ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ VLSI

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Automated Code Generation for State Estimation Algorithms

Ορέστης Κ. Καπαρουνάκης
(Orestis K. Kaparounakis)

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής Ε.Μ.Π.

*

Αθήνα, Ιούλιος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ VLSI

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Automated Code Generation for State Estimation Algorithms

Ορέστης Κ. Καπαρουνάκης
(Orestis K. Kaparounakis)

Επιβλέπων: Δημήτριος Ι. Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2020-07-21.

.....	.....	.....
Δημήτριος Σούντρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.	Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020.

.....
Ορέστης Κ. Καπαρουνάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Παρουσιάζουμε μια μέθοδο για αυτοματοποιημένη υλοποίηση αλγορίθμων εκτίμησης κατάστασης από περιγραφές αναγνωρίσιμες από υπολογιστή που περιγράφουν τους νόμους που διέπουν κάποιο φυσικό υπολογιστικό σύστημα στόχο όσο αφορά την δυναμική εξέλιξή του καθώς και τους αισθητήρες του. Υλοποιούμε την μέθοδο αυτή σαν οπίσθιο τμήμα μεταγλωττιστή μιας γλώσσας προδιαγραφής φυσικών νόμων και την χρησιμοποιούμε για να παράξουμε ολοκληρωμένες υλοποιήσεις των αλγορίθμων σε C, είτε για γραμμικά (γραμμικά φίλτρα Kalman), είτε για μη γραμμικά συστήματα (εκτεταμένα φίλτρα Kalman). Η παραγωγή του κώδικα των φίλτρων εκτίμησης κατάστασης είναι εντελώς αυτόματη και δεν απαιτεί χειροκίνητες επεμβάσεις. Τα παράγωγα φίλτρα δύναται να συμπεριλαμβάνουν λειτουργικότητα αυτόματης διαφοροποίησης, μεθόδου διαφοροποίησης που συνδυάζει την αποτίμηση μιας συνάρτησης με την αποτίμηση των παραγώγων της, ταυτόχρονα.

Χρησιμοποιούμε περιγραφές φυσικών συστημάτων διαφορετικής πολυπλοκότητας για να παράξουμε εκτεταμένα φίλτρα Kalman των οποίων αξιολογούμε την απόδοση χρησιμοποιώντας καταγεγραμμένα ίχνη από προσομοιώσεις. Η αξιολόγηση δείχνει ότι τα, μέσω της μεθόδου μας, αυτομάτως παραχθέντα φίλτρα εκτίμησης κατάστασης επιτυγχάνουν εκτίμηση εντός των ίδιων ορίων σφάλματος σε σχέση με φίλτρα γραμμένα με το χέρι. Επιπλέον, αξιολογούμε το μέγεθος αρχείου και το αριθμό δυναμικών εντολών εκτέλεσης των παραχθέντων υλοποιήσεων σε αρχιτεκτονική RISC-V. Η αξιολόγηση δείχνει ότι τα αυτομάτως παραχθέντα φίλτρα που χρησιμοποιούν την μέθοδο της αυτόματης διαφοροποίησης έχουν κατά μέσο όρο λιγότερες δυναμικές εντολές κατά την εκτέλεση, 7%–16% σε σχέση με την οριακή μέθοδο διαφοροποίησης. Η βελτίωση αυτή έχει ένα μικρό κόστος, κατά μέσο όρο 4.5%, αύξησης του μεγέθους αρχείου των φίλτρων.

Λέξεις κλειδιά: εκτίμηση κατάστασης, παραγωγή κώδικα, φίλτρα Kalman, φυσικά υπολογιστικά συστήματα, ενσωματωμένα συστήματα, σύνθεση προγράμματος

Abstract

We present a new method for automatically generating the implementation of state-estimation algorithms from a machine-readable description of the physical dynamics and signal constraints of sensing platforms. We implement the new state-estimator code generation method as a backend for a physics specification language and we apply the backend to generate complete C code implementations of state estimators for both linear systems (Kalman filters) and non-linear systems (extended Kalman filters). The state estimator code generation from physics specification is completely automated and requires no manual intervention. The generated filters can incorporate an automatic differentiation technique which combines function evaluation and differentiation in a single process.

Using the description of physical system of a range of complexities, we generate extended Kalman filters, which we evaluate in terms of prediction accuracy using simulation traces. The results show that our automatically-generated sensor fusion and state estimation implementations provide state estimation within the same error bound as the human-written counterparts. We additionally quantify the code size and dynamic instruction count requirements of the generated state estimator implementations on the RISC-V architecture. The results show that our synthesized state estimation implementation employing automatic differentiation leads to an average improvement in the dynamic instruction count of the generated Kalman filter of 7%–16% compared to the standard differentiation technique. This is improvement comes at the limited cost of an average 4.5% increase in the code size of the generated filters.

Keywords: state estimation, code generation, Kalman filters, cyber-physical systems, embedded systems, program synthesis

Ευχαριστίες

Πρωτίστως, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέπων καθηγητή μου, τον καθηγητή Δημήτριο Σούντη, Ε.Μ.Π., που μου έδωσε την ευκαιρία να εργαστώ σε αυτό το συναρπαστικό έργο για την διπλωματική μου εργασία σε συνεργασία με το εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων στο Εθνικό Μετσόβιο Πολυτεχνείο, Αθήνα. Επιπλέον, είμαι έντονα ευγνώμων προς τον καθηγητή Phillip Stanley-Marbell, στο Πανεπιστήμιο του Cambridge, που μου έδωσε την ευκαιρία να εργαστώ στο και σε συνεργασία με το εργαστήριο Physical Computation Laboratory, στο Cambridge, U.K., καθώς και για την ανεκτίμητη διορατικότητά του, συνεισφορά του και την σταθερή πίστη του στις ικανότητές μου. Επιπλέον, θα ήθελα να εκφράσω βαθειά ευγνωμοσύνη στον μεταδιδάκτορα Βασίλειο Τσούτσουρα, στο Πανεπιστήμιο του Cambridge, χωρίς την συνεχή καθοδήγηση, βοήθεια και συμβουλή του οποίου, αυτή η εργασία δεν θα ήταν δυνατή. Είμαι επίσης ευγνώμων προς τους Damien Zufferey και Marcus Pirron για τις χρήσιμες παραινέσεις και υποδείξεις τους. Πέραν αυτών, θα ήθελα να ευχαριστήσω τα μέλη των δυο εργαστηρίων, Micro-Lab και PCL, για την βοήθειά τους και την εποικοδομητική κριτκή τους. Ευχαριστώ επίσης τους καθηγητές που με βοήθησαν να προχωρήσω τις γνώσεις μου μέσα από τις σπουδές μου.

Ακόμα, θα ήθελα να ευχαριστήσω την οικογένειά μου, τους γονείς μου Κωνσταντίνο και Καλλιόπη και την αδερφή μου Χρυσή, που σταθερά με υποστηρίζουν σε όλη μου τη ζωή, τις σπουδές και τις υπιλογές μου, και μου έχουν παρέχει όλα τα απαραίτητα για να ακολουθήσω τους στόχους μου. Επεκτείνω την ευγνωμοσύνη μου και στους φίλους μου, που πάντα με υποστηρίζουν στα δύσκολα. Τέλος, ένα μεγάλο ευχαριστώ στην κοπέλα μου Ασημίνα, που με υπομονή με υποστήριζε σε στιγμές πίεσης και άγχους.

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor, professor Dimitrios Soudris, N.T.U.A., who gave me the opportunity to work on this exciting project for my thesis with the Microprocessors and Digital Systems Laboratory in the National Technical University of Athens. I'm extremely grateful to professor Phillip Stanley-Marbell, University of Cambridge, for giving me the opportunity to work at and in collaboration with the Physical Computation Laboratory, in Cambridge, U.K., as well as his invaluable insights, contributions and profound belief in my abilities throughout my endeavour. Additionally, I would like to extend my deepest gratitude to post-doctoral researcher Vasileios Tsoutsouras, University of Cambridge, without the constant guidance, assistance and cooperation of whom, this project would not have been possible. I am also grateful to Damien Zufferey and Marcus Pirron, Max Planck Institute, for their helpful advice and suggestions. Furthermore, I wish to thank the members of both MicroLab and PCL for their assistance and constructive criticism. Thanks should also go to to my professors who have helped me advance my knowledge in my studies.

Last but not least, I would like to thank my family, my parents Kostantinos and Kalliopi and my sister Chrysi, who have constantly supported me throughout my life, my studies and my decisions and provided me with whatever was necessary for me to pursue my goals. I extend my gratitude to my friends, who have always been there when the going gets tough. Finally, a big special thanks to my girlfriend Asimina, who patiently supported me in times of pressure and anxiety.

Contents

List of Figures

List of Tables

Listings

Εκτεταμένη Περίληψη

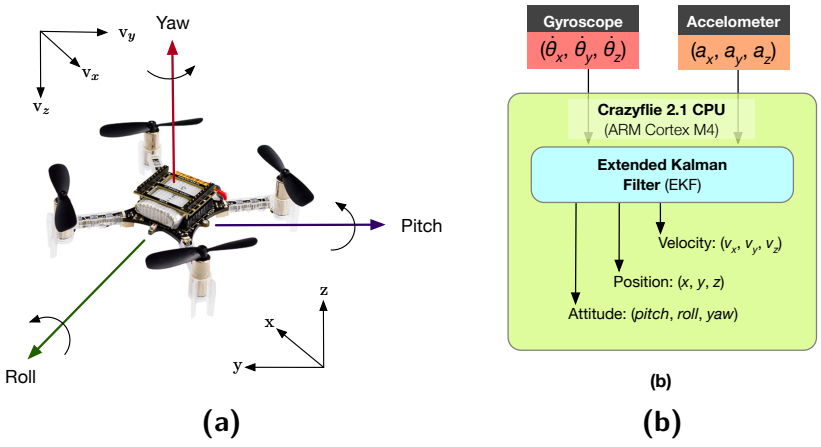
1 Εισαγωγή

Την εποχή όπου οι υπολογιστές βρίσκονται παντού τριγύρω μας, είναι εύκολο να ξεχάσει κανείς την εργασία που έχει καταβληθεί για να δουλεύουν χωρίς συνεχή προβλήματα όλες οι υπολογιστικές συσκευές. Από τα αυτοκίνητα, στα αεροπλάνα και στα έξυπνα κινητά, οι υπολογιστικές συσκευές παρακολουθούν συνεχώς τις εξελίξεις του περιβάλλοντός τους ώστε να παίρνουν αυτόματες μηχανικές αποφάσεις.

Αυτή η λειτουργικότητα είναι το αντικείμενο σπουδής της Θεωρίας των Συστημάτων Ελέγχου. Τα υπολογιστικά συστήματα μοντελοποιούνται σε μαθηματικές οντότητες όπου μελετώνται για την ορθότητα, την ακρίβεια και την ευστάθειά τους. Το αποτέλεσμα από την θεωρητική μελέτη συνήθως είναι ένας ελεγκτής που μπορεί να παίρνει αποφάσεις βάσει των τιμών των αισθητήριων σημάτων του συστήματος και της τρέχουσας κατάστασης του.

Ωστόσο, όλες οι μετρήσεις έχουν έναν αναπόφευκτο βαθμό αβεβαιότητας (*θόρυβος μέτρησης*) και έτσι, ένα σύστημα που τις χρησιμοποιεί για να πάρει αποφάσεις θα πρέπει να λαμβάνει υπόψη του και αυτήν την αλήθεια. Επιπλέον, οι φυσικοί νόμοι που ενδεχομένως περιγράφουν την εξέλιξη κάπου φυσικού συστήματος δεν ακολουθώνται πλήρως στην πραγματικότητα, λόγω της ύπαρξης *διεργασιακού θορύβου*. Γι΄ αυτούς τους λόγους έχουν αναπτυχθεί τεχνικές όπως η μετρίαση και το φιλτράρισμα, οι οποίες συνδυάζουν τις θορυβώθεις μετρήσεις από πολλαπλούς αισθητήρες μαζί με γνώση για τους φυσικούς νόμους που διέπουν ένα φυσικό σύστημα ώστε να καταφέρουν την εξομάλυνση του θορύβου.

Για παράδειγμα, ένα μικρού μεγέθους αυτόνομο μη-επαναδρωμένο αεροσκάφος, όπως είναι το CrazyFlie 2 (Giernacki, Skwierczyński, Witwicki, Wroński, & Kozierski, 2017), αντιμετωπίζει το πρόβλημα της αυτονόμησης της πτήσης. Στο Σχήμα 1 παρουσιάζεται το αεροσκάφος καθώς και μια εποπτική εικόνα για την λειτουργία του φίλτρου εκτίμησης κατάστασης που χρησιμοποιεί. Πρόκειται για ένα εκτεταμένο φίλτρο Kalman που δέχεται ως είσοδο τις μετρήσεις του γυροσκόπιου και του επιταχυνσιόμετρου του αεροσκάφους, τα οποία είναι φυσικά σήματα αναμειγμένα με θόρυβο μέτρησης, και παράγει εκτίμηση για την θέση, ταχύτητα και τον προσανατολισμό του μικρού αεροσκάφους. Ταυτόχρονα, το φίλτρο ενσωματώνει τον βαθμό πεποίθησης στο θεωρητικό δυναμικό μοντέλο του συστήματος, από το οποίο παρεκκλίνει η πραγματική συμπεριφορά του λόγω διεργασιακού θορύβου, για παράδειγμα



Σχήμα 1: **(α′)** Το μικρό μη-επανδρωμένο αεροσκάφος CrazyFlie 2.1. **(β′)** Το αεροσκάφος συνδυάζει θορυβώθεις μετρήσεις από τους αισθητήρες του (την γωνιακή ταχύτητα $\dot{\theta}$ και την επιτάχυνση a σε τρεις διαστάσεις) για να παράξει εκτιμήσεις για την κατάσταση του αεροσκάφους η οποία αποτελείται από τον προσανατολισμό, την θέση και την ταχύτητά του.

σε αυτήν την περίπτωση, κάποιος ξαφνικός άνεμος.

Σε γενικές γραμμές, όλα τα φυσικά υπολογιστικά συστήματα εμφανίζουν θορύβους μέτρησης και διεργασιακούς θορύβους. Επομένως, οι σχεδιαστές συστημάτων που αλληλεπιδρούν με τέτοιο τρόπο με το περιβάλλον τους, καλούνται πολύ συχνά να υλοποιοήσουν αλγόριθμους εκτίμησης κατάστασης όπως είναι τα φίλτρα Kalman.

Παρ΄ αυτά, η υλοποίηση αλγορίθμων εκτίμησης κατάστασης απαιτεί βαθειά γνώση για το σύστημα στόχο και τους νόμους που το διέπουν. Επίσης, για την υλοποίηση αυτήν, χρειάζεται επαρκής θεωρητική γνώση πάνω στους συγκεκριμένους αλγόριθμους εκτίμησης κατάστασης. Αυτά όμως αποτελούν μόνο ένα μέρος των δυσκολιών, αφού οι σχεδιαστές ενσωματωμένων συστημάτων καλούνται να ισορροπήσουν τα παραπάνω σε συστήματα με πολύ λίγους πόρους.

Γι΄ αυτούς τους λόγους, η υλοποίηση αλγορίθμων εκτίμησης κατάστασης για ενσωματωμένα συστήματα περιορισμένων πόρων είναι απαιτητική σε ζητήματα ορθότητας αλλά και ταχύτητας. Τέλος, επειδή η εκάστοτε υλοποίηση προκύπτει άμεσα από τις διαφορες φυσικές ιδιότητες του συστήματος, μια μικρή αλλαγή στον σχεδιασμό μπορεί να επιφέρει καθολικές αλλαγές στην υλοποίηση των αλγορίθμων, ουσιαστικά αναγκάζοντας τους μηχανικούς να ξεκινήσουν την υλοποίηση από την αρχή.

Η αυτοματοποίηση του σχεδιασμού και της διαδικασίας υλοποίησης των φίλτρων Kalman επέχει σημαντική θέση για μηχανικούς σε διάφορους τομείς. Γι΄ αυτόν τον λόγο έχουν αναπτυχθεί βιβλιοθήκες και υποστηρικτικά πακέτα που υποβοηθούν στον σχεδιασμό των φίλτρων από τη MathWorks MATLAB (Grewal & Andrews, 2014) και από το GNU Octave. Επιπλέον, η σχετική υποστήριξη στην γλώσσας προγραμματισμού python χαίρει ανοδικής τάσεως, χάρην και στην ίδια την αύξουσα διασημότητα της γλώσσας. Τα διαθέσιμα πακέτα στην python, ωστόσο, δεν είναι άμεσα

εφαρμόσιμα στα ενσωματωμένα συστήματα, όπως είναι οι υλοποιήσεις σε C.

Ίσως η πιο εξέχουσα πρηγούμενη ερευνητική εργασία πάνω στο θέμα της αυτοματοποίησης της υλοποίησης αλγορίμων εκτίμησης κατάστασης να είναι το AutoFilter (Whittle & Schumann, 2004; Richardson & Wilson, 2006), ένα εργαλείο (κλειστού κώδικα) που υποστηρίζει την παραγωγή κώδικα για γραμμικά, γραμμικοποιημένα και εκτεταμένα φίλτρα Kalman, σε διάφορους συνδυασμούς. Δύνεται να παράξει κώδικα C/++, MATLAB, Octave και Module II. Διαθέτει λογικές αφαιρέσεις, όπως αυτόματη γραμμικοποίηση μοντέλου, συμβολική διαφοροποίηση εξισώσεων, και άλλες, που υποβοηθούν στην γρήγορη υλοποίηση των αλγορίθμων. Στην νεότερη έκδοση του εργαλείου (Richardson & Wilson, 2006) οι δημιοργοί αναγνωρίζουν ότι η ολοκληρωμένη δημιουργία του κώδικα εκ μέρους του εργαλείου δημιουργεί πρόβλημα στην κυκλική φύση της ανάπτυξης των συστημάτων στα οποία υλοποιούνται οι αλγόριθμοι εκτίμησης κατάστασης. Σε γενικες γραμμές, το εργαλείο AutoFilter, έχει στόχο να απαλύνει την ανάγκη για γνώσεις προγραμματισμού χαμηλού επιπέδου και, ως εκ τούτου, απευθύνεται σε σχεδιαστές εκτίμησης κατάστασης γενικότερα.

Αντιθέτως, η παρούσα εργασία στέκεται στην απέναντι πλευρά του φάσματος σχεδιασμού. Το εργαλείο που παρέχουμε έχει ως στόχο να απαλύνει τον σχεδιαστή και προγραμματιστή ενός ενσωματωμένου φυσικού υπολογιστικού συστήματος από την ανάγκη για βαθειά θεωρητική γνώση των αλγορίθμων εκτίμησης κατάστασης. Ασφαλίζουμε ότι το εργαλείο μας παράγει κώδικα με απλές και κατανοητές διεπαφές, κατάλληλο για χρήση σε ενσωματωμένα συστήματα με περιορισμένους πόρους.

Εν ολίγοις, στην παρούσα εργασία εξερευνούμε την αυτοματοποίηση ενός σημαντικού μέρους της δουλειάς που απαιτείται για την υλοποίηση αλγορίθμων εκτίμησης κατάστασης για αξιοποίηση σε ενσωματωμένα συστήματα. Χρησιμοποιώντας περιληπτικές περιγραφές των φυσικών νόμων που διέπουν το σύστημα, γραμμένες σε μια προσφάτως ανεπτυγμένη γλώσσα προδιαγραφών (Lim & Stanley-Marbell, 2018), αυτοματοποιούμε την υλοποίηση απλών και εκτεταμένων φίλτρων Kalman σε κώδικα C.

Η αυτοματοποίηση συμπεριλαμβάνει και αυτόματη διαφοροποίηση των μοντέλων που περιλαμβάνονται στις προδιαγεγραμμένες περιγραφές χρησιμοποιώντας Ανάστροφη Αυτόματη Διαφόριση (Reverse Mode Automatic Differentiation) (Griewank, 1992; Baydin, Pearlmutter, Radul, & Siskind, 2017; Margossian, 2019). Αυτό επιτρέπει στο σύστημα την αυτόματη εύρεση των απαραίτητων παραγώγων των Ιακωβιανών πινάκων με αναλυτική ακρίβεια.

Αξιολογούμε τα παραγμένα φίλτρα σε διάφορα φυσικά συστήματα σε προσομοίωση και επιβεβαιώνουμε ότι λειτουργούν, και ότι το επιτυγχάνουν μέσα σε αποδεκτά όρια σφάλματος. Επιπλέον, αξιολογούμε τα φίλτρα σε θέματα μεγέθους αρχείου και αριθμού εντολών δυναμικής εκτέλεσης, επιβεβαιώνοντας ότι λειτουργούν μέσα σε λογικά όρια για αξιοποίηση σε συστήματα περιορισμένων πόρων.

2 Η γλώσσα Newton

Η γλώσσα Newton (Lim & Stanley-Marbell, 2018; Lim κ. συν., 2017) είναι μια γλώσσα περιγραφής φυσικών προδιαγραφών, νόμων και περιορισμών. Η γλώσσα δημιουργήθηκε με αρχικό σκοπό την παροχή ενός εργασιακού πλαισίου για την δήλωση περιορισμών και αναλλοίωτων μεταξύ ροών δεδομένων από αισθητήρες το οποίο να παρέχει παραπάνω πληροφόρηση, την ώρα της μεταγλώττισης αλλά και της εκτέλεσης, σε αλγόριθμους που τρέχουν σε ενσωματωμένα συστήματα.

Η γλώσσα είναι ακόμα νέα αλλά ήδη αποδεικνύει την χρησιμότητά της μεταξύ φυσικών περιγραφών και πολύπλοκων υλοποιήσεων αλγορίθμων στα οπίσθια μέρη του μεταγλωττιστή της. Εξέχον παράδειγμα αυτού αποτελεί η εργασία στην οποία η γλώσσα Newton χρησιμοποιείται ως το εμπρόσθιο τμήμα μιας διαστατικής ανάλυσης που εξάγει μοντέλα πιθανά φυσικά μοντέλα, έχοντας ως πληροφορία τις φυσικές διαστάσεις των εμπλεκόμεων σημάτων (Wang, Willis, Tsoutsouras, & Stanley-Marbell, 2019). Επιπλέον, σε μετέπειτα εργασία οι συγγραφείς δείχνουν ότι η μέθοδος επεκτείνεται οργανικά στην σύνθεση κυκλωμάτων (Tsoutsouras, Vigdorchik, & Stanley-Marbell, 2020), πάλι χρησιμοποιώντας την γλώσσα Newton σαν σημείο εισόδου.

Ένα βασικό χαρακτηριστικό της γλώσσας είναι ότι υλοποιεί ένα διαστατικό σύστημα τύπων χρησιμοποιώντας τα φυσικά μεγέθη ως τύπους. Οι αριθμητικές εκφράσεις στην Newton επαυξάνονται ώστε να συμπεριλαμβάνουν πληροφορία των φυσικών τους διαστάσεων, και γίνονται *ποσοτικές* εκφράσεις. Η Newton κάνει έλεγχο της ασφάλειας των διαστατικών τύπων, εξασφαλίζοντας ότι οι προδιαγεγραμμένες εκφράσεις έχουν νόημα όσο αφορά τις φυσικές τους διαστάσεις. Επομένως, για να γίνει αποδεκτή μια ποσοτική έκφραση από το εμπρόσθιο τμήμα του μεταγλωττιστή θα πρέπει να συμμορφώνεται με το σύστημα διαστατικών τύπων.

Στο πλαίσιο αυτής της εργασίας, εξετάζουμε την γλώσσα Newton σαν επιλογή για την προδιαγραφή της εισόδου του συστήματός μας. Οι ανάγκες μας αφορούν κυρίως την έκφραση των μοντέλων για τα φίλτρα Kalman, σε μορφή λίστας από εξισώσεις. Επιπλέον, υπάρχει ανάγκη για υποστήριξη σε υπολογισμούς με αβεβαιότητα καθώς και υπολογισμού παραγώγων των εξισώσεων μοντέλων.

Γραμματική υποστήριξη για την διατύπωση των εξισώσεων στην Newton υπάρχει ήδη στην γλώσσα. Από την άλλη η υποστήριξη για διατύπωση της αβεβαιότητας είναι περιορισμένη ενώ δεν υπάρχει κάποια υποστήριξη για αυτόματη διαφόρηση εκφράσεων.

Για να αντιμετωπίσουμε τους περιορισμούς αυτούς, αρχικά επεκτείνουμε την γλώσσα ώστε να συμπεριλαμβάνει την αβεβαιότητα στον ορισμό κάποιου διαστατικού τύπου. Έτσι, η αβεβαιότητα που σχετίζεται με ένα ορισμένο σήμα βρίσκεται διαθέσιμη στον εν λόγω διαστατικό τύπο. Επιπλέον, προσθέτουμε υποστήριξη για αυτόματη διαφόρηση εκφράσεων της Newton υλοποιώντας Ανάστροφη Αυτόματη Διαφόριση Reverse Mode Automatic Differentiation (Griewank, 1992)

3 Αυτοματοποιημένη Παραγωγή Κώδικα για Αλγόριθμους Εκτίμησης Κατάστασης

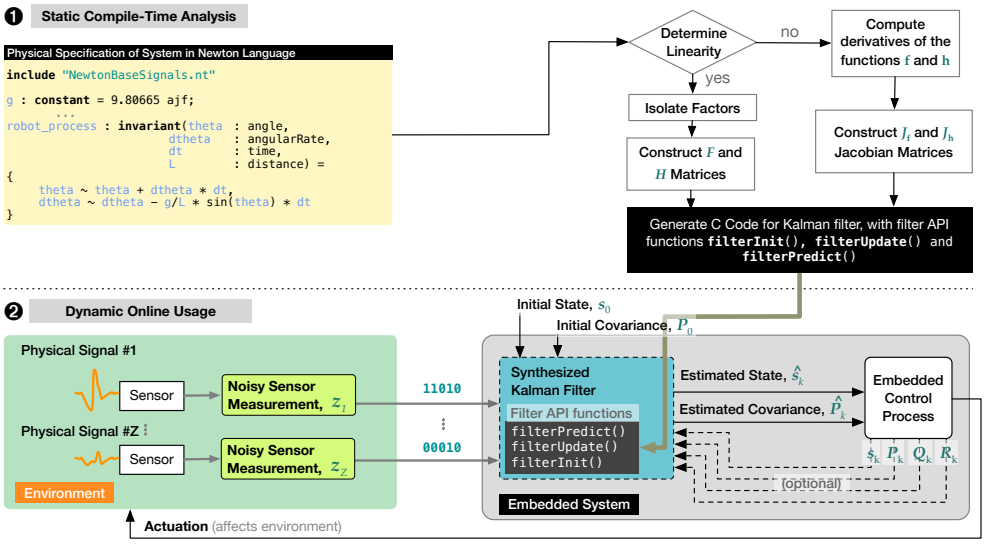
Όπως έχει ήδη αναφερθεί στις προηγούμενες ενότητες, η θορυβώδης φύση των μετρήσεων αναγκάζει την χρήση αλγορίθμων με στόχο την μείωση των σφαλμάτων. Τα φίλτρα Kalman, είτε απλά είτε εκτεταμένα, είναι έγκυρες μέθοδοι για την επίτευξη της εκτίμησης κατάστασης και της συγχώνευσης ροών αισθητήρων σε ενσωματωμένα φυσικά υπολογιστικά συστήματα. Οι μέθοδοι αυτές, όμως, συχνά αφιερώνουν μεγάλο μέρος των πόρων της ανάπτυξης μιας ενσωματωμένης λύσης στον εκ των προτέρων σχεδιασμό του, με τρόπο που περιορίζεται η ελαστικότητα της ανάπτυξης. Αντιλαμβανόμαστε, λοιπόν, την έλλειψη μιας αυτοματοποιημένης λύσης για την υλοποίηση αλγορίθμων εκτίμησης κατάστασης με στόχο περιββάλοντα περιορισμένων πόρων, και γλώσσες όπως η απλή C, που όμως να προκύπτουν από υψηλού επιπέδου φυσικές προδιαγραφές του εν λόγω συστήματος (Kaparounakis, Tsoutsouras, Soudris, & Stanley-Marbell, 2020).

Παρουσιάζουμε ένα νεο εργαλείο για την γρήγορη πρωτοτυποίηση κώδικα εκτίμησης κατάστασης για ενσωματωμένα συστήματα που χρησιμοποιεί την φυσική περιγραφή του συστήματοις και τους φυσικούς νόμους που το διέπουν σαν είσοδο. Το εργαλείο αξιοποιεί την γλώσσα Newton (Lim & Stanley-Marbell, 2018) σαν σημείο εισόδου. Επεκτείνουμε ορισμένες δυνατότητες της Newton ώστε να καλύπτουν τις ανάγκες μας για το νέο αυτό σύστημα και υλοποιούμε το ίδιο το σύστημα σαν ένα οπίσθιο τμήμα για τον μεταγλωττιστή της Newton. Το τελικό σύστημα παρέχει μια αυτοματοποιημένη μέθοδο για την απεικόνιση των υψηλού επιπέδου περιγραφών των φυσικών προδιαγραφών σε μια χαμηλού επιπέδου υλοποίηση που μπορεί να ενσωματωθεί εύκολα σε ενσωματωμένα συστήματα.

Εποπτικά, η χρήση του συστήματος είναι η ακόλουθη: Ο μηχανικός που χρησιμοποιεί το σύστημά μας παρέχει μια συνοπτική περιγραφή των φυσικών προδιαγραφών και νομων του συστήματος, που περιλαμβάνουν το δυναμικό μοντέλο και το μοντέλο μετρήσεων του συστήματος, γραμμένο στην γλώσσα Newton. Η είσοδος αναλύεται από τον συντακτικό αναλυτή του εμπρόσθιου τμήματους του μεταγλωττιστή της Newton και έπειτα ελέγχεται για θέματα ασφάλειας διαστατικών τύπων.

Εν συνεχεία, αν γίνει αποδεκτή η είσοδος, καλείται το οπίσθιο τμήμα που αναπτύχθηκε στο πλαίσιο αυτής της εργασίας. Το τμήμα αυτό αναλαμβάνει την ανάλυση και την παραγωγή κώδικα υλοποίησης των αλγορίθμων εκτίμησης κατάστασης. Σε αυτό το βήμα είναι επίσης δυνατή η χρήση Ανάστροφης Αυτόματης Διαφόρισης για την εύρεση των μερικών παραγώγων των προαναφερθέντων μοντέλων του συστήματος.

