Mathematical Software for Multiobjective Optimization Problems

Tyler H. Chang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Layne T. Watson, Chair
Sharath Raghvendra
Ali R. Butt
Christopher A. Beattie
Michael W. Trosset

May 13, 2020
Blacksburg, Virginia

Keywords: multiobjective optimization, multivariate interpolation, Delaunay triangulations, mathematical software, numerical algorithms

Mathematical Software for Multiobjective Optimization Problems

Tyler H. Chang

(ABSTRACT)

In this thesis, two distinct problems in data-driven computational science are considered. The main problem of interest is the multiobjective optimization problem, where the tradeoff surface (called the Pareto front) between multiple conflicting objectives must be approximated in order to identify designs that balance real-world tradeoffs. In order to solve multiobjective optimization problems that are derived from computationally expensive blackbox functions, such as engineering design optimization problems, several methodologies are combined, including surrogate modeling, trust region methods, and adaptive weighting. The result is a numerical software package that finds approximately Pareto optimal solutions that are evenly distributed across the Pareto front, using minimal cost function evaluations. The second problem of interest is the closely related problem of multivariate interpolation, where an unknown response surface representing an underlying phenomenon is approximated by finding a function that exactly matches available data. To solve the interpolation problem, a novel algorithm is proposed for computing only a sparse subset of the elements in the Delaunay triangulation, as needed to compute the Delaunay interpolant. For high-dimensional data, this reduces the time and space complexity of Delaunay interpolation from exponential time to polynomial time in practice. For each of the above problems, both serial and parallel implementations are described. Additionally, both solutions are demonstrated on real-world problems in computer system performance modeling.

Mathematical Software for Multiobjective Optimization Problems

Tyler H. Chang

(GENERAL AUDIENCE ABSTRACT)

Science and engineering are full of multiobjective tradeoff problems. For example, a portfolio manager may seek to build a financial portfolio with low risk, high return rates, and minimal transaction fees; an aircraft engineer may seek a design that maximizes lift, minimizes drag force, and minimizes aircraft weight; a chemist may seek a catalyst with low viscosity, low production costs, and high effective yield; or a computational scientist may seek to fit a numerical model that minimizes the fit error while also minimizing a regularization term that leverages domain knowledge. Often, these criteria are conflicting, meaning that improved performance by one criterion must be at the expense of decreased performance in another criterion. The solution to a multiobjective optimization problem allows decision makers to balance the inherent tradeoff between conflicting objectives. A related problem is the multivariate interpolation problem, where the goal is to predict the outcome of an event based on a database of past observations, while exactly matching all observations in that database. Multivariate interpolation problems are equally as prevalent and impactful as multiobjective optimization problems. For example, a pharmaceutical company may seek a prediction for the costs and effects of a proposed drug; an aerospace engineer may seek a prediction for the lift and drag of a new aircraft design; or a search engine may seek a prediction for the classification of an unlabeled image. Delaunay interpolation offers a unique solution to this problem, backed by decades of rigorous theory and analytical error bounds, but does not scale to high-dimensional "big data" problems. In this thesis, novel algorithms and software are proposed for solving both of these extremely difficult problems.

Dedication

For Sydney Adkins

Acknowledgments

First, I would like to thank my advisor, Dr. Watson, whose rigorous attention to detail, depthful knowledge in mathematics and computer science, and unique perspective based on years of real-world experience has helped make this work something that I am truly proud of. Furthermore, his generosity with his time and advice has made me a better researcher and helped me to achieve my goals in graduate school and beyond.

I would also like to thank my committee members: Drs. Butt and Raghvendra from the Dept. of Computer Science, Dr. Beattie from the Dept. of Mathematics, Dr. Hong from the Dept. of Statistics, and Dr. Trosset from the University of Indiana. They have been simultaneously encouraging, while ensuring that I've considered all options from the literature.

Another round of thanks to all of my collaborators: the VarSys group at Virginia Tech for sharing advice in our weekly meetings; Dr. Larson from Argonne National Laboratory who was my mentor during my DOE SCGSR award; and Dr. Thacker from Winthrop University who has shared his wisdom based on past experiences in producing mathematical software.

Additionally, I would like to thank all of my labmates from Torgerson 2000 and my many friends from the CS@VT community for making this journey a truly enjoyable experience. A special thanks to Thomas Lux who, in addition to becoming one of my closest friends, has consistently been someone to share ideas with and whose contributions to this work are innumerable.

I would like to give a heartfelt thanks to my family — my parents, Ginny and Bob, and my sister, Sophie — for their much-needed support and understanding as I've devoted myself to this work for the last four years. Also, I would like to thank my surrogate family, Hope and Jason, who've made me feel at home, and who even helped me move to Illinois and back, out of the kindness in their hearts.

Finally, I have to give a very special and well-deserved thanks to my better half, Sydney, who has been with me since the beginning of this journey and been my biggest supporter through every struggle and success. I could not have done this without her support, and I am looking forward to the next chapter of our life together.

This work was supported in part by the National Science Foundation under Grant No. CNS-1565314 and CNS-1838271. Any opinions, findings, and conclusions or recommendations expressed in this material are my own and do not necessarily reflect the views of the National Science Foundation.

