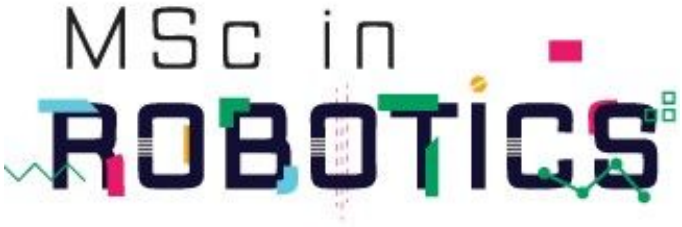


ΔΙΕΘΝΕΣ ΠΑΝΕΠΙΣΤΗΜΙΟ ΕΛΛΑΔΟΣ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΤΗΝ ΡΟΜΠΟΤΙΚΗ

Μελέτη και υλοποίηση σε λογισμικό αλγορίθμων υπολογιστικής τομογραφίας.

Study and software implementation of Computed Tomography algorithms

Μερτζεμεκιανός Μάρκος

Επιβλέπων καθηγητής:
Βουρβουλάκης Ιωάννης

Διπλωματική εργασία που υποβλήθηκε στο Πρόγραμμα Μεταπτυχιακών Σπουδών στη Ρομποτική, του Διεθνούς Πανεπιστημίου της Ελλάδος, για τη μερική εκπλήρωση υποχρεώσεων για το Δίπλωμα Ειδίκευσης στη Ρομποτική

14 Μαρτίου 2023

Περιεχόμενα

Κατάλογος σχημάτων

Περιγραφή

Η τομογραφία αποτελεί μια διαδικασία κατά την οποία δεδομένα προβολών ενός αντικειμένου μετατρέπονται σε εικόνες που απεικονίζουν τη διατομή του αντικειμένου αυτού. Η τομογραφία χρησιμοποιείται πολλά χρόνια στην ιατρική, καθώς δίνει τη δυνατότητα απεικόνισης μυϊκών τραυματισμών, αιμοφόρων αγγείων, λοιμώξεων και διάφορων άλλων ασθενειών, χωρίς να απαιτείται χειρουργική παρέμβαση εσωτερικά του ασθενούς. Η τομογραφική απεικόνιση προϋποθέτει την ύπαρξη πολλών προβολών, ώστε να οδηγήσει σε ένα ακριβές αποτέλεσμα. Οι αλγόριθμοι που χρησιμοποιούνται παρουσιάζουν μεγάλη πολυπλοκότητα και υψηλό υπολογιστικό φορτίο. Στα πλαίσια της παρούσας εργασίας θα αναπτυχθεί αλγόριθμος χρησιμοποιώντας αριθμούς σταθερής υποδιαστολής για την τομογραφική απεικόνιση. Θα μελετηθούν αλγόριθμοι που χρησιμοποιούνται στη βιβλιογραφία. Θα γίνει επιλογή ενός αλγορίθμου, όπως για παράδειγμα ο αλγόριθμος Filtered Back Projection.

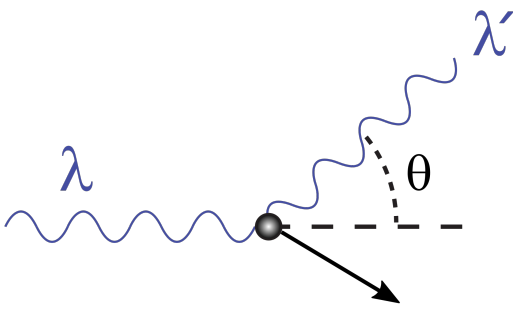
Κεφάλαιο 1

Εισαγωγή υπολογιστικής τομογραφίας σε ιατρικές εφαρμογές

1.1 Computerized tomography(CT) scans

Ο όρος υπολογιστική τομογραφία προέρχεται από τον όρο τομή, επομένως η τομογραφία, είναι η τεχνική που χρησιμοποιείται για την παρατήρηση τομών[1]. Για να επιτευχθεί η παρατήρησή τους, γίνεται χρήση ακτινών Χ, που είναι ικανές να διαπεράσουν το σώμα του ασθενούς. Κάθε ακτίνα μέσα στο σώμα μπορεί να κάνει δύο πράγματα.

Στην πρώτη περίπτωση, η ακτίνα σκεδάζεται μέσα στο σώμα και κάθε φορά που συγκρούεται με ένα ηλεκτρόνιο, του δίνει μέρος της ενέργειας της και φεύγει με διαφορετική κατεύθυνση. Το ηλεκτρόνιο από την άλλη, εκτινάσσεται από την εξωτερική του στοιβάδα. Στην δεύτερη περίπτωση η ακτίνα δεν σκεδάζεται, αλλά η ενέργεια της απορροφάται συνολικά από το άτομο. Το φαινόμενο αυτό λέγεται σκέδαση Compton.[2]



Σχήμα 1.1: Σκέδαση Compton. Η ακτίνα μεταφέρει την ενέργεια της στο ηλεκτρόνιο, όπου και αποσπάται από την στοιβάδα ενώ η ακτίνα Χ σκεδάζεται.[3]

Επισημαίνεται ότι αυτή η διαδικασία μπορεί να προκαλέσει ζημιά στο DNA, εάν η δόση της ακτινοβολίας είναι αρκετά μεγάλη. Η απώλεια ενέργειας κατά την κρούση, ακολουθεί τον

νόμο του Beer.

$$\frac{I_d}{I_0} = e^{-p} \Rightarrow \tag{1.1}$$

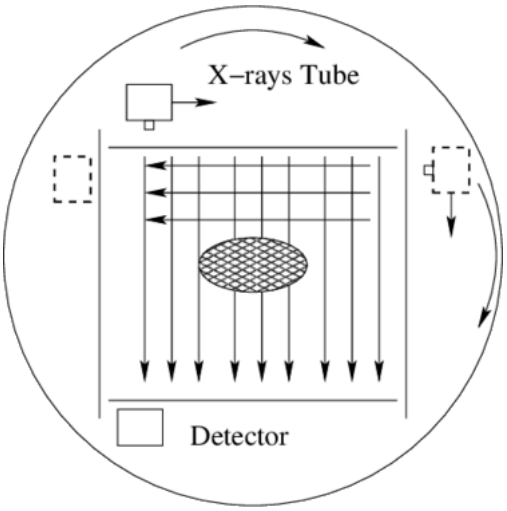
$$p = \ln \frac{I_0}{I_d} \tag{1.2}$$

Όπου I_0 είναι η ενέργεια εισόδου της ακτίνας, I_d είναι η ενέργεια κατά την αποχώρηση της και p η εξασθένηση της εκπεμπόμενης ισχύος ακτινοβολίας που προκαλείται από την απορρόφηση, την αντανάκλαση και την διασπορά [4]. Ο συγκεκριμένος όρος p χρησιμοποιείται για την ανακατασκευή εικόνας, όπου και συχνά στην βιβλιογραφία ανεφέρεται ως μ. Η τιμή του μ χρησιμοποιείται για την αντιστοίχιση του υλικού. Ενδεικτικά, παραθέτονται τιμές για ενέργεια εισόδου $E = 100keV$.

$\mu(\text{cm}^{-1})$	Υλικό
0.167	Νερό
$1.9 \cdot 10^{-4}$	αέρας
0.41	οστό

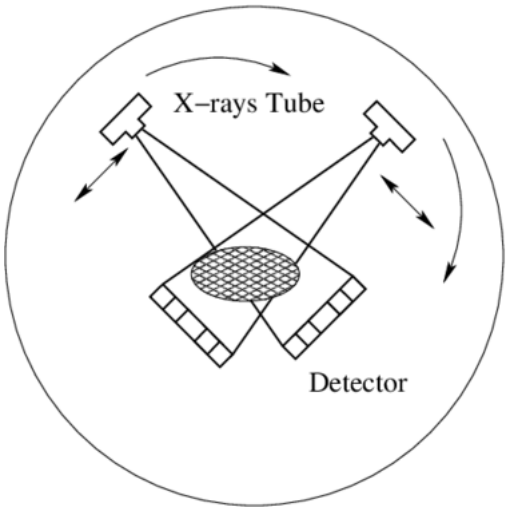
Γενιές αξονικού τομογράφου

Στην πρώτη γενιά αξονικού τομογράφου γινόταν χρήση ενός ανιχνευτή και μίας παράλληλης δέσμης δύο κινήσεων. Για να παραχθεί μία εικόνα, θα έπρεπε να μετατοπιστεί και να περιστραφεί πολλές φορές η παράλληλη δέσμη μαζί με τον ανιχνευτή. Αυτό είχε σαν αποτέλεσμα, η διαδικασία αυτή να πραγματοποιείται πολύ αργά και απαιτούνταν πολύς χρόνος για να παραχθεί μία εικόνα. Για τον λόγο αυτό, η πρώτη γενιά δεν χρησιμοποιήθηκε σε ασθενείς.



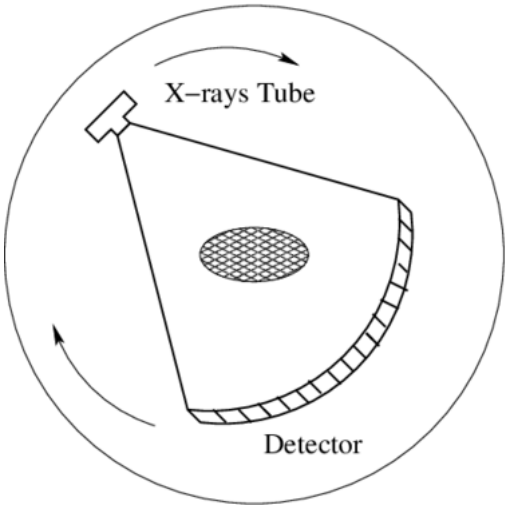
Σχήμα 1.2: Πρώτης γενιάς τομογράφος.[5]

Σε αντίθεση, στην δεύτερη γενιά χρησιμοποιήθηκε μία στενή αποκλίνουσα δέσμη, όπου πραγματοποιούνταν ο ίδιος αριθμός κινήσεων αλλά με μεγαλύτερο αριθμό ανιχνευτών. Λόγω της αποκλίνουσας δέσμης, η τομογραφία πραγματοποιούνταν σε σημαντικά μικρότερο χρόνο.



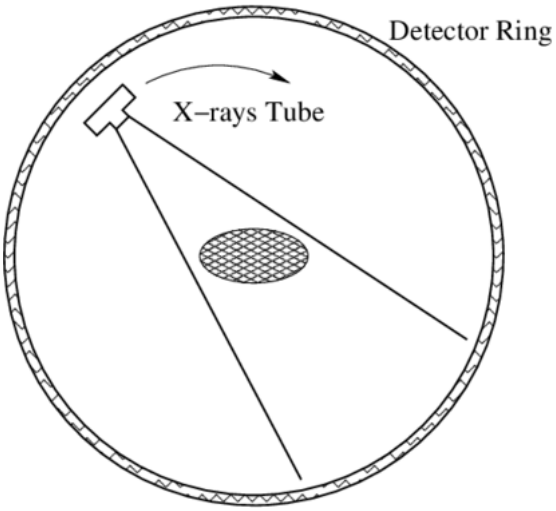
Σχήμα 1.3: Δεύτερης γενιάς τομογράφος.[5]

Στην τρίτη γενιά, η σημαντικότερη βελτίωση αφορούσε την διάρκεια της τομογραφίας. Σε αυτήν την περίπτωση, χρησιμοποιήθηκε ξανά μία αποκλίνουσα δέσμη με πολύ μεγαλύτερο εύρος. Λόγω του μεγαλύτερου εύρους, η μετατόπιση δεν χρειάζεται πλέον. Με αυτό τον τρόπο μειώθηκε η μηχανική πολυπλοκότητα, με αποτέλεσμα αυτή η διάταξη να χρησιμοποιείται και σήμερα.



Σχήμα 1.4: Τρίτης γενιάς τομογράφος.[5]

Όσον αφορά την τέταρτη και την πέμπτη γενιά αξονικών τομογράφων, μπορούν να μετακινηθούν μόνο οι δέσμες, ενώ οι ανιχτευτές παραμένουν ακίνητοι γύρω από τον ασθενή. Το κόστος κατασκευής τους είναι υψηλό λόγω του πολύ μεγαλύτερου αριθμού ανιχνευτών, χωρίς ωστόσο να προσφέρουν ιδιαίτερη βελτίωση στην διάρκεια της τομογραφίας.[6]



Σχήμα 1.5: Τέταρτης γενιάς τομογράφος.[5]

1.2 PET και SPECT

Οι τεχνικές PET (Positron emission tomography) και SPECT (Single Photon Emission Computed tomography) χρησιμοποιούν ακτινοβολία που προέρχεται από το ίδιο το σώμα του ασθενούς. Η απεικόνιση βασίζεται στην φυσιολογία του ασθενούς και των φυσιολογικών λειτουργιών των κυττάρων του σώματος του ασθενούς, με την χρήση ραδιοφαρμάκων. Αυτό τα κάνει "έξυπνα" και έχει ως αποτέλεσμα κάθε γιατρός να επιλέξει για την πάθηση θέλει που επιδιώκει να διαγνώσει, για παράδειγμα καρκίνος, πρώιμα στάδια άνοιας και άλλα. Οι δύο αυτές τεχνικές προσπαθούν να εντοπίσουν το ραδιενεργό υλικό. Σε κάθε περίπτωση τα ραδιοφάρμακα, έχουν σχετικά μικρή ημιζωή και αυτό τα καθιστά ασφαλή για τους ασθενείς.

Στην περίπτωση της τεχνικής SPECT χρησιμοποιούνται ραδιοϊσότοπα. Τα άτομα του ραδιοφαρμάκου ακτινοβολούν γ-ακτίνες, έως ότου το άτομο να σταθεροποιηθεί. Για να εντοπιστούν οι ακτίνες χρησιμοποιούνται κάμερες ακτινών γάμμα. Η γεωμετρία της διάταξης των καμερών παίζει διαδραματίζει σημαντικό ρόλο, καθώς καθορίζει την μορφή που θα έχουν οι εξερχόμενες από το σώμα ακτίνες (παράλληλη, αποκλίνουσα, κονική δέσμη) μέσω μίας διόπτρας. Κάποια από τα πλεονεκτήματα της τεχνικής αυτής, είναι ο ακριβέστερος εντοπισμός εστιών και χαρτογράφησης περιοχών με περίπλοκη ανατομία.

Στην τεχνική PET χρησιμοποιούνται ραδιοϊσότοπα τα οποία εκπέμπουν ποζιτρόνια κατά την ραδιενεργό διάσπαση τους. Γενικότερα, τα ποζιτρόνια στην φύση υπάρχουν για ένα πολύ βραχύ διάστημα. Αυτά συγκρούονται με τα ηλεκτρόνια και παράγουν ακριβώς δύο φωτόνια με ενέργεια 511 keV το κάθε ένα και 180 μοίρες γωνία μεταξύ τους. Αυτά το ζεύγος φωτονίων διαβάζονται από την κάμερα ταυτοχρόνως. Σε αυτήν την τεχνική δεν είναι εφικτό να έχουμε διόπτρες για να καθορίσουμε την μορφή της δέσμης, η οποία είναι πάντα παράλληλη. Μία από τις ουσίες που χορηγούνται είναι η ^{18}F-φθόριο-2-δεόξυ γλυκόζη. Η γλυκόζη μεταφέρεται εντός των κυττάρων όπου και μεταβολίζεται. Η υψηλή συγκέντρωση της ουσίας μπορεί να υποδηλώνει ότι βρίσκονται καρκινικά κύτταρα στην περιοχή αυτή.

Στην συνέχεια θα περιγραφτεί η μαθηματική μοντελοποίηση των προηγούμενων διαδικασιών.

Κεφάλαιο 2

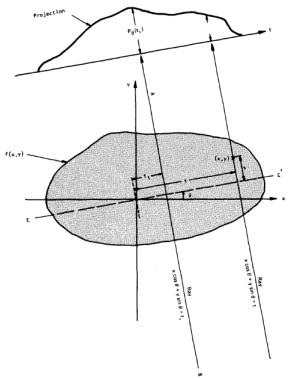
Μετασχηματισμός Radon

Ο μετασχηματισμός Radon είναι η διαδικασία που χρησιμοποιείται για την παραγωγή ενός ημιτονιοδιάγραμματος, το οποίο είναι μία εικόνα όπου κάθε γραμμή του ημιτονοδιαγράμματος αντιπροσωπεύει την γωνία της προβολής. Κάθε στήλη αντιπροσωπεύει την απόσταση της ευθείας από το πέρας της ακτίνας του κύκλου ανακατασκευής, ενώ η τιμή του κάθε στοιχείου της εικόνας αποτελεί το άθροισμα της τιμής των pixel που διατρέχει η κάθε ευθεία. Τέλος, το ημιτονοδιάγραμμα κανονικοποιείται προς την τιμή 255 για να μπορεί ώστε να προβληθεί σαν εικόνα. Έστω $f(x, y)$ οι τιμές της φωτεινότητας των pixels και $p(s, \theta)$ η τιμή της προβολής, όπου s η απόσταση της ευθείας από το πέρας της ακτίνας του κύκλου ανακατασκευής και θ η γωνία της ευθείας με το σύστημα συντεταγμένων. Τότε ισχύει

$$p(s,\theta) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\delta(x\cos\theta + y\sin\theta - s)\,\mathrm{d}x\,\mathrm{d}y \Rightarrow \tag{2.1}$$

$$p_i = \sum_{j=1}^{N} w_{ij} f_j \tag{2.2}$$

Το p_i είναι η τιμή της προβολής της i ακτίνας ενώ w_{ij} είναι το βάρος του pixel.

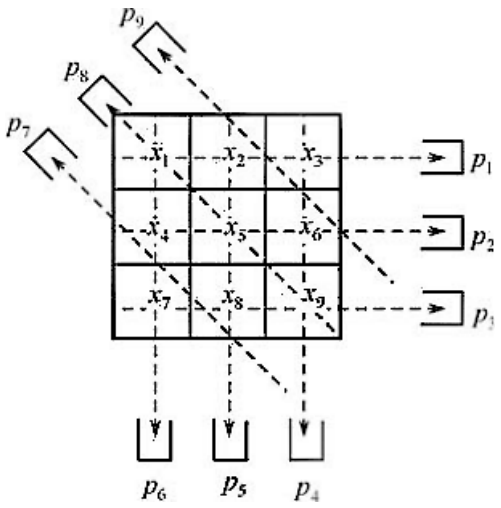


Σχήμα 2.1: Μετασχηματισμός radon. [1]

2.1 Μέθοδοι μοντελοποίησης μετασχηματισμού Radon

Υπάρχουν διάφοροι τρόποι για να μετρηθεί η συνεισφορά του κάθε pixel στην προβολή της κάθε ευθείας. Η επιλογή μεθόδου μοντελοποίησης, δεν επηρεάζει παρά ελάχιστα τα αποτελέσματα, όπως θα παρουσιαστούν και στην συνέχεια. Να σημειωθεί επίσης ότι, η εύρεση των τιμών του βάρους του κάθε pixel πρέπει να γίνει κατά της διαδικασίας της εύρεσης του μετασχηματισμού Radon. Καθώς οι ανάγκες αποθηκευτικού χώρου αυξάνονται εκθετικά με το μέγεθος της εικόνας εισόδου.

