

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Ακαδημαϊκό Έτος: 2019-2020



Τεχνολογία Και Ανάλυση Εικόνων και Βίντεο

Άσκηση 1

Θέμα: Εκτίμηση κίνησης

Τζε Χριστίνα-Ουρανία		03116079
Ψαρουδάκης Ανδρέας		03116001

30 Μαΐου 2020

# Εισαγωγή

Σκοπός του δεύτερου μέρους της εργαστηριακής αυτής άσκησης είναι η υλοποίησης ενός συστήματος ανίχνευσης χαρακτηριστικών και εκτίμησης κίνησης σε βίντεο, με εφαρμογή στο χώρο ενός parking (όπου εμφανίζεται κινητικότητα ανθρώπων και αυτοκινήτων). Συγκεκριμένα, καλούμαστε να χρησιμοποιήσουμε τεχνικές εξαγωγής χαρακτηριστικών για τον εντοπισμό των αντικειμένων που εμφανίζονται στο βίντεο αλλά και να εφαρμόσουμε τον αλγόριθμο των Lucas-Kanade για τον υπολογισμό του διανυσματικού πεδίου οπτικής ροής, μέσω του οποίου επιτυγχάνεται η παρακολούθηση της κίνησης μεταξύ διαδοχικών εικόνων (χαρέ) του βίντεο.

## 1 Resize των εικόνων του βίντεο στη μισή ανάλυση

Αρχικά, υποδειγματολειπτούμε κάθε frame του βίντεο που διαβάζουμε, μετατρέποντάς το στη μισή ανάλυση από την αρχική μέσω της συνάρτησης **resize** της βιβλιοθήκης **cv2**:

```
frame_rs = cv2.resize(frame, (int(frame.shape[1]/2), int(frame.shape[0]/2)))
```

## 2 Επιλογή χρωματικού χώρου

Κάθε frame του βίντεο διαβάζεται ως έγχρωμη εικόνα μέσω της συνάρτησης **cap.read()**. Στη συνέχεια αυτή γίνεται resize στη μισή ανάλυση, όπως αναφέραμε στο προηγούμενο βήμα, και έπειτα μετατρέπεται σε μονοκάναλη grayscale για επεξεργασία. Αυτό πρακτικά συμβαίνει για λόγους βελτιστοποίησης καθώς είναι πολύ ταχύτερη η εφαρμογή αλγορίθμων εξαγωγής χαρακτηριστικών πάνω σε μια εικόνα με 1 κανάλι σε σχέση με μια τρικάναλη. Το τελικό αποτέλεσμα της ανίχνευσης τόσο των σημείων ενδιαφέροντος όσο και της εκτίμησης κίνησης παρουσιάζεται πάνω στα έγχρωμα frames για καλύτερη οπτικοποίηση. Η μετατροπή κάθε χαρέ του βίντεο σε grayscale υλοποιείται μέσω της συνάρτησης **cvtColor** της βιβλιοθήκης **cv2**:

```
gray = cv2.cvtColor(frame_rs, cv2.COLOR_BGR2GRAY)
```



### 3 Εφαρμογή Harris και Shi-Tomasi corner detectors στο πρώτο frame του βίντεο

Έχοντας πλέον μετατρέψει το πρώτο frame σε grayscale μπορούμε να εφαρμόσουμε τους αλγορίθμους ανίχνευσης γωνιών Harris και Shi-Tomasi για τον εντοπισμό των αντικειμένων του πρώτου καρέ του βίντεο. Τόσο ο ανιχνευτής Harris όσο και ο Shi-Tomasi υλοποιούνται μέσω της συνάρτησης **goodFeaturesToTrack** της βιβλιοθήκης **cv2**. Το είδος του ανιχνευτή που θα χρησιμοποιηθεί καθορίζεται από την παράμετρο **useHarrisDetector**, η οποία by default έχει την τιμή `False`, με αποτέλεσμα αν δεν προσδιοριστεί να καλείται ο ανιχνευτής Shi-Tomasi. Για την εφαρμογή του Harris θα πρέπει να θέσουμε την παράμετρο στην τιμή `True`. Θέλουμε να πετύχουμε μια όσο το δυνατόν καλύτερη ‘περιγραφή’ των αντικειμένων της εικόνας εντοπίζοντας γωνίες πάνω σε αυτά. Οι γωνίες ορίζονται ως σημεία που είναι αμετάβλητα στη μετάφραση, την κλίμακα, την περιστροφή και τις αλλαγές έντασης. Οι συντεταγμένες των γωνιών που ανιχνεύονται βρίσκονται στον πίνακα `prev`. Για την επισημείωσή τους αρχικοποιούμε δύο κενές λίστες, τις `x_coord` και `y_coord`, που περιέχουν αντίστοιχα την τετμημένη και τεταγμένη τους. Διατρέχουμε τον πίνακα `prev` προσθέτοντας κάθε φορά στην αντίστοιχη λίστα την `x` και `y` συντεταγμένη. Έπειτα, με χρήση της συνάρτησης **zip** συνενώνουμε τις δύο λίστες σε μία ενιαία, την `coords`, η οποία λαμβάνει το σύνολο των συντεταγμένων (`x,y`) των σημείων ενδιαφέροντος. Στην συνέχεια, σημειώνουμε τα σημεία αυτά πάνω στο πρώτο frame της ακολουθίας του βίντεο με χρήση της συνάρτησης **circle** της βιβλιοθήκης **cv2**.

```
prev = cv.goodFeaturesToTrack(prev_gray, mask = None, **feature_params)

x_coord = []
y_coord = []

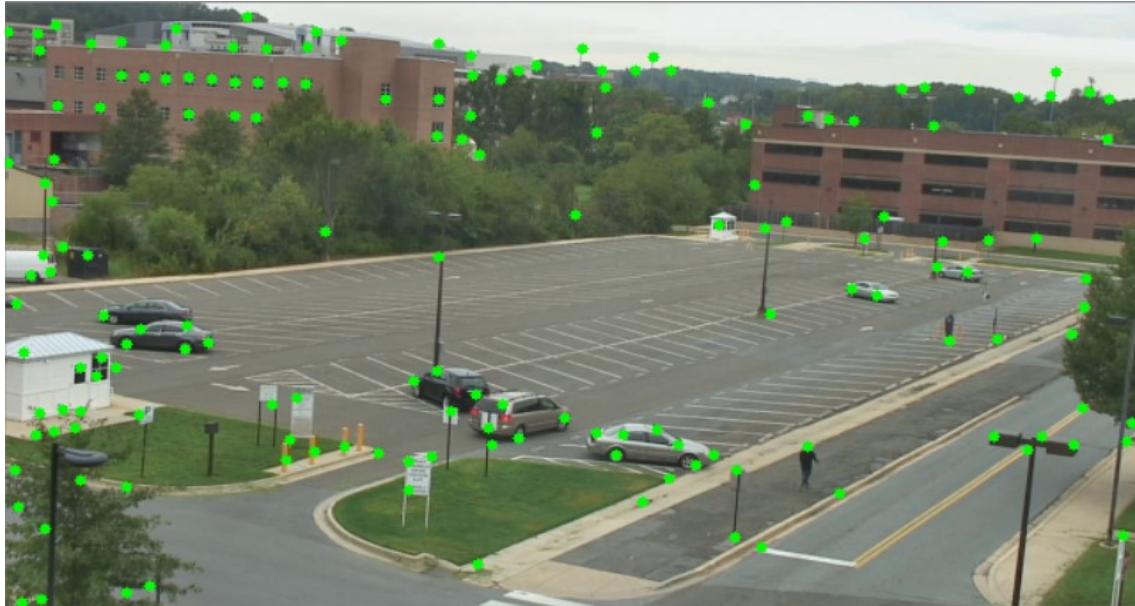
for i in range(prev.shape[0]):
    x_coord.append(prev[i][0][0])
    y_coord.append(prev[i][0][1])

coords = zip(x_coord, y_coord)

for x, y in coords:
    cv2.circle(first_frame_rs, (x, y), 3, (0, 255, 0), -1)
```

Δοκιμάζοντας ορισμένες εκτελέσεις του αλγορίθμου παρατηρούμε ότι εμφανίζονται γωνίες στα παράθυρα των κτηρίων πίσω από το parking, στους στύλους φωτισμού, στα δέντρα, στο φανάρι, στις πινακίδες, στα αυτοκίνητα καθώς και στους ανθρώπους.