Ο τελικός κώδικας μπορεί να ενταχθεί και να χρησιμοποιθεί σε ένα ενσωματωμένο σύστημα. Ο κώδικας παρέχει μια απλή και κατανοητή διεπαφή και μπορεί να χρησιμοποιθεί εύκολα ως βιβλιοθήκη. Το καλόν σύστημα αρχικοποιεί το φίλτρο Kalman με μια αρχική κατάστασης και συνδιακύμανση και έπειτα μπορεί να αρχίσει να πραγματοποιεί κλήσεις πρόβλεψης και εκτίμησης. Η εκτίμηση του φίλτρου για την κατάσταση του συστήματος είναι διαθέσιμη σε μια κεντρική δομή δεδομένων.



Σχήμα 2: Αναλυτική εποπτική εικόνα του υλοποιημένου συστήματος για την αυτοματοποιημένη παραγωγή κώδικα για αλγόριθμους εκτίμησης κατάστασης. Το πάνω μέρος αντιστοιχεί στο κομμάτι της στατικής ανάλυσης του προδιαγεγραμμένου συστήματος. Το κάτω μέρος αναπαριστά την λειτουργία ενός παραχθέντος φίλτρου σε ένα ενσωματωμένο σύστημα.

Το Σχήμα 2 παρουσιάζει μια εποπτική εικόνα της λειτουργίας του υλοποιημένου συστήματος. Η χρήση του συστήματος μπορεί να χωριστεί σε δύο φάσεις: την στατική ανάλυση και την δυναμική χρήση.

Στην φάση της στατικής ανάλυσης (πάνω μέρος του Σχήματος 2) ο χρήστης συγγράφει το αρχείο Newton με τις φυσικές προδιαγραφές του ενσωματωμένου συστήματος στόχου. Ο μεταγλωττιστής της Newton αναλύει συντακτικά την είσοδο και την ελέγχει για διαστατική ορθότητα. Το παραγμένο αφηρημένο συντακτικό δέντρο περνιέται στο οπίσθιο τμήμα του μεταγλωττιστή το οποίο εκτελεί έλεγχους για να διαπιστώσει την γραμμικότητα των μοντέλων των προδιαγραφών και προχωράει στην παραγωγή κώδικα των εκφράσεων και των πινάκων που είναι απαραίτητα για τα φίλτρα Kalman.

Στην φάση της δυναμικής χρήσης (κάτω μέρος του Σχήματος 2) ο χρήστης αξιλογή παραχθέν φίλτρο είτε σε περιβάλλον προσομοίωσης είτε σε ένα πραγματικο ενσωματωμένο σύστημα. Η εκτέλεση του φίλτρου ξεκινάει με την αρχικοποίηση της κατάστασης και της συνδιακύμανσής της. Όταν το ενσωματωμένο σύστημα εγκαθιδρύσει την αλληλεπίδρασή του με το περιβάλλον, οι συναρτήσεις πρόβλεψης και επικαιροποίησης του φίλτρου μπορούν να καλεστούν, σύμφωνα με την ροή εντολών της εφαρμογής. Οι δυναμικές εκτιμήσεις του φίλτρου είναι διαθέσιμες σε οποιαδήποτε εμβέλεια μεταβλητών έχει πρόσβαση στην κεντρική δομή δεδομένων του φίλτρου. Τέλος, υποστηρίζεται η προαιρετική έγχυση νέας κατάστασης, κατά την βούληση του κώδικα του χρήστη.

Όπως έχει ήδη προαναφερθεί, στην παρούσα εργασία κάνουμε χρήση της Ανάστροφης Αυτόματης Διαφόρησης (Griewank, 1992) με στόχο την αυτοματοποίηση

της εύρεσης των παραγώγων που απαιτούνται για το γέμισμα των Ιακωβιανών πινάκων που χρειάζεται το εκτεταμένο φίλτρο Kalman. Η τεχνική αυτή υλοποιείται στο οπίσθιο τμήμα του μεταγλωττιστή της Newton ως αλγόριθμος τριών περασμάτων που μεταλλάσει το αφηρημένο συντακτικό δέντρο των σχετικών εκφράσεων. Στο πρώτο πέρασμα κάθε κόμβος του δέντρου προσαυξάνεται με πληροφορία με πληροφορία για τις ονομασίες των μεταβλητών των ενδιάμεσων τιμών της αποτίμησης της έκφρασης. Στο δεύτερο πέρασμα παράγεται ο κώδικας σε μορφή στατικής ατομικής ανάθεσης Static Single Assignment (Alpern, Wegman, & Zadeck, 1988; Braun κ. συν., 2013) της αρχικής μορφής της έκφρασης, ενώ στο τρίτο παράγεται ο αντίστοιχος κώδικας για την αποτίμηση της Ανάστροφης Αυτόματης Διαφόρησης. Τα τελευταία 2 περάσματα, που εμπεριέχουν παραγωγή κώδικα, συνιστούν την συνολική παραχθείσα έκφραση C, η οποία υπολογίζει την κανονική τιμή της αρχικής έκφρασης, καθώς και τις μερικές παραγώγους της ως προς τις μεταβλητές της εισόδου, στην ίδια κλήση.

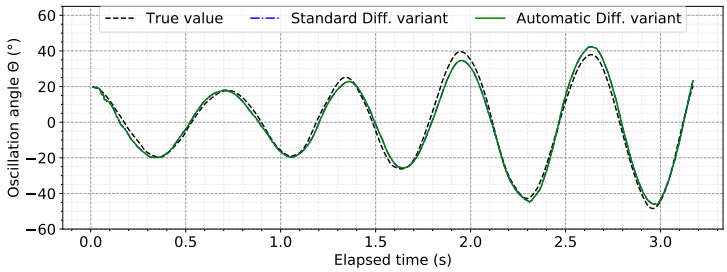
4 Πειραματική Αξιολόγηση

Αξιολογούμε τα αυτομάτως παραχθέντα φίλτρα Kalman σε δυναμικά συστήματα αύξουσας πολυπλοκότητας. Αρχικά, ξεκινάμε με μια αξιολόγηση σε προσομοίωση εκκρεμούς και έπειτα περνάμε σε αξιολόγηση ρομποτικού συστήματος δύο τροχών, εδάφους. Συγκρίνουμε τα αποτελέσματα για οριακή μέθοδο διαφοροποίησης και αυτόματη διαφοροποίηση. Στα πειράματα χρησιμοποιούμε $h = 0.0005$ για την οριακή μέθοδο διαφοροποίησης, εκτός αν αναφέρεται διαφορετικά.

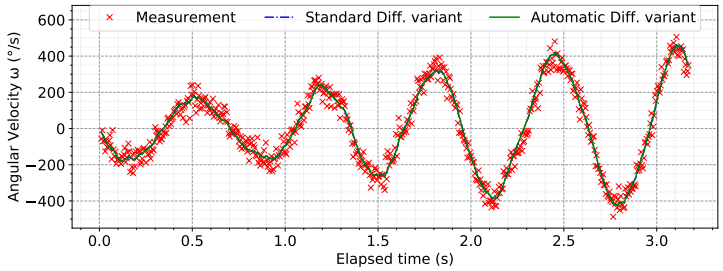
4.1 Προσομοίωση Εκκρεμούς

Αναπτύξαμε περιγραφές Newton για σύστημα εκκρεμούς, με ή χωρίς απόσβεση, τα οποία δώσαμε ως είσοδο στον μεταγλωττιστή της Newton για να παράξει υλοποιήσεις εκτιμητών κατάστασης. Η προδιαγραφή Newton που παρέχουμε βασίζεται στην Εξίσωση 2.35, και αναπτύσσεται με μεγαλύτερη λεπτομέρεια στην Ενότητα 4.4.1. Ως κατάσταση, στο πλαίσιο των δοκιμών του εκκρεμούς, θεωρούμε την γωνία ταλάντωσης και την γωνιακή ταχύτητα ταλάντωσης, τα οποία σημειώνουμε ως $< \theta_t, \omega_t >$. Η γωνιακή ταχύτητα είναι η μόνη μεταβλητή κατάσατσης την οποία μπορεί να μετρήσει το σύστημα, με ένα προσομοιωμένο γυροσκόπιο. Επομένως, το σύστημα βασίζεται καθαρά σε προβλέψεις και βελτιώσεις για την μεταβλητή της γωνίας. Ο θόρυβος του γυροσκόπιου είναι μοντελοποιημένος σύμφωνα με την κανονική (Γκαουσιανή) κατανομή (Kaparounakis κ. συν., 2020).

Το πρώτο πείραμα που εκτελούμε για το εκκρεμές θεωρούμε ότι δεν υπάρχει απόσβεση. Προσομοιώνουμε το σύστημα με αντιληπτή τάξη μεγέθους διεργασιακού θορύβου και κρατάμε αρχεία με τα ίχνη της αληθινής κατάστασης και των μετρήσεων των αισθητήρων, οι οποίες έχουν επιπλέον θόρυβο μέτρησης. Χρησιμοποιούμε στα παραχθέντα φίλτρα Kalman για να εκτιμήσουμε την κίνηση του εκκρεμούς, έχοντας πρόσβαση μόνο στην θορυβώδη μέτρηση της γωνιακής ταχύτητας από το γυροσκόπιο. Τα αποτελέσματα του πειράματος παρουσιάζονται στο Σχήμα 3. Τα Σχήματα 3(α′)

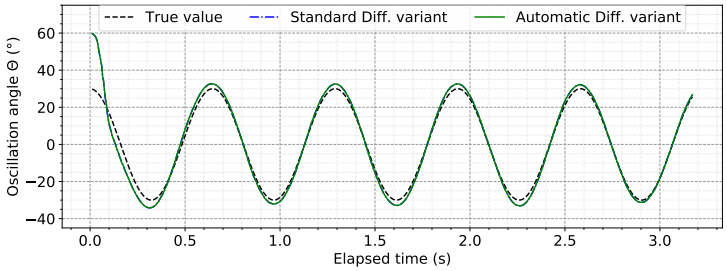


(α′) Πείραμα 1: Εκτίμηση γωνίας ταλάντωσης ως προς το χρόνο.

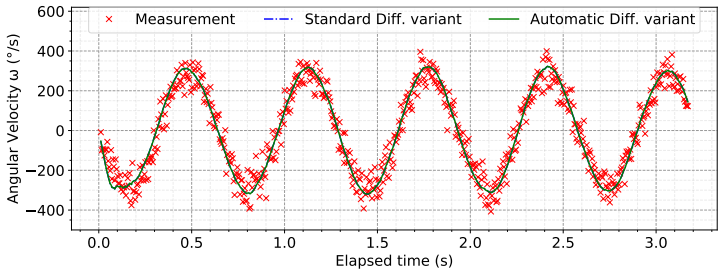


(β′) Πείραμα 1: Εκτίμηση γωνιακής ταχύτητας ως προς το χρόνο.

Σχήμα 3: Αξιολόγηση του παραχθέντος φίλτρου Kalman για την εκτίμηση της κατάστασης του εκκρεμούς. Τα υποσχήματα (α′) και (β′) αντιστοιχούν σε ταλάντωση με αρχική γωνία $20°$ και αντιληπτό διεργασιακό θόρυβο διακύμανσης $0.005\,\mathrm{rad}^2/s^2$ στην γωνιακή ταχύτητα της ταλάντωσης, ορατός στην μέγιστη γωνία ταλάντωσης ως προς το χρόνο. Η είσοδος στα φίλτρα είναι η μέτρηση της γωνιακής ταχύτητας με θόρυβο διακύμανσης $0.5\,\mathrm{rad}^2/s^2$. Όλα τα παραχθέντα φίλτρα καταφέρνουν να ακολουθήσουν επιτυχώς την διεργασία, παρά τον θόρυβο, με μέσο τετραγωνικό σφάλμα $0.0025\,\mathrm{rad}^2$ για την γωνία και $0.17\,\mathrm{rad}^2/s^2$ για την γωνιακή ταχύτητα.



(α′) Πείραμα 2: Εκτίμηση της γωνίας ταλάντωσης ως προς το χρόνο.

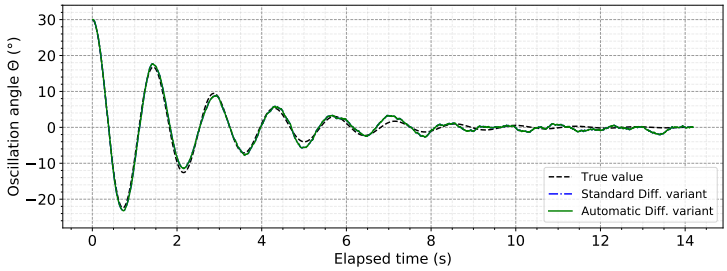


(β′) Πείραμα 2: Εκτίμηση της γωνιακής ταχύτητας ως προς το χρόνο.

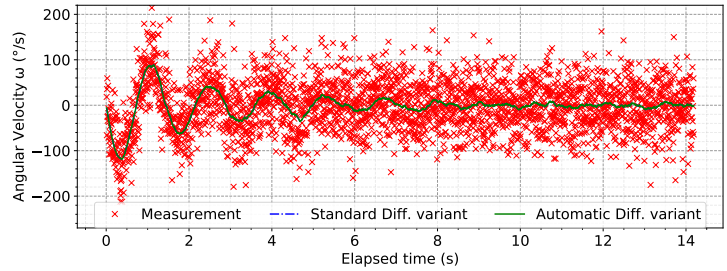
Σχήμα 4:
Τα υποσχήματα 4(α′) και 4(β′) αντιστοιχούν σε προσομοίωση του εκκρεμούς με γωνία ταλάντωσης 30°, διακύμανση θορύβου μέτρησης $0.8\,\mathrm{rad}^2/s^2$. Τα φίλτρα που μελετήθηκαν αρχικοποιήθηκαν επιτηδευμένα με εσφαλμένη αρχική γωνία 60°. Παρ′ αυτά, καταφέρνουν να συγκλίνουν σε ακριβείς εκτιμήσεις για την κατάσταση του εκκρεμούς.

και 3(β′) παρουσιάζουν ένα διάστημα εκτέλεσης της προσομοίωσης, με αρχική γωνία ταλάντωσης 20° και αντιληπτό διεργασιακό θόρυβο διακύμανσης $0.005\,\mathrm{rad}^2/s^2$ στην γωνιακή ταχύτητα της ταλάντωσης. Οι μετρημένες τιμές από το γυροσκόπιο εμφανίζονται στο Σχήμα 6.1(β) ως κόκκινα σημεία, ενώ επίσης εμφανίζονται οι προβλέψεις για την γωνιακή ταχύτητα από τα φίλτρα Kalman. Στο Σχήμα 3(α′) εμφανίζεται η πρόβλεψη για την γωνία ταλάντωσης παράλληλα με την αληθινή της τιμή. Παρατηρούμε ότι και οι δύο εκδοχές των φίλτρων επιτυγχάνουν εκτίμηση της γωνίας με μέσο τετραγωνικό σφάλμα $0.003\,\mathrm{rad}^2$.

Στο δεύτερο πείραμα που εκτελούμε θεωρούμε ξανά ότι δεν υπάρχει απόσβεση στο εκκρεμές. Αρχικοποιούμε τα παραχθέντα φίλτρα Kalman με εσφαλμένη αρχική τιμή. Έτσι αξιολογούμε την ικανότητα των φίλτρων να συγκλίνουν στην σωστή διεργασία παρά τις εσφαλμένες αρχικές υποθέσεις. Η αρχική γωνία ήταν 30° ενώ η αρχικοποίηση των φίλτρων τέθηκε στις 60°. Παραθέτουμε τα αποτελέσματα της αξιολόγησης ετούτου του πειράματος στα Σχήματα 4(α′) και 6.2(β). Παρατηρούμε στο Σχήμα 6.2(α) ότι αμφότερες οι εκδοχές των παραχθέντων φίλτρων επιτυγχάνουν την σύγκλιση στην αληθινή διεργασία. Τα μέσα τετραγωνικά σφάλματα για την γωνία και την γωνιακή ταχύτητα ήταν $0.0056\,\mathrm{rad}^2$ και $0.1757\,\mathrm{rad}^2/s^2$ αντίστοιχα, και για



(α′) Πείραμα 3: Εκτίμηση της γωνίας ταλάντωσης ως προς το χρόνο.



(β′) Πείραμα 3: Εκτίμηση της γωνιακής ταχύτητας ως προς το χρόνο.

Σχήμα 5: Στατε εστιματιον οφ α πενδυλυμ οσχιλλατιον ωιτη δραγ. Ινιτιαλ δισπλαχεμεντ ις εχυαλ το 30° ανδ μεασυρεμεντ νοισε αριανχε $0.8\,\mathrm{rad}^2/s^2$. Βοτη γενερατεδ φιλτερ αριαντς, ι.ε., ωιτη στανδαρδ ορ Αυτοματις Διφφερεντιατιον, προιδε ηιγη αχχυραχψ εστιματες οφ τηε οσχιλλατιον ωιτη διμινισηινγ αμπλιτυδε.

τις δύο εκδοχές των φίλτρων.

Στο τρίτο πείραμα που εκτελούμε θεωρούμε ότι το εκκρεμές έχει απόσβεση ίση με $0.8\,kg\,s^{-1}$. Η αρχική γωνία ταλάντωσης ήταν 30° και οι δύο εκδοχές του φίλτρου είχαν σωστή εκτίμηση αρχικοποίησης. Τα αποτελέσματα αυτού του πειράματος παρατίθενται στο Σχήμα 5. Η διακύμανση θορύβου μέτρησης ήταν $0.8\,\mathrm{rad}^2/s^2$. Παρατηρούμε ότι και οι δυο εκδοχές των παραχθέντων φίλτρων Kalman είναι ικανές να κάνουν ακριβείς προβλέψεις για την κατάσταση του συστήματος, παρά τον θόρυβο των μετρήσεων που εμφανίζονται ως κόκκινα σημεία στο Σχήμα 5(β′). Το μέσο τετραγωνικό σφάλμα και για τα δύο φίλτρα ήταν περίπου $0.0002\,\mathrm{rad}^2$ για την γωνία και $0.0054\,\mathrm{rad}^2/s^2$ για την γωνιακή ταχύτητα.

4.2 Ρομπότ TurtleBot3 Burger

Αξιολογούμε την αποδοτικότητα του συστήματός μας παράγοντας φίλτρο εκτίμησης κατάστασης για ένα πιο πολύπλοκο φυσικό υπολογιστικό σύστημα, ένα ρομπότ δυναμικής κίνησης. Συγκεκριμένα χρησιμοποιούμε το ρομπότ διαφορικής οδήγησης δύο τροχών TurtleBot3 Burger (Amsters & Slaets, 2019), το οποίο υποστηρίζει το

Robot Operating System (Quigley κ. συν., 2009).

Παράγουμε το φίλτρο εκτίμησης κατάστασης για το ρομπότ γράφοντας μια αντίστοιχη περιγραφή Newton. Το κινηματικό μοντέλο της διεργασίας του ρομπότ παρουσιάζεται στην Ενότητα 2.4.3 ενώ η συγγραφή της αντίστοιχης περιγραφής Newton παρουσιάζεται στην Ενότητα 4.4.2.

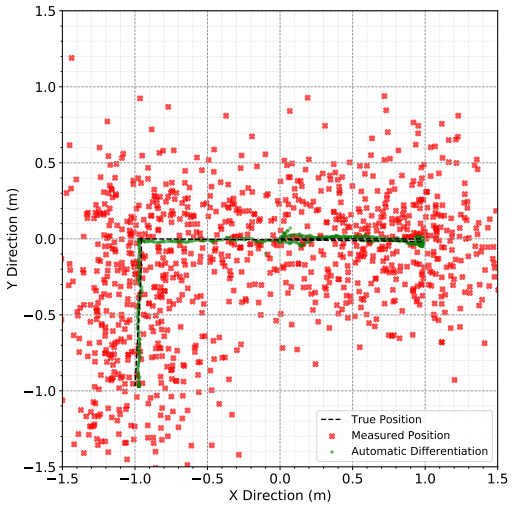
Το μοντέλο μέτρησης λαμβάνει θορυβώδη πληροφορία για την θέση στο επίπεδο και την γωνιακή ταχύτητα του ρομπότ ως προς τον άξονα που περνάει από το κέντρο του και είναι κάθετος ως προς το επίπεδο κίνησής του. Η διακύμανση των μετρήσεων στο επίπεδο $\mathrm{X\Upsilon}$ ήταν $0.1\,m^2$. Προσομοιώσαμε μια βόλτα του ρομπότ στο πρόγραμμα προσομοίωσης Gazebo (Koenig & Howard, 2004) και αξιολογούμε την συμπεριφορά των φίλτρων πάνω σε συλλεγμένα ίχνη από την προσομοίωση.

Σε αυτό το πείραμα αξιολογούμε συγκεκριμένα το φίλτρο που υλοποιεί Ανάστροφη Αυτόματη Διαφόριση. Θεωρούμε την κατάσταση του ρομπότ ως το διάνυσμα $< x_t, y_t, \theta_t >$, δηλαδή την θέση ως προς τον άξονα x, την θέση ως προς τον άξονα y και την περιστροφή του ρομπότ, αντίστοιχα.

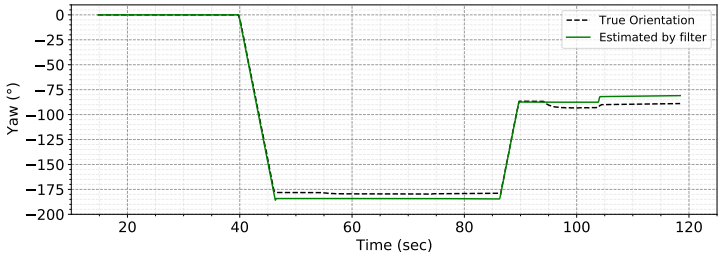
Στο Σχήμα 6 παρουσιάζονται τα αποτελέσματα της εκτίμησης για την θέση (Σχήμα 6(α′)) και την περιστροφή (Σχήμα 6(β′)) του ρομπότ, ως πράσινες γραμμές. Οι μετρηθείσες θέσεις του ρομπότ παρουσιάζονται ως κόκκινα σημεία στο Σχήμα 6(α′). Το ρομπότ ξεκίνησε την βόλτα του στο σημείο $(0, 0)$ στραμμένο προς την θετική πλευρά του άξονα x. Του δώσαμε την οδηγία να κινηθεί ευθεία προς το σημείο $(1, 0)$, να στρίψει 180° δεξία, και να συνεχίσει προς το σημείο $(-1, 0)$. Εκεί, να στρίψει 90° αριστερά και να σταματήσει την κίνησή του στο σημείο $(-1, -1)$. Η διακεκομμένη μαύρη γραμμή στο Σχήμα 6(α′) αντιστοιχεί στην αληθινή πορεία του ρομπότ. Παρατηρούμε ότι το παραχθένα φίλτρο Kalman παρέχει μεγάλη ακρίβεια εκτίμησης παρά το θόρυβο των μετρήσεων. Η μέση ευκλείδια απόσταση σφάλματος της εκτίμησης της θέσης ήταν $0.0185\,m$, ενώ το μέσο σφάλμα εκτίμησης στην περιστροφική κίνηση ήταν (Σχήμα 6(β′)) 4.72°.

4.3 Μέγεθος Αρχείου και Αριθμός Δυναμικών Εντολών

Επιπλέον της αξιολόγησης της ακρίβειας των φίλτρων για τα προαναφερθέντα φυσικά συστήματα, αξιολογούμε το σύστημά μας για να ποσοτικοποιήσουμε τις απαιτήσεις σε μνήμη και υπολογιστική ισχύ. Γι΄ αυτό αναλύουμε τα παραχθέντα φίλτρα για το εκκρεμές χωρίς απόσβεση από την Ενότητα 4.1. Επιλέγουμε την αρχιτεκτονική RISC-V ως αρχιτεκτονική αναφοράς για ένα επεξεργαστή RISC 32-bit και εξετάζουμε διάφορες επεκτάσεις του βασικού αρχιτεκτονικού συνόλου εντολών ((ISA)) για να λάβουμε υπόψη τις ενδεχομένως διαφορετικές δυνατότητες μεταξύ διαφορετικών ενσωματωμένων συστημάτων. Ξεκινάμε από το βασικό σύνολο εντολών RV32I που παρέχει μόνο αριθμητική με ακεραίους και λογικές πράξεις, έπειτα εξετάζουμε την επέκταση RV32IM που προσθέτει λειτουργικότητα πολλαπλασιασμού και διαίρεσης ακεραίων, και τελειώνουμε με τις RV32IF/RV32IFD που επεκτείνουν το σύνολο εντολών με πράξεις δεκαδικών αριθμών κινητής υποδιαστολής μονής και διπλής α-

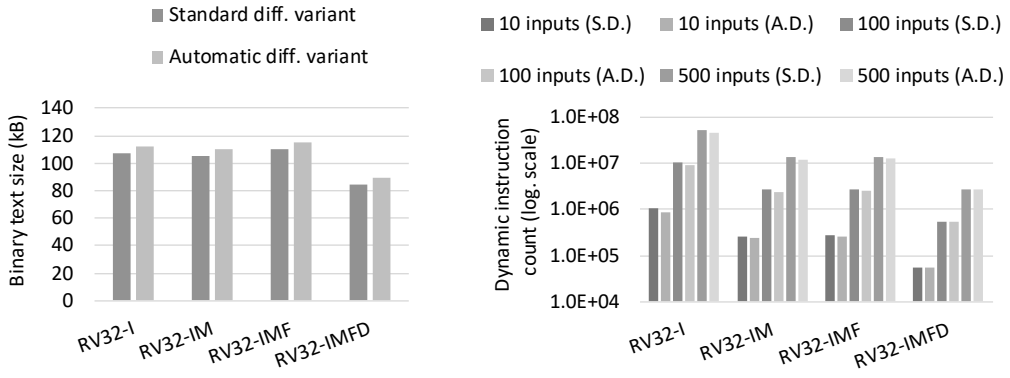


(α′) Αληθής διαδρομή και εκτίμηση διαδρομής του TurtleBot3.



(β′) Εκτίμηση περιστροφικής κίνησης του TurtleBot3 με γυροσκόπιο.

Σχήμα 6: Πρόβλεψη του μονοπατιού του ρομπότ TurtleBot3Πρεδιστιον οφ τηε πατη οφ τηε ΤυρτλεΒοτ3 Βυργερ ροβοτ. Ξεκινάει στο σημείο $(0, 0)$, κοιτώντας προς την θετική πλευρά του άξονα x. Κινείται ευθεία μπροστά προς το σημείο $(1, 0)$, όπου στρίβει $180°$ δεξία. Συνεχίζει στο $(-1, 0)$, όπου στρίβει $90°$ αριστερά και τελειώνει την κίνησή του στο $(-1, -1)$. Η κίνηση αυτή φαίνετια στο υπο-σχήμα (α′). Το υπο-σχήμα (β′) δείχνει την πραγματική περιστροφή του ρομπότ καθώς και την αντίστοιχη εκτίμηση του φίλτρου.



(α΄) Δυαδικό μέγεθος αρχίου σε kB. **(β΄)** Αριθμός εντολών δυναμικής εκτέλεσης (λογ. κλίμακα).

Σχήμα 7: Ανάλυση των μεγαλωττισμένων δυαδικών αρχείων του πηγιαίου κώδικα των παραχθέντων φίλτρων για τις διάφορες επεκτάσεις RISC-V ISA. Στο υπο-σχήμα (α΄) φαίνεται το μέγεθος του κομματιού text των δυαδικών αρχείων, δείχνοντας μια ελαφριά αύξηση για τα φίλτρα με την Αυτόματη Διαφόριση. Το υπο-σχήμα (β΄) δείχνει τον απαιτούμενο αριθμό εντολών δυναμικής εκτέλεσης που χρειάστηκε κάθε φίλτρο, όπου και τα φίλτρα με Αυτόματη Διαφόριση επιτυγχάνουν μια σημαντική μείωση σε σύγκριση με την οριακή μέθοδο διαφόρισης.

κρίβειας, αντίστοιχα.

Το μέγεθος του κομματιού text του μεταλγωττισμένου αρχείου για τις διαφορετικές επεκτάσεις της αρχιτεκτονικής RISC-V παρουσιάζεται στο Σχήμα 7(α΄). Η οριακή και η αυτόματη διαφόριση υποσημειώνονται στα σχήματα με "S.D." και "A.D.", αντίστοιχα. Έκαστη μπάρα στο Σχήμα 7(β΄) αντιστοιχεί σε διαφορετικό φόρτο εργασίας, δηλαδή διαφορετικό συνολικό αριθμό εισόδων στα φίλτρα.

Παρατηρούμε ότι σε όλες τις περιπτώσεις τα φίλτρα με την Αυτόματη Διαφόριση είναι ελαφρώς μεγαλύτερα σε μέγεθος κειμένου (text size) αλλά απαιρούν λιγότερες δυναμικές εντολές την ώρα της εκτέλεσης. Στην Αυτόματη Διαφόριση προκύπτει ένα κέρδος 7.5% στον αριθμό εντολών δυναμικής εκτέλεσης, ξεπερνώντας το 16% στο μεγιστό του στην περίπτωση του βασικού συνόλου εντολών RV32I. Το ώφελος αυτό έρχεται σε αντάλλαγμα με περίπου 4.7% αύξηση του μεγέθους αρχείου.

Αποδίδουμε την αύξηση στο μέγεθος αρχείου καταρχήν στην επέκταση των συναρτήσεων των μοντέλων σε μορφή SSA και κατέπέκταση στην επαύξησή τους με τον κώδικα σε μορφή SSA της Ανάστροφης Αυτόματης Διαφόρισης. Η μείωση του δυναμικού αριθμού εντολών αποδίδεται στην χρήση της Δυτόματης Διαφόρισης, με την οποία όλες οι τιμές των μερικών παραγώγων κάθε γραμμής του Ιακωβιανού πίνακα παράγονται με μία αποτίμηση. Εν αντιθέσει, στην οριακή διαφόριση απαιτούνται δύο παραπάνω αποτιμήσεις για κάθε θέση του Ιακωβιανού πίνακα.