This work was also supported in part by the U.S. Dept. of Energy (DOE), Office of Science Graduate Student Research (SCGSR) program. The SCGSR program is administered by the Oak Ridge Institute for Science and Education (ORISE), which is managed by ORAU under contract number DE-SC0014664. All opinions in this thesis are my own and do not necessarily reflect the policies and views of the DOE, ORAU, or ORISE. I gratefully acknowledge the computing resources provided on Bebop, a high-performance computing system operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Related and Motivating Problems . . . 2

1.3 Common Notation and Terminology . . . 3

1.4 Contributions . . . 4

2 Multivariate Interpolation and DELAUNAYSPARSE 6

2.1 Introduction . . . 6

2.2 Literature Review . . . 6

2.3 Interpolation via the Delaunay Triangulation . . . 8

2.3.1 The DELAUNAYSPARSE Algorithm . . . 9

2.3.2 Relationship to Linear Programming . . . 11

2.4 Computational Aspects . . . 11

2.4.1 Growing the Seed Simplex . . . 12

2.4.2 The Visibility Walk . . . 15

2.4.3 Flipping Across a Facet . . . 16

2.5 Algorithm Analysis . . . 18

2.5.1 Robustness and Backward Stability . . . 18

2.5.2 Time Complexity . . . 19

2.5.3 Approximation Accuracy . . . 19

2.6 Serial Implementation . . . 20

2.6.1 Handling Multiple Interpolation Points . . . 20

2.6.2 Extrapolation . . . 21
2.6.3 Data Scaling . . . 22
2.6.4 Memory Usage . . . 23
2.6.5 The Cost of Robustness and Correctness . . . 23
2.7 Parallel Implementation . . . 24
2.7.1 Level 1 Parallelism . . . 25
2.7.2 Level 2 Parallelism . . . 25
2.8 Performance . . . 26
2.8.1 Serial Performance . . . 27
2.8.2 Parallel Performance . . . 28

3 Multiobjective Optimization and VTMOP **31**
3.1 Introduction . . . 31
3.2 Literature Review . . . 32
3.2.1 Multiobjective Optimization Techniques and Algorithms . . . 33
3.2.2 Multiobjective Optimization Software . . . 35
3.3 A Framework for Multiobjective Optimization . . . 36
3.3.1 Computing the Local Trust Region & Choosing Adaptive Weights . . . 37
3.3.2 Searching the Design Space and Trust Regions . . . 43
3.3.3 Solving the Surrogate Optimization Problem . . . 44
3.3.4 Evaluating Candidate Designs and Iterating . . . 47
3.4 Solvers for Multiobjective Optimization Problems . . . 47
3.4.1 The Return-To-Caller Interface . . . 47
3.4.2 The Driver Subroutine . . . 48
3.4.3 Parallel Implementation . . . 48
3.4.4 Use Cases for VTMOP . . . 50
3.5 Integration with libEnsemble . . . 50
3.5.1 The libEnsemble Library . . . 51
3.5.2 Implementing VTMOP as a libEnsemble Generator . . . 51

3.6 Approximation Performance . . . 53
3.6.1 Description of Test Problems . . . 54
3.6.2 Algorithm Variations and Parameter Settings . . . 55
3.6.3 Results . . . 55
3.7 Parallel Performance . . . 57
3.7.1 Description of Hardware and Problem Expense . . . 57
3.7.2 Runtime Performance Results . . . 58

4 Applications **62**
4.1 The Variability Problem . . . 62
4.2 Literature Review . . . 62
4.3 Using DELAUNAYSPARSE to Predict I/O Throughput Variance . . . 63
4.3.1 Data Collection . . . 63
4.3.2 Model Evaluation . . . 65
4.3.3 Other Applications . . . 65
4.4 Tuning the HPL Benchmark Solver with VTMOP . . . 67
4.4.1 The HPL Benchmark Problem . . . 67
4.4.2 The HPL Solver . . . 68
4.4.3 Integrating HPL as a VTMOP Objective Function . . . 70
4.4.4 Tuning HPL on a Single Node with VTMOP . . . 71
4.4.5 Tuning HPL on Multiple Nodes . . . 74

5 Conclusion and Future Work **79**
5.1 Conclusion . . . 79
5.2 Future Work . . . 79
5.2.1 The Future of DELAUNAYSPARSE . . . 80
5.2.2 The Future of VTMOP . . . 80
5.3 The Big Picture . . . 81

Bibliography **83**

Appendices **93**

Appendix A Proofs **94**

A.1 Proof of Lemma 2.9 94

A.2 Definition and Proof of Acyclicity 94

A.3 Proof of Lemma 2.14 95

A.4 Proof of Theorem 3.8 95

Appendix B Code for DELAUNAYSPARSE **95**

B.1 delsparse.f90 95

Appendix C Code for VTMOP **126**

C.1 vtmop.f90 126

C.2 bVTdirect.f90 174

C.3 Code for VTMOP with libEnsemble 196

C.3.1 vtmop_initializer.f90 196

C.3.2 vtmop_generator.f90 197

C.3.3 vtmop.py 199

List of Figures

2.1 A triangulation in $\mathbb{R}^2$ (left) and the Delaunay triangulation (right). 8

2.2 A two-dimensional visualization of the property described in Remark 2.3. The solid line segment is a facet Γ, $\mathcal{H}$ is the halfspace above the dashed line, and $\{x^{(1)}, x^{(2)}, x^{(3)}\}$ is a sequence satisfying $x^{(1)} \in \mathcal{B}^{(2)}$ and $x^{(2)} \in \mathcal{B}^{(3)}$. 9

2.3 Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 1000$ and $m = 1024$. 28

2.4 Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 20,000$ and $m = 64$. 29

2.5 Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 50-dimensional problem, with $n = 500$ and $m = 64$. 29