Ένα παράδειγμα τέτοιας ανίχνευσης φαίνεται στην παρακάτω εικόνα:

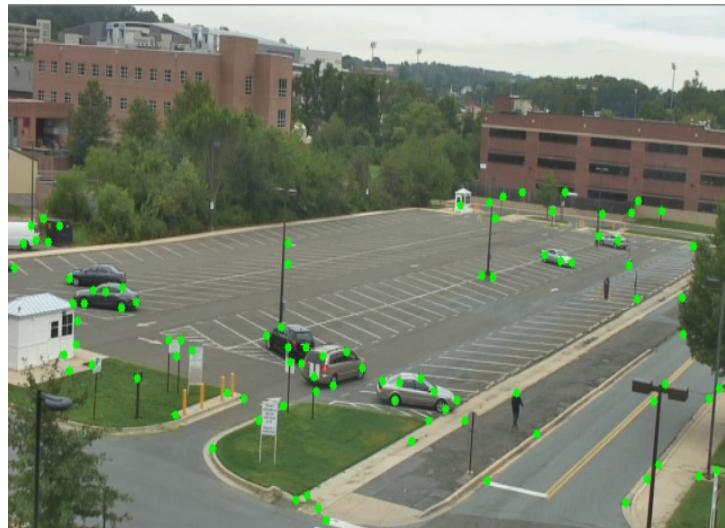


Corner detection on first frame

Η παραπάνω ανίχνευση φαίνεται ικανοποιητική, ωστόσο εμείς ιδανικά θα θέλαμε να ανιχνεύσουμε σημεία ενδιαφέροντος (γωνίες) μόνο σε ανθρώπους και αυτοκίνητα καθώς αυτά θα αποτελέσουν τα ‘αντικείμενα’ κίνησης στο βίντεο μας. Όλα τα υπόλοιπα μη-κινούμενα αντικείμενα της εικόνας μας είναι ουσιαστικά αδιάφορα για το tracking. Αξιοποιούμε λοιπόν την παράμετρο **mask** της συνάρτησης **goodFeaturesToTrack** η οποία μας επιτρέπει να ορίζουμε εμείς τις περιοχές ενδιαφέροντος όπου θέλουμε να γίνει η ανίχνευση των γωνιών. Αποκλείουμε λοιπόν τεχνητά πολλές από τις μη-κινούμενες περιοχές που προαναφέραμε, κατασκευάζοντας μια δισδιάστατη μάσκα **mask2D** με τις διαστάσεις της graysacle εικόνας, η οποία έχει σε όλα τα pixels της 1 (άσπρο) εκτός από αυτά στα οποία επιθυμούμε να μην γίνει ανίχνευση, τα οποία και θέτουμε ίσα με 0 (μαύρο). Έτσι, το σύνολο των ανιχνευθέντων σημείων περιορίζεται στο τμήμα εκείνο της εικόνας που φαίνεται να περιέχει τη σημαντικότερη πληροφορία αναφορικά με την κίνηση. Αυτό μάλιστα καθιστά τον αλγόριθμό μας πιο αποδοτικό αφού μπορούμε πλέον να χρησιμοποιήσουμε μικρότερο μέγιστο αριθμό από corners (παράμετρος **maxCorners**) για τον εντοπισμό των αντικειμένων που μας ενδιαφέρουν (καθώς δεν θα εντοπίζονται πλέον γωνίες σε αδιάφορες περιοχές-π.χ δέντρα). Η μάσκα αυτή υλοποιείται μέσω της συνάρτησης **create\_mask** που ορίζουμε στο βοηθητικό αρχείο **mask.py**. Η συνάρτηση αυτή δέχεται ως όρισμα ένα resized frame και επιστρέφει τόσο την δισδιάστατη μάσκα **mask2D** που περιγράψαμε όσο και μια τρισδιάστατη εικόνα **mask3D** η οποία οπτικοποιεί την εφαμοργή της μάσκας πάνω στην εικόνα, απεικονίζοντας με άσπρο χρώμα τα pixels εκείνα του frame τα οποία αποκλείουμε από την διαδικασία της ανίχνευσης γωνιών.

Η εφαρμογή της μάσκας στην προηγούμενη ανίχνευση γωνιών (διατηρώντας σταθερές τις υπόλοιπες παραμέτρους της `goodFeaturesToTrack`) καθώς και η οπτικοποίηση της μάσκας φαίνονται στις ακόλουθες εικόνες:

```
prev = cv.goodFeaturesToTrack(prev_gray, mask = mask2D, **feature_params)
```



Corner detection with mask applied



Mask Visualisation

Έχοντας λοιπόν εστιάσει σε συγκεκριμένες περιοχές της εικόνας, με χρήση κατάλληλης μάσκας, πειραματίζόμαστε τώρα με διαφορετικές τιμές των ακόλουθων παραμέτρων της συνάρτησης `goodFeaturesToTrack`:

- **maxCorners**: Μέγιστος αριθμός γωνιών προς ανίχνευση. Όσο μεγαλύτερη τιμή λαμβάνει τόσο περισσότερες γωνίες ενδέχεται να επιστρέψει η συνάρτηση.
- **qualityLevel**: Κατώφλι για την επιλογή των γωνιών. Γωνίες με quality κάτω από αυτό το κατώφλι απορρίπτονται. Συνεπώς, όσο μικρότερη η τιμή της παραμέτρου τόσο πιο ‘ελαστικό’ είναι το κριτήριο ανίχνευσης και κατά συνέπεια τόσο περισσότερες γωνίες επιτρέπεται να ανιχνευτούν. Από την άλλη, μεγάλα κατώφλια λειτουργούν πολύ περιοριστικά.
- **minDistance**: Ελάχιστη απόσταση μεταξύ δύο γωνιών που ανιχνεύονται

Ακολουθούν τα αποτελέσματα για όλες τις παραμετρικοποιήσεις που δοκιμάσαμε τόσο για τον Harris όσο και για τον Shi-Tomasi detector:

- **maxCorners = 300** , qualityLevel = 0.12, minDistance = 20



Harris detector



Shi-Tomasi detector

- **maxCorners = 150** , qualityLevel = 0.12, minDistance = 20



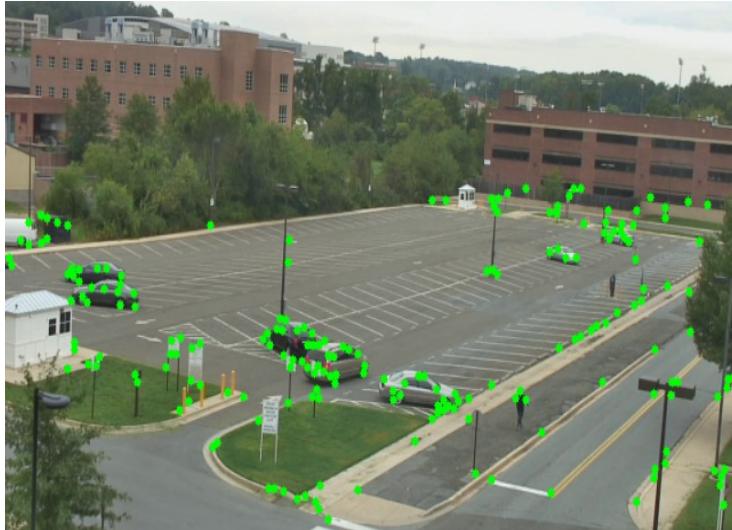
Harris detector



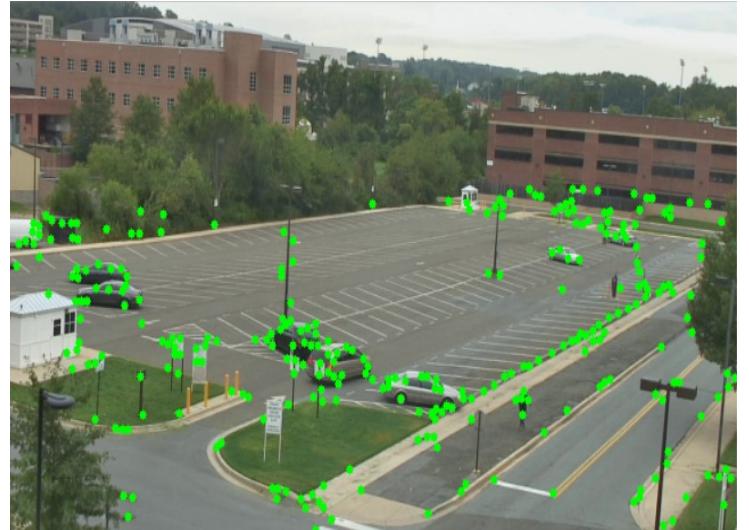
Shi-Tomasi detector

Παρατηρούμε ότι και στις δύο περιπτώσεις ο ανιχνευτής Shi-Tomasi λειτουργεί αρκετά πιο αποτελεσματικά από ότι ο Harris. Αυτό συμβαίνει καθώς δεν χρησιμοποιούμε πολύ μικρή τιμή κατωφλίου qualityLevel, με αποτέλεσμα ο Harris να μην πραγματοποιεί καλή ανίχνευση. Σχετικά με την παράμετρο **maxCorners**, βλέπουμε ότι παρά τη μείωση από 300 σε 150 δεν διαπιστώνεται κάποια διαφορά. Αυτό είναι φυσιολογικό, καθώς όπως εξηγήσαμε και προηγουμένως, έχουμε εστιάσει την ανίχνευση σε συγκεκριμένες περιοχές της εικόνας, μέσω της μάσκας, με αποτέλεσμα να αρκεί και ένας μικρότερος μέγιστος αριθμός γωνιών για τον εντοπισμό των σημείων ενδιαφέροντος στα βασικά αντικείμενα της εικόνας. Μάλιστα, σημαντικό ρόλο παίζει και η παράμετρος **minDistance**, η οποία εδώ είναι αρκετά μεγάλη, με αποτέλεσμα να περιορίζεται η ανίχνευση. Δεδομένου λοιπόν ότι η παράμετρος **maxCorners** επηρεάζει την ταχύτητα εκτέλεσης του αλγορίθμου, μπορούμε να πετύχουμε πολύ καλά αποτελέσματα σε ταχύτερο χρόνο (δηλαδή εξίσου καλή ανίχνευση για μικρότερη τιμή maxCorners).