5 Σύνοψη και Μελλοντική Εργασία

Σε αυτήν την εργασία παρουσιάζουμε μια πρόοδο στην σύγχρονη αυτόματη σύνθεση αλγορίθμων εκτίμησης κατάστασης. Η μέθοδος που παρουσιάζουμε ξεκινάει από μια περιγραφή των φυσικών προδιαγραφών και των νόμων που διέπουν ένα φυσικό ενσωματωμένο αισθητήριο υπολογιστικό σύστημα. Η μέθοδος παράγει ως έξοδο κώδικα σε C με μικρό αποτύπωμα μνήμης και υπολογιστικές απαιτήσεις, κατάλληλο για αξιοποίηση σε συστήματα με περιορισμένους πόρους. Η αυτοματοποίηση της μεθόδου, ως οπίσθιο τμήμα μιας γλώσσας προδιαγραφής φυσικών νόμων, μειώνει τον χρόνο και την πιθανότητα σφαλμάτων της, παραδοσιακά, χρονοβόρας και επιρρεπούς σε λάθη διαδικασίας του σχεδιασμού και της υλοποίησης απλών και εκτεταμένων φίλτρων Kalman (Kaparounakis κ. συν., 2020).

Η μέθοδος που παρουσιάζουμε μπορεί εύκολα να επεκταθεί για να υποστηρίζει και άλλους αλγόριθμους εκτίμησης κατάστασης όπως είναι τα unscented φίλτρα Kalman και τα φίλτρα σωματιδίων, ή διαφορετικές προσεγγίσεις σε υποσυστήματα αυτών των φίλτρων, όπως η χρήση πίνακα πληροφορίας αντί για πίνακα συνδιακύμανσης (Terejanu, 2013), ενεργοποιώντας έτσι την ταχύτατη σύγκριση μεταξύ διαφορετικών φίλτρων και εσωτερικών επιλογών για ένα ενσωματωμένο σύστημα. Επιπλέον, η υλοποίηση του συστήματος ως μέρος ενός μεταγλωττιστή αυξάνει τις δυνατότητες για στατικούς ελέγχους και βελτιστοποιήσεις.

Η υλοποίησή μας χρησιμοποιεί την μέθοδο της Ανάστροφης Αυτόματης Διαφόρησης (Griewank κ. συν., 1989), η οποία χαίρει μεγάλης πρόσφατης υιοθέτησης στον τομέα της μηχανικής μάθησης (Baydin κ. συν., 2017), για την αυτόματη παραγωγή των μερικών παραγώγων των Ιακωβιανών πινάκων των μοντέλων των φίλτρων. Χρησιμοποιώντας συστήματα διαφορετικής πολυπλοκότητας, αξιολογούμε την απόδοση της μεθόδου εξετάζοντας παραχθέντα φίλτρα σε όρους ακρίβειας, ευστάθειας, σύγκλισης, καθώς και απαιτήσεων κατά την εκτέλεση, στο πλαίσιο της αξιοποίησης τους σε φυσικά υπολογιστικά συστήματα περιορισμένων πόρων. Επιβεβαιώνουμε, ότι η εκδοχή των φίλτρων που χρησιμοποιεί την Αυτόματη Διαφόριση λειτουργεί εντός των ίδιων ορίων σφάλματος με παραδοσιακές μεθόδους, ενώ ταυτόχρονα υπολογίζει με ακρίβεια τις τιμές των απαιτούμενων μερικών παραγώγων.

Το σύστημα που έχουμε παροσιάζει μπορεί να χρησιμοποιηθεί σαν βάση για μελλοντική έρευνα που θα εξερευνεί τις διάφορες επιλογές για εκτίμηση κατάστασης στο πλαίσιο του σχεδιασμού ενσωματωμένων συστημάτων. Πιο συγκεκριμένα, παρουσιάζονται παρακάτω μερικές πρώτες ιδέες για εξερεύνηση.

Ένα σημαντικό ζήτημα που εμφανίζεται στην χρήση των φίλτρων Kalman είναι επιλογή για τους πίνακης της συνδιακύμανσης του εμπλεκόμενου θορύβου. Το σύστημα που παρουσιάστηκε μπορεί να επεκταθεί ώστε να εκτελεί αλγόριθμους αναγνώρισης θορύβου ώστε να εμπλουτίζει τους πίνακες συνδιακύμανσης με εμπειρική γνώση (Åkesson, Jørgensen, Poulsen, & Jørgensen, 2008; Odelson, Rajamani, & Rawlings, 2006). Εκτός των στοχαστικών στοιχείων, είναι φυσική η επέκταση του συστήματος για την αναγνώριση δυναμικών χαρακτηριστικών ενός συστήματος (Chui, Chen, κ. συν., 2017)

Είναι φυσική η επέκταση του συστήματος ώστε να δέχεται στην είσοδο διαφορικές εξισώσεις και να εξάγει, με αυτόματο τρόπο, τα μοντέλα στην μορφή συνάρτησης μεταφοράς και, κατ΄ επέκταση, εξισώσεων κατάστασης. Γενικά, γίνεται να αυτοματοποιηθεί, εώς κάποιο βαθμό, η αυτόματη μετατροπή μιας διαφορικής εξίσωσης αντίστοιχες εξισώσεις κατάστασης. Ωστόσο, δεν φαίνεται να υπάρχει κάποια απλή γενίκευση σε περίπτωση πολλαπλών διαφορικών εξισώσεων, πέραν της ξεχωριστής ανάλυσης και, εν τέλει, σύνδεσης.

Στην απλή τους μορφή, οι πράξεις πινάκων που εμπεριέχονται στα φίλτρα Kalman ενδεχομένως εκτελούν αχρείαστες πράξεις ξανά και ξανά. Αυτό είναι ιδιαίτερα αντιληπτό στην περίπτωση της ενημέρωσης μετά από μέτρηση. Σε κάθε νέα μέτρηση εκτελείται εκ νέου και εξ΄ ολοκλήρου το βήμα ενημέρωσης της κατάστασης, ενώ, ενδεχομένως, μόνο ένα υποσύνολο της κατάστασης επηρεάζεται από μια δεδομένη μεταβλητή μέτρησης. Μια περιοχή ανάπτυξης του συστήματος που παρουσιάστηκε θα ήταν ο εντοπισμός αυτών το σχέσων στην στατική φάση της μεταγλώττισης, ώστε να τρέχουν μόνο οι απαραίτητες πράξεις κατά την δυναμική εκτέλεση του παραχθέντος φίλτρου.

Κλείνοντας, μια εναλλακτική μορφή του φίλτρου Kalman είναι το φίλτρο πληροφορίας (Khan, 2005; De Jong κ. συν., 1991), στο οποίο ο πίνακας συνδιακύμανσης των μεταβλητών κατάστασης αντικαθίσταται από έναν πίνακα πληροφορίας. Με αυτήν την αλλαγή, προκύπτει ότι η ενσωμάτωση πολλαπλών νέων μετρήσεων καταλήγει στην πρόσθεση των αντίστοιχων πινάκων πληροφορίας, εκεί που αντιστοίχως ο κλασικό αλγόριθμος εμπεριέχει πολλαπλασιασμό και αντιστροφή πινάκων. Το παραπάνω ανταλλάσει αυξημένη πολυπλοκότητα του βήματος πρόβλεψης για χαμηλότερη πολυπλοκότητα στο βήμα ενημέρωσης. Η ενσωμάτωση αυτής της εναλλακτικής, στο σύστημά μας, θα προσέδιδε μεγαλύτερη ευελιξία κατά την φάση του σχεδιασμού, για συστήματα που ενδεχομένως υλοποιούν μεγάλο αριθμό μετρήσεων στην μονάδα του χρόνου.

Chapter 1

Introduction

Any sufficiently advanced technology is indistinguishable from magic.

– Arthur C. Clarke

In the era of the undeniably ubiquitous computing it is easy to forget the amount of work that has been poured into making all this "just work". The facts that airplanes fly, cars drive, smartphones communicate are often taken for granted. Each of these, in order to succeed, mantled engineering evolutions in its respective field. There is, however, a common axis; they all interact with the physical environment. Airplanes use subsystems to determine their location, altitude and attitude, as well as monitor things like oxygen level in the cabin, total load and even the operation of other subsystems. Cars employ similar principles for pretty much anything that appears on the dashboard. Lastly, smartphones constantly use sensor readings to detect movement and gestures on the screen, discern location, monitor subsystem operationality and countless other things.

Systems like the ones above tend to have a supervising operator; (usually) a human, that is in charge of making decisions based on their interpretation of the system's sensor readings but, more importantly, also based on the readings from their own sensory organs, e.g. the eyes. In the advent of automation it is investigated which parts of the interpretation of sensor readings and decision making can be left to the machine, so that the operator can focus on the bigger picture. In fact, the majority of computing systems use data from sensors to drive control decisions.

This kind of functionality is the main object of study in Control Theory. The systems are abstracted to mathematical models and can be theoretically studied in terms of performance, correctness, stability and accuracy. The result from these analyses might be a controller, a system in which the logic of the aforementioned automation is implemented. In a typical scenario, the controller observes the situation of the environment (i.e. readings from sensors) and performs the necessary

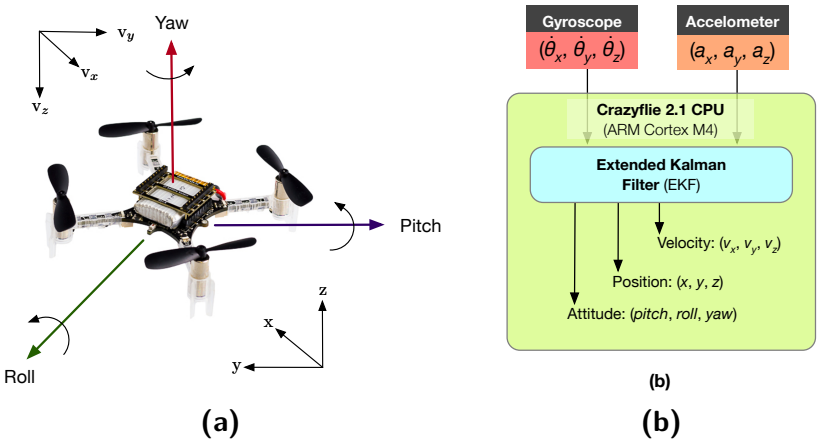


Figure 1.1: **(a)** The CrazyFlie 2.1 micro-UAV. **(b)** The micro-UAV combines noisy readings from its sensors (angular rate $\dot{\theta}$ and acceleration a in three dimensions) to obtain a stable estimate of the state parameters *pitch*, *roll*, *yaw*, position (x, y, z), and velocity (v_x, v_y, v_z).

actuation with the aim of reaching a target input situation. This "situation", which consists of a given configuration or circumstances of the system, is often called the "state".

There's a catch, though. All kinds of measurements have some degree of measurement uncertainty (*measurement noise*) and, thus, computing systems that consume sensor data must compensate for this. In addition, the formulated physical laws used to describe the physics of the system in question can also have deviations with time (*process noise*). Due to these, such systems often use techniques, ranging from averaging or filtering (Oppenheim & Schafer, 1983), to more sophisticated state-estimation techniques (R. Kalman, 1960; Jazwinski, 1969; Wan & Van Der Merwe, 2000; Arulampalam, Maskell, Gordon, & Clapp, 2002), to combine the signals from multiple sensors in an effort to obtain improved noise rejection.

Drone Example

Let's discuss an example of a small modern micro-unmanned aerial vehicle (micro-UAV), the CrazyFlie 2.1 (Giernacki κ. συν., 2017). Figure 1.1(a) shows the physical layout of the drone (Kaparounakis κ. συν., 2020). The micro-UAV weighs 27 g and its control system comprises an ARM Cortex-M4 microcontroller with 192kB RAM. The system's flight control uses data from an inertial measurement unit (IMU) for 3-axis acceleration and 3-axis angular rate measurement as well as a high precision pressure sensor for elevation monitoring. The micro-UAV's control system uses a Kalman filter to fuse the (noisy) readings from the seven dimensions of these sensors to generate a stable pitch, roll, yaw, and elevation estimate in real time (Mueller, Hamer, & D'Andrea, 2015; Mueller, Hehn, & D'Andrea, 2016).

Bringing this information into our previous framework; the drone is a *system*

that interacts with the environment. The drone's *sensors* are constantly providing the drone firmware with new measurements, that, apart from the true signal, also unavoidably contain background interference or *measurement noise*. The firmware uses these readings to try to discern the *state* of the drone, which comprises the drones attitude, position and velocity. This is achieved by *combining* together the measurements and knowledge of the system's dynamics using a state-estimation *technique* (namely, an extended Kalman filter), an illustration of which appears in Figure 1.1(b). The algorithm also takes into account unforeseeable random variables, e.g. local winds, represented as *process noise*. The *controller* is constantly informed of the current state and uses it in the propeller actuation logic. Notice that while the sensors measure only linear acceleration and angular velocity the state contains their antiderivatives.

1.1 Problem Statement

The behavior of physical systems deviates from the intended behavior with time (*process noise*) and measurements of signals which influence the state of a system will always have some degree of measurement uncertainty (*measurement noise*). State estimation methods estimate the future behavior or state of a system in the presence of process noise and measurement noise. In the absence of process noise, a fixed time-independent model of a system's behavior would be able to be derived and would suffice. Similarly, in the absence of measurement noise and given a model of system state as a function of external signals, one could estimate state perfectly (Kaparounakis κ. συν., 2020).

State estimation algorithms, as are the Kalman filters and particle filters, use a combination of models of the dynamics of a system, models of the relations between the sensor signals and the system state, and models of the noise in the system to calculate improved estimates of state variables in the presence of process noise and measurement noise.

That said, implementing state-estimation algorithms requires system-specific knowledge of the physical properties of the sensor-instrumented system (Barfoot, 2017). Their implementations also typically require system-specific knowledge of the constraints imposed by physics on sensor signals. When implementing state estimation methods on resource-constrained embedded systems, this knowledge of the physics of the system is however just one part of the challenge: System designers must combine their physical understanding with efficient implementations of the core linear-algebraic methods or nonlinear dynamical systems, often in a low-level language such as C, for a hardware platform that has only tens or hundreds of kilobytes of memory and thus may not be able to host common libraries such as Eigen (Guennebaud, Jacob, κ. συν., 2010). As a result, state estimation algorithms for resource-constrained embedded systems are challenging to implement correctly, challenging to implement efficiently, and even more challenging to implement quickly. Additionally, because the implementations of state estimation

algorithms can depend on the physical properties of a system and its sensors in complex ways, small changes to the underlying assumptions of the state estimation algorithms can require significant changes to the state estimation algorithms and their implementation.

1.2 Related Research

Automating Kalman filter design and implementation process is a task of high importance for engineers in various domains. As a result, MathWorks MATLAB & Simulink and GNU Octave both provide libraries that assist the design of Kalman filters (Grewal & Andrews, 2014; Särkkä, 2013). MATLAB[1] provides the Kalman filter, the extended Kalman filter, the unscented Kalman filter and the particle filter as blocks for Simulink. Functions for each are also available for scripts. These return objects which are passed to other functions that initialize the filter, make estimate predictions or update the estimate.

MATLAB provides the ability for automatically generate C/C++ source code for filters. This is also available for Simulink blocks, as long as other blocks are also able to be generated into C/C++ code. MATLAB imposes some limitations on the filter design. For example, it is assumed that the measurement rate is constant or that the Jacobian matrices (for the extended Kalman filter) are already available. We could say that the main limitations of MATLAB's approach is that it targets a very broad user spectrum, thus, not being able to cover the "niche" needs of an embedded system platform on its own. It is assumed that, after a filter is generated into C/C++ from MATLAB, an engineer will spend time on it to fit it into the constraints of the target platform. Last but not least, it must be noted that MathWorks MATLAB is proprietary software and, as such, not widely available in all embedded system development environments.

GNU Octave[2], on the other hand does not offer a built-in estimation library. A `control`[3] package is available through OctaveForge, that includes functionality for modelling controller and observer systems. The package is, however, limited to linear time-invariant systems. Octave, though, does not offer an automated code generation solution, requiring the manual implementation of the filter in the post-design phase. GNU Octave and the `control` package are free software under the GNU GPL.

The recent rise in the popularity of the python programming language for machine learning and system scaffolding has driven the appearance of libraries, modules and resources related to state estimation and sensor fusion in the python

[1]The following information is taken from the online documentation of MATLAB R2020a.

[2]Information is taken from the online documentation of Octave version 5.2.0.

[3]Information is taken from the online documentation of version 3.2.0 of the package.

ecosystem. The Python Control Systems Library[4], available as the python module "`control`" with `pip`, is a first step in that direction. The library's direct support of variant Kalman filter algorithms is limited; it offers the building blocks for implementing them.

A more complete python package, specifically for Kalman filtering, is also available (Labbe, 2015), offering a wide range of filters and great documentation. This package is more focused on pedagogy rather than implementation efficiency. None of these two packages directly offer code generation that targets C or C++. Of course, python code is in most cases not a suitable solution for the majority of embedded applications, as these require small code size and small memory usage.

1.2.1 The AutoFilter system by NASA Ames

The most prominent related research is the AutoFilter (Whittle & Schumann, 2004; Richardson & Wilson, 2006), a closed-source tool developed in the NASA Ames Research Center, which supports code generation for the linear and linearized Kalman filter, the extended Kalman filter (EKF), as well as parallel banks of filters. The available targets are C/C++ (MATLAB, Octave or bare-metal), Modula II and MATLAB script.

AutoFilter offers abstractions for transformations such as linearization and transforming the system from continuous to discrete. The generated code can be in the form of C/C++, Modula-II source code, or a MATLAB script. AutoFilter relies on an input grammar, which supports the definition of constants, input data, datatypes, vectors, matrices and distributions (Whittle & Schumann, 2004). The target model in this grammar is declared as a list of equations, a choice which favors the description of non-linear models but is unintuitive in the case of linear systems. Furthermore, the tool supports differential equations in the input grammar in the specific form of equations of differentials of state variables on the left-hand side and an algebraic expressions on the right-hand side. The tool also features a rewrite rule engine for solving common tasks such as symbolic differentiation, linearization, approximations and others.

AutoFilter, as well as the other automated solutions and libraries described in this section (Section 1.2), assumes a static time difference between execution steps. In real-world embedded systems however, the time difference between consecutive readings from one sensor may vary and readings across multiple sensors are often at different timestamps. The implementation method that we will present in this work makes no such assumptions about time steps. Our method exploits information about the physics of a system, is generalizable beyond Kalman filters, also leverages Automatic Differentiation (Baydin κ. συν., 2017; Margossian, 2019) to automate the generation of the Jacobians in the non-linear EKF.

In a more recent reiteration of AutoFilter (Richardson & Wilson, 2006), the authors acknowledge that, after a filter's code has been generated by the tool,

[4]Information is taken from the online documentation of version 0.8.3 of the library.

there is often need by the engineers to hand-tune some specific aspects manually. This, they note, includes user-written code that needs to be interleaved with the generated code (e.g. logging functionality). In our approach, we focus on providing the building blocks and maintaining transparency of internal data structures.

Overall, the purpose of the AutoFilter tool is to alleviate need for glueing together bits of code, by providing an infrastructure that negates the need for low-level programming skills. In contrast to this, our work situates itself in the opposite position; we provide a tool that alleviates the need for deep control systems knowledge in designing and implementing sensor fusion and state estimation algorithms. We ensure that our tool generates easily interfaced code suitable to be deployed on cyber-physical embedded systems.

1.2.2 Automatic Differentiation

Automatic Differentiation is by no means a new concept. It was popularized by Griewank A. (Griewank κ. συν., 1989; Griewank, 1992) and has been one of the reasons for the speedup of machine learning techniques in the decade after its theoretical establishment (Griewank & Walther, 2008). Recent works by the Machine Learning community survey the usages and implementation techniques of Automatic Differentiation (Baydin κ. συν., 2017; Margossian, 2019). There are various ways to implement A.D. and one of them is source transformation, where a pre-processor makes a pass on a source file containing expressions and also generates expressions for their derivatives.

In our work we use Reverse Mode Automatic Differentiation to calculate the derivatives of the abstract-syntax trees of expressions. We use the Newton compiler to parse expressions into ASTs which we then pass to our A.D. implementation. The final outcome is C expressions that calculate the result of the original expression as well as its derivatives. From this perspective, our work most closely identifies with the source transformation implementation of A.D., however, the Newton compiler is not a simple pre-processor but a compiler which generates complete C code from the Newton language.

There are many tools and libraries that enhance a given programming language with support for Automatic Differentiation, like PyTorch for python (Paszke κ. συν., 2017), math for Stan (Carpenter κ. συν., 2017), or JuliaDiff for Julia. Recently, there has also been research into differential programming languages; programming languages that inherently support the differentiation of any of their structures (Breuleux & van Merriënboer, 2017; Hu κ. συν., 2019; Innes κ. συν., 2019; Abadi & Plotkin, 2019).

In the context of our work we are interested in tools and libraries that specifically target the C language, since it is the lowest common denominator of embedded systems programming and, for this reason, the target language of our system. OpenAD (Utke κ. συν., 2008) is an modular open-source source transformation tool that provides a independent framework for A.D. algorithms in C/C++ and

FORTRAN77/95. That said, linking to OpenAD would impose a sizable dependency in our otherwise low-dependency system, so we have opted to implement our own A.D. backend.

1.2.3 Program synthesis

The concept of automatically generating from a simple specification a program that implements an otherwise potentially complex function is an age-old question. The derivation of a program that satisfies a given theoretical model but is the best possible with regard to a specific resource (e.g. code size) is often called superoptimization (Massalin, 1987). Of course, in our case the compiler is not actively employing any kind of program space exploration with an aim to minimize some cost function, but this is a possibility in the future.

As it stands, the code generation process implemented as part of this work is deterministic and stable as it does not use any kind of stochastic calculations (albeit the modelled system most certainly is stochastic). What this means is that the exact same input will always yield the exact same output.

1.3 Contributions of this Work

In this work we explore automating a substantial part of the development process for embedded systems that require state-estimation algorithms for their operation. Using succinct descriptions of the physical laws describing the dynamics, the physical laws relating the measurements to the measurands and the noise properties of a system, all of which are specified in a recently-developed physics description language (Lim & Stanley-Marbell, 2018), we automate the implementation of Kalman filters and extended Kalman filters in C code.

This automation is also combined with methods recently-developed in the machine learning community for Automatic Differentiation of arithmetic expressions (Baydin κ. συν., 2017; Margossian, 2019). We implement Reverse-Mode Automatic Differentiation for representations of expression abstract syntax trees (ASTs) within a compiler. This allows the system to generate filter variants that use Automatic Differentiation for the precise calculation of the Jacobian matrices involved in the filter equations.

We evaluate the generated filters in a variety of simulated physical systems and verify that they work within an acceptable error margin, comparable to manual implementations. We also evaluate the generated filters in terms of code size and verify they are within reasonable bounds for integration within resource-constrained embedded systems.

Chapter 2

Filtering

Filtering is the formal process of obtaining the desired information signal from another signal that contains interference. In the context of state-estimation, the contained interference refers to process and measurement noise. There are three approaches to this that are distinguished by the time difference between the measurements and the application of filtering on them. Firstly, when filtering is applied after the fact, after the measurement data series has been produced and making adjustments to previous values, it is referred to as smoothing. This can either be a wholly offline process, for example when some data has been measured on an embedded device but is transferred to another computing system for processing, or be happening while the measurements are still taking place, for example if when making new measurement some past, saved value, is also adjusted. Secondly, when a new measurement affects only the current values, the process is called filtering. And thirdly, when using past measurements of a process to try and determine where a new potential measurement should lie, it is called predicting or forecasting. In order to make an informed prediction, it is reasonable to either use some axiom about the expected behavior of the signal, e.g. using the dynamics of a physical system, or base it on the results of the previous observations.

The Kalman filter (R. Kalman, 1960; R. E. Kalman & Bucy, 1961) is one method to estimate the state of a dynamic system from noisy measurements of signals that are related to the state being estimated. It works by iterating between making predictions based on current estimates and updating current estimates based on new (noisy) measurements. This iterative nature of its equations allows it to be deployed in real-time systems, continuously integrating new measurements.

Historical overview

A quite famous smoothing algorithm is the method of least-squares, believed to be independently invented by C.F. Gauss and A-M Legendre in the turn of the 19th century (Sorenson, 1970). This method is, as per above, after the fact. The objective of the method is to approximate the true state by minimizing the

sum of the squares of the residuals of each equation. It should be noted that the method assumes the process to be stationary, that is, the statistical properties of the stochastic components do not vary with time. Essentially, that the mean and variance of the noise do not change.

Almost 150 years later, in the early 1940s, A. Kolmogorov and N. Wiener independently developed another estimation technique that was later called the Wiener–Kolmogorov filtering theory and served as a basis for the development of the Kalman filter. The objective of this filter was to minimize the mean square error (MSE) between the estimated and the actual process. These kind of filters are called minimum mean square error (MMSE) estimators. A significant difference of the Wiener–Kolmogorov filter over Gauss' method is that the former was formulated as an algorithm to be continuously run on an ongoing process.

R.E. Kalman, in his work on mean-square filtering, published his formulation and proof for a MMSE estimator, that has since been called the Kalman filter (R. Kalman, 1960; R. E. Kalman & Bucy, 1961). His work marks a transition from classical control theory, where the analysis and synthesis of systems was of emphatic importance, to a more modern approach, albeit also more similar to Gauss' work, that uses the differential models of the system in what is called the state-space approach.

2.1 Notation

We will now formulate the notation that will be used throughout this work. In general, there is some notational consensus apparent in literature, but, inevitably as this theory has been spanning throughout the century, there are many deviations. We will try to give an unobfuscated presentation of the theoretical equations using the following notation (Kaparounakis κ. συν., 2020). A lot of the listed notation will be restated at relevant points.

Matrices, vectors, scalars and their diacritics

We notate matrices in an uppercase math boldface font (e.g., matrix $\mathbf{H}$) and vectors in a lowercase math boldface font (e.g., $\mathbf{v}$). We notate all scalar variables in a standard math italic font (e.g., k) and scalar constants (e.g., cardinalities) in an uppercase standard math italic font (e.g., N). We notate estimates with a cap (e.g., $\hat{\mathbf{s}}_k$) and the prior, previously-known, or *a priori* assumptions with a $-$ superscript (e.g., $\hat{\mathbf{s}}_k^-$).

State and state transition

Let the state of the system being modeled be denoted with vector $\mathbf{s}$ of cardinality N and let k be an index in time. Over time, the state of a system will invariably change across time steps and we define $\mathbf{s}_k$ as the state vector at time index k. Let $\mathbf{F}$ be the $N \times N$ *state transition matrix*, a matrix that describes the transformations of state across time steps.

Control input

Many systems have an ability to influence their state (e.g., by controlling motors

that determine their propulsion). Let vector $\mathbf{u}_k$ be the vector of controllable parameters of the system, with cardinality U, at time step k. Knowledge of these controllable parameters is useful in state estimation if available, but is not essential to most state estimation methods. Let $\mathbf{B}$ be the $N \times U$ matrix that scales the elements of the *controllable parameter vector* or *control vector* $\mathbf{u}_k$. In a system for which these controllable parameters are not known, $\mathbf{B}$ is not used. For example, in a land vehicle where we can control the angular rate of wheels but we cannot directly control the linear velocity of the vehicle, the matrix $\mathbf{B}$ would be the transform mapping from wheel rotation rate to velocity.

Stochastic components and measurements

Let $\mathbf{n}_{p,k}$ be the noise vector with cardinality N (in a system with N process states) in a process at time step k and let $\mathbf{n}_{m,k}$ be the noise vector of cardinality Z (in a system with Z sensors), at time step k. Let $\mathbf{z}_k$ be the vector of cardinality Z comprising the estimates of the true underlying signals being measured by sensors, based on the current state and the assumption of the measurement noise.

2.2 State-Space System Representation

In modernity, the most common representation of a system in the context of Control Theory is the state-space representation. In contrast with earlier representations that used equations of integrals, in the state-space approach a system is formulated as a set of differential equations. Both approaches are mathematically equivalent but the state-space one seems more natural because it is frequent to describe dynamic systems with sets of differential equations in other disciplines.

The state-space representation of physical dynamic systems that interest us in this work can be formulated with two equations; a *process equation* and a *measurement equation*. The first one describes how to calculate the transition to the current state of the system $\mathbf{s}_k$ based on the previous state $\mathbf{s}_{k-1}$ and the current control input $\mathbf{u}_k$. The new state is a linear combination of two components; what the new state would have been if there were no control input (i.e. if $\mathbf{u}_k = 0$) and what it would have been if there were no previous state (i.e. if $\mathbf{s}_{k-1} = 0$). This linear combination appears as a sum in the equation. The first component is calculated by propagating the previous state $\mathbf{s}_{k-1}$ in time to time index k using a $N \times N$ matrix $\mathbf{F}$, called the *state transition matrix*. The second component is calculated by projecting the control input from the control space onto the state space using a $N \times U$ matrix $\mathbf{B}$, called the *control transition matrix*. The formulation of the above appears in Equation 2.1.

The measurement equation describes how a given state $\mathbf{s}_k$ would relate to a measurement $\mathbf{z}_k$ and the current input vector $\mathbf{u}_k$. As with the process equation (Equation 2.1), it is a linear combination of these two components. The first component essentially models how the measurand is measured from the current state using a $N \times Z$ matrix $\mathbf{H}$, called the *measurement matrix*. Note that $\mathbf{z}_k$ is not an actual measurement but rather the measurement vector of a hypothetical

measurement on $\mathbf{s}_k$ and that the relation presented is actually the inverse of what is used more often elsewhere. The second component represents a transformation from the control space onto the measurement space using a $N \times Z$ matrix $\mathbf{D}$, called the *measurement control matrix*. This component is in fact rarely used in practice, but can be useful to model controllable parameters that will directly skew a specific measurement. The formulation of the above appears in Equation 2.2.

$$\textbf{Process} : \mathbf{s}_k = \mathbf{F}\mathbf{s}_{k-1} + \mathbf{B}\mathbf{u}_k. \tag{2.1}$$

$$\textbf{Measurement} : \mathbf{z}_k = \mathbf{H}\mathbf{s}_k + \mathbf{D}\mathbf{u}_k. \tag{2.2}$$

Equations 2.1 and 2.2 constitute a model template that we can use to model various systems in order to study them. We can make predictions about the future state of a system using Equation 2.1 and we can determine how well our predictions have fared in light of actual measurements using Equation 2.2. This is the central concept in Kalman filtering, as will become apparent in the following sections.

2.2.1 Examples

Before jumping into the filters, let's first formulate the state-space representation of some simple dynamic systems as examples, in an effort to showcase Equations 2.1 and 2.2.