3.1 An illustration of one of the drawbacks of weighted sum scalarization. The dashed line depicts a nonconvex Pareto front for $p = 2$, The solid line depicts the supporting hyperplane (in this case a line) whose normal is w, and the solution to (3.2) is marked at the point of tangency between the hyperplane and the Pareto front. Note that there is no vector w that can attain solutions in the shaded region. 34

3.2 CPU utilization over time for a single run of VTMOP using the search via DIRECT with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation. 60

3.3 CPU utilization over time for a single run of VTMOP using the Latin hypercube search with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation. Note that the performance valleys correspond to iteration tasks, which were done serially in this experiment. 61

3.4 CPU utilization over time for a single run of VTMOP using `libEnsemble` with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation. Note that the performance valleys correspond to iteration tasks, which were done serially in this experiment. 61

4.1 Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(1)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated. . . 66

4.2 Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(2)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated. . . 66

4.3 Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(3)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data. 67

4.4 Tradeoff curve between mean throughput and throughput standard deviation in 40 runs of `HPL` when using the approximate Pareto optimal configurations on a single node with $N = 10,000$. 72

4.5 Histograms of observed throughputs when running `HPL` on a single node with $N = 10,000$, for the 12 highest mean throughput configurations. Distributions are ordered from left to right, top to bottom in ascending order by mean/standard deviation. 74

4.6 Tradeoff curve between mean throughput and throughput standard deviation in 30 runs of `HPL` when using the approximate Pareto optimal configurations on four nodes with $N = 20,000$. 75

4.7 Histograms of observed throughputs when running `HPL` on four nodes with $N = 20,000$, $NB = 128$. The throughput distributions for the 12 highest mean throughput configurations are from Table 4.5 (left) and the two non-dominated configurations are from Table 4.6 (right). 78

List of Tables

2.1 Average number (with a sample size of 20) of Delaunay simplices computed in a simplex walk from the simplex built off the nearest neighbor to y for n uniform randomly generated points in d dimensions. 19

2.2 Time and space requirements (as reported by Boissonnat et al. [14]) for computing the complete Delaunay triangulation using Quickhull and the Delaunay Graph algorithm. Entries containing the word "swap" indicate that the process exceeded RAM limitations. 26

2.3 Serial runtimes (in seconds) for computing the Delaunay interpolant at a single interpolation point. Values shown represent the average over 20 independent trials with n pseudo-randomly generated data points in the d-dimensional unit hypercube. 27

3.1 Number of solutions found by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list. 56

3.2 RMSE reported by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list. 56

3.3 Discrepancy reported by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list. 57

3.4 Runtime performance summary (CPU time / wall time) for four variations of VTMOP. Times are given in seconds. 59

4.1 System and `IOzone` configuration settings that are used to build the variability map with DELAUNAYSPARSE. Note, the frequency 3.001 GHz results from overclocking. 64

4.2 Tuning Parameters for `HPL` 69

4.3 Bounds for Adjustable Inputs When Tuning `HPL` 70

4.4 Approximate Pareto Optimal Set for Single-Node Runs of HPL with $N = 10,000$ 73

4.5 Approximate Pareto Optimal Set for Four-Node Runs of HPL with $N = 20,000$, $NB = 128$ 76

4.6 Additional Evaluations of HPL with $N = 20,000$, $NB = 128$ 77

Chapter 1

Introduction

"*Life is about decisions. Decisions, no matter if made by a group or an individual, usually involve several conflicting objectives. The observation that real world problems have to be solved optimally according to criteria, which prohibit an "ideal" solution — optimal for each decision-maker under each of the criteria considered — has led to the development of multicriteria optimization.*"
— Matthias Ehrgott (Preface to [46])

1.1 Motivation

Multiobjective optimization problems (MOPs) are ubiquitous in science and engineering. For example, a portfolio manager may seek to build a financial portfolio with low risk, high return rates, and minimal transaction fees; an aircraft engineer may seek a design that maximizes lift, minimizes drag force, and minimizes aircraft weight; a chemist may seek a catalyst with low viscosity, low production costs, and high effective yield; or a computational scientist may seek to fit a numerical model that minimizes the "fit error" (often referred to as the loss function) while also minimizing a regularization term that leverages domain knowledge. Often, these objectives are conflicting, meaning that improved performance by one criterion must be at the expense of decreased performance in another criterion. Therefore, the solution to a MOP must allow decision makers to balance the inherent tradeoff between objectives.

The most general and also the most difficult class of MOPs is the class of computationally expensive blackbox MOPs. In this context, the terminology *computationally expensive* implies that simply evaluating each of the above criteria requires a significant amount of time and occupies significant computational resources. The word *blackbox* implies that the problem offers no additional information (e.g., partial derivatives) that can be exploited. In many cases, computationally expensive blackbox MOPs are the result of massive numerical simulations (e.g., aerospace design optimization). When asked about the existence of production quality software for solving computationally expensive blackbox MOPs with an arbitrary number of objective criteria, a prominent name in the field (whose identity is left anonymous here) replied "maybe in 100 years," indicating that this class of problems is simply too prohibitively expensive to solve with modern technology.

This thesis explores the algorithms and technology required to produce scalable software for

solving computationally expensive blackbox MOPs with many objectives. The results include a production quality Fortran solver for blackbox MOPs of moderate computational expense, and significant advancement toward solutions to extremely expensive blackbox MOPs.

