

# Accelerating Embedded HOWFSC Algorithms

by

Brandon Eickert

B.S./B.A., Electrical Engineering and Physics, University of San Diego (2023)

Submitted to the Department of Aeronautics and Astronautics  
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2025

© 2025 Brandon Eickert. This work is licensed under a [CC BY-NC-SA 4.0](#) license.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Brandon Eickert  
Department of Aeronautics and Astronautics  
May 16, 2025

Certified by: Kerri L. Cahoy  
Sheila Evans Widnall Professor of Aeronautics and Astronautics, Thesis Supervisor

Accepted by: Jonathan P. How  
Richard Cockburn Maclaurin Professor in Aeronautics and Astronautics  
Chair, Graduate Program Committee



# Accelerating Embedded HOWFSC Algorithms

by

Brandon Eickert

Submitted to the Department of Aeronautics and Astronautics  
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS

## ABSTRACT

The quest to directly image planets of other solar systems demands not only state-of-the-art coronagraphs, but also places extreme performance demands on space-based processors. Direct imaging requires precise wavefront control to acquire the  $10^{10}$  contrast necessary to reveal a dim, Earth-like exoplanet. This precise level of control is only possible if high-order wavefront sensing and control (HOWFSC) algorithms are executed with enough speed to offset wavefront error accumulation. Of the many aspects that make high-contrast imaging difficult, a central bottleneck is the speed at which we can run these algorithms. At the center of this work, we aim to accelerate the execution of two foundational HOWFSC algorithms: optical modeling and Electric Field Conjugation (EFC). Optical modeling underpins both Jacobian-based EFC, and a relatively new variant of EFC, called adjoint-based EFC.

The two main contributions of this thesis are to port bottleneck HOWFSC algorithms to the relevant computing environments, and quantify speedups attained by both algorithm choice and implementation optimization. This work explores the acceleration of optical modeling for a vector vortex coronagraph through the use of the FFTW library, and the acceleration of EFC by implementing adjoint-based EFC in an embedded context. We utilize functional analogs to radiation-hardened processors, using the NXP T1040 in place of the BAE RAD5545, and the NXP LS1046 in place of the LS1046-Space. We find that the FFTW library enabled a factor of six speedup for  $4096 \times 4096$  fast Fourier transforms (FFTs), and a factor of five for  $2048 \times 2048$  FFTs. With these significant speedups, the bottleneck within the vortex operations of the optical model shifts from the FFT to matrix multiplication. We additionally time the execution of the underlying routines of Jacobian-based EFC and AD-EFC to estimate that AD-EFC is 46 times faster than Jacobian-based EFC. Despite these speedups, AD-EFC is still a factor of 124 away from 100-second latency for our specific optical model. These results demonstrate that one to two orders of magnitude of speedup must be attained by either further optimizing algorithm implementations, or exploring other parallelization strategies, computing architectures, and mission paradigms to achieve a latency on the order of 100 seconds.

Thesis supervisor: Kerri L. Cahoy

Title: Sheila Evans Widnall Professor of Aeronautics and Astronautics



# Acknowledgments

First and foremost, I would like to thank my advisor and mentor Prof. Kerri Cahoy, without whom this work would not have been possible. Her continued support since I arrived at MIT just two years ago has been essential in preparing me in a personally uncharted research domain.

I would also like to thank my colleague and mentor Nicholas Belsten, who has provided indispensable support to boost my technical understanding of the intersection between computing and wavefront control. He kindly offered up many hours, helped fill in gaps in my knowledge, and provided a great example in performing quality research.

My sincerest gratitude goes to my family – my mom, brother, sister-in-law, and stepdad – for their unwavering support and encouragement throughout this journey. Their belief in my abilities has sustained me through countless psets, hours in the lab, and many ponderings in the space of ideas. This work would not have been possible without their constant presence as my anchor and inspiration.

It has been a privilege to join the MIT community these past two years. The intellectual enthusiasm here, both within STAR Lab and across campus, has fueled countless insightful discussions. I am grateful for the collaborative spirit that has enriched this work and inspired me daily. Special thanks to my colleagues Leo Gallo, Adam Bahlous-Boldi, Sai Manojkumar, Paige Forester, Joachim Bron, Andy Eskenazi, Álvaro Martínez-Sánchez, Emma Shafer, Ajay Gill, Paul Serra, and many others for their friendship and support.

I would also like to thank NASA for supporting this work under the NASA Astrophysics Technology Division under APRA grant #80NSSC22K1412.



# Contents

<i>List of Figures</i>	9
<i>List of Tables</i>	11
<b>1 Introduction</b>	<b>13</b>
1.1 The Search For Life Beyond Our Solar System . . . . .	13
1.2 The Computational Bottleneck . . . . .	14
1.3 Computing Platform Selection . . . . .	18
1.3.1 Functional Equivalent Processors . . . . .	19
1.4 The Focus of this Thesis . . . . .	19
<b>2 HOWFSC Algorithm Overview</b>	<b>23</b>
2.1 Electric Field Conjugation (EFC) . . . . .	25
2.2 Optical Modeling . . . . .	28
2.3 Adjoint-Based EFC (AD-EFC) . . . . .	31
2.3.1 Selecting an Optimizer . . . . .	33
2.3.2 Computing $J_{\text{AD-EFC}}$ . . . . .	34
2.3.3 Computing $\frac{dJ_{\text{AD-EFC}}}{d\delta\mathbf{a}}$ with Automatic Differentiation . . . . .	36
2.3.4 Iterative Minimization . . . . .	37
<b>3 Implementation of HOWFSC Algorithms</b>	<b>41</b>
3.1 Processor Setup . . . . .	41
3.2 Software Considerations . . . . .	44

3.2.1	Selection of Programming Language . . . . .	45
3.2.2	Incorporating Source Code . . . . .	49
3.2.3	Third Party Libraries for Speed-Up . . . . .	51
3.3	Implementing Adjoint-Based EFC . . . . .	56
3.4	Flavors of EFC . . . . .	65
<b>4</b>	<b>Key Experiments</b>	<b>71</b>
4.1	FFTW . . . . .	72
4.1.1	Isolated FFTs . . . . .	72
4.1.2	FFTs in the Optical Model . . . . .	74
4.2	Adjoint-Based EFC in C . . . . .	78
<b>5</b>	<b>Conclusions and Future Work</b>	<b>85</b>
<b>A</b>	<b>Automatic Differentiation</b>	<b>89</b>
	<i>References</i>	91

# List of Figures

1.1	Moore’s Law trend for computing in space vs. terrestrial computing platforms. Adapted from Belsten et al. [7] and Keys et al. [8]. . . . .	17
2.1	Wavefront sensing and control loop on a space telescope, adapted from Belsten et al. [6]. . . . .	23
2.2	Computational flow of optical modeling with a vector vortex coronagraph. The wavefront evolves via angular spectrum propagation (ASP) between DMs, and via FFTs and MFTs from the pupil to the vortex coronagraph, then to the Lyot stop, and finally, to the science camera. . . . .	29
2.3	The forward DM model transforms DM actuator commands $\mathbf{a}$ via an MFT, an elementwise product with the influence function $\hat{\mathbf{F}}$ , an IFFT to obtain the surface $\mathbf{S}_{\text{DM}}$ , and then elementwise exponentiation to obtain a phasor representation. The forward vortex model transforms the electric field at the pupil plane $\mathbf{E}_{\text{PUP}}$ into the electric field at the Lyot plane $\mathbf{E}_{\text{LP}}$ through the utilization of vortex low resolution (LR) and high resolution (HR) masks $\mathbf{V}$ and window functions $\mathbf{W}$ . [1] is used to denote a matrix of all one’s. . . . .	31
3.1	Important commands from the host setup script for the P5040 processor. . .	43
3.2	Building the FFTW library for the T1040 processor. . . . .	52
3.3	How the Numerical Recipes <code>dvector</code> data type stores complex matrices in contiguous memory with a <code>double</code> at each entry. . . . .	53

3.4	Converting from NR's <code>dfourn()</code> call to FFTW. . . . .	54
3.5	Building the LibLBFGS library for the T1040 processor. . . . .	56
3.6	Data sent from the full Fresnel model in Python to the embedded processor. . . . .	58
3.7	The <code>evaluate_context_t</code> custom structure in C. . . . .	60
3.8	The <code>forward()</code> model definition header in C. . . . .	62
3.9	The first few adjoint variables, <code>dJ_dE_DMs</code> , and <code>dJ_dE_LS</code> . . . . .	65
3.10	The log amplitude and phase of example adjoint variables <code>dJ_dE_DMs</code> , and <code>dJ_dE_LS</code> . . . . .	66
3.11	Transcription of conjugating complex matrices from Python to C. . . . .	67
4.1	Average run times of <code>dfourn()</code> vs. FFTW plan + execute on the T1040 processor for complex-valued matrices of different sizes. . . . .	74
4.2	Average run times of <code>dfourn()</code> vs. FFTW plan + execute on the LS1046 processor for complex-valued matrices of different sizes. . . . .	75
4.3	Visualization of the run time required to execute the <code>forward()</code> optical model with all calls either to <code>dfourn()</code> or FFTW compared across three processors. . . . .	77
4.4	Execution-time comparison of several EFC flavors on a desktop workstation. . . . .	80
4.5	Execution-time comparison of several EFC flavors on the LS1046. . . . .	81
A.1	Computational graph for $\mathbf{y} = \mathbf{A}\mathbf{x}_1 + \mathbf{x}_2$ . . . . .	89

# List of Tables

2.1	Summary of HOWFSC algorithms, adapted from Pogorelyuk et al. [5]. . . . .	24
2.2	The <code>forward()</code> optical model decomposed into the operations that evolve the wavefront from the entrance pupil to the final focal plane. AP is the electric field at the input aperture, WFE is the representative wavefront error at the input aperture, FDM is the forward DM model, and FV is the forward vortex coronagraph model. $z_{1,2}$ denotes the propagation distance between DM1 and DM2, and $\odot$ denotes elementwise matrix multiplication, . . . . .	30
2.3	Optical modeling parameters . . . . .	32
2.4	Hessian sizes corresponding to different DM sizes. $\delta\mathbf{a} \in \mathbb{R}^{N_{\text{acts}}}$ , where $N_{\text{acts}}$ is the number of unmasked DM actuators in two DMs sized by the left-most column. . . . .	34
2.5	Forward vs. adjoint model primitives, adapted from Will et al. [22] . . . . .	37
2.6	Adjoint vortex model operations, adapted from Milani et al. [34]. All variable definitions are provided in Table 2.3b. . . . .	38
3.1	Available Compilers by Processor . . . . .	44
3.2	Forward vs. Adjoint Calls . . . . .	64
4.1	Execution times of <code>dfourn()</code> vs. FFTW across three processors. The final digit is uncertain in each reported time. . . . .	73

4.2	Time it takes to execute the <code>forward()</code> optical model with all calls either to <code>dfourn()</code> or FFTW on three different processors. . . . .	76
4.3	Runtimes for key EFC sub-routines on the x86_64 i7 desktop workstation and the LS1046 embedded processor. . . . .	80

# Chapter 1

## Introduction

### 1.1 The Search For Life Beyond Our Solar System

Is life on Earth an exception or the norm? To progress in answering this question is to better understand humanity's relationship with the universe. The presence of life on Earth causes detectable changes in Earth's atmosphere, in the form of water, oxygen, methane, carbon dioxide, and other biosignatures. Thus, Earth provides the prototype for searching for life on other planets.

To date, most documented exoplanets have been discovered via the transit method. The transit method, reliant on the orbiting planet passing in front of its host star, is primarily suited for finding planets close to their host stars, like red M dwarfs [1]. The transit method is particularly effective for discovering large exoplanets around dim stars, since these planets block more light during transit. While the transit method has significantly increased the number of known exoplanets, the star-planet systems it tends to discover are not like our system, where a rocky planet is at an optimal distance from its host star to permit liquid water. The spherical shell around a host star that enables the formation of liquid water, and thus, life, is known as the habitable zone.

With a goal of finding Earth-like exoplanets orbiting Sun-like stars, the transit method is

less useful, as the perceived reduction in brightness of an Earth around a Sun is only about a few percent or less during transit [2]. The M dwarf systems that transit works well with may not be the best candidate for habitability, because the host stars can have large super-flares and highly energetic emissions that are hostile to life-formation.

Direct imaging is a different approach with promise for detecting Earth-like exoplanets in the habitable zone around its host star. A direct image, which supports the use of spectroscopy, can measure a reflected light spectrum of an exoplanet, which can contain information about its composition, clouds, thermal structure, and variability. A concept study for NASA's next flagship space telescope, the Habitable Worlds Observatory (HWO), is being designed for spectroscopy of Earth-like exoplanets, searching for biosignatures [1]. However, the physical realization of such a space-based direct imaging system will require progress in critical technologies.

## 1.2 The Computational Bottleneck

The primary difficulty associated with directly imaging Earth-like exoplanets is that they are typically  $10^{10}$  times dimmer than their host star [3]. In order to perform high-contrast imaging, we need extremely precise control of the wavefront propagating onto the telescope's detector. Thermal deformations of the telescope and other dynamic sources of disturbance must be actively corrected at a rate of seconds to minutes to maintain high contrast. High spatial frequency corrections to the wavefront can be made through the use of large actuator count deformable mirrors (DMs), which can control for high frequency modes that would otherwise cause diffracted light from the host star to fall into the region of interest, causing a bright speckle that would dominate a faint exoplanet. This can be done with the use of an adaptive optics system, which can actively control wavefront errors to achieve a desired outcome. Image-plane wavefront sensing, in conjunction with high order wavefront sensing and control (HOWFSC) algorithms, can be used to generate the optimal actuator commands

to cancel out undesired high order wavefront errors.

In order for HWO to image high-contrast exoplanets, the wavefront control loop needs to be closed in seconds to minutes, depending on the rate of wavefront error evolution. If the algorithms required to correct the aberrated wavefront take too long to execute, then wavefront drifts cause the region of interest in the image plane (the “dark hole”) to be eroded. As will be discussed in Chapter 2, current state-of-the-art HOWFSC algorithms require both the construction and handling of the Jacobian matrix  $\mathbf{G} \in \mathbb{C}^{m \times n}$ , where  $m = (\# \text{ pixels in the dark hole} \cdot \# \text{ wavelengths})$ , and  $n = \# \text{ actuators}$ . The Electric Field Conjugation (EFC) algorithm [4] requires the Jacobian, which parameterizes the linearized relationship between DM actuator states and pixels at the science camera. Previous work has utilized the “easy” HabEx ( $m = 130,000$ ,  $n = 6,400$ ) and “difficult” LUVOIR ( $m = 510,000$ ,  $n = 26,000$ ) concept studies to estimate single-precision Jacobian sizes between 3 and 50 GB<sup>1</sup>[5]. Unfortunately, it is unlikely that a Jacobian of this size could even fit in the memory of a radiation-hardened processor for a deep-space mission like HWO. Without even considering the complexities of computer architecture, deploying classical Jacobian-based wavefront control methods on HWO may simply be bottlenecked by memory requirements.

Beyond memory usage, previous work combined complexity analyses of HOWFSC algorithms with notional latency requirements to produce estimates of the required processor throughput [6]. A full treatment of computationally intense HOWFSC algorithms will be presented in Chapter 2, but the central operations have been included here to illustrate their difficulties. In Belsten et al. [6], it was estimated that Jacobian generation via optical modeling requires 540,000 giga floating-point operations (GFLOPs) in the easy case and 2,400,000 GFLOPs in the difficult case<sup>2</sup>. After generating a Jacobian, computing the matrix product  $\mathbf{G}^\dagger \mathbf{G}$  (a typical next step in generating a control solution) requires 11,000 GFLOPs

---

<sup>1</sup>Size =  $m \cdot n \cdot 4$  bytes.

<sup>2</sup>These FLOP counts are derived from a complexity model for Jacobian generation via optical modeling, which requires  $2n \cdot (112N^2 \log_2(N) + 4N^3 + 2N^2m^{1/2})$  operations. The easy case has  $m = 130,000$ ,  $n = 6,400$ , and  $N = \text{edge size of Fourier optics model} = 2,048$ . For the difficult case,  $m = 510,000$ ,  $n = 26,000$ , and  $N = 2,048$ .  $m$  and  $n$  are the same  $m$  and  $n$  that were defined for the Jacobian’s dimensions.

in the easy case, and 690,000 GFLOPs in the difficult case<sup>3</sup>.

Jacobian generation via optical modeling may only need to be done once during the dark hole creation phase, requiring only one computation of  $\mathbf{G}^\dagger\mathbf{G}$ , but more computations are needed to continue making progress in digging a dark hole. Digging a dark hole requires generating control solutions with different regularization parameters, which, in this work, is done through QR decomposition and QR solving techniques. Computing control solutions with QR decomposition and QR solve requires approximately 350 GFLOPs in the easy case, and 23,500 GFLOPs in the difficult case<sup>4</sup>. These numbers, coupled with notional latency requirements of 100 and 1 s for the easy and difficult cases, give a required throughput range of 350 giga floating-point operations per second (GFLOPS) and 23,500 GFLOPS to perform EFC with a new QR decomposition and QR solve. For Jacobian generation via optical modeling with a latency requirement of 1,200 s, the required processor performance balloons to 450 GLOPS and 2,000 GLOPS for the easy and difficult cases.

On terrestrial computing platforms, these throughput requirements are unremarkable, but for a processor designed for the space environment, this level of performance is likely infeasible. The radiation environment of deep space, coupled with the necessity to minimize the computing platform’s size, weight, and power (SWaP) footprint, results in constraints that severely limit processor performance.

Figure 1.1 demonstrates the approximate 10-15 year lag in space-optimized computing platforms vs. their terrestrial counterparts in instructions-per-second (IPS). Given the decade-level lifetime for a mission like HWO, the resilience to total ionizing dose (TID) by radiation-hardened processors provides reliability that is difficult to establish with commercial-off-the-shelf (COTS) components. The BAE RAD5545, the radiation-hardened successor to the flight-heritage BAE RAD750 processor, is marketed with a computational throughput of 3.7 GFLOPS [9]. This is not sufficient for the difficult-case algorithms.

Commercial space activity is largely in low-earth orbit (LEO), and for much shorter

---

<sup>3</sup>FLOP count derived from a complex matrix product, which requires  $2n^2m$  operations.

<sup>4</sup>FLOP count derived from  $\frac{4}{3}n^3$  operations for QR decomposition, and  $2mn$  operations for QR solve.

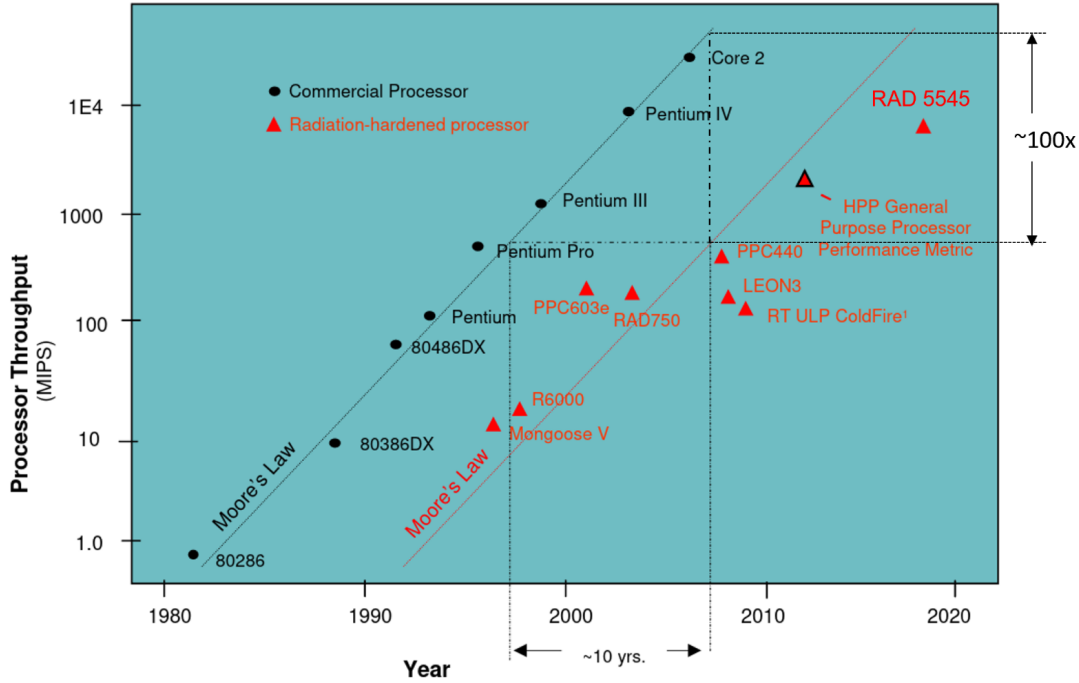


Figure 1.1: Moore’s Law trend for computing in space vs. terrestrial computing platforms. Adapted from Belsten et al. [7] and Keys et al. [8].

mission lifetimes than the decade(s) of operation expected for HWO. HWO is not planned for LEO, but rather scheduled to operate at Earth-Sun Lagrange point 2, which doesn’t provide the same level of radiation protection from the magnetosphere and Earth’s atmosphere. The milder radiation environments, coupled with short mission lifetimes and different risk postures, are the drivers behind the commercial use of COTS processors in space.

There are a variety of ways to address both memory and expected performance bottlenecks from a mission architect’s perspective. The memory and throughput limitations of available space processors prompted the soon-to-launch Nancy Grace Roman telescope to use a ground-in-the-loop scheme to execute the wavefront control algorithms needed for high-contrast imaging [10]. The choice of Roman’s designers to employ a ground-in-the-loop scheme, highlights the fact that onboard computers struggle to rival ground-in-the-loop, even considering a large amount of data link delay and without full ground station coverage. Ideally, we would have onboard processors capable of high-contrast imaging techniques without the

need for ground computation. Other mission architecture changes that could help address this bottleneck include in-space servicing and on-orbit replacement of a processor with less robustness to TID, but with greater performance. Another possibility includes deploying COTS devices that weren't explicitly designed for the space environment, but respond well to radiation tests, which could be combined with additional radiation shielding [11].