Elevator

Consider the physical system of a tall, strong, vertical steel beam and a robot wrapped around it that is able to both ascend and descend the beam. Let's assume that the robot employs an infinitely strong machine that uses belts that wrap around a pulley at the top of the beam to propel itself upwards and downwards. We can see that what we just described resembles an elevator.

The elevator's state $\mathbf{s}_k$ at time k contains only one *state variable*: the distance from the bottom of the beam i.e. its current height h_k. The control input $\mathbf{u}_k$ is the absolute value of the vertical speed at which the belt machine can move the elevator. The control transition matrix $\mathbf{B}$, whose cardinality in this case will be 1×1 and will only contain the scalar variable b, represents the choice of moving or not moving the elevator by being either zero or not zero. It also represents the choice for upward or downward motion by being positive or negative, respectively. We have, then, $b \in [-1, 1]$. Let v_{max} be the maximum vertical velocity achievable by the robot, Equation 2.3 presents the process equation of the elevator as discussed.

We assume that the elevator uses a time-of-flight (ToF) laser ranging sensor to measure its distance from the ground. A naive, ad-hoc, model for this measurement would be that the distance from the ground is half the distance covered by the laser light emission in the time difference between emission and detection. This laser

light is assumed to travel, unhindered, at the speed of light C. The measurement control matrix $\mathbf{D}$ is zero, because normally the vertical speed of the elevator does not affect this measurement. If the elevator was moving at speeds comparable to the speed of light, we would have to take it into account. Equation 2.4 presents the measurement model discussed above.

$$\textbf{Elevator process}: \left[h_k\right] = \left[1\right]\left[h_{k-1}\right] + \left[b\right]\left[v_{max}\right] \tag{2.3}$$

$$\textbf{Elevator measurement}: \left[\text{tof}_k\right] = \left[2/C\right]\left[h_k\right] + \left[0\right]\left[v_{max}\right] \tag{2.4}$$

Both models of the elevator, process and measurement, essentially comprise only scalar values but, for the sake of consistency with Equations 2.1 and 2.2, are presented as vectors and matrices.

Space Elevator

Consider again the same physical system of the beam and the robot from the previous example. This time let's assume that, instead of the belt system, the robot uses a rocket engine to accelerate itself upwards. This description bears some resemblance to the concept of a space elevator (Edwards, 2000). It now becomes relevant that the robot is also subject to the Earth's gravitational pull G, which, without lack of generality, we will consider static here. Notice that now the control input is a force, which we will notate as F, and it is only a upwards force. This means that in order to descend the space elevator has to lower this force below G, or keep it equal with G to stay in a "hovering" situation.

Let M be the mass of the space elevator and a its vertical acceleration. Because of $F = Ma$, the height of the elevator is now the second derivative of the controllable parameter (i.e. the force F) and in this case we will also include the vertical velocity v_k of the space elevator in the state vector $\mathbf{s}_k$. This linearization is a common approach in order to avoid integration over the acceleration a. Let dt be the time difference between the two time indexes k and $k-1$.

We keep our ToF sensor from the previous example but, since we have a more complex process now, we decide to also include an accelerometer for the measurement model of the system. Since we can now directly measure vertical acceleration, we need to make acceleration a_k part of the state $\mathbf{s}_k$ in order to avoid integration over v_k in the measurement equation. We can see now that the state contains three state variables: $\mathbf{s}_k = [h_k, v_k, a_k]^\mathsf{T}$. Notice that now we use $\mathbf{D}$ to model the fact that our accelerometer is also picking up the acceleration due to gravity, which is actually not a component of the system's acceleration.

The process and the measurement models for the space elevator discussed above are presented in Equations 2.5 and 2.6, respectively.

$$\textbf{Process}: \begin{bmatrix} h_k \\ v_k \\ a_k \end{bmatrix} = \begin{bmatrix} 1 & dt & 0.5\,dt^2 \\ 0 & 1 & dt \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} h_{k-1} \\ v_{k-1} \\ a_{k-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ dt/M & -dt/M \end{bmatrix} \begin{bmatrix} F \\ G \end{bmatrix} \tag{2.5}$$

$$\textbf{Measurement}: \begin{bmatrix} \text{tof}_k \\ \text{acc}_k \end{bmatrix} = \begin{bmatrix} 2/C & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_k \\ v_k \\ a_k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} F \\ G \end{bmatrix} \tag{2.6}$$

We these examples in mind we can now head on to the formulation of the filters.

2.3 Linear Kalman Filter

The systems presented in the examples of Section 2.2.1 are linear and linearized, respectively. That is, both their process and measurement models can be represented by linear operations on the state and the control input. On these kinds of systems the simplest and original form of the Kalman filter, the linear Kalman filter (LKF), can be used. This is the case when when the dynamic physical process and physical laws of the measurements can be described by linear equations. A key insight of the LKF is that it incorporates already known information (the process model) about the properties of the physical system that state variables represent. This prior information on the state vector and control vector enable the Kalman filter to predict both the state and the inter-state-variable covariance while capturing the uncertainty (Kaparounakis κ. συν., 2020).

These examples, though, only tell half the story. After the formulation of the process and measurement models for the previous examples, one might be wondering why don't we just use these for the state estimation. And one would be correct, were there not for one small detail: we have not taken into account any kind of uncertainty at all in these models. If this were the case then one could obtain an analytic solution for the state from the measurements by simply symbolically solving for $\mathbf{s}_k$ in Equation 2.2. These models are ideal and, thus, not applicable without further specification. We need a version of the models that will also take uncertainty into account.

Let $\mathbf{n}_{p,k}$ be a noise vector of cardinality N (the same as the state vector $\mathbf{u}_k$) that represents the uncertainty of the process model at time k. This is called the *process noise* and corresponds arbitrary physical deviations from the theoretical model. Let $\mathbf{n}_{m,k}$ be a noise vector of cardinality Z (the same as the measurement vector $\mathbf{z}_k$) that represents the uncertainty of the measurement model at time k. This is referred to as the *measurement noise* and corresponds to the stochasticity of sensor measurements. Then, Equations 2.1 and 2.2 can be rewritten to also incorporate the uncertainty of the respective model. The new formulation of the process and measurement model appears in Equations 2.7 and 2.8.

$$\textbf{Process} : \mathbf{s}_k = \mathbf{F}\mathbf{s}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{n}_{p,k}. \tag{2.7}$$

$$\textbf{Measurement} : \mathbf{z}_k = \mathbf{H}\mathbf{s}_k + \mathbf{D}\mathbf{u}_k + \mathbf{n}_{m,k}. \tag{2.8}$$

In Kalman filter theory, the noise represented by $\mathbf{n}_{p,k}$ and $\mathbf{n}_{m,k}$ is assumed in to be zero-mean uncorrelated white noise. This can often not be the case in practice, but is a necessary assumption for the underlying math to be able to be worked out. Nevertheless, the filters fare quite well in reality.

2.3.1 State prediction

The algorithm of the LKF consists of two steps: the *state prediction* step and the *measurement update* step, *Predict* and *Update* for short. In the Predict step the state is propagated forward in time [1] using Equation 2.1, with the $\mathbf{F}$ matrix, to produce an estimate of the state, notated as $\hat{\mathbf{s}}_k^-$. As implied by the "–" superscript, this is the *a priori* estimate of the state, before taking into account measurements at time index k.

We cannot use Equation 2.7 here because the process noise term $\mathbf{n}_{p,k}$ is a stochastic variable; it is random and unknown. This is why the uncertainty of the estimation is tracked separately using a $N \times N$ covariance matrix, the *state estimate covariance matrix*. Let $\mathbf{P}$ be this matrix. $\mathbf{P}$ contains the covariance of each state variable with regards to each state variable. Notice that the main diagonal of $\mathbf{P}$ contains the variance of our estimation for each separate state variable. Alongside the propagation of the state, we will use the process model to also propagate the estimate covariance matrix forward in time. Because the covariance $\mathbf{P}$ is a $N \times N$ matrix, and because we need to do a linear transformation onto the same space ($N \times N \rightarrow N \times N$), we cannot simply multiply by $\mathbf{F}$, like in the case of the state vector. We need to multiply twice, with the propagation matrix and with its transpose, to obtain the covariance matrix of the a priori state estimate: $\mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^\mathsf{T}$.

We can now propagate the covariance forward in time but merely the fact that the process is taking place is introducing new uncertainty, due to the process noise. Let $\mathbf{Q}$ be the $N \times N$ process uncertainty matrix. This matrix encapsulates the covariance of the process noise, to be added to $\mathbf{P}$. The values of the covariances in $\mathbf{Q}$ are inherent to the process and considered here to be known, though that may not always be the case. There are ways of determining the process noise covariance matrix for a process. A simple approach is to only populate the main diagonal with the expected variance of the sub-process of each state variable. A more involved approach is to simulate the system and find out a performant $\mathbf{Q}$ through trial and error. Another approach is to infer it using auto-covariance methods on sample data (Odelson κ. συν., 2006).

[1] While almost all uses of these filters have time as the dimension with which the system is "evolving", there is no inherent theoretical reason that requires this.

We can now formulate the prediction step of the Kalman filter. It is presented in Equations 2.9 and 2.10. The hat symbol over $\hat{\mathbf{s}}_k$ reminds us that this is an estimate and not the true value.

$$\textbf{Predict}: \quad \hat{\mathbf{s}}_k^- = \mathbf{F}\hat{\mathbf{s}}_{k-1} + \mathbf{B}\mathbf{u}_k. \tag{2.9}$$

$$\mathbf{P}_k^- = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^\mathsf{T} + \mathbf{Q}. \tag{2.10}$$

2.3.2 Measurement update

We can now obtain an *a priori* (or prior) estimate for the state; a prediction of what the state should be based on the dynamics model of the system. The next step is the measurement update. With actual measurements, available e.g. by a system's sensors, the Update step of the LKF enhances the current estimate of the state using a weighted average method.

Let $\mathbf{H}$ be the $Z \times N$ measurement matrix and let $\mathbf{D}$ be the $Z \times U$ controllable parameter measurement matrix. To make a comparison between our *a priori* state estimate and the actual measurements we use the measurement matrix $\mathbf{H}$ to project the prior $\hat{\mathbf{s}}_k^-$ onto the measurement space. In case that control input affects the measurement vector, i.e. $\mathbf{D} \neq 0$, we need to bias the transformation accordingly. Let $\hat{\mathbf{z}}_k^-$ represent the prior estimate measurement vector that is the result of the above transformation. Equation 2.11 summarizes this transformation. It bears resemblance to the measurement model (Equation 2.8).

$$\hat{\mathbf{z}}_k^- = \mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k. \tag{2.11}$$

We can now compare $\hat{\mathbf{z}}_k^-$ against the actual measurement vector $\mathbf{z}_k$ to determine how well our prediction of the state $\hat{\mathbf{s}}_k^-$ has fared. We scale and project the difference of these vectors from the measurement space back to the state space by multiplying it with a gain matrix. Let $\mathbf{K}$ be this $N \times Z$ gain matrix, then Equation 2.12 summarizes this weighted average update on the state estimate.

$$\hat{\mathbf{s}}_k = \hat{\mathbf{s}}_k^- + \mathbf{K}(\mathbf{z}_k - \hat{\mathbf{z}}_k^-). \tag{2.12}$$

The calculation of $\mathbf{K}$ is a very important step. This is actually what R.E. Kalman famously derived and proved in his work (R. Kalman, 1960). We will get back to calculating the gain matrix $\mathbf{K}$ (Section 2.3.3) after also formulating the update step for the covariance matrix $\mathbf{P}$.

We need to update the estimate covariance matrix $\mathbf{P}$ with new information from the actual measurement. To achieve that, we project $\mathbf{P}$ from the state space to the measurement space by multiplying with $\mathbf{H}$ and then back to the state space by multiplying with the gain $\mathbf{K}$. The last operation also scales it, as discussed

for Equation 2.12. What we get from this operation is an adjustment matrix for the prior estimate covariance with this. Equation 2.14 summarizes the covariance update. Together with Equation 2.13, which is the same as Equation 2.12 with Equation 2.11 replaced, they comprise the equations for the Update step.

$$\textbf{Update}: \qquad \hat{\mathbf{s}}_k = \hat{\mathbf{s}}_k^- + \mathbf{K}_k(\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)). \tag{2.13}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^-. \tag{2.14}$$

2.3.3 Kalman gain

We will now return our attention to the missing piece of the equations above: the gain matrix $\mathbf{K}$ or Kalman gain. Remember that $\mathbf{K}$ is used to project the difference between the actual and the estimate measurement vectors ($\mathbf{z}_k$ and $\hat{\mathbf{z}}_k^-$, respectively), from the measurement space back onto the state space, while also accordingly scaling it to update the state estimate towards a better one. Since $\mathbf{z}_k$ and $\hat{\mathbf{z}}_k^-$ will almost certainly vary with time, it is reasonable to assume that $\mathbf{K}$ will, too. Hence, we denote it $\mathbf{K}_k$ from now on.

Let $\mathbf{R}$ be the $Z \times Z$ measurement uncertainty matrix, that encapsulates the covariance of the measurement noise. This matrix represents the uncertainty added to our estimate of the state $\hat{\mathbf{s}}_k$ during the Update step. In some sense, $\mathbf{R}$ is similar to the process uncertainty matrix $\mathbf{Q}$, but is usually easier to populate because one can consult the sensor's datasheet for the quoted variance of measurement or perform a set of measurements and derive it experimentally. Like $\mathbf{Q}$, it can be inferred using auto-covariance methods on sample data (Odelson κ. συν., 2006). The drawback of automatic inference for $\mathbf{R}$ is that it is usually less accurate compared to the case where the sensor noise distributions are provided.

Equation 2.15 corresponds to the calculation of the gain matrix K (R. Kalman, 1960).

$$\textbf{Gain}: \qquad \mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^\mathsf{T}(\mathbf{H}\mathbf{P}_k^-\mathbf{H}^\mathsf{T} + \mathbf{R})^{-1}. \tag{2.15}$$

Intuition: This equation packs a lot in, but we will try to unpack it with some intuition gained by rearranging the right-hand side of the equation to from a fraction, and presented in Equation 2.16. Keep in mind that this and the following derivations are not mathematically correct, because we are dealing with matrices, but are just to help intuitively understanding Equation 2.15.

$$\mathbf{K}_k = \frac{\mathbf{P}_k^-\mathbf{H}^\mathsf{T}}{\mathbf{H}\mathbf{P}_k^-\mathbf{H}^\mathsf{T} + \mathbf{R}}. \tag{2.16}$$

We can now see that $\mathbf{K}$ is a ratio that represents a comparison between the current covariance and the measurement noise matrix. The larger the covariances in the matrix $\mathbf{P}$ or the smaller the measurement uncertainty (entries in the matrix $\mathbf{R}$), the more the Kalman gain favors the measurements. In fact, if we imagine a limit as $\mathbf{P}$ "approaches" infinity or as $\mathbf{R}$ approaches 0, then $\mathbf{K}$ will be "approaching" $\mathbf{H}^{-1}$ (Kaparounakis κ. συν., 2020), or:

$$\lim_{\mathbf{P}_k^- \to \infty} \mathbf{K}_k = \lim_{\mathbf{R} \to 0} \mathbf{K}_k = \mathbf{H}^{-1}. \tag{2.17}$$

We can see how $\mathbf{K}_k$ favors the measurements, when this is the case, when we substitute the result from Limit 2.17 into the state Update equation (Equation 2.13).

$$\begin{aligned}
\hat{\mathbf{s}}_k &= \hat{\mathbf{s}}_k^- + \mathbf{K}_k(\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)) \\
&= \hat{\mathbf{s}}_k^- + \mathbf{H}^{-1}(\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)) \\
&= \hat{\mathbf{s}}_k^- + \mathbf{H}^{-1}\mathbf{z}_k - \cancel{\mathbf{H}^{-1}}\cancel{\mathbf{H}}\hat{\mathbf{s}}_k^- - \mathbf{H}^{-1}\mathbf{D}\mathbf{u}_k \\
&= \mathbf{H}^{-1}\mathbf{z}_k - \mathbf{H}^{-1}\mathbf{D}\mathbf{u}_k, \text{ and} \\
&= \mathbf{H}^{-1}\mathbf{z}_k, \text{ if } \mathbf{D} = 0.
\end{aligned}$$

We can see that in this case the updated state would be based solely on the measurement and the control bias, if any. Additionally, we notice that the updated covariance matrix $\mathbf{P}_k$ (Equation 2.14), will be zero, meaning that we would be entirely certain of our estimate, which would make sense when our measurement uncertainty is zero ($\mathbf{R} = 0$).

Similarly, in the limit when all the entries in $\mathbf{R}$ approach infinity, the new state will be based entirely on the prior state estimate $\hat{\mathbf{s}}_k^-$, from the physics model. The Kalman gain $\mathbf{K}_k$ will "approach" 0,

$$\lim_{\mathbf{R} \to \infty} \mathbf{K}_k = 0. \tag{2.18}$$

and, substituting into Equation 2.13,

$$\begin{aligned}
\hat{\mathbf{s}}_k &= \hat{\mathbf{s}}_k^- + \mathbf{K}_k(\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)) \\
&= \hat{\mathbf{s}}_k^- + 0 \cdot (\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)) \\
&= \hat{\mathbf{s}}_k^-.
\end{aligned}$$

At the same time, we can see that $\mathbf{P}_k$ would remain the same.

Of course, these are extreme scenarios and do not happen in practice. In fact, if they happen, it is probable that the filter malconfigured and is diverging.

2.3.4 LKF summary

Input of the Kalman filter: In a linear Kalman filter the state-space is usually represented as a set of matrices, with N, Z, and U being the dimensions of the state vector, measurement vector, and input vector, respectively. The inputs of the system in this case are:

1. $\mathbf{F}$: State transition $N \times N$ matrix.
2. $\mathbf{H}$: Measurement $Z \times N$ matrix.
3. $\mathbf{s}_0$: Initial state estimate $N \times 1$ vector.
4. $\mathbf{P}_0$: Initial state estimate covariance $N \times N$ matrix.
5. $\mathbf{B}$: Control transition $N \times U$ matrix.
6. $\mathbf{D}$: Control measurement $Z \times U$ matrix.
7. $\mathbf{Q}$: Process uncertainty $N \times N$ matrix.
8. $\mathbf{R}$: Measurement uncertainty $Z \times Z$ matrix.

Of these, $\mathbf{F}$ and $\mathbf{H}$ are crucial to describing the model of the system because the filter equations need a way to propagate the state (achieved via $\mathbf{F}$) and to correlate sensor readings with the predicted state (achieved via $\mathbf{H}$) (Kaparounakis κ. συν., 2020). On the other hand, $\mathbf{D}$ is not often used in practice. The process uncertainty $\mathbf{Q}$ and the measurement uncertainty $\mathbf{R}$ matrices can be derived experimentally. It is possible that $\mathbf{R}$ can be derived from sensor descriptions.

The equations of the linear Kalman filter are repeated, all together, in Equations 2.19 to 2.23.

$$\textbf{Predict}: \quad \hat{\mathbf{s}}_k^- = \mathbf{F}\hat{\mathbf{s}}_{k-1} + \mathbf{B}\mathbf{u}_k, \tag{2.19}$$

$$\mathbf{P}_k^- = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^\mathsf{T} + \mathbf{Q}. \tag{2.20}$$

$$\textbf{Gain}: \quad \mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^\mathsf{T}(\mathbf{H}\mathbf{P}_k^-\mathbf{H}^\mathsf{T} + \mathbf{R})^{-1}. \tag{2.21}$$

$$\textbf{Update}: \quad \hat{\mathbf{s}}_k = \hat{\mathbf{s}}_k^- + \mathbf{K}_k(\mathbf{z}_k - (\mathbf{H}\hat{\mathbf{s}}_k^- + \mathbf{D}\mathbf{u}_k)), \tag{2.22}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^-. \tag{2.23}$$

2.4 Extended Kalman Filter

The linear Kalman filter (LKF), presented in Section 2.3, is an optimal estimator for the linear estimation problem i.e. a proven solution for minimum mean-square error (MSSE) estimation. That said, the LKF is only applicable to

a small subset of the possible dynamic systems; those that can be described with linear equations.

Perhaps the most widely known variant of the Kalman filter is the extended Kalman filter (EKF), derived in an effort to extend the original linear equations for usage with non-linear systems (McElhoe, 1966; Smith, Schmidt, & McGee, 1962). In the EKF, the new state vector and the measurement vector result from the previous state by the application of functions instead of a matrix multiplication. We call these functions f and h, which correspond to matrices $\mathbf{F}$ and $\mathbf{H}$, respectively. These functions can be non-linear, such as transcendental functions (e.g., sin and cos), which are very often useful when describing physical systems. The incorporation of non-linear components in the process or the measurement model mandates the adaptation of the the *Predict* and *Update* equations. Functions f and h have to be differentiable, because their partial derivatives are needed by the filter, as discussed in Section 2.4.1.

The process and measurement model of the system are now function applications of f and h, presented in Equations 2.24 and 2.25.

$$\textbf{Process} : \mathbf{s}_k = \mathrm{f}\left(\mathbf{s}_{k-1}, \mathbf{u}_k\right) + \mathbf{n}_{p,k}. \tag{2.24}$$

$$\textbf{Measurement} : \mathbf{z}_k = \mathrm{h}\left(\mathbf{s}_k, \mathbf{u}_k\right) + \mathbf{n}_{m,k}. \tag{2.25}$$

Where, using similar terms and notations with Section 2.3, $\mathbf{n}_{p,k}$ and $\mathbf{n}_{m,k}$ are assumed to be zero-mean multivariate Gaussian noises with covariances $\mathbf{Q}$ and $\mathbf{R}$, respectively. In Equations 2.24 and 2.25 the noise is additive, i.e. it's simply added to the result of f or h.

2.4.1 Predict and Update equations

The filter's algorithm comprises the same 2 steps: Predict and Update. Reformulating the Kalman filter equations to use functions in the process and measurement model, we are confronted with the following problem: while we can use f and h in the equations that involve the state (Equations 2.9 and 2.13), we cannot do the same for the equations that involve the covariance and the Kalman gain (Equations 2.10, 2.14 and 2.15).

Given that there are N state variables, function f must be a function that takes in at least N arguments (the state, and perhaps the control input) and returns exactly N values, that constitute the new state. Similarly, h should take at least N arguments and return exactly Z values, the cardinality of the measurement vector. The above can be summarized with Expressions 2.26 and 2.27.

$$\mathrm{f} : \mathbb{R}^{N+\cdots} \to \mathbb{R}^{N}. \tag{2.26}$$

$$\mathrm{h} : \mathbb{R}^{N+\cdots} \to \mathbb{R}^{Z}. \tag{2.27}$$

In the LKF (Section 2.3), the propagation of the state estimate covariance matrix $\mathbf{P}$ forward in time, through the dynamics model of the system, is a linear transformation with $\mathbf{F}$ and its transpose (presented in 2.3.1). In the EKF, we have replaced this matrix with function f, so, in order to achieve the covariance matrix propagation we perform a local linearization of function f to a matrix using a first-order Taylor series expansion. This essentially is the Jacobian matrix of function f. Since this is a local linearization, i.e. a linearization specifically for the current state vector $\mathbf{s}_k$, the Jacobian matrix will not be static but will have to be recomputed at each time index k. The Jacobian matrix of f is denoted as $\mathbf{F}_{\mathrm{J},k}$, subscribed with a J for *Jacobian*, differentiating it from the state transition matrix of the LKF. Similarly, in the measurement update and the calculation of the Kalman gain, the Jacobian matrix of function h is used, denoted $\mathbf{H}_{\mathrm{J},k}$.

With i representing the numbers between (and including) 1 and N, let $\mathbf{s}_i$ represent the i-th state variable, let f_i represent the sub-function of f that has type $\mathrm{f}_i : \mathbb{R}^{N+\cdots} \to \mathbb{R}$ and computes the state propagation for the i-th state variable. Similarly, with i representing the numbers between (and including) 1 and Z, let h_i represent the sub-function of h that has type $\mathrm{h}_i : \mathbb{R}^{N+\cdots} \to \mathbb{R}$ and computes the measurement projection for the i-th measurement variable. Then, the Jacobian matrices of f and h are defined in Equations 2.28 and 2.29.

$$\mathbf{F}_{\mathrm{J},k} = \left.\frac{\partial \mathrm{f}_i}{\partial \mathbf{s}_i}\right|_{\hat{\mathbf{s}}_k,\mathbf{u}_k} = \begin{bmatrix} \frac{\partial \mathrm{f}_1}{\partial s_1} & \frac{\partial \mathrm{f}_1}{\partial s_2} & \cdots & \frac{\partial \mathrm{f}_1}{\partial s_N} \\ \frac{\partial \mathrm{f}_2}{\partial s_1} & \frac{\partial \mathrm{f}_2}{\partial s_2} & \cdots & \frac{\partial \mathrm{f}_2}{\partial s_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathrm{f}_N}{\partial s_1} & \frac{\partial \mathrm{f}_N}{\partial s_2} & \cdots & \frac{\partial \mathrm{f}_N}{\partial s_N} \end{bmatrix}. \tag{2.28}$$

$$\mathbf{H}_{\mathrm{J},k} = \left.\frac{\partial \mathrm{h}_i}{\partial \mathbf{s}_i}\right|_{\hat{\mathbf{s}}_k,\mathbf{u}_k} = \begin{bmatrix} \frac{\partial \mathrm{h}_1}{\partial s_1} & \frac{\partial \mathrm{h}_1}{\partial s_2} & \cdots & \frac{\partial \mathrm{h}_1}{\partial s_N} \\ \frac{\partial \mathrm{h}_2}{\partial s_1} & \frac{\partial \mathrm{h}_2}{\partial s_2} & \cdots & \frac{\partial \mathrm{h}_2}{\partial s_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathrm{h}_Z}{\partial s_1} & \frac{\partial \mathrm{h}_Z}{\partial s_2} & \cdots & \frac{\partial \mathrm{h}_Z}{\partial s_N} \end{bmatrix}. \tag{2.29}$$

We can now re-formulate the Kalman filter equations for the non-linear case using functions f and h and their Jacobians $\mathbf{F}_{\mathrm{J},k}$ and $\mathbf{H}_{\mathrm{J},k}$, respectively. The updated equations are summarized in Equations 2.30 to 2.34.

$$\textbf{EKF Predict}: \quad \hat{\mathbf{s}}_k^- = \mathrm{f}(\hat{\mathbf{s}}_{k-1}, \mathbf{u}_k). \tag{2.30}$$

$$\mathbf{P}_k^- = \mathbf{F}_{\mathrm{J},k}\mathbf{P}_{k-1}\mathbf{F}_{\mathrm{J},k}^\mathsf{T} + \mathbf{Q}. \tag{2.31}$$

$$\textbf{EKF Gain}: \quad \mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}_{\mathrm{J},k}^\mathsf{T}(\mathbf{H}_{\mathrm{J},k}\mathbf{P}_k^-\mathbf{H}_{\mathrm{J},k}^\mathsf{T} + \mathbf{R})^{-1}. \tag{2.32}$$

$$\textbf{EKF Update}: \quad \hat{\mathbf{s}}_k = \hat{\mathbf{s}}_k^- + \mathbf{K}_k(\mathbf{z}_k - \mathrm{h}(\hat{\mathbf{s}}_k^-, \mathbf{u}_k)). \tag{2.33}$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H}_{\mathrm{J},k})\mathbf{P}_k^-. \tag{2.34}$$

2.4.2 EKF limitations

The EKF has been the *de facto* standard for state estimation in non-linear systems due to its theoretical simplicity and ease of implementation compared to other approaches (Wan & Van Der Merwe, 2000). Despite its popularity, it suffers from fundamental drawbacks (Julier & Uhlmann, 2004). The EKF uses a first-order Taylor series expansion to locally linearize the non-linear model. Firstly, this approximation imposes the requirement that the error propagation should be able to be approximated with a linear function, thus requiring that the non-linearities of the system be mild (Haykin & Arasaratnam, 2009). Secondly, its employment requires the derivation of the Jacobian matrices of the non-linear dynamic models of the system. This matrix of partial derivatives may not exist at all for some physical systems (e.g. for highly quantized sensor measurements). If it exists, it can be highly impractical to compute (Kaparounakis κ. συν., 2020). Traditionally, it is derived manually symbolically; a process which is error-prone.

In the context of this work, we overcome the impracticality of manually deriving accurate partial derivatives for the Jacobian matrix by using Automatic Differentiation (Baydin κ. συν., 2017; Margossian, 2019) as we discuss in Chapter 3.

2.4.3 Examples

We will now present two working examples for the non-linear case of the Kalman filter. A simple pendulum and a differential wheel drive robot, the TurtleBot3 (Amsters & Slaets, 2019). We revisit these systems in Chapter 4, Section 4.4, presenting their Newton descriptions, and in Chapter 6, where we use them in the evaluation of our implemented system.

Pendulum

The pendulum is a great example non-linear system; it is a simple and familiar concept. The pendulum is a system that is oscillating due to the effect of gravity on the pendulum's bob, a small dense object at the far end of the rod. On a

damped pendulum, the volume of the rod and the bob might produce air drag during movement, slowing the pendulum down. On an ideal pendulum this drag is considered zero.

The physical dynamics of a pendulum system are often described using the oscillation differential equation. Let b be the damping factor with units of $kg\,s^{-1}$ in the S.I. system (Mohazzabi, Shankar, κ. συν., 2017), and let m be the mass of the pendulum's bob. The equation of the dynamics including drag are presented in Equation 2.35. The equation for the ideal pendulum is the same, but with $b = 0$.

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{b}{m}\frac{\mathrm{d}\theta}{\mathrm{d}t} + \frac{g}{l}\sin(\theta) = 0. \tag{2.35}$$

We consider the state of the pendulum to be the current angular displacement θ_k and angular speed ω_k, thus the state vector is $< \theta_k, \omega_k >$. While we might only be interested in knowing the angular displacement of the pendulum, it is common practice to also track derivatives or other variables in the estimation of the state.