- maxCorners = 300, **qualityLevel = 0.01** , minDistance = 5

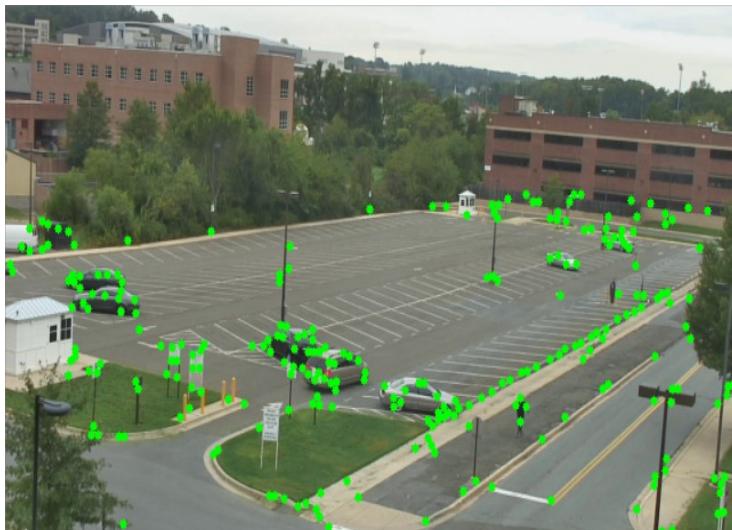


Harris detector

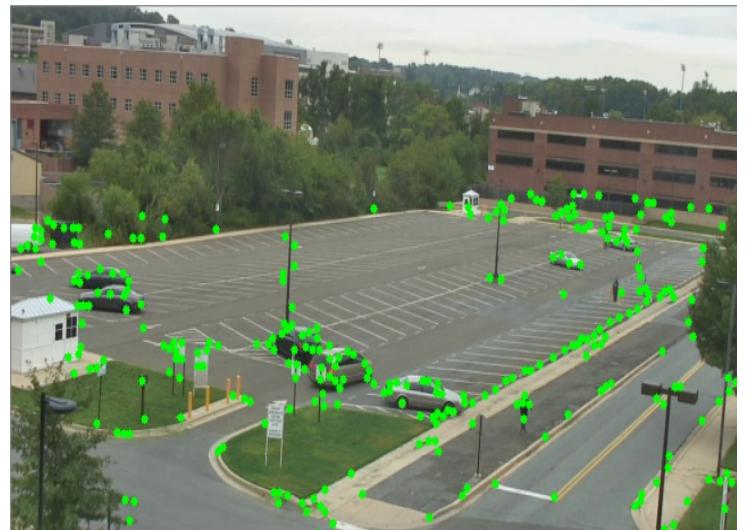


Shi-Tomasi detector

- maxCorners = 300, **qualityLevel = 0.005** , minDistance = 5



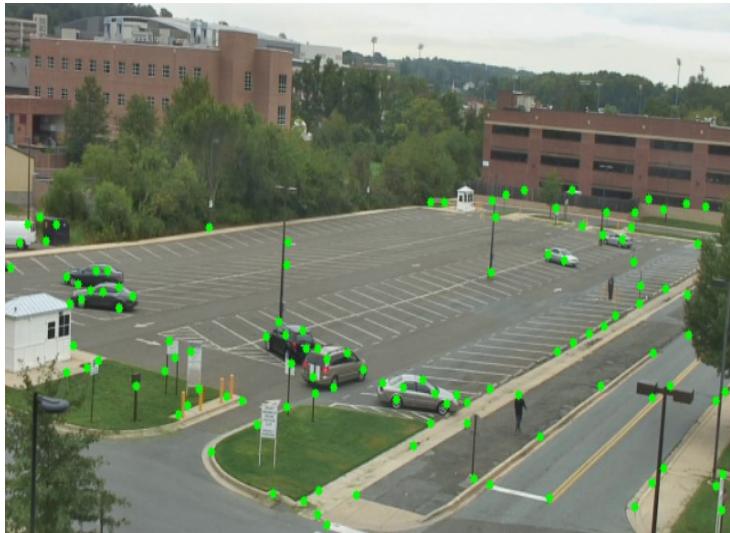
Harris detector



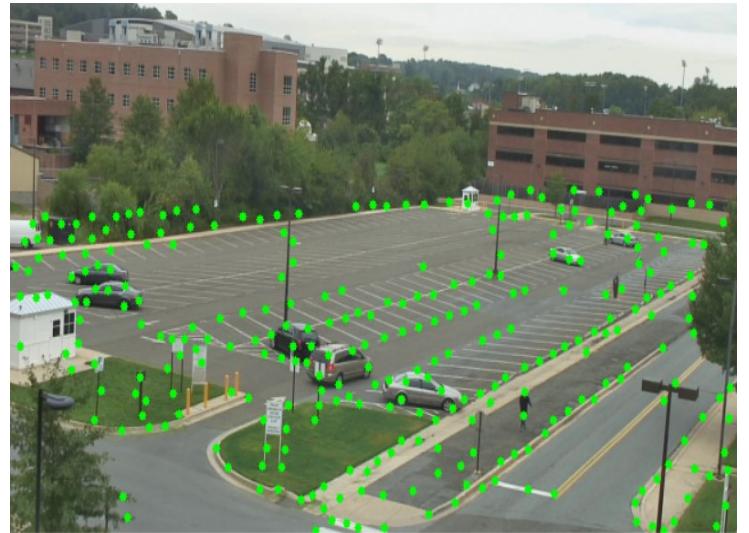
Shi-Tomasi detector

Στις παραπάνω περιπτώσεις χρησιμοποιούμε πολύ μικρότερο threshold (**qualityLevel**) σε σχέση με πριν ενώ παράλληλα μειώνουμε και την παράμετρο **minDistance** ώστε να επιτρέπεται ανίχνευση γωνιών σε πολύ κοντινή απόσταση. Παρατηρούμε πως τώρα οι δύο ανιχνευτές συμπεριφέρονται πολύ παρόμοια, με τον Shi-Tomasi ωστόσο να εξακολουθεί να εντοπίζει κάποια παραπάνω σημεία ενδιαφέροντος που ο ανιχνευτής Harris δεν βρίσκει. Αυτό είναι αναμενόμενο καθώς ο Shi-Tomasi θεωρείται ότι βελτιώνει τον Harris προσφέροντας μια πιο robust και uniform αναπαράσταση των σχημάτων. Για περαιτέρω ωστόσο μείωση της τιμής της παραμέτρου **qualityLevel**, στην τιμή 0.005, τα αποτελέσματα είναι παραπλήσια με πριν καθώς ήδη για κατώφλι ίσο με 0.01 έχει γίνει εντοπισμός σχεδόν όλων των δυνατών προς ανίχνευση γωνιών.

- `maxCorners = 300`, `qualityLevel = 0.01` , **minDistance = 10**



Harris detector

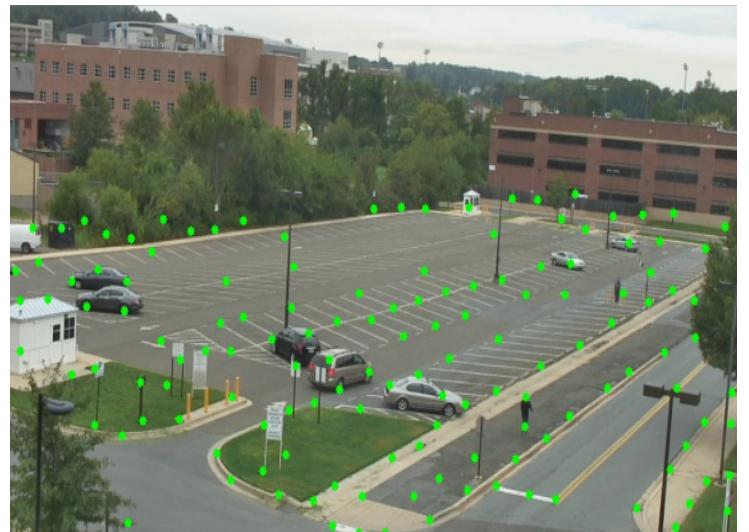


Shi-Tomasi detector

- `maxCorners = 300`, `qualityLevel = 0.01` , **minDistance = 20**



Harris detector



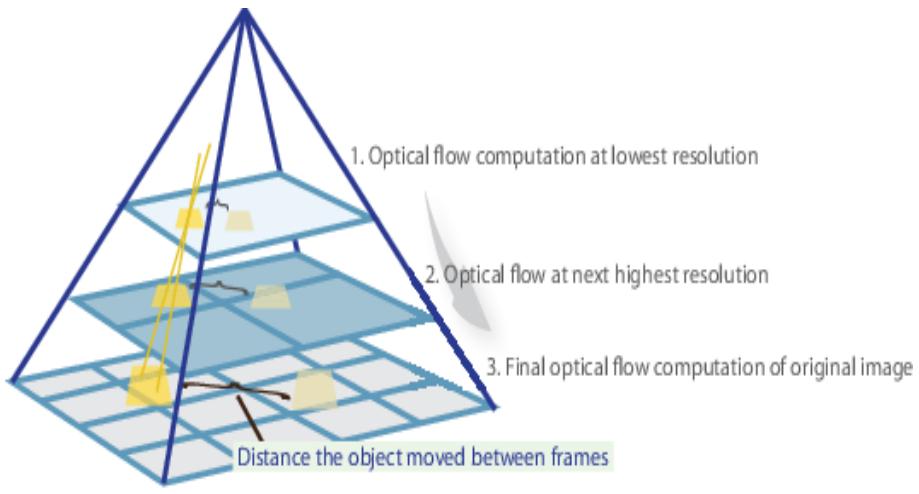
Shi-Tomasi detector

Τώρα εξετάζουμε τον τρόπο που επηρεάζει την ανίχνευση γωνιών η παράμετρος **minDistance** της συνάρτησης **goodFeaturesToTrack**. Η παράμετρος αυτή καθορίζει την ελάχιστη απόσταση μεταξύ δύο pixels στα οποία επιτρέπεται να ανιχνευθεί ακμή. Συνεπώς, όπως και το `qualityLevel`, λειτουργεί ως ένας περιοριστικός παράγοντας για την ανίχνευση. Συγκεκριμένα, αν η τιμή της παραμέτρου δεν είναι πολύ μεγάλη (π.χ είναι ίση με 5, όπως συναντήσαμε στις 4 προηγούμενες περιπτώσεις) τότε μπορούν να βρεθούν πολλές γωνίες που να είναι χοντά μεταξύ τους. Αν πάλι αυτή λάβει μεγαλύτερες τιμές, όπως φαίνεται και στις παραπάνω εικόνες, που λαμβάνει την τιμή 10 και 20 αντίστοιχα, οι γωνίες που ανιχνεύονται βρίσκονται πιο αραιά μεταξύ τους. Ειδικότερα, για την τιμή 20 εμφανίζονται μόλις 1-2 σε κάθε αντικείμενο. Επομένως, ισχύει γενικά ότι μεγάλες τιμές της παραμέτρου μπορεί να φανούν χρήσιμες αν μας ενδιαφέρει να έχουμε μονάχα 1 ή 2 σημεία ενδιαφέροντος πάνω σε κάθε αντικείμενο.