The goal of this work is to improve the performance of the most demanding HOWFSC algorithms implemented on embedded devices, and to highlight the speedups attainable. The first hurdle in performing this kind of analysis is the selection of the embedded processors for possible implementation.

### 1.3 Computing Platform Selection

In a detailed processor survey and requirements flowdown performed by Belsten et al., the most likely embedded processor candidates for a future space telescope were assembled [6]. The selection of a processor for this work is driven by large computational throughput, as well as sufficient robustness to TID. Maximizing both of these criteria makes the processor more likely to be flown on a Class A NASA mission. Heritage significantly impacts the selection, as an established processor family translates to a high technological readiness-level (TRL), effectively de-risking that portion of the mission design. While ASICs, FPGAs, and GPUs are worth considering for the execution of HOWFSC algorithms, this work focuses explicitly on reprogrammable CPU-based processors. The reason for this is that ASICs require a significant budget and lead times [12], FPGAs require a completely different development workflow than the focus here, and GPUs are still experimental when it comes to robustness to the radiation environment of space [13].

For past flagship deep-space NASA missions like the Mars Reconnaissance Orbiter, Curiosity and Perseverance, and the James Webb Space Telescope, the BAE RAD750 has been the primary processor of choice [14]. The BAE RAD750 has traditionally been utilized

for its reliability and ability to withstand very high doses of radiation that would likely disable other processors. The flight heritage of the BAE RAD750 makes its successor, the BAE RAD5545, one of the two processors under consideration in this work.

The other processor candidate under consideration is the Teledyne LS1046-Space. The RAD5545 uses a PowerPC architecture, whereas the LS1046-Space is equipped with ARM cores. The comparison between these two processors can provide some insight into the speedups attainable across different computing architectures.

### 1.3.1 Functional Equivalent Processors

For research use at a university, primarily financial considerations limit the possibility of implementing algorithms directly on these radiation-tolerant platforms. In this work, we utilize “functional equivalent” processors [15] that share similar performance, memory bandwidth, and processor architecture. The NXP T1040 was selected as the functional equivalent to the RAD5545 [16], and the LS1046A RDB was selected as the functional equivalent to the LS1046-Space [17]. Both functional equivalents use the same architecture as their space-based parents, and the LS1046A’s clock frequency, memory bandwidth, and L2 cache size match precisely, making it a well-suited substitute.

## 1.4 The Focus of this Thesis

With HWO’s highly interconnected web of dependencies between science capabilities (like size of the dark hole) and available technology (the speed at which we can run HOWFSC algorithms), the goal of this thesis is to evaluate and improve upon expected HOWFSC performance on realistic space-based processors. The primary contribution of this thesis is to provide a realistic performance delta between naive algorithm implementations and speedups attainable from both third-party libraries and alternate algorithm choices.

**The focus of this work is on how we can obtain first-order speedups in both**

**optical modeling and EFC on two specific embedded processors.** In optical modeling, the Fastest Fourier Transform in the West (FFTW) library is leveraged to achieve direct speedup over a naive FFT implementation. To obtain speedups in EFC, we explore adjoint-based EFC, a Jacobian-less counterpart to EFC that has the potential for large memory and throughput savings. To the knowledge of the author, this thesis provides a programming roadmap for the first implementation of adjoint-based EFC in an embedded context.

Through an implementation-based approach, this thesis attempts to provide insight to the designers and project architects of future space telescopes like the HWO. From a systems-designer perspective, we recognize that selecting a computing platform is only one dimension of a combinatoric explosion of the other decisions accompanied with a full technology stack: the operating system, the computing architecture, the programming language, the available third-party libraries, and compiler optimizations, to name a few. The implementation provides a story and insight into the other layers of the technology stack that operations-count-based analysis can overlook. This work will focus on the higher levels of the technology stack necessary to deploy HOWFSC algorithms, diving into the algorithm selection and software optimization available on two specific processors: the LS1046 and the T1040.

In Chapter 2, we outline the mathematical background required to understand the nuances of the algorithm implementations. We also ground the significance of adjoint-based EFC's preferability over classical Jacobian-less EFC. In Chapter 3, we will provide a detailed outline of the specifics of programming implementations of FFTW and the accompanying libraries required to perform adjoint-based EFC. We also offer a detailed transcription from an adjoint-based EFC implementation in Python to an implementation in C. In Chapter 4, we dive into the results of the speedups obtained from implementing FFTW over the naive `dfourn()` algorithm provided by Numerical Recipes in C. We also ground the speedups in the context of how much quicker the optical model can be run with FFTW calls instead of `dfourn()` calls. Additionally, we provide detailed estimates of how long adjoint-based EFC is expected to take on an embedded processor. Finally, in Chapter 5, we summarize our findings, as well

as provide some directions for future work.



# Chapter 2

## HOWFSC Algorithm Overview

In this chapter, we provide an overview of the critical high order wavefront sensing and control algorithms needed to perform high-contrast imaging. We begin by outlining the general high-contrast imaging procedure, and then outline the specifics of the *bottleneck* algorithms. The details of electric field conjugation are presented in Section 2.1, optical modeling in Section 2.2, and adjoint-based EFC in Section 2.3.

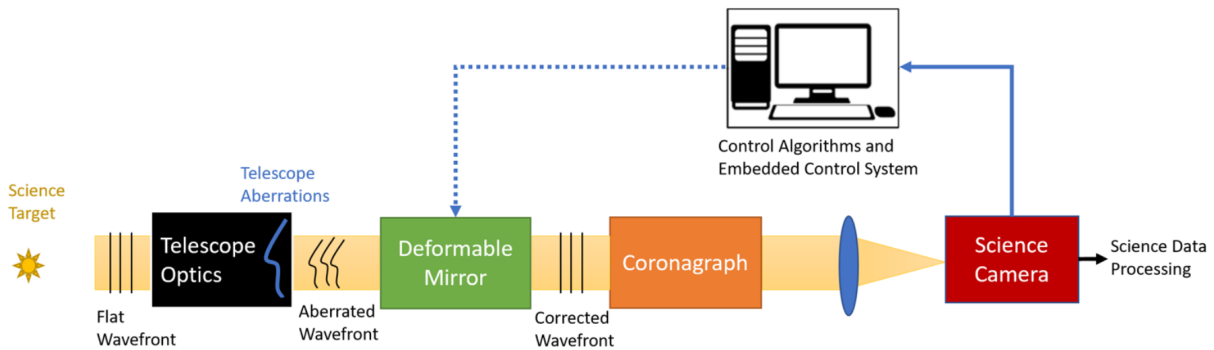


Figure 2.1: Wavefront sensing and control loop on a space telescope, adapted from Belsten et al. [6].

Figure 2.1 shows a representative block diagram of a space telescope where the control system utilizes the following elements: deformable mirrors (DMs) for *control*, the science camera for *sensing* the electric field, and the coronagraph for passively blocking out the light

of the host star. The high-level procedural workflow for imaging an exoplanet is as follows:

1. Point the telescope at a star that potentially has an exoplanet.<sup>1</sup>
2. *Sense & Estimate*: Image with the science camera.
3. *Control*: Use the estimated electric field as input to a control algorithm which outputs DM actuator commands to improve contrast, with the goal of revealing a dim exoplanet.
4. Iterate between steps 2 and 3.

This procedure aims at generating what is known as a “dark hole”. The dark hole is a semi-annular region around the host star where starlight and speckles get canceled out where we would hope to find a habitable zone exoplanet. Popular high-contrast imaging approaches, like EFC, can be grounded in an optimization framework [18]. We have an *objective*, which is to reveal an exoplanet by *deciding* how to move individual actuators of the DM. There are a variety of ways to go about solving the high-contrast problem, with variants of EFC as some of the more prevalent in the literature. The speeds at which you can run EFC are a key factor in determining the level of contrast attainable. If the algorithm is too slow, the wavefront accumulates too much error, and the dark zone is degraded, obscuring the exoplanet.

Table 2.1: Summary of HOWFSC algorithms, adapted from Pogorelyuk et al. [5].

<b>Purpose</b>	<b>Algorithm</b>
Jacobian calculation	<b>Optical modeling</b> [19] Expectation maximization [20]
Controls calculation	<b>Electric field conjugation</b> [4] Linear dark-field control [21] <b>Adjoint-based EFC</b> [22]
Electric field estimation	Pairwise probing [23] Extended Kalman filter [24] [25] Self-coherent camera [26]

---

<sup>1</sup>There may be additional setup steps before step 1, which may include pointing at a reference star.

HOWFSC algorithms involve the creation of the Jacobian, as well as estimation and control, as summarized in Table 2.1. If Jacobian-based control methods are utilized, then a Jacobian can be generated via iterative optical modeling [19] or through system identification with expectation maximization [20]. To generate control commands to the DMs actuators, either electric field conjugation [4], linear dark-field control (LDFC) [21], or adjoint-based EFC [22] can be employed. EFC and LDFC are both Jacobian-based methods, while adjoint-based EFC is Jacobian-less. Lastly, to perform estimation of the electric field, images with DM probes, pupil masks in the self-coherent camera [26], or bright speckles outside of the dark hole with DM dithers all can introduce phase diversity to help estimate and correct the electric field. To turn pixels into electric field states, both batch (pairwise probing) and recursive estimation methods like the extended Kalman filter can be utilized. While there are a variety of HOWFSC algorithms involved in dark hole creation and maintenance, this work pays special attention to those algorithms that have been identified as computational bottlenecks [5], namely EFC and optical modeling.

## 2.1 Electric Field Conjugation (EFC)

EFC is a state-of-the-art method for both generating and maintaining a dark hole. The idea of EFC is to conjugate the electric field that is present in the region of interest at the science camera [4]. An iteration of EFC selects a change in actuator commands  $\delta\mathbf{a}$  to minimize the presence of electric field intensity in the dark zone  $\mathbf{E}_{\text{DZ}}$ , while simultaneously penalizing large actuator movements,

$$J_{\text{EFC}} = \|\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})\|^2 + \|\mathbf{\Gamma}\delta\mathbf{a}\|^2. \quad (2.1)$$

$J_{\text{EFC}}$  is our cost function for EFC, and  $\mathbf{\Gamma}$  is a Tikhonov regularization matrix, typically selected to be a multiple of the identity (a.k.a. ridge regression) [27]. While this cost function can be expressed concisely, the difficulty arises from the  $\mathbf{E}_{\text{DZ}}$  term, which is non-trivial to obtain.

In general, the relationship between the electric field in the entrance pupil (at the first DM) to the starlight at the science camera is a complicated composition of functions applied on  $\delta\mathbf{a}$ , so some assumptions are made to make the formulation more tractable. The electric field in the dark zone can be viewed as a combination of the aberrated electric field  $\hat{\mathbf{E}}_{\text{ab}}$  (without any DM correction) superimposed on the corrected wavefront generated by the DMs, denoted  $\mathbf{E}_{\text{DM}}(\delta\mathbf{a})$ ,

$$\mathbf{E}_{\text{DZ}}(\delta\mathbf{a}) \approx \hat{\mathbf{E}}_{\text{ab}} + \mathbf{E}_{\text{DM}}(\delta\mathbf{a}). \quad (2.2)$$

Assuming corrections to the DMs are small, EFC approximates the mapping from DM to science camera as linear,

$$\mathbf{E}_{\text{DZ}}(\delta\mathbf{a}) = \hat{\mathbf{E}}_{\text{ab}} + \mathbf{G}\delta\mathbf{a}, \quad (2.3)$$

where  $\mathbf{G} \in \mathbb{C}^{m \times n}$  is the Jacobian matrix that maps the DM commands  $\mathbf{a}$  to a pixel-wise electric field contribution at the science camera. Expanding the  $\mathbf{E}_{\text{DZ}}$  term in the cost function, we obtain

$$\|\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})\|^2 = \|\mathbf{G}\delta\mathbf{a}\|^2 + \|\hat{\mathbf{E}}_{\text{ab}}\|^2 + 2\text{Re}\{\hat{\mathbf{E}}_{\text{ab}}^\dagger \mathbf{G}\}\delta\mathbf{a}. \quad (2.4)$$

Collecting terms, we can write the entire cost function  $J_{\text{EFC}}$  in closed form,

$$J_{\text{EFC}} = \|\mathbf{G}\delta\mathbf{a}\|^2 + \|\hat{\mathbf{E}}_{\text{ab}}\|^2 + 2\text{Re}\{\hat{\mathbf{E}}_{\text{ab}}^\dagger \mathbf{G}\}\delta\mathbf{a} + \|\mathbf{\Gamma}\delta\mathbf{a}\|^2. \quad (2.5)$$

We see that the cost function is quadratic, as all terms involve the L2 norm of  $\delta\mathbf{a}$  squared, or are linear in  $\delta\mathbf{a}$ . Since quadratic terms of  $\delta\mathbf{a}$  are generated via a squared L2 norm, we have a strictly convex function with a unique, global minimizer [28]. We only need to find where the gradient vanishes to find the globally optimal solution, which can be done analytically to find

$$(\text{Re}\{\mathbf{G}^\dagger \mathbf{G}\} + \mathbf{\Gamma}^T \mathbf{\Gamma})\delta\mathbf{a}^* = -\text{Re}\{\mathbf{G}^\dagger \hat{\mathbf{E}}_{\text{ab}}\}. \quad (2.6)$$

Equation 2.6 is simply just a linear system of equations  $\mathbf{Ax} = \mathbf{b}$ . Concisely put, the solution

to the regularized, unconstrained optimization problem is  $\delta\mathbf{a}^*$ , which is the optimal DM command *given the current linearization of the DM*. Inevitably, as this linear system of equations is solved for  $\delta\mathbf{a}^*$ , the Jacobian linearization matrix becomes a worse approximation, needing eventual Jacobian re-generation. By iteratively re-linearizing and solving for  $\delta\mathbf{a}^*$ , this procedure both digs and maintains a dark hole as long as the linearization (Jacobian) is a good approximation.

Once a Jacobian is obtained, the question is, what is the most efficient way to solve  $\mathbf{Ax} = \mathbf{b}$  in Equation 2.6? Before exploring the most efficient methods for this calculation, it is worth highlighting that this system of equations is not sparse, as the Jacobian is typically not a sparse matrix. Typical methods for solving linear systems of equations include Gaussian elimination, LU decomposition, QR decomposition, Cholesky decomposition, and singular value decomposition. LU decomposition, QR decomposition, Cholesky decomposition, and Singular Value decomposition hover around the same computational complexity. For the first implementation of performing EFC, we chose Householder QR decomposition as the method of choice to leverage off-the-shelf algorithms from Numerical Recipes in C [29].

Briefly, QR decomposition works to solve  $\mathbf{Ax} = \mathbf{b}$  by factorizing  $\mathbf{A}$  into

$$\mathbf{A} = \mathbf{QR},$$

where  $\mathbf{R}$  is an upper triangular matrix, and  $\mathbf{Q}$  is an orthogonal matrix ( $\mathbf{Q}^\top\mathbf{Q} = \mathbb{I}$ ). The algorithm `qrncmp()` as described in Numerical Recipes in C first forms  $\mathbf{Q}^\top\mathbf{b}$ , then solves

$$\mathbf{Rx} = \mathbf{Q}^\top\mathbf{b}$$

with back substitution. More details on the algorithm can be found in Numerical Recipes in C.

The above procedure involves the use of the Jacobian matrix. To construct the Jacobian matrix, the central finite-difference method is used by probing each DM actuator up and

down by a small amount, and observing the mapping of this variation ( $\delta\mathbf{a}$ ) to the electric field at the science camera. In practice, this needs to be done by executing a representative optical model (called `forward()` in Section 2.2) of the coronagraphic system twice per actuator, which is computationally significant. The Jacobian is generated for a case of  $p$  pixels,  $w$  wavelengths, and  $N_{\text{acts}}$  actuators by executing the `forward()` model  $2wN_{\text{acts}}$  times. The final  $m \times n$  Jacobian collects terms such that  $m = pw$  and  $n = N_{\text{acts}}$ . For two  $96 \times 96$  actuator DMs, five wavelengths, and a circularly-inscribed DM mask (see Figure 3.6b for example), this results in approximately  $10^5$  executions of the `forward()` optical model, which itself involves large Fourier transforms and matrix multiplications. Other methods, such as adjoint-based EFC, were created to ameliorate the issue of expensive Jacobian computation [22], which will be discussed in Section 2.3.

## 2.2 Optical Modeling

The `forward()` optical model is the essential routine underlying both Jacobian creation and adjoint-based EFC. In this work, we focus our attention on modeling a specific instantiation of a coronagraph instrument. Specifically, our optical model utilizes a vector vortex coronagraph with two DMs as outlined by Milani et al. [19].

The optical model accepts DM commands as inputs, propagates light from DM1 to DM2, then to the pupil, then to the coronagraph, and outputs the electric field at the science camera. Given our implementation of optical modeling algorithms will reside on an embedded processor, there exists a tension between the fidelity of optical simulation and execution time. Previous work has delineated between “compact optical modeling” and “full optical modeling” [19] – the former residing on the embedded processor, and the latter residing on a GPU-equipped desktop computer to simulate the telescope’s optics with high fidelity. In the compact optical model, propagation methods involve angular spectrum propagation (ASP) between the DMs, fast Fourier transforms (FFTs), and matrix Fourier transforms (MFTs) as

outlined in Krist et al. [30]. A pictorial representation of the optical model can be seen in Figure 2.2.

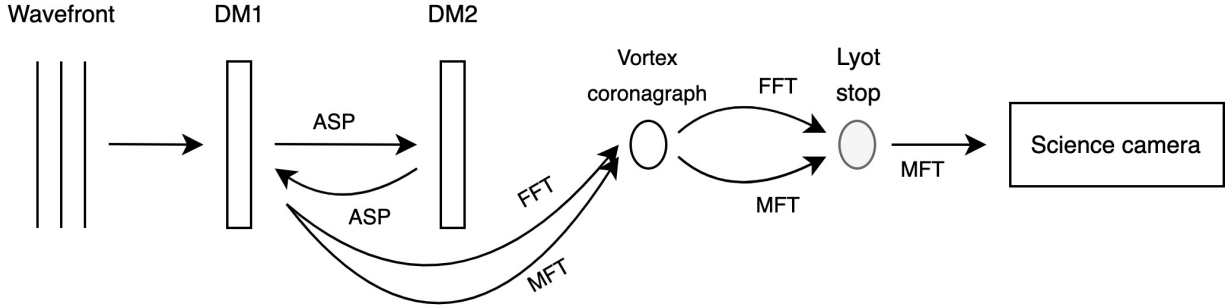


Figure 2.2: Computational flow of optical modeling with a vector vortex coronagraph. The wavefront evolves via angular spectrum propagation (ASP) between DMs, and via FFTs and MFTs from the pupil to the vortex coronagraph, then to the Lyot stop, and finally, to the science camera.

We can pry open the internals of these propagation methods to highlight their computational aspects. Angular spectrum propagation involves an FFT, a complex elementwise matrix multiplication, and an inverse FFT (IFFT). The MFT, which allows for different sampling of the input and output wavefronts, uses two complex matrix multiplications (not elementwise). The `forward()` model is fully articulated in Table 2.2, which rests on the forward DM (FDM) model and the forward vortex (FV) model outlined in Tables 2.3a and 2.3b.

Matrix sizes were included in Table 2.2 to give an idea of computational complexity, but it is worth highlighting the most computationally complex routines of this model. Inside the forward vortex model, the wavefront is padded such that the first FFT ( $\mathcal{O}(n^2 \log(n))$ ) is performed on a  $4096 \times 4096$  matrix of complex values. Followed by the FFT, two complex Hadamard products of matrices ( $\mathcal{O}(n^2)$ ) that are each  $4096 \times 4096$  in size are computed, followed by an inverse FFT ( $\mathcal{O}(n^2 \log(n))$ ) on a  $4096 \times 4096$  matrix. The other computationally dominant components are the MFT and the inverse MFT (IMFT). The MFT performs two matrix multiplications ( $\mathcal{O}(mnp)$  for an  $m \times n$  matrix times an  $n \times p$  matrix), a  $1000 \times 1000$  times a  $1000 \times 1200$ , then a  $1200 \times 1000$  times a  $1000 \times 1200$ . The IMFT exhibits the same

<b>forward() Model Operation</b>	<b>Matrix Size</b>
$\mathbf{E}_{\text{EP}} = \text{AP}(x, y) \odot \text{WFE}(x, y)$	$1000 \times 1000$
$\boldsymbol{\psi}_{\text{DM1}} = \text{FDM}[\mathbf{a}_1(x, y)]$	$1000 \times 1000$
$\boldsymbol{\psi}_{\text{DM2}} = \text{FDM}[\mathbf{a}_2(x, y)]$	$2048 \times 2048$
$\mathbf{E}_{\text{DM1}} = \mathbf{E}_{\text{EP}} \odot \boldsymbol{\psi}_{\text{DM1}}$	$1000 \times 1000$
$\mathbf{E}_{\text{DM2P}} = \text{ASP}[\mathbf{E}_{\text{DM1}}, z_{1,2}]$	$2048 \times 2048$
$\mathbf{E}_{\text{DM2}} = \mathbf{E}_{\text{DM2P}} \odot \boldsymbol{\psi}_{\text{DM2}}$	$2048 \times 2048$
$\mathbf{E}_{\text{PUP}} = \text{ASP}[\mathbf{E}_{\text{DM2}}, -z_{1,2}]$	$2048 \times 2048$
$\mathbf{E}_{\text{LP}} = \text{FV}[\mathbf{E}_{\text{PUP}}]$	$1000 \times 1000$
$\mathbf{E}_{\text{LS}} = \mathbf{L}_{\text{ap}} \odot \mathbf{E}_{\text{LP}}$	$1000 \times 1000$
$\mathbf{E}_{\text{FP}} = \text{MFT}[\mathbf{E}_{\text{LS}}]$	$256 \times 256$

Table 2.2: The `forward()` optical model decomposed into the operations that evolve the wavefront from the entrance pupil to the final focal plane. AP is the electric field at the input aperture, WFE is the representative wavefront error at the input aperture, FDM is the forward DM model, and FV is the forward vortex coronagraph model.  $z_{1,2}$  denotes the propagation distance between DM1 and DM2, and  $\odot$  denotes elementwise matrix multiplication,

computational complexity.