Η εύρεση των τιμών του βάρους για κάθε pixel μπορεί να μοντελοποιηθεί μέσω μίας διαδικασίας. Συγκεκριμένα, υπολογίζοντας για κάθε pixel (το οποίο θεωρείται σαν τετράγωνο με διαστάσεις 1 επί 1) το μήκος της ευθείας που βρίσκεται στο εσωτερικό του. Για παράδειγμα η εικόνα 2.2



Σχήμα 2.2: Εικόνα 3 επί 3 με 9 μετρήσεις. [2]

θα δώσει το σύστημα

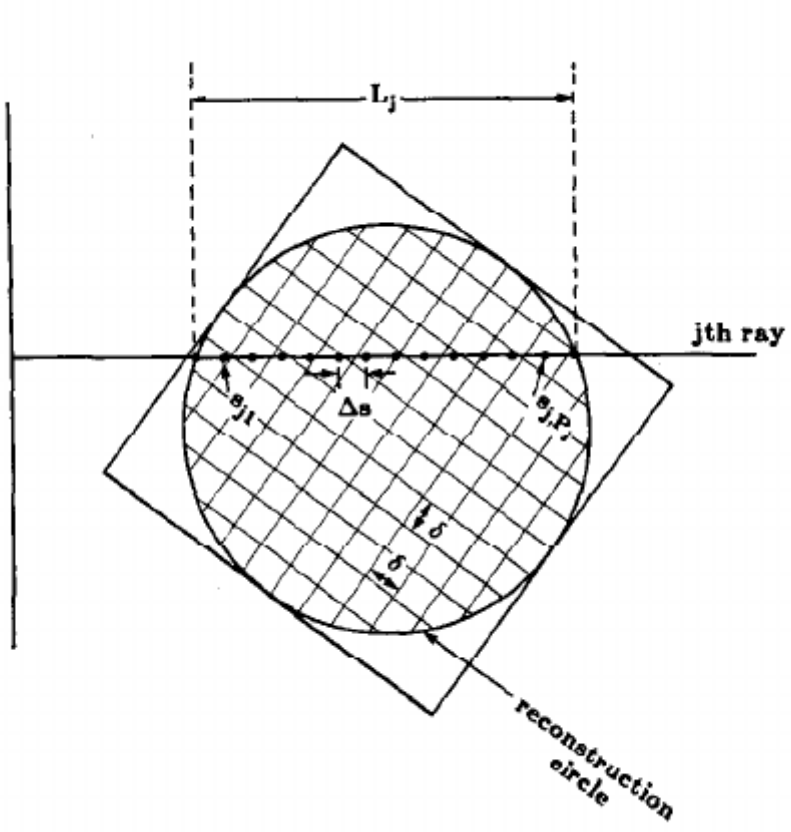
$$\begin{cases} x_1 + x_2 + x_3 = p_1 \\ x_4 + x_5 + x_6 = p_2 \\ x_7 + x_8 + x_9 = p_3 \\ x_3 + x_6 + x_9 = p_4 \\ x_2 + x_5 + x_8 = p_5 \\ x_1 + x_4 + x_7 = p_6 \\ 2(\sqrt{2}-1)x_4 + (2-\sqrt{2})x_7 + 2(\sqrt{2}-1)x_8 = p_7 \\ \sqrt{2}x_1 + \sqrt{2}x_5 + \sqrt{2}x_9 = p_8 \\ 2(\sqrt{2}-1)x_2 + (2-\sqrt{2})x_3 + 2(\sqrt{2}-1)x_6 = p_9 \end{cases} \tag{2.3}$$

Αυτή η μέθοδος είναι έχει μεγάλο υπολογιστικό φόρτο και δεν χρησιμοποιείται γενικότερα σε πραγματικές εφαρμογές. Μια άλλη προσέγγιση είναι να προστεθούν οι τιμές των pixel από τις οποίες διανύει μία ακτίνα. Σε αυτή τη μέθοδο, το βάρος του κάθε pixel είναι 1. Αυτή

η μέθοδος χρησιμοποιείται συχνά με την μέθοδο οπισθοπροβολής. Το σύστημα απλοποιείται στο 2.4.

$$\begin{cases} x_1 + x_2 + x_3 = p_1 \\ x_4 + x_5 + x_6 = p_2 \\ x_7 + x_8 + x_9 = p_3 \\ x_3 + x_6 + x_9 = p_4 \\ x_2 + x_5 + x_8 = p_5 \\ x_1 + x_4 + x_7 = p_6 \\ x_4 + x_7 + x_8 = p_7 \\ x_1 + x_5 + x_9 = p_8 \\ x_2 + x_3 + x_6 = p_9 \end{cases} \tag{2.4}$$

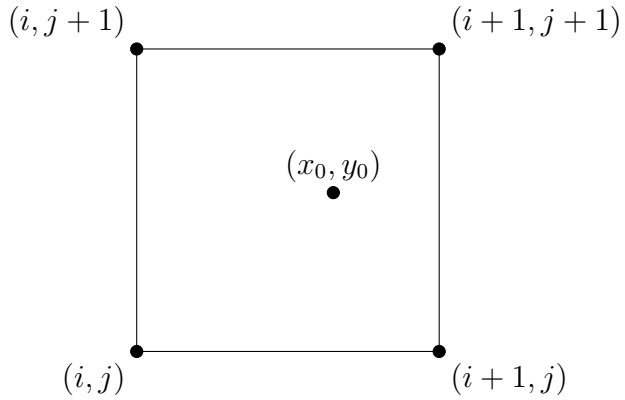
Μια άλλη προσέγγιση είναι να θεωρηθούν τα pixel σαν σημεία. Επίσης, θεωρείται ένας κύκλος ανακατασκευής. Ο κύκλος ανακατασκευής είναι η περιοχή μέσα στην οποία θα περιστραφούν οι ακτίνες. Μέσα στον κύκλο ανακατασκευής, πραγματοποιείται η δειγματοληψία, πάνω στις ακτίνες όπως απεικονίζεται στην εικόνα 2.3.



Σχήμα 2.3: Υπολογισμός του βάρους του κάθε pixel με τη χρήση δειγματοληψίας. [1]

Σε κάθε σημείο δειγματοληψίας μπορεί να χρησιμοποιηθεί είτε μια μέθοδος που επιλέγει τον κοντινότερο γείτονα ή αξιοποιώντας κάποια παρεμβολή με τα 4 pixel, που σχηματίζουν ένα

τετράγωνο γύρω από το σημείο. Με τον πρώτο τρόπο, δημιουργούνται σφάλματα λόγω ασυνέχειας, όμως κάνοντας χρήση παρεμβολής που σχηματίζεται από τις 4 κορυφές των pixels, επιτυγχάνεται η συνέχεια κατά τον μετασχηματισμό Radon (και της οπισθοπροβολής). Συγκεκριμένα, αν το σημείο έχει συντεταγμένες (x_0, y_0), τότε το βάρος του κάθε pixel θα είναι:



Σχήμα 2.4: Τετράγωνο όπου κάθε πλευρά του αποτελεί ένα pixel και (x_0, y_0) σημείο δειγματοληψίας.

$$w_{i,j} = ((i+1) - x_0) * ((j+1) - y_0) \tag{2.5}$$
$$w_{i+1,j} = (x_0 - i) * ((j+1) - y_0) \tag{2.6}$$
$$w_{i+1,j+1} = (x_0 - i) * (y_0 - i) \tag{2.7}$$
$$w_{i,j+1} = ((i+1) - x_0) * (j - y_0) \tag{2.8}$$
$$\tag{2.9}$$

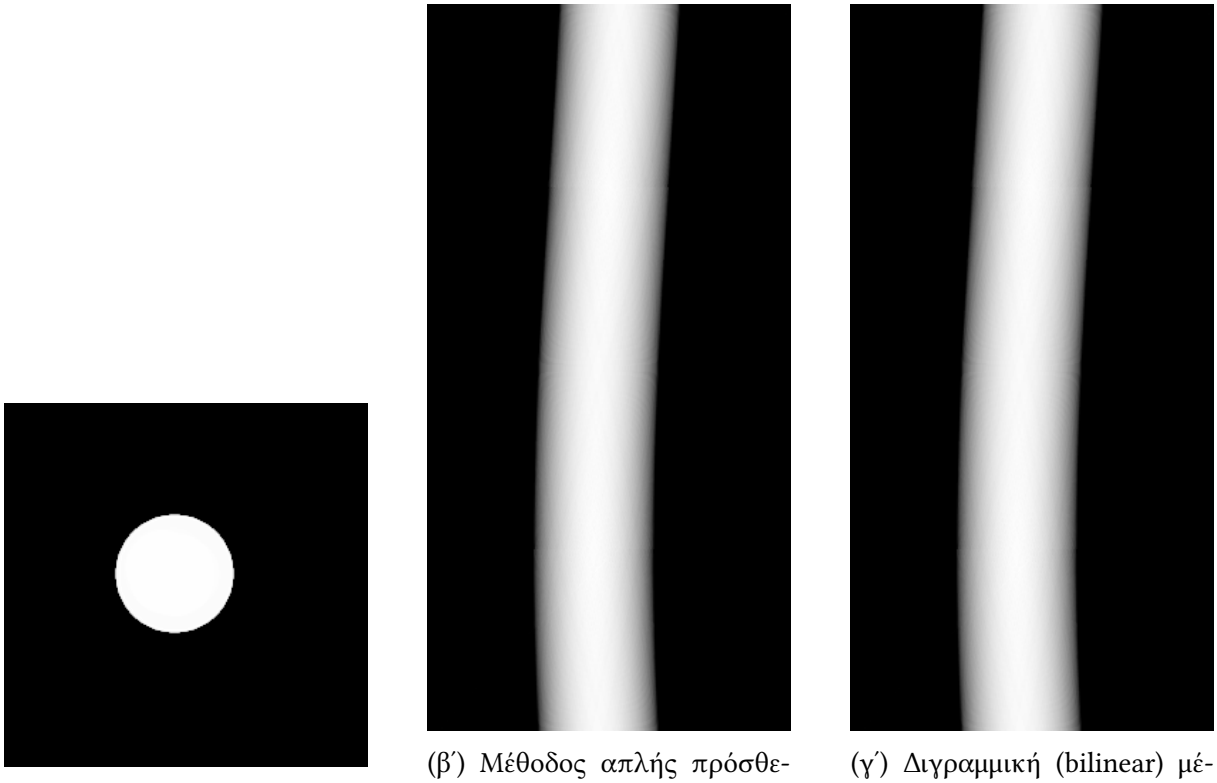
2.2 Παραδείγματα Μετασχηματισμού Radon

Στις εικόνες που παραθέτονται δίνονται κάποια παραδείγματα απλών σχημάτων, με στόχο της σύγκριση των εξής δύο μεθόδων. Την μέθοδο όπου γίνεται πρόσθεση στα pixel και την διγραμμική (bilinear) μέθοδο (σχέσεις 2.5-2.8).

Στην εικόνα 2.5 το αποτέλεσμα αποτελεί μία ευθεία γραμμή όπου το εύρος της είναι ίσο με την ακτίνα του κύκλου. Η μικρή καμπύλη της ευθείας πηγάζει από το γεγονός ότι, ο κύκλος δεν είναι κεντραρισμένος στην εικόνα.

Στην εικόνα 2.6 μπορεί να παρατηρηθεί πότε οι ακτίνες είναι ευθυγραμμισμένες με το σχήμα του τετραγώνου. Όσο πιο ευθυγραμμισμένες είναι οι ακτίνες με τις πλευρές του τετραγώνου, τόσο το χρώμα τείνει στη απόχρωση του άσπρου.

Το σχήμα Shepp-Logan[7] χρησιμοποιείται γενικά σαν σχήμα στους αλγορίθμους ανακατασκευής εικόνας. Εδώ παρατηρείται ότι το εύρος της ευθείας έχει μεγαλώσει στο κέντρο και αυτό οφείλεται στο γεγονός ότι το σχήμα που εικονίζεται έχει σχήμα έλλειψης.

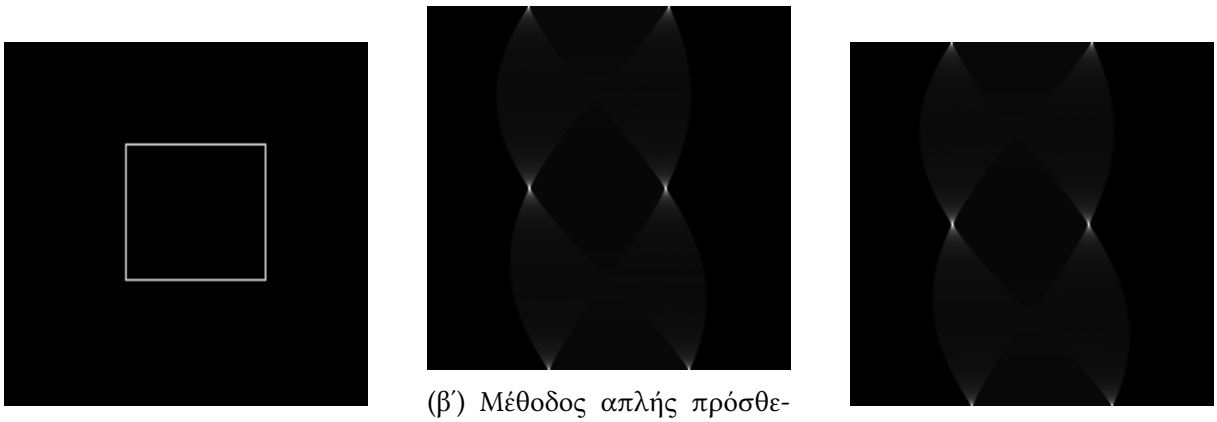


(α΄) Αρχική εικόνα.

(β΄) Μέθοδος απλής πρόσθεσης τιμών των pixel

(γ΄) Διγραμμική (bilinear) μέθοδος.

Σχήμα 2.5: Μετασχηματισμός Radon κύκλου



(α΄) Αρχική εικόνα.

(β΄) Μέθοδος απλής πρόσθεσης τιμών των pixel

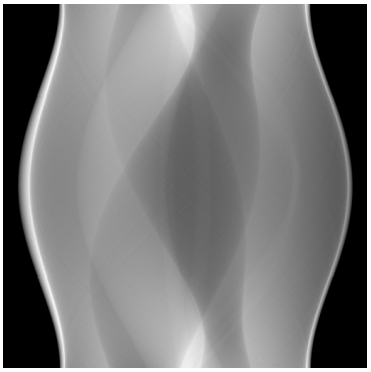
(γ΄) Διγραμμική μέθοδος.

Σχήμα 2.6: Μετασχηματισμός Radon κενού τετραγώνου

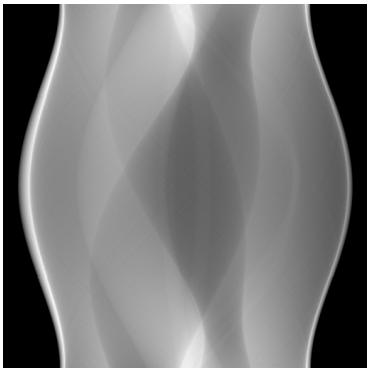
Παρατηρείται ότι, οι δύο μέθοδοι έχουν πολύ μικρές διαφορές. Γενικότερα, για επαναληπτικές μεθόδους ανακατασκευής εικόνας προτιμάται η διγραμμική μέθοδος, ενώ στην περίπτωση που χρησιμοποιείται η μέθοδος της οπισθοπροβολής, προτείνεται η απλή μέθοδος της πρόσθεσης των pixels.

Σε αυτό το κεφάλαιο περιγράφτηκε η διαδικασία εύρεσης του ημιτονοδιαγράμματος, στο επόμενο κεφάλαιο θα περιγραφτεί η αντίστροφη διαδικασία, δηλαδή η ανακατασκευή της εικόνας από το ημιτονοδιάγραμμα.

(α΄) Αρχική εικόνα.



(β΄) Μέθοδος απλής πρόσθεσης τιμών των pixel



(γ΄) Διγραμμική μέθοδος.

Σχήμα 2.7: Μετασχηματισμός Radon σχήματος Shepp Logan phantom.

Κεφάλαιο 3

Αλγόριθμοι ανακατασκευής εικόνας

Οι αλγόριθμοι ανακατασκευής εικόνας, αφορούν τους αλγορίθμους που με είσοδο το ημιτονοδιάγραμμα και ως έξοδο την εικόνα από την οποία προήλθε αυτό. Υπάρχουν διάφοροι τρόποι ανακατασκευής της εικόνας οι οποίοι αναλύονται στη συνέχεια.

3.1 Αλγεβρική προσέγγιση

Μία πρώτη προσέγγιση θα ήταν να λυθεί το σύστημα αντίστροφα από τον τρόπο που προήλθε το ημιτονοδιάγραμμα. Αν A είναι ο πίνακας που περιέχει τις τιμές του βάρους του κάθε pixel, X είναι ο πίνακας που περιέχει τις τιμές των pixel και P είναι το διάνυσμα που περιέχει τις τιμές των προβολών. Συνεπώς ισχύει:

$$AX = P \Leftrightarrow \tag{3.1}$$
$$X = A^{-1}P \tag{3.2}$$

Οι τιμές του πίνακα A επηρεάζονται από την μέθοδο μοντελοποίησης που έχει επιλεγεί. Το σύστημα από την παραπάνω εξίσωση ποτέ δεν εμφανίζει τις ίδιες εξισώσεις με αγνώστους και ανάλογα με την εφαρμογή, μπορεί να υπάρχουν λιγότερες (underdetermined) ή περισσότερες (overdetermined) εξισώσεις από αγνώστους. Αυτό οδηγεί τον πίνακα A να μην είναι τετραγωνικός. Για την επίλυση των συστημάτων χρησιμοποιείται ο ψευδοαντίστροφος πίνακας. Αν στον πίνακα $A^{m\times n}, m \geq n$ τότε χρησιμοποιείται ο αριστερός αντίστροφος πίνακας, ενώ στην αντίθετη περίπτωση χρησιμοποιείται ο δεξιός αντίστροφος πίνακας. Εδώ να σημειωθεί, ότι για την εύρεση του τύπου του αριστερού αντίστροφου πίνακα, χρησιμοποιείται η μέθοδος βελτιστοποίησης των ελαχίστων τετραγώνων. Ενώ για τον δεξιό χρησιμοποιείται μέθοδος της ελαχιστοποίησης της νόρμας του πίνακα . Συνεπώς η λύση δίνεται με την παρακάτω εξίσωση

$$X = \begin{cases} (A^TA)^{-1}A^TP, \text{ αν το σύστημα έχει περισσότερες εξισώσεις από αγνώστους} \\ A^T(AA^T)^{-1}P, \text{ αν το σύστημα έχει λιγότερες εξισώσεις από αγνώστους} \end{cases} \tag{3.3}$$

Η αλγεβρική προσέγγιση εμφανίζει το μειονέκτημα ότι, ο πίνακας A, τις περισσότερες φορές, είναι πολύ μεγάλος. Αυτό το κάνει αδύνατο να αποθηκευτεί, σε συσκευές με περιορισμένο αποθηκευτικό χώρο.

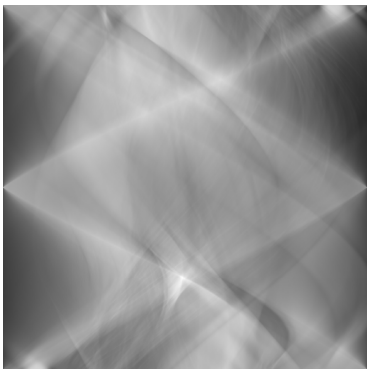
3.2 Μέθοδος οπισθοπροβολής

Η μέθοδος οπισθοπροβολής, είναι η διαδικασία κατά την οποία το ημιτονοδιάγραμμα μετατρέπεται στην αρχική εικόνα. Η διαδικασία αυτή υλοποιείται βάζοντας την τιμή της προβολής στα pixels όπου διέσχισε η ακτίνα. Έστω το σύστημα που απεικονίζεται στην εικόνα 2.2. Τότε μετά την οπισθοπροβολή, το κάθε pixel θα έχει τιμή:

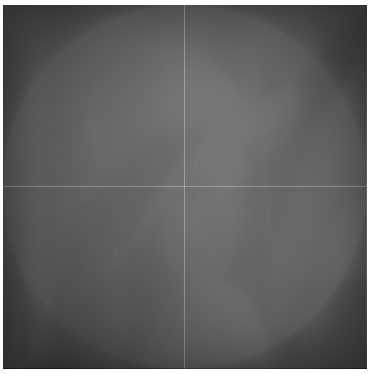
$$\begin{cases} x_1 = p_1 + p_6 + p_8 \\ x_2 = p_1 + p_5 + p_9 \\ x_3 = p_1 + p_4 + p_9 \\ x_4 = p_2 + p_6 + p_7 \\ x_5 = p_2 + p_5 + p_8 \\ x_6 = p_2 + p_4 + p_9 \\ x_7 = p_3 + p_6 + p_7 \\ x_8 = p_3 + p_5 + p_7 \\ x_9 = p_3 + p_4 + p_8 \end{cases} \tag{3.4}$$

Η αρχική εικόνα (α') ανακατασκευάζεται με την μέθοδο της οπισθοπροβολής χωρίς όμως το αποτέλεσμα να είναι ισοδύναμο με την αρχική εικόνα. Γενικότερα η εικόνα εμφανίζεται θολή, επομένως είναι δύσκολο να διακριθεί η αρχική. Παράδειγμα δίνεται στην εικόνα 3.1.

(α΄) Αρχική εικόνα.



(β΄) Ημιτονοδιάγραμμα.



(γ΄) Οπισθοπροβολή.

Σχήμα 3.1: Αποτέλεσμα διαδικασίας οπισθοπροβολής

Αυτό το αποτέλεσμα οφείλεται στο γεγονός ότι η εικόνα στο κέντρο του κύκλου ανακατασκευής, έχει γίνει δειγματοληψία πιο πυκνά στο κέντρο παρά στα άκρα. Αυτό αποδεικνύεται και μαθηματικά με την χρήση του μετασχηματισμού Fourier.