#### 4 Εφαρμογή του αλγορίθμου Lucas-Kanade για τον υπολογισμό optical flow στα Harris και Shi-Tomasi σημεία του προηγούμενου βήματος

Στο προηγούμενο βήμα εντοπίσαμε σημεία ενδιαφέροντος (γωνίες) στα αντικείμενα του πρώτου καρέ του βίντεο, χρησιμοποιώντας τους ανιχνευτές Harris και Shi-Tomasi. Τώρα θα εφαρμόσουμε τον αλγόριθμο των Lucas-Kanade για τον υπολογισμό του optical flow στα σημεία αυτά. Η υλοποίηση του αλγορίθμου πραγματοποιείται μέσω της έτοιμης συνάρτησης **calcOpticalFlowPyrLK** της βιβλιοθήκης **cv2**. Οι παράμετροι της συνάρτησης με τις οποίες θα πειραματιστούμε είναι οι ακόλουθες:

- **winSize:** Καθορίζει το μέγενθος του παραθύρου αναζήτησης σε κάθε επίπεδο της πυραμίδας. Μικρά παράθυρα είναι αρκετά ευαίσθητα στον θόρυβο ενώ ενδέχεται να μην ανιχνεύσουν μεγάλες κινήσεις. Ωστόσο, παρέχουν μεγαλύτερη ακρίβεια και συμβάλλουν στην ‘προσκόλληση’ των ανιχνευθέντων σημείων πάνω στα αντικείμενα.
- **maxLevel:** Καθορίζει το πλήθος των επιπέδων της πυραμίδας. Όταν αυτό έχει την τιμή 0 δεν χρησιμοποιείται καθόλου πυραμίδα και έχουμε ανίχνευση μονής κλίμακας (η συνάρτηση ταυτίζεται με τη **calcOpticalFlowLK**). Αντίθετα, για τιμές διάφορες του μηδενός έχουμε πολυκλιμακωτή ανάλυση σε τόσες κλίμακες όσες και τα επίπεδα της πυραμίδας. Για παράδειγμα, αν η παράμετρος λάβει την τιμή 1 έχουμε πυραμίδα 2 επιπέδων και κατά συνέπεια ανάλυση σε 2 κλίμακες. Η πολυκλιμακωτή ανάλυση συμβάλει στην βελτίωση της εκτίμησης κίνησης αφού εξετάζει πολλαπλές αναλύσεις, ωστόσο αυξάνει την υπολογιστική πολυπλοκότητα του αλγορίθμου μας. Μια οπτικοποίηση της πυραμίδας που χρησιμοποιεί ο αλγόριθμος φαίνεται στην ακόλουθη εικόνα:



- **criteria:** Καθορίζει το κριτήριο τερματισμού του επαναληπτικού αλγορίθμου αναζήτησης. Αποτελείται από δύο υποπαραμέτρους: την **cv.TERM\_CRITERIA\_EPS** και την **cv.TERM\_CRITERIA\_COUNT**. Η πρώτη παράμετρος λειτουργεί ως κατώφλι για την μετακίνηση του παραθύρου αναζήτησης, οδηγώντας στον τερματισμό του αλγορίθμου όταν

αυτό μετακινηθεί λιγότερο από την τιμή αυτή. Επομένως, όσο μικρότερο είναι το epsilon τόσο πιο γρήγορα τερματίζει ο αλγόριθμος. Η δεύτερη αφορά το μέγιστο πλήθος επαναλήψεων του αλγορίθμου. Όσο μεγαλύτερη τιμή λαμβάνει τόσο πιο εξαντλητική είναι η αναζήτηση. Οι τυπικές τιμές για τις παραμέτρους αυτές είναι 0.03 και 10 αντίστοιχα ενώ, όπως είναι λογικό, οποιεσδήποτε μεταβολές τους επηρεάζουν άμεσα την ταχύτητα και την ακρίβεια του αλγορίθμου.

Η επισημείωση της οπτικής ροής θέλουμε με κάποιο τρόπο να ανανεώνεται καθώς σε αντίθετη περίπτωση μετά από κάποιο χρονικό διάστημα υπάρχουν επικαλύψεις παλιών και καινούριων γραμμών που σχεδιάζονται, με αποτέλεσμα να δημιουργείται ένα συγκεχυμένο οπτικό αποτέλεσμα, όπως αυτό της παρακάτω εικόνας:



Optical flow without refreshing

Αυτό μάλιστα αναμένεται να είναι ακόμα χειρότερο αν εντοπίζονται και 'αντικείμενα' που εισέρχονται στο βίντεο μετά το πρώτο frame. Μια καλή λύση για την αντιμετώπιση του παραπάνω προβλήματος είναι να καταγράψουμε την οπτική ροή για έναν συγκεκριμένο αριθμό από καρέ (π.χ 50) και έπειτα για κάθε νέα επισημείωση που πρόκειται να κάνουμε, να σβήνουμε την παλαιότερη από αυτές που έχουμε ήδη κάνει. Με τον τρόπο αυτό λαμβάνουμε κάποιες γραμμές πεπερασμένου μήκους που ακολουθούν τα κινούμενα αντικείμενα του βίντεο ενώ παράλληλα μαζεύουν, διατηρώντας σταθερό το μήκος τους. Έτσι, παρέχουμε μια καλή εποπτεία της κίνησης. Για να το πετύχουμε αυτό χρησιμοποιούμε μια ουρά σταθερού μεγέθους, στην οποία σε κάθε επανάληψη κάνουμε append μια τρισδιάστατη μάσκα, η οποία έχει διαστάσεις ίδιες με αυτές των frames, είναι παντού μαύρη (δηλαδή έχει τιμή 0) ενώ έχει σημειωμένες πάνω της τις γραμμές που ενώνουν τα σημεία ενδιαφέροντος του προηγούμενου frame με αυτά του τρέχοντος, όπως προέκυψαν από τον LK. Έτσι, κρατάμε ουσιαστικά την οπτική ροή μεταξύ κάθε ζεύγους διαδοχικών καρέ που διαβάζουμε. Ελέγχουμε παράλληλα αν το μέγεθος της ουράς υπερβαίνει έναν συγκεκριμένο μέγιστο μήκος (π.χ 50) έτσι ώστε όταν αυτό