These large FFTs and matrix multiplications are some of the most expensive routines in the entire optical model. In addition, the forward DM model includes an FFT of the  $1024 \times 1024$  influence function, along with a  $1024 \times 1024$  IFFT to obtain the phase of the DM command, and the forward DM model is evaluated for both DM commands. In addition to the above FFTs, there are also two  $2048 \times 2048$  FFTs inside ASP, and ASP is run twice. The optical model has a total of eight FFTs: two of shape  $4096 \times 4096$  to model the vortex, four of shape  $2048 \times 2048$  from ASP, and four of shape  $1024 \times 1024$  in the forward DM model.<sup>2</sup>

The optical model as described [19] is wavelength dependent. To perform broadband versions of EFC or variants, an execution of the optical model will be needed for each wavelength. The specific parameters used in the optical model described in this work can be

---

<sup>2</sup>It is worth noting that most modern FFT implementations can achieve the FFT speedup originally documented in Cooley and Tukey [31] without the restriction of data sizes to powers of two.

Forward DM Model
Given $\mathbf{a}(x, y)$
$\hat{\mathbf{a}} = \text{MFT}[\mathbf{a}(x, y)]$
$\hat{\mathbf{S}}_{\text{DM}} = \hat{\mathbf{a}} \odot \hat{\mathbf{F}}(x, y)$
$\mathbf{S}_{\text{DM}} = \text{IFFT}[\hat{\mathbf{S}}_{\text{DM}}]$
$\phi_{\text{DM}} = \frac{4\pi}{\lambda} \mathbf{S}_{\text{DM}}$
$\psi_{\text{DM}} = e^{i\phi_{\text{DM}}}$

(a) The forward DM (FDM) model.

Forward Vortex Model
Given $\mathbf{E}_{\text{PUP}}$
$\mathbf{E}_{\text{FP,LR}} = \text{FFT}[\mathbf{E}_{\text{PUP}}]$
$\mathbf{E}_{\text{FPM,LR}} = \mathbf{E}_{\text{FP,LR}} \odot \mathbf{V}_{\text{LR}}(x, y) \odot ([\mathbf{1}] - \mathbf{W}_{\text{LR}}(x, y))$
$\mathbf{E}_{\text{LP,LR}} = \text{IFFT}[\mathbf{E}_{\text{FPM,LR}}]$
$\mathbf{E}_{\text{FP,HR}} = \text{MFT}[\mathbf{E}_{\text{PUP}}]$
$\mathbf{E}_{\text{FPM,HR}} = \mathbf{E}_{\text{FP,HR}} \odot \mathbf{V}_{\text{HR}}(x, y) \odot \mathbf{W}_{\text{HR}}(x, y)$
$\mathbf{E}_{\text{LP,HR}} = \text{IMFT}[\mathbf{E}_{\text{FPM,HR}}]$
$\mathbf{E}_{\text{LP}} = \mathbf{E}_{\text{LP,LR}} + \mathbf{E}_{\text{LP,HR}}$

(b) The forward vortex (FV) model.

Figure 2.3: The forward DM model transforms DM actuator commands  $\mathbf{a}$  via an MFT, an elementwise product with the influence function  $\hat{\mathbf{F}}$ , an IFFT to obtain the surface  $\mathbf{S}_{\text{DM}}$ , and then elementwise exponentiation to obtain a phasor representation. The forward vortex model transforms the electric field at the pupil plane  $\mathbf{E}_{\text{PUP}}$  into the electric field at the Lyot plane  $\mathbf{E}_{\text{LP}}$  through the utilization of vortex low resolution (LR) and high resolution (HR) masks  $\mathbf{V}$  and window functions  $\mathbf{W}$ .  $[\mathbf{1}]$  is used to denote a matrix of all one's.

found in Table 2.3.

## 2.3 Adjoint-Based EFC (AD-EFC)

In 2021, Scott Will et al. showed that a new approach to EFC, named Adjoint-Based EFC (AD-EFC), had the potential to exhibit asymptotic computational efficiency in memory consumption and CPU time as the number of DMs and pixels in the dark zone increase [22]. Following this work, in 2023, Will et al. showed the first experimental demonstration of AD-EFC on the High-contrast Imager for Complex Aperture Telescopes at the Space Telescope Science Institute [27]. In this work, we explore the implementation of AD-EFC for the first time in an embedded context.

The formulation of adjoint-based EFC looks strikingly similar to EFC, with cost function

Table 2.3: Optical modeling parameters

Parameter	Value
Actuators across DM	64
Actuator spacing	1.4 mm
Actuator-to-actuator coupling	15%
DM beam diameter	47 mm
DM1 to DM2 distance	700 mm
Central wavelength	650 nm
Lyot stop ratio	0.9
Detector sampling	$0.347 \lambda/D$ at 650 nm

$J_{\text{AD-EFC}}$  as follows,

$$J_{\text{AD-EFC}} = \frac{1}{\|\hat{\mathbf{E}}_{\text{ab}}\|^2} J_{\text{EFC}} = \frac{1}{\|\hat{\mathbf{E}}_{\text{ab}}\|^2} (\|\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})\|^2 + \|\mathbf{\Gamma}\delta\mathbf{a}\|^2). \quad (2.7)$$

Where the adjoint-based approach diverges from traditional EFC is in the representation of  $\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})$ , and how the corresponding control iteration ensues. In EFC,  $\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})$  is modeled with a linear map from DM1 to the science camera by constructing the Jacobian matrix via the finite-difference method. In traditional EFC, the cost function explicitly depended on  $\mathbf{a}$ , and there existed an analytical minimizer  $\delta\mathbf{a}^*$  of the cost function. This turned the optimization framework from an unconstrained minimization to solving a system of linear equations. The iterative element only came from the construction of the Jacobian via repeated optical modeling [32].

In AD-EFC, the construction of the Jacobian matrix (and the linearization assumption) is abandoned, and instead we notice the cost function implicitly depends on  $\delta\mathbf{a}$ . As long as one can compute both the *value* and the *gradient* of the cost function, iterative optimization can be used to find the minimum of the cost function. To compute the cost function, the regularization term is evaluated trivially, and  $\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})$  can be computed with a single iteration of the `forward()` optical model. That is, for a current iteration's input  $\delta\mathbf{a}$ , the `forward()`

model is run to generate  $\mathbf{E}_{\text{DM}}(\delta\mathbf{a})$ , and in conjunction with the sensed electric field  $\hat{\mathbf{E}}_{\text{ab}}$ ,  $\mathbf{E}_{\text{DZ}}(\delta\mathbf{a})$  is calculated via Equation 2.2. We will reserve the discussion of gradient calculation for Section 2.3.3.

A common choice for regularization matrix  $\mathbf{\Gamma}$  is a small constant times the identity matrix, which gives the user a level of control over the penalty for large actuator commands. In conjunction with Equation 2.2 and the definition of  $\mathbf{E}_{\text{DM}}$ , the optimization problem utilizing the cost from Equation 2.7 can be expressed as

$$\min_{\delta\mathbf{a} \in \mathbb{R}^{N_{\text{acts}}}} \frac{1}{\|\hat{\mathbf{E}}_{\text{ab}}\|^2} (\|\hat{\mathbf{E}}_{\text{ab}} + \mathbf{E}_{\text{DM}}(\delta\mathbf{a})\|^2 + \alpha \|\delta\mathbf{a}\|^2). \quad (2.8)$$

In other words, we seek to select DM commands  $\delta\mathbf{a}$  that minimize the electric field in the control-masked region of the image plane while penalizing large DM movements.

### 2.3.1 Selecting an Optimizer

The selection of an optimization algorithm involves important tradeoffs driven by the characteristics of the objective function. We can use a few questions to guide our selection: Is the cost function high-dimensional? Is the cost function convex? How can gradients be calculated? Does stochasticity aid in finding the optimum? Which algorithm has the best convergence guarantees? In principal, we would like to select the algorithm that has the quickest convergence guarantees without too much computational overhead. For AD-EFC, problem 2.8 is unconstrained, and  $J_{\text{AD-EFC}}$  is convex in  $\delta\mathbf{a}$ , where  $\delta\mathbf{a}$  is a very high-dimensional input, numbering in the 1000s.

First, addressing the most fundamental algorithms, we could go the simplest route with gradient descent, but it doesn't offer very much in the way of quick convergence guarantees. Newton's method could be a great choice for a convex cost function, as it delivers local quadratic convergence to the optimum, but it requires the explicit formation of the Hessian matrix. For reference, the sizes of the Hessian matrix are provided for small, medium, and

large problem sizes (with respect to two circularly masked DMs for sizes  $48 \times 48$ ,  $64 \times 64$ , and  $96 \times 96$  actuators, respectively).

DM Size	$N_{\text{acts}}$	Hessian Size
$48 \times 48$	3752	113 MB
$64 \times 64$	6600	348 MB
$96 \times 96$	14736	1.74 GB

Table 2.4: Hessian sizes corresponding to different DM sizes.  $\delta \mathbf{a} \in \mathbb{R}^{N_{\text{acts}}}$ , where  $N_{\text{acts}}$  is the number of unmasked DM actuators in two DMs sized by the left-most column.

Table 2.4 shows just how unwieldy Hessians – let alone inverse Hessians – would be to store and perform computation with. For this reason, quasi-Newton methods, which aim to avoid the explicit formation of the Hessian, are an appealing suite of methods. Quasi-Newton methods utilize first-order information with a little extra computation to exhibit superlinear convergence.

While there are a variety of quasi-Newton methods, reasoning outlined by Scott Will et al. [22] highlighted that L-BFGS (the limited-memory BFGS algorithm) exhibits quick convergence without extreme memory requirements. In generic BFGS, the descent direction resembles Newton’s method, except an approximation of the inverse Hessian is used in place of the true inverse at each step  $k$  [33].

From an embedded programmer’s point of view, the interface to an L-BFGS minimizer requires the programmer to provide an initial condition  $\delta \mathbf{a}_0$ , and a function that can compute the value and the gradient of the cost function for a given  $\delta \mathbf{a}$ . In this work, we’ll call this function `val_and_grad()`.

### 2.3.2 Computing $J_{\text{AD-EFC}}$

$J_{\text{AD-EFC}}$ , as defined in Equation 2.8, is a function of both the current state of the DMs  $\mathbf{a}$ , as well as the change in DM command  $\delta \mathbf{a}$ . Inspired by Milani et al. [34],  $\mathbf{E}_{\text{DM}}(\delta \mathbf{a})$  can be further decomposed into  $\mathbf{E}_{\text{DM}}(\delta \mathbf{a}) \equiv \mathbf{E}_{\text{F}}(\mathbf{a} + \delta \mathbf{a}) - \mathbf{E}_{\text{F}}(\mathbf{a})$ , where  $\mathbf{E}_{\text{F}}(\mathbf{a})$  is the *nominal* focal plane electric field computed by a single `forward()` pass for the current state of the DMs.

Thus,  $\mathbf{E}_F(\mathbf{a} + \delta\mathbf{a})$  is the focal plane electric field as evaluated by the `forward()` optical model, but with the decision variable  $\delta\mathbf{a}$  superimposed onto the current state of the DMs. Collecting terms, we have the following:

- $\hat{\mathbf{E}}_{ab}$ : The estimated electric field sensed at the science camera after being masked. Computed once per control iteration.
- $\mathbf{E}_F(\mathbf{a} + \delta\mathbf{a})$ : The focal plane electric field computed by a single `forward()` optical model pass given the current state  $\mathbf{a}$  of the DMs, as well as the current set of decision variables  $\delta\mathbf{a}$ . Needs to be recomputed iteratively inside of `val_and_grad()` for each optimizer iteration.
- $\mathbf{E}_F(\mathbf{a})$ : The *nominal* focal plane electric field computed by a single `forward()` optical model pass given the *current* state  $\mathbf{a}$  of the DMs. Computed once per control iteration.
- $\delta\mathbf{a}$ : The decision variables of the optimizer.

Thus, for a given control iteration,  $\hat{\mathbf{E}}_{ab}$  is constant, and  $\mathbf{E}_F(\mathbf{a})$  is obtained from a single `forward()` optical model execution for the current DM state. Then, within optimizer iterations (many optimizer iterations within one control iteration), every time the decision variables  $\delta\mathbf{a}$  are updated, the `forward()` model is executed to obtain  $\mathbf{E}_F(\mathbf{a} + \delta\mathbf{a})$ . To evaluate the L2 norm squared,  $\delta\mathbf{a}$  is a vector, so  $\|\delta\mathbf{a}\|_2^2 = \delta\mathbf{a}^\top \delta\mathbf{a}$ . The complex electric field term is vectorized, and the L2 norm squared is simply the complex conjugate times itself.

$J_{AD-EFC}$  is therefore very easy to compute, but we need to address the computation of the gradient. Since the cost function implicitly depends on  $\delta\mathbf{a}$  by a web of computations, it is not straightforward to calculate the gradient of the cost function. Just like the loss function in training a neural network, a useful tool to calculate the gradient is reverse-mode algorithmic differentiation (RMAD).

### 2.3.3 Computing $\frac{dJ_{\text{AD-EFC}}}{d\delta\mathbf{a}}$ with Automatic Differentiation

We can rewrite  $J_{\text{AD-EFC}}$  as

$$J_{\text{AD-EFC}}(\delta\mathbf{a}) = \frac{1}{\|\hat{\mathbf{E}}_{\text{ab}}\|^2} (\|\hat{\mathbf{E}}_{\text{ab}} + \mathbf{E}_{\text{F}}(\mathbf{a} + \delta\mathbf{a}) - \mathbf{E}_{\text{F}}(\mathbf{a})\|^2 + \alpha\|\delta\mathbf{a}\|^2). \quad (2.9)$$

As highlighted in the previous sections,  $\mathbf{E}_{\text{F}}(\mathbf{a} + \delta\mathbf{a})$  is obtained via `forward()`, which itself is none other than a composition of functions applied on  $\delta\mathbf{a}$ . We can rewrite  $J_{\text{AD-EFC}}$  as a composition of functions applied on  $\delta\mathbf{a}$ ,

$$J_{\text{AD-EFC}}(\delta\mathbf{a}) = f \circ g \circ \dots \circ h(\delta\mathbf{a}).$$

Algorithmic (or auto-) differentiation is a process to compute precise derivatives of smooth functions which can be decomposed into many elementary operations involving one or two arguments at a time [33]. If we decompose a function into many “atomic” operations, we can construct the gradient using the chain rule with each intermediate variable. For more details on introductory algorithmic differentiation, see Appendix A.

The same decomposition can be done with the cost function  $J_{\text{AD-EFC}}(\delta\mathbf{a})$ . In general, the gradient at each node is a function of the *value* of the input and the *value* of the intermediate variables. Gradient construction with RMAD is done by first a forward pass to evaluate the numerical value of the cost function, and a corresponding backward pass to construct the gradient. Additionally, we cache the “forward variables” (intermediate variables) used along the way that we need for the gradient computation.

In general, we can have arbitrary mathematical entities as nodes in our computational graph (scalars, vectors, tensors), and all edges can be viewed as local linear operations operating on the underlying vector space. The operations involved in optical modeling, like matrix-vector multiplication, complex Hadamard matrix multiplication, FFTs, MFTs, and zero-padding/cropping, can be viewed from the same paradigm. **The tool that allows us to**

propagate gradients backward with these more complex objects is what is known as the “adjoint” model. The “adjoint variables” are then the entities that appear in the chain rule – they are simply just a more general representation of the derivative. The adjoint of variable  $\mathbf{x}$  is denoted as  $\bar{\mathbf{x}} = \partial J / \partial \mathbf{x}$ . Table 2.5 shows the primitive adjoint operations.

Table 2.5: Forward vs. adjoint model primitives, adapted from Will et al. [22]

Forward model	Adjoint model
$y = x + a$	$\bar{x} = \bar{y}$
$y = a \odot x$	$\bar{x} = a^* \odot \bar{y}$
$y = Ax$	$\bar{x} = A^\dagger \bar{y}$
$y =  x ^2$	$\bar{x} = 2x \odot \text{Re}(\bar{y})$
$y = \exp(jx)$	$\bar{x} = \text{Im}(\bar{y} \odot y^*)$
$y = x[1_s]$	$\bar{x}[1_s] = \bar{y}$
$y = \text{FFT}\{x\}$	$\bar{x} = \text{IFFT}\{\bar{y}\}$
$y = \text{MFT}\{x\}$	$\bar{x} = \text{IMFT}\{\bar{y}\}$
$y = \text{ASP}\{x; \Delta z\}$	$\bar{x} = \text{ASP}\{\bar{y}; -\Delta z\}$

In Table 2.5,  $\odot$  denotes the Hadamard product, and  $x[1_s]$  refers to masking the  $x$  variable (i.e., plucking off  $x$ ’s components at certain locations). Additional nuances to the notation and detailed derivations can be found in Will et al. [22]. With these primitives, we can fully construct an adjoint model to the forward model detailed in Tables 2.2, 2.3a, and 2.3b.

Recent work from Milani et al. [34] provides a detailed derivation of the adjoint operations for the vortex coronagraph, which we’ve included here for brevity in Table 2.6. With a complete implementation of the adjoint model,  $\frac{dJ_{\text{AD-EFC}}}{d\delta\mathbf{a}}$  can be computed, and the L-BFGS optimization can ensue.

### 2.3.4 Iterative Minimization

Like BFGS, L-BFGS continues to perform rank-two updates at each iteration, but does not store the full dense approximation of the inverse Hessian. Instead, L-BFGS stores a representative set of vectors that account for the bulk structure of the inverse Hessian. Particularly, L-BFGS only stores the last  $\ell$  pairs of vectors that were updated in the previous  $\ell$  iterations. L-BFGS can achieve good performance with  $\ell$  being somewhere between 3 and

Table 2.6: Adjoint vortex model operations, adapted from Milani et al. [34]. All variable definitions are provided in Table 2.3b.

Forward Vortex Model Operations	Adjoint Vortex Model Operations
Given $\mathbf{E}_{\text{PUP}}$	$\frac{\partial J}{\partial \mathbf{E}_{\text{PUP}}} = \frac{\partial J}{\partial \mathbf{E}_{\text{PUP,LR}}} + \frac{\partial J}{\partial \mathbf{E}_{\text{PUP,HR}}}$
$\mathbf{E}_{\text{FP,LR}} = \text{FFT}[\mathbf{E}_{\text{PUP}}]$	$\frac{\partial J}{\partial \mathbf{E}_{\text{PUP,LR}}} = \text{IFFT}\left[\frac{\partial J}{\partial \mathbf{E}_{\text{FP,LR}}}\right]$
$\mathbf{E}_{\text{FPM,LR}} = \mathbf{E}_{\text{FP,LR}} \odot \mathbf{V}_{\text{LR}}(x, y) \odot (1 - \mathbf{W}_{\text{LR}}(x, y))$	$\frac{\partial J}{\partial \mathbf{E}_{\text{FP,LR}}} = \frac{\partial J}{\partial \mathbf{E}_{\text{FPM,LR}}} \odot \mathbf{V}_{\text{LR}}^*(x, y) \odot (1 - \mathbf{W}_{\text{LR}}(x, y))^*$
$\mathbf{E}_{\text{LP,LR}} = \text{IFFT}[\mathbf{E}_{\text{FPM,LR}}]$	$\frac{\partial J}{\partial \mathbf{E}_{\text{FPM,LR}}} = \text{FFT}\left[\frac{\partial J}{\partial \mathbf{E}_{\text{LP}}}\right]$
$\mathbf{E}_{\text{FP,HR}} = \text{MFT}[\mathbf{E}_{\text{PUP}}]$	$\frac{\partial J}{\partial \mathbf{E}_{\text{PUP,HR}}} = \text{IMFT}\left[\frac{\partial J}{\partial \mathbf{E}_{\text{FP,HR}}}\right]$
$\mathbf{E}_{\text{FPM,HR}} = \mathbf{E}_{\text{FP,HR}} \odot \mathbf{V}_{\text{HR}}(x, y) \odot \mathbf{W}_{\text{HR}}(x, y)$	$\frac{\partial J}{\partial \mathbf{E}_{\text{FP,HR}}} = \frac{\partial J}{\partial \mathbf{E}_{\text{FPM,HR}}} \odot \mathbf{V}_{\text{HR}}^*(x, y) \odot \mathbf{W}_{\text{HR}}^*(x, y)$
$\mathbf{E}_{\text{LP,HR}} = \text{IMFT}[\mathbf{E}_{\text{FPM,HR}}]$	$\frac{\partial J}{\partial \mathbf{E}_{\text{FPM,HR}}} = \text{MFT}\left[\frac{\partial J}{\partial \mathbf{E}_{\text{LP}}}\right]$
$\mathbf{E}_{\text{LP}} = \mathbf{E}_{\text{LP,LR}} + \mathbf{E}_{\text{LP,HR}}$	Given $\frac{\partial J}{\partial \mathbf{E}_{\text{LP}}}$

20 [22].