3.3 Μετασχηματισμός Fourier

Ο μετασχηματισμός Fourier $P(\omega)$ ορίζεται για μία συνάρτηση $p(s)$ ως:

$$P(\omega) = \int_{-\infty}^{\infty} p(s) e^{-2\pi i s\omega}\, ds \tag{3.5}$$

, όπου $\sqrt{-1} = i$

Οι συναρτήσεις $P(\omega)$ και $p(s)$ είναι ίδιες συναρτήσεις, αλλά από διαφορετική σκοπιά. Η μία είναι στο πεδίο συχνότητας ενώ η άλλη στο πεδίο του χρόνου. Ο περιορισμός που εμφανίζεται μεταξύ τους είναι ότι, το πεδίο του χρόνου δεν περιέχει πληροφορίες για το πεδίο συχνοτήτων και το αντίστροφο. Επομένως, ο μετασχηματισμός Fourier μας δίνει την δυνατότητα της διαχείρισης ενός σήματος από την σκοπιά του πεδίου της συχνότητας. Η διακριτή μορφή του μετασχηματισμού Fourier γράφεται ως:

$$P(n) = \sum_{0}^{N-1} p_k e^{-2\pi i k n/N} \tag{3.6}$$

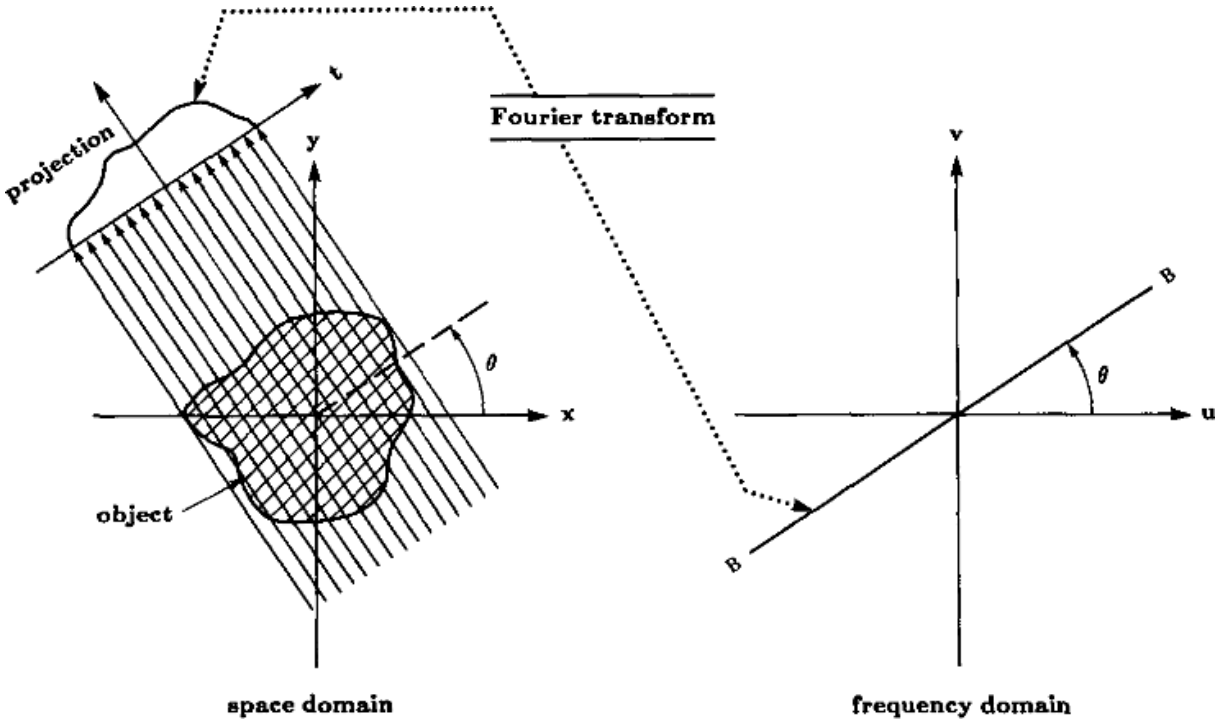
Ο προηγούμενος τύπος έχει πολυπλοκότητα $O(N^2)$. Για να υπολογιστεί ο μετασχηματισμός Fourier σε $O(N\log(N))$ χρόνο θα πρέπει να χρησιμοποιηθεί ο γρήγορος μετασχηματισμός Fourier. Η ταχύτητα του γρήγορου μετασχηματισμού Fourier οφείλεται στην παρατήρηση ότι αποτελείται από μικρότερους μετασχηματισμούς, χρησιμοποιώντας τους άρτιους και περιττούς όρους. Αυτό παρουσιάζεται παρακάτω:

$$\begin{aligned}
P_k &= \sum_{j=0}^{N-1} e^{2\pi i j k/N} * p_j && (3.7)\\
&= \sum_{j=0}^{N/2-1} e^{2\pi i k(2j)/N} p_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i k(2j+1)/N} p_{2j+1} && (3.8)\\
&= \sum_{j=0}^{N/2-1} e^{2\pi i k(j)/(N/2)} p_{2j} + W_k \sum_{j=0}^{N/2-1} e^{2\pi i k j/(N/2)} p_{2j+1} && (3.9)\\
&= P_k^e + W_k P_k^o && (3.10)
\end{aligned}$$

, όπου $W = e^{2\pi i/N}$

3.4 Κεντρικό θεώρημα τομής

Στο κεντρικό θεώρημα τομής ο μετασχηματισμός Fourier της προβολής της συνάρτηση $f(x, y)$ ισοδυναμεί με την ευθεία που διέρχεται από το κέντρο του δισδιάστατου μετασχηματισμού Fourier της συνάρτησης $f(x, y)$ και είναι παράλληλη προς τον δέκτη από τον οποίο προέκυψε αυτή3.2. Αυτή η ιδιότητα είναι πάρα πολύ σημαντική, διότι επιτρέπει ένα δισδιάστατο



Σχήμα 3.2: Κεντρικό θεώρημα τομής.[1]

μετασχηματισμό Fourier να μελετηθεί σαν πολλούς μονοδιάστατους. Έστω $p(s,\theta)$ ο μετασχηματισμός Radon της εικόνας και $P(\omega,\theta)$ ο μετασχηματισμός Fourier του $p(s,\theta)$ με $f(x,y)$ και $F(\omega_x,\omega_y)$ ο μετασχηματισμός της.

$$P(\omega) = \int_{-\infty}^{\infty} p(s)e^{-2\pi is\omega}\,ds \tag{3.11}$$

$$\tag{3.12}$$

Από τον ορισμό του μετασχηματισμού Radon, η εξίσωση προκύπτει

$$\begin{aligned} P(\omega,\theta) &= \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\delta(x\cos\theta + y\sin\theta - s)\,dx\,dy\right] e^{-2\pi is\omega}\,ds && (3.13)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty}\delta(x\cos\theta + y\sin\theta - s)e^{-2\pi is\omega}\,ds\right] dx\,dy && (3.14)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-2\pi i(x\cos\theta + y\sin\theta)\omega}\,dx\,dy && (3.15)\\ &= F(\omega_x,\omega_y)|_{\omega_x=\omega\cos\theta,\ \omega_y=\omega\sin\theta} && (3.16) \end{aligned}$$

Από το θεώρημα κεντρικής τομής μπορεί να υπολογιστεί η συνάρτηση που πρέπει να φιλτραριστεί ο μετασχηματισμός Radon ώστε να προσεγγιστεί η αρχική εικόνα. Ο δισδιάστατος

αντίστροφος μετασχηματισμός Fourier παρουσιάζεται παρακάτω

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(\omega_x,\omega_y)e^{2\pi i(x\omega_x+y\omega_y)}\,d\omega_x d\omega_y \tag{3.17}$$

$$= \int_0^{2\pi}\int_0^{\infty} F_{\text{polar}}(\omega,\theta)\begin{vmatrix}\frac{\partial\omega_x}{\partial\omega} & \frac{\partial\omega_x}{\partial\theta}\\ \frac{\partial\omega_y}{\partial\omega} & \frac{\partial\omega_y}{\partial\theta}\end{vmatrix} d\omega d\theta \tag{3.18}$$

$$= \int_0^{2\pi}\int_0^{\infty} F_{\text{polar}}(\omega,\theta)\omega e^{2\pi i(x\omega_x+y\omega_y)}\,d\omega d\theta \tag{3.19}$$

Στον χώρο συχνοτήτων ισχύει ότι $F_{\text{polar}}(\omega,\theta) = F_{\text{polar}}(-\omega,\theta+\pi)$

$$f(x,y) = \int_0^{\pi}\int_{-\infty}^{\infty} F_{\text{polar}}(\omega,\theta) * |\omega|\,d\omega d\theta \tag{3.20}$$

Λόγω του κεντρικού θεωρήματος τομής λαμβάνουμε:

$$f(x,y) = \int_0^{\pi}\int_{-\infty}^{\infty} P(\omega,\theta) * |\omega| e^{2\pi i(x\omega_x+y\omega_y)}\,d\omega d\theta \tag{3.21}$$

Επομένως, προκύπτει ότι για να υπολογισθεί η αρχική εικόνα θα πρέπει η μετασχηματισμένη εικόνα να πολλαπλασιαστεί με τον όρο $|\omega|$. Έστω $Q(\omega,\theta) = |\omega|P(\omega,\theta)$ τότε:

$$f(x,y) = \int_0^{\pi}\int_{-\infty}^{\infty} Q(\omega,\theta)e^{2\pi i\omega(x\cos\theta+y\sin\theta)}\,d\omega d\theta \tag{3.22}$$

Η εξίσωση ευθείας έχει τον ακόλουθο τύπο $x\cos\theta + y\sin\theta = s$, άρα αν χρησιμοποιηθεί και ο αντίστροφος μετασχηματισμός Fourier, τότε

$$f(x,y) = \int_0^{\pi} q(s,\theta)\,d\theta \tag{3.23}$$

Σύμφωνα με όσα αναλύθηκαν παραπάνω κρίνεται απαραίτητη η ανάγκη φιλτραρίσματος της εικόνας με τον όρο $|\omega|$.

3.5 Οπισθοπροβολή με φιλτράρισμα

Όσον αφορά τη στρατηγική φιλτραρίσματος εικόνας με οπισθοπροβολή παρουσιάζονται τα βήματα για την εφαρμογή της μεθόδου.

1. Ως είσοδος χρησιμοποιούνται οι τιμές της εικόνας του ημιτονιοδιαγράμματος που αντιστοιχούν σε μία γωνία θ.
2. Χρησιμοποιείται μετασχηματισμός Fourier πάνω σε αυτές τις τιμές.
3. Πραγματοποιείται φιλτράρισμα του μετασχηματισμένου διανύσματος με τον όρο $|\omega|$.

4. Εκτέλεση αντίστροφου μετασχηματισμού Fourier.

5. Χρησιμοποίηση αλγορίθμου οπισθοπροβολής για να μεταφερθούν οι τιμές στα pixel της εικόνας.

Στο σημείο αυτό, διαπιστώνεται πως ο αλγόριθμος έχει την δυνατότητα να παραλληλοποιηθεί. Άλλες τεχνικές περιλαμβάνουν την υλοποίηση του φιλτραρίσματος στο πεδίο του χώρου, ή την χρησιμοποίηση του μετασχηματισμού Hilbert για το φιλτράρισμα του.

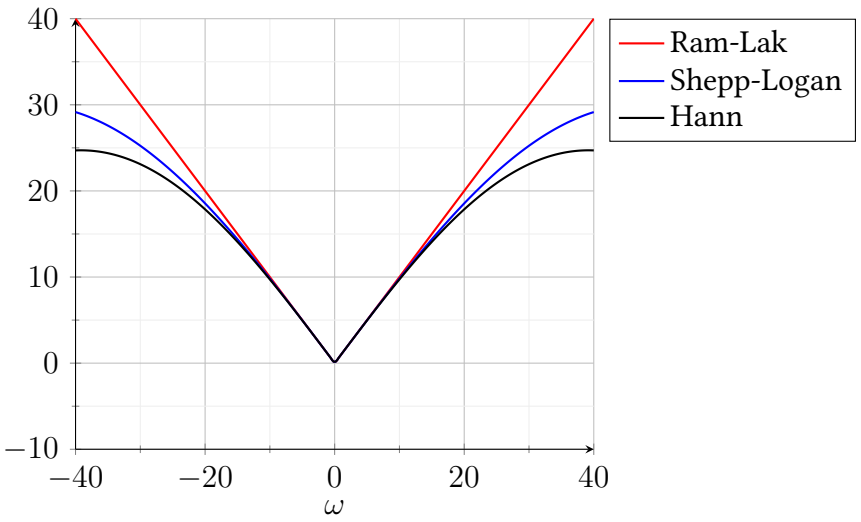
3.5.1 Είδη φίλτρων

Στην οπισθοπροβολή με φιλτράρισμα γίνεται χρήση διαφόρων φίλτρων. Τα φίλτρα εκτός από υψιπερατά φίλτρα, βρίσκονται και κάποια χαμηλοπερατά τα οποία βοηθούν στην εξάλειψη των ατελειών λόγω της οπισθοπροβολής. Μερικά από αυτά είναι:

Φίλτρο Ram-Lak $$H(\omega) = \begin{cases} |\omega|, \text{αν } |\omega| < W \\ 0, \text{αλλού} \end{cases}$$

Φίλτρο Shepp-Logan $$H(\omega) = \begin{cases} |\omega|\text{sinc}(\omega/W), \text{ αν } |\omega| < W \\ 0, \text{αλλού} \end{cases}$$

Φίλτρο Hann $$H(\omega) = \begin{cases} |\omega|(.5 + .5cos(\frac{\pi\omega}{W})), \text{ αν } |\omega| < W \\ 0, \text{ αλλού} \end{cases}$$

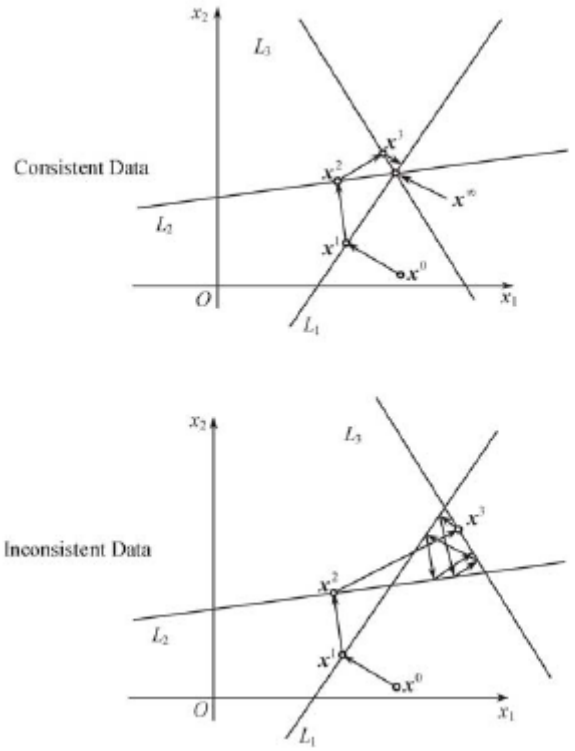


3.6 Επαναληπτικές Μέθοδοι

Στις επαναληπτικές μεθόδους η ανάκτηση της εικόνας γίνεται χρησιμοποιώντας τεχνικές βελτιστοποίησης. Μετά από κάθε επανάληψη γίνεται σύγκριση της προβολής του αποτελέσματος με το ημιτονιοδιάγραμμα που ο αλγόριθμος είχε σαν είσοδο. Μερικές από αυτές τις τεχνικές παρουσιάζονται παρακάτω.

3.6.1 ART

Η τεχνική ART (αλγεβρικής ανακατασκευής εικόνας, algebraic recontruction technique) βρίσκει τις λύσεις μίας εξίσωσης ανά προβολή. Ποιο συγκεκριμένα επιδιώκει να εντοπίσει το μοναδικό κοινό σημείο τομής όλων των υπερεπιπέδων που κατασκευάζουν οι εξισώσεις. Σε περίπτωση που υπάρχει θόρυβος στην εικόνα, δεν υπάρχει κοινό σημείο που να ικανοποιεί όλες τις εξισώσεις. Κάθε φορά που λύνεται μία εξίσωση ουσιαστικά αποτελεί τη προβολή του σημείου στην ευθεία. Παράδειγμα του συγκεκριμένου εμφανίζεται στην εικόνα 3.3.



Σχήμα 3.3: Εύρεση λύσεων στον αλγόριθμο ART.[2]

Στο σχήμα 3.3 γίνεται επίλυση μίας εικόνας με 2 pixel. Η επίλυση ξεκινάει με την αρχική εκτίμηση της εικόνας από την οπισθοπροβολή. Σε αυτήν την περίπτωση, χρησιμοποιούνται 3 εξισώσεις για την εύρεση των 2 αγνώστων. Όπως παρουσιάζεται στην εικόνα πραγματοποιώντας συνεχόμενες προβολές πάνω στις ευθείες, μετά από κάποιο αριθμό επαναλήψεων μπορεί να αποκτηθεί ένα αρκετά ικανοποιητικό αποτέλεσμα. Στην περίπτωση που η εικόνα έχει θόρυβο, τότε δεν μπορεί να βρεθεί ένα κοινό σημείο και η μέθοδος δεν είναι εφικτό να συγκλίνει. Ο αλγόριθμος μπορεί να εκφραστεί από την σχέση

$$x^{\text{next}} = x^{\text{current}} - \text{BP}_{ray}\left(\frac{\text{Project}_{\text{ray}} - \text{Measurement}}{\text{Normalization Factor}}\right) \tag{3.24}$$

Υπάρχουν αρκετές παραλλαγές του αλγορίθμου και κυρίως πραγματεύονται τον τρόπο που ανανεώνονται οι τιμές στην εικόνα προς εύρεση. Για παράδειγμα, η ανανέωση των τιμών μπορεί να γίνει ανά γωνία ή μετά από μετασχηματισμό όλης της εικόνας (SIRT[1]).

Στο επόμενο κεφάλαιο θα χρησιμοποιηθούν οι παραπάνω τεχνικές για να υλοποιηθεί μία πειραματική εφαρμογή.

Κεφάλαιο 4

Λογισμικό υπολογιστή

Μέσω της χρήσης του λογισμικού υπολογιστή αξιοποιήθηκαν, οι προηγούμενες τεχνικές για την υλοποίηση του μετασχηματισμού Radon και της διαδικασία της φιλτραρισμένης οπισθοπροβολής. Σε αυτό το κεφάλαιο θα συζητηθούν οι προσεγγίσεις που έγιναν για την μοντελοποίηση της διαδικασίας. Όπως επίσης, και τις τεχνικές που επιλέχθηκαν για μοντελοποίηση, αλλά και τη δομή του κώδικα και τα αποτελέσματά του.

Για την υλοποίηση του λογισμικού χρησιμοποιήθηκε η γλώσσα προγραμματισμού C++. Η C++ είναι μία αντικειμενοστρεφής γλώσσα προγραμματισμού που απευθύνεται, τόσο σε υψηλού επιπέδου προγραμματιστές, όσο και σε χαμηλού. Παρότι η γλώσσα είναι αντικειμενοστρεφής η χρήση κλάσεων είναι προαιρετική. Για την είσοδο των εικόνων, την έξοδο τους, καθώς και για κάποιους μετασχηματισμούς των εικόνων χρησιμοποιήθηκε η βιβλιοθήκη OpenCV. Επίσης, για την είσοδο δεδομένων μέσω του τερματικού χρησιμοποιήθηκε η βιβλιοθήκη boost.

4.1 Αρχική υλοποίηση

4.1.1 Μετασχηματισμός Radon με περιστροφή

Για να πραγματοποιηθεί η προβολή της εικόνας και να παραχθεί το ημιτονοδιάγραμμα με αυτήν την συγκεκριμένη προσέγγιση, γίνεται περιστροφή της εικόνας κατά θ μοίρες. Έπειτα αθροίζονται τα pixels σε κάθε στήλη και έτσι υπολογίζονται οι τιμές για γωνίες από 0 έως και 180 μοίρες. Το μέγεθος του ημιτονοδιαγράμματος εξαρτάται από το μέγεθος της εικόνας, όπως επίσης και ο αριθμός των γωνιών. Παράδειγμα της εφαρμογής παρουσιάζεται στην εικόνα 4.1.

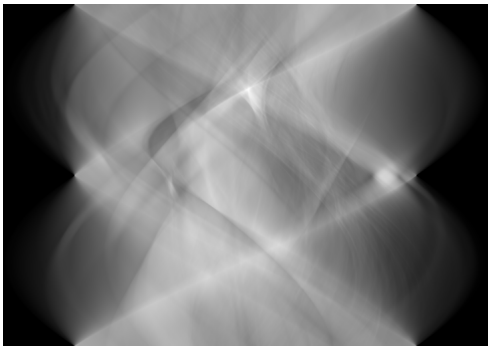
Περιγραφή υλοποίησης

Στην συνέχεια περιγράφεται το πρόγραμμα του μετασχηματισμού Radon με περιστροφή. Ξεκινώντας το πρόγραμμα πραγματοποιείται η εισαγωγή του ονόματος της εικόνας και του αριθμού των γωνιών σαρώσεων(χρησιμοποίειται η Βιβλιοθήκη Boost). Έπειτα, χρησιμοποιώντας την βιβλιοθήκη OpenCV μπορεί να διαβαστεί η εικόνα. Η εισαγωγή της εικόνας υλοποιείται με την συνάρτηση imread. Με την συνάρτηση imread μπορούν να διαβαστούν

Σχήμα 4.1: Μετασχηματισμός Radon με περιστροφή εικόνας.