μεγιστοποιηθεί να κάνουμε ρορ το τελευταίο στοιχείο που έχουμε εισάγει στην ουρά (που περιέχει την πιο παλιά οπτική ροή) και να εισάγουμε τη νέα μάσκα. Ταυτόχρονα, σε κάθε επανάληψη, ορίζουμε και μια δεύτερη τρισδιάστατη μάσκα, επίσης ίδιων διαστάσεων με τα frames, η οποία και σχετίζεται με την συνολική απεικόνιση που θέλουμε να προβάλλουμε. Η μάσκα αυτή ορίζεται να είναι ίση με το λογικό or όλων των διαδοχικών μασκών που περιέχει μέχρι εκείνη τη στιγμή η ουρά μας. Με αυτόν τον τρόπο λαμβάνουμε μια μάσκα η οποία περιέχει τη συνολική οπτική ροή από τα τελευταία το πολύ 50 frames. Την μάσκα αυτή την προσθέτουμε στο τρέχον έγχρωμο frame και παρουσιάζουμε το αποτέλεσμα. Η διαδικασία που περιγράφηκε υλοποιείται με τον ακόλουθο κώδικα:

```

mask_queue = []

while(cap.isOpened()):

    #code
    #code
    #code

mask = np.zeros_like(first_frame_rs)
mask_temp = np.zeros_like(first_frame_rs)

for masks in mask_queue: mask = cv.bitwise_or(mask,masks)

if len(mask_queue) > line_length: mask_queue.pop(0)

# Draws the optical flow tracks
for i, (new, old) in enumerate(zip(good_new, good_old)):

    # a, b = coordinates of new point
    a, b = new.ravel()

    # c, d = coordinates of old point
    c, d = old.ravel()

    # Draws line between new and old position with green color and 2
    # thickness
    if distance.euclidean(new,old) > points_distance :
        mask = cv.line(mask, (a, b), (c, d), color, 2)
        mask_temp = cv.line(mask_temp, (a, b), (c, d), color, 2)

```

```

# Draws filled circle (thickness of -1) at new position with
# green color and radius of 3
frame_rs = cv.circle(frame_rs, (a, b), 2, color, -1)

mask_queue.append(mask_temp)

output = cv.add(frame_rs, mask)

```

Τα οπτικά αποτελέσματα που λαμβάνουμε στην περίπτωση αυτή είναι αρκετά καλύτερα καθώς είναι πολύ πιο εύχολο να εστιάσει κανείς στα κινούμενα ‘αντικείμενα’ του βίντεο. Ένα παράδειγμα ανανέωσης των επισημειώσεων της οπτικής ροής φαίνεται στην ακόλουθη εικόνα:



Optical flow with refreshing

Εφαρμόζουμε λοιπόν τώρα τον αλγόριθμο Lucas-Kanade για τον υπολογισμό του optical flow στα Harris και Shi-Tomasi σημεία του προηγούμενου βήματος και πειραματίζόμαστε με τις παραμέτρους της συνάρτησης calcOpticalFlowPyrLK με στόχο το καλύτερο δυνατό αποτέλεσμα. Δοκιμάζουμε αρκετές διαφορετικές τιμές για τις παραμέτρους winSize, maxLevel και criteria πάνω σε όλες τις παραμετροποιήσεις του προηγούμενου βήματος. Προφανώς, είναι αδύνατο αλλά και ανούσιο να παρουσιάσουμε όλους αυτούς τους συνδυασμούς παραμέτρων στην παρούσα αναφορά. Έτσι λοιπόν, επιλέγουμε σε πρώτη φάση μια παραμετροποίηση για τον Lucas-Kanade που λειτουργεί καλά στην πλειοψηφία των περιπτώσεων και την εφαρμόζουμε στα 4 καλύτερα αποτελέσματα ανίχνευσης που λάβαμε στο προηγούμενο ερώτημα. Η παραμετροποίηση αυτή παρατίθεται στον ακόλουθο πίνακα:

Lucas Kanade Parameters	
winSize	(15,15)
maxLevel	4
cv.TERM_CRITERIA_EPS	0.03
cv.TERM_CRITERIA_COUNT	10

Στη συνέχεια πρόκειται να κρατήσουμε εκείνο τον συνδυασμό παραμέτρων που μας δίνει την καλύτερη εκτίμηση κίνησης. Πάνω σε αυτόν τον συνδυασμό πρόκειται να δούμε πως η μεταβολή των παραμέτρων **winSize**, **maxLevel** και **criteria** επηρεάζουν τον αλγόριθμο των Lucas-Kanade και κατά συνέπεια την εξαγωγή της οπτικής ροής. Για κάθε εκτέλεση του αλγορίθμου παραθέτουμε ενδεικτικά screenshots.

- maxCorners = 150, qualityLevel = 0.12 , minDistance = 20, useHarrisDetector = True



- maxCorners = 150, qualityLevel = 0.12 , minDistance = 20, useHarrisDetector = False



- maxCorners = 300, qualityLevel = 0.01 , minDistance = 20, useHarrisDetector = True





- maxCorners = 300, qualityLevel = 0.01 , minDistance = 20, useHarrisDetector = False



Με βάση τις εκτελέσεις του αλγορίθμου καταλήγουμε στο ότι για τη δοσμένη τυπική παραμετροποίηση του Lucas-Kanade, το καλύτερο αποτέλεσμα ανίχνευσης παρατηρείται στην περίπτωση που έχουμε **maxCorners = 150**, **qualityLevel = 0.12**, **minDistance = 20** και **useHarrisDetector = False**. Μεταβάλλουμε τώρα τις τιμές **winSize**, **maxLevel** και **criteria** της τυπικής παραμετροποίησης της **calcOpticalFlowPyrLK** έτσι ώστε να εξετάσουμε πως αυτές επηρεάζουν τον αλγόριθμο εξαγωγής της οπτικής ροής. Για κάθε αλλαγή που πραγματοποιούμε παραθέτουμε ενδεικτικά screenshots από την νέα εκτέλεση του αλγορίθμου μας:

Lucas Kanade Parameters	
winSize	(5,5)
maxLevel	4
cv.TERM_CRITERIA_EPS	0.03
cv.TERM_CRITERIA_COUNT	10



Παρατηρούμε ότι για μικρότερο μέγεθος παραθύρου ( $5, 5$ ) εμφανίζονται ορισμένα σφάλματα κατά την εκτίμησης της κίνησης, τα οποία και δείχνουμε εντός κόκκινων ορθογωνίων πλαισίων. Αυτό είναι αναμενόμενο καθώς, όπως εξηγήσαμε, τα μικρά παράθυρα είναι επιρρεπή στο θόρυβο ενώ αδυνατούν να ανιχνεύσουν μεγάλες κινήσεις.

Lucas Kanade Parameters	
winSize	(15,15)
maxLevel	1
cv.TERM_CRITERIA_EPS	0.03
cv.TERM_CRITERIA_COUNT	10



Παρατηρούμε ότι τα αποτελέσματα είναι παραπλήσια με πριν. Αυτό συμβαίνει καθώς συνήθως αρκούν 2 επίπεδα της πυραμίδας για να λάβουμε αρκετά ικανοποιητικά αποτελέσματα. Ωστόσο, στην περίπτωση των 5 επιπέδων (**maxLevel = 4**) υπάρχει μια μικρή βελτίωση ως προς την ανίχνευση καθώς εξετάζονται περισσότερες κλίμακες σε κάθε βήμα.