Equation 2.35 is non-linear because the angular displacement θ is argument to function sin, therefore we will be using the extended Kalman filter. In order to formulate the process model of the pendulum we convert Equation 2.35 to two state predict equations: one for the angular displacement and one for the angular speed. We assume that some small amount of time, denoted Δt, has elapsed between two successive cycles of the filter. Using Δt we linearize the prediction equation for θ; the change to the angular displacement is the same as before but with a small change of Δt times the angular speed. The same procedure is done for the prediction of the new value for ω; it is the previous value plus the changes during time Δt as described by Equation 2.35. The process model of the pendulum is summarized in Equations 2.36 and 2.37.

$$\theta_k = \theta_{k-1} + \omega_{k-1}\Delta t. \tag{2.36}$$

$$\omega_k = \omega_{k-1} + \frac{B}{M}\omega_{k-1}\Delta t - \frac{G}{L}\sin(\theta_{k-1})\Delta t. \tag{2.37}$$

A simplistic choice for a measurement model is to place a gyroscope sensor on the pendulum's bob, such that the gyro's Z-axis always faces the direction of oscillation. If the gyro measurement is given in Gs we can scale it using the measurement matrix $\mathbf{H}$. Equation 2.38 relates a given ω_k with its measurement, summarizing the measurement model for the pendulum with $\mathbf{H} = 1$.

$$z_{gyro,k} = \omega_k. \tag{2.38}$$

The given process model is non-linear because of Equation 2.37, thus requiring the Predict step from the EKF, as opposed to the measurement model (Equation 2.38) for which the Update step is the same with the LKF. We then need to derive the Jacobian matrix $\mathbf{F}_{\mathrm{J},k}$ of f, function f being Equations 2.36 and 2.37. Let f_1 and f_2 denote those two equations, respectively. Then Equation 2.39 presents

the Jacobian matrix needed for the formulation the EKF and the equations of the filter (Equations 2.30 to 2.34) can be applied in a straightforward manner.

$$\mathbf{F}_{\mathrm{J},k} = \begin{bmatrix} \frac{\partial \mathrm{f}_1}{\partial \theta} & \frac{\partial \mathrm{f}_1}{\partial \omega} \\ \frac{\partial \mathrm{f}_2}{\partial \theta} & \frac{\partial \mathrm{f}_2}{\partial \omega} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ -\frac{G\cos(\theta)\Delta t}{L} & 1 + \frac{B\Delta t}{M} \end{bmatrix}. \tag{2.39}$$

The Jacobian matrix in this example is easy to manually derive. More complex systems would lead to a more intricate and prone-to-errors derivation process. In the context of automating the code generation of the Kalman filter algorithms, which is the purpose of this work, we also provide Automatic Differentiation of non-linear models, presented in Chapters 3 and 5.

The description of the pendulum system in the input language, Newton, is presented in Chapter 4.

TurtleBot3 Robot

Now we look at the formulation of the process and measurement models for a more complex case; a differential wheel drive robot: The TurtleBot3 Burger (Amsters & Slaets, 2019), a robot built on open-source software operating system (ROS) (Quigley κ. συν., 2009). The TurtleBot3 Burger is a differential drive robot, i.e., it controls its linear and angular velocities by actuating on a set of wheels with individual speed setpoints v_r and v_l, for the right and the left wheel, respectively (Kaparounakis κ. συν., 2020).

The general kinematic model of such a robot exhibits a non-holonomic constraint and thus the process model, described by Dudek et al. (Dudek & Jenkin, 2010), is split in two special cases: when the robot is moving and when it's turning. The first case describes the robots translational movement when the two wheels have the same angular velocity and is presented in Equation 2.40. The second case manages the rotation of the robot, when the two wheels have opposite angular velocities, and is presented in Equation 2.41.

If $v_r = v_l = v$:

$$\begin{bmatrix} x_{t+\Delta t} \\ y_{t+\Delta t} \\ \theta_{t+\Delta t} \end{bmatrix} = \begin{bmatrix} x_t + v\cos(\theta_t)\Delta t \\ y_t + v\sin(\theta_t)\Delta t \\ \theta_t \end{bmatrix} \tag{2.40}$$

and if $v_r = -v_l = v$:

$$\begin{bmatrix} x_{t+\Delta t} \\ y_{t+\Delta t} \\ \theta_{t+\Delta t} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ \theta_t + 2v\Delta t/l \end{bmatrix} \tag{2.41}$$

where x, y and θ are the robot's position and yaw in the fixed frame, v is the velocity in the fixed frame, Δt is the time difference between subsequent runs of the model and l is the distance between the two wheels. The subscript denotes a given point in time t.

We can see that the dynamic process of the robot is split in two cases. This creates a problem because the process model for the Kalman filter does not take such scenario into account. We can counter this problem by either manually adding a control flow command in the generated code or generate the two processes as separate Predict functions and call them accordingly from user code.

A third method is presented here, in which we use a mathematical trick based on the fact that the two angular velocities will be either equal or opposite.This model uses the angular velocities directly, as opposed to just v in Equations 2.40 and 2.41, denoted v_r for the right wheel and v_l for the left wheel. We also track the translational velocities of the robot, as well as the angular velocity on the Z-axis, by adding them to the state vector as v_x, v_y and ω. We do this for the same reason we added the angular velocity of the pendulum to the state vector in Section 2.4.3. The result is presented in Equation 2.42.

$$\begin{bmatrix} x_k \\ y_k \\ v_{x,k} \\ v_{y,k} \\ \theta_k \\ \omega_k \end{bmatrix} = \begin{bmatrix} x_{k-1} + v_{x,k-1}\Delta t \\ y_{k-1} + v_{y,k-1}\Delta t \\ v_{x,k-1}\frac{(v_r+v_l)}{2}\sin(\theta_{k-1}) \\ v_{y,k-1}\frac{(v_r+v_l)}{2}\cos(\theta_{k-1}) \\ \theta_{k-1} + \omega_{k-1}\Delta t \\ \frac{(v_r-v_l)}{L} \end{bmatrix} \tag{2.42}$$

This is a possible unified process model for the TurtleBot3 robot, though, the two options proposed before this third one are likely to fare better experimentally.

It is obvious that the process model would be non-linear in any case, thus making it a candidate for the EKF. Of course, we would need the Jacobian matrix of the model which is much more involved for this system as opposed to the pendulum one. The derivation is handled automatically using Automatic Differentiation presented in Chapter 3 and a manual derivation will not be presented here.

We have discussed simple possible process models for the TurtleBot3 Robot. We know specify a hypothetical measurement model for the robot. For this model we assume the placement of beacons that wirelessly inform the robot of its position in space. A separate gyroscope captures the angular velocity of the robot on its Z-axis. This measurement model is simple and linear; it is presented in Equation 2.43.

$$\begin{bmatrix} z_{x,k} \\ z_{y,k} \\ z_{\omega,k} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \omega_k \end{bmatrix} \tag{2.43}$$

Chapter 3

Automatic Differentiation

Derivatives of functions frequently have an important role in numerical computations where some kind of optimization or extrapolation is taking place. A prominent example of this is in the field of Machine Learning, wherein the abstract goal is the optimization of an objective function, that may have thousands of inputs, using numerical methods such as the Newton method or gradient descent (Widrow & Lehr, 1990; Sra, Nowozin, & Wright, 2012). Another example is computational finance, where derivatives are needed to solve the necessary financial differential equations (Homescu, 2011). Additionally, in the general field of scientific computing, the calculation of derivatives is vital for making useful observations and conclusions in biology, medicine, engineering and many others.

Of course, the calculation of derivatives is also highly relevant in the context of state-estimation in non-linearly modeled cyber-physical embedded computing systems. Such systems may be large scale, such as for making meteorological predictions (Talagrand & Courtier, 1987), or more local, as in the context of UAV state-estimation (Mueller κ. συν., 2015).

In general, there are 4 approaches to implementing a computing system where derivatives will be needed *during* the computation: (a) manual derivation (i.e. by a human) of the subject function, (b) the standard limit approach, (c) symbolic differentiation and (d) automatic differentiation (Griewank κ. συν., 1989; Baydin κ. συν., 2017; Margossian, 2019).

The **manual derivation** of the subject function is a hand-coded technique where the designer analytically derives the desired derivative and hard-codes it in the system. This is an exact and fast solution if the derivation is not hard. That said, it is not automated, and thus, is not general solution that can be time-consuming and error-prone, if applicable at all.

The **standard limit** approach (from hereon referred to as Standard Differentiation, Standard Diff or S.D. in this work), also known as finite differentiation, is the calculation of a derivative using the classical limit definition. This is relatively easy to implement but can suffer in terms of precision, because of floating-point errors, and time, because it requires many evaluations of the function.

Symbolic differentiation methods try to automatically find an analytical solution to the derivation by performing various symbolic transformations to the equation, much like a manual derivation. This an exact and one-off solution for each function that nevertheless is memory-intensive, slow and limited as to the complexity of the possible functions.

Finally, **automatic differentiation** (from hereon also referred to as AutoDiff or A.D.), also known as algorithmic differentiation, (Griewank κ. συν., 1989; Griewank, 1992; Griewank & Walther, 2008; Baydin κ. συν., 2017; Margossian, 2019) is neither a numerical approximation of the derivative nor a symbolic method for deriving it. It can be considered as a non-standard interpretation of a computer program where the output is an augmented computer-program that also contains the exact calculation for various derivatives.

Automatic Differentiation leverages the chain rule of derivatives to supplement a sequence of basic operations, that have known derivatives, with the calculation of the values of derivatives with respect to some inputs. This means that, along with the computation of an expression, we can compute the intermediate partial derivatives whose calculation leads down to the resulting partial derivatives that we care for. It becomes apparent, therefore, that to be able to perform this technique on an arbitrary expression of a grammar, we would first need to reformulate the expression into a sequence of basic operations. This form is often called static single assignment form (SSA) (Alpern κ. συν., 1988; Braun κ. συν., 2013), an equivalent form of the functions computation which statically "unrolls" the function's intermediate operations so that only one operation is used at each step.

Let x and y represent arguments of a numerical expression function and z the resulting return value. Notationally we may write this as $\mathrm{z}(x, y)$. An example expression (Kaparounakis κ. συν., 2020) is presented in Equation 3.1. We split the expression for z to its single static assignment sub-expressions. Let r_i be the value of the i-th intermediate assignment of the SSA from, excluding the assignment of the "return" value, which remains as z. The SSA form of the original expression is presented in Equations 3.2 to 3.5, using Leibniz's notation for differentiation.

$$z = x \cdot y^2 + \sin(x). \tag{3.1}$$

$$\text{❶ } r_1 = y^2; \tag{3.2}$$

$$\text{❷ } r_2 = x \cdot r_1; \tag{3.3}$$

$$\text{❸ } r_3 = \sin(x); \tag{3.4}$$

$$\text{❹ } z = r_2 + r_3. \tag{3.5}$$

Automatic Differentiation has two modes of operation: *forward mode* and *reverse mode*. The difference between them is related to the order in which they use the intermediate values of the SSA form of the original expression to calculate derivatives, hence their name. In forward mode, at every step of the computation

of the SSA form, the partial derivative of the current assignment with respect to *one* input variable is also calculated. In reverse mode, the complete evaluation of the function takes place first, while keeping intermediate values available. Then, starting from the last assignment and going upwards, the partial derivative of the output (z in Equation 3.1) with respect to each assignment's variable is calculated. The two modes are presented in more detail in the following sections.

3.1 Forward Mode

In forward mode the intermediate derivatives of the expression with respect to *one* of the input variables can be calculated alongside the original function calculation. This is achieved by utilizing the chain rule of derivatives, a widely known formula that is used to compute derivatives of composite functions.

The chain rule of derivatives states that if a variable z depends on a variable r and variable r depends on a variable t, then

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \frac{\mathrm{d}z}{\mathrm{d}r}\frac{\mathrm{d}r}{\mathrm{d}t}. \tag{3.6}$$

In the case that z depends on multiple variables, e.g. $z = \mathrm{f}(r_1, r_2, \ldots, r_k)$ and r_1, r_2, …, r_k all depend on t, then the chain rule can be written using the partial derivatives, presented in Equation 3.7 for the example function f.

$$\frac{\partial z}{\partial t} \equiv \frac{\partial \mathrm{f}}{\partial t} = \frac{\partial z}{\partial r_1}\frac{\partial r_1}{\partial t} + \frac{\partial z}{\partial r_2}\frac{\partial r_2}{\partial t} + \cdots + \frac{\partial z}{\partial r_k}\frac{\partial r_k}{\partial t} = \sum_{i\in[1,k]} \frac{\partial z}{\partial r_i}\frac{\partial r_i}{\partial t}. \tag{3.7}$$

We can see that it is possible to obtain $\frac{\partial z}{\partial t}$ for any t that is either one of the function inputs or one of the intermediate assignments r_i using the chain rule of derivatives, as presented. To achieve this, we obtain the partial derivative of each assignment of the SSA form (Equations 3.2 to 3.5) with respect to an undefined, for now, input variable t. The benefit of using the SSA form is that we can predefine the derivatives of the basic operations to use them as building blocks for more complex expressions.

Before we start we will need to state the basic operations and their derivatives. These usually are generally known derivatives of fundamental operation. In the context of A.D. in programming, these cases are the building blocks and would need to be predefined in the source code. Here are some sample predefined rules:

- Constant Scaling: $\frac{\partial(at)}{\partial t} = a$,
- Constant Addition: $\frac{\partial(a+t)}{\partial t} = 1$,
- Exponentiation: $\frac{\partial(t^n)}{\partial t} = n \cdot t^{n-1}$,
- Product: $\frac{\partial(\mathrm{f}_1(t)\mathrm{f}_2(t))}{\partial t} = \mathrm{f}_1(t)\frac{\partial(\mathrm{f}_2(t))}{\partial t} + \mathrm{f}_2(t)\frac{\partial(\mathrm{f}_1(t))}{\partial t}$,

- Sum: $\frac{\partial(\mathrm{f}_1(t)+\mathrm{f}_2(t))}{\partial t} = \frac{\partial(\mathrm{f}_1(t))}{\partial t} + \frac{\partial(\mathrm{f}_2(t))}{\partial t}$,
- Sine: $\frac{\partial(\sin(t))}{\partial t} = \cos(t)$.

We now go step by step through the derivation of the Forward Mode Automatic Differentiation of Equation 3.1.

① Equation 3.2 w.r.t variable t is $\frac{\partial r_1}{\partial t}$. Assignment r_1 depends only on variable y. Using the chain rule from Equation 3.6 we write $\frac{\partial r_1}{\partial t} = \frac{\partial r_1}{\partial y}\frac{\partial y}{\partial t}$. Factor $\frac{\partial r_1}{\partial y}$ is exponentiation, one of the basic operations for which we know the derivative is $\frac{\partial r_1}{\partial y} = 2y$. Thus, the partial derivative is presented in Equation 3.8.

② Equation 3.3 depends on x and r_1, so, using the chain rule from Equation 3.7, it is written $\frac{\partial r_2}{\partial t} = \frac{\partial r_2}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial r_2}{\partial r_1}\frac{\partial r_1}{\partial t}$. It is apparent here that the chain rule is a generalization of the product derivative rule. The partial derivatives $\frac{\partial r_2}{\partial x}$ and $\frac{\partial r_2}{\partial r_1}$ are trivial operations, which we also know the derivative of. They are $\frac{\partial r_2}{\partial x} = r_1$ and $\frac{\partial r_2}{\partial r_1} = x$. Notice how $\frac{\partial r_1}{\partial t}$ will become available from step ①. The result is presented in Equation 3.9.

③ Equation 3.4 depends only on x. It is written $\frac{\partial r_3}{\partial t} = \frac{\partial r_3}{\partial x}\frac{\partial x}{\partial t} = \cos(x)\frac{\partial x}{\partial t}$. The result is presented in Equation 3.10.

④ Equation 3.5, the final value of the original expression, depends on r_2 and r_3 and is thus written $\frac{\partial z}{\partial t} = \frac{\partial z}{\partial r_2}\frac{\partial r_2}{\partial t} + \frac{\partial z}{\partial r_3}\frac{\partial r_3}{\partial t} = 1 \cdot \frac{\partial r_2}{\partial t} + 1 \cdot \frac{\partial r_3}{\partial t}$. Notice how both $\frac{\partial r_2}{\partial t}$ and $\frac{\partial r_3}{\partial t}$ will become available in steps ② and ③, respectively.

All the resulting partial derivatives of each intermediate value w.r.t. variable t appear in Equations 3.8 to 3.11.

$$r_1 = y^2; \qquad \frac{\partial r_1}{\partial t} = 2y \cdot \frac{\partial y}{\partial t}; \tag{3.8}$$

$$r_2 = x \cdot r_1; \qquad \frac{\partial r_2}{\partial t} = x \cdot \frac{\partial r_1}{\partial t} + r_1 \cdot \frac{\partial x}{\partial t}; \tag{3.9}$$

$$r_3 = \sin(x); \qquad \frac{\partial r_3}{\partial t} = \cos(x) \cdot \frac{\partial x}{\partial t}; \tag{3.10}$$

$$z = r_2 + r_3. \qquad \frac{\partial z}{\partial t} = \frac{\partial r_2}{\partial t} + \frac{\partial r_3}{\partial t}. \tag{3.11}$$

Now that we have this derivation we can calculate $\frac{\partial z}{\partial x}$ by setting $t = x$, assuming that y and x are independent, that is $\frac{\partial y}{\partial x} = 0$. Similarly, we can calculate $\frac{\partial z}{\partial y}$ by setting $t = y$. The 2 different resulting computational paths are presented in Equations 3.13 to 3.15 for $\frac{\partial z}{\partial x}$ and Equations 3.17 to 3.19 for $\frac{\partial z}{\partial y}$.

$$\frac{\partial r_1}{\partial x} = 2y \cdot \frac{\cancelto{0}{\partial y}}{\cancel{\partial x}} = 0; \tag{3.12}$$

$$\frac{\partial r_2}{\partial x} = x \cdot \frac{\cancelto{0}{\partial r_1}}{\cancel{\partial x}} + r_1 \cdot \frac{\cancelto{1}{\partial x}}{\cancel{\partial x}} = r_1 = y^2; \tag{3.13}$$

$$\frac{\partial r_3}{\partial x} = \cos(x) \cdot \frac{\cancelto{1}{\partial x}}{\cancel{\partial x}} = \cos(x); \tag{3.14}$$

$$\frac{\partial z}{\partial x} = \frac{\partial r_2}{\partial x} + \frac{\partial r_3}{\partial x} = y^2 + \cos(x). \tag{3.15}$$

$$\frac{\partial r_1}{\partial y} = 2y \cdot \frac{\cancelto{1}{\partial y}}{\cancel{\partial y}} = 2y; \tag{3.16}$$

$$\frac{\partial r_2}{\partial y} = x \cdot \frac{\partial r_1}{\partial y} + r_1 \cdot \frac{\cancelto{0}{\partial x}}{\cancel{\partial y}} = x \cdot 2y; \tag{3.17}$$

$$\frac{\partial r_3}{\partial y} = \cos(x) \cdot \frac{\cancelto{0}{\partial x}}{\cancel{\partial y}} = 0; \tag{3.18}$$

$$\frac{\partial z}{\partial y} = \frac{\partial r_2}{\partial y} + \frac{\partial r_3}{\partial y} = 2xy. \tag{3.19}$$

At this point a question of how can we "set" $t = x$ or $t = y$ in a computer program might naturally arise. The simple solution to this is to track each partial derivative as its own variable (as opposed to a composition of 2 variables). This way we can start the computation of $\frac{\partial z}{\partial x} \equiv$ `dzdx` by setting $\frac{\partial x}{\partial t} \equiv$ `dxdt` $= 1$ and $\frac{\partial y}{\partial t} \equiv$ `dydt` $= 0$ and the computation of $\frac{\partial z}{\partial y} \equiv$ `dzdy` by setting `dxdt = 0` and `dydt = 1`.

In the example expression above (Equation 3.1), we observe that in order to calculate the expression's partial derivatives with respect to the two input variables x and y we need to do two function evaluations. One for $\frac{\partial z}{\partial x}$ and one for $\frac{\partial z}{\partial y}$. In a general scenario of Forward Mode A.D. of an expression with N inputs and one output, N partial derivatives can be calculated with N separate function evaluations.

If it were the case that $\mathrm{f} : \mathbb{R}^N \to \mathbb{R}^Z$, meaning that function f, apart from multiple inputs, also has multiple outputs, it would become apparent that we can calculate the partial derivatives of all Z output variables with respect to a single input variable (out of the N) in one expression evaluation. In the context of calculating the Jacobian matrix of function f, using Forward Mode A.D. we can calculate each column with a separate evaluation of f.

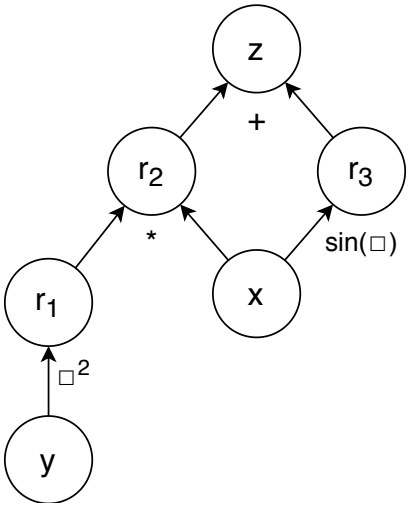


Figure 3.1: The Abstract Syntax Tree of the original expression. The inputs, output and intermediate variables of the expression appear inside the nodes. The arrows (edges) show the flow of evaluation of the original expression (Equation 3.1). Near the destination node of the edges, the related operation is marked. Binary operations (i.e. addition and multiplication) have two incoming edges while unary operations (i.e. exponentiation and the sine function) have one incoming arrow.

3.2 Reverse Mode

In this mode, the partial derivative of the expression with respect to all input variables can be calculated after the original function calculation. Like in the forward mode, this is achieved utilizing the chain rule of derivatives.

The reason this is called Reverse Mode is because the calculation of partial derivatives starts from the end of the evaluation, i.e. the last assignment of the SSA form, and proceeds downwards on the expression tree, in a cascading manner. In this process, the partial derivatives of the output with respect to each intermediate result variable are calculated; down to those that are with respect to the input variables. Due to this order of evaluation, the reverse mode is more complicated than the forward mode.

Using the same basic operations as with the Forward Mode A.D. example (Section 3.1), we go step by step through the attainment process for the Reverse Mode Automatic Differentiation of Equation 3.1. Its SSA form remains the same (Equations 3.2 to 3.5).

To help with the increased intricacy of the reverse mode derivation of the partial derivatives of z w.r.t. x and y, and to help highlight the difference in the manner of operation between the Forward and the Reverse modes, we will utilize an expression tree. The Abstract Syntax Tree of the original expression (Equation 3.1) is presented in Figure 3.1.

On this tree, starting from the end of evaluation, i.e. the the node marked

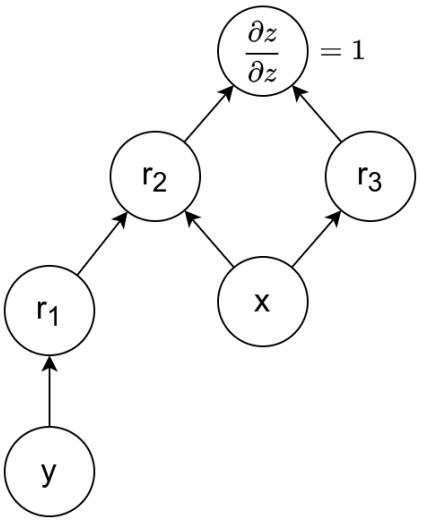


Figure 3.2: Step ① of Reverse Mode A.D. on the expression in Equation 3.1.

with z, we will proceed downwards, deriving the partial derivatives of z w.r.t. each node's marked variable. For each variable we will find the assignments of the SSA form in which it appears in the right-hand side. We will use these assignments to express the chain rule for the partial derivative of z w.r.t. the current variable.

① We start from z. This is a trivial case as z is the output. It does not appear in any other assignment and obviously $\frac{\partial z}{\partial z} = 1$. This value is usually hardcoded to kickstart the process.

② Next, we tackle r_2. It only appears in assignment ❹ of the SSA form (Equation 3.5). Using this assignment we can trivially derive that $\frac{\partial z}{\partial r_2} = 1$.

③ The same goes for r_3, with $\frac{\partial z}{\partial r_3} = 1$. The updated tree appears in Figure 3.3.

④ The next variable node is x. This is a bit more tricky, because it appears in two assignments, but it's here that the power of Automatic Differentiation shows. The assignments in which it appears are ❸ and ❷ (Equations 3.4 and 3.3, respectively). Using the chain rule, we can state that $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial r_2}\frac{\partial r_2}{\partial x} + \frac{\partial z}{\partial r_3}\frac{\partial r_3}{\partial x}$. This intuitively means that the change brought to variable z by the change of variable x is the sum of the changes brought to z through variable r_2 and through variable r_3.

Here we notice that each one of these partial derivatives either is known or can be trivially derived. From steps ② and ③ we know that $\frac{\partial z}{\partial r_2} = 1$ and $\frac{\partial z}{\partial r_3} = 1$. From Equations 3.4 and 3.3 we can find that $\frac{\partial r_3}{\partial x} = \cos(x)$ and $\frac{\partial r_2}{\partial x} = r_1 = y^2$. The updated tree appears in Figure 3.4.

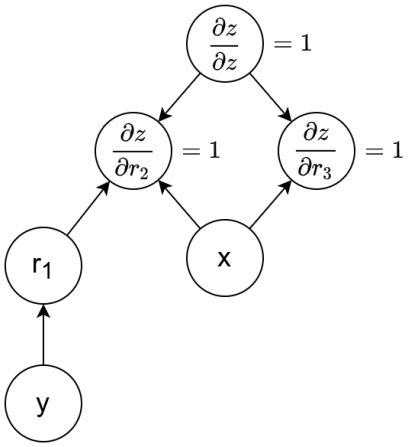


Figure 3.3: Steps ② and ③ of Reverse Mode A.D. on the expression in Equation 3.1.

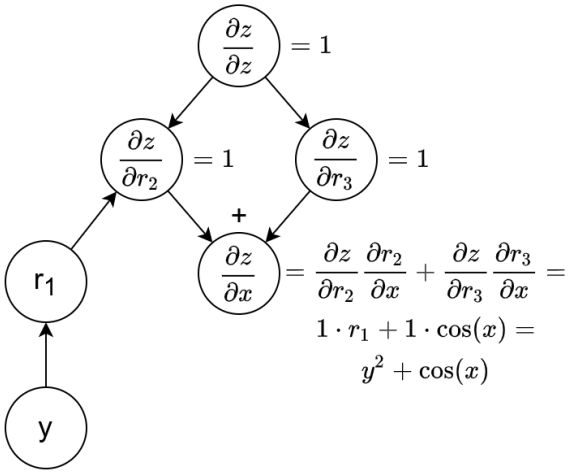


Figure 3.4: Step ④ of Reverse Mode A.D. on the expression in Equation 3.1.

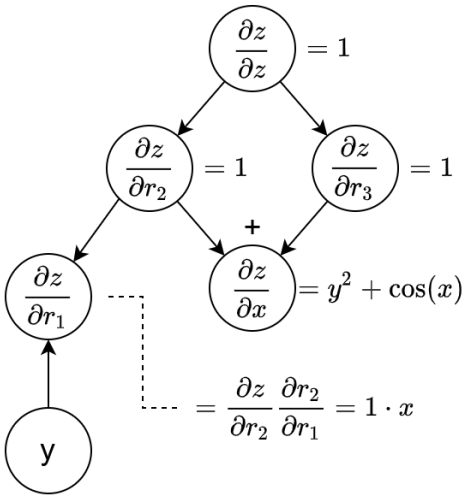


Figure 3.5: Step ⑤ of Reverse Mode A.D. on the expression in Equation 3.1.

⑤ The next variable is r_1, which only appears in the assignment for r_2 (Equation 3.3). Thus, the required partial derivative can be written as $\frac{\partial z}{\partial r_1} = \frac{\partial z}{\partial r_2}\frac{\partial r_2}{\partial r_1}$ where we can again easily find that $\frac{\partial z}{\partial r_2} = 1$ and $\frac{\partial r_2}{\partial r_1} = x$. The updated tree appears in Figure 3.5.

⑥ Finally, y only appears in the assignment for r_1 (Equation 3.2). We write $\frac{\partial z}{\partial y} = \frac{\partial z}{\partial r_1}\frac{\partial r_1}{\partial y}$, and we find that $\frac{\partial z}{\partial r_1} = x$ and $\frac{\partial r_1}{\partial y} = 2y$. The final tree appears in Figure 3.6.

In the leaf nodes of the final tree, in Figure 3.6, the expressions for the partial derivatives of z with respect to each of the input variables, namely x and y, are available. We, therefore, observe that we can calculate the partial derivatives of a single output w.r.t *all* the input variables in a single pass of the algorithm. In a general scenario of Reverse Mode A.D. of an expression with N inputs and one output, N partial derivatives can be calculated with a single function evaluation.

To compare with Forward Mode A.D. (Section 3.1), if, again, it were the case that $\mathrm{f} : \mathbb{R}^N \to \mathbb{R}^Z$ (thus, f being a vector function) we can calculate the partial derivatives of a single output (out of the Z) with respect to all N input variables in one expression evaluation. In the context of calculating the Jacobian matrix of f, using Reverse Mode A.D., we can calculate each row with a separate evaluation of f.

3.3 Overhead Analysis

It is important to highlight the practical implications, in terms of runtime resource requirements, of using either mode Automatic Differentiation over the other, or over some other differentiation method.

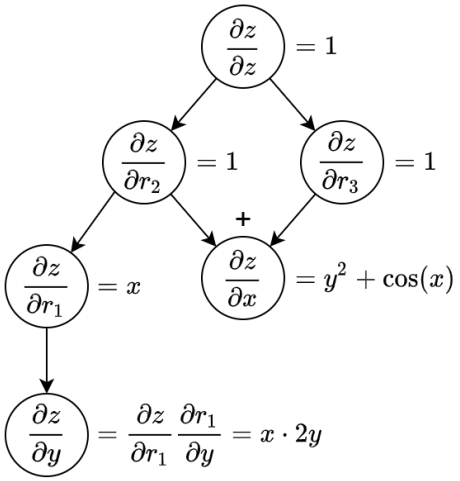


Figure 3.6: Step ⑥ of Reverse Mode A.D. on the expression in Equation 3.1.

Regarding the forward mode of A.D., the additional arithmetic operations needed to calculate the derivatives are guaranteed to scale linearly with the original expression, even with a naïve implementation. Essentially, this is because each operation in the original expression's SSA is hijacked to also compute the derivative for this operation. On a more complex implementation, that would take into account to only compute the derivatives for the various same component functions only once, this overhead can be brought closer to the cost of evaluating the original function.