διάφορα format εικόνων. Αποτέλεσμα της συνάρτησης είναι ένα αντικείμενο τύπου Mat. Το αντικείμενο Mat περιέχει τον δισδιάστατο πίνακα τιμών της εικόνας. Η εικόνα επεκτείνεται στην επόμενη μεγαλύτερη δύναμη του 2 σύμφωνα με το μέγεθος της διαγωνίου, και αυτό επιτυγχάνεται με την εντολή copyMakeBorders. Η εικόνα επεκτείνεται στην επόμενη 2 διότι βοηθάει στα επόμενα βήματα και ειδικότερα για την οπισθοπροβολή. Εφόσον ολοκληρωθούν τα προηγούμενα βήματα που αποτελούν την προετοιμασία των δεδομένων, ακολουθεί η κύρια επανάληψη. Στην κύρια επανάληψη υπολογίζεται η καινούργια περιστροφή της εικόνας και προστίθενται οι τιμές των στηλών. Έτσι εκτελώντας μία επανάληψη, αποκτάται ο μετασχηματισμός Radon για μία δεδομένη γωνία. Αφού ολοκληρωθούν όλες οι επαναλήψεις, γίνεται μία κανονικοποίηση στο ημιτονοδιάγραμμα ώστε να μπορέσει να προβληθεί και να αποθηκευτεί. Αποτελέσματα αυτής της μεθόδου, φαίνονται παρακάτω. Στην συγκεκριμένη υλοποίηση του αλγορίθμου, ο κύκλος ανακατασκευής βρίσκεται περιέχει την εικόνα. Συνεπώς, όλη η πληροφορία της εικόνας περιέχεται στο ημιτονοδιάγραμμα.

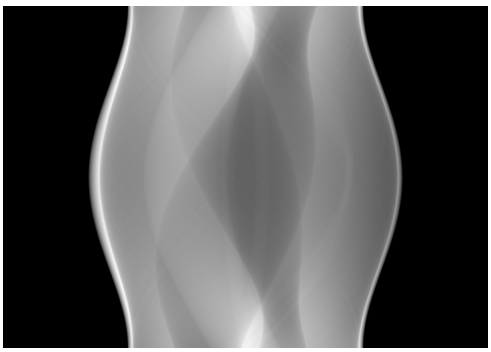
(α΄) Αρχική εικόνα



(β΄) Μετασχηματισμός Radon

Σχήμα 4.2: Μετασχηματισμός Radon της εικόνας Lena με την μέθοδο περιστροφής της εικόνας με 512 ακτίνες.

(α΄) Αρχική εικόνα



(β΄) Μετασχηματισμός Radon

Σχήμα 4.3: Μετασχηματισμός Radon της εικόνας Shepp-Logan με μέθοδο της περιστροφής της εικόνας με 512 ακτίνες.

4.1.2 Οπισθοπροβολή με επανάληψη στα pixel

Σε αυτήν την περίπτωση η οπισθοπροβολή γίνεται από τα pixel. Το πρόγραμμα δέχεται σαν είσοδο το ημιτονοδιάγραμμα και ως έξοδο την ανακατασκευασμένη εικόνα. Ξεκινώντας το πρόγραμμα διαβάζει το όνομα του ημιτονοδιαγράμματος-εικόνας το οποίο έχει δοθεί από τον χρήστη. Επεκτείνει το ημιτονοδιάγραμμα ώστε ο αριθμός των στηλών της εικόνας να είναι ίσος με την επόμενη δύναμη του 2. Η κύρια επανάληψη διατρέχει τις γωνίες σύμφωνα το υπολογισμένο βήμα. Το βήμα της γωνίας βρίσκεται από τις γραμμές της εικόνας, όπου ισοδυναμούν με τον αριθμό των γωνιών και πιο συγκεκριμένα βρίσκεται από την διαίρεση $\frac{\pi}{\text{num\_angles}}$. Για κάθε γραμμή της εικόνας χρησιμοποιείται το θεώρημα κεντρικής τιμής και φιλτράρεται. Στο σημείο αυτό επισημαίνεται, πως η φιλτραρισμένη γραμμή είναι ένα διάνυσμα από τιμες fp_i. Εφόσον φιλτραριστεί η γραμμή, προσπελάσσεται κάθε ζευγάρι (x, y) pixel και υπολογίζεται η τιμή i μέσω της εξίσωσης της ευθείας.

$$s = -(x - \text{side}/2)\cos(\theta) - (y - \text{side}/2)\sin(\theta) + \text{side}/2 \tag{4.1}$$

$$i = [s] \tag{4.2}$$

Αν για κάποιο ζευγάρι (x, y) και κάποια γωνία θ βρεθεί ότι η τιμή i είναι έξω από το διάνυσμα των φιλτραρισμένων προβολών, η τιμή απορρίπτεται. Αυτό συνεχίζεται για όλες τις ενδιάμεσες τιμές γωνιών μέχρι π. Στο τέλος η εικόνα αντιστοιχίζεται στο διάστημα $[0, 255]$.

Περιγραφή υλοποίησης

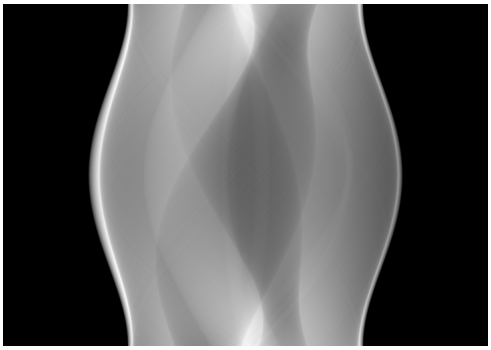
Η υλοποίηση στον κώδικα περιγράφεται ως εξής. Διαβάζεται η εικόνα με την εντολή imread. Οι στήλες επεκτείνονται σε περίπτωση που οι στήλες δεν είναι δύναμη του 2. Στην συνέχεια, αφού βρεθούν πόσα pixel υπολείπονται χρησιμοποιείται η συνάρτηση copyMakeBorders. Αυτή η εντολή, όπως περιγράφτηκε και πιο πάνω, τοποθετεί pixel γύρω από την εικόνα. Μετέπειτα ακολουθεί η συνάρτηση doInverseRadonTransformation όπου γίνεται η οπισθοπροβολή και το φιλτράρισμα. Υπολογίζεται to angle_step σύμφωνα με τις γραμμές του ημιτονοδιαγράμματος και έπεται η κύρια επανάληψη. Εντός της κύριας επανάληψης, υπολογίζεται πρωτίστως η γωνία του μετασχηματισμού και στην συνέχεια η φιλτράρεται η γραμμή, που αντιστοιχεί σε αυτήν την γωνία.

Η διαδικασία για την εύρεση της φιλτραρισμένης γραμμής περιγράφεται ως εξής. Γίνεται μετασχηματισμός Fourier με βάση το λήμμα του Danielson-Lanczos [8]. Η πράξη της συνέλιξης μετατρέπεται σε πολλαπλασιασμό στο πεδίο συχνοτήτων, όπου και πολλαπλασιάζονται οι τιμές με την παράσταση $|\omega|$. Μετέπειτα εφαρμόζεται ο αντίστροφος μετασχηματισμός Fourier. Ο αντίστροφος μετασχηματισμός Fourier δίνει ως αποτέλεσμα ένα διάνυσμα από μιγαδικούς αριθμούς. Προϋπόθεση για να συνεχιστεί η διαδικασία, είναι η απόρριψη του φανταστικού μέρους των μιγαδικών αριθμών και επιστρέφει η ροή του προγράμματος στην κύρια επανάληψη.

Το επόμενο βήμα είναι η κύρια επανάληψη να διατρέξει όλα τα pixel και να προσθέσει τις τιμές του φιλτραρισμένου διανύσματος, σύμφωνα με τον τύπο 4.1. Επομένως, για την συγκεκριμένη γωνία θ και την θέση (x, y) του pixel υπολογίζεται η συνιστώσα του φιλτραρισμένου διανύσματος και προστίθενται στην εικόνα-αποτέλεσμα. Στην περίπτωση που το i ξεπερνάει τις διαστάσεις του διανύσματος, τότε η τιμή απορρίπτεται. Αφού τελειώσει η διαδικασία για κάθε pixel αντιστοιχίζονται οι υπολογισμένες τιμές στο εύρος $[0, 255]$ (κανονικοποίηση της εικόνας), μέσω της εντολής normalize με όρισμα στο norm_type NORM_MINMAX.

Το μειονέκτημα αυτής της μεθόδου είναι ότι κάποιες τιμές απορρίπτονται και αυτό έχει ως συνέπεια την αναποτελεσματικότητα του προγράμματος. Εν αντιθέσει ως κύριο πλεονέκτημα εντοπίζεται η ευκολία υλοποίησής της μεθόδου.

(α΄) Αρχική εικόνα



(β΄) Ημιτονιοδιάγραμμα

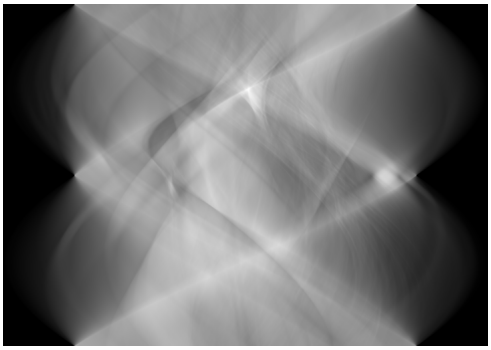
(γ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες

(δ΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες

(ε΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 4.4: Οπισθοπροβολή εικόνας shepp_logan με επανάληψη στα pixel

(α΄) Αρχική εικόνα



(β΄) Ημιτονιοδιάγραμμα

(γ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες

(δ΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες

(ε΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 4.5: Οπισθοπροβολή εικόνας lena με επανάληψη στα pixel

4.2 Τελική υλοποίηση

Τα προβλήματα που προκύπτουν από την ολοκλήρωση της πρώτης υλοποίησης και των δύο μεθόδων, είναι η μη αποδοτικότητα τους αλλά και η μη χρήση κοινού κώδικα με αποτέλεσμα

να κρίνεται αναγκαίο την συγγραφή διαφορετικού κώδικα τόσο για τον μετασχηματισμό Radon όσο και για την φιλτραρισμένη οπισθοπροβολή. Οι δύο μέθοδοι χρησιμοποιούν ως κοινό στοιχείο τις ευθείες που αξιοποιούνται για το μετασχηματισμό Radon και για την οπισθοπροβολή. Οι ευθείες στην περίπτωση του μετασχηματισμού Radon διαπερνούν τα pixel και προβάλλουν την τιμή τους. Ενώ η αντίστροφη διαδικασία υλοποιείται από την οπισθοπροβολή. Τονίζεται πως η προβολή μίας ευθείας η οποία διέρχεται από την εικόνα ορίζεται ως το άθροισμα των φωτεινοτήτων των pixel που τέμνονται από την ευθεία. Επομένως πρέπει να εντοπιστούν οι τομές των ευθειών με τα pixels. Στην απλή περίπτωση που όλα τα pixel έχουν βάρος 1, το παραπάνω μπορεί να πραγματοποιηθεί με τον εξής τρόπο. Για την εύρεση των ευθειών χρησιμοποιείται η μορφή

$$s = x\cos\theta + y\sin\theta \tag{4.3}$$

Παραμετρικά ο τύπος 4.3 μπορεί να δώσει δύο μορφές. Έναν τύπο για $sin\theta \neq 0$ και για $\cos\theta \neq$. Άρα με αυτόν τον τρόπο ισχύει

$$y = -x\frac{cos\theta}{\sin\theta} + \frac{s}{\sin\theta}, \text{για } \theta \neq 0 \tag{4.4}$$

και

$$x = -y\frac{\sin\theta}{\cos\theta} + \frac{s}{\cos\theta}, \text{για } \theta \neq \frac{\pi}{2} \tag{4.5}$$

Επομένως, στον κώδικα υπάρχουν δύο τύποι ευθειών. Ένας τύπος που έχει σαν ανεξάρτητη μεταβλητή το x και εξαρτημένη το y και ένας που έχει σαν ανεξάρτητη το y και σαν εξαρτημένη το x. Το όνομα των δύο τύπων ευθειών στον κώδικα είναι rayx και rayy αντίστοιχα. Το πλεονέκτημα αυτής της μορφής της εξίσωσης των ευθειών είναι ότι χρησιμοποιούν μία μεταβλητή. Ενώ η ανεξάρτητη μεταβλητή ανήκει στους ακέραιους αριθμούς. Αυτό έχει σαν αποτέλεσμα, την εύρεση της τομής του κάθε pixel με την ευθεία χρησιμοποιώντας την εξίσωση της ευθείας. Συνέπεια της διαφορετικής παραμετροποίησης της εξίσωσης των ευθειών, είναι η τροποποίηση του τύπου του μετασχηματισμού Radon. Για $|\sin\theta| < \frac{\sqrt{2}}{2}$

$$F(s,\theta) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(scos\theta - s\sin\theta, s\sin\theta + s\cos\theta)\ \text{ds} \tag{4.6}$$

$$= \frac{1}{\sin\theta}\int_{-\infty}^{\infty} f(x, \frac{s}{\sin\theta} - x\cot\theta)\ \text{dx} \tag{4.7}$$

ενώ για $|\sin\theta| > \frac{\sqrt{2}}{2}$ η 4.7 γίνεται

$$F(s,\theta) = \frac{1}{\cos\theta}\int_{-\infty}^{\infty} f(\frac{s}{\cos\theta} + y\tan\theta, y)\ \text{dy} \tag{4.8}$$

Συνεπώς πρέπει να βρεθούν το ερώτημα οι τιμές που επιτρέπεται να πάρει η ανεξάρτητη μεταβλητή. Λύνοντας τις εξισώσεις αντίστροφα, καταλήγουμε στις εξής εξισώσεις Σε μια εικόνα $N \times N$ για ευθεία $y = \alpha \cdot m + \beta$ τα όρια που μπορεί να πάρει η ανεξάρτητη μεταβλητή θα είναι

$$m_{\min} = \begin{cases} \max\left\{ -\frac{N}{2}, \left\lceil \frac{-\frac{N}{2}-\beta+0.5}{\alpha} \right\rceil \right\}, \text{για } \alpha > 0 \\ \min\left\{ -\frac{N}{2}, \left\lceil \frac{\frac{N}{2}-1-\beta-0.5}{\alpha} \right\rceil \right\}, \text{για } \alpha < 0 \end{cases} \tag{4.9}$$

$$m_{\max} = \begin{cases} \min\left\{ \frac{N}{2}, \left\lfloor \frac{\frac{N}{2}-1-\beta-0.5}{\alpha} \right\rfloor \right\}, \text{για } \alpha > 0 \\ \max\left\{ \frac{N}{2}, \left\lfloor \frac{-\frac{N}{2}-\beta+0.5}{\alpha} \right\rfloor \right\}, \text{για } \alpha < 0 \end{cases} \tag{4.10}$$

Αυτό οδήγησε στην υλοποίηση μίας κύριας κλάσης που ονομάζεται ProjectorNN. Σκοπός αυτής της κλάσης, είναι η εύρεση των ευθειών, καθώς και η εύρεση των βαρών που αντιστοιχεί σε κάθε pixel. Η κλάση χρησιμοποιείται ως εξής. Αφού διαβάσει τις διαστάσεις της εικόνας διατρέχει τις ευθείες που προκύπτουν από τις διαστάσεις της εικόνας. Ο χρήστης μπορεί να θέσει μία callback συνάρτηση η οποία καλείται σε κάθε τομή του pixel με την ευθεία. Μέσα στην συνάρτηση δίνεται σαν όρισμα, ο αριθμός γωνίας και η θέση της αρχής της ευθείας, καθώς και την αντιστοίχηση της με το εκάστοτε pixel της εικόνας που διατρέχει. Αυτό επιτρέπει, να υλοποιηθεί και ο μετασχηματισμός Radon και η οπισθοπροβολής.

4.2.1 Υλοποίηση κλάσης ProjectorNN

Το όνομα της κλάσης περιγράφει τον τρόπο που κάνει προβολή. Τα αποτελέσματα της εξαρτημένης μεταβλητής της ευθείας είναι δεκαδικοί αριθμοί και επομένως πρέπει να στρογγυλοποιηθούν ώστε να γίνεται αντιστοιχία σε κάποιο pixel. Για το συγκεκριμένο λόγο χρησιμοποιείται στρογγυλοποίηση στον κοντινότερο ακέραιο ή στον κοντινότερο γείτονα επειδή αυτή η μέθοδος εφαρμόζεται σε pixel.

Η κλάση για τον εντοπισμό των ευθειών ξεκινάει βρίσκοντας το βήμα της γωνίας και δημιουργείται ένα αντικείμενο τύπου PixelWeightData, το οποίο παρέχει τα δεδομένα στην callback συνάρτηση. Η πρώτη κύρια επανάληψη διατρέχει την γωνία, ενώ η εμβόλιμη επανάληψη αφορά την μετατόπιση της ευθείας. Αναλόγως την γωνία χρησιμοποιείται είτε ο τύπος rayx ή rayy. Στην προαναφερθείσα ευθεία βρίσκονται τα όρια της με την εικόνα. Επιπλέον, γίνεται μία ακόμη εμβόλιμη επανάληψη για κάθε ακέραιο σημείο της ευθείας. Σαν αποτέλεσμα, εξασφαλίζεται ότι έχουμε κοινό κώδικα.

4.2.2 Υλοποίηση μετασχηματισμού Radon

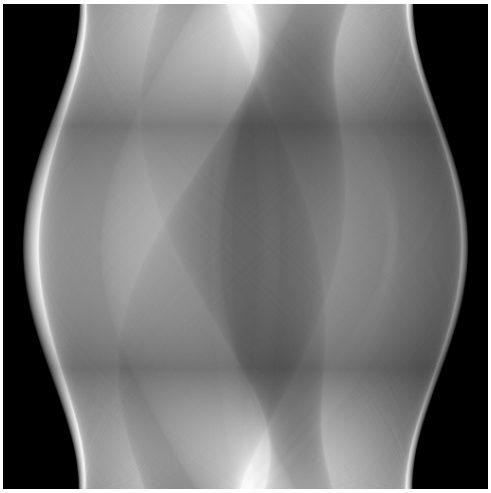
Αφού διαβαστεί από το πρόγραμμα το όνομα της εικόνας και τον αριθμό των γωνιών που επιθυμεί ο χρήστης, διαβάζεται η εικόνα με την χρήση της OpenCV. Έπειτα κατασκευάζεται ένα αντικείμενο τύπου ProjectorNN και θέτεται μία callback συνάρτηση. Η callback συνάρτηση γράφει μόνο τις τιμές που δίνει σαν όρισμα η ProjectorNN. Αφού τελειώσει η διαδικασία, κανονικοποιείται η εικόνα για να προβληθεί και να αποθηκευτεί.

4.2.3 Υλοποίηση οπισθοπροβολής

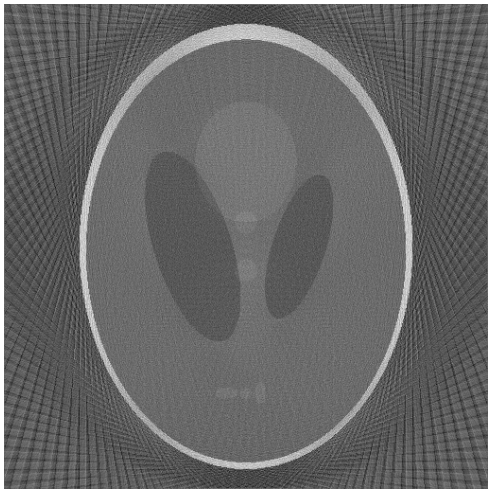
Στην περίπτωση της οπισθοπροβολής, πρωτίστως φιλτράρεται το ημιτονοδιάγραμμα. Τέλος, οι φιλτραρισμένες γραμμές θα χρησιμοποιηθούν για να προβληθούν πίσω στη εικόνα. Έπειτα γίνεται μία κανονικοποίηση της εικόνας όπως στις προηγούμενες μεθόδους.

Στο παρόν κεφάλαιο υλοποιήθηκαν διάφορες μέθοδοι ανακατασκευής της εικόνας, όμως χρησιμοποιώντας αριθμούς κινητής υποδιαστολής, στην συνέχεια θα υλοποιηθεί η τελευταία μέθοδος ανακατασκευής εικόνας χρησιμοποιώντας αριθμούς σταθερής υποδιαστολής, όπου αποτελεί τον τελικό στόχο αυτής της εργασίας.