Τέλος, δοκιμάζουμε να μεταβάλλουμε και την παράμετρο **criteria** η οποία αποτελείται από τις υποπαραμέτρους **cv.TERM\_CRITERIA\_EPS** και **cv.TERM\_CRITERIA\_COUNT**. Σε πρώτη φάση μεταβάλλουμε την **cv.TERM\_CRITERIA\_EPS** από την τιμή 0.03 στην τιμή 0.01:

Lucas Kanade Parameters	
winSize	(15,15)
maxLevel	4
cv.TERM_CRITERIA_EPS	0.01
cv.TERM_CRITERIA_COUNT	10

Δεν παρατηρούμε κάποια αλλαγή στα αποτελέσματα της οπτικής ροής, συνεπώς δεν υπάρχει λόγος να παραθέσουμε κάποια screenshots για τη συγκεκριμένη εκτέλεση του κώδικα.  
Μεταβάλλουμε λοιπόν τώρα την **cv.TERM\_CRITERIA\_COUNT** από την τιμή 10 στην τιμή 5:

Lucas Kanade Parameters	
winSize	(15,15)
maxLevel	4
cv.TERM_CRITERIA_EPS	0.03
cv.TERM_CRITERIA_COUNT	5

Ούτε σε αυτή την περίπτωση παρατηρούμε κάποια αλλαγή στα αποτελέσματα της οπτικής ροής, συνεπώς δεν παρουσιάζουμε κάτι για τη συγκεκριμένη εκτέλεση.

Επομένως, λαμβάνοντας υπόψιν όλες τις παραπάνω παραμετροποιήσεις που εξετάσαμε, καταλήγουμε στο συμπέρασμα ότι ο καλύτερος συνδυασμός παραμέτρων για τη μέθοδο ανίχνευσης γωνιών και τον αλγόριθμο Lucas-Kanade είναι αυτός που παρουσιάζεται στους ακόλουθους πίνακες:

Corner Detection Parameters		Lucas Kanade Parameters	
maxCorners	150	winSize	(15,15)
qualityLevel	0.12	maxLevel	4
minDistance	20	cv.TERM_CRITERIA_EPS	0.03
useHarrisDetector	False	cv.TERM_CRITERIA_COUNT	10

## 5 Τροποποίηση του κώδικα του βήματος 4, για παρακολούθηση γωνιών Harris και Shi-Tomasi που εμφανίσθηκαν μετά το πρώτο frame

Θέλουμε τώρα να τροποποιήσουμε τον κώδικα του προηγούμενου βήματος έτσι ώστε να ανιχνεύουμε κίνηση και ‘αντικειμένων’ που εμφανίζονται στο βίντεο μετά το πρώτο καρέ, δηλαδή θέλουμε να πραγματοποιούμε εκτίμηση κίνησης για αυτοκίνητα και ανθρώπους που εισέρχονται ή εξέρχονται από το parking καθώς το βίντεο εξελίσσεται. Αυτό μπορούμε να το πετύχουμε με ενημέρωση των σημείων που παρακολουθούνται ανά τακτικά διαστήματα. Η ανανέωση δεν θέλουμε να είναι πολύ συχνή (π.χ σε κάθε frame), καθώς θα αυξήσουμε πολύ την υπολογιστική πολυπλοκότητα του αλγορίθμου μας ενώ παράλληλα η οπτικοποίηση της οπτικής ροής δεν θα είναι καλή αφού θα εμφανίζεται με διακεκομένες γραμμές. Από την άλλη, δεν θέλουμε κιόλας η ανανέωση να γίνεται πολύ αραιά καθώς τότε θα καθυστερεί πολύ η ανίχνευση ‘αντικειμένων’ που εμφανίζονται μετέπειτα στο βίντεο. Μετά από δοκιμές, καταλήγουμε στο συμπέρασμα ότι μια καλή επιλογή είναι να ενημερώνουμε τα corners κάθε 20 frames. Χρησιμοποιούμε λοιπόν έναν μετρητή (frames) καθώς και μια σταθερά feature\_update\_frames που καθορίζει αυτή τη συχνότητα ενημέρωσης:

```
# Updates feature points
frames = 0
while(cap.isOpened()):
    frames = frames + 1
    ret, frame = cap.read()

    #code
    #code
    #code

    if frames==feature_update_frames:
        prev = cv.goodFeaturesToTrack(gray, mask = mask2D, **feature_params)
        frames=0
    else:
        prev = good_new.reshape(-1, 1, 2)
```

Παράλληλα, επιψυμούμε να μην επισημειώνουμε όλα τα σημεία, όπως πριν, αλλά μόνο εκείνα που αλλάζουν σημαντικά θέση. Αυτό μπορεί να υλοποιηθεί εύκολα εξετάζοντας την απόσταση μεταξύ των παλιών θέσεων των σημείων ενδιαφέροντος και των αντίστοιχων νέων, όπως αυτές εκτιμώνται από τον αλγόριθμο Lucas-Kanade. Αν η απόσταση αυτή είναι μεγαλύτερη από κάποιο κατάλληλο κατώφλι τότε αυτό σημαίνει πως υπάρχει κίνηση στο αντικείμενο όπου βρίσκονται τα σημεία αυτά, οπότε και θα πρέπει να σχεδιάσουμε μια γραμμή που να τα συνδέει. Αν από την άλλη, η απόσταση αυτή είναι μικρότερη από το threshold μας, τότε αυτό σημαίνει

πως πιθανότατα το αντικείμενο στο οποίο έγινε η ανίχνευση των corners είναι ακίνητο ή εμφανίζει κάποια πολύ μικρή κίνηση. Στην περίπτωση αυτή, δεν χρειάζεται να επισημειώσουμε κάτι, οπότε προχωράμε εξετάζοντας το επόμενο ζεύγος σημείων (new,old). Για τον υπολογισμό της απόστασης αξιοποιούμε τη συνάρτηση `scipy.spatial.distance.euclidean`. Δοκιμάζουμε αρχικά να θέσουμε το κατώφλι απόστασης ίσο με 1 για να εξετάσουμε μετακίνηση πάνω από ένα pixel σε μια από τις διαστάσεις,. Ωστόσο, στην περίπτωση αυτή, καταγράφονται μόνο πολύ απότομες και γρήγορες κινήσεις, όπως ταχύτατη κίνηση αυτοκινήτων, ενώ οποιεσδήποτε άλλες πιο αργές δεν γίνονται ορατές. Καταλήγουμε λοιπόν, μετά από πειραματισμό, σε ένα κατώφλι απόστασης αρκετά μικρότερο και συγκεκριμένα ίσο με 0.1 (points\_distance = 0.1)

```
if distance.euclidean(new,old) > points_distance :
    mask = cv.line(mask, (a, b), (c, d), color, 2)
    mask_temp = cv.line(mask_temp, (a, b), (c, d), color, 2)
    # Draws filled circle (thickness of -1) at new position with green color
    # and radius of 3
    frame_rs = cv.circle(frame_rs, (a, b), 2, color, -1)
```

Χρησιμοποιούμε τώρα τις παραμέτρους που μας έδωσαν το καλύτερο αποτέλεσμα στα βήματα 3 και 4. Αυτές είναι οι εξής:

```
feature_params = dict(maxCorners = 150,qualityLevel = 0.12,minDistance = 20,
                      blockSize = 7,useHarrisDetector = False)
lk_params = dict(winSize = (15,15), maxLevel = 4, criteria =
                  (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03))
```

Παρουσιάζουμε ενδεικτικά screenshots από την εκτέλεση του αλγορίθμου:





Παρατητούμε ότι πραγματοποιείται μια πολύ ικανοποιητική εκτίμηση κίνησης ανθρώπων και αυτοκινήτων που εισέρχονται ή/και εξέρχονται από τον χώρο του parking.