The issue of the overhead of an implementation of the reverse mode of A.D. is more complicated. The prime reason for this being the fact that the derivatives are calculated after the evaluation of the original function. This complicates things because computational artifacts from the evaluation will be needed at later parts, thus increasing the memory footprint of the process. The increase in computational complexity is in principle still linear but, in a naïve implementation of reverse mode A.D. with source transformation it easy for the result code to grow faster-than-linearly as the original expression becomes more complex.

A more detailed analysis of the computational complexity and memory characteristics of the different possible implementations of A.D. can be found in the work by Griewank A. and Walther A. (Griewank, 1992; Griewank & Walther, 2008).

Of particular interest in our case is specifically the cost of computing the partial derivatives that complete the Jacobian matrix for a given function. This is due to needing the derivatives of the model functions for the estimate covariance propagation equations in the extended Kalman filter as discussed in Section 2.4.2. We make our choice regarding the differentiation method and modes to implement, later, in Chapter 5.

Chapter 4

The Newton Language

The Newton language (Lim & Stanley-Marbell, 2018) is a specification language for describing physics. The original implementation of Newton (Lim κ. συν., 2017) had the aim of providing a framework to specify invariants between sensor data for both compile-time and run-time purposes. In other words, to act as a front-end in computations involving the physical properties of signals in cyber-physical systems.

The language is still in its infancy but is indeed showing growing potential as a middle-man (-program?) between physical system descriptions and complicated back-ends that perform complex computations. In particular, the authors in (Wang κ. συν., 2019) use it as a medium for input specification in showcasing a new method for automatic model inference between multiple signals in a physical system, which they call Dimensional Function Synthesis. Moreover, in (Tsoutsouras κ. συν., 2020) they show how the method can be organically extended to circuit synthesis, again using Newton as input language.

4.1 Newton Language Constructs

In this section we present the language constructs that comprise the Newton language. A language construct is a syntactically allowable part of a program, formed by using one or more lexical tokens in accordance with the rules of a language's grammar [1]. A subset of the Newton language's syntax in the EBNF form will be presented incrementally for ease of access, starting with Figure 4.1, which presents the highest grammar rules for Newton. The full syntax of the original implementation may be found in (Lim κ. συν., 2017), while the current state of the grammar, presented here, is available via correspondence with the authors in (Lim & Stanley-Marbell, 2018).

[1] Language construct: term and definition standardized by ISO/IEC [ISO/IEC 2382-15:1999].

```
1 newtonDescription ::= ruleList.
2 ruleList ::= {rule}.
3 rule ::= constantDefinition   | invariantDefinition
4      |   baseSignalDefinition | sensorDefinition.
5
```

Figure 4.1: The syntax of the highest rules.

In Figure 4.1 we observe that every Newton specification file, designated `newtonDescription` in line 1 of the syntax, is essentially a list of rules. Each rule can be a constant definition, an invariant definition, a base signal definition, or a sensor definition. We will discuss these options in the subsequent paragraphs.

4.1.1 Signal definition

We start from the base signal definition, presented in Figure 4.2, as it proves to be a fundamental concept for the Newton language itself. Newton implements a dimensional type system using the signal construct. Each signal definition semantically defines a new signal type to be added to the existing available dimensions. The language does not define any primitive signals by itself but there exists a "`NewtonBaseSignals.nt`" file which is typically included, adding primitive signals like S.I. units (e.g. `angle`, `time` and `distance`) and common constants. Signals can also be defined as non-recursive compositions of other signals using the `derivationStatement`. This is useful to define derivative dimensions like for example `speed` or `acceleration`.

Expressions with signals must satisfy this dimensional type system to be accepted by the compiler. This ensures, as far as physical dimensions are concerned, the physical plausibility of the statements in a Newton description file (Kaparounakis κ. συν., 2020).

A signal definition is comprised at the very least from two statements: a symbol symbol statement (line 11) and a derivation statement (line 12). The symbol of a signal is used in physical signal (or unit) expressions alongside numerical expressions to make up quantity expressions. The units of an expression imply that the expression is of this signal (i.e. type); for example the expression `1 m` implies that the value `1` is of signal type `distance` (from `NewtonBaseSignals.nt`). The derivation of a signal is a signal expression (i.e. type expression) that shows how this signal may be derived from other signals. This is an essential part of ensuring dimensional type safety of expressions. That said, a physical quantity may also be dimensionless, so this is also supported with an homonymous keyword. The signal definition also optionally includes: A subdimension tuple (line 7); for multi-dimensional signals. A name statement (line 8); usually for the full name of

```
6  baseSignalDefinition ::= identifier ":" "signal"
7                           [subdimensionTuple] "=" "{"
8                           [nameStatement]
9                           [signalUncertaintyStatement]
10                          [sensorStatement]
11                          symbolStatement
12                          derivationStatement "}" .
13
```

Figure 4.2: The syntax of the base signal definition in the Newton language.

```
14 constantDefinition ::= identifier ":" "constant" "="
15                        numericFactor [unitFactor] ";" .
16
```

Figure 4.3: The syntax of the constant definition in the Newton language.

the symbol. A signal uncertainty statement (line 9); to indicate whether the signal is stochastic and following a stated distribution. And a sensor statement (line 10); to directly tie the signal with a certain sensor definition.

4.1.2 Constant definition

The constant definition, which appears in Figure 4.3 is used to define constants: components of physical laws that may not be variable, either in value or in dimensions. Each constant can be optionally associated with a signal type by specifying its symbol as a unit factor (line 15) after the value of the constant. Constants are frequently useful in physical descriptions; for example, the gravitational constant, the speed of light, the Planck constant and others can be specified along with their physical units for ease of writing and readability.

4.1.3 Sensor definition

The grammar of the Newton language also defines syntax for declaring sensors and sensor properties (Figure 4.4). Using the sensor definition (lines 17–18) a sensor and its properties can be coded as a language entity. In property list the Newton specification of the sensor is encoded. This specification can contain nominal characteristics about the sensor (e.g. from the manufacturer) such as range, uncertainty, accuracy and precision, as well as information on how to operate the sensor (e.g. erasure value and how to interface the sensor).

```
17 sensorDefinition ::= identifier ":" "sensor" parameterTuple
18                      "=" "{" sensorPropertyList "}" .
19 sensorPropertyList ::= sensorProperty {"," sensorProperty} .
20 sensorProperty ::= rangeStatement
21                 |   uncertaintyStatement
22                 |   erasureValueStatement
23                 |   accuracyStatement
24                 |   precisionStatement
25                 |   sensorInterfaceStatement .
26
```

Figure 4.4: The syntax of the sensor definition in the Newton language.

The main purpose of this statement is to automate interface programming in the context of cyber-physical systems. By encoding information about multiple sensors inside Newton description files, algorithm implementations may be abstracted from the various sensor readings necessary. Such algorithm implementation would make all sensor related calls to a stable Newton API, which would be responsible for interfacing the sensors.

4.1.4 Invariant definition

The invariant construct (Figure 4.5) is perhaps the most expressive one in the Newton language. It serves to specify physical laws between signals as constraints (lines 31–33). A quantity is a numerical value along with a signal type. Similarly, a quantity expression is a signal-typed numerical expression. A constraint is a comparison operation between quantity expressions that must be upheld whenever the invariant is expected to hold. A constraint could also be another invariant, written by invoking its identifier and passing its parameters.

Using the invariant definition we can specify expressions that we expect to hold true for a given physical system and compile an API that provides calls for us to ensure the upholding of these constraints at runtime. We can also use it to specify physical relationships between signals for input to some Newton compiler backend e.g. the estimator synthesis backend, which is the subject of this thesis, the C backend, that transpiles[2] Newton expressions to C code, or the dimensional function synthesis backend, that tries to infer the model of the parameters in the parameter tuple of the invariant based on dimensional analysis (Wang κ. συν., 2019).

The possible comparison operations appear in Figure 4.5 lines 34–35. Apart from the common operators with obvious semantics, there are three not so common

[2]Source-to-source compilation is often called transcompilation or transpilation.

```
invariantDefinition ::= identifier ":" "invariant"
                        parameterTuple "=" "{"
                        [constraintList] "}" .
constraintList ::= [constraint] {"," constraint} .
constraint ::= quantityExpression comparisonOperator
               quantityExpression
           |   identifier parameterTuple .
comparisonOperator ::= "o<" | "∼" | "<" | "<="
                   |   ">" | ">=" | "==" | "<->" .

```

Figure 4.5: The syntax of the invariant definition in the Newton language.

ones: `o<`, ∼ and `<->`. The first one (`o<`) is the *dimensionally agnostic proportional* comparison operator: it signifies that the two quantity expressions are numerically proportional, i.e. the value of the first equals the value of the other multiplied by some factor. The tilde (∼) is the *dimensionally matching proportional* comparison operator: it means that the two quantity expressions must have the same dimensional type as well as be numerically proportional. The last one (`<->`) simply states that the two quantity expressions are *related*, without specifying their relation. This can be useful in the context of model inference, by providing the inference program with extra information about the relationships between the physical signals.

4.1.5 Other remarks

Multidimensional signals

The signal definition statement supports declaring new signals that have more than one sub-dimensions. If a signal is multidimensional, its derivation statement should provide a derivation for each sub-dimensions. This is possible by either replicating an existing signal for each dimension (e.g. for 3-D distance), or by providing a tuple of quantity expressions inside braces.

When using a multidimensional signal inside expressions, it is possible to refer to one of the subdimensions using its index, bracketed after the signal identifier. For example, one could refer to the 2nd dimension of a 3-D distance signal like this: `distance3d[1]`.

Derivatives and integrals

There exist keywords for denoting derivatives and integrals inside the Newton language grammar as part of the expressions. The Newton compiler is able to

parse such expressions but they have no semantic value, since none of the existing backends are utilizing them.

Arrays and matrices

Declaring static arrays and matrices could be possible using the tuple notation for quantity expressions, although that's not the intended usage of that construct. As a result, there is no semantic support for such a thing. For example there's no ensuring that rows and columns are in accordance across the structure.

Note, however, that Newton is not a programming language and, as such, has no memory model and no variables.

Expression grammar

Newton uses the commonplace expression–term–factor distinction for parsing the `quantityExpression` production, apparent in lines 31– 32 in the grammar listing of Figure 4.5. This distinction facilitates the top-down parsing of expressions and enables effectuation of operator precedence at the grammar level. The syntax of quantity expressions in Newton is presented in the listing in Figure 4.6. The prefix "`quantity`" is used to underline that these expressions are dimensionally typed, as opposed the prefix "`numeric`" which indicates simple arithmetic expressions. In lines 44–46 the syntax for stating derivatives and integrals is shown. In lines 47–48 we see the notation for stochastic variables as distributions, added to the language in the context of this work. The same goes for support for transcendental functions (line 49).

4.2 Altering the Grammar

The Newton compiler uses a recursive-descent parsing scheme to parse the input. Generally, each non-terminal symbol of the Newton grammar has its own parser function, with the exception of a few that are simple and only appear once in other rules in the grammar. The parser is an LL(1) [3] parser (Lewis & Stearns, 1968) and utilizes FIRST and FOLLOW sets for the parsing table.

The FIRST and FOLLOW sets are sets of tokens that guide the parser on choosing the next rule to parse the rest of the input (Aho, Lam, Sethi, & Ullman, 2007). In the context of recursive-descent parsing, the FIRST set allows to parser to choose the next grammar rule to try and parse. When errors occur, the FOLLOW set allows the parser to provide cleaner feedback and error recovery. The construction of these sets is not a subject of this section; it is a well-established technique.

[3] Strictly speaking, the parser does use a lookahead of 3 on a single particular occasion. That said, the FIRST and FOLLOW sets are constructed for an LL(1) parser.

```
37 quantityExpression ::= quantityTerm
38                        {("+" | "-") quantityTerm} .
39 quantityTerm ::= [unaryOp] quantityFactor
40                  {("*" | "/"| vectorOp | "><")
41                  quantityFactor} .
42 quantityFactor ::= quantity
43                     ["**" numericFactor]
44     | ("derivative" | "integral")
45                    {("derivative" | "integral")}
46                     quantityFactor quantityFactor
47     | distribution "(" quantityExpression
48                    {"," quantityExpression} ")"
49     | transcendental "(" quantityExpression ")"
50     | "(" quantityExpression ")"
51                     ["**" numericFactor]
52     | "{" quantityExpression {"," quantityExpression} "}" .
53 quantity ::= numericConst
54          | (identifier ["[" numericFactor "]"]) .
```

Figure 4.6: The syntax of the quantity expression in the Newton language.

Using the Newton grammar, we can construct the FIRST and FOLLOW sets for the compiler's parser. In the Newton compiler codebase, these sets take the form of token-indexed arrays of token arrays. With the token for the grammar production we are trying to parse as index, we can see what tokens of terminal symbols we should see first and what terminal tokens could follow after the production has been parsed. These arrays are constructed automatically using the file `newton.ffi`, which encodes information about the rules and tokens that each FIRST and FOLLOW set must contain. Note that the final FIRST and FOLLOW sets contain only tokens. By being able to simply specify the tokens from another rule we can generate the final sets programmatically. This would otherwise be a daunting task.

The program `ffi2code` generates the final FIRST and FOLLOW sets as C arrays from the `newton.ffi` file. It is found in the `wirth-tools` repository [4].

The generated code must then be placed at specific parts of the codebase, this task is done manually. The auto-generated tokens for terminal symbols and production symbols must be placed in the common data structures header file, in the `IrNodeType` enumerator type definition, while ensuring the preservation of other entries that are not automatically generated. There are comments in the sources that guide this task.

[4] https://github.com/phillipstanleymarbell/Wirth-tools

4.2.1 Step-by-step guide

Here is a technical step-by-step guide for making alterations to the Newton grammar and its parser in the compiler. Ensure familiarity with the concepts presented in Section 4.2.

① Project the new changes in the grammar documentation file `newton.grammar`. Note that this file is just for reference; it is not parsed anywhere.

② Change the FIRST and FOLLOW sets in `newton.ffi` according to the changes from step ①. The definitions in `newton.ffi` can refer to tokens of other rules. There is no need to manually compute the final sets.

③ Generate the final sets via `ffi2code` from `wirth-tools`. The program is called using the following convention, where `*` is substituted with either `linux` or `darwin`.

```
./ffi2code-*-EN <path>/<to>/newton.ffi > autoGeneratedSets.c
```

The contents of the generated file (`autoGeneratedSets.c`) will be copied over to `newton-ffi2code-autoGeneratedSets.c` after the following changes.

❶ Cut the token and production `IrNodeType` values into the common data structures header file.

❷ Copy-over initial comment and definitions from `newton-ffi2code-autoGeneratedSets.c` into the newly generated file.

❸ Ensure correctness of variable names and dimensions in the newly generated file.

❹ Ensure each set ends with `kNewtonIrNodeTypeMax`.

❺ Add `Zeof` token after grammar production FIRSTs. Consult previous auto-generated sets file comments.

❻ Ensure FIRST set of every terminal token to be itself and the type max node value. Consult previous auto-generated sets file comments.

❼ Ensure strings array ends with an `XSeq` value. Consult previous auto-generated sets file comments.

❽ Ensure preservation of comments.

④ Copy the generated file contents to `newton-ffi2code-autoGeneratedSets.c`

⑤ Add any newly-created or remove any newly-removed tokens or productions in `newton-tokens.c` and `newton-productions.c`.

⑥ Make the necessary changes in the parser (in `newton-parser.c`) to be able to parse the new changes.

⑦ Test by trying to compile and by trying to use the compiler on a Newton description.

4.3 Extensions and Limitations

In the context of this work, we examine the Newton language as a choice for the specification of the input. Our needs, in terms of what the input language of our system would supply, have been mentioned in Chapters 2 and 3. Our central need is to be able to express the process and measurement models for the Kalman filters. Our secondary needs comprise support for computations with uncertainty and calculations of derivatives.

Support for expressing the two input models of the Kalman filter in the Newton language varies. On the one hand, the LKF (Section 2.3) most commonly uses matrix notation for the models and this is not directly supported in Newton, as explained in Section 4.1.5. On the other hand, the models in the EKF (Section 2.4) use functions and would thus benefit from being expressed freely, as expressions. This is very easy to do in Newton using the constraint list of the invariant definition from Section 4.1.

Notation for declaring uncertainty in Newton exists only in the context of the sensor definition. We extend this by adding an optional statement in the signal declaration statement. This addition is already present in Figure 4.2. This way, any signal can be optionally enhanced with uncertainty information by providing the expected distribution and its parameters as an argument. In the context of this work we have only worked with Gaussian distributions, but the keywords for many other common distributions were also added and, now, the compiler can be extended in a straightforward manner to support them.

Although there exists notation for derivatives, in the context of this work we do not care for reading derivatives from the input, but rather for finding the derivatives of the equations provided. Granted; if the derivatives are easy to find manually the user can easily instruct the Newton compiler to use them, declaring them using this notation.

There is no support for automatically finding derivatives of expressions in Newton. We overcome this issue by implementing Reverse Mode A.D., presented in Section 3.2, as a backend for the Newton compiler. This implementation is presented in Chapter 5.

4.4 Examples

We will now develop the Newton description files for the two working examples, the pendulum and the TurtleBot3 robot, first introduced in Section 2.4.3. These examples are used in the evaluation of the implemented system, subject of this work, in Chapter 6.

```
1  include "NewtonBaseSignals.nt"
2
3  g : constant = 9.80665 ajf; # Gravitational constant
4  bobMass : constant = 1 kg;  # Mass of the pendulum's bob
5
6  # Signal definition for damping factor
7  dampType : signal =
8  {
9      name         = "damptype" English;
10     symbol       = dtp;
11     derivation   = mass / second;
12 }
13
14 # Pendulum process model
15 pendulum_damped_process : invariant( theta   : angle,
16                                       dtheta  : angularRate,
17                                       dt      : time,
18                                       damp    : dampType,
19                                       L       : distance ) =
20 {
21     theta ~ theta + dtheta*dt,
22     dtheta ~ dtheta - damp/bobMass*dtheta*dt - g/L*sin(theta)*dt
23 }
24
25 # Pendulum measurement model
26 pendulum_measure : invariant( theta     : angle,
27                               dtheta    : angularRate,
28                               gyro_z    : angularRate ) =
29 {
30     gyro_z ~ dtheta
31 }
```

Figure 4.7: Physical description equations for the process and measurement models (Kaparounakis κ. συν., 2020) written in Newton (Lim & Stanley-Marbell, 2018). The pendulum process invariant is modeled after the damped pendulum (Equation 2.35), derived as explained in Section 2.4.3. The invariants' arguments list all identifiers involved in the equations, and their respective dimensions. Any identifier that is not a state or a measurement variable will be passed on to the generated functions as an extra argument.

4.4.1 Pendulum

To encapsulate the process and measurement models of the pendulum system (Equations 2.36, 2.37 and 2.38) we declare them as invariants in the Newton language. Figure 4.7 presents the Newton file developed for encoding this system that was also used in the evaluation of this work (Chapter 6). The models for the pendulum have been discussed in Section 2.4.3.

The process model of the pendulum system is coded with the invariant definition at lines 15 to 23. The invariant takes as parameters all the variables of the model: the state variables and other items such as the time difference `dt` (line 17), the damping factor `damp` (line 18) and the length of the pendulum's rod `L` (line 19). The global variables `g` and `bobMass` are declared as constants in lines 3 and 4 and can be used inside the invariant without being specified at the

parameter tuple.

The process model for the damped pendulum specifies two constraints inside its body (lines 21 and 22). The first and second constraints are the direct equivalents of Equations 2.36 and 2.37, respectively. The damping factor's dimensions are $\mathrm{kg\,s^{-1}}$, which is not represented by one of the provided signals included in `NewtonBaseSignals.nt`. For this reason we define a new signal with that derivation rule in lines 7-12 for declaring the type of `damp` in line 18 in the process model invariant parameter tuple.

The measurement model is encoded in the Newton description file using another invariant, named `pendulum_measure` (lines 26-31) for specifying Equation 2.38. The parameter tuple for the measurement invariant declares the state variables again, for this scope, in lines 26 and 27. This is because the state variables might be used at the left-hand sides of the constraints listed inside the invariant body. After the state variables, the measurement variable for the z-axis of the gyro is declared in line 28.

4.4.2 TurtleBot3 Burger

The Newton description of the TurtleBot3 Burger robot (Amsters & Slaets, 2019) is presented in Figure 4.8. Similarly, to the turtlebot models, we encode the physical dynamics behavior of the process and the laws governing the measurements inside Newton invariant declarations. The models for the TurtleBot3 robot have been discussed in Section 2.4.3.

The process model is encoded in invariant `turt_process` in lines 6-24. The invariant parameter tuple declares all the state variables (lines 6-11) and as well as other input variables (lines 12-14), like the speeds of the two wheels, `vr` and `vl`. The invariant body lists six constraints. Lines 17-20 describe the translational movement of the robot, corresponding to the first sub-case of the model (Equation 2.40), while lines 22-23 describe its rotational movement (Equation 2.41).

The measurement model is encoded in invariant `turt_measure` in lines 26-41. After re-declaring the state variables (lines 27-32), the parameter tuple also contains the measurement variables relevant to the model (lines 34-36). The constraints listed in the measurement invariant body (lines 38-40) describe the direct relation between measurements and the state variable, also encoded in Equation 2.43.

```
include "NewtonBaseSignals.nt"

l : constant = 0.16 m;  # Wheel separation distance

# Model makes the assumption that either vl == vr or vl == -vr
turt_process : invariant( x  : distance,
                          y  : distance,
                          vx : speed,
                          vy : speed,
                          theta : angularDisplacement,
                          omega : angularVelocity,
                          dt : time,
                          vr : speed,     # Right wheel speed
                          vl : speed ) =  # Left  wheel speed
{
    # if vl == vr:
    x ~ x + vx*dt,
    y ~ y + vy*dt,  # Only translational movement
    vx ~ (vr + vl)/2 * cos(theta),
    vy ~ (vr + vl)/2 * sin(theta),
    # if vl == -vr:
    theta ~ theta +  omega*dt,
    omega ~ (vr - vl)/l
}

turt_measure : invariant( # State
                          x  : distance,
                          y  : distance,
                          vx : speed,
                          vy : speed,
                          theta  : angularDisplacement,
                          omega  : angularVelocity,
                          # Measurement
                          odom_x : distance,
                          odom_y : distance,
                          gyro_z: angularVelocity ) =
{
    odom_x ~ x,
    odom_y ~ y,
    gyro_z ~ omega
}
```

Figure 4.8: Physical description equations for the process and measurement model of the TurtleBot3 Burger robot written in Newton (Lim & Stanley-Marbell, 2018). The robot process invariant is modeled after differential wheel drive dynamics of Equations 2.40 and 2.41, discussed in Section 2.4.3. The measurement model invariant is modeled after Equation 2.43. Similarly to Figure 4.8, the invariants' arguments list all identifiers involved in the equations, and their respective dimensions. Any identifier that is not a state or a measurement variable will be passed on to the generated functions as an extra argument.

Chapter 5

Automated Code Generation for State Estimation Algorithms

As has been discussed in Section 1.1, the noisy nature of sensors introduces error in measurements and impels the usage of state estimation methods in an effort to minimize it. Linear Kalman filters (Section 2.3) and extended Kalman filters (Section 2.4) are often an algorithm of choice for the state estimation or sensor fusion needs of cyber-physical embedded systems. Such methods dedicate a lot of effort in the fore-planning of a system and can limit the flexibility of development. We realize that there is an unmet need for automated methods for synthesizing small-footprint implementations of state-estimation algorithms in low-level languages, such as C, from high-level descriptions of the system's physical dynamics and measurement physical laws (Kaparounakis κ. συν., 2020).

We present a new tool for fast prototyping of state estimation code for embedded systems using the target system's physical description and governing physics laws. The tool uses the Newton language (Lim & Stanley-Marbell, 2018), presented in Chapter 4 as a front-end for system description input. We extend the specification capabilities of Newton, as detailed in Section 4.3, to accommodate for information needed for the generation of Kalman filters, and we implement a code generation methodology as a new backend for the Newton compiler. The resulting system provides an automated method for mapping high-level system descriptions to low-level source code that can be integrated into an embedded system.

The system's usage overview is the following: A researcher or engineer using our state estimation synthesis methodology provides a succinct input description of the process and measurement model of the Kalman filter, in the Newton language. The input is then processed by the Newton compiler's front-end, which also checks it in terms of dimensional type safety (see Section 4.1.1).

An accepted input is passed to the analysis and code generation backend back-

end which is the object of this work. The necessary analyses are performed to determine the linearity and derivatives of the state and measurement model equations and orchestrate the subsequent steps. At this step, it is also possible to employ Reverse Mode Automatic Differentiation for the calculation of the derivatives, which has been presented in Section 3.2. The code generation backend outputs code fragments by traversing the final AST.

The result code can then be integrated and used in an embedded system. The code provides a clear API and can, thus, easily function as a library or shared object. The calling system initializes the generated Kalman filter with an initial state and state covariance and can then ask for predictions and provide update measurements. The estimated state is available in the core data structure of the generated algorithm.

5.1 Implementation Algorithm

We implemented a system that automates the generation of linear and extended Kalman filters based on the physical description of the target system. We implemented this by writing a new backend for the Newton language compiler (Lim & Stanley-Marbell, 2018).

This design allows us to use and transform the parsed AST of the expressions in the input simplifying the algorithms we need to use. In general, the system performs a static analysis of the input during which it:

① determines the state and measurement vectors,

② locates every state and measurement vector variable appearance,

③ determines the linearity of the process and measurement model,

④ and orchestrates the C code generation based on all of the above.

In essence, the user-provided description of the target physical system must be processed in order to generate the source code of a Kalman filter. The resulting code is composed of the following three functions which are essential for the correct operation of the filter:

- **`filterInit()`:** Initializes state and state covariance.
- **`filterUpdate()`:** Ingests the input sensor measurements and incorporates them to the current state and covariance, in accordance with the *measurement* model.
- **`filterPredict()`:** Projects the state in time according to the *process* model.

Data: AST of input Newton description
Result: The generated Kalman filter

Step 1: Identify process and measurement model invariants.
Step 2: Discern state and measurement variables.
Step 3: Generate data required filter data structures.
Step 4: Generate `filterInit()` function.
Step 5: Generate `filterPredict()` function:
- **Step 5a:** Determine linearity of process model.
- **Step 5b:** Generate state transition functions.
- **Step 5c:** Construct either state transition matrix or Jacobian (**F**).
- **Step 5d:** Generate predict step matrix operations.

Step 6: Generate `filterUpdate()` function:
- **Step 6a:** Determine linearity of measurement model.
- **Step 6b:** Generate state measurement functions.
- **Step 6c:** Construct either state measurement matrix or Jacobian (**H**).
- **Step 6d:** Generate update step matrix operations.

Algorithm 1: Automatic code generation algorithm for Kalman filters.

The user input Newton description is parsed by the Newton parser on entry and converted into an abstract syntax tree (AST). This AST is the input to Algorithm 1, the Kalman filter code generation algorithm.

First, from the AST, the algorithm identifies the invariants that describe the *process* and *measurement* models (Step 1). Next (Step 2), the algorithm differentiates between variables that belong to the state versus those that belong to the measurement, as well as any extra variables that must be passed to the filter functions, according to the input specification. Next (Step 3), the algorithm generates the required data structures for the filter operation. This includes C language `struct` definitions for the core data of the filter (e.g., the current state and covariance), as well as enumerations of constant values. Based on the information gathered by the three initial steps, the algorithm identifies which structures and variables need to be initialized before the execution of the filter and thus produces the `filterInit()` function in Step 4 (Kaparounakis κ. συν., 2020).

Next (Step 5) the algorithm performs the operations for generating the source code of the `filterPredict()` function. The algorithm performs a linearity check in Step 5a, by examining the factors of the state variables and the functions used in the expressions. If it detects a non-linear case, it generates the required C expressions in Step 5b. In a linear case Step 5b is skipped. In Step 5c the algorithm generates the propagation matrix for the linear case or the Jacobian for the non-linear case. In Step 5d the algorithm generates the body of the Kalman filter `filterPredict()` and it connects the generated function to the results of the previous steps.

The algorithm follows a similar code generation flow in Step 6, to generate

the source code of the `filterUpdate()` function. The algorithm again performs a linearity check in Step 6a, and accordingly executes Steps 6b and 6c. In Step 6d the algorithm generates the necessary matrix operations for the calculations of the `filterUpdate()` equations of the Kalman filter (Kaparounakis κ. συν., 2020).

The code generation back-end outputs code fragments by traversing the AST corresponding to each user input. Reusable intermediate results are kept internally for use at subsequent steps of the code generation back-end. The source code is incrementally stored in a buffer and eventually written to a user-specified file.

5.2 System Overview

Our code generation process consists of two main stages: **Static Compile-Time Analysis** and **Dynamic Online Usage**. Figure 5.1 illustrates an end-to-end example of the operation and connections of all the sub-components discussed to achieve automated Kalman filter source code generation (Kaparounakis κ. συν., 2020).

In the **Static Compile-Time Analysis** stage the user provides a physical specification of the embedded system in question, encoded as an Newton language description file. The Newton compiler parses and dimensionally checks the file. The generated AST is directed to the code generation back-end which starts with a linearity check and proceeds accordingly to the generation of the expressions and the matrices required by the Kalman filter algorithm. The algorithmic steps involved in this process have been presented in Section 5.1.

In the **Dynamic Online Usage** stage the user evaluates the synthesized Kalman filter either in simulation or in the actual target embedded systems. The run-time stage of the filter commences with the initialization of the state and covariance. When the target computing system establishes its interaction with the physical environment, the prediction and update functions of the Kalman filter are invoked according to the target application control flow. The dynamically estimated state and covariance are available to any scope that can access the core data structures of the filter. An optional injection of a state is also supported.