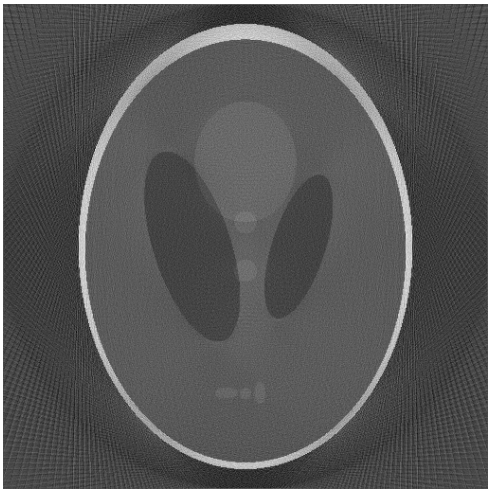
(α΄) Αρχική εικόνα



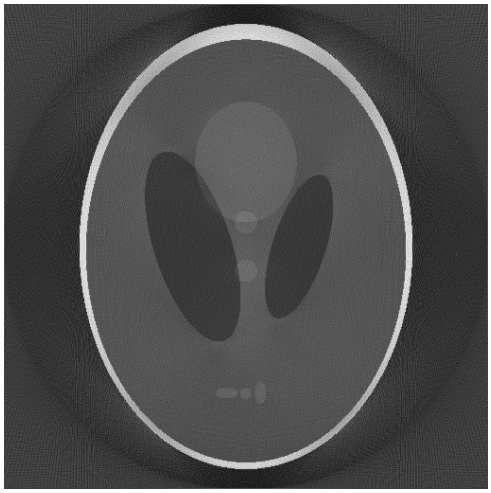
(β΄) Ημιτονιοδιάγραμμα



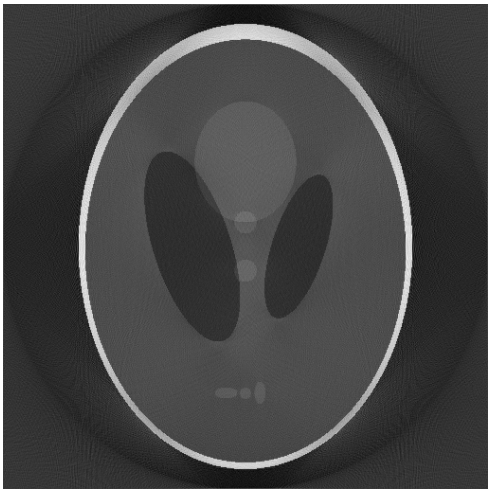
(γ΄) Οπισθοπροβολή χρησιμοποιώντας 64 γωνίες



(δ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες



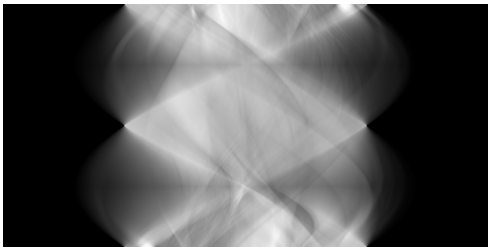
(ε΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες



(ϛ΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 4.6: Οπισθοπροβολή εικόνας Shepp_Logan με την τελική έκδοση της μεθόδου προβολής.

(α΄) Αρχική εικόνα



(β΄) Ημιτονιοδιάγραμμα

(γ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες

(δ΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες

(ε΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 4.7: Οπισθοπροβολή εικόνας Lena με την τελική έκδοση της μεθόδου προβολής.

Κεφάλαιο 5

Λογισμικό υπολογιστή με χρήση αριθμών σταθερής υποδιαστολής

Στην υλοποίηση αυτή αξιοποιήθηκαν οι αριθμοί σταθερής υποδιαστολής. Η τελική υλοποίηση έχει ως κύριο σκοπό την σχετικά ομαλότερη μετάβαση σε μία υλοποίηση σε επεξεργαστές οι οποίοι δεν διαθέτουν κάποιο υποσύστημα κινητής υποδιαστολής ή σε κάποιο σύστημα προγραμματιζόμενου υλικού παραδείγματος χάρη FPGA. Εδώ να σημειωθεί ότι όλος ο κώδικας βρισκεται στο Παράρτημα Α

5.1 Αριθμοί κινητής υποδιαστολής

Ένας αριθμός κινητής υποδιαστολής έχει γενικότερα την μορφή

$$x = m \cdot \beta^e \tag{5.1}$$

Όπου e είναι ο εκθέτης όπου μπορεί να είναι θετικός ή αρνητικόςβ η βάση και m είναι το κλασματικό μέρος ή mantissa. Έτσι για παράδειγμα αν η βάση είναι 10 τότε ο αριθμός 3.14 τότε ο αριθμός αυτός μπορεί να γραφτεί ως

$$3.14 = 0.314 \cdot 10^1 = 3.14 \cdot 10^0 \tag{5.2}$$

Εδώ παρατηρείται ότι για έναν αριθμό μπορεί να έχουμε πολλές αναπαραστάσεις σε αριθμό κινητής υποδιαστολής. Για να γίνει μοναδική η αναπαράσταση χρησιμοποιείται η κανονικοποιημένη μορφή. Η κανονικοποιημένη μορφή του κλασματικού μέρους είναι η θεώρηση του κλασματικού αριθμού ότι το ακέραιο του μέρος είναι 0 και το πιο σημαντικό ψηφίο του είναι διάφορο του μηδενός.

5.2 Αριθμοί σταθερής υποδιαστολής

Οι αριθμοί σταθερής υποδιαστολής είναι είναι αριθμοί οι οποίοι αποτελούνται από δύο μέρη, από το ακέραιο και το κλασματικό μέρος. Για να κατασκευαστεί ένας τέτοιος αριθμός, γίνεται ένας πολλαπλασιασμός με τον επιθυμητό δεκαδικό αριθμό και μία αυθαίρετη σταθερά. Η αυθαίρετη σταθερά είναι ένας αριθμός ο οποίος είναι δύναμη του 2 και είναι ευθύνη του

μηχανικού ώστε να βρει μία σταθερά τέτοια ώστε να μην συμβαίνει υπερχείλιση ή να χάνει σε ακρίβεια. Για να συμβολιστούν αυτοί οι αριθμοί χρησιμοποιείται ο συμβολισμός Q συνοδευόμενος από δύο αριθμούς χωρισμένοι με μία τελεία $Q[I.F]$. Εδώ θα πρέπει να σημειωθεί ότι ιδιαίτερη σημασία θα πρέπει να δωθεί στο πρόσημο, καθώς για την αναπαράσταση αρνητικών αριθμών χρησιμοποιούνται αριθμοί συμπληρώματος του 2.

Στην πρόσθεση δύο αριθμών η τελική αναπαράσταση έχει πάντα αποτέλεσμα ο τελικός αριθμός να πάρει την μέγιστη τιμή από το ακέραιο και το κλασματικό μέρος των επιμέρους αριθμών.

$$Q[I_1, F_1] + Q[I_2, F_2] = Q[\max(I_1, I_2), \max(F_1, F_2)] \tag{5.3}$$

Στον πολλαπλασιασμό τα δύο μέρη του αριθμού απλά προστίθενται μεταξύ τους

$$Q[I_1, F_1] \times Q[I_2, F_2] = Q[I_1 + I_2, F_1 + F_2] \tag{5.4}$$

Στην διαίρεση γίνεται αφαίρεση ανάμεσα στα δύο μέρη. Εδώ χρειάζεται προσοχή καθώς υπάρχει περίπτωση ο διαιρετέος να είναι μικρότερος από τον διαιρέτη σε αυτή την περίπτωση αρκεί να γίνει αρχικά ένας πολλαπλασιασμός του διαιρετέου με μία δύναμη του 2.

$$\frac{Q[I_1, F_1]}{Q[I_2, F_2]} = Q[I_1 - I_2, F_1 - F_2] \tag{5.5}$$

5.3 Υλοποίηση βασικών συναρτήσεων με αριθμούς σταθερής υποδιαστολής

Με την αξιοποίηση αυτών των αριθμών υλοποιήθηκαν και κάποιες βασικές ρουτίνες. Αυτές είναι:

- double2FP, μετατρέπει έναν αριθμό από κινητή υποδιαστολή σε ακέραιο
- FP2double, Μετατρέπει έναν αριθμό από σταθερή υποδιαστολή σε κινητή
- round, Στρογγυλοποιεί τον αριθμό στον κοντινότερο ακέραιο.
- floor, Στρογγυλοποιεί τον αριθμό στον αμέσως μικρότερο ακέραιο.
- ceil, Στρογγυλοποιεί τον αριθμό στον αμέσως μεγαλύτερο ακέραιο.
- half. Επιστρέφει την τιμή του 0.5 στην εκάστοτε Q αριθμητική.
- round_towards_zero, Στρογγυλοποιεί τον αριθμό στον επόμενο ακέραιο που βρίσκεται πιο κοντά στο μηδέν.

Όλες οι ρουτίνες χρησιμοποιούν δύο template ορίσματα. Το πρώτο που αναγράφεται σαν qn δίνει την Q αριθμητική και το δεύτερο είναι το INT_TYPE που συμβολίζει τον τύπο του ακεραίου που χρησιμοποιείται(char, short, int, long int).

Τέλος, υλοποιήθηκαν και τριγωνομετρικές συναρτήσεις. Οι τριγωνομετρικές συναρτήσεις υλοποιήθηκαν με δύο τρόπους, με πολυώνυμο και με LUT(lookup table).

Η υλοποίηση πολυώνυμου τρίτου βαθμού ερευνήθηκε αρχικά ως πιθανά καλύτερη εναλλακτική από την εύρεση τιμών από LUT[9]. Η πρώτη προσσέγγιση του προβλήματος αυτού είναι η ανάπτυξη του ημιτόνου με την σειρά MacLaurin.

$$\sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}h^5 - \frac{1}{7!}h^7 + ... \tag{5.6}$$

Από την άλλη, μπορεί να βρεθεί ένα μία καλύτερη προσσέγγιση αν θεωρηθεί ότι το ημίτονο θα είναι της μορφής

$$S_3(x) = ax - bx^3 \tag{5.7}$$
$$S_3'(x) = x - 3bx^2 \tag{5.8}$$

Έτσι για το σημείο $\frac{\pi}{2}$ οι παράμετροι a, b θα δώσουν $\frac{3}{\pi}$ και $\frac{4}{\pi^3}$ αντίστοιχα. όπως παρατηρείται στην εξίσωση υπάρχουν αριθμοί που είναι σχετικά δύσκολο για να παρασταθούν. Γενικά θα ήταν προτιμότερο να χρησιμοποιηθούν καθαρές μονάδες και όχι ακτίνια. Έτσι θεωρείται ότι $z = \frac{x}{\frac{\pi}{2}}$. Επομένως αυτό θα δώσει

$$S_3(z) = 0.5z(3 - z^2) \tag{5.9}$$

Στο τέλος, αυτός ο τύπος υλοποιείται σαν συνάρτηση. Αυτή η συνάρτηση μπορεί να βρεθεί με το όνομα fp::sin3rd.

Η δεύτερη υλοποίηση, όπου και αυτή στο τέλος χρησιμοποιήθηκε, έχει να κάνει με την χρήση LUTs. Μέσα από αυτή έγινε χρήση των templates στην C++ ώστε να παρθεί κατά την μεταγλώττιση οι τιμές του πίνακα. Η υλοποίηση ενός πίνακα LUT βρίσκεται στο αρχείο lut.hpp. Μέσα από το template δημιουργήθηκαν οι πίνακες για το ημίτονο, την εφαπτομένη και το αντίστροφο ημίτονο, ενώ μέσω μετασχηματισμών αποκτήθηκαν και οι υπόλοιπες τριγωνομετρικές συναρτήσεις. Οι πίνακες αυτοί ισχύουν από 0 έως π. Για πάρουν οι εξισώσεις και όλο το υπόλοιπο εύρος και στους αρνητικούς αριθμούς άλλα και μέχρι 2π έγιναν κάποιοι χειρισμοί πάνω στα bit άλλα και στην συμμετρία των τριγωνομετρικών αριθμών. Οι τριγωνομετρικές συναρτήσεις μπορούν να βρεθούν στο αρχείο trig_funcs_fixed_point.hpp.

Για τις συναρτήσεις που αφορούν την οπισθοπροβολή έγινε μία απευθείας μετάφραση των ρουτινών από αριθμούς κινητής υποδιαστολής σε αριθμούς σταθερής υποδιαστολής.

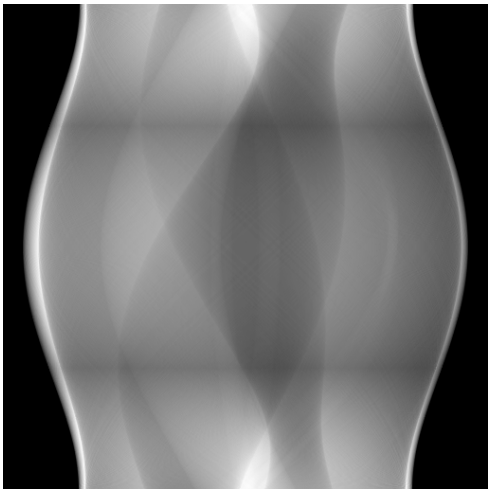
Για την εύρεση ορίων της ευθείας, χρησιμοποιήθηκε η συνάρτηση calcLowerUpperLimitsFP. Σε αντίθεση με την ρουτίνα κινητής υποδιαστολής προστέθηκε και μία επιπλέον διαδικασία κατά της οποία η ρουτίνα κάνει μία αναζήτηση μέχρι να βρει σε ποια τιμή τέμνει την εικόνα για ανακατασκευή. Για την υλοποίηση του γρήγορου μετασχηματισμού Fourier χρησιμοποιήθηκε μία ρουτίνα που υλοποιεί τον αλγόριθμο Cooley-Tucky. Όμως για τον αντίστροφο μετασχηματισμό Fourier χρησιμοποιήθηκε μία ιδιότητα του μετασχηματισμού. Αν αντιστραφεί το πραγματικό και το φανταστικό κομμάτι της εισόδου τότε το αποτέλεσμα από τον FFT θα είναι ο αντίστροφος μετασχηματισμός. Στο τέλος, θα πρέπει να γίνει εκ νέου αντιστροφή των εξόδων μεταξύ τους. Ύστερα μπορούν οι τιμές της φιλτραρισμένης εξόδου να προβληθούν ξανά πίσω στην εικόνα.

5.4 Αποτελέσματα υλοποίησης

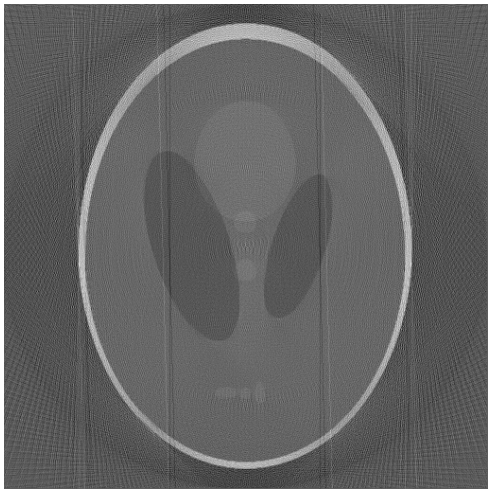
5.4.1 Ανάλυση αποτελεσμάτων

Τα αποτελέσματα και στις δύο εικόνες μετά από τον αντίστροφο, φαίνεται να έχουν περισσότερες ατέλειες. Επίσης, παρατηρήθηκε ότι σε γενικές γραμμές χρειάζονται περισσότερες προβολές οι εικόνες για την πιο πιστή απεικόνιση τους μετά την διαδικασία οπισθοπροβολής. Επιπλέον, ένα πρόβλημα που παρατηρήθηκε και στις δύο εικόνες είναι ότι χρειάζονται επιπλέον φιλτράρισμα για την ομαλοποίηση των μεταβάσεων των χρωμάτων της εικόνας λόγω της μεθόδου. Σε γενικές γραμμές, φαίνεται ότι ο αλγόριθμος της οπισθοπροβολής δύναται να δουλέψει και με αριθμούς σταθερής υποδιαστολής.

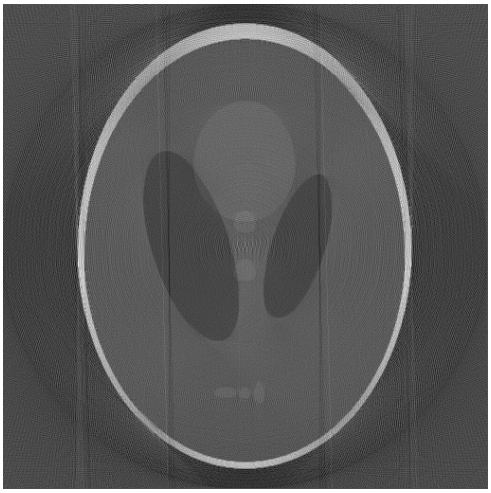
(α΄) Αρχική εικόνα



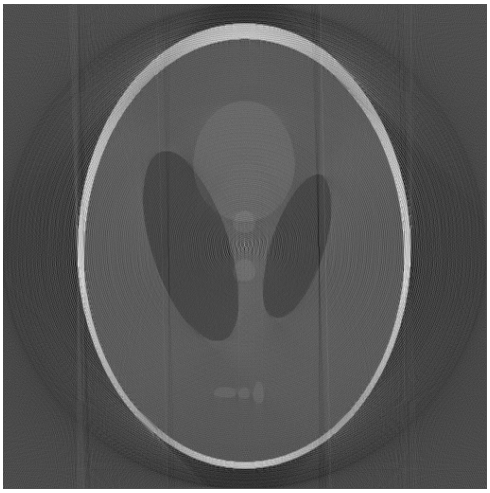
(β΄) Ημιτονιοδιάγραμμα



(γ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες



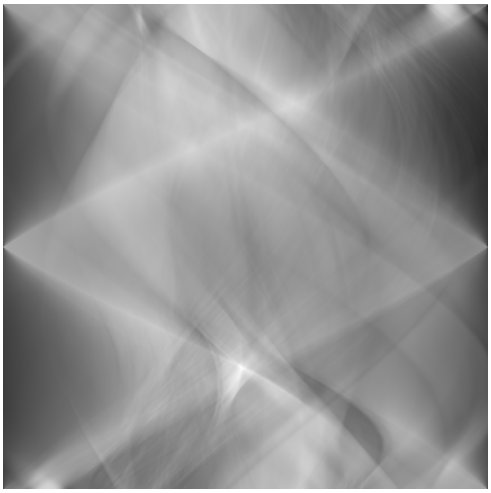
(δ΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες



(ε΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 5.1: Οπισθοπροβολή εικόνας Shepp_Logan με την τελική έκδοση της μεθόδου προβολής και με αξιοποίηση αριθμών σταθερής υποδιαστολής.

(α΄) Αρχική εικόνα



(β΄) Ημιτονιοδιάγραμμα

(γ΄) Οπισθοπροβολή χρησιμοποιώντας 128 γωνίες

(δ΄) Οπισθοπροβολή χρησιμοποιώντας 256 γωνίες

(ε΄) Οπισθοπροβολή χρησιμοποιώντας 512 γωνίες

Σχήμα 5.2: Οπισθοπροβολή εικόνας Lena με την τελική έκδοση της μεθόδου προβολής και με αξιοποίηση αριθμών σταθερής υποδιαστολής.

Κεφάλαιο 6

Συμπεράσματα

Σε αυτήν την εργασία περιγράφτηκαν μέθοδοι που είναι ικανές να μοντελοποίησουν την διαδικασία των CT scans, την αντίστροφη διαδικασία, καθώς και να χρησιμοποιηθούν αριθμοί σταθερής υποδιαστολής για την υλοποίηση τους. Υλοποιήθηκαν αλγόριθμοι ανακατασκευής εικόνας για να εξερευνηθούν οι δυνατότητες και οι αδυναμίες των συγκεκριμένων μεθόδων. Ύστερα, επιλέχθηκε η μέθοδος ProjectorNN όπου υλοποιήθηκε με την χρήση αριθμών σταθερής υποδιαστολής.

Με την υλοποίηση με αριθμούς σταθερής υποδιαστολής, εξερευνήθηκε η αξιοποίηση τους για την υλοποίηση σύνθετων αλγορίθμων και το πως μπορεί να αντιμετωπιστεί απώλεια ακρίβειας μέσω ορισμένων τεχνικών. Η εργασία αυτή μέσω αυτής της υλοποίησης και μελέτης επιτρέπει την υλοποίηση του αλγορίθμου της οπισθοπροβολής σε συσκευές που δεν διαθέτουν συστήματα για την επεξεργασία αριθμών κινητής υποδιαστολής. Ακόμα θα μπορούσε να γίνει κάποιος συσχεδιασμός υλικού και λογισμικού, καθώς ο αλγόριθμος της οπισθοπροβολής ενδείκνυται για παραλληλοποίηση.