It turns out that the inverse Hessian can be built with  $4\ell n$  FLOPs where  $\ell$  is the number of previous iterations aiding in the approximation, and  $n$  is the input dimension (Algorithm 7.4 in [33]). Therefore, the computation of the inverse Hessian in one iteration becomes  $\mathcal{O}(N_{\text{acts}})$  (ignoring the constant factor times small  $\ell$ ).

In summary, Newton’s method would have required both the explicit computation and explicit inversion of the Hessian at every iteration. This would have involved constructing an  $N_{\text{acts}} \times N_{\text{acts}}$  matrix, and inverting it would cost  $\mathcal{O}(N_{\text{acts}}^3)$ , yielding even worse computational complexity than constructing the Jacobian.

The descent direction in BFGS looks like,

$$\mathbf{d}_k = -\mathbf{H}_k^{-1} \nabla J_{\text{AD-EFC}, k}, \quad (2.10)$$

where  $\mathbf{H}_k^{-1}$  is an approximation of the inverse Hessian, and the input is updated according to

$$\delta \mathbf{a}_{k+1} = \delta \mathbf{a}_k + \alpha_k \mathbf{d}_k. \quad (2.11)$$

Importantly, the inverse Hessian  $\mathbf{P}_k := \mathbf{H}_k^{-1}$  is updated via sequential rank-two updates given the estimate from the previous iteration:

$$\mathbf{P}_{k+1} = (\mathbb{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) \mathbf{P}_k (\mathbb{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T, \quad (2.12)$$

where  $\mathbf{s}_k = \delta \mathbf{a}_{k+1} - \delta \mathbf{a}_k$ ,  $\mathbf{y}_k = \nabla J_{\text{AD-EFC}, k+1} - \nabla J_{\text{AD-EFC}, k}$  and  $\rho$  is a scalar to normalize the update. The advantage of utilizing rank-two updates to the Hessian lies in the fact that the structure of the inverse Hessian is largely preserved. Since the singular values of  $\mathbf{P}$  are preserved, previous decompositions can be reused with slight modifications to speed up computation rather than computing the decomposition from scratch.

Since the execution of `val_and_grad()` is very large for our circumstances, the complexity due to the approximation of the inverse Hessian at each step gets drowned out by the

computation of `val_and_grad()`. However, in conjunction with determining a descent direction, a line search is also executed to find a suitable  $\alpha_k$ .

Both `SciPy` and the `liblbfgs` library (the underpinning libraries for AD-EFC in both Python and C, discussed in Section 3.3) make use of the Moré-Thuente line search. The Moré-Thuente line search is effectively done in two parts: bracketing and refinement [35]. To briefly summarize, the bracketing stage finds an interval that brackets a point satisfying the Wolfe conditions [36]. This may involve expanding the step if it's too small or reducing the step if it overshoots significantly. The refinement step uses an interpolation scheme (typically cubic, but could be quadratic or secant), based on function values and derivatives at the interval endpoints to efficiently zoom in on a suitable step. Critically, depending on the local geometry of the cost function, the line search could involve many more calls to `val_and_grad()`. The line search is the main source of variance in the execution of AD-EFC due to the uncertainty in the number of `val_and_grad()` executions. In the `liblbfgs` library, the line search includes a default upper bound of 40 calls to the `val_and_grad()` function, but in practice, it is typically much less. Still, the AD-EFC algorithm is fundamentally driven by the number of calls to `val_and_grad()`, in which line search takes a sizeable share.

Having outlined the algorithmic background of EFC, optical modeling, and AD-EFC, we can begin to detail our implementation choices and speedups on embedded processors. We will carefully step through the programming and some numerical details of both libraries and specific algorithm implementations.

# Chapter 3

## Implementation of HOWFSC Algorithms

### 3.1 Processor Setup

Initially, the first processor we worked with was another functional equivalent to the BAE RAD5545, named the QorIQ P5040RDB from NXP. While the P5040 did not end up getting used for this work due to a malfunctioning boot environment, the bring-up of the P5040 processor is a representative case of what is to be expected when developing on functional equivalent processors to radiation-tolerant processors. The processor bring-up illustrates the difficulties working with decade-old embedded processors, and the older documentation, old tooling, and quirky embedded workflows that come along with them.

Out-of-the-box, some simple documentation included instructions on the proper dip switches to set, enabling a boot-up the processor. Communications with the P5040 was then created over a serial terminal via the UART port. Once a communication link with the P5040 was established, our attention shifted to attempting to deploy applications onto the P5040. The main piece of documentation provided by NXP was the Freescale Linux SDK for QorIQ Processors [37]. This guide highlighted that the SDK was verified only on a specific subset of Linux distributions (e.g. Ubuntu 10.04, 12.04, and 13.04). Thus, a virtual machine (VM) on a modern laptop was booted to launch Ubuntu 12.04, where the software development

kit (SDK) (existing in the form of three binary files) were concatenated into an image and mounted to read-only memory.

Inside the SDK, there were many “meta” folders that contained files related to “recipes”. The language in question comes from the lexicon of the Yocto build project, which is an open source project dedicated to enabling the creation of custom Linux distributions for even obscure embedded targets [38]. The SDK was full of metadata, packages, and recipes, which are all of the crucial infrastructure required to make a custom image for the P5040. The documentation further reinforced the idea that in order to get applications onto the target device, an entire image needed to get BitBake’d and deployed somehow to the target. Methods of deploying the image included utilizing Network File System (NFS) and Trivial File Transfer Protocol (TFTP), both of which allow the user to keep crucial files on the host environment, and allowing the embedded environment to access certain files over network.

Being unordained in this embedded systems workflow, an alternate direction was sought to eliminate the deployment of a whole new image to the device. Especially considering the P5040 already had a functional image onboard, it would have been suboptimal to brick the device with an image from a self-constructed build process. Attention was turned to seeking alternate routes of deploying applications onto the device. Inquiring into the slim image already existing on the P5040, it was discovered that the image was pre-built and deployed by the manufacturer via a specific version of the Yocto build process. The image on the P5040 corresponded to version Denzil (April 2012 1.2). It was at this point that it was realized that Yocto included separate documentation that instructed users how to build and deploy applications with, or on top of, Yocto images [39].

The Yocto Application Development Toolkit (ADT) included a variety of workflows, one of which involved “Using a Cross-Toolchain Tarball”. This workflow only required downloading a cross-toolchain tarball, untarring it, and launching a host environment setup script, rather than building a separate image for the device with one’s application on board. As per the original documentations instructions, the tarball was downloaded on the VM, and with proper

permissions via the `chmod` command, the environment setup script was executed. The most important commands from the host setup script are included in Figure 3.1.

```
export PATH=/opt/poky/1.2.1/sysroots/x86_64-pokysdk-linux/usr
/bin:/opt/poky/1.2.1/sysroots/x86_64-pokysdk-linux/usr/bin
/ppc603e-poky-linux:$PATH
export PKG_CONFIG_SYSROOT_DIR=/opt/poky/1.2.1/sysroots/
ppc603e-poky-linux
export PKG_CONFIG_PATH=/opt/poky/1.2.1/sysroots/ppc603e-poky-
linux/usr/lib/pkgconfig
export CC=powerpc-poky-linux-gcc
export CONFIGURE_FLAGS="--target=powerpc-poky-linux --host=
powerpc-poky-linux --build=x86_64-linux --with-libtool-
sysroot=/opt/poky/1.2.1/sysroots/ppc603e-poky-linux"
export CFLAGS=" -m32 -mhard-float -mcpu=603e --sysroot=/opt/
poky/1.2.1/sysroots/ppc603e-poky-linux"
export LDFLAGS=" --sysroot=/opt/poky/1.2.1/sysroots/ppc603e-
poky-linux"
```

Figure 3.1: Important commands from the host setup script for the P5040 processor.

The host setup script adds the target's sysroot to the `PATH`, sets the default compiler to the cross-compiler, and sets critical compiler flags that are target-architecture-aware. In general, this setup script configures all necessary environment variables to let the shell session know that compilation will occur on the host architecture to construct binaries for the target architecture. Sysroots are a very important part of this process, as they provide a snapshot of the target's filetree (from `root`) on the host platform so that the host processor knows what headers and libraries are available on the target.

In our nascent development process, simple applications were written in the Ubuntu 12.04 on a VM equipped with the cross-toolchain. To deploy and run these applications, a UART to USB converter was utilized between the laptop hosting the VM and the P5040. A serial terminal (`minicom`) was set up inside the VM to interface with the P5040. Then, the application would be `uencode`'d on the VM so that it could be pasted into `vi` inside the P5040 serial terminal. To run the binary, the `.txt` file was `udecode`'d on the target and could successfully run.

As our development process matured, we took advantage of networking. We set a static IP address to the P5040, and connected it with our desktop utilizing ethernet and a network switch. This workflow transferred our communications from the serial terminal to Secure Shell Protocol (SSH). To further simplify the workflow, the cross-toolchain tarball was then successfully downloaded on a modern version of Ubuntu (Jammy Jellyfish) on a desktop computer. The final workflow involved cross-compiling from a modern operating system, communicating with the target over SSH, and deploying binaries to the target over Secure Copy Protocol (SCP).

These learnings radically sped up our transition to developing on both the T1040 and LS1046. These processors had very similar bring-ups, which involved setting dipswitches, setting the target’s static IP address, and locating a cross-toolchain tarball for the target. Host-setup scripts could be found for both the T1040 and LS1046, which included very similar commands to those found in Figure 3.1. Thus, to compile and deploy binaries to these embedded target, just a few crucial environment variables needed to be set which instruct the shell session which compiler to use, where to locate sysroots, and other details about the host and the target. The C compilers utilized for all development environments can be seen in Table 3.1.

Table 3.1: Available Compilers by Processor

<b>Processor</b>	<b>Compiler</b>
Desktop	gcc
P5040	powerpc-poky-linux-gcc
T1040	powerpc-fsl-linux-gcc
LS1046	aarch64-fsl-linux-gcc

## 3.2 Software Considerations

In conjunction with the processor setup, the next most crucial layer of the technology stack is the selection of programming language and the algorithms to follow. After a programming

language is selected, we detail the selection between incorporating third-party library source code or writing our own algorithms from scratch.

### 3.2.1 Selection of Programming Language

The selection of a programming language for implementing HOWFSC algorithms was guided by a few criteria:

1. **Speed:** The chosen language should be commonplace in low-latency applications, and should contain minimal/zero unplanned overhead by the programmer to maximize performance. Overhead *can* come in the form of interpreters, dynamic typing, and garbage collection.
2. **Embedded maturity:** Embedded maturity is qualified by having ubiquitous compilers, where even less popular computing architectures have cross-compilation toolchains readily available. Without embedded maturity, the process of writing and deploying code onto one's embedded device can be very cumbersome. Further, the language needs to be able to support real-time, low-latency applications, which typically involve manual memory management, and discourages the presence of a garbage collector in order to ensure deterministic runtime behavior.
3. **Ease of development, and access to libraries:** The open-source-driven nature of many programming languages naturally causes specific communities to congregate and develop their preferred tooling and libraries to solve problems the community cares about most. For example, Python has become home to the deep learning and machine learning communities, which have given rise to the standard deep learning frameworks like PyTorch [40] and Tensorflow [41]. Certain programming languages also have specific kinds of tooling that were created out of necessity for the community, like the extensive static analysis tools and source code analyzers in C with embedded applications in mind. While most things are possible in many programming languages,

some languages naturally lend themselves to specific programming paradigms, like functional programming in C, and object-oriented programming in C++, for example.

Letting these factors guide our choices, C became the language of choice for the implementation of HOWFSC algorithms.

## C/C++

C/C++ are the only languages that can be deployed “out-of-the-box” onto our embedded platforms with minimal oversight. For the developers who deploy the slim image onto the T1040 and LS1046, C is the expected language of use, as shown in Section 3.1. C has been the primary language of choice at NASA for decades, with a mature style guide to write performant and safe embedded code [42]. C and C++ have been the de facto standard for safety-critical and low-latency embedded applications for much longer than competing languages, yielding very extensive tool support, with many source code analyzers, debuggers, logic model extractors, and a large selection of stable compilers. Cross-compilers are typically provided by the manufacturer for even the most unpopular architectures, giving an extra sense of security that code written in C will likely be of value.

Even though the cross-toolchains for both the T1040 and the LS1046 provided a C++ compiler, C was chosen over C++ because HOWFSC algorithms lend themselves particularly well to the functional programming paradigm. For example, optical modeling can be seen as simply a composition of functions, as detailed in Section 2.2. Also, there is a rich ecosystem of libraries and code written in C for common scientific computing operations, like linear algebra and Fourier transforms. The extra levels of abstraction gained by C++ would not have provided a strong enough advantage to justify the added complexity to a largely functional workflow.

It is worth mentioning that despite C’s industry dominance in embedded and low-latency applications for many years, recent developments in other programming languages are challenging C’s dominance. Other languages gaining traction in both the embedded and

scientific communities that deserve some attention are Rust and Julia.

## Rust

Rust, a programming language known for being a memory-safe competitor to C/C++, gained a lot of traction in the 2010s. The LLVM-based language has gained stature as a viable alternative to C/C++ in low-latency or high-performance computing applications [43]. Rust distinguishes itself from C and C++ by building in extra tooling into the compiler directly, rendering memory-unsafe code much more difficult, and in certain circumstances, impossible to compile. Rust boasts the functionality of resolving dangling references with lifetime analysis and forced static analysis with the compiler. While “Resource Acquisition Is Initialization” (RAII) is implemented in C++ [44], C++ still requires savvy embedded development to avoid making these mistakes.

Rust, however, is still relatively nascent in its ability to support many embedded targets *out-of-the-box*. Like many things, it is possible to build and compile Rust for a variety of target architectures, but the T1040’s PowerPC e5500 sub-architecture is not officially listed in Rust’s supported targets [45]. However, there are many close-neighbor sub-architectures that are listed under Tier 3 support, meaning the Rust codebase “may or may not work” on those targets. Rust’s support for Power architecture is primarily aimed at server-grade PowerPC chipsets. Cross-compiling Rust projects for the PowerPC e5500 is likely possible, but would take some additional work, a deep familiarity with the Rust language, and would require different cross-compilation workflows.

The LS1046’s 64-bit Arm Cortex-A72 on the other hand, is listed even as high as Tier 1-level support depending on the kind of Linux operating system in use. However, the T1040 and LS1046’s out-of-the-box support for C was the reason why Rust was not selected in the end. NASA’s High Performance Spaceflight Computer (HPSC) developers list Rust as a viable programming language [46]. The HPSC leverages the open-source RISC-V instruction set, for which at least one version has Tier 1 support, and most versions are at least partially

supported ( $\geq$  Tier 3), depending on the OS.

Overall, Rust is a language worth considering in future embedded HOWFSC applications. The primary reason for considering Rust is the sheer difficulty of committing the same memory or resource handling mistakes as one can do in C. While modern C++ concepts like `unique_ptr` ameliorate some of these issues, Rust simply disables the user from making the same mistakes. Rust provides a very active community and helpful compiler messages, both of which can make development a more seamless process. It is also worth noting that Rust doesn't have a specific focus on numerical computing or common linear algebra and spectral method routines that are found in HOWFSC. For this reason, we explore Julia as another option.

## Julia

The Julia programming language was primarily designed for scientific computing, trying to incorporate a blend of high-level syntactic elements the scientific community enjoys from MATLAB, R, and Python, with the computational performance of C and Fortran [47]. Julia, despite having a garbage collector and high-level syntax, shows speed comparable to C/C++ [48]. This makes Julia a very attractive alternative for low-latency applications. Julia's speed boost largely comes from the Just-in-time (JIT) compilation style with accompanying compiler optimizations. Extra speed is also afforded via multiple dispatch, which turns run-time knowledge into specialized code.

Since the critical node in Julia's compilation process is the lowering of the abstract syntax tree into LLVM intermediate representation (IR) [49], theoretically, any target that supports the LLVM backend (determined by the existence of a clang C++ compiler for the target architecture) can support Julia. In reality, it requires more development time for porting Julia code to embedded platforms. Julia was designed for scientists whose workflows include a lot of numerical computing, and who enjoy the interactivity of Read-Eval-Print Loop (REPL) workflows and Jupyter notebooks. Simply put, while the support in the Julia community for

numerical computing has developed a speedy high-level language, it was not really designed with embedded cross-compilation in mind.

Julia supports the option for ahead-of-time compilation, and the effects of Julia’s garbage collector can be dulled, but not brought to zero as is necessary for super real-time execution. Utilizing Julia’s performance tips, you can almost completely emulate the behavior of NASA’s C style guide, which minimizes heap allocations with in-place operations and memory allocation at initialization. There are a number of members of the Julia community who want to bring it into the embedded world. A few projects working on this include `StaticCompiler.jl` [50] and `jlcross` [51]. Some are even writing embedded code for a Raspberry Pi with some success [52]. However, it remains unclear if Julia binaries for generic PowerPC and ARM architectures can be deployed directly to the LS1046 and T1040 without re-inventing the wheel.

### 3.2.2 Incorporating Source Code

The fast Fourier transform (FFT) is one of the most central algorithms to HOWFSC. In Chapter 2, we identified the `forward()` optical model as the backbone to EFC, which is comprised of eight FFTs. When faced with the decision to write an algorithm like the FFT, one must confront selecting from a few different approaches. The approaches are to write one’s own numerical routine from scratch, incorporate a third party’s library into your application’s source code, or build and link against a third-party library using an API call. All have their drawbacks and advantages.

To reason through this space, one has to fully consider the trades at hand. For a routine as fundamental and ubiquitous in the scientific computing community as the FFT, writing one’s own routine can add some unnecessary overhead. Yet, this approach has the advantage of allowing for precise control of the underlying routine, where you can directly implement the algorithm best suited for your specific purposes. This comes with the extra overhead of development time for an algorithm that some have devoted their entire careers into making

as efficient as possible.

The second option is to incorporate a third party’s library into your project’s source code. This approach has the advantage of leveraging the work of others while also being able to make modifications to the underlying routines for your own purposes.

The third option, and one that is the most common among the scientific computing community is to build a third-party library that you link against and call through a simple API. The most popular option is the Fastest Fourier Transform in the West (FFTW) library developed by Matteo Frigo and Steven Johnson [53]. This option will be discussed in Section 3.2.3. The tricky part for deploying this option on an embedded platform is cross-compiling and building source code for the target. Thus, for the early stages of development, option two was chosen, and we decided to implement the FFT in our embedded HOWFSC code via the `dfourn()` routine provided by Numerical Recipes in C (NR in C) [29]. The `dfourn()` routine is the n-dimensional fast Fourier transform which operates on double floating-point precision inputs. The NR in C FFT was a well-established routine written in barebones C, making it easily incorporable into the HOWFSC codebase with relative ease.

This decision also provided a convenient reference point to explore the achievable speedups when one upgrades from naive algorithm implementations, as will be explored in Section 3.2.3. The crux of this work is to explore what extra embedded development time can offer in the way of algorithmic speedups – especially for ubiquitous algorithms like the FFT. The choice of utilizing NR in C’s FFT brought with it the inheritance of NR in C-style data types, which were primarily the `dvector()` and `dmatrix()` data types. These data types would become the way for us to store the complex-valued 2D wavefront as it evolves through the optical model.

### 3.2.3 Third Party Libraries for Speed-Up

#### FFTW

The `dfourn()` routine from NR in C provided a nice zeroth-order implementation, but as the embedded HOWFSC codebase started to be built out and unit tested, we wanted to incorporate state-of-the-art FFT algorithms. The state-of-the-art open source software to run the FFT is the **FFTW** library [53]. FFTW achieves speedup of the FFT by specifically tuning the most optimized FFT it can in the runtime environment. FFTW achieves this by using adaptive code generation, using a variety of FFT algorithms and implementation styles that can be chosen given the machine/compiler characteristics. FFTW also exploits memory hierarchy, something which the naive NR implementation does not do.

With the FFTW's software architecture, incorporation of FFTW into our codebase looks a little different than calls to `dfourn`. FFTW includes a planning stage, in which the routine is tuned to be optimized to the underlying hardware, and an execution stage, where the FFT is executed. Because of this adaptive optimization to the runtime environment, as well as the need to deploy an application with library dependencies, it was originally thought that FFTW would be difficult to deploy onto an embedded platform. However, given enough thought and attempts, a way was found.

For one thing, the FFTW library needed to be built with cross-compilation in mind, and for another, the library needed to be linked to, either statically or dynamically. Dynamic linking a library in a C program rests on the ability to have access to the library in the runtime environment. We could potentially build the FFTW library and dynamically link our application to it during the compilation process, but we would need to somehow deploy the FFTW shared library (`.so`) to the embedded platform. Dynamic linking is built on the principle of being more flexible, so one can upload new libraries to the target device without breaking the application or needing to re-compile.

Yet we didn't need this kind of flexibility for our purposes, and were further halted at the

step of deploying a shared library to the target in the exact filepath the application expects. A simpler route was chosen, static linking, which hammers all the executable machine code necessary from the library into a single, large application executable. With static linking, the library need not be present on the device, which proved simpler to accomplish. The static linking approach has the drawback of having bloated executables and frozen binaries, which do not represent the most updated state of the library. For our purposes however, that was just fine. For example, an example HOWFSC program which executes `dfourn()` (not requiring third party libraries) is  $\sim 500$  KB, whereas the executable including FFTW calls is  $\sim 8$  MB. This is quite a lot of bloat, but even if dynamic linking was pursued, the library would still need to exist in memory on the device anyway.