## 6 Εισαγωγή Salt and Pepper θορύβου μετά από κάθε διάβασμα ενός frame

Θέλουμε τώρα να εξετάσουμε την επίδραση θορύβου τύπου Salt and Pepper στην εκτίμηση κίνησης. Για το σκοπό αυτό αξιοποιούμε την συνάρτηση **random\_noise** του πακέτου **skimage.util**, την οποία και καλούμε μετά το διάβασμα κάθε καρέ. Η συνάρτηση δέχεται ως ορίσματα το αρχικό μη θορυβώδες frame καθώς επίσης τα seed και amount που χρησιμοποιήσαμε για την προσθήκη θορύβου στο βήμα 3 του Ερωτήματος 1. Η **random\_noise** επιστρέφει μια θορυβώδη εικόνα η οποία λαμβάνει πραγματικές τιμές στο διάστημα [0,1]. Ωστόσο, εμείς θέλουμε αυτή να εκφρασθεί στο εύρος [0,255] καθώς και να έχει ακέραιες τιμές ώστε να μπορεί να δοθεί ως είσοδος στην συνάρτηση **calcOpticalFlowPyrLK** που υλοποιεί τον αλγόριθμο των Lucas-Kanade. Έτσι, πολλαπλασιάζουμε το αποτέλεσμα της **random\_noise** με το 255 και έπειτα μετατρέπουμε την τελική θορυβώδη εικόνα σε np.uint8:

```
def snpAmount(x): return x/90 + 0.3

# Add s&p noise to frame and convert the result to range [0,255]
gray = (random_noise(gray, mode='s&p', seed=9, amount = snpAmount(7))) * 255

# Convert the noisy frame from type float to type uint8 (necessary for
# function calcOpticalFlowPyrLK)
gray = gray.astype(np.uint8)
```

Μια οπτικοποίηση ενός τυχαίου θορύβου τύπου Salt and Pepper που εισάγεται στο πρώτο frame της ακολουθίας του βίντεο είναι η εξής:



First frame with s&p noise

Όπως παρατηρούμε, ο θόρυβος είναι αρχετά ισχυρός και αναμένεται να επηρεάσει σημαντικά την διαδικασία εξαγωγής χαρακτηριστικών και εκτίμησης κίνησης.

Στις εικόνες που ακολουθούν παρουσιάζουμε ενδεικτικά screenshots από την εκτέλεση του αλγορίθμου μετά την προσθήκη θορύβου σε κάθε frame που διαβάζεται. Να σημειωθεί ότι η απεικόνιση της οπτικής ροής γίνεται πάνω στα μη θορυβώδη καρέ του βίντεο για καυτά εποπτικούς λόγους, καθώς σε αντίθετη περίπτωση θα αναμειγνύονταν με το θόρυβο και θα προέκυπτε ένα συγκεχυμένο οπτικό αποτέλεσμα.





Βλέπουμε πως στην περίπτωση αυτή πράγματι η ανίχνευση κίνησης δεν δίνει καλά αποτελέσματα αφού εντοπίζονται μερικώς μόνο ορισμένες κινήσεις αυτοκινήτων. Ωστόσο, ακόμα και αυτές δεν παρουσιάζονται με μια συνεχή οπτική ροή αλλά με διαχεκομένα σημεία που απέχουν αρκετά μεταξύ τους. Οι κινήσεις των ανθρώπων δεν εντοπίζονται παρά μόνο σε ελάχιστα καρέ όπου εμφανίζονται κάποιες αραιές επισημειώσεις στα σημεία που περνούν. Για την αντιμετώπιση του προβλήματος αυτού θα πρέπει να εφαρμόσουμε σε κάθε θόρυβώδες frame αποθορυβοποίηση με κατάλληλα επιλεγμένο φίλτρο ώστε να εξαλείψουμε το θόρυβο από την εικόνα και να μπορέσουμε να πετύχουμε μια καλή ανίχνευση κίνησης. Τη διαδικασία αυτή εφαρμόζουμε στο αμέσως επόμενο βήμα.

## 7 Εφαρμογή κατάλληλης αποθορυβοποίησης πριν την εισαγωγή του κάθε frame στον αλγόριθμο

Όπως είδαμε στο βήμα 4 του Ερωτήματος 1, το πιο αποτελεσματικό φίλτρο για την εξαγωγή του ψηφιακού τύπου Salt and Pepper είναι το φίλτρο ενδιάμεσης τιμής (Median Filter). Το φίλτρο αυτό είναι μη-γραμμικό και ενδείκνυται για αποθορυβοποίηση εικόνων καθώς διατηρεί τα χαρακτηριστικά της αφαιρώντας μόνο το ψήφιο. Έτσι λοιπόν, επεκτείνουμε τον κώδικα του προηγούμενου ερωτήματος, εφαρμόζοντας ένα φίλτρο ενδιάμεσης τιμής σε κάθε ψηφιακή εικόνα που προκύπτει. Το φίλτρο αυτό υλοποιείται μέσω της συνάρτησης **filters.rank.median** της βιβλιοθήκης **skimage**. Η συνάρτηση δέχεται ως ορίσματα την ψηφιακή εικόνα καθώς και μια μάσκα που καθορίζει την τοπική γειτονιά εφαρμογής του φίλτρου. Επιλέγουμε ως μάσκα έναν δίσκο ακτίνας 3 τον οποίο ορίζουμε με χρήση της συνάρτησης **disk** του πακέτου **skimage.morphology**:

```
neighborhood = disk(radius=3)
gray = filters.rank.median(gray, neighborhood)
```

Μια οπτικοποίηση της αποθορυβοποίησης μετά από προσθήκη ενός τυχαίου ψηφιακού τύπου Salt and Pepper στο πρώτο frame της ακολουθίας του βίντεο είναι η εξής:



First frame after denoising

Όπως βλέπουμε, το φίλτρο απομάκρυνε σχεδόν όλο το ψήφιο από την εικόνα διατηρώντας τα βασικά χαρακτηριστικά της, όπως οι ακμές.

Στις εικόνες που ακολουθούν παρουσιάζουμε ενδεικτικά screenshots από την εκτέλεση του αλγορίθμου μετά την αποθήρυβοποίηση κάθε θορυβώδους frame. Να σημειωθεί ότι η απεικόνιση της οπτικής ροής γίνεται πάλι πάνω στα μη θορυβώδη καρέ του βίντεο για καθαρά εποπτικούς λόγους, καθώς σε αντίθετη περίπτωση θα προέκυπτε ένα θολό οπτικό αποτέλεσμα.





Όπως παρατηρούμε, η εκτίμηση κίνησης βελτιώνεται αισθητά σε σχέση με πριν, χωρίς ωστόσο να είναι τόσο καλή όσο είναι στην περίπτωση που δεν εισάγεται καθόλου θόρυβος. Αυτό προφανώς είναι φυσιολογικό καθώς ο θόρυβος που προστίθεται σε κάθε επανάληψη είναι αρκετά ισχυρός και κατά συνέπεια η αποθορυβοποίηση δεν είναι δυνατόν να πραγματοποιηθεί χωρίς καμία αλλοίωση των αντικειμένων της εικόνας.