Παράρτημα Α΄

Κώδικας

Α΄.1 Δομή αρχείων

image projection
- backprojection
  - backprojection.cpp
  - backprojection_nn.cpp
  - CMakeLists.txt
- bilinear_based
  - bilinear_based.cpp
  - bilinear_projector.cpp
  - bilinear_projector.hpp
  - CMakeLists.txt
- common_utils
  - CMakeLists.txt
  - fft.cpp
  - fft.hpp
  - fixed_point_utils.cpp
  - fixed_point_utils.hpp
  - general_utils.cpp
  - general_utils.hpp
  - lut.hpp
  - LUT_256.h
  - LUT_512.h
  - LUT_1024.h
  - math_utils.cpp
  - math_utils.hpp
  - program_options.cpp
  - program_options.hpp
  - projector.cpp
  - projector.hpp
  - projector_fixed_point_nn.cpp
  - projector_fixed_point_nn.hpp

```
        ├── projector_nn.cpp
        ├── projector_nn.hpp
        ├── sparse_matrix.cpp
        ├── sparse_matrix.hpp
        └── trig_funcs_fixed_point.hpp
    ├── pixel_based
        ├── CMakeLists.txt
        ├── pixel_based_simple.cpp
        └── pixel_based_w_rotate.cpp
    └── CMakeLists.txt
```

backprojection/backprojection.cpp

```
#include <opencv2/opencv.hpp>
#include <cmath>
#include "../common_utils/general_utils.hpp"
#include "../common_utils/fft.hpp"

static void doInverseRadonTransformation( const cv::Mat &input_im, cv::Mat *output_im );
static void makeImageOfPowerOf2( cv::Mat *input_im );

int main(){
  cv::Mat input_im = cv::imread( "../images/sin.png", cv::IMREAD_GRAYSCALE );
  cv::Mat output_im;
  cv::Mat temp;

  makeImageOfPowerOf2( &input_im );

  doInverseRadonTransformation( input_im, &output_im );
  cv::normalize( output_im, temp, 0, 255, cv::NORM_MINMAX, CV_8UC1 );
//  cv::normalize( output_im, temp, 255, 0, cv::NORM_INF, CV_8UC1 );
//  std::cout << temp << std::endl;
  cv::imshow( "", temp );
  cv::imwrite("../images/output.png", temp );
  cv::waitKey();

  return 0;
}


static std::vector< float > getFilteredRow( const cv::Mat &input_im, unsigned int i )
{
  using namespace fft;
  std::vector< float > ret;
  std::vector< float > input;
  std::vector< fcomplex > out_fft;
  std::vector< fcomplex > filtered_data;
  std::vector< fcomplex > out_fft_complex;

  ret.resize( input_im.cols, 0. );
  input.resize( input_im.cols, 0. );
  out_fft.resize( input_im.cols, 0. );

  for( unsigned int k = 0; k < input_im.cols; ++k ) {
//    input[k] = float(input_im.at< unsigned char >( i, k ))/float(255);
```

```
    input[k] = float(input_im.at<unsigned char>(i, k));
//      ret[k] = input_im.at<unsigned char>(i, k);
  }

  fft::doRealFFT_forward(input, &out_fft);
//  fft::doDFT(input, &out_fft ,0);

  out_fft[0] = 0;
  double fcutoff = 1;
  for(unsigned int k = 1; k < out_fft.size()/2; ++k) {
    double w = k*2*M_PI/out_fft.size();
//      double filt0 = 3*100000.*1./(1+.414*std::pow(fcutoff/(w), 2)));
//      double filt0 = abs(std::cos(w));
//      float filt0 = (out_fft.size() - k)*2*M_PI/out_fft.size();
//      double filt0 = .5*k*std::sin(M_PI*k/
    float filt1 = k*2*M_PI/out_fft.size();
//      float filt1 = k;
    float filt0 = 1.;
//      double filt1 = 1.;
    out_fft[k] *= filt0 * filt1;
    out_fft[out_fft.size() - k] *= filt0 * filt1;
  }

  unsigned int k = out_fft.size()/2;
  double w = k*2*M_PI/out_fft.size();
  float filt1 = k*2*M_PI/out_fft.size();
//  double filt0 = 3*10000.*1./(1+.414*std::pow(fcutoff/(w), 2)));
//  float filt0 = (out_fft.size() - k)*2*M_PI/out_fft.size();
//  double filt0 = abs(std::sin(w));
  double filt0 = 1.;
//  double filt1 = 1.;
  out_fft[out_fft.size()/2] *= filt0 * filt1;

  fft::doFFT_dl(out_fft, &filtered_data, true);
//  fft::doDFT(out_fft, &filtered_data, true);


  for(unsigned int k = 0; k < filtered_data.size(); ++k) {
    ret[k] = (filtered_data[k]).real();
  }

  return ret;
}

static void doInverseRadonTransformation(const cv::Mat &input_im, cv::Mat *output_im)
{
  // calculate angle steps in radians
  int side = input_im.cols;
  int angle_num = input_im.rows;
  *output_im = cv::Mat(side, side, CV_32F); // the new matrix will have the same dims
  double angle_step = M_PI/double(angle_num);

  int side_2 = side >> 1;

  for(int i = 0; i < angle_num; i+=1) {
    double theta = i*angle_step;
```

```
    auto filtered_row = getFilteredRow( input_im, i );

    for( int x = 0; x < side; ++x ) {
      for( int y = 0; y < side; ++y ) {
//        int prj_index = int( std::floor( ( x - side_2 ) * cos(theta) + (y - side_2 ) * sin(theta)
    + 0.5 + side_2 ) );
        int prj_index = int( ( +( x - side_2 ) * sin(theta) - (y - side_2 ) * cos(theta) + side_2 ) )
    ;
        bool do_not_add = false;

//        if it is out of bounds circle around
        if( prj_index >= side ) {
//          if( i == 1 ) std::cout << prj_index << std::endl;
          prj_index -= side;
          do_not_add = true;
        } else if( prj_index < 0 ) {
//          if( i == 1 )std::cout << prj_index << std::endl;
          prj_index += side;
          do_not_add = true;
        }
//        if( prj_index < side && prj_index >= 0 )
//        output_im->at<float>( x, y ) += float( filtered_row[ prj_index ] );
      if( !do_not_add )
        output_im->at<float>( x, y ) += float( filtered_row[ prj_index ] )/float(input_im.cols);
      }
    }
  }

//  std::cout << *output_im << std::endl;
//  cv::normalize( *output_im, ( *output_im ), 1., 0, cv::NORM_MINMAX );
}

void makeImageOfPowerOf2( cv::Mat *input_im )
{
  int pad1 = 0;
  int pad2 = 0;
  int new_size = nextPowerOf2( input_im->cols );
  int extra_pad = new_size - input_im->cols;
  if( extra_pad & 0x1 ) {
    pad2 = extra_pad >> 1;
    pad1 = pad2 + 1;
  } else {
    pad2 = extra_pad >> 1;
    pad1 = pad2;
  }
  cv::copyMakeBorder( *input_im, *input_im, 0, 0, pad1, pad2, cv::BORDER_CONSTANT, cv::Scalar( 0, 0,
     0 ) );
}
```

backprojection/backprojection_nn.cpp

```
//
// Created by markos on 17/4/22.
//

#include <opencv2/opencv.hpp>
```

```
#include <cmath>
#include "../common_utils/general_utils.hpp"
#include "../common_utils/projector_nn.hpp"
#include "../common_utils/fft.hpp"

static void doBackprojection( const cv::Mat &input_im, cv::Mat *output_im );

int main()
{
  cv::Mat input_im = cv::imread( "../images/sin.png", cv::IMREAD_GRAYSCALE );
  cv::Mat output_im;
  cv::Mat temp;
  double min_val, max_val;
  cv::Point min_loc, max_loc;

  if( input_im.cols != nextPowerOf2( input_im.cols ) ) {
  std::cerr << "Sinogram is not power of 2." << std::endl;
  std::abort();
  }
  doBackprojection( input_im, &output_im );
  cv::normalize( output_im, temp, 0, 255, cv::NORM_MINMAX, CV_8UC1 );
  cv::minMaxLoc( output_im, &min_val, &max_val, &min_loc, &max_loc );
  std::cout << "Max_val\t" << max_val << "\t" << max_loc << std::endl;
  cv::imwrite("../images/output.png", temp );
  cv::imshow("", temp );
  cv::waitKey();
}

static std::vector< float > getFilteredRow( const cv::Mat &input_im, unsigned int i )
{
  using namespace fft;
  std::vector< float > ret;
  std::vector< float > input;
  std::vector< fcomplex > out_fft;
  std::vector< fcomplex > filtered_data;
  std::vector< fcomplex > out_fft_complex;

  ret.resize( input_im.cols, 0. );
  input.resize( input_im.cols, 0. );
  out_fft.resize( input_im.cols, 0. );

  for( unsigned int k = 0; k < input_im.cols; ++k ) {
//      input[k] = float(input_im.at< unsigned char >( i, k ))/float(255);
  input[k] = float(input_im.at< unsigned char >( i, k ));
//      ret[k] = input_im.at< unsigned char >( i, k );
  }

  fft::doRealFFT_forward( input, &out_fft );
//  fft::doDFT( input, &out_fft ,0 );

  out_fft[0] = 0;
  double fcutoff = 1;
  for( unsigned int k = 1; k < out_fft.size()/2; ++k ) {
    double w = k*2*M_PI/out_fft.size();
  //      double filt0 = .5*k*std::sin( M_PI*k/
```

```
      float  filt1  = 0.;
      if( k < out_fft.size()/2 ) {
        filt1  = k*2*M_PI/out_fft.size();
//        filt1  *= ( out_fft.size() - k )*2*M_PI/out_fft.size();
//        filt1  *= abs( std::cos( w ) );
//        filt1  = k;
//        filt1  *=  3*100.*1./((  1 + .414*std::pow( fcutoff/( w ), 2 )));
      }
//      filt1  = 1.;
      float  filt0  = 1.;
    //    double  filt1  = 1.;
      out_fft[k] *=  filt0 * filt1 ;
      out_fft[ out_fft.size() - k ] *=  filt0 * filt1 ;
    }

    unsigned int k = out_fft.size()/2;
    double w = k*2*M_PI/out_fft.size();
    float  filt1  = k*2*M_PI/out_fft.size();
//    double  filt0  = 3*10000.*1./(  (1+.414*std::pow(fcutoff/(w), 2))) ;
//    float  filt0  = ( out_fft.size() - k )*2*M_PI/out_fft.size();
// //    double  filt0  = abs( std::sin( w ) );
    double  filt0  = 1.;
//    double  filt1  = 1.;
    out_fft[ out_fft.size()/2] *=  filt0 * filt1 ;

    fft::doFFT_dl( out_fft, &filtered_data, true );
//    fft::doDFT( out_fft, &filtered_data, true );


    for( unsigned int k = 0; k < filtered_data.size(); ++k ) {
    ret[k] = ( filtered_data[k]).real();
    }

    return ret;
}

void doFFTOnImage( const cv::Mat &input_im, cv::Mat *output_im )
{
//    input_im.copyTo( *output_im ):
//    *output_im = input_im;
    output_im->create( input_im.rows, input_im.cols, CV_64F );
    for( unsigned int k = 0; k < input_im.rows; ++k ) {
      auto filtered_row = getFilteredRow( input_im, k );
      for( unsigned int i = 0; i < filtered_row.size(); ++i ) {
        output_im->at< double >( k ,i ) = filtered_row[ i ];
//        output_im->at< float >( k ,i ) = 1.f*input_im.at<unsigned char>(k,i);
      }
    }
}
void doBackprojection (const cv::Mat &input_im, cv::Mat *output_im)
{
    cv::Mat temp;
    cv::Mat ttemp( input_im.cols, input_im.cols, CV_8U );
    doFFTOnImage( input_im, &temp );
    *output_im = cv::Mat( input_im.cols, input_im.cols, CV_64F );
    ProjectorNN prj( *output_im, input_im.cols, input_im.rows );
```

```
  int na = 0;
  prj.setPixelWeightFun( [&temp, output_im, &na, &ttemp]( const PixelWeightData &pwd ) {
//      if( na != pwd.na ) {
//        cv::imshow( "", ttemp );
//        cv::waitKey(10);
//        ttemp = 0;
//        na = pwd.na;
//      }
    if( ( ( 0 <= pwd.i && pwd.i < 512 ) && ( 0 <= pwd.j && pwd.j < 512 ) ) ) {
      output_im->at< double >( pwd.i, pwd.j ) += double( temp.at< double >( pwd.na, pwd.nr ))/temp.
     cols;

//        if( pwd.nr == 255 )
//        ttemp.at< unsigned char >( pwd.i, pwd.j ) = 254;
//      } else {
//        std::cout << "i did it" << std::endl;
//      }
//      output_im->at< float >( pwd.i, pwd.j ) += float( temp.at< float >( pwd.na, pwd.nr ) )/temp.cols
     ;
    }
  } );

  prj.run();
//   output_im->at< float >( 255,255 ) = 0.;
//   double min_val, max_val;
//   cv::Point min_loc, max_loc;
//   cv::minMaxLoc( *output_im, &min_val, &max_val, &min_loc, &max_loc );
//   std::cout << "max_val: " << max_val << "\t" << max_loc << std::endl;
}
```

bilinear_based/bilinear_based.cpp

```
//
// Created by markos on 12/3/22.
//
#include <iostream>
#include <opencv2/opencv.hpp>
#include "../common_utils/program_options.hpp"
#include "../common_utils/math_utils.hpp"
#include "bilinear_projector.hpp"

int main( int argc, char *argv[] )
{
  ProgramOptions po( argv, argc );
  const std::string &input = po.getIn();
  const std::string &output = po.getOut();
  const std::string weights_file_name = po.getWeightsFile();
  unsigned int num_scans = po.getAnglesNum();
  unsigned int num_rays = po.getRaysNum();

  std::cout << "Input file: " << input << std::endl;
  std::cout << "Output file: " << output << std::endl;
  std::cout << "Output weight file: " << weights_file_name << std::endl;
  cv::Mat image = cv::imread( input, cv::IMREAD_GRAYSCALE);

  if( image.cols != image.rows ) {
```

```
    std :: cerr << "Image is not square" << std :: endl;
    exit (1) ;
  }

  if ( image.empty() ) {
    std :: cerr << "Could not find file " << std :: endl;
    exit (1) ;
  }

  if ( !num_rays ) {
    num_rays = image.rows - 1;
  }

  double step_angle = M_PI/double( num_scans );
  cv :: Mat sinogram( num_scans, num_rays, CV_64F);
  std :: vector< double > prj_data ( num_rays, 0. ) ;

   BilinearProjector bil_prj ( image, num_rays, num_scans );
   bil_prj .run() ;

  if ( !weights_file_name.empty() ) {
    bil_prj . writeToFile ( weights_file_name ) ;
  }

  auto *row = sinogram.ptr<double>(0);
  auto res = bil_prj .getRes() ;

  std :: copy( res .begin() , res .end() , row ) ;

  cv :: Mat sin_normalized;
  cv :: normalize( sinogram, sin_normalized , 1., 0., cv :: NORM_INF );

  cv :: imshow("", sin_normalized ) ;

  cv :: Mat out_mat;
  sin_normalized.convertTo(out_mat, CV_8U, 255 ) ;
  cv :: imwrite( output , out_mat ) ;

  cv :: waitKey() ;

}
```

bilinear_based/bilinear_projector.cpp

```
//
// Created by markos on 19/3/22.
//

#include " bilinear_projector .hpp"
#include " ../ common_utils/sparse_matrix.hpp"

 BilinearProjector :: BilinearProjector (const cv :: Mat& mat, std :: size_t nr , std :: size_t na)
     : Projector (mat, nr , na) , m_prj_mat( mat )
{

}
```

```
static double calcNewSpacing( double dist, double spacing, unsigned int *points )
{
  *points = std::floor( dist/spacing + .5 );
  return dist/double( *points );
}

static void saveToMap( std::unordered_map< unsigned int, double > *pi_wei, const math::Cell &cell,
    unsigned int side, double spacing )
{
  unsigned int offset = cell.m_i*side + cell.m_j;

  auto it = pi_wei->find( offset );
  if( it != pi_wei->end() ) {
    it->second += spacing * cell.m_w;
  } else {
    pi_wei->emplace( offset, cell.m_w*spacing );
  }
}

static double calcLineProjection ( math::PrjMat& prj_mat, math::Ray& ray, std::vector<std::pair<
    unsigned int, double > > *pixel_weights )
{
  using namespace std;
  using namespace std::placeholders;
  double ret = 0.;
  std::unordered_map< unsigned int, double > pi_wei;
  const double spacing = 0.5;
  double new_spacing = spacing; /* spacing has to be adjusted */
  double t0 = 0.;
  double t1 = 1.;
  double int_dist = 0.;
  unsigned int points = 0;
  math::Vec2 p0;
  math::Vec2 p1;
  std::array< math::Cell, 4 > cells;

  math::intCircle ( ray, prj_mat.getRad(), &t0, &t1 );
  pi_wei.reserve( prj_mat.getSide() - 1 );

  p0 = ray.getPnt( t0 );
  p1 = ray.getPnt( t1 );

  int_dist = math::dist( p0, p1 );

  new_spacing = calcNewSpacing(int_dist, spacing, &points );
  math::Vec2 new_pnt;

  ret += prj_mat.getWeights(p0, &cells );
  for( auto &cell: cells ) saveToMap( &pi_wei, cell, prj_mat.getSide(), .5*new_spacing );
  for( unsigned int i = 1; i < points; ++i ) {
    new_pnt = p0 + i*new_spacing*ray.getDir();
    ret += prj_mat.getWeights(new_pnt, &cells );

    for( auto &cell: cells ) saveToMap( &pi_wei, cell, prj_mat.getSide(), new_spacing );
  }
```

```
    ret += prj_mat.getWeights(p1, &cells);
    for(auto &cell: cells) saveToMap(&pi_wei, cell, prj_mat.getSide(), .5*new_spacing);
    if(pixel_weights) {
      pixel_weights->clear();
      pixel_weights->reserve(pi_wei.size());
      for(auto &p_w: pi_wei) {
        pixel_weights->emplace_back(std::make_pair(p_w.first, p_w.second));
      }
    }

    return ret;
}

void BilinearProjector::run()
{
    m_weights.clear();
    m_res.clear();
    m_weights.reserve(m_nr*m_na);
    m_res.reserve(m_nr*m_na);

    std::vector<double> prj_vals(m_nr);
    std::vector<std::pair<unsigned int, double>> ray_weights(m_nr);

    double step = m_prj_mat.getSide()/double(m_nr);
    double step_angle = M_PI/double(m_na);

    for(std::size_t as = 0; as < m_na; ++as) {
      double cur_angle = as*step_angle;
      for (std::size_t nr = 0; nr<m_nr; ++nr) {
        int j = int(nr) - int((m_nr >> 1));
        math::Ray ray(step*j, cur_angle);
        m_res.push_back(calcLineProjection(m_prj_mat, ray, &ray_weights));
        m_weights.emplace_back(SparseMatrix(m_mat.rows, m_mat.cols, ray_weights));
      }
    }
}
```

bilinear_based/bilinear_projector.hpp

```
//
// Created by markos on 19/3/22.
//

#ifndef IMAGE_PROJECTION_BILINEAR_PROJECTOR_HPP
#define IMAGE_PROJECTION_BILINEAR_PROJECTOR_HPP

#include "../common_utils/projector.hpp"

class BilinearProjector: public Projector {
  public:
    BilinearProjector(const cv::Mat &mat, std::size_t nr, std::size_t na);
    void run() override;

  protected:
    math::PrjMat m_prj_mat;
```

```
};

#endif //IMAGE_PROJECTION_BILINEAR_PROJECTOR_HPP
```

common_utils/fft.cpp

```
//
// Created by markos on 30/10/21.
//

#include "fft.hpp"
#include <vector>
#include <complex>

#include "general_utils.hpp"


using namespace fft;


int fft::doDFT(const std::vector<fcomplex> &in_data, std::vector<fcomplex> *out_data, bool do_inverse
    )
{
  std::size_t sz = in_data.size();
  out_data->resize( sz );
  std::fill( out_data->begin(), out_data->end(), fcomplex(0., 0.));

  int isign = ( do_inverse )? 1: -1;

  for( unsigned int i = 0; i < sz; ++i ) {
    for( std::size_t j = 0; j < sz; ++j ) {
      (*out_data)[i] += in_data.at(j)*std::polar<float>(1., float(isign)*2.*M_PI/float(sz)*float(j)*
     float(i));
    }
  }
  if( do_inverse ) {
    for( auto &num:*out_data ) {
      num /= float(out_data->size());
    }
  }
  return 0;
}


int reverseNumber(unsigned int lg_n, unsigned int num)
{
  int res = 0;
  for (int i = 0; i<lg_n; i++) {
    if (num & (1 << i))
      res |= 1 << (lg_n-1-i);
  }
  return res;
}

template <typename T>
static void constructBitReversedOrder( std::vector< T > *in_data )
```

```
{
  unsigned int power_of_2 = getPowerOf2( in_data->size() );
  unsigned int ss = in_data->size();
  for( unsigned int i = 0; i < ss; ++i ) {
    unsigned int reverse_num = reverseNumber( power_of_2, i );
    if( reverse_num > i ) {
      std::swap( (*in_data)[reverse_num], (*in_data)[i] );
    }
  }
}

//need to be optimized
//algorithm from https://cp-algorithms.com/algebra/fft.html and numerical recipes
int fft::doFFT_dl( const std::vector<fcomplex> &in_data, std::vector<fcomplex> *out_data, bool
    do_inverse = false )
{
  *out_data = in_data;
  float theta = 0.;

  constructBitReversedOrder(out_data);
  unsigned int nn = in_data.size();

  int isign = do_inverse + !do_inverse*(-1);
  fcomplex u, v;

  for( unsigned int len = 2; len <= nn; len <<=1 ) {
    theta = isign*2*M_PI/float(len);
    fcomplex exp_theta = std::polar<float>( 1., theta );
    for( unsigned int i = 0; i < nn; i+=len ) {
      fcomplex w(1.);
      for( unsigned int j = 0; j < len/2; ++j ) {
        u = out_data->at(i+j);
        v = w*out_data->at(i+j+len/2);
        (*out_data)[i+j]       = u + v;
        (*out_data)[i+j+len/2] = u - v;
        w *= exp_theta;
      }
    }
  }

  if( do_inverse ) {
    for( auto &num: *out_data ) {
      num /= float(nn);
    }
  }
  return 0;
}

static std::vector< fcomplex > partitionRealVector ( const std::vector< float > &in_data )
{
  std::vector< fcomplex > ret = std::vector< fcomplex >( in_data.size() >> 1, fcomplex(0., 0.) );
  for( unsigned int i = 0; i < in_data.size(); i+=2 ) {
    ret[i/2] = fcomplex( in_data[i], in_data[i+1] );
  }

  return ret;
```

```
}

int fft::doRealFFT_forward(const std::vector<float>& in_data, std::vector<fcomplex>*out_data)
{
  int ans = 0;
  auto in_data_complex = partitionRealVector(in_data);
  auto out_data_complex = std::vector<fcomplex>(in_data.size(), fcomplex(0., 0.));
  *out_data = std::vector<fcomplex>(in_data.size(), fcomplex(0., 0.));
  ans = fft::doFFT_dl(in_data_complex, &out_data_complex, false);

//  out_data_complex.push_back(out_data_complex.at(0));

  unsigned int sz_2 = in_data_complex.size();
  for(unsigned int i = 0; i <= sz_2; ++i) {
    unsigned int j = i & (~sz_2); // bit trick to avoid a push_back refer to numerical recipes page 619
    auto temp = std::conj(out_data_complex.at((sz_2 - i)&(~sz_2)));
    auto exp_angle = std::polar<float>(1.F, -2.0F*M_PI*float(i)/float(in_data.size()));
    (*out_data)[i] = 0.5F*((out_data_complex.at(j) + temp) - fcomplex(0, 1)*(out_data_complex.at
    (j) - temp)*exp_angle);
  }

  for(unsigned int i = 1; i < sz_2; ++i) {
    (*out_data)[sz_2 + i] = std::conj(out_data->at(sz_2 - i));
  }

  return 0;
}
```