1.2 Related and Motivating Problems

Revisiting the terminology "computationally expensive," what constitutes significant time and resources is subjective. In less extreme cases, a matter of seconds could be considered a significant amount of time and a single computing node or workstation could be considered significant resources. In these cases, it may be possible to evaluate the objective criteria tens of thousands of times, and existing software such as genetic algorithms [38] offer a reasonable solution. In extreme cases such as computational fluid dynamics analyses, the objective criteria could take hours or even days to evaluate and occupy hundreds of CPU cores during evaluation [12, 16]. In these cases, a budget of less than 100 evaluations of the objective criteria is often imposed. The work in this thesis addresses blackbox problems that are in between these two extremes of computational expense and is best suited for problems that allow for approximately 1000–2000 objective criteria evaluations. To help quantify this amount of expense, a specific motivating problem is given.

The motivating problem for this thesis is the problem of managing computational performance variability in high-performance computing (HPC) systems. The VarSys project at Virginia Polytechnic Institute and State University (Virginia Tech) is a NSF funded study to predict and manage the effects of computational performance variability in HPC systems [17]. From a mathematics/algorithms perspective, there are two components to this study (1) predicting the effects of performance variability and (2) managing these effects by controlling system or application configuration parameters. The second goal (managing the effects of performance variability) is a natural application for multiobjective optimization since there is an inherent tradeoff between mean computational performance and computational performance variability [81]. In order to pose this problem as a MOP, one methodology is to perform numerous runs of a benchmark problem with each proposed system or application configuration and then use the performance results reported by the benchmark to compute an estimate for the mean computational performance and computational performance variability [18]. For the benchmark problems considered in this paper, each such batch of benchmark evaluations requires several minutes, on average, and occupies whatever resources are allocated to solving the benchmark problem. Therefore, for the rest of this thesis, the term "computationally expensive" could be interpreted as "requiring on the order of several minutes in compute time."

Toward the first goal for the VarSys project (predicting the effects of performance variability), another problem that is closely related to the MOP is the multivariate interpolation problem (MIP). For the MIP, the goal is to predict the outcome of an event based on a database of past observations, while exactly matching all observations in that database. MIPs are

equally as prevalent and impactful as MOPs. For example, a pharmaceutical company may seek a prediction for the costs and effects of a proposed drug; an aerospace engineer may seek a prediction for the lift and drag of a new aircraft design; or a search engine may seek a prediction for the classification of an unlabeled image. In recent years, these problems have been dominated by the usage of machine learning techniques such as neural network regression. However, classical techniques such as Delaunay interpolation offer decades of rigorous theory and analytical error bounds. In addition to solving the MOP, this thesis proposes and implements a novel algorithm to "cheat" the curse of dimensionality, which had previously made Delaunay interpolation computationally prohibitive for modern data science applications.

1.3 Common Notation and Terminology

The material in this thesis is necessarily dense in notation. In an effort to improve the readability, the following common notation is used throughout. Each chapter may also introduce additional notation that is specific to itself.

For any two real n-tuples u and v, the following partial ordering relations are used.

- Write $u < v$ if u is componentwise strictly less than v.
- Write $u \leq v$ if u is componentwise less than or equal to v, and this inequality is strict in at least one component.
- Write $u \leqq v$ if u is componentwise less than or equal to v, allowing for equality in all components.

The MOP (and MIP) will consider minimizing (and approximating) a vector valued function $F : \mathbb{R}^d \to \mathbb{R}^p$. When discussing MOPs, it is important to distinguish the input and output spaces for F.

- The variables x, y, and z are reserved to denote points in $\mathbb{R}^d$.
- The variables X, Y, and Z are reserved to denote points in $\mathbb{R}^p$ (or a projection of $\mathbb{R}^p$, as described in Section 3.3.1).
- The variables i and j are used as arbitrary indices for lists and vectors. A superscript is used to index elements from a list, and a subscript is used to index components of a vector. For example, $u_j^{(i)}$ indicates the jth component of the ith vector from a list.
- The function $\hat{F}$ denotes any approximation to F. If $\hat{F}$ is annotated with a subscript (e.g., $\hat{F}_{DT}$), then this denotes a specific approximation to F (e.g., the Delaunay interpolant).

1.4 Contributions

The rest of this thesis is organized into the following four chapters. Chapters 2, 3, and 4 each provide their own self-contained literature reviews.

Chapter 2 describes and analyzes the package DELAUNAYSPARSE, which implements a novel algorithm for solving the MIP by performing Delaunay interpolation in $\mathbb{R}^d$. The proposed algorithm cheats the curse of dimensionality in order to scale to hundreds of dimensions. A robust, backward stable Fortran 2003 implementation is provided with both serial and parallel execution modes. This chapter is based on work that has been published in [22] and is currently under review with *ACM Transactions on Mathematical Software* [24].

Chapter 3 describes and analyzes the package VTMOP, which extends a recently proposed algorithm [41] for blackbox multiobjective optimization into a flexible framework for solving MOPs. Within this framework is a novel algorithm for computing the Delaunay graph, which is most likely of interest independently. The framework and a corresponding driver subroutine are implemented in Fortran 2008, with both serial and parallel execution modes. The VTMOP framework is integrated into a real-world simulation distributing software library from Argonne National Laboratory, which was designed with the goal of achieving increased levels of concurrency in simulation optimization problems in exascale HPC systems. Parts of this chapter are based on work that has been published in [26].

Chapter 4 circles back to the VarSys problem and demonstrates how both DELAUNAYSPARSE and VTMOP can be used to solve real-world problems. This chapter is based on work that has been published in [23] and is currently under review for publication in *the 2020 Winter Simulation Conference* [25].

Chapter 5 recaps the contributions and conclusions of this thesis and provides directions for future work.

The major contributions of this thesis are as follows.

