficients depending on the filter type
```

```
        if filter_type[i] == 'notch':
            b0 = 1
            b1 = -2 * np.cos(wo)
            b2 = 1
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowpass':
            b0 = (1 - np.cos(wo)) / 2
            b1 = 1 - np.cos(wo)
            b2 = (1 - np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'highpass':
            b0 = (1 + np.cos(wo)) / 2
            b1 = -(1 + np.cos(wo))
            b2 = (1 + np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'bandpass':
            b0 = np.sin(wo) / 2
            b1 = 0
            b2 = -np.sin(wo) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'allpass':
            b0 = 1 - alpha
            b1 = -2 * np.cos(wo)
            b2 = 1 + alpha
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowshelf':
            # For shelving filters, alpha is different. In this case, the input Qo is S
(Slope)
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
```

```
            b1 = 2 * A * ((A - 1) - (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = -2 * ((A - 1) + (A + 1) * np.cos(wo))
            a2 = (A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'highshelf':
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
            b1 = -2 * A * ((A - 1) + (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = 2 * ((A - 1) - (A + 1) * np.cos(wo))
            a2 = (A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'bell': # Bell (peaking) filter
            b0 = 1 + alpha * A
            b1 = -2 * np.cos(wo)
            b2 = 1 - alpha * A
            a0 = 1 + alpha / A
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha / A

        else:
            raise ValueError('Unsupported filter type')

        b = [b0, b1, b2]
        a = [a0, a1, a2]
        # Normalize coefficients:
        b = np.array(b) / a0
        a = np.array(a) / a0

        # Formulas taken from AudioeqCookbook - source:
        # https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

        filters.append(tf2sos(b, a)) # tf2sos transforms b,a coefficients to SOS

    # Source for using vstack to stack vertically filters:
    # https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/
    return np.vstack(filters)

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward')  # norm='forward' for 1/n scaling
```

```
hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

plt.figure(i + 1)

# Plot target responses in frequency domain
for j in range(targetsnum):
    # Calculate frequency response of target
    target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
    target_mag = np.abs(target_fft)/2
    target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
    target_mag_db = 20 * np.log10(target_mag) - scaling_db

    # Calculate initial MSE and RMSE
    mse = np.mean((hp_mag - target_mag) ** 2)
    rmse = np.sqrt(mse)
    # Source for mse and rmse formulas:
    # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

    # Plot headphone response vs target response
    plt.subplot(3, targetsnum, j + 1)
    plt.plot(hp_freq, hp_mag_db, color='b')
    plt.plot(target_freq, target_mag_db, color='r')
    plt.gca().set_xscale('log')
    plt.gca().xaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
    plt.gca().yaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
    plt.xlim([20, 20000])
    plt.ylim([-80, 40])
    legend = 'Flat Target' if j == 0 else 'Harman Target'
    plt.title(f'Headphone Response {i + 1} vs {legend} Response')
    plt.legend(['Headphone Response', legend], loc='lower right')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude (dB)')
    plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
    plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
    plt.gca().get_ygridlines()[3].set_color('k')
    plt.gca().get_ygridlines()[3].set_linewidth(1)
    plt.gca().minorticks_on()
    plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
    plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
    # Source for changing color and linewidth for specific gridlines:
```

```
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        # https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Example usage of design_parametric function
        tmpsos = design_parametric([18000, 100, 1000, 10000], [1, 0.8, 1, 2], [1, 30, 10, -10], hp_fs[i], ['lowpass', 'bell', 'iirpeak', 'bell'])

        # Apply the parametric equalizer to the headphone response
        filtered_hp = sosfilt(tmpsos, hp[i])

        # Calculate frequency response of filtered headphone
        filtered_hp_fft = np.fft.rfft(filtered_hp, len(filtered_hp), norm='forward')
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i] / 2, len(filtered_hp_mag_db))

        # Calculate desired filter frequency response
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Calculate frequency response of actual filter
        w, h = sosfreqz(tmpsos, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag + 1e-12)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title('Desired Filter Response vs Actual Filter Response')
```

```
            plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
            plt.xlabel('Frequency (Hz)')
            plt.ylabel('Magnitude (dB)')
            plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
            plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
            plt.gca().get_ygridlines()[3].set_color('k')
            plt.gca().get_ygridlines()[3].set_linewidth(1)
            plt.gca().minorticks_on()