common_utils/fft.hpp

```
//
// Created by markos on 30/10/21.
//

#ifndef FFT_HPP
#define FFT_HPP

#include <vector>
#include <complex>

namespace fft {
  using fcomplex = std::complex<float>;

  //Do FFT using Daniel-Lanczos lemma
  int doFFT_dl(const std::vector<fcomplex> &in_data, std::vector<fcomplex> *out_data, bool
    do_inverse);
  int doRealFFT_forward(const std::vector<float> &in_data, std::vector<fcomplex> *out_data);

  int doDFT(const std::vector<fcomplex> &in_data, std::vector<fcomplex> *out_data, bool
    do_inverse);
};


#endif //FFT_HPP
```

common_utils/fixed_point_utils.cpp

```
//
// Created by markos on 18/7/2022.
//

#include " fixed_point_utils .hpp"
```

common_utils/fixed_point_utils.hpp

```
//
// Created by markos on 18/7/2022.
//
#ifndef _FIXED_POINT_UTILS_HPP_
#define _FIXED_POINT_UTILS_HPP_
#include < cstdint >
#include <cmath>

namespace fp {
  template< typename INT_TYPE, int qr >
  INT_TYPE double2FP( double fl )
  {
    return (INT_TYPE)( fl *( 1 << qr ));
  }

  template< typename INT_TYPE, int qr >
  double FP2double( INT_TYPE n )
  {
    return ((double)n/(double)( 1 << qr ));
  }

//template< typename INT_TYPE, int qn1, int qn2, int qr >
//INT_TYPE mul( INT_TYPE n1, INT_TYPE n2 )
//{
//return n1*n2 >> ( qn1 + qn2 - qr );
//}

  template< typename INT_TYPE, int qn >
  INT_TYPE round( INT_TYPE z )
  {
    INT_TYPE ret = z >> ( qn - 1 );
    ret = ( ( ret & 0x1 ) + ret ) >> 1;
    return ret << qn;
  }

  template< typename INT_TYPE, int qn >
  INT_TYPE floor( INT_TYPE z )
  {
    INT_TYPE ret = z >> ( qn );
//      ret = ( ( ret & 0x1 ) + ret ) >> 1;
    return ret << qn;
  }

  template< typename INT_TYPE, int qn >
  INT_TYPE half()
  {
    return double2FP< INT_TYPE, qn >( 0.5 );
```

```
  }

  template< typename INT_TYPE, int qn >
  INT_TYPE ceil( INT_TYPE z )
  {
    INT_TYPE ret = floor< INT_TYPE, qn >( z );
    if( z > ret ) {
      ret += ( 1 << qn );
    }

    return ret;
  }

  template< typename INT_TYPE, int qn >
  INT_TYPE round_towards_zero( INT_TYPE z )
  {
    INT_TYPE ret;
    if( z > 0 ) {
      ret = floor< INT_TYPE, qn >( z );
    } else {
      ret = ceil< INT_TYPE, qn >( z );
    }

    return ret;
  }
}
#endif // _FIXED_POINT_UTILS_HPP_
```

common_utils/general_utils.cpp

```
//
// Created by markos on 5/11/21.
//

#include "general_utils.hpp"
unsigned int nextPowerOf2(unsigned int n)
{
  n--;
  n |= n >> 1;
  n |= n >> 2;
  n |= n >> 4;
  n |= n >> 8;
  n |= n >> 16;
  n++;
  return n;
}

unsigned int getPowerOf2( unsigned int n )
{
  unsigned int ret = 0;
  while ((1 << ret) < n)
        ret++;

  return ret;
}
```

common_utils/general_utils.hpp

```
//
// Created by markos on 5/11/21.
//

#ifndef INVERSE_RADON_TRANSFORM_GENERAL_UTILS_HPP
#define INVERSE_RADON_TRANSFORM_GENERAL_UTILS_HPP

unsigned int nextPowerOf2(unsigned int n);
unsigned int getPowerOf2(unsigned int n);

#endif //INVERSE_RADON_TRANSFORM_GENERAL_UTILS_HPP
```

common_utils/lut.hpp

```
//
// Created by markos on 2/8/2022.
//

#ifndef _LUT_HPP_
#define _LUT_HPP_

#include <array>

//code from https://joelfilho.com/blog/2020/compile_time_lookup_tables_in_cpp/

template<std::size_t Length, typename Generator>
constexpr auto lut(Generator&& f)
{
  using content_type = decltype(f(std::size_t{0}));
  std::array<content_type, Length> arr{};

  for(std::size_t i = 0; i < Length; i++) {
    arr[i] = f(i);
  }

  return arr;
}

#endif //_LUT_HPP_
```

common_utils/math_utils.cpp

```
//
// Created by markos on 12/3/22.
//
#include "math_utils.hpp"
#include <opencv2/core/mat.hpp>
#include <utility>
#include <numeric>
using namespace math;

Vec2::Vec2()
{
  m[0] = 0;
  m[1] = 0;
```

```
}

Vec2::Vec2(const double *v)
{
  m[0] = v[0];
  m[1] = v[1];
}

Vec2::Vec2( double x, double y )
{
  m[0] = x;
  m[1] = y;
}

Vec2& Vec2::operator=(const Vec2& rhs)
{
  if( this != &rhs ) {
    m[0] = rhs.m[0];
    m[1] = rhs.m[1];
  }

  return *this;
}

Vec2 Vec2::operator+(const Vec2& vec) const
{
  Vec2 ret;
  ret.m[0] = vec.m[0] + this->m[0];
  ret.m[1] = vec.m[1] + this->m[1];

  return ret;
}
Vec2 Vec2::operator-(const Vec2& vec) const
{
  Vec2 ret;
  ret.m[0] = vec.m[0] - this->m[0];
  ret.m[1] = vec.m[1] - this->m[1];

  return ret;
}
Vec2& Vec2::operator+=(const Vec2& rhs )
{
  this->m[0] += rhs.m[0];
  this->m[1] += rhs.m[1];

  return *this;
}
Vec2& Vec2::operator-=(const Vec2& rhs)
{
  this->m[0] -= rhs.m[0];
  this->m[1] -= rhs.m[1];

  return *this;
}

Vec2 &Vec2::operator*=(double t)
```

```
{
  m[0] *= t;
  m[1] *= t;
  return *this;
}

double* Vec2::getDat()
{
  return m;
}
double Vec2::x() const
{
  return m[0];
}
double Vec2::y() const
{
  return m[1];
}

double &Vec2::operator[]( size_t i )
{
  return m[i];
}

const double &Vec2::operator[]( size_t i ) const
{
  return m[i];
}
double Vec2::len_2() const
{
  return math::dot( *this, *this );
}

math::Vec2 operator*( double t, const math::Vec2 &vec )
{
  return { t*vec.x(), t*vec.y() };
}

Ray::Ray():
m_orig( 0, 0 ),
m_dir( 1, 0 )
{}

Ray::Ray(const Vec2 &orig, const Vec2& dir):
m_orig( orig ), m_dir( dir )
{
}

Ray::Ray( double s, double phi )
{
  using std::cos;
  using std::sin;
  m_orig[0] = cos(phi)+s*sin(phi);
  m_orig[1] = sin(phi)-s*cos(phi);

  m_dir[0] = cos(phi);
```

```
  m_dir[1] = sin(phi);
}

Vec2 Ray::getPnt(double t) const
{
  return m_orig + t*m_dir;
}
const Vec2& Ray::getDir() const
{
  return m_dir;
}
const Vec2& Ray::getOrig() const
{
  return m_orig;
}

void math:: intCircle ( const Ray &ray, double rad, double* t1, double* t2)
{
  double temp = dot( ray.getOrig(), ray.getDir() );
  double diak = std :: sqrt( temp - ray.getOrig().len_2() + rad*rad );

  if( diak < 0 ) {
    std :: cerr << "Ray does not intersect the circle " << std :: endl;
    assert( diak < 0 );
  }

  *t1 = -temp - diak;
  *t2 = -temp + diak;
}


double math::dot( const Vec2 &vec1, const Vec2 &vec2 )
{
  return vec1[0]*vec2[0] + vec1[1]*vec2[1];
}

double math:: dist ( const Vec2 &vec1, const Vec2 &vec2 )
{
  return std :: sqrt( ( vec1 - vec2 ).len_2() );
}

PrjMat :: PrjMat( cv :: Mat mat)
:m_mat(std::move( mat )), m_rad( 0.9999*( m_mat.rows/2 + .5 ) )
{}

double PrjMat :: getRad() const
{
  return m_rad;
}

double PrjMat :: getWeights(const Vec2& vec, CellArray* cells ) const
{
  double ret = 0.;
  Vec2 ov = vec + Vec2( m_mat.rows >> 1, m_mat.rows >> 1 );
  unsigned int i = std :: floor ( ov.x() );
  unsigned int j = std :: floor ( ov.y() );
```

```
  {
    Cell& cell = (*cells)[0];
    cell.m_i = i;
    cell.m_j = j;
    cell.m_w = (double(i+1)-ov.x())*(double(j+1)-ov.y());
  }

  {
    Cell& cell = (*cells)[1];
    cell.m_i = i+1;
    cell.m_j = j;
    cell.m_w = (ov.x()-double(i))*(double(j+1)-ov.y());
  }

  {
    Cell& cell = (*cells)[2];
    cell.m_i = i+1;
    cell.m_j = j+1;
    cell.m_w = (ov.x()-double(i))*(ov.y()-double(j));
  }
  {
    Cell& cell = (*cells)[3];
    cell.m_i = i;
    cell.m_j = j+1;
    cell.m_w = (double(i+1)-ov.x())*(ov.y()-double(j));
  }

   ret = std::accumulate(cells->begin(), cells->end(), 0., [this](double sum, const Cell &cell) {
    return sum + cell.m_w * m_mat.at<unsigned char>(cv::Point2i(int(cell.m_i), int(cell.m_j)));
  });

  return ret;
}

unsigned int PrjMat::getSide() const
{
  return m_mat.rows;
}

RayX::RayX(double a, double b):
Ray(Vec2(b, 0), Vec2(a, 1))
{
}
double RayX::getb() const
{
  return m_orig.x();
}

double RayX::geta() const
{
  return m_dir.x();
}
double RayX::getPntInt(double m) const
{
  return m_orig.x() + m_dir.x()*m;
```

```
}

RayY::RayY(double a, double b):
    Ray( Vec2( 0, b ), Vec2( 1, a ) )
{
}

double RayY::geta() const
{
  return m_dir.y();
}

double RayY::getb() const
{
  return m_orig.y();
}

double RayY::getPntInt(double m) const
{
  return m_orig.y() + m_dir.y()*m;
}
```

common_utils/math_utils.hpp

```
//
// Created by markos on 12/3/22.
//

#ifndef IMAGE_PROJECTION_MATH_UTILS_HPP
#define IMAGE_PROJECTION_MATH_UTILS_HPP

#include <opencv2/opencv.hpp>

namespace math{
  class Vec2;
}

math::Vec2 operator*( double t, const math::Vec2 &vec );
namespace math{
class Vec2;

  class Vec2 {
    public:
      Vec2();
      Vec2( const double * );
      Vec2( double x, double y );

      Vec2 &operator=( const Vec2 &rhs );
      Vec2 operator+( const Vec2 &vec ) const;
      Vec2 operator-( const Vec2 &vec ) const;
      Vec2 &operator+=( const Vec2 &vec );
      Vec2 &operator-=( const Vec2 &vec );
      Vec2 &operator*=( double t );
      double &operator[]( size_t i );
      const double &operator[]( size_t i ) const;
```

```
      [[nodiscard]] double len_2() const;
      friend Vec2 (::operator*)( double t, const Vec2 &vec );
//        friend Vec2 operator*( const Vec2 vec, double t );

      double *getDat();
      [[nodiscard]] double x() const;
      [[nodiscard]] double y() const;
    private:
      double m[2];
  };

  class Ray {
    public:
      Ray();
      Ray( const Vec2 &orig, const Vec2 &dir );
      Ray( double s, double phi );

      [[nodiscard]] Vec2 getPnt( double t ) const;
      [[nodiscard]] const Vec2 &getDir() const;
      [[nodiscard]] const Vec2 &getOrig() const;
    protected:
      Vec2 m_orig, m_dir;
  };

  class RayX: public Ray {
    public:
      RayX( double a, double b );

      [[nodiscard]]double geta() const;
      [[nodiscard]]double getb() const;

      double getPntInt( double m ) const;
  };

  class RayY: public Ray {
    public:
      RayY( double a, double b );

      [[nodiscard]]double geta() const;
      [[nodiscard]]double getb() const;

      double getPntInt( double m ) const;
  };

  struct Cell {
    public:
      unsigned int m_i, m_j;
      double m_w;
  };

  using CellArray = std::array< Cell, 4 >;
//PrjMat is a square with even elements
//PrjMat is an offseted mat by w/2, h/2
  class PrjMat {
    public:
      explicit PrjMat( cv::Mat mat );
```

```
        double getWeights( const Vec2 &vec, CellArray * cells ) const;
        [[nodiscard]] double getRad() const;
        [[nodiscard]] unsigned int getSide () const;

      private:
        cv::Mat m_mat;
        double m_rad;
    public:

    };

    double dot( const Vec2 &vec1, const Vec2 &vec2 );
    double dist ( const Vec2 &vec1, const Vec2 &vec2 );
    void intCircle ( const Ray &ray, double rad, double *t1, double *t2 );

}

#endif //IMAGE_PROJECTION_MATH_UTILS_HPP
```

common_utils/program_options.cpp

```
//
// Created by markos on 12/3/22.
//
#include "program_options.hpp"
#include <boost/program_options.hpp>
#include <iostream>

ProgramOptions::ProgramOptions(char** argv, size_t argc)
:m_argv(argv), m_argc(argc), m_num_rays(0), m_num_angles(0)
{
    this->readArgs();
}

const std::string& ProgramOptions::getIn() const
{
    return m_in;
}

const std::string& ProgramOptions::getWeightsFile() const
{
    return m_weights;
}

const std::string& ProgramOptions::getOut() const
{
    return m_out;
}

size_t ProgramOptions::getAnglesNum() const
{
    return m_num_angles;
}

std::size_t ProgramOptions::getRaysNum() const
{
```

```
  return m_num_rays;
}

void ProgramOptions::readArgs()
{
  namespace po = boost::program_options;
  po::options_description description("Usage");

  description.add_options()
      ("help,h", "Displays this help message")
      ("input_image,I", po::value<std::string>()->default_value(""), "Assign input image")
      ("output_image,O", po::value<std::string>()->default_value(""), "Assign output image")
      ("weights_data,W", po::value<std::string>()->default_value(""), "Assign weights output file")
      ("num_scans,N", po::value<unsigned int>()->default_value(360), "Number of scans")
      ("num_rays,R", po::value<unsigned int>()->default_value(0), "Number of rays");

  po::positional_options_description p;
  po::variables_map vm;
  p.add("input_image", -1);

  po::store(po::command_line_parser(m_argc, m_argv).
      options(description).positional(p).run(), vm);
  po::notify(vm);

  try {
    po::store(po::command_line_parser(m_argc, m_argv).options(description).run(), vm);
  } catch(...) {
    std::cerr << "Unrecognized option received, exiting" << std::endl;
    std::exit(1);
  }

  po::notify(vm);

  m_in = vm["input_image"].as<std::string>();
  m_out = vm["output_image"].as<std::string>();
  m_weights = vm["weights_data"].as<std::string>();
  m_num_angles = vm["num_scans"].as<unsigned int>();
  m_num_rays = vm["num_rays"].as<unsigned int>();
}
```

common_utils/program_options.hpp

```
//
// Created by markos on 12/3/22.
//

#ifndef IMAGE_PROJECTION_PROGRAM_OPTIONS_HPP
#define IMAGE_PROJECTION_PROGRAM_OPTIONS_HPP

#include <string>

class ProgramOptions {
public:
  ProgramOptions(char **m_argv, size_t m_argc);

  [[nodiscard]] const std::string& getIn() const;
```

```
  [[nodiscard]] const std::string& getOut() const;
  [[nodiscard]] const std::string& getWeightsFile() const;
  [[nodiscard]] std::size_t getAnglesNum() const;
  [[nodiscard]] std::size_t getRaysNum() const;
private:
  void readArgs();
private:
  char **m_argv;
  size_t m_argc;
  std::string m_in, m_out, m_weights;
  std::size_t m_num_angles;
  std::size_t m_num_rays;
};

#endif //IMAGE_PROJECTION_PROGRAM_OPTIONS_HPP
```

common_utils/math_utils.cpp

```
//
// Created by markos on 19/3/22.
//


#include <fstream>
#include "projector.hpp"

const std::vector<SparseMatrix>& Projector::getWeights() const
{
  return m_weights;
}
const std::vector<double>& Projector::getRes() const
{
  return m_res;
}

void Projector::writeToFile( const std::string &name) const
{
  using namespace std;
  std::ofstream fl(name, ios::out | ios::binary);

  fl << "size," << m_mat.rows << std::endl;
  fl << "num_rays," << m_nr << std::endl;
  fl << "num_angle," << m_na << std::endl;
  fl << "angle_lower," << 0 << std::endl;
  fl << "angle_upper," << M_PI << std::endl;
  fl << "weights" << std::endl;

  for(auto &sm: m_weights) {
    for(auto &inval: sm) {
//      fl << inval.first << "," << inval.second << ";";
      fl.write((char*)&inval.first, sizeof(unsigned int));
      fl.write((char*)&inval.second, sizeof(double));
    }
  }
}
void Projector::setPixelWeightFun(PixelWeightFun fun)
```

```
{
 m_fun = fun;
}
```

common_utils/math_utils.hpp

```
//
// Created by markos on 19/3/22.
//

#ifndef IMAGE_PROJECTION_PROJECTOR_HPP
#define IMAGE_PROJECTION_PROJECTOR_HPP

#include < utility >
#include <functional>
#include <opencv2/core/mat.hpp>
#include "math_utils.hpp"
#include "sparse_matrix.hpp"

struct PixelWeightData {
  int i, j;
  double weight;
  int na;
  int nr;
  const math::Ray *ray;
  double angle_step;
};

using PixelWeightFun = std :: function< void( const PixelWeightData & ) >;
class Projector {
  public:
    Projector ( const cv :: Mat &mat, std :: size_t nr, std :: size_t na ):
    m_mat( mat ), m_na( na ), m_nr( nr ) {}
    virtual void run() = 0;

    void writeToFile ( const std :: string &name ) const;
    void setPixelWeightFun( PixelWeightFun fun );

    [[nodiscard]] const std :: vector<SparseMatrix>& getWeights() const;
    [[nodiscard]] const std :: vector<double>& getRes() const;
  protected:
    cv :: Mat m_mat;
    std :: size_t m_nr, m_na;
    std :: vector< SparseMatrix > m_weights;
    PixelWeightFun m_fun;
public:
protected:
  std :: vector< double > m_res;
};

#endif //IMAGE_PROJECTION_PROJECTOR_HPP
```