- A novel and scalable algorithm for Delaunay interpolation is proposed.
- A robust, backward stable implementation (DELAUNAYSPARSE) of this algorithm is provided in Fortran 2003 with both serial and parallel execution modes.
- A novel algorithm for computing the Delaunay graph is proposed, which uses DELAUNAYSPARSE in a nontraditional way and is capable of scaling to dimensions that are well beyond the limitations of current state-of-the-art algorithms [14] and software [53].
- A novel framework and software package (VTMOP) is provided for solving computationally expensive blackbox MOPs.
- This framework is integrated into a software library for solving similar types of problems on exascale HPC systems.

- The algorithms and software developed throughout this thesis are demonstrated on a real-world problem in HPC systems.

Chapter 2

Multivariate Interpolation and DELAUNAYSPARSE

2.1 Introduction

This chapter formally introduces and solves the Delaunay interpolation problem. The results in this chapter are based on work that has been published in [22] and is currently under review with *ACM Transactions on Mathematical Software* [24].

The rest of this chapter is organized as follows. Section 2.2 provides a review of the literature on properties of Delaunay triangulations, algorithms for computing Delaunay triangulations, and applications of Delaunay triangulations in solving MIPs. Section 2.3 formally defines the Delaunay interpolant and outlines a novel algorithm for computing it. Section 2.4 details the computational aspects of the algorithm, with an emphasis on numerical stability and efficiency. Section 2.5 provides an analysis of the algorithm. Section 2.6 describes the serial implementation of the algorithm and its additional features. Section 2.7 describes the parallel implementation, which uses OpenMP in a shared memory paradigm. Section 2.8 shows performance statistics, demonstrating the scalability of the algorithm.

2.2 Literature Review

The Delaunay triangulation is an unstructured simplicial mesh that is widely studied in the field of computational geometry. Due to its many favorable properties, the Delaunay triangulation finds wide use as a mesh for multivariate interpolation in the fields of geographic information systems (GIS), civil engineering, physics, and computer graphics. See Section 9.6 of the book by de Berg et al. [36] for a brief discussion of the usefulness of Delaunay triangulations. The viability of Delaunay triangulations as a means for interpolating arbitrary nonlinear functions in the context of data science and machine learning has been explored in [78] and more recently in [11]. For a discussion of the general properties of Delaunay triangulations, see the book by Aurenhammer et al. [8].

In this work, the main problem of interest is the MIP. Interpolation via meshes such as triangulations and other tessellations is a classic practice. Delaunay triangulations are widely

considered optimal simplicial meshes for many meshing applications, including interpolation. See the first chapter of the book by Cheng et al. [29] for an overview of Delaunay meshing applications and theory, and see [85] for specific theorems on the optimality of Delaunay triangulations in arbitrary dimension.

In two dimensions, the Delaunay triangulation of n points can be efficiently computed in $\mathcal{O}(n \log n)$ time [92]. After the Delaunay triangulation has been computed, the cost of evaluating each interpolation point is reduced to the cost of point location. In two dimensions, point location can be performed in $\mathcal{O}(n^{\frac{1}{3}})$ time [74], so the total cost of interpolating at m points in two dimensions is $\mathcal{O}(n \log n + n^{\frac{1}{3}} m)$. However, [60] shows that in $\mathbb{R}^d$, the worst case size of the Delaunay triangulation is $\Omega(n^{\lceil d/2 \rceil})$. In practice, the size of Delaunay triangulations can grow much slower for favorable data sets. In particular, for *well-spaced* point sets, the size of the Delaunay triangulation depends only linearly on n [73] but may still depend exponentially on d, a phenomenon often associated with *the curse of dimensionality*.

Despite the complexity of high-dimensional Delaunay triangulations, there are currently a wide variety of algorithms for computing them. The first algorithm capable of computing Delaunay triangulations in arbitrary dimension was proposed independently in both [15] and [99]. Perhaps the most widely used algorithm for computing Delaunay triangulations in arbitrary dimension is the algorithm Quickhull [9]. Quickhull is a time-efficient algorithm running in $\Theta(n \log n + t)$ time, where t denotes the size of the Delaunay triangulation. Quickhull also boasts a highly optimized numerically stable implementation that is used in `SciPy 1.0` [98] and the Matlab Computational Geometry Toolbox (v. 1.2) [80]. An alternative to Quickhull is the graph based algorithm proposed in [14], which stores the Delaunay triangulation in a memory efficient graph structure, at the cost of a slightly greater compute time. An exact arithmetic implementation of the algorithm in [14] is provided in the Delaunay triangulation package of the Computational Geometry Algorithms Library (CGAL) [53]. One final algorithm of interest is the DeWall algorithm [31]. The DeWall algorithm, though not in widespread use, features a unique divide-and-conquer paradigm, and was a major inspiration behind this work.

Due to the exponential growth of Delaunay triangulations in high dimensions, none of the above mentioned algorithms are intended to scale past six or seven dimensions. In fact, this failure to scale to high dimensions is suffered by nearly every mesh based approximation. Consequently, high-dimensional approximation is generally dominated by mesh free methods such as multivariate polynomials, radial basis functions, low order splines, inverse distance weightings, kernel methods, and machine learning techniques such as support vector regressors and artificial neural networks (see the book by Cheney and Light [28]). By leveraging the sparse nature of the interpolation problem, the DELAUNAYSPARSE package aims to add the high-dimensional Delaunay mesh based interpolant to the numerical analyst's toolbox.