Given these considerations, we desired to cross-compile the FFTW library, and prepare it to be statically linked at compile-time to our HOWFSC application. The steps to build the FFTW library for static linking can be seen in Figure 3.2.

```
$ wget http://www.fftw.org/fftw-3.3.10.tar.gz
$ tar -xzvf fftw-3.3.10.tar.gz # untar the library
$ source setup_for_t1040.sh # host setup script
$ ./fftw-3.3.10/configure \${CONFIGURE}_FLAGS --disable-shared
  --enable-static LDFLAGS="-static"
$ make
```

Figure 3.2: Building the FFTW library for the T1040 processor.

In Figure 3.2, the most important step is sourcing the host setup script before the `configure` command. This makes it so that the Autotools build system recognizes it must create a makefile for a cross-compiled version of the library. By running the `configure` command with the `CONFIGURE_FLAGS` (set, for example, in Figure 3.1), the Makefile is instructed which machine architecture and operating system are on the target, and which machine architecture and operating system are on the “build system” (x86\_64 host desktop

running Ubuntu Linux). This command also instructs the Autotools build system where the sysroot for the target is as well, so the library is aware of the filetree context on the target.

Additionally, the `configure` command is passed the necessary flags to ensure the Autotools build system constructs a statically linked library, disabling shared linking. With this style of build in place, the FFTW library could be built for the T1040 and the LS1046. To build FFTW for the x86\_64 host desktop running linux, the setup script line was omitted. We could then utilize FFTW's FFT by simply including the header with `#include <fftw3.h>`.

Upon cross-compiling the library, we needed to adapt our current optical model's calls to `dfourn` on the wavefront to calls to FFTW. The main way to accomplish this was by copying memory stored in the `dvector` data type to the FFTW data type `fftw_complex` data type. `dvector` in NR stores 2D complex data with a contiguous memory block by storing the matrix in row-major, and with alternating real/imaginary parts as shown in Figure 3.3.

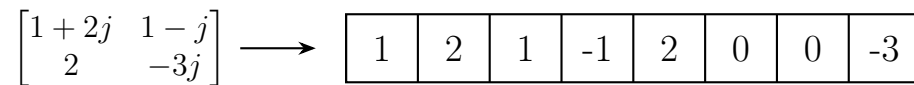


Figure 3.3: How the Numerical Recipes `dvector` data type stores complex matrices in contiguous memory with a `double` at each entry.

The `fftw_complex` data type, on the other hand, uses C's native `_Complex double` data type since our embedded machines support `<complex.h>`. With this convention, `fftw_complex` stores 2D complex matrices in memory in the exact same way as `dvector` since the `_Complex double`'s are stored with real and imaginary parts placed consecutively. The central difference between the two data types are the way they are indexed, with 0- and 1-based indexing being used for `fftw_complex` and `dvector`, respectively. Another subtle difference is in how memory is allocated for `fftw_complex` arrays. `fftw_alloc_complex` pays special attention to aligning memory to special multiples so that single instruction multiple data (SIMD) operations can be executed on the arrays. Thankfully, this still does not affect the fact that the two arrays are stored sequentially in memory starting from the first memory address, so C's `memcpy()` can be utilized to transfer data between data types.

```

1 // nn[1]=N; nn[2]=N;
2 // dfourn(wavefront, nn, 2, 1);
3
4 fftw_plan p;
5 fftw_complex *wavefront_FFTW = fftw_alloc_complex(N*N);
6
7 memcpy(wavefront_FFTW, &(wavefront[1]), sizeof(double) * 2*N*
8       N);
9
10 p = fftw_plan_dft_2d(N, N, wavefront_FFTW, wavefront_FFTW,
11      FFTW_FORWARD, FFTW_ESTIMATE);
12 fftw_execute(p);

```

Figure 3.4: Converting from NR's `dfourn()` call to FFTW.

The transfer from a call to `dfourn()` to an FFTW call, including a swapping of data types, is captured in Figure 3.4. Lines 1 and 2 show what a previous call to `dfourn()`, which accepts the data type to be executed on (`wavefront`), the shape of the array (N by N), the dimension of the transform (2), and the kind of FFT (1 = sign of FFT phase convention). To transfer data types, the memory is copied from the address of the first element of `wavefront` into the zeroth element of `wavefront_FFTW` for the size of the 2D complex array. Then, notice that FFTW's FFT is accomplished in two steps: plan, and execute. At the planning stage, the shape of the input is provided (arguments 1 and 2), the input data (argument 3), the output data (argument 4: same as input to do an in-place transform), the style of transform (`FFTW_FORWARD` sets the same convention as in NR), and the flag `FFTW_ESTIMATE` is provided as well.

All of the heavy lifting for FFTW's speedup is done at the planning stage, so when it is time to execute, the optimized plan for the target machine is ready to run. FFTW includes a few options for planning style. The two that were tried were `FFTW_ESTIMATE` and `FFTW_MEASURE`. Briefly, `FFTW_MEASURE` attempts earnestly to plan the most optimized FFT for the machine, which could result in a very long planning step. `FFTW_ESTIMATE`, on the other hand, returns a plan quickly, even if it is sub-optimal, resulting in a slightly less fast FFT. The `FFTW_ESTIMATE` mode was used for all the FFTs in the `forward()` optical model,

as it was found to strike a nice balance for extra speed. However, in future work, it is worth exploring pre-planning large FFTs with `FFTW_MEASURE`, then using these plans in the form of *wisdom* (FFTW’s technical term) to dispatch extra fast FFTs.

## L-BFGS

For adjoint-based EFC, recall that we need to dispatch a minimizer that employs the L-BFGS algorithm to find the minimum of  $J_{\text{AD-EFC}}$ . For such a responsibility, a similar chain of reasoning can be employed as was done in Section 3.2.2. One is faced with the choice of either writing the routine from scratch for precise control, or incorporating calls to a third-party library into your application. Just as with the FFT, one can save orders of magnitude of development time, and avoid a myriad of implementation-specific details by utilizing a third-party library. L-BFGS is an agnostic framework that accepts functions and gradients of functions, then proceeds to minimize with the approach outlined in Section 2.3. This fact makes a third-party L-BFGS library a very appealing option, because it would require only understanding the API.

The `liblbfgs` library, a C port of a library written by Jorge Nocedal and Naoaki Okazaki, was the library of choice to deploy an L-BFGS optimizer for adjoint-based EFC [54]. This library was chosen for three primary reasons:

1. The authors are authoritative voices on the underlying algorithm.
2. The source code corresponds directly to the algorithms discussed in Nocedal’s *Numerical Optimization*, serving as a nice pedagogical reference [33].
3. The source code is highly readable and well-commented.

Just as was done for FFTW, the `liblbfgs` library needed to be built for cross-compilation to deploy it on the embedded targets. The [GitHub](#) [54] includes instructions on how to build the library, but it does not include how the library could be built for cross-compilation. A

tedious process of trial and error revealed the instructions could be modified as shown in Figure 3.5 for our purposes.

```
# Instructions from liblbfgs
$ ./autogen.sh
$ ./configure
$ make

# Necessary modifications
$ ./autogen.sh
$ source setup_for_t1040.sh # host setup script
$ ./configure $CONFIGURE_FLAGS --disable-shared --enable-static LDFLAGS="-static"
$ make
```

Figure 3.5: Building the LibLBFGS library for the T1040 processor.

Analogous to building the FFTW library, the crucial step was the execution of the host setup script, and a passing off of the necessary environment flags to the Autotools build system. Again, this is achieved by the host setup script setting environment variables like `CC` (which points to the cross-compiler), and sets the proper `sysroot`. The other flags set in the `./configure` line ensure the library is not shared (not dynamically linked), and is built to statically link. Just as with FFTW, the statements in Figure 3.5 prepare the `liblbfgs` library to be statically linked to binaries that are deployed to the T1040.

An analogous process could be done for the LS1046. To build the library for the i7 x86\_64 desktop, the unmodified instructions were utilized in the first section of Figure 3.5.

### 3.3 Implementing Adjoint-Based EFC

From the embedded computer’s perspective, a few things are required to perform a control iteration of adjoint-based EFC:

- **a**: The current state of the DM commands.
- $\hat{\mathbf{E}}_{\text{ab}}$ : The aberrated electric field as estimated from science camera measurements.

- DM Mask: A boolean 2D array which corresponds to which DM actuators will be physically used for control.
- Control Mask: A boolean 2D array which corresponds to the region of the image plane we plan on digging the dark hole in.

In this work, the processor-in-the-loop (PITL) version of adjoint-based EFC is outlined, where a telescope’s optics are simulated in a high-fidelity Zemax / POPPY model in Python, and the embedded hardware consists of either the T1040, or the LS1046 running AD-EFC. The first stage in performing AD-EFC is to package up the data and send them to the embedded processor. The compact optical model in `forward()` was transcribed from `fraunhofer_2dm.py` in the `aefc-vortex` repository by Kian Milani [55]. The full Fresnel model for the telescope’s optics thus can be found at `fresnel_2dm.py` in the `aefc-vortex` repository.

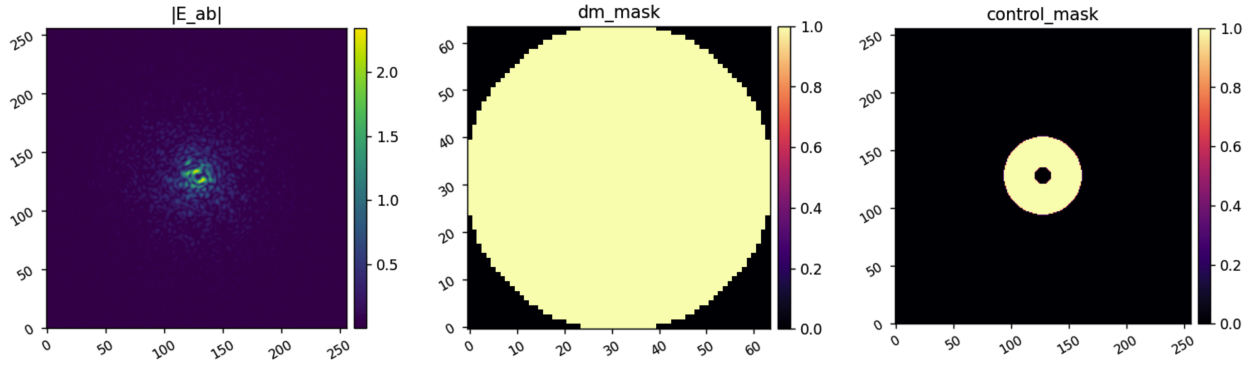
A Python script was constructed that instantiates a full Fresnel model of the telescope and generates a focal-plane wavefront  $\hat{\mathbf{E}}_{ab}$ . Below is an example of how critical data is communicated between the “telescope” in Python, and the embedded computer.

```
# In Python, on host desktop
E_ab = I.calc_wf() # calculates focal-plane wavefront from
                  full Fresnel model
E_ab_contiguous = np.array(E_ab, copy=True) # ensures array
                  is contiguous in memory
E_ab_flat = E_ab_contiguous.ravel().view(np.float64)
send_vector(E_ab_flat)

// In C, on embedded processor
double* E_ab;
size_t E_ab_size = receive_vector(&E_ab, mysocket);
```

This code segment shows the 2D NumPy array `E_ab` with `np.complex128` entries being serialized so that it can be sent over a socket programming interface. On the C side, the

number of bytes received is logged into `E_ab_size`, and the data is placed inside of `E_ab`, which is a `dvector` data type. This same process is repeated for the DM vector `a`, the DM mask, and the control mask.



(a) `E_ab` from the Fresnel model. (b) Example 64 x 64 DM mask. (c) Example control mask.

Figure 3.6: Data sent from the full Fresnel model in Python to the embedded processor.

Once these variables are acquired, the next step is to funnel the current actuator states `a` provided by the full Fresnel model in Python into the separate `dmatrix`'s `dm1_command` and `dm2_command`, then run the `main_optical_model()` to generate  $\mathbf{E}_F(\mathbf{a})$ . The nuances between the `forward()` model and the `main_optical_model()` will be addressed in this Section, but it is only a matter of interface.  $\mathbf{E}_F(\mathbf{a})$  is then placed in variable `E_F`, and remains constant throughout an entire control iteration (note the lack of dependence on  $\delta\mathbf{a}$ ), and will be utilized to compute the cost function in Equation 2.9.

There is just a bit more preparatory work to make sure there is minimal redundant computing inside of `val_and_grad()`. Every time the `forward()` model is run, it utilizes a few mask files:

- `lyot_stop_complex`: Elementwise multiplied with the wavefront to model the wavefront passing the Lyot stop.
- Masks used inside the vortex coronagraph:

- `vortex_mask_size_Nfpm`

- vortex\_mask\_size\_Nmft
- low\_res\_window
- high\_res\_window

Thus, the following mask files are decoded from binary and placed into dvector. With the following variables prepared: `E_ab`, `control_mask`, `dm_mask`, `E_F`, `lyot_stop_complex`, `vortex_mask_size_Nfpm`, `vortex_mask_size_Nmft`, `low_res_window`, `high_res_window`, we are ready to set up our optimization.

The adjoint-based EFC implementation can be found on [GitHub \[56\]](#), which was inspired by the Python implementation in `aefc-vortex`. In Python, minimizing  $J_{AD-EFC}$  involves SciPy's minimizer [57].

```
# In Python
from scipy.optimize import minimize

res = minimize(val_and_grad, jac=True, x0=del_acts0,
args=(M, rmad_vars), method='L-BFGS-B', tol=1e-3,
options=None)
```

There are a few important things to highlight in this Python excerpt we seek to emulate. `val_and_grad()` is a function, that when handed the decision variables  $\delta\mathbf{a}$ , computes both the *value*  $J_{AD-EFC}$ , and *gradient* of the cost function  $\frac{dJ_{AD-EFC}}{d\delta\mathbf{a}}$ . The bulk of the implementation exists inside the `val_and_grad()` function. `M` is an instance of a Fraunhofer model, which has associated methods like `forward()`, which make it easy to call methods of the Fraunhofer model inside of `val_and_grad()`. The crucial variables, like the ones listed at the beginning of this section, are passed to `val_and_grad()` via the `rmad_vars` dictionary. This is where the object-oriented approach to adjoint-based C really differs from the C implementation. In C, we are not afforded the luxury of having objects with associated methods and straightforward hash maps. In C, we utilize our own customizable structure.

In order to compute the value and gradient of the cost function inside of `val_and_grad()` in C, a customized structure is created called `evaluate_context_t`, shown in Figure 3.7.

```

evaluate_context_t context; // Defines the "context"

context.Nact = Nact; // # of actuators along one dimension (e.g.
    64)
context.E_ab = E_ab;
context.E_ab_size = E_ab_size;
context.E_F = E_F;
context.control_mask = control_mask;
context.control_mask_size = control_mask_size;
context.current_acts = a;
context.current_acts_size = a_size; // 2*64*64 for two 64x64
    actuator DMs
context.dm_mask = dm_mask; // 64*64 dvector representing mask to
    a single 64x64 DM
context.dm_mask_counter = dm_mask_counter; // # of active DM
    actuators
context.lyot_stop_complex = lyot_stop_complex;
context.vortex_mask_size_Nfpm = vortex_mask_size_Nfpm;
context.vortex_mask_size_Nmft = vortex_mask_size_Nmft;
context.low_res_window = low_res_window;
context.high_res_window = high_res_window;
context.wavelength = wavelength;
context.wfe = wfe; // Wavefront error

```

Figure 3.7: The `evaluate_context_t` custom structure in C.

The sizes of `E_ab`, `control_mask`, and `current_acts` (**a**) are included in the `context` structure (shown in Figure 3.7) for allocating proper memory sizes later. Also, `dm_mask_counter` counts the number of “active” actuators in the DM mask. Getting into the call to the `lbfgs()` function, we allocate the decision variables  $\delta\mathbf{a}$  as `del_acts`.

```

lbfgsfloatval_t J_ADEFC; // Value of the cost function

lbfgsfloatval_t *del_acts = lbfgs_malloc((int)dm_mask_counter);

```

The `lbfgsfloatval_t` data type is simply dispatched as a double on both of our target processors. The above two lines first declare the cost value variable `J_ADEFC`, and allocates a contiguous array of doubles for the independent variable `del_acts`. A very important

nuance of this implementation is two-fold. Notice that `del_acts` only represents the “active” actuators as indicated by the DM mask (see Figure 3.6b), so we are not optimizing over DM actuators outside of the masked regime. In addition, `lbfgs_malloc()` rounds up the size of memory allocated so that the number of decision variables `n` is a multiple of eight. The `lbfgs` library does this to make memory alignment more seamless to leverage Single Instruction, Multiple Data (SIMD). This will be an important nuance when we are evaluating  $J_{\text{AD-EFC}}$ . The call to `lbfgs()` requires the independent variable to be aligned like this as well.

Upon allocating, the `del_acts` variable is filled with a DM vector corresponding to all zeros. Note, the DM vector for two  $64 \times 64$  actuator DMs will be of length  $2 * 64 * 64 = 8192$ . However, `del_acts` only has a subset of that full length, since we only care about “active” DM actuators. For example, a 8192-length DM vector might have a corresponding `del_acts` of length `n=6600`. This ratio corresponds approximately to the ratio of an inscribed circle’s area to the square’s area ( $\frac{\pi}{4}$ ). This can be seen visually in Figure 3.6b. For the zero-valued initial condition, this nuance doesn’t quite matter, but it will in a moment.

To invoke the L-BFGS minimizer, `lbfgs()` is called as follows:

```
ret = lbfgs(n, del_acts, &J_ADEFC, val_and_grad, NULL, &context,
           &param);
```

`n` is the size of the independent variable (number of “active” actuators), `del_acts` is the array of decision variables, filled with the initial condition of all zeros, `&J_ADEFC` is the address of the cost function value, `val_and_grad` is the function which computes the value and gradient of the cost function, `&context` is the address of the `context` structure, and `&param` is the address to the `param` structure which holds information unimportant for our purposes.

The machinery of the `lbfgs()` function relies fundamentally on `val_and_grad`, which is invoked repeatedly during the minimization. The interface of `lbfgs()` is such that the 6th argument is expecting a void pointer. The first thing we do in `val_and_grad` is cast the context data structure from a null pointer back into the `evaluate_context_t` data type so the important information is available from inside of `val_and_grad`. We next consider how

the value and gradient of the cost function is computed in `val_and_grad`.

## Evaluating $J_{\text{AD-EFC}}$

A critical action that needs to be taken to compute both the value of the cost function and the gradient is the execution of the `forward()` optical model with `del_acts` superimposed on `current_acts`. In our codebase, we delineate between two very similar functions: the `main_optical_model()` and the `forward()` model. They both accomplish the exact same functionality, but `forward()` was developed after `main_optical_model()` to have the extra capability to cache forward variables that will be needed to evaluate the adjoint model. The interface to the forward model can be seen in Figure 3.8

```
/*
 * dm1_command: command to DM1 that specifies which actuators to
 *               push by what amount
 * dm2_command: command to DM2 that specifies which actuators to
 *               push by what amount
 * wavefront: holds a 2D matrix that dynamically changes as it
 *             propagates between each plane
 * E_EP: electric field at the entrance pupil, as generated by the
 *       geometry of the aperture
 * E_DM2P: wavefront before DM2
 * DM1_PHASOR: dm1_command convolved with the Gaussian influence
 *             function to represent shape of the DM1 surface represented in
 *             phasor notation
 * DM2_PHASOR: dm2_command convolved with the Gaussian influence
 *             function to represent shape of the DM2 surface represented in
 *             phasor notation
 * final_size: number of pixels representing one dimension of the
 *             wavefront (e.g. 256)
 * wavelength: wavelength of light we wish to model
 * wfe: wavefront error generated by the optics before DM1 */

int forward(double** dm1_command, double** dm2_command,
            double** wavefront, double** E_EP, double** E_DM2P,
            double** DM1_PHASOR, double** DM2_PHASOR,
            size_t* final_size, double wavelength, double* wfe)
{...}
```

Figure 3.8: The `forward()` model definition header in C.

The forward variables are `E_EP`, `E_DM2P`, `DM1_PHASOR`, `DM2_PHASOR`, and `wavefront` (holds the final wavefront). Only these five variables are needed to compute every adjoint variable. In the previous optical model, which was unconcerned with caching forward variables, an operation looked like the following:

```
double* dm2_phasor = create_phasor(dm2_surf, wavelength, Nsurf);
free_dvector(dm2_phasor, 1, 2*N*N); // dm2_phasor is no longer
needed
```

Instead of the above, the `forward()` model caches forward variables so that they can be returned through the interface:

```
double* dm2_phasor = create_phasor(dm2_surf, wavelength, Nsurf);
*DM2_PHASOR = dvector(1, 2*N*N); // Allocates memory
memcpy(*DM2_PHASOR, dm2_phasor, 2*N*N * sizeof(double)); //
Places dm2_phasor variable
//into the pointer location of DM2_PHASOR
free_dvector(dm2_phasor, 1, 2*N*N); //Now we can free dm2_phasor
```

So, we evaluate `forward()` with the current `del_acts` command (all zeros), yielding every term we need in Equation 2.9, included again for reference below:

$$J_{\text{AD-EFC}}(\delta \mathbf{a}) = \frac{1}{\|\hat{\mathbf{E}}_{\text{ab}}\|^2} (\|\hat{\mathbf{E}}_{\text{ab}} + \mathbf{E}_{\text{F}}(\mathbf{a} + \delta \mathbf{a}) - \mathbf{E}_{\text{F}}(\mathbf{a})\|^2 + \alpha \|\delta \mathbf{a}\|^2).$$

The L2-norm squared of the left-hand term can be computed by simply adding up each entry squared for values that fall inside of the control mask. That is, we only penalize the presence of electric field inside of the region of interest, signalled by the control mask.