common_utils/projector_fixed_point_nn.cpp

```
//
// Created by markos on 20/7/2022.
```

```
//
#include " projector_fixed_point_nn .hpp"
#include " trig_funcs_fixed_point .hpp"
#include " fixed_point_utils .hpp"

#define QN 12
using int_type = std :: int64_t ;

// int_type sin_fun( int_type z ) { return fp :: sin_lut < int_type , 10, QN, QN >( z ); }
// int_type cos_fun( int_type z ) { return fp :: cos_lut < int_type , 10, QN, QN >( z ); }
int_type tan_fun( int_type z ) { return fp :: tan_lut < int_type , 10, QN, QN >( z ); }
int_type cot_fun( int_type z ) { return fp :: cot_lut < int_type , 10, QN, QN >( z ); }
int_type csc_fun( int_type z ) { return fp :: csc_lut < int_type , 10, QN, QN >( z ); }
int_type sec_fun( int_type z ) { return fp :: sec_lut < int_type , 10, QN, QN >( z ); }

void ProjectorFPNN::run()
{
  // We will assume m_mat.cols == m_nr
  unsigned int side = m_mat.cols;
  int side_2 = (int) side >> 1;
  int_type step_angle = fp :: double2FP< int_type , QN >( 2./ double( m_na ) );
  int_type off = ( int_type ) m_nr >> 1;
  int_type upper = 0;
  int_type lower = 0;
  const int_type quarter = fp :: double2FP< int_type , QN >( .5 );

  PixelWeightData data = {0};
  data . angle_step = M_PI/ double( m_na );

  for( unsigned int as = 0; as < m_na; ++as ) {
    bool in_quarter ;
    int_type cur_angle = (int_type )as* step_angle ;
    int_type inv_num;
    int_type ratio ;
    data . na = (int )as ;
    if( cur_angle < quarter || cur_angle > ( 2 << QN ) - quarter ) {
      ratio = tan_fun( cur_angle );
      inv_num = sec_fun( cur_angle );
      in_quarter = true ;
    } else {
      ratio = cot_fun( cur_angle );
      inv_num = csc_fun( cur_angle );
      in_quarter = false ;
    }

    data . weight = fp :: FP2double< int_type , QN >(( inv_num < 0 )? -inv_num : inv_num );
//     data . weight = 1;

    for( unsigned int nr = 0; nr < m_nr; ++nr ) {
//       std :: cout << "Another Ray\n";
      data . nr = (int )nr ;
      int_type x, y;
      int_type a = ratio ;
      int_type s_off = (int_type )nr - off ;
      int_type b = ( (int_type ) ( in_quarter *( int_type )(-1) + ! in_quarter *1 ) )*( inv_num*s_off + (
     inv_num >> 1 ) );
```

```
      if( a != 0 ) {
        calcLowerUpperLimitsFP< int_type, QN >( a, b, &lower, &upper, side );
      } else {
        int ls = -int( side>>1 );
        int us =  int( side>>1 );
        lower = (int_type) ls*( 1 << QN ) + fp::half< int_type, QN >();
        upper = (int_type) us*( 1 << QN ) - fp::half< int_type, QN >();
      }

      int step_lim = (int)fp::round< int_type, QN >( upper - lower ) >> QN;

      if( in_quarter ) {
        for( int step = 0; step < step_lim; ++step ) {
          int_type m = ( fp::round_towards_zero< int_type, QN >( lower ) >> QN ) + ( step );
          x = m + ( side_2 );
          y = ( fp::floor< int_type, QN >( ( a*m ) + b - fp::half< int_type, QN >() ) >> QN ) +
    side_2 ;

          data.i = (int) ( x );
          data.j = (int) ( y );
          if( !( 0 <= data.j  && data.j < 1024 ) ) {
            std::cout << "I am outside x:\t" << data.i << "\ty:\t" << data.j << "\n";
          } else {
            if( m_fun ) {
              m_fun( data );
            }
          }
        }
      } else {
        for( int step = 0; step < step_lim; ++step ) {
          int_type m = ( fp::round_towards_zero< int_type, QN >( lower ) >> QN ) + ( step );
          y = m + ( side_2 );
          x = ( fp::floor< int_type, QN >(( a*m ) + b - fp::half< int_type, QN >()) >> QN ) + side_2;

          data.i = (int)( x );
          data.j = (int)( y );
          if( !( 0 <= data.j && data.j < 1024 )) {
            std::cout << "I am outside x:\t" << data.i << "\ty:\t" << data.j << "\n";
          } else {
            if( m_fun ) {
              m_fun( data );
            }
          }
        }
      }
    }
  }
}
```

common_utils/projector_fixed_point_nn.hpp

```
//
// Created by markos on 20/7/2022.
//
#ifndef _PROJECTOR_FIXED_POINT_NN_HPP_
#define _PROJECTOR_FIXED_POINT_NN_HPP_
```

```
#include "projector.hpp"
#include "fixed_point_utils.hpp"
#include <cmath>

class ProjectorFPNN: public Projector {
  public:
      using Projector::Projector;
    void run() override;

};

template< typename INT_TYPE, int qn >
INT_TYPE calcLineVal( INT_TYPE a, INT_TYPE b, INT_TYPE m )
{
  return ( fp::floor< INT_TYPE, qn >( ( a*(m) ) ) + b ) >> qn;
}

template< typename INT_TYPE, int qn >
void correctLowerLimit_a_pos( INT_TYPE a, INT_TYPE b, INT_TYPE *ret, INT_TYPE lower_scaled )
{
  INT_TYPE y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  while( y < lower_scaled ) {
    (*ret) += ( 1 << qn );
    y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  }
}

template< typename INT_TYPE, int qn >
void correctUpperLimit_a_pos( INT_TYPE a, INT_TYPE b, INT_TYPE *ret, INT_TYPE upper_scaled )
{
  INT_TYPE y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  while( y > upper_scaled ) {
    (*ret) -= ( 1 << qn );
    y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  }
}

template< typename INT_TYPE, int qn >
void correctLowerLimit_a_neg( INT_TYPE a, INT_TYPE b, INT_TYPE *ret, INT_TYPE upper_scaled )
{
  INT_TYPE y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  while( y > upper_scaled ) {
    (*ret) += ( 1 << qn );
    y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  }
}

template< typename INT_TYPE, int qn >
void correctUpperLimit_a_neg( INT_TYPE a, INT_TYPE b, INT_TYPE *ret, INT_TYPE lower_scaled )
{
  INT_TYPE y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  while( y < lower_scaled ) {
    (*ret) -= ( 1 << qn );
    y = calcLineVal< INT_TYPE, qn >( a, b, *ret );
  }
}
```

```
// Calculates the lower and upper limits a ray intersects with a square with size = side
// The ray is of type y = a*x + b where a is the dir and b is the offset
template< typename INT_TYPE, int qn >
void calcLowerUpperLimitsFP (INT_TYPE a, INT_TYPE b, INT_TYPE *lower, INT_TYPE *upper, unsigned int
    side )
{
  using std::max;
  using std::min;

  int ls = -int( side>>1 );
  int us = int( side>>1 );
  INT_TYPE y;
  INT_TYPE ls_scaled = (INT_TYPE) ls * ( 1 << 2*qn ) + fp::half< INT_TYPE, 2*qn >();
  INT_TYPE us_scaled = (INT_TYPE) us * ( 1 << 2*qn ) - fp::half< INT_TYPE, 2*qn >();

  *lower = ls_scaled;
  *upper = us_scaled;
  if( a > 0 ) {
    *lower = max( ls_scaled >> qn, (( ls_scaled - ( b << qn ) )/a) );
    *upper = min( us_scaled >> qn, (( us_scaled - ( b << qn ) )/a) );
    correctLowerLimit_a_pos< INT_TYPE, qn >( a, b, lower, ls_scaled >> qn );
    correctUpperLimit_a_pos< INT_TYPE, qn >( a, b, upper, us_scaled >> qn );

  } else {
    *lower = max( ls_scaled >> qn, (( us_scaled - ( b << qn ) )/a) );
    *upper = min( us_scaled >> qn, (( ls_scaled - ( b << qn ) )/a) );
    correctLowerLimit_a_neg< INT_TYPE, qn >( a, b, lower, us_scaled >> qn );
    correctUpperLimit_a_neg< INT_TYPE, qn >( a, b, upper, ls_scaled >> qn );
  }

  *lower = fp::ceil< INT_TYPE, qn >( *lower );
  *upper = fp::floor< INT_TYPE, qn >( *upper );
}

#endif // _PROJECTOR_FIXED_POINT_NN_HPP_
```

common_utils/projector_nn.cpp

```
//
// Created by markos on 20/7/2022.
//
#include "projector_nn.hpp"
#include <opencv2/core/mat.hpp>

void calcLowerUpperLimits(double a, double b, double *lower, double *upper, unsigned int side)
{
  double ls = -int(side>>1) + .5;
  double us = int(side>>1) - .5;

  if( std::abs( a ) <= std::numeric_limits<decltype(a)>::epsilon() ) {
    *lower = ls;
    *upper = us;
  } else if( a > 0 ) {
    *lower = std::max( ls, ( ( ls - b )/double(a) ) );
    *upper = std::min( us, ( ( us - b )/double(a) ) );
```

```
  } else {
    *lower = std::max( ls, ( ( us - b )/double(a) ) );
    *upper = std::min( us, ( ( ls - b )/double(a) ) );
  }

  assert( ls <= *lower );
  assert( us >= *upper );
}

//both limits must be offseted
static void calcLowerUpperLimitsRay( const math::RayY &ray, double *lower, double *upper, unsigned
    int side )
{
  calcLowerUpperLimits( ray.geta(), ray.getb(), lower, upper, side );
}

static void calcLowerUpperLimitsRay( const math::RayX &ray, double *lower, double *upper, unsigned
    int side )
{
  calcLowerUpperLimits( ray.geta(), ray.getb(), lower, upper, side );
}

void ProjectorNN::run()
{
//  using namespace std;
  using std::sin;
  using std::cos;

  unsigned int side = m_mat.cols;
  int side_2 = (int)( side >> 1 );
//  math::Vec2 transf_vec( side >> 1, side >> 1 );
//  double step_ray = double(m_mat.cols)/double(m_nr);
  double step_angle = M_PI/double( m_na );
  int off = (int) m_nr/2;
  double mult = double(m_mat.cols)/double(m_nr);
//  double add = (!( m_nr&0x1))*.5;
  double upper = 0.;
  double lower = 0.;
//  std::pair< int, int > pos;
  PixelWeightData data = {0};

  data.angle_step = step_angle;

  for( std::size_t as = 0; as < m_na; as += 1 ) {
    double cur_angle = (double) as*step_angle;
    for( std::size_t nr = 0; nr < m_nr; ++nr ) {
      double sina = sin( cur_angle );
      double cosa = cos( cur_angle );
      double s_off = ( int(nr) - off + .5 ) * mult;
      data.nr = (int)nr;
      data.na = (int)as;
      if( sina <= M_SQRT1_2 ){
        math::RayY rayy( sina/cosa, -s_off/cosa );
        calcLowerUpperLimitsRay( rayy, &lower, &upper, side );
        data.weight = 1./std::abs( cosa );
        int step_lim = ( int )round( upper - lower );
```

```
        for( int step = 0; step <= step_lim; ++step ) {
          double m = lower + step;
          double y = rayy.getPntInt( m );
          data.i = std::floor(m + side_2 );
          data.j = std::floor(y + side_2 );
          data.ray = &rayy;
          if( m_fun ) {
            m_fun( data );
          }

        }
      } else {
        math::RayX rayx( cosa/sina, s_off/sina );
        calcLowerUpperLimitsRay( rayx, &lower, &upper, side );
        data.weight = 1./std::abs( sina );
        int step_lim = (int)round( upper - lower );
        for( int step = 0; step <= step_lim; ++step ) {
          double m = lower + step;
          double x = rayx.getPntInt( m );
          data.i = std::floor( x + side_2 );
          data.j = std::floor( m + side_2 );
          data.ray = &rayx;
          if( m_fun ) {
            m_fun( data );
          }
        }
      }
    }
  }

}
```

common_utils/projector_nn.hpp

```
//
// Created by markos on 20/7/2022.
//
#ifndef _PROJECTOR_NN_HPP_
#define _PROJECTOR_NN_HPP_

#include "projector.hpp"

//Nearest neighbour projector
class ProjectorNN: public Projector {
 public:
  using Projector::Projector;
  void run() override;
};

void calcLowerUpperLimits( double a, double b, double *lower, double *upper, unsigned int side );

#endif // _PROJECTOR_NN_HPP_
```

common_utils/sparse_matrix.cpp

```
//
```

```
// Created by markos on 19/3/22.
//

#include <limits>
#include <algorithm>
#include "sparse_matrix.hpp"

SparseMatrix::SparseMatrix(std::size_t rows, std::size_t columns,
    const std::unordered_map<unsigned int, double>& sparse_vals):
    m_rows(rows), m_cols(columns)
{
  m_index_vals.insert(m_index_vals.begin(), sparse_vals.begin(), sparse_vals.end());
  std::sort(m_index_vals.begin(), m_index_vals.end(), compFunc);
}

SparseMatrix::SparseMatrix(std::size_t rows, std::size_t columns,
    const std::vector<std::pair<unsigned int, double>>& sparse_vals):
    m_rows(rows), m_cols(columns)
{
  m_index_vals.insert(m_index_vals.begin(), sparse_vals.begin(), sparse_vals.end());
  std::sort(m_index_vals.begin(), m_index_vals.end(), compFunc);
}

bool SparseMatrix::compFunc(const SparseMatrix::InVal& val1, const SparseMatrix::InVal& val2)
{
  return val1.first < val2.first;
}
void SparseMatrix::addVal(std::size_t row, std::size_t column, double val)
{
  std::size_t index = row*m_rows + column;

  InVal temp{index, val};
  auto it = std::lower_bound(m_index_vals.begin(), m_index_vals.end(), temp, compFunc);

  if(it->first != index) {
    m_index_vals.insert(it, temp);
  } else {
    it->second += val;
  }
}
double SparseMatrix::getVal(std::size_t in) const
{
  double ret = 0.;
  InVal temp{in, 0.};
  auto it = std::lower_bound(m_index_vals.begin(), m_index_vals.end(), temp, compFunc);

  if(it->first != in) {
    ret = std::numeric_limits<double>::max();
  } else {
    ret = it->second;
  }

  return ret;
}
double SparseMatrix::getVal(std::size_t row, std::size_t column) const
{
```

```
  return this->getVal( row*m_rows + column );
}

std::vector<SparseMatrix::InVal>::iterator SparseMatrix::begin()
{
  return m_index_vals.begin();
}
std::vector<SparseMatrix::InVal>::iterator SparseMatrix::end()
{
  return m_index_vals.end();
}
std::vector<SparseMatrix::InVal>::const_iterator SparseMatrix::begin() const
{
  return m_index_vals.cbegin();
}

std::vector<SparseMatrix::InVal>::const_iterator SparseMatrix::end() const
{
  return m_index_vals.cend();
}
```

common_utils/sparse_matrix.hpp

```
//
// Created by markos on 19/3/22.
//

#ifndef IMAGE_PROJECTION_SPARSE_MATRIX_HPP
#define IMAGE_PROJECTION_SPARSE_MATRIX_HPP

#include <utility>
#include <vector>
#include <unordered_map>

class SparseMatrix {
  public:
    using InVal = std::pair<std::size_t, double>;
    SparseMatrix( std::size_t rows, std::size_t columns, const std::unordered_map<unsigned int,
     double> &sparse_vals );
    SparseMatrix( std::size_t rows, std::size_t columns, const std::vector<std::pair<unsigned int,
     double>> &sparse_vals );

    void addVal( std::size_t row, std::size_t column, double val ); //slow function try to avoid

    [[nodiscard]] double getVal( std::size_t in ) const;
    [[nodiscard]] double getVal( std::size_t row, std::size_t column ) const;

    std::vector<InVal>::iterator begin();
    std::vector<InVal>::iterator end();

    [[nodiscard]] std::vector<InVal>::const_iterator begin() const;
    [[nodiscard]] std::vector<InVal>::const_iterator end() const;
private:
    static bool compFunc( const InVal &val1, const InVal &val2 );
  private:
    std::vector<InVal> m_index_vals;
```

```
    std :: size_t  m_rows, m_cols;

};

#endif  // IMAGE_PROJECTION_SPARSE_MATRIX_HPP
```

pixel_based/pixel_based_simple.cpp

```
//
//  Created by markos on 29/3/22.
//
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include " ../ common_utils/program_options.hpp"
#include " ../ common_utils/projector_nn.hpp"

int main( int argc, char **argv ) {
  ProgramOptions po( argv, argc );
  const std :: string &input = po.getIn () ;
  const std :: string &output = po.getOut() ;
  unsigned int num_scans = po.getAnglesNum();
  unsigned int nr        = po.getRaysNum();

  std :: cout << "Input file : " << input. c_str () << std :: endl;
  std :: cout << "Output file : " << output. c_str () << std :: endl;
  cv :: Mat image = cv :: imread(input, cv :: IMREAD_GRAYSCALE);
  if ( !nr ) {
    nr = image.cols ;
  }

  if ( image.empty() ) {
    std :: cerr << "Could not find file " << std :: endl;
    exit (1) ;
  }
  cv :: Mat padded_image;

  cv :: Mat sinogram( cv :: Size ( nr, num_scans ), CV_32F );
  ProjectorNN projector ( image, nr, num_scans );

  projector . setPixelWeightFun([&sinogram, &image]( const PixelWeightData &pw ) {
    sinogram.at<float >( pw.na, pw.nr ) +=
            pw.weight*double( image.at<unsigned char >( pw.i, pw.j ) ) ;
  } ) ;

  projector . run () ;


  cv :: Mat sin_normalized;
  cv :: normalize( sinogram, sin_normalized, 1., 0., cv :: NORM_INF );

  cv :: imshow("", sin_normalized ) ;

  cv :: Mat out_mat;
  sin_normalized . convertTo(out_mat, CV_8U, 255 ) ;
  cv :: imwrite( output, out_mat ) ;
```

```
  cv :: waitKey() ;

  return 0;
}
```

pixel_based/pixel_based_w_rotate.cpp

```
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include " ../ common_utils/program_options.hpp"

void addColumnsNAddItToSinogram( const cv::Mat &mat, int num_row, cv::Mat *p_mat );

int main( int argc, char **argv ) {
  ProgramOptions po( argv, argc );
  const std :: string &input = po. getIn () ;
  const std :: string &output = po.getOut() ;
  unsigned int num_scans = po.getAnglesNum();

  std :: cout << "Input file : " << input. c_str () << std :: endl;
  std :: cout << "Output file : " << output. c_str () << std :: endl;
  cv :: Mat image = cv :: imread(input, cv :: IMREAD_GRAYSCALE);

  if ( image.empty() ) {
    std :: cerr << "Could not find file " << std :: endl;
    exit (1) ;
  }
  cv :: Mat padded_image;

  cv :: Mat sinogram;

  auto s = image. size () ;
  int border_type = cv :: BORDER_CONSTANT;
  float diag_len = std :: sqrt ( float ( s.width*s.width + s.height*s.height ) ) ;
  int width_pad = signed( std :: ceil ( diag_len - float ( s.width ) ) )/2 + 1;
  int height_pad = signed( std :: ceil ( diag_len - float ( s.height ) ) )/2 + 1;

  cv :: copyMakeBorder( image, padded_image, height_pad, height_pad, width_pad, width_pad, border_type,
              cv :: Scalar ( 0, 0, 0 ) ) ;

  sinogram = cv :: Mat( cv :: Size ( padded_image.size() .width, int (num_scans) ), CV_32F ) ;
  for ( unsigned int i = 0; i < (num_scans); ++i ) {
    cv :: Mat rotated_image;
    double angle_incr = i * 180./( num_scans);
    cv :: Mat rot_mat = cv :: getRotationMatrix2D(
        cv :: Point2d( ( padded_image.rows - 1 ) /2., ( padded_image.cols - 1 ) /2.) ,
        90 - angle_incr , 1. ) ;

    cv :: warpAffine( padded_image, rotated_image, rot_mat, padded_image.size () ) ;

    addColumnsNAddItToSinogram( rotated_image, int(i), &sinogram );
  }
  cv :: Mat sin_normalized;
  cv :: normalize( sinogram, sin_normalized, 1., 0., cv :: NORM_INF ) ;

  cv :: imshow("", sin_normalized ) ;
```

```
  cv::Mat out_mat;
  sin_normalized.convertTo(out_mat, CV_8U, 255);
  cv::imwrite( output, out_mat );

  imshow( "", out_mat );
  cv::waitKey();

  return 0;
}

void addColumnsNAddItToSinogram( const cv::Mat &mat, int num_row, cv::Mat *p_mat )
{
  cv::Size s = mat.size();
  for( int i = 0; i < s.width; ++i ) {
    unsigned int sum = 0;
    for( int j = 0; j < s.height; ++j ) {
      sum += mat.at< unsigned char >( i, j );
    }

    p_mat->at<float>( num_row, i ) = float(sum)/float(s.area());
  }
}
```

Αναφορές

[1] Avinash C. Kak και Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial και Applied Mathematics, 2001. DOI: 10.1137/1.9780898719277.

[2] Gensheng Lawrence Zeng. *Medical Image Reconstruction*. Springer.

[3] Wikipedia. *Compton scattering*. Last edited: 15 May 2022. URL: https://en.wikipedia.org/wiki/Compton_scattering.

[4] *Απορρόφηση*. https://hmn.wiki/el/Absorbance.

[5] Cristian Venanzi. «Detection systems for medical imaging with synchrotron radiation». Στο: (Αύγ. 2022).

[6] CT Brian Nett PhD/Physics. *Animated CT Generations [1st, 2nd, 3rd, 4th, 5th Gen CT] for Radiologic Technologists*. URL: https://howradiologyworks.com/ctgenerations/#1st-generation.

[7] I. Arimura. «Photoabsorption effects in low temperature electron-irradiated Germanium». Στο: *IEEE Transactions on Nuclear Science* 21.6 (Δεκ. 1974), σσ. 21–25. DOI: 10.1109/TNS.1974.6498900.

[8] William H. Press, Saul A. Teukolsky, William T. Vetterling και Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3η έκδοση. USA: Cambridge University Press, 2007. ISBN: 0521880688.

[9] Jasper Vijn. *Another fast fixed-point sine approximation*. URL: https://www.coranac.com/2009/07/sines/.