2.3 Interpolation via the Delaunay Triangulation

Let $\mathcal{D}$ be a set of $n > d$ data points in $\mathbb{R}^d$. A d-dimensional triangulation is defined as a mesh of d-simplices that (1) are disjoint except on their shared boundaries, (2) whose set of vertices is $\mathcal{D}$, and (3) whose union is the convex hull of $\mathcal{D}$, denoted $CH(\mathcal{D})$. The interpolation problem is: given values $F(x)$ for all points $x \in \mathcal{D}$ where $F : \mathbb{R}^d \to \mathbb{R}^p$, find an approximation $\hat{F} \approx F$ such that $\hat{F}(x) = F(x)$ for all $x \in \mathcal{D}$, where $\hat{F}$ has support in $CH(\mathcal{D})$.

Let $T(\mathcal{D})$ be a d-dimensional triangulation of $\mathcal{D}$. To define an interpolant in terms of $T(\mathcal{D})$, let $y \in CH(\mathcal{D})$ be an interpolation point, and let $\mathcal{S}$ be a simplex in $T(\mathcal{D})$ with vertices $s^{(1)}$, $\ldots$, $s^{(d+1)}$ such that $y \in \mathcal{S}$. Then there exist weights $w_1, \ldots, w_{d+1}$ such that $y = \sum_{i=1}^{d+1} w_i s^{(i)}$, $\sum_{i=1}^{d+1} w_i = 1$, and $w_i \geq 0$ for $i = 1, \ldots, d+1$, and the interpolant $\hat{F}_T$ is given by

$$\hat{F}_T(y) = w_1 F(s^{(1)}) + w_2 F(s^{(2)}) + \ldots + w_{d+1} F(s^{(d+1)}). \tag{2.1}$$

In DELAUNAYSPARSE, the interpolant $\hat{F}_{DT}$ is computed, where $DT(\mathcal{D})$ denotes a Delaunay triangulation of $\mathcal{D}$. The Delaunay triangulation is often defined as the geometric dual of the Voronoi diagram (also called the Dirichlet tessellation). In this work, the equivalent definition in 2.1 is preferred.

Definition 2.1. For a d-simplex $\mathcal{S}$, let $\mathcal{B}^{(\mathcal{S})}$ denote the open ball whose center and radius are given by the $(d-1)$-sphere circumscribing $\mathcal{S}$. Then a Delaunay triangulation $DT(\mathcal{D})$ of a finite set of points $\mathcal{D} \subset \mathbb{R}^d$ is any triangulation of $\mathcal{D}$ such that for each $\mathcal{S} \in DT(\mathcal{D})$, $\mathcal{B}^{(\mathcal{S})}$ satisfies $\mathcal{B}^{(\mathcal{S})} \cap \mathcal{D} = \emptyset$.

A simple two-dimensional illustration of Definition 2.1 is given in Figure 2.1. Remarks 2.2, 2.3, and 2.4 describe several key properties of a Delaunay triangulation.

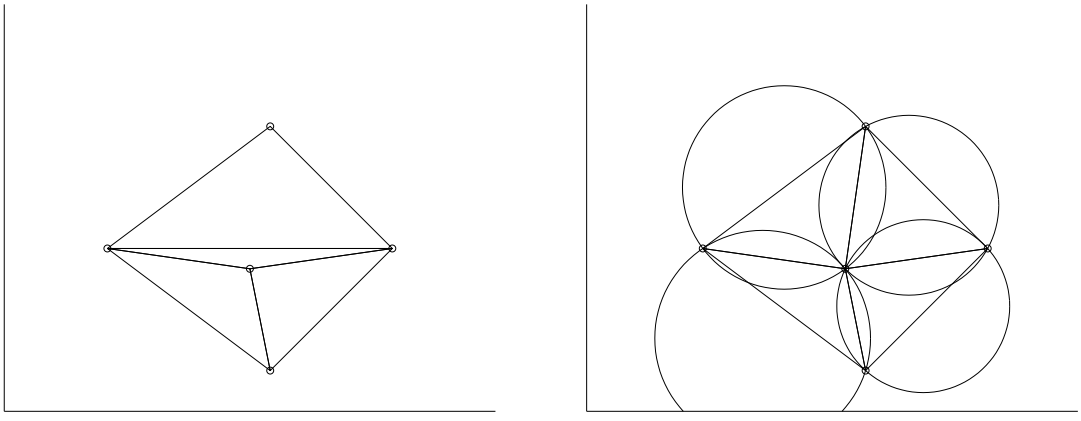

Figure 2.1: A triangulation in $\mathbb{R}^2$ (left) and the Delaunay triangulation (right).

Remark 2.2. Given a set of $d+1$ vertices in $\mathcal{D}$ that define a d-simplex $\mathcal{S}$, the condition that $\mathcal{B}^{(\mathcal{S})} \cap \mathcal{D} = \emptyset$ is not only necessary, but also sufficient to conclude that $\mathcal{S} \in DT(\mathcal{D})$ for *some* Delaunay triangulation $DT(\mathcal{D})$.

Remark 2.3. Let Γ be a facet of a simplex $\mathcal{S} \in DT(\mathcal{D})$. Let $\mathcal{H}$ denote any halfspace whose boundary is the hyperplane containing Γ. Let $x^{(1)}, \ldots, x^{(l)}$ be a sequence of points that are in $\mathcal{D} \cap \mathcal{H}$. Define the open circumballs $\mathcal{B}^{(1)}, \ldots, \mathcal{B}^{(l)}$ such that each $\mathcal{B}^{(i)}$ circumscribes Γ and $x^{(i)}$. Assume $x^{(1)}, \ldots, x^{(l)}$ satisfies $x^{(i)} \in \mathcal{B}^{(i+1)}$ for all $1 \leqq i < l$. Then $\mathcal{B}^{(1)} \cap \mathcal{H} \subset \mathcal{B}^{(2)} \cap \mathcal{H} \subset \cdots \subset \mathcal{B}^{(l)} \cap \mathcal{H}$. A two-dimensional illustration of this property is given in Figure 2.2.