### Evaluating $\frac{dJ_{\text{AD-EFC}}}{d\delta \mathbf{a}}$

To build up a picture of the gradient, the adjoint model is executed. While there exist a lot of similarities between the `forward()` model and the adjoint model, there are some operations

in the adjoint model that exist outside the `forward()` paradigm. Having already developed the code for `forward()`, Table 3.2 shows some calls that exist in the forward model and their corresponding adjoint operations.

Table 3.2: Forward vs. Adjoint Calls

Forward call	Adjoint call
<code>fftw_plan_dft_2d(FFTW_FORWARD)</code>	<code>fftw_plan_dft_2d(FFTW_BACKWARD)</code>
<code>angular_spectrum_prop(distance)</code>	<code>angular_spectrum_prop(-distance)</code>
<code>mft_forward()</code>	<code>mft_reverse()</code>
<code>adjust_complex_matrix_size(N1,N2)</code>	<code>adjust_complex_matrix_size(N2,N1)</code>

Some of the adjoint model could be developed by simply piecing together similar calls in the forward model, but some sections could not afford this luxury. For example, the vector vortex coronagraph in the forward model was implemented with an `apply_vortex()` function. Since the `apply_vortex()` function itself is composed of FFTs, MFTs, and Hadamard matrix products, the adjoint vortex model needed to be more granularly implemented. Another location where the adjoint model diverged from the forward model was in the masking of `E_ab`, which is not done in the forward model.

Let  $\delta\mathbf{E} := \hat{\mathbf{E}}_{ab} + \mathbf{E}_F(\mathbf{a} + \delta\mathbf{a}) - \mathbf{E}_F(\mathbf{a})$ , which we call `delE` in the code. To get a sense of what the adjoint model looks like, the first few adjoint variables are shown in Figure 3.9. Figure 3.9 shows how the Boolean (0's and 1's) control mask is used to mask `delE`, and fill the masked entries with zero. Then `delE_masked` is used to compute the first true adjoint variable, `dJ_dE_DMs`. This operation effectively corresponds to differentiating the first term in Equation 2.9. Then, since the final operation in the forward model is `mft_forward()`, we take the adjoint and execute an `mft_reverse()`, which obtains the second adjoint variable, `dJ_dE_LS`.

A few small functions were developed to aid in taking adjoint operations, namely

```
void dconjugate(double* A, int rows, int cols);
double *dimag(double* A, int rows, int cols);
```

```

for(int i=1;i <= control_mask_size; i++){
    if(control_mask[i] > 0.5){
        delE_masked[2*i-1] = delE[2*i-1];
        delE_masked[2*i] = delE[2*i];
    }
    else{
        delE_masked[2*i-1] = 0;
        delE_masked[2*i] = 0;
    }
}

double *dJ_dE_DMs = dvector(1, 2*npsf*npsf);
for(int i=1;i <= E_ab_size; i++){
    dJ_dE_DMs[i] = 2 * delE_masked[i] / E_ab_l2norm;
}

mft_reverse(&dJ_dE_DMs, npsf, psf_pixelscale_ld, npix*LYOT_RATIO,
            npix);
double *dJ_dE_LS = dvector(1, 2*npix*npix);
memcpy(dJ_dE_LS+1, dJ_dE_DMs+1, 2*npix*npix * sizeof(double));

```

Figure 3.9: The first few adjoint variables, `dJ_dE_DMs`, and `dJ_dE_LS`.

`dconjugate()` takes the complex conjugate of a complex matrix stored as a `dvector`, and `dimag()` sets all the real parts equal to zero of a complex matrix stored as a `dvector`. A representative transcription from the Python implementation to the C implementation can be seen in Figure 3.11. This same style of transcription could be used to deploy adjoint operations all the way until `dJ_dA` could be computed, which is the gradient of the cost function with respect to `del_acts`.

## 3.4 Flavors of EFC

As discussed in Chapter 2, there exist two primary ways to do EFC, which we have categorized as “classical” EFC, and adjoint-based EFC. When performing classical EFC, information from previous control iterations can be re-used to continue making progress in digging a dark hole, without having to go through excessive computation. In this work, we introduce the concept of EFC “flavors” to capture this variety in how EFC looks algorithmically.

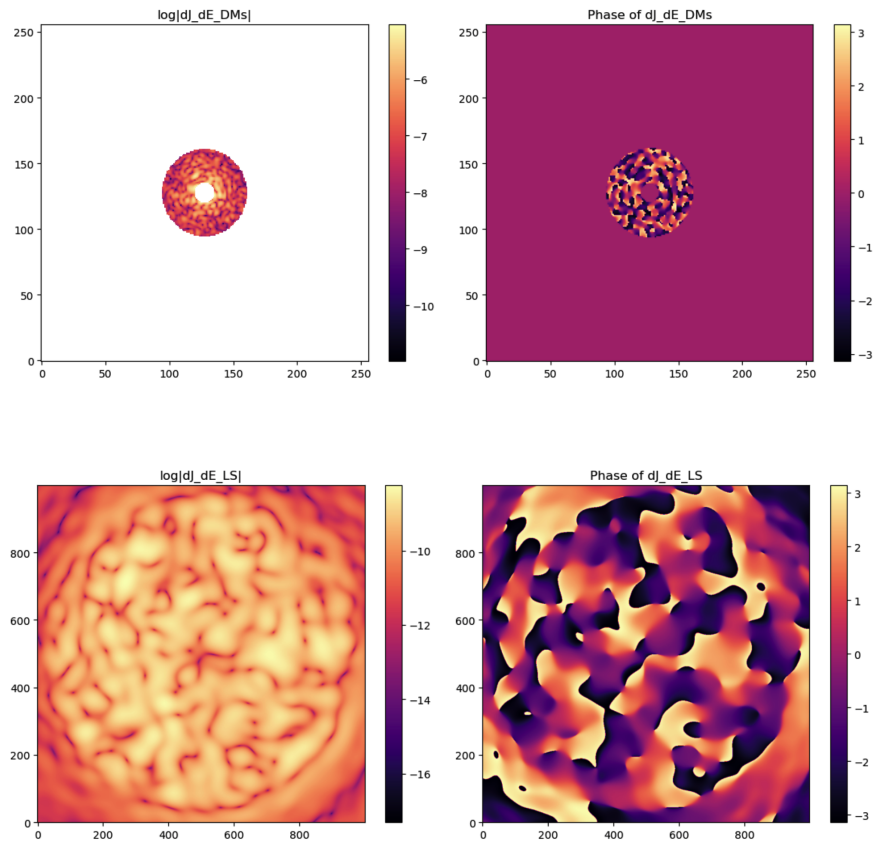


Figure 3.10: The log amplitude and phase of example adjoint variables  $dJ_{dE\_DMs}$ , and  $dJ_{dE\_LS}$ .

```

# In Python
dJ_dS_DM1 = 4*np.pi/M.wavelength * np.imag(dJ_dE_DM1 * E_EP.conj
        ()) * DM1_PHASOR.conj())

// In C
dconjugate(E_EP, npix, npix); // Conjugates in place
dconjugate(DM1_PHASOR, npix, npix); // Conjugates in place
double* dJ_dE_DM1_cropped = adjust_complex_matrix_size(dJ_dE_DM1,
        N,npix);

elementwise_complex_dmatmul(dJ_dE_DM1_cropped, E_EP, npix, npix);
elementwise_complex_dmatmul(dJ_dE_DM1_cropped, DM1_PHASOR, npix,
        npix);

double *temp_imag2 = dimag(dJ_dE_DM1_cropped,npix,npix);
double *dJ_dS_DM1 = dvector(1,2*npix*npix);
for(int i=1;i<=2*npix*npix;i++){
    dJ_dS_DM1[i] = (4*M_PI/wavelength) * temp_imag2[i];
}

```

Figure 3.11: Transcription of conjugating complex matrices from Python to C.

Recall that classical EFC is governed by solving the linear system of equations in Equation 2.6, which we have included again here, using a scalar regularization parameter  $\alpha$  times the identity as in [58],

$$(\text{Re}\{\mathbf{G}^\dagger\mathbf{G}\} + \alpha\mathbb{I})\delta\mathbf{a}^* = -\text{Re}\{\mathbf{G}^\dagger\hat{\mathbf{E}}_{\text{ab}}\}.$$

As mentioned in Section 2.1, we solve Equation 2.6 with three sequential steps: generate  $\mathbf{G}$  via repeated optical modeling, compute the Gram matrix  $\mathbf{G}^\dagger\mathbf{G}$ , add the regularization term  $\alpha\mathbb{I}$  to the Gram matrix, perform the QR decomposition on  $\text{Re}\{\mathbf{G}^\dagger\mathbf{G}\} + \alpha\mathbb{I}$ , then perform QR solve to solve for  $\delta\mathbf{a}^*$ . These sequential steps are contrasted with AD-EFC, which algorithmically simply looks like a call to the L-BFGS optimizer, without any previous information. We can then partition EFC into several flavors:

- EFC from scratch (Jacobian creation + EFC)
  - $2 * (\# \text{ actuators})$  executions of `forward()` executions

- `compute_gram()`
- `do_decomposition()`
- `do_EFC_with_decomp()`
- EFC with updated pre-computed Jacobian
  - `compute_gram()`
  - `do_decomposition()`
  - `do_EFC_with_decomp()`
- EFC with updated regularization ( $\mathbf{G}^\dagger\mathbf{G}$  precomputed)
  - `do_decomposition()`
  - `do_EFC_with_decomp()`
- EFC with precomputed QR decomposition
  - `do_EFC_with_decomp()`
- Adjoint-based EFC
  - 100\* executions of `val_and_grad()`

In a single optimizer iteration of adjoint-based EFC, there are three central algorithmic parts: the first evaluation of `val_and_grad()` to obtain an update to the search direction, a recursive algorithm to approximate the inverse Hessian (with the updated gradient), and a line search to compute an optimal step size. Upon testing the adjoint-based EFC implementation in Python, it was found that there is a consistent overhead of 14 extra `val_and_grad()` calls on top of the total number of L-BFGS iterations. The entirety of the overhead is due to the first iteration, where the 14 `val_and_grad()` calls are used to update the default step size (which begins at `step = 1`). The line search gets executed until one is found that satisfies the Wolfe conditions, and that step size remains for future iterations, requiring no more updating.

AD-EFC was run in Python for a range of tolerances between  $10^{-3}$  and  $10^{-5}$ , and the range of total calls to `val_and_grad()` ranged from 30 to 59. To capture extra margin on top of tested tolerances, a safe margin was 100 calls to the `val_and_grad()` function to complete AD-EFC.

It is worth pointing out that each flavor of EFC is not equivalent, and at some point, using the same Jacobian for many control iterations **may stagnate at some level below the desired contrast level**. So in this work, we time and include all of the flavors for reference. If Jacobian-based EFC is pursued, then the Jacobian at least needs to be generated once or maybe twice throughout dark hole digging. Performing re-regularization must also be done several times throughout the digging of the dark hole.

In Chapter 4, we perform some experiments to directly compare the execution times the EFC flavors outlined in this Section. We also perform some experiments to determine the level of speedup attainable with FFTW in the context of optical modeling, and therefore classical and AD-EFC. The above analysis will ground the reasoning for the performed experiments in context.



# Chapter 4

## Key Experiments

The central aim of this thesis is to reasonably estimate the performance delta via the use of FFTW over a naive Fourier transform, and AD-EFC over traditional EFC. Thus far, we've detailed the crucial algorithms involved in HOWFSC, and carefully outlined the implementation of both the FFTW library, and AD-EFC onto embedded targets in C. What's left is to unveil the realistic speedups attained by timing some of these central HOWFSC algorithms.

Three experiments were devised in such a way as to explicitly illustrate this. The first experiment was devised to benchmark FFTW vs the Numerical Recipes in C `dfourn()` routine to reveal the expected speedups across a range of expected wavefront sizes. The second experiment places this speedup in context of the broader `forward()` optical model, and illustrates what speeding up the FFT means for the runtime composition of `forward()`. Lastly, the third experiment compares the different flavors of EFC with each other, directly showing the possible speedups due to AD-EFC.

All experiments are attempted on three separate processors: the x86\_64 desktop with an i7 chip, the T1040, and the LS1046. The desktop, while obviously will not be flown on a future space telescope, is included in all of these results for reference to see the expected computation time for a reasonably modern CPU-based desktop machine.

Throughout the experiments that include optical modeling and EFC, there is obvious sensitivity to the science case being evaluated. The choice of outer working angle affects the number of pixels in the dark hole, the number of actuators affects the number of controllable modes, and the number of wavelengths affects the ability to do broadband control. We will reserve further commentary on this until those respective sections.

## 4.1 FFTW

### 4.1.1 Isolated FFTs

The aim of this experiment is to reveal precisely the speedup obtained due to swapping `dfourn()` calls (from Numerical Recipes in C) with FFTW. Instead of grounding the calls inside the larger optical model, the calls to both routines are assessed in isolation. In our compact optical model discussed in Section 2.2, there are FFTs of three specific wavefront sizes:

- $1024 \times 1024$
- $2048 \times 2048$
- $4096 \times 4096$

Thus, `dfourn()` and FFTW will be compared across these three sizes. While optical modeling, generally speaking, requires a whole host of explicit modeling choices, FFTs of these three sizes are relatively indispensable from most representative optical models of a future space telescope. Further, while FFT algorithms are becoming increasingly agnostic to the input data size, `dfourn()` and other older algorithms require using data sizes that are powers of two. This strengthens the choice of the above three sizes, as benchmarking the FFT with power-of-two-sized data provides a number that can be easily compared to any other FFT routine. Therefore, evaluating these FFTs in isolation provides insight that

reaches beyond our specific optical modeling choices. We will test across the following three processors:

- The x86\_64 i7 desktop
- The T1040 from NXP
- The LS1046 from NXP

We time each 2D complex matrix size with both `dfourn()` and FFTW, and report the average across 10 trials. As discussed in Section 3.2.3, there are not many knobs to turn in interfacing with the FFTW library, but one of them is the option to include *planning* and *wisdom*. In this experiment, the `FFTW_ESTIMATE` option is used to reduce complexity of having disparate planning and execution stages. However, in future experiments, it is worth looking into how much more the FFT could be sped up by utilizing pre-planning, since the FFT can be asymptotically sped up given a long planning horizon. Thus, in this work we time both the planning and execution summed into one timing result for FFTW.

Table 4.1: Execution times of `dfourn()` vs. FFTW across three processors. The final digit is uncertain in each reported time.

Size	Desktop (s)		T1040 (s)		LS1046 (s)	
	<code>dfourn()</code>	FFTW	<code>dfourn()</code>	FFTW	<code>dfourn()</code>	FFTW
$1024 \times 1024$	0.076	0.019	2.68	0.81	1.12	0.26
$2048 \times 2048$	0.52	0.11	12.3	3.52	6.4	1.24
$4096 \times 4096$	2.74	0.69	65	17.9	32	5.6

The execution times of `dfourn()` along with planning and execution times of FFTW can be seen across the three processors in Table 4.1. Moreover, Figures 4.1 and 4.2 provide a nice at-a-glance view of the speedup attainable via FFTW. As can be clearly seen, replacing the naive FFT with FFTW can make a large difference in execution times. With the naive `dfourn()` FFT, the vortex coronagraph (which includes two  $4096 \times 4096$  FFTs in the optical model) cost 130 seconds on the T1040 and 64 seconds on the LS1046. With FFTW, those times are brought down to approximately 36 seconds and 11 seconds, respectively.

The LS1046 permits even more emphasized speedups, where we can see a factor of 6 speedup for  $4096 \times 4096$  transforms, and a factor of 5 speedup for  $2048 \times 2048$  transforms. On the LS1046, all eight FFTs in optical modeling cost a small total of 17.2 seconds. This has pretty sweeping implications for the optical model. Previously, FFTs made up a large proportion of the compute time due to optical modeling on both the T1040 and the LS1046, but with FFTW, the execution of the optical model becomes completely dominated by matrix multiplication.

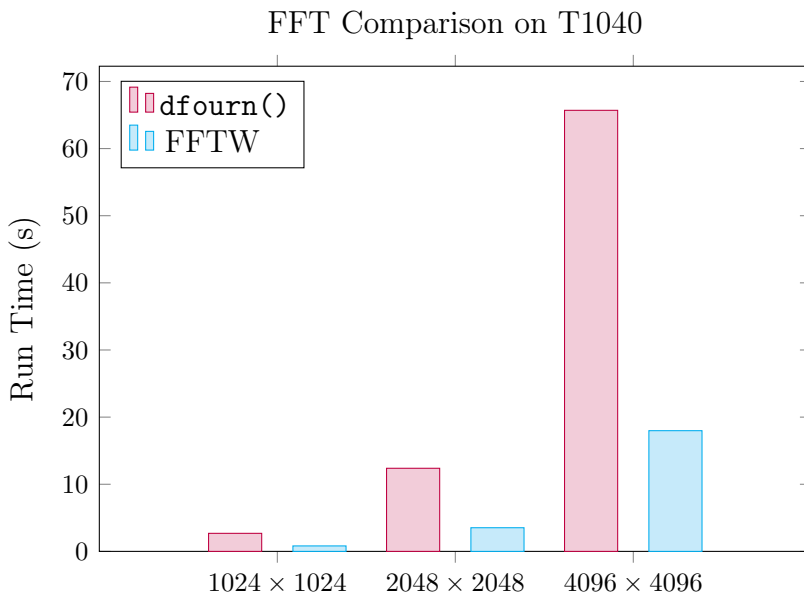


Figure 4.1: Average run times of `dfourn()` vs. FFTW plan + execute on the T1040 processor for complex-valued matrices of different sizes.

### 4.1.2 FFTs in the Optical Model

Following the isolated FFT experiment, we aim to ground those speedups in the larger context of optical modeling. The aim of this experiment is to reveal the performance delta in the `forward()` model between using `dfourn()` and FFTW. This experiment instantiates the speedups due to FFTW in the context of our specific optical model. For this experiment, we utilize two masked  $64 \times 64$  DMs. The size of the PSF at the science camera does not affect the results here, as optical modeling is being run in isolation (without being used for control

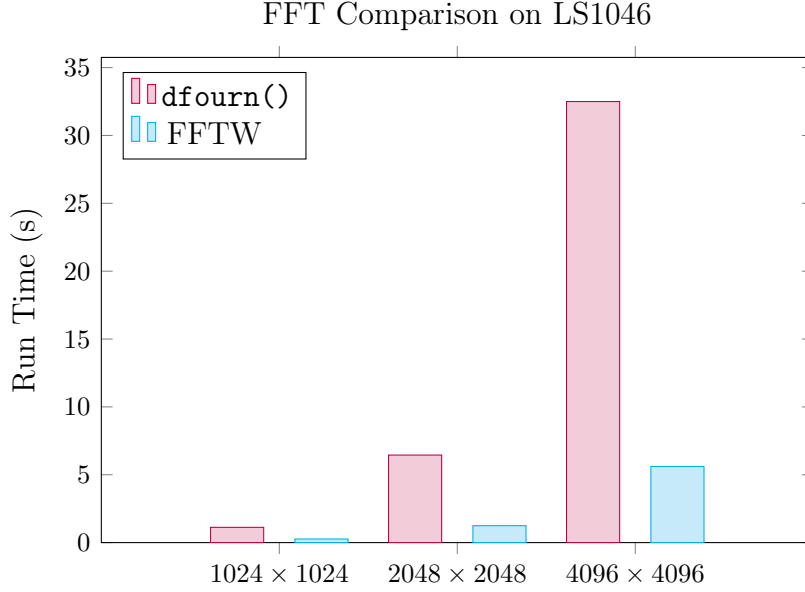


Figure 4.2: Average run times of `dfourn()` vs. FFTW plan + execute on the LS1046 processor for complex-valued matrices of different sizes.

or estimation). The optical model utilized here is exactly as described in Section 2.2. To perform the test, 10 executions of the optical model were completed for two cases:

- All FFTs in the optical model are `dfourn()` from NR in C
- All FFTs in the optical model include both planning and execution of FFTW

This test was conducted across the same three processors as the last experiment: the x86\_64 i7 desktop, the T1040, and the LS1046. To briefly revisit our modeling assumptions, we chose to use two  $64 \times 64$  actuator DMs for this test as a “representative” case between the two bounding cases of HabEx and LUVOIR. It is worth highlighting that even though more actuators may be required on the HWO to effectively control high-order modes, the computation time of optical modeling is not very sensitive to the choice of number of actuators.

In optical modeling, the number of actuators ( $N_{\text{act}}$ ) controls the square size of the two DM commands  $\mathbf{a}_1(x, y)$ , and  $\mathbf{a}_2(x, y)$ . But, as will be shown, very few operations are sensitive to  $N_{\text{act}}$ . As shown in the forward DM model in Table 2.3a, the DM command is immediately brought into Fourier space by a matrix Fourier transform. The matrix Fourier transform

operation looks like  $\mathbf{M}_x \mathbf{a}_i(x, y) \mathbf{M}_y$ , where the sizes of  $\mathbf{M}_x$  and  $\mathbf{M}_y$  in this work are  $N_{\text{surf}} \times N_{\text{act}}$  and  $N_{\text{act}} \times N_{\text{surf}}$ , respectively, where  $N_{\text{surf}} = 1024$ . Complexity analysis yields  $\mathcal{O}(N_{\text{surf}}^2 N_{\text{act}})$  time complexity for the first MFT in the forward DM model, which is linear in  $N_{\text{act}}$ , and requires a small number of FLOPs relative to other sections of the optical model. For reference, holding  $N_{\text{surf}}$  at 1024,  $\mathcal{O}(N_{\text{surf}}^2 N_{\text{act}})$  time complexity for  $N_{\text{act}} = 64$  requires approximately 67 MFLOPs, whereas for  $N_{\text{act}} = 96$ , requires approximately 100 MFLOPs. For reference, a single MFT inside the vortex coronagraph (in which there are two) is on the order of 15 GLOPs.