5.3 API and Calling Conventions

The output of the code generation algorithm is the generated Kalman filter source code, which can be either a LKF or an EKF, depending on the linearity of the process or measurement model (Steps 5a and 6a in Algorithm 1). The generated filter depends on a small subset of linear-algebraic operations on matrices so we instruct the synthesized source code to automatically interface the final binary against a lightweight implementation of the necessary linear algebra functions. The users have the option to instruct our tool to avoid any automated hooking

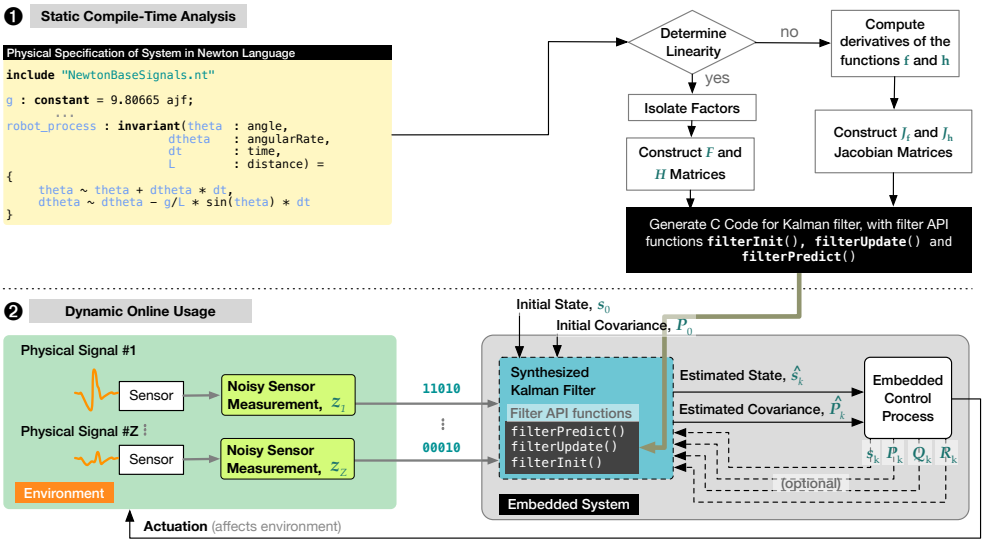


Figure 5.1: Detailed overview of the implemented system for automated code generation of state estimation and sensor fusion algorithms. The top part corresponds to offline work (e.g. compile-time work) on the part of the user or the system. The bottom part represents the functionality of the generated filter after deployment.

with linear algebra libraries so that they can use their libraries of preference or implement their own.

In parallels with the inner workings of the Kalman filters, presented in Chapter 2, the generated filter encompasses functionality for the Predict and Update steps. The operations to carry out these steps are encoded in the homonymously named functions `filterPredict` and `filterUpdate`, while the initialization of the filter with the initial state estimate ($\mathbf{s}_0$) and state estimate covariance matrix ($\mathbf{P}_0$) is achieved with a call to function `filterInit`. Table 5.1 summarizes the calling convention and functionality of the generated filter's API.

In order to ensure that the end-user maintains total control over data struc-

`void filterInit(CoreState *cs, double s0[N], double P0[N][N]);` → Initialize state estimate and state estimate covariance matrix of structure `cs`.
`void filterPredict(CoreState *cs, double dt, double input[U]);` → Projects the state estimate and state estimate covariance matrix from structure `cs` forward in time by `dt` according to the process model.
`void filterUpdate(CoreState *cs, double measure[Z]);` → Incorporates the measurements from the measurement array `measure` to the current state estimate and covariance matrix in `cs`.

Table 5.1: Exported API for interaction with synthesized Kalman filter. The cardinalities of the arrays of the arguments N, U and Z correspond to the state vector cardinality, the control input vector cardinality and the measurement vector cardinality respectively, and in accordance with the notation presented in Section 2.1.

tures (and, by extension, memory space) as well as to facilitate data access in a straightforward way, the core data of the filter (e.g. the current state estimate and its covariance matrix) are passed by reference in the arguments to the previously mentioned functions. The general scheme of the core data C `struct` definition is presented in Listing 5.1.

The C `struct` for containing the core data of the filter must be allocated by the user, in whatever allocation scheme they deem necessary for their application, and must be initialized with an initial estimate and estimate covariance with `filterInit` prior to calling either of `filterPredict` or `filterUpdate`. Lines 8, 13, 18 and 23 of Listing 5.1 contain references to matrix representations of the C array in each previous line. This are intended for usage with any linear algebra library that might offer such representation. In case the user opts to implement their own operations they may alter them or delete them, accordingly. Finally, the estimate of the filter can be directly accessed in either of the two first structure variables (lines 7 or 8) with the restriction that no API call is in process.

Listing 5.1: C struct of the generated filter's core data.

```
1  typedef struct CoreState CoreState;
2  struct CoreState
3  {
4    /*
5     *  State estimate
6     */
7    double S[STATE_DIMENSION];
8    matrix *Sm;
9    /*
10    *  State estimate covariance matrix
11    */
12   double P[STATE_DIMENSION][STATE_DIMENSION];
13   matrix *Pm;
14   /*
15    *  Process noise covariance matrix
16    */
17   double Q[STATE_DIMENSION][STATE_DIMENSION];
18   matrix *Qm;
19   /*
20    *  Measurement noise covariance matrix
21    */
22   double R[MEASURE_DIMENSION][MEASURE_DIMENSION];
23   matrix *Rm;
24 };
```

5.4 Automatic Differentiation on the AST

As mentioned in Section 2.4.2, a considerable obstacle in the incorporation of the EKF in embedded systems arises from the need to derive the Jacobian matrices of the non-linear physical models of the system. We address this challenge by using Automatic Differentiation (A.D.), an established technique in mathematics and computer science, presented in Chapter 3, that accurately computes the derivatives

of code operations using the chain rule of derivatives.

Specifically, in the context of this work, we have chosen to implement Reverse Mode A.D. (Section 3.2) because of the input setup. As explained in Section 3.3, Reverse Mode A.D. we can calculate the partial derivatives of one output variable with respect to all input variables in the same pass. Since our input is essentially a list of N functions, one for each state variable, where N is the cardinality of the state vector. As such, we can calculate the Jacobian matrix with N evaluations (one per function). Conversely, with this setup choosing Forward Mode A.D. would force us to do N^2 evaluations of the same functions to get the whole Jacobian matrix.

Using an Automatic Differentiation technique as part of the compiler grants us access to the AST generated for the expressions. We can then traverse the tree and provide annotations for subsequent passes to the same expression.

The A.D. engine for Newton was implemented as compiler back-end and uses a three-pass method to annotate the AST nodes and then generate the output code for both the original expression and its derivative in the SSA form. In the first pass (Listing 5.2), with a pre-order traversal, each node of the tree is assigned a "variable identifier". This will identify the intermediate result of this node in the SSA form. In the second pass (Listing 5.3), with a post-order traversal, the SSA form of the expression is printed to a buffer using those identifiers. In the third (Listing 5.4), with a pre-order traversal, the reverse SSA form for the calculation of the chained partial derivatives is exported (Kaparounakis κ. συν., 2020). By design, these three passes are abstracted from the estimator synthesis back-end, making the technique available to any C code generation back-end or other compiler with similar AST structure. A more technical discussion of the three relevant listings will be given in later paragraphs of this section.

For the case of EKF, the user can choose between Standard and Automatic Differentiation. The key differences between these in the generated code are: (i) the state and measurement equations use functions instead of matrices; (ii) the Jacobian matrix is calculated at each step using the derivatives of state and measurement functions as follows. **Standard differentiation:** Another set of $O(N^2+Z^2)$ functions are created for the derivative of each function with regards to each state variable. **Automatic differentiation:** The generation of the first $O(N+Z)$ functions is hi-jacked to produce the SSA form code of the expression and the Reverse Mode AutoDiff calculation of the derivatives, detailed in Section 3.2 (Kaparounakis κ. συν., 2020).

We will know present the C code of the Reverse Mode A.D. implementation for the Newton compiler's AST of the parsed expressions of the Newton language. It is appropriate to point out here that the AST follows the rules for quantity expressions, presented in Section 4.1.5. What this means for our implementation, from a structural perspective, is that `quantityExpression` AST nodes contain a list of `quantityTerm` nodes, which in turn contain lists of `quantityFactor` nodes. The `quantityFactor` nodes can then contain further operations like exponentiation,

function application, or sub-`quantityExpression` nodes.

The three passes are spread across three C functions that either mutate the AST given to them as an argument or generate C code. These functions make calls to functions `annotate(char *str, const char *format,...)`, `generate(const char *format,...)` and `nodeToStr(State* N, IrNode* node)`. Function `annotate` formats a string and attaches it to the node. Function `generate` formats a string and appends it to an internal buffer. This buffer contains the result C code is printed to a file after successful compilation. Finally, `nodeToStr` generates a string from a terminal node.

5.4.1 Annotation pass

The C code for the annotation pass, the first pass of the Automatic Differentiation implementation for the expression AST, is listed in Listing 5.2. The pass is encoded in function `astADAnnotate`. The Newton compiler makes effort to have stateless computations, so the state of the compiler is the first argument of almost any function. The second argument of `astADAnnotate` is the current AST node, while the third is the initiating token string, as explained in this section.

The goal of the function is to recursively attach a unique identifier string to the root node and its child nodes. It achieves this by extending a string, passed by the caller in the arguments as `parentTokenString`, with the integer in the `uid` variable, declared at line 2. The variable `uid` is incremented after each use and, this way, all child nodes are assigned a unique identifer string which is the concatenation of their parent's string and child's serial number (lines 10, 34 and 49–50). The actions of the function are guided by the type of the node using a `switch` statement (line 69). In the case that the root is either a `quantityTerm` or a `quantityFactor`, we do not need to annotate the root, as it has already been annotated by the caller. This is coded in the condition at lines 30–32, which excludes `quantityTerm` nodes from being annotated, and at lines 14–25 which simply do not annotate the root node.

Afterwards, `astADAnnotate` is called for each child, passing their newly created unique identifer string as the `parentTokenString`, to recursively continue the annotation process (lines 56, 18 and 22). Fist annotating and then recursing amounts to the function annotating the AST in a pre-order manner. The `for` loops of the code bypass the operator nodes that are in-between value nodes.

Listing 5.2: Annotation of the AST for Reverse Mode A.D..

```
1 void astADAnnotate(State* N, IrNode* root, char* parentTokenString) {
2   int uid = 1;
3   switch (root->type)
4   {
5     case typeQuantity:
6     {
7       /*
8        *  Self-assign name for Static Single Assignment expression
9        */
```

```
10        annotate(root->tokenString, "%s_%d", parentTokenString, uid++);
11        break;
12      }
13
14      case typeQuantityFactor:
15      {
16        if (root->irLeftChild->type == typeTranscendental)
17        {
18          astADAnnotate(N, RL(root), root->tokenString);
19        }
20        else
21        {
22          astADAnnotate(N, root->irLeftChild, root->tokenString);
23        }
24        break;
25      }
26
27      case typeQuantityExpression:
28      case typeQuantityTerm:
29      {
30        if (root->type == typeQuantityExpression &&
31          root->tokenString == NULL &&
32          parentTokenString != NULL)
33        {
34          annotate(root->tokenString, "%s_%d", parentTokenString, uid++);
35        }
36
37        IrNode* currXSeq = NULL;
38        uid = 1;
39        for (currXSeq = root; currXSeq != NULL;
40                              currXSeq = currXSeq->irRightChild)
41        {
42          if (currXSeq->irLeftChild->type == typeQuantityTerm ||
43              currXSeq->irLeftChild->type == typeQuantityFactor )
44          {
45            /*
46             *  Assign each child a name for
47             *  Static Single Assignment expression
48             */
49            annotate(currXSeq->irLeftChild->tokenString,
50                     "%s_%d", root->tokenString, uid++);
51          }
52        }
53        for (currXSeq = root; currXSeq != NULL;
54                              currXSeq = currXSeq->irRightChild)
55        {
56          astADAnnotate(N, currXSeq->irLeftChild, root->tokenString);
57        }
58      }
59      default:
60        /*
61         *  Code relies on irrelevant cases (e.g. operators)
62         *  hitting the default rule.
63         */
64        break;
65    }
66  }
```

5.4.2 SSA form code generation pass

The C code for the second pass, the generation of the original expression into SSA from, is listed in Listing 5.3. Function `astADGenExpressionSSA` takes as arguments the state of the compiler and the current AST node (line 67). The goal of the function is to generate each assignment of the SSA form of original expression using the unique identifier tokens of each AST node from the annotation pass (Section 5.4.1), format it as a string and store it in an internal buffer of the compiler (using function `generate`). Similarly to the annotation pass, the function is recursive and we again use the node's type to guide our actions (line 69).

In this pass we are generating SSA code for the evaluation of the original expression. The AST node passed to the second argument of the initial call to function `astADGenExpressionSSA` is the root AST node represents the result of the whole expression and is comprised from sub-expressions (terms, factors or other expressions in parentheses). So, to achieve the correct order of operations, with the result being evaluated last, we need to generate the code for this expression last. So, we first call `astADGenExpressionSSA` for each child (lines 108–112) and afterwards generate the last assignment in lines 114–147. It is evident that we need to recursively apply this scheme to any node type that is not a single value. Terms are handled in the same case as expressions. Lines 78–102 handle factors, taking care to correctly generate code for exponentiation operation or transcendental function application.

The order in which the operations are generated amounts to a post-order traversal of the AST.

Listing 5.3: Generation of the SSA form of the AST expression for Reverse Mode A.D..

```
67 void astADGenExpressionSSA(State* N, IrNode* root)
68 {
69   switch (root->type)
70   {
71     case typeQuantity:
72     {
73       generate("double %s = %s;\n", root->tokenString,
74                                     nodeToStr(N, root->irLeftChild));
75       break;
76     }
77
78     case typeQuantityFactor:
79     {
80       if (R(root) &&
81           RL(root)->type == typeExponentiationOperator)
82       {
83         astADGenExpressionSSA(N, L(root));
84         generate("double %s = pow(%s, %f);\n", root->tokenString,
85                                                root->irLeftChild->tokenString,
86                                                RRL(root)->value);
87       }
88       else if (root->irLeftChild->type == typeTranscendental)
89       {
90         astADGenExpressionSSA(N, RL(root));
91         generate("double %s = %s(%s);\n", root->tokenString,
92                                           nodeToStr(N, LL(root)),
```

```
93                                          RL(root)->tokenString);
94        }
95        else
96        {
97          astADGenExpressionSSA(N, root->irLeftChild);
98          generate("double %s = %s;\n", root->tokenString,
99                                        root->irLeftChild->tokenString);
100       }
101       break;
102     }
103
104     case typeQuantityExpression:
105     case typeQuantityTerm:
106     {
107       IrNode* currXSeq = NULL;
108       for (currXSeq = root; currXSeq != NULL;
109                             currXSeq = currXSeq->irRightChild)
110       {
111         astADGenExpressionSSA(N, currXSeq->irLeftChild);
112       }
113
114       generate("double %s = ", root->tokenString);
115       if (root->irLeftChild->type == typeUnaryOp)
116       {
117         /*
118          *  Root contains the unary operator
119          */
120         generate("%s", nodeToStr(N, LL(root)));
121       }
122       else
123       {
124         /*
125          *  Root contains the actual first factor
126          */
127         generate(" %s ", root->irLeftChild->tokenString);
128       }
129
130       for (currXSeq = root->irRightChild; currXSeq != NULL;
131                                           currXSeq = currXSeq->irRightChild)
132       {
133         if (currXSeq->irLeftChild->type == typeLowPrecedenceOperator)
134         {
135           generate("%s", nodeToStr(N, LL(currXSeq));
136         }
137         else if (currXSeq->irLeftChild->type ==
138                                         typeHighPrecedenceQuantityOperator)
139         {
140           generate("%s", nodeToStr(N, LLL(currXSeq)));
141         }
142         else
143         {
144           generate("%s", currXSeq->irLeftChild->tokenString);
145         }
146       }
147       generate(";\n\n");
148     }
149
150     default:
151       /*
152        *  Code relies on irrelevant cases (e.g. operators)
153        *  hitting the default rule.
154        */
```

```
155         break;
156     }
157 }
```

5.4.3 Reverse A.D. SSA code generation pass

In the third and final pass, SSA code of the reverse calculation of the partial derivatives using the chain rule (Section 3.2) is generated. The relevant code appears in Listing 5.4. Function `astADGenReverse` is also recursive and has the same arguments and `switch` structure. We are now generating the assignments in a reverse order so the root node must be generated first. This amounts to a pre-order traversal of the AST.

We now have to take care to calculate the partial derivative of each assignment with respect to the unique identifer tokens that appear in them. We prefix derivative variables with the character "`g`" to denote that the concatenated string is a partial derivative of the root of the expression with this respect to the corresponding unique identifier token variable. This is an arbitrary choice that has nothing to do with Street Fighter.

On the case of a `typeQuantityExpression` (lines 162–179), for each child term, we generate the reverse operation according to rules presented in Section 3.2 (lines 165–166, 172–174) and call the function on the child (lines 167 and 175). On line 173, we make sure to generate the correct relation according to the low precedence operator between the expressions.

The procedure is a bit more complicated in the case of a `typeQuantityTerm`, because the common derivation rules become more intricate in the case of multiplication and division. We generate a separate Reverse A.D. assignment for each of the term's factors by iterating on the list of factors and operators (`for` loop at line 194), ensuring to bypass operators with lines 196–199.

If the current partial derivative is w.r.t. the first factor of the term then it's handled on lines 208–215. We do not check any operators here because it's impossible to start a term with a division's denominator, i.e. if the first operation of a term is a division the first factor will be the numerator. Then the rest of the factors are generated inside the nested `for` loop at lines 236–243. The partial derivatives w.r.t the rest of the factors are handled inside the nested `for` loop at lines 223–235, according to the preceding operator. If it's a multiplication, "`1`" is generated; if it's a division, the division's derivation rule is used. At line 246, the generation of the reverse assignments for each of the term's factors is invoked via recursion.

The `typeQuantityFactor switch` case handles the partial derivative generation for exponentiation (lines 254–262) and transcendental function application (lines 263–270). The `generate` call format string of the exponentiation `if` case might seem somewhat unwelcoming due to the usage of the POSIX-defined po-

sitional printf argument notation[1]. If, instead, a special function is called that encodes transcendental derivatives as another switch statement. A subset of that switch statement appears in Listing 5.5.

Finally, the typeQuantity (lines 280–294) is trivial, simply checking if the node corresponds to a constant (lines 282–283) before generating the node's unique token as a variable name. The reason for this is that the partial derivative of the result w.r.t. a constant is irrelevant since constant do not change. Constants normally do not present a problem to the algorithm but they do present a problem to the variable naming scheme here.

Listing 5.4: Generation of the reverse SSA that calculates the partial derivatives of the original AST expression for Reverse Mode A.D..

```
158 void astADGenReverse(State* N, IrNode* root)
159 {
160   switch (root->type)
161   {
162     case typeQuantityExpression:
163     {
164       IrNode* currXSeq = NULL;
165       generate("double g%s = g%s;\n\n", root->irLeftChild->tokenString,
166                                         root->tokenString);
167       astADGenReverse(N, root->irLeftChild);
168       generate("\n");
169       for (currXSeq = root->irRightChild; currXSeq != NULL;
170                                           currXSeq = RR(currXSeq))
171       {
172         generate("double g%s = %sg%s;\n", RL(currXSeq)->tokenString,
173                                           nodeToStr(N, LL(currXSeq)),
174                                           root->tokenString);
175         astADGenReverse(N, RL(currXSeq));
176         generate("\n");
177       }
178       break;
179     }
180
181     case typeQuantityTerm:
182     {
183       IrNode* firstFactor = root;
184       IrNode* firstOperator = root->irRightChild;
185       bool startsWithUnaryOp = false;
186       if (root->irLeftChild->type == typeUnaryOp)
187       {
188         firstFactor = root->irRightChild;
189         firstOperator = RR(root);
190         startsWithUnaryOp = true;
191       }
192
193       for (IrNode * currXSeq = firstFactor; currXSeq != NULL;
194                                             currXSeq = R(currXSeq))
195       {
196         if (L(currXSeq)->type == typeHighPrecedenceQuantityOperator)
197         {
198           continue;
199         }
200
```

[1]See man 3 printf paragraph "Format of the format string"

```
201         generate("double g%s = ", currXSeq->irLeftChild->tokenString);
202         if (startsWithUnaryOp == true)
203         {
204           generate("%s", nodeToStr(N, LL(root)));
205         }
206         generate("g%s", root->tokenString);
207 
208         if (currXSeq == firstFactor)
209         {
210           generate(" * 1");
211         }
212         else
213         {
214           generate(" * %s", firstFactor->irLeftChild->tokenString);
215         }
216 
217         /*
218         * Iterate over operators only (notice index update clause)
219         */
220         for (IrNode* currXSeqOp = firstOperator; currXSeqOp != NULL;
221                                                  currXSeqOp = RR(currXSeqOp))
222         {
223           if (currXSeq == currXSeqOp->irRightChild)
224           {
225             /*
226              *  Derivation is w.r.t. R(currXSeqOp)
227              */
228             if (LLL(currXSeqOp)->type == typeDiv) {
229               generate(" * (-1/pow(%s, 2))", RL(currXSeqOp)->tokenString);
230             }
231             else
232             {
233               generate(" * 1");
234             }
235           }
236           else
237           {
238             /*
239              *  R(currXSeqOp) is simply a factor.
240              */
241             generate("%s%s", nodeToStr(N, LLL(currXSeqOp)),
242                              RL(currXSeqOp)->tokenString);
243           }
244         }
245         generate(";\n");
246         astADGenReverse(N, currXSeq->irLeftChild);
247         generate("\n");
248       }
249       break;
250     }
251 
252     case typeQuantityFactor:
253     {
254       if (R(root) && RL(root)->type == typeExponentiationOperator)
255       {
256         generate("double g%1$s = g%2$s * %3$f * pow(%1$s, %4$f);\n",
257                                                L(root)->tokenString,
258                                                root->tokenString,
259                                                RRL(root)->value,
260                                                RRL(root)->value - 1);
261         astADGenReverse(N, root->irLeftChild);
262       }
```

```
263       else if (root->irLeftChild->type == typeTranscendental)
264       {
265         generate("double g%s = g%s * ", RL(root)->tokenString,
266                                         root->tokenString);
267         generateTranscendentalDerivative(LL(root)->type
268                                         RL(root)->tokenString);
269         astADGenReverse(N, RL(root));
270       }
271       else
272       {
273         generate("double g%s = g%s;\n", root->irLeftChild->tokenString,
274                                         root->tokenString);
275         astADGenReverse(N, root->irLeftChild);
276       }
277       break;
278     }
279
280     case typeQuantity:
281     {
282       if (root->irLeftChild->type == typeIdentifier &&
283         root->irLeftChild->physics->isConstant == false)
284       {
285         generate("g%s += g%s;\n", nodeToStr(N, root->irLeftChild),
286                                   root->tokenString);
287       }
288       else
289       {
290         generate("// %1$s is a constant, g%1$s undefined\n",
291                               nodeToStr(N, root->irLeftChild));
292       }
293       break;
294     }
295
296     default:
297       /*
298        *  Code relies on irrelevant cases (e.g. operators)
299        *  hitting the default rule.
300        */
301       break;
302   }
303 }
```

Listing 5.5: Subset of the transcendental reverse assignment generation `switch` statement.

```
304 ...
305 case kNewtonIrNodeType_Tsin:
306 {
307   flexprint(N->Fe, N->Fm, N->Fpc, "cos(%s);\n", RL(root)->tokenString);
308   break;
309 }
310 case kNewtonIrNodeType_Tcos:
311 {
312   flexprint(N->Fe, N->Fm, N->Fpc, "(-sin(%s));\n", RL(root)->tokenString);
313   break;
314 }
315 ...
```

5.5 Matrix Inference from AST

The implemented system's algorithm (Algorithm 1) examines whether either of the process and the measurement model are linear or not. This is achieved by checking if all the equations inside either `invariant` definition are linear. The prerequisites for linearity, in this scenario, are that no more than one state variables are present in the same term and no transcendental function or exponentiation is applied on any state variable.

In the case that either `invariant` is detected to be linear we forgo Automatic Differentiation of the expressions and directly construct the, now static, corresponding matrix (either **F** or **H**). The code that generates matrix **F** is presented in Listing 5.6. Essentially, it generates an array declaration (line 318) and initialization (lines 319–346). Each element of the array is corresponds to the respective factor of each state variable in the current constraint. So, iterating on the constraints (line 319) and the state variables (line 328), we isolate the AST nodes of the factors (lines 330–333) for the current state variable. If there are no factors, we generate "0" for the current matrix position (line 336), else if there are, we generate their string expression using `irPassCConstraintTreeWalk` (line 340).

Listing 5.6: State transitiona matrix inference from the AST.

```
316 IrNode* fMatrixIrNodes[stateDimension][stateDimension];
317 int fRow = 0;
318 generate("double fMatrix[STATE_DIMENSION][STATE_DIMENSION] = \n{");
319 for (constraintXSeq = processInvariant->constraints;
320                       constraintXSeq != NULL;
321                       fRow++, constraintXSeq = constraintXSeq->irRightChild)
322 {
323   /*
324    *  Right Hand Side Expression of the constraint:
325    */
326   IrNode *  RHSExpressionXSeq = LRRL(constraintXSeq);
327   generate("{ ");
328   for (int fColumn = 0; fColumn < stateDimension; fColumn++)
329   {
330     fMatrixIrNodes[fRow][fColumn] =
331           irPassEstimatorSynthesisIsolateSymbolFactors(N,
332                                       RHSExpressionXSeq,
333                                       stateVariableSymbols[fColumn]);
334     if (fMatrixIrNodes[fRow][fColumn]->irRightChild == NULL &&
335         fMatrixIrNodes[fRow][fColumn]->irLeftChild  == NULL) {
336       generate("( 0 ), ");
337     }
338     else
339     {
340       irPassCConstraintTreeWalk(N, fMatrixIrNodes[fRow][fColumn]);
341       generate(", ");
342     }
343   }
344   generate("},\n");
345 }
346 generate("};\n");
```

Chapter 6

Evaluation

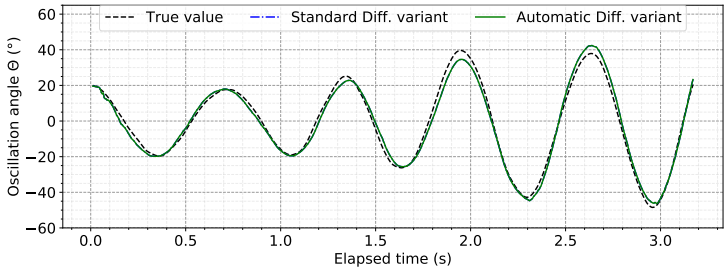
We evaluate the generated Kalman filters for progressively complex dynamic systems. We start with an evaluation on a simulation of a Pendulum and then go on to use the filter on simulated data from a 2-wheel ground robot. We compare the performance of using either Standard Differentiation or Automatic Differentiation for a given system. Throughout these experiments we use $h = 0.0005$ for the Standard Limit Differentiation, unless otherwise stated.

6.1 Simulated Pendulum

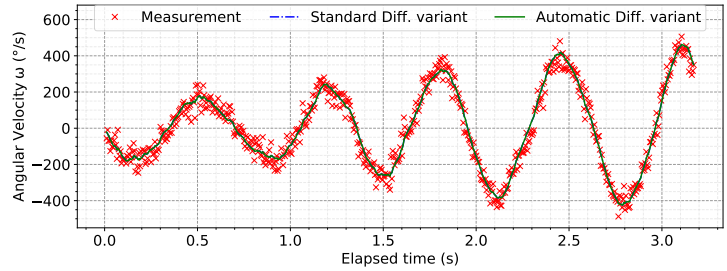
In our first experiments we target the problem of state estimation of a physical system which simulates the oscillation of an ideal pendulum. Let b be the damping constant with units of $kg\,s^{-1}$ in the S.I. system (Mohazzabi κ. συν., 2017), and let m be the mass of the pendulum's bob. The equation of the dynamics including drag are presented in Equation 6.1. The equation presented here has also been discussed in Section 2.4.3.

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \frac{b}{m}\frac{\mathrm{d}\theta}{\mathrm{d}t} + \frac{g}{l}\sin(\theta) = 0 \tag{6.1}$$

We developed Newton descriptions similar to the one presented in Figure 4.7 to describe the pendulum with and without drag in order to use our implemented Newton compiler back-end for the generation of Kalman filters. The state vector of the system is $< \theta_t, \omega_t >$, composed of the oscillation angle and the angular velocity, respectively. The input of the filters is the angular velocity provided by a simulated gyroscope. The measurements from the gyroscope are noisy and the noise was modelled, without lack of generality, according to a Gaussian distribution. The angular rate of the gyroscope is the only sensory input to the generated Kalman filters, which are completely agnostic to the deviation of the estimated angle from the true one. We generated two different variants of the filter, one



(a) Experiment 1: Estimation of oscillation angle over time.



(b) Experiment 1: Estimation of angular velocity over time.

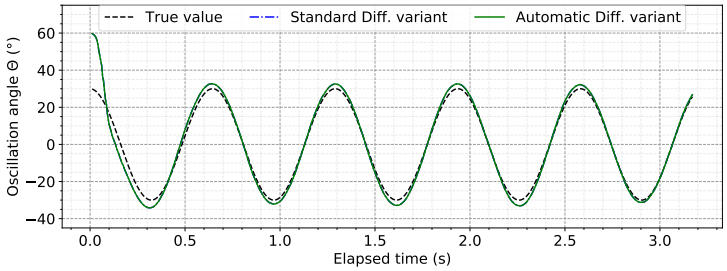
Figure 6.1: Evaluation of generated Kalman filter for estimation of the state a pendulum. Sub-figures (a) and (b) correspond to an oscillation with initial displacement of 20° and observable process noise of variance $0.005\,\text{rad}^2/s^2$ in the angular velocity ω of the oscillation (notice the changes in maximum amplitude and frequency). The input to the filters is angular velocity from a gyro with noise of variance $0.5\,\text{rad}^2/s^2$. All generated filter variants manage to follow the noisy process with a mean square error of $0.0025\,\text{rad}^2$ for the angle and $0.17\,\text{rad}^2/s^2$ for the angular velocity.

that uses standard differentiation and one that uses Automatic Differentiation (Kaparounakis κ. συν., 2020).