            # Plot filtered headphone response vs target response
            plt.subplot(3, targetsnum, 5 + j)
            plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
            plt.plot(target_freq, target_mag_db, color='r')
            plt.gca().set_xscale('log')
            plt.gca().xaxis.set_major_formatter(ScalarFormatter())
            plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
            plt.gca().yaxis.set_major_formatter(ScalarFormatter())
            plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
            plt.xlim([20, 20000])
            plt.ylim([-80, 40])
            legend = 'Flat Target' if j == 0 else 'Harman Target'
            plt.title(f'Filtered Headphone Response vs {legend} Response')
            plt.legend(['Filtered Headphone Response', legend], loc='lower right')
            plt.xlabel('Frequency (Hz)')
            plt.ylabel('Magnitude (dB)')
            plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
            plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
            plt.gca().get_ygridlines()[3].set_color('k')
            plt.gca().get_ygridlines()[3].set_linewidth(1)
            plt.gca().minorticks_on()
            mse = np.mean((filtered_hp_mag - target_mag) ** 2)
            rmse = np.sqrt(mse)
            plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
            plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)

    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```

Ο κώδικας του task 6 είναι ίδιος με τον κώδικα του task 5 καθώς ο αλγόριθμος για την αυτόματη εφαρμογή των φίλτρων με βέλτιστο τρόπο υλοποιείται στο task 7.

```python
# Panos Lelakis, 1083712, Headphone EQ, task 7

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.signal import sosfilt, sosfreqz, tf2sos, find_peaks
from matplotlib.ticker import ScalarFormatter

# Load headphone responses
hpfilesnum = 7
hp = []
hp_fs = []
for i in range(1, hpfilesnum + 1):
    fs, data = wavfile.read(f'headphone_responses/hp0{i}.wav')
    hp.append(data)
    hp_fs.append(fs)

# Load targets
targetsnum = 2
target = []
target_fs = []
fs, data = wavfile.read('targets/flat_target.wav')
target.append(data)
target_fs.append(fs)
fs, data = wavfile.read('targets/harman_target.wav')
target.append(data)
target_fs.append(fs)

# Scaling factor in dB for correct plots (further scaling)
scaling_db = 0

# Function to desgin SOS filter, based on the MATLAB designParamEQ function. Source:
# https://www.mathworks.com/help/audio/ug/parametric-equalizer-design.html
def design_parametric(Wo, Qo, gain, fs, filter_type):
    if not (len(Wo) == len(Qo) and len(Wo) == len(gain) and len(Wo) == len(filter_type)):
        raise ValueError('Error: all input arrays must have same length')

    filters = []

    for i in range(len(Wo)):
        wo = 2 * np.pi * Wo[i] / fs
        alpha = np.sin(wo) / (2 * Qo[i])
        A = 10 ** (gain[i] / 40)

        # Get coefficients depending on the filter type
```

```
        if filter_type[i] == 'notch':
            b0 = 1
            b1 = -2 * np.cos(wo)
            b2 = 1
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowpass':
            b0 = (1 - np.cos(wo)) / 2
            b1 = 1 - np.cos(wo)
            b2 = (1 - np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'highpass':
            b0 = (1 + np.cos(wo)) / 2
            b1 = -(1 + np.cos(wo))
            b2 = (1 + np.cos(wo)) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'bandpass':
            b0 = np.sin(wo) / 2
            b1 = 0
            b2 = -np.sin(wo) / 2
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'allpass':
            b0 = 1 - alpha
            b1 = -2 * np.cos(wo)
            b2 = 1 + alpha
            a0 = 1 + alpha
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha

        elif filter_type[i] == 'lowshelf':
            # For shelving filters, alpha is different. In this case, the input Qo is S
(Slope)
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
```

```
            b1 = 2 * A * ((A - 1) - (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = -2 * ((A - 1) + (A + 1) * np.cos(wo))
            a2 = (A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'highshelf':
            alpha = np.sin(wo) / 2 * np.sqrt((A + 1 / A) * (1 / Qo[i] - 1) + 2)
            b0 = A * ((A + 1) + (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha)
            b1 = -2 * A * ((A - 1) + (A + 1) * np.cos(wo))
            b2 = A * ((A + 1) + (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha)