Further, all forward DM operations to follow are determined by the modeling choice of  $N_{\text{surf}}$  and not  $N_{\text{act}}$ . Thus, achieving timing results for optical modeling using  $N_{\text{act}} = 64$  gleans insight that extends to greater actuator counts. The timing results for optical modeling can be seen exactly in Table 4.2 and visualized in Figure 4.3.

<b>Optical Model Run Time (s)</b>		
	<code>dfourn()</code>	FFTW
Desktop	$64 \pm 2$	$52.7 \pm 0.5$
T1040	$461.8 \pm 0.3$	$299.6 \pm 0.3$
LS1046	$137.2 \pm 0.6$	$42.5 \pm 0.2$

Table 4.2: Time it takes to execute the `forward()` optical model with all calls either to `dfourn()` or FFTW on three different processors.

The LS1046 exhibited a 3x speedup when using FFTW over `dfourn()`, showcasing some potentially architecture-specific advantages of ARM over the T1040’s e5500 PowerPC microarchitecture. Nonetheless, the T1040 exhibited a 1.5x speedup in optical modeling, which is still a sizeable performance boost. Comparing these results to the isolated FFTW test, we can see that the relative proportion of optical modeling allocated to FFTs has been greatly reduced via the use of FFTW.

With the documented performance boost in FFTs, the operations inside the vortex coronagraph now comprise the vast proportion of time needed to execute `forward()`. On the LS1046, the vortex coronagraph still includes two five second FFTs (Figure 4.1), but the rest is almost entirely allocated to the bulky complex matrix multiplies inside the MFTs to finely

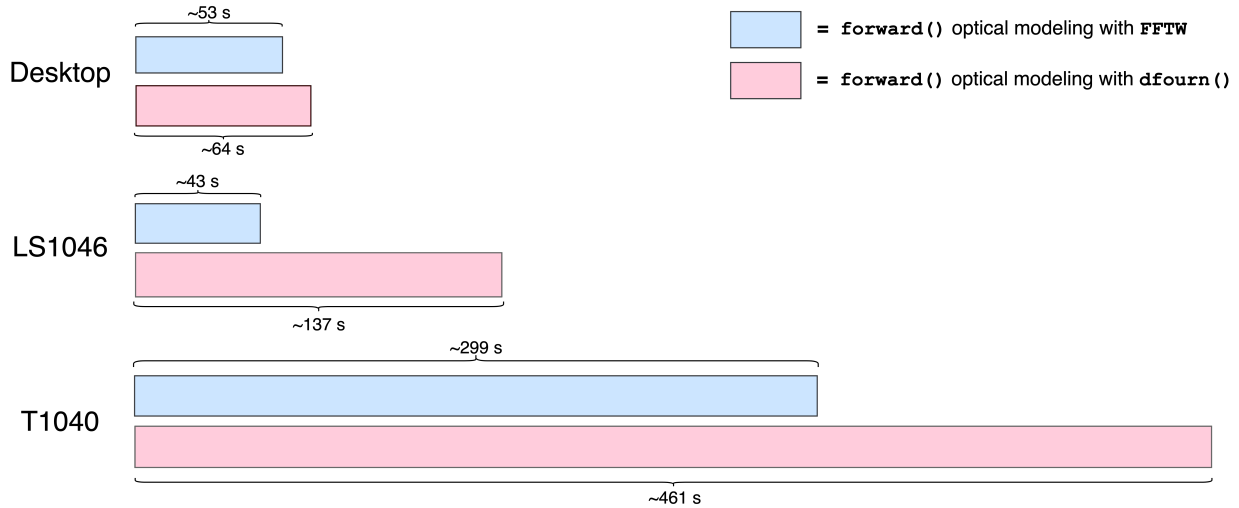


Figure 4.3: Visualization of the run time required to execute the `forward()` optical model with all calls either to `dfourn()` or `FFTW` compared across three processors.

sample the singularity. All things considered, the optical model is now completely dominated by complex matrix multiplications inside the vortex, and Hadamard matrix multiplies both inside and outside the vortex.

With the aspiration of driving the amount of time necessary to perform optical modeling down to zero, a few things should be investigated to follow on to this work. The first of which includes speeding up the FFT even more by utilizing the `FFTW`'s pre-planning and wisdom.

Anecdotally, in personal benchmarking, it was found that by replacing the `FFTW_ESTIMATE` flag with `FFTW_MEASURE`, a  $1024 \times 1024$  FFT took 0.18 s, a  $2048 \times 2048$  FFT took 0.79 s, and a  $4096 \times 4096$  FFT took 3.49 s to execute. Of course, these times only include the time it took to execute, and not the time it took to plan, which is longer using `FFTW_MEASURE`, which generates a more optimized FFT for your machine. For reference, planning for the above three times took 5.7 s, 14.4 s and 25.5 s, respectively. Thus, anytime that optical modeling is needed to be executed iteratively, and plans can be re-used, then it is worth trying to employ a strategy to pre-plan all of the FFTs in the optical model, all 8 FFTs would take less than 11 s total on the LS1046. The overhead due to planning would be quickly amortized over a few optical model executions. This was not formally included in our results, because it would

have required a different software architecture to re-use previous plans to execute FFTs on future optical modeling iterations with new data.

Nonetheless, with FFTW utilized for all FFTs in optical modeling, the proportional change in which algorithm dominates lends itself to an updated sensitivity analysis. Since the optical model is now dominated by the vortex operations, the total execution time is very sensitive to the wavefront sizes we use inside of the vortex. Due to the prevalence of large matrix multiplies and  $4096 \times 4096$  FFTs inside the vortex, it is worth conducting a rigorous analysis on how small percentage changes in the number of pixels inside the vortex affect the fidelity of the final PSF. For our specific optical model,  $N_{\text{MFT}} = 1200$  and  $N_{\text{pix}} = 1000$ , which are utilized to conduct two MFTs with time complexity  $(N_{\text{pix}} \times N_{\text{MFT}}^2) + (N_{\text{pix}}^2 \times N_{\text{MFT}})$ . This is where the bulk of computation lies. Thus, a simple change in both of these numbers can cause a large difference in operations count, due to the quadratic scaling of both  $N_{\text{MFT}}$  and  $N_{\text{pix}}$ . For example, decreasing  $N_{\text{MFT}}$  by 20% yields a 30% reduction in total operations count for the two MFTs.

## 4.2 Adjoint-Based EFC in C

The aim of this experiment is to compare AD-EFC with other “flavors” of EFC (as described in Section 3.4). The specific focus is to reveal the performance delta between AD-EFC and other flavors of EFC. The same “representative” testing case was utilized for this experiment, which requires a few more assumptions about telescope parameters.

Two  $64 \times 64$  actuator DMs are utilized, masked in a similar fashion to Figure 3.6b, yielding a total of 6,600 actuators ( $N_{\text{acts}} = 6,600$ ). While we showed in the previous section that the sensitivity of the isolated optical model with respect to  $N_{\text{acts}}$  was small, the choice of  $N_{\text{acts}}$  directly drives the complexity of all flavors of EFC. For the Jacobian-based flavors,  $N_{\text{acts}}$  determines the number of columns in the Jacobian matrix, which directly affects the computation of  $\mathbf{G}^\dagger \mathbf{G}$ , and the decomposition to follow. For AD-EFC,  $N_{\text{acts}}$  determines the

number of independent variables we are optimizing over. This determines the length of the gradient that needs to be stored from past iterations which are utilized to compute the approximate inverse Hessian.

The number of pixels in the dark hole was decided to be  $N_{\text{psf}} = 16,060$ .  $N_{\text{psf}}$  drives the number of rows in the Jacobian matrix, and thus directly affects the decomposition and other following operations in Jacobian-based EFC. AD-EFC on the other hand is almost completely agnostic to  $N_{\text{psf}}$ . In AD-EFC,  $N_{\text{psf}}$  only shows up in the masking step in the evaluation of the cost function, determining how many values to sum in the L2-norm squared of the left hand term of Equation 2.9, which isn't expensive. The final MFT of the forward model (and first MFT in the adjoint model) is a computation that clearly involves the number of pixels at the science camera, but the code architecture generates a  $256 \times 256$  wavefront, regardless of the number of pixels in the dark hole (determined by inner and outer working angles). Thus, the final MFT in the forward model is insensitive to pixel count in the dark hole.

Lastly, the number of wavelengths was decided to be one, yielding single-wavelength EFC. We recognize that broadband EFC would likely be required for the desired science cases on the HWO, but single wavelength EFC was selected due to the Jacobian becoming too large to store in memory on the embedded devices. Since the operations count of Jacobian generation, computing  $\mathbf{G}^\dagger \mathbf{G}$ , and the number of required executions of the optical model in AD-EFC are linear in wavelength, it is easy to extrapolate these results and comparisons to broadband EFC.

Recall from the discussion in Section 3.4 that the flavors of EFC are derivative of the following lower-level routines:

- `compute_gram()`
- `do_decomposition()`
- `do_EFC_with_decomp()`
- `forward()`

- `val_and_grad()`

Thus, for this experiment, the above routines are only executed once, due to long computation times of  $\mathbf{G}^\dagger \mathbf{G}$ . Any lack of determinism in algorithm runtime due to operating system processes is typically negligible on our embedded processors, as the operating system is not doing much else besides executing the program at hand.

Algorithm	Processor Runtime (s)	
	Desktop	LS1046
<code>compute_gram()</code>	580	550
<code>do_decomposition()</code>	1100	1700
<code>do_EFC_with_decomp()</code>	0.7	1.8
<code>forward()</code>	53	43
<code>val_and_grad()</code>	129	124

Table 4.3: Runtimes for key EFC sub-routines on the x86\_64 i7 desktop workstation and the LS1046 embedded processor.

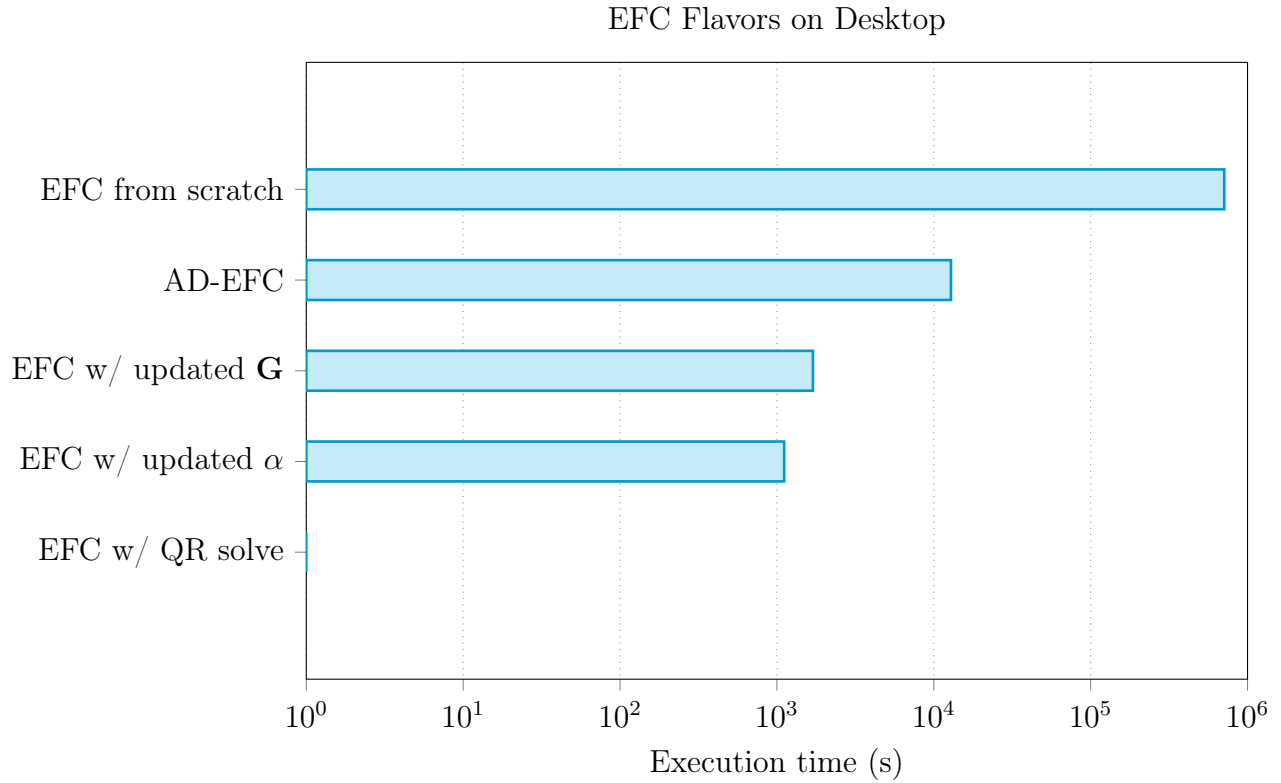


Figure 4.4: Execution-time comparison of several EFC flavors on a desktop workstation.

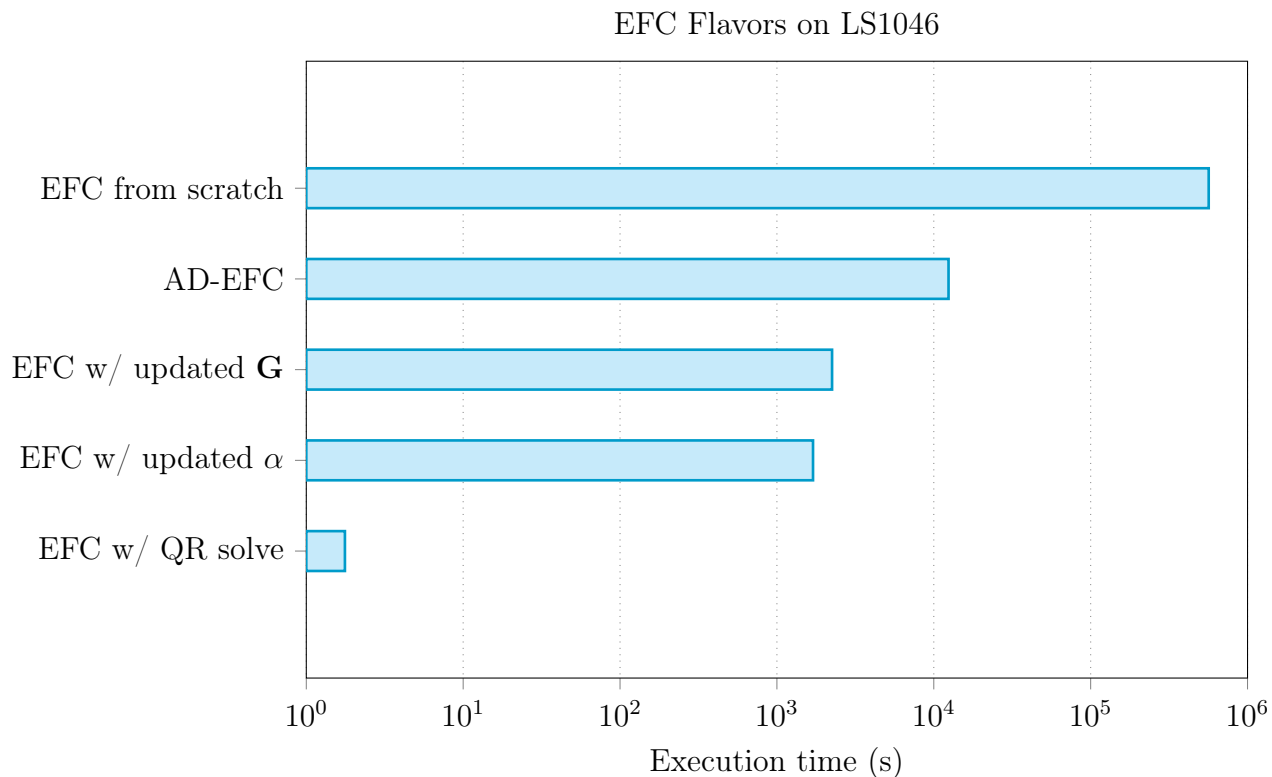


Figure 4.5: Execution-time comparison of several EFC flavors on the LS1046.

Each lower-level algorithm was executed on each available processor (shown in Table 4.3) to derive the EFC runtimes shown in Figures 4.4 and 4.5. Unfortunately, results could not be generated on the T1040. At test time, the T1040 would terminate executing the program after `compute_gram()`, which is thought to be caused by a lack of memory and odd behavior due to a sustained period of maximum CPU usage.

We recognize that this comparison is not necessarily “apples-to-apples”, as there is nuance hidden inside the EFC flavors that require pre-computed information. The most fair comparison can be made between Jacobian-based EFC from scratch and AD-EFC. On the LS1046, AD-EFC is 46x faster, with a total runtime of 3.45 hours compared to 157 hours for EFC from scratch.

If the Jacobian is pre-computed, on the other hand, then other flavors of EFC are much faster than AD-EFC. On the LS1046, EFC with a pre-computed Jacobian is 5x faster than AD-EFC, and EFC with an updated regularization term (which requires another QR

decomposition and solve) is 7.3x faster than AD-EFC. Lastly, EFC with QR solve has a negligible runtime.

**If 100 second latency is required to avoid the accumulation of too much wavefront drift, then the three flavors of EFC that utilize previous information are approximately an order of magnitude away from that threshold. AD-EFC still needs to be sped up by a factor of 124x if the aim is achieve a latency of 100 seconds.**

Let us revisit both the algorithms and the specific modeling assumptions that went into these estimates. EFC from scratch is almost entirely driven by the time required to execute `forward()`, and the number of times the optical model needs to run (which is  $2N_{\text{acts}}$ ). Both EFC with a pre-computed Jacobian and with an updated regularization are entirely dominated by QR decomposition. The time it takes to do QR solve once a decomposition is obtained is negligible. AD-EFC, like EFC from scratch, is dominated by the time it takes to execute `forward()`, but in the context of `val_and_grad()`, which requires the execution of the adjoint model as well. We further assumed 100 total executions of `val_and_grad()`, which was a conservative upper bound on the typical range we've seen with the execution of AD-EFC in Python.

Zooming into optical modeling, our specific optical model includes eight FFTs of varying sizes, all of which are executed with FFTW (with the `FFTW_ESTIMATE` flag. The matrix multiplies inside of the matrix Fourier transform are executed using a recursive divide-and-conquer algorithm, where the matrices are tiled until the tiles of each matrix reasonably fit into L1 cache. Given FFTs and matrix multiplies are the bulk of the optical model, these timing results are representative of the first-order speedups attainable. Again, these timing results use one set of optical modeling design choices, detailed in [19]. The most important modeling choices were the selections of  $N_{\text{MFT}} = 1200$  and  $N_{\text{pix}} = 1000$ , which the size of FFTs and matrix multiplies in the vortex.

To conclude the discussion of the achieved results, we will comment on where the next

lowest hanging fruit is for the speedup of embedded HOWFSC algorithms. These results beg the question, if these results show critical HOWFSC algorithms with first-order speedups, then what is a realistic expectation of the maximum speed achievable, and where can it come from? Further, we will discuss what these results mean for HWO.



# Chapter 5

## Conclusions and Future Work

In this thesis, we provided detailed implementations of third-party libraries like `FFTW` and `libbfgs` to accelerate both optical modeling and EFC. In Chapter 3, we selected a programming language for embedded development, and showed how `FFTW` could be deployed to an embedded platform. Moreover, we outlined an embedded implementation of the Jacobianless AD-EFC method to elucidate the performance delta over Jacobian-based EFC. We also detailed for the first time what an implementation of adjoint-based EFC could look like in C, down to the specifics of which third-party library to select, and how to build and deploy it on an embedded platform.

In Chapter 4, we showed the speedups attainable by using `FFTW` over a naive implementation of the FFT from Numerical Recipes in C. We also grounded those speedups in the context of optical modeling, and showed that the optical model could be sped up by a factor of approximately 3.2 on the LS1046 processor, and a factor of 1.5 on the T1040 processor. We focused on speeding up the optical model, since it underpins both classical EFC, and AD-EFC. We proceeded to give high-fidelity estimates of the different flavors of EFC, including an estimate of adjoint-based EFC derived from 100 executions of `val_and_grad()` in C.

By using Figure 4.5, we can directly pluck off factors of speedup necessary to achieve a certain controller bandwidth. Even with the use of `FFTW` and a recursive divide-and-conquer

tiled matrix multiply strategy, AD-EFC would still need a 124x speedup to run with a latency of 100 seconds. The question remains, what are the shortest paths to achieve lower latency execution?

In this work, we remained within a radiation-hardened, CPU-based paradigm in C. We can begin to answer this question within this CPU-based paradigm, and then extend out. Assuming a static state of wavefront control algorithms over the next decade(s) while HWO is in development, we can start by trying to make the optical model as fast as possible. To build on this thesis, the FFT can still be sped up through the use of pre-planning and *wisdom* in FFTW. This could potentially bring the execution time of the eight FFTs in the optical model down to less than 10 seconds. But, given the sizeable speedup of FFTs in this work, FFTs are no longer the bottleneck routine inside of optical modeling. Regardless of how fast the FFT is, there are still very large matrix multiplies and Hadamard products in the vortex coronagraph, and throughout the rest of the optical model. While a recursive tiling matrix multiply method was used in this work, there is quite a bit more room for speeding up matrix multiplication.

The most direct path forward to speeding up matrix multiplies is through the use of an OpenBLAS implementation [59]. BLAS implementations involve linking to the OpenBLAS library, and allow for leveraging inherent parallelism in linear algebra routines, like matrix multiplication. The speedups attainable from OpenBLAS rely on cache-blocking, SIMD vectorization, and parallelization with OpenMP. This is where the selection of a processor that supports an OpenBLAS implementation and vectorized instructions, like AVX512, becomes extremely important.