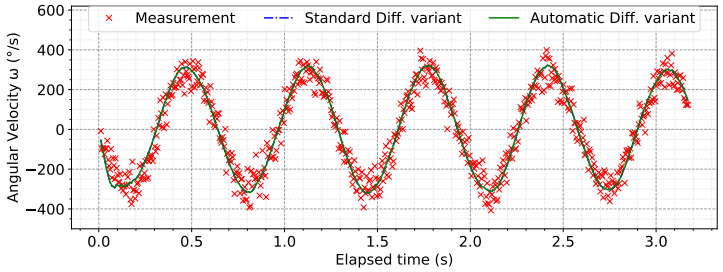
6.1.1 Pendulum observable process noise

We first evaluate a pendulum without drag (meaning that $b = 0$ in Equation 2.35), using its dynamics equation to simulate its oscillation in time we gather a trace of the state and sensor values. We then use the generated Kalman filters to predict the time-evolution of the pendulum state parameters in the presence of noisy input sensor measurements (Kaparounakis κ. συν., 2020). The results of the experiment are presented in Figure 6.1.

Figures 6.1(a) and 6.1(b) present an execution interval of the simulation, with initial pendulum displacement of 20° and observable process noise of variance $0.005\,\text{rad}^2/s^2$ on the angular velocity ω. The sensor error distribution has zero mean value and variance equal to $0.5\,\text{rad}^2/s^2$. The sensed values from the gyroscope are annotated in Figure 6.1(b) with red points, combined with the pendulum angular velocity predicted by the two Kalman filter variants. In Figure 6.1(b), we



(a) Experiment 2: Estimation of oscillation angle over time.



(b) Experiment 2: Estimation of angular velocity over time.

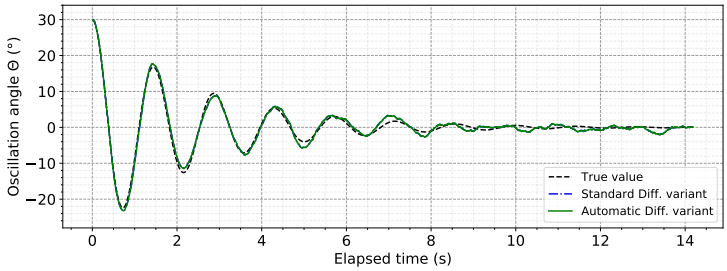
Figure 6.2: Sub-figures 6.2(a) and 6.2(b) correspond to a different pendulum simulation with initial displacement of 30°, measurement noise of variance $0.8\,\mathrm{rad}^2/s^2$. Examined filters were deliberately erroneously initialized with the knowledge of an initial displacement of 60°. Nevertheless, they manage to converge to accurate estimations about the state of the pendulum.

provide the predicted pendulum angle by the two Kalman filters compared against the true value. We observe that both filter variants are capable of correctly estimating the oscillation with mean square error (MSE) of $0.003\,\mathrm{rad}^2$.

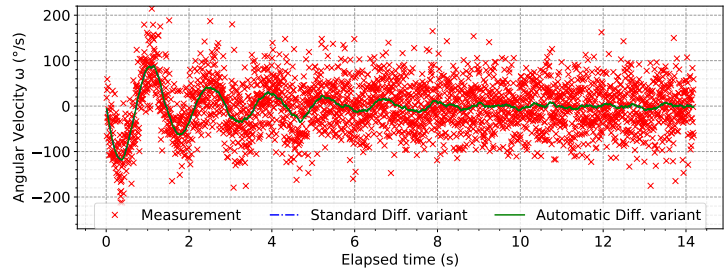
6.1.2 Pendulum with false initial displacement

In our second experiment using the pendulum with no drag, we initialize the generated Kalman filters with a "false" initial angle displacement value. In this way, we evaluate their ability to estimate the state of the system when the initial state information are not accurate (Kaparounakis κ. συν., 2020). The initial displacement was set to 30°, while the filters were initialized with knowledge of 60° displacement.

We provide the results of this experiment in Figures 6.2(a) and 6.2(b). We observe in Figure 6.2(a) that both filter variants are capable of converging to accurate predictions of the system's state. For both filters, the MSE of the prediction of the oscillation angle θ was $0.0056\,\mathrm{rad}^2$, while the MSE value for the prediction of the angular rate ω was $0.1757\,\mathrm{rad}^2/s^2$.



(a) Experiment 3: Estimation of oscillation angle over time.



(b) Experiment 3: Estimation of angular velocity over time.

Figure 6.3: State estimation of a pendulum oscillation with drag. Initial displacement is equal to 30° and measurement noise variance $0.8\,\text{rad}^2/s^2$. Both generated filter variants, i.e., with standard or Automatic Differentiation, provide high accuracy estimates of the oscillation with diminishing amplitude.

6.1.3 Pendulum with drag

We performed a third experiment for the state estimation of a pendulum with drag, length equal to $0.5\,m$ and damping factor equal to $0.8\,kg\,s^{-1}$ (Kaparounakis κ. συν., 2020). The initial angle displacement was 30° and both filter variants where provided correct information about this value. The results of this experiment are shown in Figure 6.3. The measurement noise variance was $0.8\,\text{rad}^2/s^2$.

We observe that both Kalman filter variants are capable of making accurate predictions about the system state, despite the noise in measurements, annotated with red points in Figure 6.3(b). The MSE of the prediction for both filters was approximately $0.0002\,\text{rad}^2$ for the prediction of the oscillation angle θ and $0.0054\,\text{rad}^2/s^2$ for the prediction of the angular rate ω.

6.2 TurtleBot3 Robot Simulation

To evaluate the effectiveness of the generated filters on the state estimation of a complex cyber-physical system, we make use of the TurtleBot3 Burger (Amsters & Slaets, 2019), a robot built on open-source software operating system (ROS) (Quigley κ. συν., 2009). The TurtleBot3 Burger is a differential drive robot, i.e., it controls its linear and angular velocities by actuating on a set of wheels with

individual speed setpoints v_r and v_l, for the right and the left wheel, respectively (Kaparounakis κ. συν., 2020). The robot's build is illustrated in Figure 6.4 and its general kinematic model has been presented in Section 2.4.3

We generated the robot process Newton invariant as a unified model and hardcoded a C control flow command in the generated *Predict* function to check the system's input, i.e. the velocities v_r, v_l, and perform the correct operations according to either Equation 2.40 or Equation 2.41.

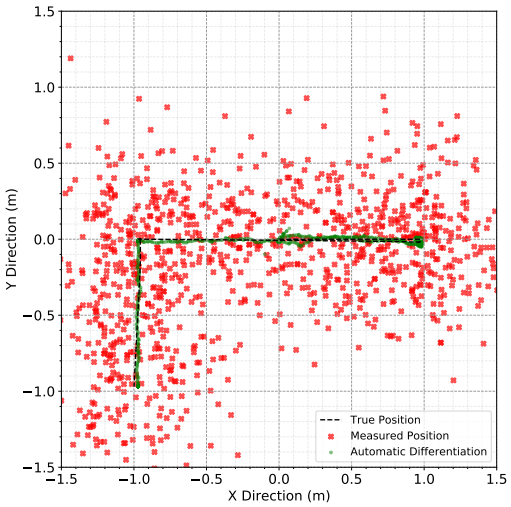
We instructed the measurement model to use the IMU and simulated localization information from laser scanners of the robot. This information is treated as noisy information on the robot's position on the XY plane, with variance $0.1\,m^2$ for both directions. We simulated a stroll of the robot on the Gazebo (Koenig & Howard, 2004) simulation platform and recorded the sensor measurements trace (Kaparounakis κ. συν., 2020).

Figure 6.4: The TurtleBot3 Burger differential drive robot. The robot offers an open-source simulation environment with ROS packages and Gazebo support.

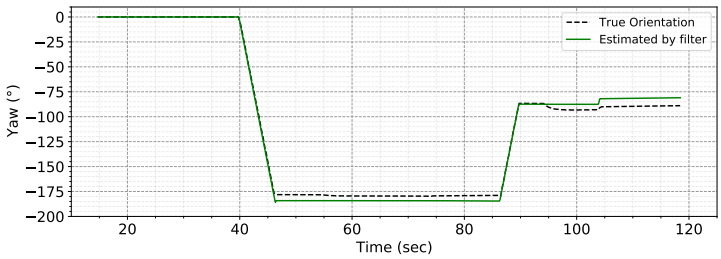
We utilized the recorded trace to evaluate the effectiveness of our generated Kalman filter with Automatic Differentiation in predicting the state vector $< x_t, y_t, \theta_t >$ of the robot. Figure 6.5 presents the estimation results for the position (Figure 6.5(a)) and yaw (Figure 6.5(b)) of the robot, both represented using green lines. The recorded measurements from the sensor of the robot are annotated in Figure 6.5(a) with red points. The robot started its stroll at position $(0, 0)$ facing towards the positive side of the x axis. We instructed it to to move straight to position $(1, 0)$, turn 180° right, continue to $(-1, 0)$, turn 90° left and finish its movement at position $(-1, -1)$. The dashed black line in Figure 6.5(a) corresponds to the actual trajectory of the robot and we can observe the high estimation accuracy of our generated Kalman filter, despite the noise in the measured values. The average Euclidean error of the translational movement of the robot is $0.0185\,m$ whereas the average error of the rotational movement (Figure 6.5(b)) is 4.72°.

6.3 Code Size & Performance Evaluation

Additionally to evaluating the accuracy of the generated filters, it is also important to quantify their memory and computational requirements. Thus, we profiled the generated filters for state estimation of a pendulum without drag, as presented in Section 6.1. We choose RISC-V as the target reference 32-bit RISC CPU architecture and examine different extensions of the instruction set architecture (ISA)



(a) Actual path and path estimation of TurtleBot3.



(b) Rotational movement estimation of TurtleBot3 using a gyroscope.

Figure 6.5: Prediction of the path of the TurtleBot3 Burger robot. It starts at position $(0, 0)$ facing the positive values of x axis. It moves straight to $(1, 0)$, turns 180° left, continues to $(-1, 0)$, turns 90° left and finally moves to $(-1, -1)$ as shown in Sub-figure (a). Sub-figure (b) shows the actual and estimated value of the yaw of the robot using a generated Kalman filter.

to take into account embedded systems of different computational competency. Starting from the RV32I base integer ISA, which provides only integer addition/-logical operations, we also examine RV32IM extension with integer multiplication and division instructions and RV32IMF/RV32IMFD extensions for single/double precision floating-point arithmetic operations.

Summarizing, the examined 32-bit ISA extensions are:

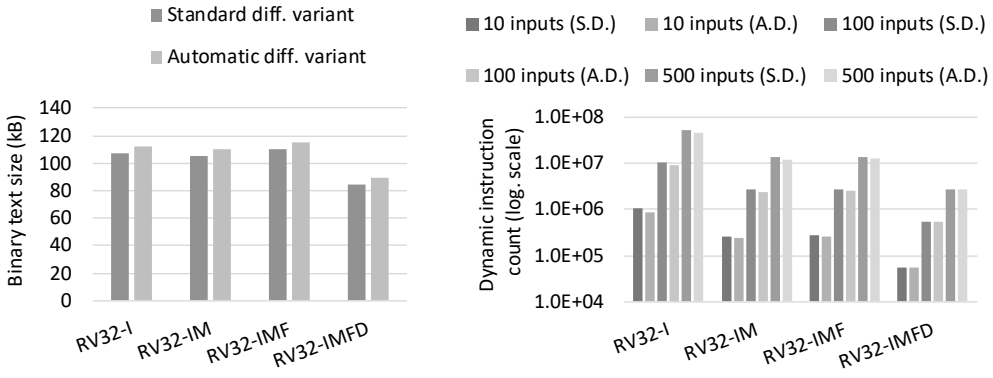
① **RV32I**: The base integer ISA of RISC-V including only integer additions/-subtraction/logical operations.
② **RV32IM**: Extension of the above to include integer multiplication and division instructions.
③ **RV32IMF**: Extension of the above for single precision floating-point arithmetic operations.
④ **RV32IMFD**: Extension of the above for double precision floating-point arithmetic operations.

The generated Kalman filter source code contains double precision variables and has been compiled using GCC 8.2.0 for RISC-V that produces binaries of the `riscv32-elf` architecture. We examined both variants of standard and Automatic Differentiation and benchmarked the generated filters for an increasing amount of input workload by varying the amount of total sensor inputs that are being processed for estimation of the state (Kaparounakis κ. συν., 2020). For this experiment, we examine inputs of 10, 100 and 500 sensor measurements of the angular rate ω of the pendulum.

Both filters were compiled with the `-O3` optimization option and we evaluated the binary size of the generated Kalman filters using the `riscv32-elf-size` tool. We additionally evaluated the dynamic instruction count for the execution of the Kalman filters using Sunflower (Stanley-Marbell & Marculescu, 2007), an open-source embedded system micro-architectural emulator. Sunflower emulates processor cores faithfully enough to run gcc-compiled code. The input sensor samples have been hardcoded in the source in order to decouple our measurements from the required instructions for disk I/O.

The size of the `text` segment of the filter binaries, for the different ISA extensions is illustrated in Figure 6.6(a). The dynamic instructions count for the completion of the filters' execution for varying input size is presented in Figure 6.6(b). Standard and Automatic Differentiation filter variants are annotated using "S.D." and "A.D", respectively. Each bar of Figure 6.6(b) corresponds to a different workload, i.e. different total number of input sensor vectors processed by the filter.

We observe that in all cases the filter variant with Automatic Differentiation is slightly larger in size but requires less instructions for its completion (note that the Y-axis scale of Figure 6.6(b) is logarithmic). Automatic differentiation results in an average gain of more than 7.5% in the required instructions for the filter execution, exceeding 16% at its maximum in the case of RV32I ISA. The tradeoff is an average increase of 4.7% in the `text` segment of the compiled binary (Kaparounakis κ. συν., 2020).



(a) Binary text size in kB. **(b)** Dyn. instruction count (log scale).

Figure 6.6: Profiling of the compiled binaries of the generated filters' source code for various RISC-V ISA extensions. Sub-figure (a) shows the size of the `text` segment of the binaries, showing a slight increase for filters with Automatic Differentiation. Sub-figure (b) shows the required instructions for the processing of input sensor vectors, where the filters with Automatic Differentiation achieve significant reduction compared to standard differentiation.

We attribute the increase in `text` size to (i) the expansions of the Kalman filter equations to their SSA form and (ii) the inclusion of the SSA form of the Reverse Mode A.D. (Section 5.4). The reduction of the dynamic instructions is attributed to the usage of Automatic Differentiation, which calculates the partial derivatives in the rows of the Jacobian matrices in a single function evaluation (Section 3.3). Conversely, standard differentiation needs to evaluate the required functions two times more for each partial derivative of a row (Kaparounakis κ. συν., 2020).

Chapter 7

Conclusions

This work presents an advance in the state of the art in automated synthesis of state estimation and sensor fusion algorithms. The method we present starts from a specification of the physics of an embedded sensor-driven system and its environment. It generates, as output, C code with small code and memory footprint, suitable for deployment of ultra-low-power microcontrollers. The automation of the method by its implementation within a compiler for a physics specification language reduces the time-consuming and potentially error-prone process of designing and updating state estimation algorithms such as linear and extended Kalman filters (Kaparounakis κ. συν., 2020).

The method we present can be easily extended to support other state estimation algorithms, such as the unscented Kalman filter and the particle filter, or different approaches to filter subcomponents, such as using an information matrix instead of a covariance matrix (Terejanu, 2013), enabling rapid comparison between multiple filter variants for the same embedded system solution. Additionally, the implementation of the method within a compiler for a physical system specification language makes it possible to add more static compile-time analysis for the output system, optimizing multiple parts of the resulting code in terms of efficiency, complexity and code size as well as providing this information to the user before deployment (Kaparounakis κ. συν., 2020). Lastly, the system we propose offers flexibility in terms of the needed linear-algebraic library functions, by making no concrete assumptions about them and enabling static or dynamic linking with the desired implementation.

Our implementation exploits advances in Reverse-Mode Automatic Differentiation of program sequences, which has recently seen great adaptation in the world of machine learning, to automate the generation of the partial derivatives of the state equation for the Jacobian matrices for the extended Kalman filter. Using descriptions of physical systems of a range of complexities, we evaluate and validate the generated filters in terms of accuracy, stability, convergence, and run-time requirements for deployment in resource-constrained environments (Kaparounakis κ. συν., 2020). The A.D. variant of the filter was verified to work within the

same error margins as conventional methods, while accuracy calculating the partial derivative matrices of the filter functions.

Chapter 8

Future Work

We have provided an system automates the task of implementing a linear or extended Kalman filter by providing a physical specification of the target embedded system. There are many interesting capabilities to be explored now that we have the ability to rapidly generate state-estimation algorithm implementations. Our implementation can serve as a basis for this exploration.

Below follow various interesting capabilities that can be explored using the proposed system.

8.1 System identification

A notable matter when implementing a Kalman filter is the choice of the noise covariance matrices for the process ($\mathbf{Q}$) and the measurement ($\mathbf{R}$). Options concerning this were briefly mentioned in Chapter 2 to be either to rely on the nominal characteristics of the sensors and a theoretical derivation of the process's noise or try to detect the correct noise value by examining real data from the system. Using these data one can perform auto-covariance methods (Åkesson κ. συν., 2008; Odelson κ. συν., 2006) to estimate the noise covariance.

These methods essentially use another filter to perform the inference. Thus, it would be possible to also provide the option to generate Kalman filters with the aim of discovering system noise characteristics.

Apart from estimating the stochastic characteristics, it would be also natural to extend the system presented to estimate nonrandom characteristics of the model. By supposing a system model where a specific non-stochastic variable, that is not part of the state vector, is unknown, it is possible to reconstruct the filter to estimate this variable alongside the state. This subject is briefly touched upon in Chapter 8 of the work by (Chui κ. συν., 2017).

8.2 Differential Equations

Physical systems descriptions often begin from a differential equation, so, ideally, that's where an automated synthesis system, like the one presented in this work, would start from.

The interpretation of the differential equation as a system model would begin by transforming it into a transfer function. The transfer function can then be transliterated to a state-space representation for inferring the matrices needed by the state-estimation algorithms (Chapter 2). An abstract overview of the stages appears below.

$$\frac{d^3y}{dt^3} + a_1\frac{d^2y}{dt^2} + a_2\frac{dy}{dt} + a_3y = b_0\frac{d^2u}{dt^2} + b_1\frac{du}{dt} + b_2u$$

```
(derivative 3 y t) + a3*(derivative 2 y t) + a2*(derivative y t) + a1*y
== b0*(derivative 2 u t) + b1*(derivative u t) + b2*u
```

$$H(s) = \frac{Y(s)}{U(s)} = \frac{b_0s^2 + b_1s + b_2}{s^3 + a_1s^2 + a_2s + a_3}$$

```
F = [[  0,  1,  0],
     [  0,  0,  1],
     [-a3,-a2,-a1]]

B = [[0],
     [0],
     [1]]

H = [[b2,b1,b0]]

D = [[0]]
```

A system capable to carry these operations through would require heavier support for calculus. This arises from the underlying usage of the Laplace Transform, factorization and the symbolical solution the equation.

It should be noted here that doesn't seem to be a concrete and formal method for transitioning from multiple differential equations to a transfer function. An idea would be to to process each one individually, as mentioned, and merge the

resulting state-space representation together at the end. Otherwise, this would be restricted to one differential equation only.

8.3 Partial-Run Optimizations

The Kalman filter's Update step, first presented in Section 2.3.2, operates on a vector of measurements. This creates the following complications. One could wait for the whole measurement vector to be filled before executing the Update step, but this would imply some synchronization between the different sensor readings; a scheme too elaborate for many low-cost cyber-physical systems. Another option would be to run the Update step whenever a new measurement becomes available, thus eliminating the need for synchronization. This would result in operations, on the parts of the measurement vector that have not changed, to be repeated.

The typical solution for this problem in the industry is to split the Update step into multiple sub-steps, one for each measurement variable. The Update step then operates only on the new measurement and performs only the operations of whose results are affected by this new information.

The proposed system could be enhanced to deduce which measurement variable changes propagate to which state variable in an automated manner. This information would then be used in the generation of multiple Update step substeps that would be used like the original Update step, but would imply to run the equations "partially", as in, only perform the operations that are affected by the new measurements.

Work by Mayhew D. extends (Mayhew, 1999) the formal formulation of the filters to account for multi-rate sensor fusion. Taking a holistic approach it describes the steps needed to implement a sensor-fusion algorithm for various target systems, inspiring further work in the specific case of IMU and GPU measurements (Caron, Duflos, Pomorski, & Vanheeghe, 2006). These could serve as a good starting point for implementing automated solutions as extensions of the work presented in this thesis.

8.4 Information Filter

An alternative variant form of the Kalman filter is the Information filter (Khan, 2005; De Jong κ. συν., 1991). This filter uses an information matrix to encapsulate the uncertainty of our estimations, as opposed to using a covariance matrix. This change leverages added complexity in the propagation of the covariance during the Predict step in order to decrease computational complexity of the Update step.

The prior and the posterior information matrices and vectors are defined in Equations 8.1 and 8.2, respectively. The measurement covariance matrix and measurement vector are then defined in Equations 8.3 and 8.4, respectively.

$$\mathbf{Y}_k = \mathbf{P}_k^{-1} \tag{8.1}$$
$$\hat{\mathbf{y}}_k = \mathbf{P}_k^{-1}\hat{\mathbf{x}}_k \tag{8.2}$$

$$\mathbf{I}_k = \mathbf{H}_k^{\mathsf{T}}\mathbf{R}^{-1}\mathbf{H}_k \tag{8.3}$$
$$\mathbf{i}_k = \mathbf{H}_k^{\mathsf{T}}\mathbf{R}^{-1}\mathbf{z}_k \tag{8.4}$$

In the Information filter the Update step now becomes a simple addition of information matrices.

$$\mathbf{Y}_k = \mathbf{Y}_k^{-} + \mathbf{I}_k \tag{8.5}$$
$$\hat{\mathbf{y}}_k = \hat{\mathbf{y}}_k^{-} + \mathbf{i}_k \tag{8.6}$$

The advantage of this approach is that a great number measurements be filtered by summation of their information matrices and vectors (Khan, 2005).

Bibliography

Abadi, M., & Plotkin, G. D. (2019). A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, *4*(POPL), 1–28.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques*. Greg Tobin.

Åkesson, B. M., Jørgensen, J. B., Poulsen, N. K., & Jørgensen, S. B. (2008). A generalized autocovariance least-squares method for kalman filter tuning. *Journal of Process control*, *18*(7-8), 769–779.

Alpern, B., Wegman, M. N., & Zadeck, F. K. (1988). Detecting equality of variables in programs. Στο *Proceedings of the 15th acm sigplan-sigact symposium on principles of programming languages* (σελ. 1–11).

Amsters, R., & Slaets, P. (2019). Turtlebot 3 as a robotics education platform. Στο *International conference on robotics and education rie 2017* (σελ. 170–181).

Arulampalam, M. S., Maskell, S., Gordon, N., & Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on signal processing*, *50*(2), 174–188.

Barfoot, T. D. (2017). *State estimation for robotics*. Cambridge University Press.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2017). Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, *18*(1), 5595–5637.

Braun, M., Buchwald, S., Hack, S., Leißa, R., Mallon, C., & Zwinkau, A. (2013). Simple and efficient construction of static single assignment form. Στο *International conference on compiler construction* (σελ. 102–122).

Breuleux, O., & van Merriënboer, B. (2017). Automatic differentiation in myia.

Caron, F., Duflos, E., Pomorski, D., & Vanheeghe, P. (2006). Gps/imu data fusion using multisensor kalman filtering: introduction of contextual aspects. *Information fusion*, *7*(2), 221–230.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., ... Riddell, A. (2017, 1). Stan : A probabilistic programming language. *Journal of Statistical Software*, *76*(1). doi: 10.18637/jss.v076.i01

Chui, C. K., Chen, G., κ. συν. (2017). *Kalman filtering*. Springer.

De Jong, P., κ. συν. (1991). The diffuse kalman filter. *The Annals of Statistics*, *19*(2), 1073–1083.

Dudek, G., & Jenkin, M. (2010). *Computational principles of mobile robotics*. Cambridge university press.

Edwards, B. C. (2000). Design and deployment of a space elevator. *Acta Astronautica*, *47*(10), 735–744.

Giernacki, W., Skwierczyński, M., Witwicki, W., Wroński, P., & Kozierski, P. (2017). Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. Στο *2017 22nd international conference on methods and models in automation and robotics (mmar)* (σελ. 37–42).

Grewal, M. S., & Andrews, A. P. (2014). *Kalman filtering: Theory and practice with matlab*. John Wiley & Sons.

Griewank, A. (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, *1*(1), 35–54.

Griewank, A., κ. συν. (1989). On automatic differentiation. *Mathematical Programming: recent developments and applications*, *6*(6), 83–107.

Griewank, A., & Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation* (τ. 105). Siam.

Guennebaud, G., Jacob, B., κ. συν. (2010). Eigen. *URl: http://eigen. tuxfamily. org*.

Haykin, S., & Arasaratnam, I. (2009). Cubature kalman filters. *IEEE Trans. Autom. Control*, *54*(6), 1254–1269.

Homescu, C. (2011). Adjoints and automatic (algorithmic) differentiation in computational finance. *Available at SSRN 1828503*.

Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., & Durand, F. (2019). Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*.

Innes, M., Edelman, A., Fischer, K., Rackauckus, C., Saba, E., Shah, V. B., & Tebbutt, W. (2019). Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*.

Jazwinski, A. H. (1969). Adaptive filtering. *Automatica*, *5*(4), 475–485.

Julier, S. J., & Uhlmann, J. K. (2004). Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, *92*(3), 401–422.

Kalman, R. (1960). E. 1960. a new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, *82*, 35–45.

Kalman, R. E., & Bucy, R. S. (1961). New results in linear filtering and prediction theory.

Kaparounakis, O., Tsoutsouras, V., Soudris, D., & Stanley-Marbell, P. (2020). Automated physics-derived code generation for sensor fusion and state estimation. *arXiv preprint arXiv:2004.13873*.

Khan, M. E. (2005). Matrix inversion lemma and information filter. *Honeywell Techonology Solutions Lab, Bangalore, India*.

Koenig, N., & Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. Στο *2004 ieee/rsj international conference on intelligent robots and systems (iros)(ieee cat. no. 04ch37566)* (τ. 3, σελ. 2149–2154).

Labbe, R. (2015). Kalman and bayesian filters in python.

Lewis, P. M., & Stearns, R. E. (1968). Syntax-directed transduction. *Journal of the ACM (JACM)*, *15*(3), 465–488.

Lim, J., κ. συν. (2017). *Newton: a language for describing physics* (Αδημοσίευτη διδακτορική διατριβή). Massachusetts Institute of Technology.

Lim, J., & Stanley-Marbell, P. (2018). Newton: A language for describing physics. *arXiv preprint arXiv:1811.04626*.

Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *9*(4), e1305.

Massalin, H. (1987). Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, *15*(5), 122–126.

Mayhew, D. M. (1999). *Multi-rate sensor fusion for gps navigation using kalman filtering* (Αδημοσίευτη διδακτορική διατριβή). Virginia Tech.

McElhoe, B. A. (1966). An assessment of the navigation and course corrections for a manned flyby of mars or venus. *IEEE Transactions on Aerospace and Electronic Systems*(4), 613–623.

Mohazzabi, P., Shankar, S. P., κ. συν. (2017). Damping of a simple pendulum due to drag on its string. *Journal of Applied Mathematics and Physics*, *5*(01), 122.

Mueller, M. W., Hamer, M., & D'Andrea, R. (2015, May). Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation. Στο *2015 ieee international conference on robotics and automation (icra)* (σελ. 1730-1736). doi: 10.1109/ICRA.2015.7139421

Mueller, M. W., Hehn, M., & D'Andrea, R. (2016). Covariance correction

step for kalman filtering with an attitude. *Journal of Guidance, Control, and Dynamics*, 1–7.

Odelson, B. J., Rajamani, M. R., & Rawlings, J. B. (2006). A new autocovariance least-squares method for estimating noise covariances. *Automatica*, *42*(2), 303–308.

Oppenheim, A. V., & Schafer, R. W. (1983). Digital signal processing. 1975. *Englewood Cliffs, New York*.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., ... Lerer, A. (2017). Automatic differentiation in pytorch.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., ... Ng, A. Y. (2009). Ros: an open-source robot operating system. Στο *Icra workshop on open source software* (τ. 3, σελ. 5).

Richardson, J., & Wilson, E. (2006). Flexible generation of kalman filter code. Στο *2006 ieee aerospace conference* (σελ. 8–pp).

Särkkä, S. (2013). *Bayesian filtering and smoothing* (τ. 3). Cambridge University Press.

Smith, G. L., Schmidt, S. F., & McGee, L. A. (1962). *Application of statistical filter theory to the optimal estimation of position and velocity on board a circumlunar vehicle*. National Aeronautics and Space Administration.

Sorenson, H. W. (1970). Least-squares estimation: from gauss to kalman. *IEEE spectrum*, *7*(7), 63–68.

Sra, S., Nowozin, S., & Wright, S. J. (2012). *Optimization for machine learning*. Mit Press.

Stanley-Marbell, P., & Marculescu, D. (2007). Sunflower: Full-system, embedded microarchitecture evaluation. Στο *International conference on high-performance embedded architectures and compilers* (σελ. 168–182).

Talagrand, O., & Courtier, P. (1987). Variational assimilation of meteorological observations with the adjoint vorticity equation. i: Theory. *Quarterly Journal of the Royal Meteorological Society*, *113*(478), 1311–1328.

Terejanu, G. A. (2013). Discrete kalman filter tutorial. *University at Buffalo, Department of Computer Science and Engineering, NY*, *14260*.

Tsoutsouras, V., Vigdorchik, M., & Stanley-Marbell, P. (2020). Synthesizing compact hardware for accelerating inference from physical signals in sensors. *arXiv preprint arXiv:2002.01241*.

Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., ... Wunsch, C. (2008). Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Transactions on Mathematical Software (TOMS)*, *34*(4), 1–36.

Wan, E. A., & Van Der Merwe, R. (2000). The unscented kalman filter for

nonlinear estimation. Στο *Proceedings of the ieee 2000 adaptive systems for signal processing, communications, and control symposium (cat. no. 00ex373)* (σελ. 153–158).

Wang, Y., Willis, S., Tsoutsouras, V., & Stanley-Marbell, P. (2019). Deriving equations from sensor data using dimensional function synthesis. *ACM Transactions on Embedded Computing Systems (TECS)*, *18*(5s), 1–22.

Whittle, J., & Schumann, J. (2004). Automating the implementation of kalman filter algorithms. *ACM Transactions on Mathematical Software (TOMS)*, *30*(4), 434–453.

Widrow, B., & Lehr, M. A. (1990). 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, *78*(9), 1415–1442.