            a0 = (A + 1) - (A - 1) * np.cos(wo) + 2 * np.sqrt(A) * alpha
            a1 = 2 * ((A - 1) - (A + 1) * np.cos(wo))
            a2 = (A + 1) - (A - 1) * np.cos(wo) - 2 * np.sqrt(A) * alpha

        elif filter_type[i] == 'bell': # Bell (peaking) filter
            b0 = 1 + alpha * A
            b1 = -2 * np.cos(wo)
            b2 = 1 - alpha * A
            a0 = 1 + alpha / A
            a1 = -2 * np.cos(wo)
            a2 = 1 - alpha / A

        else:
            raise ValueError('Unsupported filter type')

        b = [b0, b1, b2]
        a = [a0, a1, a2]
        # Normalize coefficients:
        b = np.array(b) / a0
        a = np.array(a) / a0

        # Formulas taken from AudioeqCookbook - source:
        # https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

        filters.append(tf2sos(b, a)) # tf2sos transforms b,a coefficients to SOS

    # Source for using vstack to stack vertically filters:
    # https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/
    return np.vstack(filters)

# Function to detect bells in the desired filter response plot
def detect_bells(signal, freqs, threshold=0.1, min_distance=10, min_prominence=0.01):
    # Detect bells in the range 20 Hz - 20 kHz
    low_index = np.where(freqs >= 20)[0][0]
```

```
    high_index = np.where(freqs <= 20000)[0][-1]
    signal = signal[low_index:high_index + 1]
    freqs = freqs[low_index:high_index + 1]

    # Sub-function to detect bells
    def find_bells(peaks, properties, signal, freqs):
        bells = []

        for i, peak in enumerate(peaks):
            peak_gain = properties['peak_heights'][i]

            # To calculate the desired Q factor, the left and right frequencies of the bell
are needed.
            # Based on the formula from the source:
            # https://ecstudiosystems.com/discover/textbooks/basic-
electronics/filters/quality-factor-and-bandwidth/

            # Start and end of the 3dB bandwidth
            W1 = None
            W2 = None

            # Find W1 (left side of the peak)
            for j in range(peak, 0, -1):
                if signal[j] <= peak_gain - 3:
                    W1 = freqs[j]
                    break

            # Find W2 (right side of the peak)
            for j in range(peak, len(signal)):
                if signal[j] <= peak_gain - 3:
                    W2 = freqs[j]
                    break

            # Calculate the Q factor if W1 and W2 exist
            if ((W1 is not None) and (W2 is not None) and (W2 > W1)):
                Qo = freqs[peak] / (W2 - W1)
            else:
                continue

            bell = {
                'Wo': freqs[peak],
                'gain': peak_gain,
                'Qo': Qo,
                'left_index': W1,
                'right_index': W2
```

```
            }

            bells.append(bell)

        return bells

    # Source for finding peak point and range of a bell:
    # https://stackoverflow.com/questions/56810147/find-closest-minima-to-peak/

    # Find positive peaks (positive bells)
    positive_peaks, positive_properties = find_peaks(signal, height=threshold,
distance=min_distance, prominence=min_prominence)
    positive_bells = find_bells(positive_peaks, positive_properties, signal, freqs)

    # Find negative peaks (negative bells)
    negative_peaks, negative_properties = find_peaks(-signal, height=threshold,
distance=min_distance, prominence=min_prominence)
    negative_bells = find_bells(negative_peaks, negative_properties, -signal, freqs)  # Use -
signal to identify negative bells (as positive)

    # Combine positive and negative bells
    bells = positive_bells + negative_bells

    # Remove overlapping bells based on their gain
    filtered_bells = [] # filtered_bells contains the bells that will be kept
    for bell in bells:
        overlap_found = False

        for existing_bell in filtered_bells:
            # Check if the current bell overlaps with any existing bell
            if (bell['left_index'] <= existing_bell['right_index'] and bell['right_index'] >=
existing_bell['left_index']):
                overlap_found = True
                # If the current bell overlaps with an existing one, keep the one with the
bigger gain
                if abs(bell['gain']) >= abs(existing_bell['gain']):
                    filtered_bells.remove(existing_bell)
                    filtered_bells.append(bell)
                break

        # If the current bell doesn't overlap with any existing bell, add it to the filtered
list
        if not overlap_found:
            filtered_bells.append(bell)
```

```
    # Gain was calculated as positive for the negative bells (due to -signal), so replace with
negative gain
    for bell in negative_bells:
        bell['gain'] = -bell['gain']

    return filtered_bells

# Function to detect sloped curves in the desired filter response plot
def detect_slopes(signal, freqs, bells):
    slopes = []
    indices = [] # indices list holds the indices at which the shelf slope will be calculated

    # Identify the first (minf) and last (maxf) bell center frequencies
    if bells:
        minf = bells[0]['Wo']
        maxf = bells[0]['Wo']
        for bell in bells:
            if bell['Wo'] < minf:
                minf = bell['Wo']
            if bell['Wo'] > maxf:
                maxf = bell['Wo']

    indices.append(np.where(freqs >= 20)[0][0])
    indices.append(np.where(freqs == minf)[0][0])
    indices.append(np.where(freqs <= 20000)[0][-1])
    indices.append(np.where(freqs == maxf)[0][-1])

    # Calculate slopes for the regions: 20 Hz - minf and maxf - 20 kHz
    for i in range(2):
        slope = (signal[indices[2*i+1]] - signal[indices[2*i]]) / (freqs[indices[2*i+1]] -
freqs[indices[2*i]]) # Slope of a simple line
        S = np.cos(np.arctan(slope)) # Slope is the tangent of the angle of the line, so get
the angle and use its cosine as slope
        if i == 0: # For the first (starting) frequency range, boost or attenuate the left
(low) frequencies -> lowshelf
            type = 'lowshelf'
        else: # For the last (ending) frequency range, boost or attenuate the right (high)
frequencies -> highshelf
            type = 'highshelf'
        Wo = (freqs[indices[2*i+1]] + freqs[indices[2*i]]) / 2  # Geometric mean for center
frequency
        gain =  (signal[indices[2*i+1]] + signal[indices[2*i]]) / 2 # Average gain

        slopes.append({
            'type': type,
```

```
            'Wo': Wo,
            'S': S,
            'gain': gain
        })

    return slopes

# Function to apply required filters after detecting bells and slopes
def apply_filters(hp, fs, bells, slopes, message):
    filters = [] # filters list holds all the filters applied to the signal
    counter = 0 # Filter counter

    for bell in bells: # For each detected bell, apply a corresponding bell filter
        Wo = bell['Wo']
        Qo = bell['Qo']
        gain = bell['gain']
        sos = design_parametric([Wo], [Qo], [gain], fs, ['bell'])
        filters.append(sos)
        counter += 1
        print(f'bell filter at {Wo:.1f} Hz with Q = {Qo:.2f} and Gain = {gain:.2f}')

    for slope in slopes:
        Wo = slope['Wo']
        S = slope['S']
        gain = slope['gain']
        type = slope['type']
        sos = design_parametric([Wo], [S], [gain], fs, [type])
        filters.append(sos)
        counter += 1
        print(f'{type} filter at {Wo:.1f} Hz with S = {S:.2f} and Gain = {gain:.2f}')

    # Apply all filters and return the combined SOS
    combined_sos = np.vstack(filters)
    filtered_hp = sosfilt(combined_sos, hp)

    print(f'{message}')
    print(f'Total number of filters: {counter}')
    print('\n-----------------------------------\n\n')

    return combined_sos, filtered_hp

# Plot magnitude responses
for i in range(hpfilesnum):
    # Calculate frequency response for headphone
    hp_fft = np.fft.rfft(hp[i], len(hp[i]), norm='forward')  # norm='forward' for 1/n scaling
```

```
hp_mag = np.abs(hp_fft)/2 # Divide by 2 for further scaling
hp_freq = np.linspace(0, hp_fs[i]/2, len(hp_mag))
hp_mag_db = 20 * np.log10(hp_mag) - scaling_db

plt.figure(i + 1)

# Plot target responses in frequency domain
for j in range(targetsnum):
    # Calculate frequency response of target
    target_fft = np.fft.rfft(target[j], len(hp[i]), norm='forward')
    target_mag = np.abs(target_fft)/2
    target_freq = np.linspace(0, target_fs[j]/2, len(target_mag))
    target_mag_db = 20 * np.log10(target_mag) - scaling_db

    # Calculate initial MSE and RMSE
    mse = np.mean((hp_mag - target_mag) ** 2)
    rmse = np.sqrt(mse)
    # Source for mse and rmse formulas:
    # https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=1207422018

    # Plot headphone response vs target response
    plt.subplot(3, targetsnum, j + 1)
    plt.plot(hp_freq, hp_mag_db, color='b')
    plt.plot(target_freq, target_mag_db, color='r')
    plt.gca().set_xscale('log')
    plt.gca().xaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
    plt.gca().yaxis.set_major_formatter(ScalarFormatter())
    plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