Remark 2.4. In randomly generated data, the cases where $DT(\mathcal{D})$ does not exist or is not unique occur with probability zero. Therefore, for algorithmic analysis, it is common to make the simplifying assumption that $\mathcal{D}$ is in *general position*, meaning $DT(\mathcal{D})$ exists and is unique. Furthermore, note that the case where $DT(\mathcal{D})$ does not exist occurs only if all the points in $\mathcal{D}$ are contained in some lower-dimensional linear manifold. In the context of interpolation, this corresponds to an over parameterization of the underlying function and can be resolved with dimension reduction techniques. The case where $DT(\mathcal{D})$ is not unique can occur in real-world problems and will be addressed in the implementation, discussed in Section 2.4.

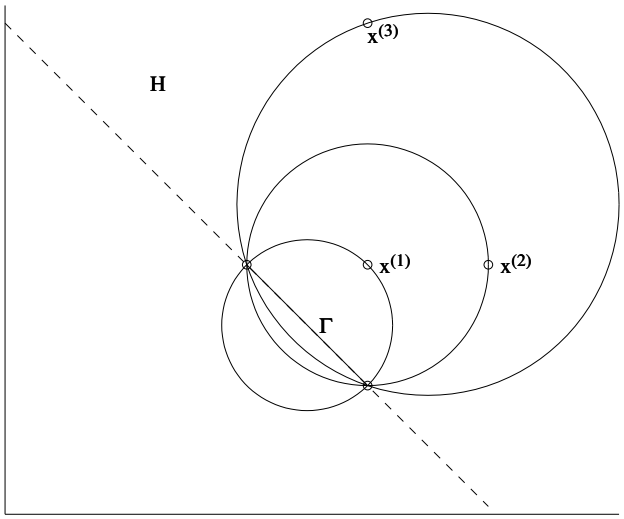

Figure 2.2: A two-dimensional visualization of the property described in Remark 2.3. The solid line segment is a facet Γ, $\mathcal{H}$ is the halfspace above the dashed line, and $\{x^{(1)}, x^{(2)}, x^{(3)}\}$ is a sequence satisfying $x^{(1)} \in \mathcal{B}^{(2)}$ and $x^{(2)} \in \mathcal{B}^{(3)}$.

2.3.1 The DELAUNAYSPARSE Algorithm

Note that given a set of m interpolation points $\mathcal{Q}$, at most m simplices in $DT(\mathcal{D})$ are needed to compute $\hat{F}_{DT}(y)$ for all $y \in \mathcal{Q}$. Therefore, for this particular problem, it is possible to "cheat" the curse of dimensionality when m is significantly less than the size of $DT(\mathcal{D})$. To do so, it suffices to compute a sparse subset of $DT(\mathcal{D})$ such that $\mathcal{Q}$ is contained in the subset.

As previously mentioned, one of the major inspirations behind this work was the DeWall algorithm [31]. The DeWall algorithm features a divide-and-conquer paradigm where construction of each Delaunay simplex is carefully guided so as to construct a "wall" of simplices, bisecting the data set. Each successive simplex is completed from a facet of a previously constructed simplex, using the same methodology as the classic gift-wrapping approach (described in Section 5.6 of the book by Cheng et al. [29]).

The two key components of the DeWall and gift-wrapping algorithms that are used in DELAUNAYSPARSE are the growth of the seed simplex and the completion of an open Delaunay facet. After iteratively constructing a seed simplex, the idea is to perform a *visibility walk* to the simplex containing each interpolation point, as described in [42]. Since the complete triangulation is never computed, each step of the walk is performed by completing the Delaunay facet designated by the visibility walk protocol. Once each interpolation point y has been located, each response value $\hat{F}_{DT}(y)$ can be computed using (2.1). Pseudocode for interpolating at a single point follows.

Algorithm 2.5.
input
$\mathcal{D}$ contains n d-dimensional data points;
$y \in CH(\mathcal{D})$ is the interpolation point;

output
$\hat{F}_{DT}(y)$ is the value of the Delaunay interpolant;

begin
$\mathcal{S}$ denotes the current d-simplex;
Γ denotes the facet of $\mathcal{S}$ from which y is visible;
Grow an initial seed d-simplex $\mathcal{S}$, as described in Section 2.4.1.
while $y \notin \mathcal{S}$ **do**
    Select the facet Γ of $\mathcal{S}$ from which y is visible as described in Section 2.4.2;
    Complete a new d-simplex $\mathcal{S}^*$ from the facet Γ as described in Section 2.4.3;
    $\mathcal{S} \leftarrow \mathcal{S}^*$.
enddo
Since the loop has terminated, $y \in \mathcal{S}$. Compute $\hat{F}_{DT}(y)$ using (2.1).
return $\hat{F}_{DT}(y)$

The advantages of this technique are maximized when the interpolation points are sparse with respect to the size of the triangulation. In practice, the number of simplices constructed during the walk to each interpolation point is a polynomial function of d and often seemingly independent of n. Given the exponential nature of the problem, this makes for an effective sparse solution, particularly in high dimensions.

2.3.2 Relationship to Linear Programming

For those experienced in linear programming, Algorithm 2.5 may seem reminiscent of the Dantzig simplex method [33] for solving linear programs. In fact, computing the Delaunay simplex containing an interpolation point can be connected to linear programming.