With an optimized FFT and an optimized matrix multiplication routine, optical modeling execution time will begin to approach its theoretical limit. At the end of the day, the theoretical maximum performance of a CPU-based processor is determined by its clock frequency, the number of instructions per cycle, FLOPs per instruction, memory controller speed, and total RAM. These numbers set the bound on how quickly we could possibly run

HOWFSC algorithms.

Outside of optical modeling, we also saw in Figure 4.5 that given a pre-computed Jacobian, the other flavors of EFC are only one order of magnitude away from that 100-second latency mark. The same set of reasoning applies. If OpenBLAS can be implemented, then these flavors of EFC, which consist of canonical linear algebra operations, could be sped up by a significant factor. Yet, with this route, the Jacobian,  $\mathbf{G}^\dagger\mathbf{G}$ , or the decomposition of  $\text{Re}\{\mathbf{G}^\dagger\mathbf{G}\} + \alpha\mathbb{I}$ , must still somehow be stored in memory of the embedded processor.

Previous work has allocated some attention to using a pre-launch Jacobian, followed with Expectation Maximization to update the Jacobian with measurements on orbit [20]. This is worth implementing on embedded processors to accurately estimate how long EM takes to execute. Within the current set of algorithms, and bounds set by the performance expectation of radiation-hardened processors, deploying the HOWFSC algorithms necessary to directly image exoplanets is going to be difficult.

Considering other processors, NASA’s HPSC is worth deploying HOWFSC algorithms to, as it boasts dual quad-core application processors which consist of RISC-V x280 CPUs [60]. The HPSC features several radiation-hardened by design approaches, as well as 512-bit length vector registers, making it capable of SIMD and CPU-based parallelism. Further, the HPSC also supports OpenCL, which allows for parallelizing across several processors.

Further, given the vast amount of attention and capital allocated to COTS processors, COTS processors in a serviceable scheme are worth considering. Whether it is N-modular redundancy, error correction, or other single event effect-resistant techniques, there exists a possibility of leveraging powerful terrestrial computing platforms. The restriction to radiation-hardened processors severely constrains the clock frequency and overall computational throughput of the device. In order to serve a decade(s)-long mission lifetime, COTS processors (like GPUs) with natural robustness to TID are worth testing and exploring.

Given all the previous considerations, other work has provided a systems-level analysis to determine if HWO will be able to execute HOWFSC algorithms at a quick enough rate

[12]. Given the current throughput rates of radiation-hardened processors, it seems unlikely that HWO will be able to perform HOWFSC onboard. If computation onboard is infeasible, a ground-in-the-loop scheme with low-latency communication could potentially achieve the control bandwidth required.

The aim of this work was to estimate performance deltas between naive algorithms and first-order accelerations through third-party libraries and alternate algorithms like AD-EFC. To build on this work, it is worth pushing the realistic processors that could fly on HWO to their theoretical limit. Using the most optimized algorithm implementations, the computational aspect of directly imaging exoplanets on HWO could be de-risked, providing the designers with a blueprint to achieve its mission.

# Appendix A

## Automatic Differentiation

Automatic differentiation, which is the *precise* numerical computation of the derivative, can be seen most easily through the view of a function as a computational graph. For example, the function  $\mathbf{y} = \mathbf{A}\mathbf{x}_1 + \mathbf{x}_2$  can be decomposed into intermediate representations  $\mathbf{x}_3 = \mathbf{A}\mathbf{x}_1$ , and  $\mathbf{x}_4 = \mathbf{x}_3 + \mathbf{x}_2$ . The computational graph can be constructed as in Figure A.1.

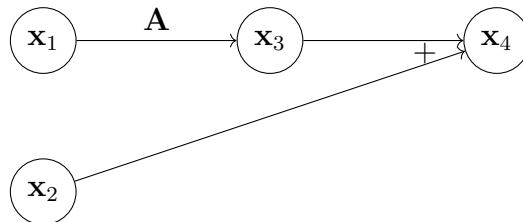


Figure A.1: Computational graph for  $\mathbf{y} = \mathbf{A}\mathbf{x}_1 + \mathbf{x}_2$ .

The inputs are placed on the left side of the graph, and elementary operations on the inputs as well as intermediate variables create the final output on the right ( $\mathbf{y} = \mathbf{x}_4$ ). To construct the derivative with reverse-mode algorithmic differentiation (RMAD), we iteratively build up the gradient information by differentiating each *child* node with respect to its *parents*. To begin, we obtain:

$$\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} = 1, \quad \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_2} = 1.$$

Since  $\mathbf{x}_2$  is one of our inputs, and there are no other children of  $\mathbf{x}_2$ , we already have one entry of the gradient. Next, we propagate  $\partial\mathbf{x}_4/\partial\mathbf{x}_3$  backward:

$$\frac{\partial\mathbf{x}_3}{\partial\mathbf{x}_1} = \frac{\partial}{\partial\mathbf{x}_1}(\mathbf{A}\mathbf{x}_1) = \mathbf{A},$$

$$\frac{\partial\mathbf{x}_4}{\partial\mathbf{x}_1} = \frac{\partial\mathbf{x}_3}{\partial\mathbf{x}_1} \cdot \frac{\partial\mathbf{x}_4}{\partial\mathbf{x}_3} = \mathbf{A} \cdot 1 = \mathbf{A}.$$

Since  $\mathbf{x}_1$  is our other input, we can completely form the gradient with  $\partial\mathbf{x}_4/\partial\mathbf{x}_1$  and  $\partial\mathbf{x}_4/\partial\mathbf{x}_2$ .

# References

- [1] National Academies of Sciences Engineering and Medicine. *Pathways to Discovery in Astronomy and Astrophysics for the 2020s*. Washington, DC: The National Academies Press, 2021. ISBN: 978-0-309-46734-6. DOI: [10.17226/26141](https://doi.org/10.17226/26141). URL: <https://nap.nationalacademies.org/catalog/26141/pathways-to-discovery-in-astronomy-and-astrophysics-for-the-2020s>.
- [2] Las Cumbres Observatory. *Transit Method*. 2025. URL: <https://lco.global/spacebook/exoplanets/transit-method/>.
- [3] R. Juanola-Parramon, N. T. Zimmerman, L. Pueyo, M. Bolcar, G. Ruane, J. Krist, and T. Groff. “The LUVOIR Extreme Coronagraph for Living Planetary Systems (ECLIPS) II. Performance evaluation, aberration sensitivity analysis and exoplanet detection simulations”. In: *Techniques and Instrumentation for Detection of Exoplanets IX*. Vol. 11117. SPIE. 2019, pp. 21–36.
- [4] A. Give'on, B. Kern, S. Shaklan, D. C. Moody, and L. Pueyo. “Broadband wavefront correction algorithm for high-contrast imaging systems”. In: *Astronomical Adaptive Optics Systems and Applications III*. Vol. 6691. SPIE. 2007, pp. 63–73.
- [5] L. Pogorelyuk, C. Haughwout, N. Belsten, E. Cady, and K. Cahoy. “Computational complexities of image plane algorithms for high contrast imaging in space telescopes”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 8.4 (2022), pp. 049003–049003.

- [6] N. Belsten, B. Eickert, K. Milani, S. Rao, E. Douglas, L. Pogorelyuk, and K. Cahoy. “Selecting Space Processors for High Order Wavefront Control Adaptive Optics Systems”. In: *2024 IEEE Space Computing Conference (SCC)*. 2024, pp. 104–115. DOI: [10.1109/SCC61854.2024.00018](https://doi.org/10.1109/SCC61854.2024.00018).
- [7] N. Belsten, K. Milani, L. Pogorelyuk, B. Eickert, S. Rao, E. S. Douglas, and K. Cahoy. “Evaluating embedded hardware for high-order wavefront sensing and control”. In: *Techniques and Instrumentation for Detection of Exoplanets XI*. Vol. 12680. SPIE. 2023, pp. 546–562.
- [8] A. S. Keys, J. H. Adams, D. O. Frazier, M. C. Patrick, M. D. Watson, M. A. Johnson, J. D. Cressler, and E. A. Kolawa. “Radiation hardened electronics for space environments (RHESE)”. In: *AIAA Space 2007: Avionics, Surface and Mission Operations Logistics Session*. 2007.
- [9] BAE Systems. *RAD5545 SoC based single board computer*. 2025. URL: <https://www.baesystems.com/en-us/our-company/inc-businesses/electronic-systems/product-sites/space-products-and-processing/radiation-hardened-electronics>.
- [10] I. Poberezhskiy et al. “Roman space telescope coronagraph: engineering design and operating concept”. In: *Space Telescopes and Instrumentation 2020: Optical, Infrared, and Millimeter Wave*. Ed. by M. Lystrup, M. D. Perrin, N. Batalha, N. Siegler, and E. C. Tong. Vol. 11443. International Society for Optics and Photonics. SPIE, 2021, p. 114431V. DOI: [10.1117/12.2563480](https://doi.org/10.1117/12.2563480). URL: <https://doi.org/10.1117/12.2563480>.
- [11] A. Rosh-Gorsky, A. Coon, D. Beck, R. D’Onofrio, Q. Binney, I. Queen, A. Barney, R. Longton, A. C. Long, P. Gouker, et al. “3D printing of composite radiation shielding for broad spectrum protection of electronic systems”. In: *Advanced Materials* 36.33 (2024), p. 2403822.
- [12] N. Belsten. “Embedded Computing for Wavefront Control on Future Space Telescopes”. 2025, in prep.

- [13] W. S. Slater, N. P. Tiwari, T. M. Lovelly, and J. K. Mee. “Total Ionizing Dose Radiation Testing of NVIDIA Jetson Nano GPUs”. In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 2020, pp. 1–3. DOI: [10.1109/HPEC43674.2020.9286222](https://doi.org/10.1109/HPEC43674.2020.9286222).
- [14] J. Goodwill, C. Wilson, and J. MacKinnon. *Current AI Technology in Space*. Book Chapter. NASA Goddard Space Flight Center, 2024. URL: <https://ntrs.nasa.gov/citations/20240001139>.
- [15] T. M. Lovelly. “Comparative analysis of space-grade processors”. PhD thesis. University of Florida, 2017.
- [16] NXP Semiconductors. *QorIQ® T1040 and T1020 Multicore Communications Processors*. 2025. URL: <https://www.nxp.com/products/T1040>.
- [17] NXP Semiconductors. *QorIQ LS1046A, LS1026A Data Sheet*. 2020. URL: <https://www.nxp.com/docs/en/data-sheet/LS1046A.pdf>.
- [18] P. J. Bordé and W. A. Traub. “High-contrast imaging from space: speckle nulling in a low-aberration regime”. In: *The Astrophysical Journal* 638.1 (2006), p. 488.
- [19] K. Milani, E. Douglas, L. Pogorelyuk, K. Cahoy, N. Belsten, B. Eickert, C. Mendillo, and S. Rao. “Optical modeling for the evaluation of HOWFSC on embedded processors”. In: *Adaptive Optics Systems IX*. Ed. by K. J. Jackson, D. Schmidt, and E. Vernet. Vol. 13097. International Society for Optics and Photonics. SPIE, 2024, 130976B. DOI: [10.1117/12.3019056](https://doi.org/10.1117/12.3019056). URL: <https://doi.org/10.1117/12.3019056>.
- [20] H. Sun, N. J. Kasdin, and R. Vanderbei. “Identification and adaptive control of a high-contrast focal plane wavefront correction system”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 4.4 (2018), pp. 049006–049006.
- [21] K. Miller, O. Guyon, and J. Males. “Spatial linear dark field control: stabilizing deep contrast for exoplanet imaging using bright speckles”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 3.4 (2017), p. 049002. DOI: [10.1117/1.JATIS.3.4.049002](https://doi.org/10.1117/1.JATIS.3.4.049002). URL: <https://doi.org/10.1117/1.JATIS.3.4.049002>.

- [22] S. D. Will, T. D. Groff, and J. R. Fienup. “Jacobian-free coronagraphic wavefront control using nonlinear optimization”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 7.1 (Feb. 18, 2021). ISSN: 2329-4124. DOI: [10.1117/1.JATIS.7.1.019002](https://doi.org/10.1117/1.JATIS.7.1.019002). URL: <https://www.spiedigitallibrary.org/journals/Journal-of-Astronomical-Telescopes-Instruments-and-Systems/volume-7/issue-01/019002/Jacobian-free-coronagraphic-wavefront-control-using-nonlinear-optimization/10.1117/1.JATIS.7.1.019002.full> (visited on 10/02/2023).
- [23] A. Give'on, B. D. Kern, and S. Shaklan. “Pair-wise, deformable mirror, image plane-based diversity electric field estimation for high contrast coronagraphy”. In: *Techniques and Instrumentation for Detection of Exoplanets V*. Vol. 8151. SPIE. 2011, pp. 376–385.
- [24] A. J. E. Riggs, N. J. Kasdin, and T. D. Groff. “Recursive starlight and bias estimation for high-contrast imaging with an extended Kalman filter”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 2.1 (2016), p. 011017. DOI: [10.1117/1.JATIS.2.1.011017](https://doi.org/10.1117/1.JATIS.2.1.011017). URL: <https://doi.org/10.1117/1.JATIS.2.1.011017>.
- [25] L. Pogorelyuk and N. J. Kasdin. “Dark Hole Maintenance and A Posteriori Intensity Estimation in the Presence of Speckle Drift in a High-contrast Space Coronagraph”. In: *The Astrophysical Journal* 873.1 (Mar. 2019), p. 95. DOI: [10.3847/1538-4357/ab0461](https://doi.org/10.3847/1538-4357/ab0461). URL: <https://dx.doi.org/10.3847/1538-4357/ab0461>.
- [26] P. Baudoz, A. Boccaletti, J. Baudrand, and D. Rouan. “The Self-Coherent Camera: a new tool for planet detection”. In: *Proceedings of the International Astronomical Union* 1.C200 (2005), pp. 553–558.
- [27] S. D. Will et al. “High-order coronagraphic wavefront control with algorithmic differentiation: first experimental demonstration”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 9.4 (Dec. 19, 2023). ISSN: 2329-4124. DOI: [10.1117/1.JATIS.9.4.045004](https://doi.org/10.1117/1.JATIS.9.4.045004). URL: <https://www.spiedigitallibrary.org/journals/Journal-of-Astronomical-Telescopes-Instruments-and-Systems/volume-9/issue-04/045004/High->

- order-coronagraphic-wavefront-control-with-algorithmic-differentiation--first/10.1117/1.JATIS.9.4.045004.full.
- [28] J.-B. Hiriart-Urruty and C. Lemaréchal. *Fundamentals of Convex Analysis*. 2001. ISBN: 1618-2685. DOI: [10.1007/978-3-642-56468-0\\_4](https://doi.org/10.1007/978-3-642-56468-0_4).
- [29] S. A. Teukolsky, B. P. Flannery, W. Press, and W. Vetterling. “Numerical recipes in C”. In: *SMR* 693.1 (1992), pp. 59–70.
- [30] J. Krist, S. Martin, G. Kuan, B. Mennesson, G. Ruane, N. Saini, J. Trauger, J. Breckinridge, D. Mawet, P. Stahl, et al. “Numerical modeling of the Habex coronagraph”. In: *Techniques and Instrumentation for Detection of Exoplanets IX*. Vol. 11117. SPIE. 2019, pp. 85–100.
- [31] J. W. Cooley and J. W. Tukey. “An Algorithm for the Machine Calculation of Complex Fourier Series”. In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: <http://www.jstor.org/stable/2003354>.
- [32] B. Eickert, N. Belsten, K. Milani, L. Pogorelyuk, S. Rao, E. S. Douglas, and K. Cahoy. “Evaluating HOWFSC algorithm implementations on embedded space processors”. In: *Space Telescopes and Instrumentation 2024: Optical, Infrared, and Millimeter Wave*. Ed. by L. E. Coyle, S. Matsuura, and M. D. Perrin. Vol. 13092. International Society for Optics and Photonics. SPIE, 2024, 130926G. DOI: [10.1117/12.3019146](https://doi.org/10.1117/12.3019146). URL: <https://doi.org/10.1117/12.3019146>.
- [33] J. Nocedal and S. J. Wright. *Numerical Optimization*. 2e. New York, NY, USA: Springer, 2006.
- [34] K. Milani, S. D. Will, K. Van Gorkom, E. S. Douglas, J. N. Ashcraft, and K. Cahoy. “Demonstrations of adjoint electric field conjugation for a vortex coronagraph”. In: *Journal of Astronomical Telescopes, Instruments, and Systems*.

- [35] J. J. Moré and D. J. Thuente. “Line search algorithms with guaranteed sufficient decrease”. In: *ACM Trans. Math. Softw.* 20.3 (Sept. 1994), pp. 286–307. ISSN: 0098-3500. DOI: [10.1145/192115.192132](https://doi.org/10.1145/192115.192132). URL: <https://doi.org/10.1145/192115.192132>.
- [36] P. Wolfe. “Convergence Conditions for Ascent Methods”. In: *SIAM Review* 11.2 (1969), pp. 226–235. DOI: [10.1137/1011036](https://doi.org/10.1137/1011036). eprint: <https://doi.org/10.1137/1011036>. URL: <https://doi.org/10.1137/1011036>.
- [37] Freescale Semiconductor, Inc. *Freescale Linux SDK for QorIQ Processors*. 2012. URL: <https://www.nxp.com/docs/en/supporting-information/QorIQ-SDK-1.2-IC-RevA.pdf>.
- [38] *Yocto Project*. URL: <https://www.yoctoproject.org/>.
- [39] *The Yocto Project Application Development Toolkit (ADT) User’s Guide*. URL: <https://docs.yoctoproject.org/1.1.2/adt-manual/adt-manual.html#using-an-existing-toolchain-tarball>.
- [40] A. Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf).
- [41] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [42] G. Holzmann. “The power of 10: rules for developing safety-critical code”. In: *Computer* 39.6 (2006), pp. 95–99. DOI: [10.1109/MC.2006.212](https://doi.org/10.1109/MC.2006.212).
- [43] M. Sudwoj. “Rust programming language in the high-performance computing environment”. B.S. thesis. ETH Zurich, 2020.

- [44] Bjarne Stroustrup. *RAII*. 2024. URL: [https://en.cppreference.com/w/cpp/language/raii#:~:text=Resource%20Acquisition%20Is%20Initialization%20or,in%20limited%20supply\)%20to%20the](https://en.cppreference.com/w/cpp/language/raii#:~:text=Resource%20Acquisition%20Is%20Initialization%20or,in%20limited%20supply)%20to%20the).
- [45] The Rust Community. *The rustc book: Platform Support*. 2025. URL: <https://doc.rust-lang.org/nightly/rustc/platform-support.html>.
- [46] W. Powell. *High Performance Spaceflight Computing (HPSC) for Earth Science Applications*. NASA Technical Reports Server, Document ID 20240006570. Presentation, Earth Science Technology Forum (ESTF), Crystal City, VA, 11–12 June 2024. May 2024. URL: <https://ntrs.nasa.gov/api/citations/20240006570/downloads/Powell-ESTF-HPSC-2024.pdf>.
- [47] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. “Julia: A Fresh Approach to Numerical Computing”. In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: [10.1137/141000671](https://doi.org/10.1137/141000671). URL: <https://doi.org/10.1137/141000671>.
- [48] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. “Julia: A fast dynamic language for technical computing”. In: *arXiv preprint arXiv:1209.5145* (2012).
- [49] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88.
- [50] T. Short and Contributors. *StaticCompiler.jl*. Version 0.7.2. Julia Package. June 16, 2024. URL: <https://github.com/tshort/StaticCompiler.jl>.
- [51] S. Terasaki and Contributors. *jlcross.jl*. Julia-Embedded, Nov. 3, 2024. URL: <https://github.com/Julia-Embedded/jlcross>.
- [52] JuliaBerry Organization. *JuliaBerry: Julia for the Raspberry Pi*. 2025. URL: <https://juliaberry.github.io>.
- [53] M. Frigo and S. G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation”, pp. 216–231. DOI: [10.1109/JPROC.2004.840301](https://doi.org/10.1109/JPROC.2004.840301).

- [54] J. Nocedal and Contributors. *liblbfgs*. chokkan, June 19, 2023. URL: <https://github.com/chokkan/liblbfgs>.
- [55] K. Milani. *aefc-vortex*. Apr. 28, 2025. URL: [https://github.com/kian1377/aefc-vortex/tree/main/aefc\\_vortex](https://github.com/kian1377/aefc-vortex/tree/main/aefc_vortex).
- [56] N. Belsten, B. Eickert, and K. Milani. *embedded<sub>h</sub>owfsc*. May 15, 2025. URL: [https://github.com/MIT-STARLab/embedded\\_howfsc](https://github.com/MIT-STARLab/embedded_howfsc).
- [57] P. Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).
- [58] D. Marx, B.-J. Seo, B. Kern, E. Sidick, B. Nemati, and I. Poberzhskiy. “Electric field conjugation in the presence of model uncertainty”. In: *Techniques and Instrumentation for Detection of Exoplanets VIII*. Ed. by S. Shaklan. Vol. 10400. International Society for Optics and Photonics. SPIE, 2017, 104000P. DOI: [10.1117/12.2274541](https://doi.org/10.1117/12.2274541). URL: <https://doi.org/10.1117/12.2274541>.
- [59] Xianyi, Zhang and Contributors. *OpenBLAS: An optimized BLAS library*. 2025. URL: <http://www.openmathlib.org/OpenBLAS/>.
- [60] T.-M. Gauthier. *NASA’s High Performance Spaceflight Computer*. White Paper. National Aeronautics and Space Administration, June 2024. URL: <https://www.nasa.gov/wp-content/uploads/2024/07/hpsc-white-paper-tmg-26jun2024-final.pdf?emrc=66904bca260a4>.