    plt.xlim([20, 20000])
    plt.ylim([-80, 40])
    legend = 'Flat Target' if j == 0 else 'Harman Target'
    plt.title(f'Headphone Response {i + 1} vs {legend} Response')
    plt.legend(['Headphone Response', legend], loc='lower right')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude (dB)')
    plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
    plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
    plt.gca().get_ygridlines()[3].set_color('k')
    plt.gca().get_ygridlines()[3].set_linewidth(1)
    plt.gca().minorticks_on()
    plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
    plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)
    # Source for changing color and linewidth for specific gridlines:
```

```
        # https://stackoverflow.com/questions/32073616/matplotlib-change-color-of-individual-grid-lines
        # Source for making all ticks visible using ScalarFormatter():
        # https://www.reddit.com/r/learnpython/comments/epwteg/matplotlib_ticklabels_disappearing_in_log_scale/
        # Source for displaying text inside the plot using transAxes:
        # https://stackoverflow.com/questions/42932908/matplotlib-coordinates-tranformation

        # Calculate desired filter frequency response
        desired_mag = target_mag / hp_mag
        desired_mag_db = 20 * np.log10(desired_mag)
        desired_freq = target_freq

        # Detect bells and slopes
        bells = detect_bells(desired_mag_db, desired_freq, threshold=0.1, min_distance=10, min_prominence=0.01)
        slopes = detect_slopes(desired_mag_db, desired_freq, bells)

        # Apply filters and get overall response
        combined_sos, filtered_hp = apply_filters(hp[i], hp_fs[i], bells, slopes, f'Headphone Response {i+1} vs {legend}')

        # Calculate frequency response of filtered headphone
        filtered_hp_fft = np.fft.rfft(filtered_hp, len(filtered_hp), norm='forward')
        filtered_hp_mag = np.abs(filtered_hp_fft)/2
        filtered_hp_mag_db = 20 * np.log10(filtered_hp_mag) - scaling_db
        filtered_hp_freq = np.linspace(0, hp_fs[i] / 2, len(filtered_hp_mag_db))

        # Calculate frequency response of actual filter
        w, h = sosfreqz(combined_sos, worN=len(desired_mag))
        h_mag = np.abs(h)
        h_mag_db = 20 * np.log10(h_mag + 1e-12)
        h_freq = 0.5 * hp_fs[i] * w / np.pi

        # Calculate new MSE and RMSE
        mse = np.mean((filtered_hp_mag - target_mag) ** 2)
        rmse = np.sqrt(mse)

        # Plot desired filter response vs actual filter response
        plt.subplot(3, targetsnum, j + 3)
        plt.plot(desired_freq, desired_mag_db, color='g')
        plt.plot(h_freq, h_mag_db, color='tab:orange')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
```

```
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        plt.title('Desired Filter Response vs Actual Filter Response')
        plt.legend(['Desired Filter Response', 'Actual Filter Response'], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()

        # Plot filtered headphone response vs target response
        plt.subplot(3, targetsnum, 5 + j)
        plt.plot(filtered_hp_freq, filtered_hp_mag_db, color='b')
        plt.plot(target_freq, target_mag_db, color='r')
        plt.gca().set_xscale('log')
        plt.gca().xaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_xticks([20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000])
        plt.gca().yaxis.set_major_formatter(ScalarFormatter())
        plt.gca().set_yticks([-80, -50, -20, 0, 20, 40])
        plt.xlim([20, 20000])
        plt.ylim([-80, 40])
        legend = 'Flat Target' if j == 0 else 'Harman Target'
        plt.title(f'Filtered Headphone Response vs {legend} Response')
        plt.legend(['Filtered Headphone Response', legend], loc='lower right')
        plt.xlabel('Frequency (Hz)')
        plt.ylabel('Magnitude (dB)')
        plt.gca().grid(which='major', linestyle='--', linewidth=0.7, color='tab:brown')
        plt.gca().grid(which='minor', linestyle=':', linewidth=0.5, color='tab:gray')
        plt.gca().get_ygridlines()[3].set_color('k')
        plt.gca().get_ygridlines()[3].set_linewidth(1)
        plt.gca().minorticks_on()
        # Display new MSE and RMSE inside the plot
        plt.text(0.03, 0.35, f'MSE = {mse:.2f} dB^2', transform=plt.gca().transAxes)
        plt.text(0.03, 0.05, f'RMSE = {rmse:.2f} dB', transform=plt.gca().transAxes)

    # Adjust layout to prevent overlap
    plt.tight_layout()

# Keep all figures open
plt.show(block=True)
```