Let $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$, and let

$$\tilde{A} = \begin{bmatrix} (-x^{(1)})^T & 1 \\ (-x^{(2)})^T & 1 \\ \vdots & \vdots \\ (-x^{(n)})^T & 1 \end{bmatrix}, \quad \tilde{b} = \begin{bmatrix} \|x^{(1)}\|_2^2 \\ \|x^{(2)}\|_2^2 \\ \vdots \\ \|x^{(n)}\|_2^2 \end{bmatrix}, \text{ and } \quad \tilde{c} = \begin{bmatrix} -y \\ 1 \end{bmatrix}.$$

Consider the primal and dual linear programs

$$\max_{\tilde{u}} \ \tilde{c}^T\tilde{u} \text{ such that } \tilde{A}\tilde{u} \leq \tilde{b}, \ \tilde{u} \text{ free}, \qquad \min_{\tilde{v}} \ \tilde{b}^T\tilde{v} \text{ such that } \tilde{A}^T\tilde{v} = \tilde{c}, \ \tilde{v} \geq 0.$$

For the primal problem, every extreme point of the feasible set satisfies $\tilde{u} = (-2\tilde{\zeta}, \tilde{\eta}^2 - \|\tilde{\zeta}\|_2^2)$ where $\tilde{\zeta}$ and $\tilde{\eta}$ are the circumsphere center and radius, respectively, for a simplex in $DT(\mathcal{D})$ (not necessarily containing y). For the dual problem, every extreme point of the feasible set contains a vector of weights expressing y as a convex combination of $d+1$ vertices of a simplex (not necessarily Delaunay) containing y. The optimal solution for both corresponds to a Delaunay simplex containing y. The vertex set for a nondegenerate Delaunay simplex containing y is not immediately given by the solution to the primal or dual problem. Rather, the vertices can be inferred from a nondegenerate basic solution to the dual problem.

When $\mathcal{D}$ is in general position, DELAUNAYSPARSE can be interpreted as a different strategy for flipping through simplices than the Dantzig simplex algorithm. For real-world data, which could be degenerate, finding a nondegenerate basic solution is significantly harder than solving the linear program [71]. By favoring a geometric interpretation of the problem, DELAUNAYSPARSE is robust for degenerate input sets. DELAUNAYSPARSE also avoids significant computational expense and memory burden, which would be introduced through auxiliary variables when reducing the primal problem to canonical form.

2.4 Computational Aspects

In this section, the computational operations referenced in Algorithm 2.5 will be fully detailed. These operations are the growth of the seed simplex, the visibility walk, and flipping across a Delaunay facet.

2.4.1 Growing the Seed Simplex

The seed simplex is constructed through a greedy algorithm, by adding vertices one at a time such that each new vertex minimizes the radius of the minimum radius circumsphere through the current vertex set. The initial vertex $s^{(1)}$ is chosen to be the closest point in the data set $\mathcal{D}$ to the interpolation point y, and ties are resolved by choosing the point in $\mathcal{D}$ with the lowest index. The second vertex $s^{(2)} \in \mathcal{D} \setminus \{s^{(1)}\}$ is chosen such that

$$\|s^{(2)} - s^{(1)}\| = \min_{\substack{x \in \mathcal{D}, \\ x \neq s^{(1)}}} \|x - s^{(1)}\|.$$

Each subsequent vertex is chosen to minimize the radius of the minimum radius circumsphere passing through the resulting vertex set.

For $2 \leqq j \leqq d$ and $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$, define the $j \times d$ matrix

$$A^{(j,x)} = \begin{bmatrix} \left(s^{(2)} - s^{(1)}\right)^T \\ \vdots \\ \left(s^{(j)} - s^{(1)}\right)^T \\ \left(x - s^{(1)}\right)^T \end{bmatrix}$$

and the j-vector

$$b^{(j,x)} = \frac{1}{2} \begin{bmatrix} \|s^{(2)} - s^{(1)}\|^2 \\ \vdots \\ \|s^{(j)} - s^{(1)}\|^2 \\ \|x - s^{(1)}\|^2 \end{bmatrix}.$$

If rank $A^{(j,x)} = j$, then the minimum norm solution to the under determined system

$$A^{(j,x)}\xi = b^{(j,x)} \tag{2.2}$$

is $\xi^* = c - s^{(1)}$, where c denotes the center of the minimum radius circumsphere about $s^{(1)}$, $\ldots$, $s^{(j)}$, x. So, each subsequent vertex $s^{(j+1)}$ is given by the $x^* \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$ such that solving (2.2) with $A^{(j,x^*)}$ and $b^{(j,x^*)}$ produces the minimum 2-norm solution ξ^*.

If rank $A^{(j,x)} < j$, then $s^{(1)}, \ldots, s^{(j)}$, x are not the vertices of a j-simplex, and x cannot be a vertex of any d-simplex with vertices $s^{(1)}, \ldots, s^{(j)}$. Hence, x can be skipped and need not be considered again when constructing the seed simplex. Due to memory constraints, there is no mechanism for marking a x that can be skipped, and hence any such x could be revisited in the future, though it will always be skipped.

If $\mathcal{D}$ is in general position, then there will always exist a unique point x^* that minimizes $\|\xi^*\|$. However, if $\mathcal{D}$ lies in a $(j-1)$-dimensional linear manifold where $j \leqq d$, then any set of $j+1$ or more points in $\mathcal{D}$ will be affinely dependent. Therefore rank $A^{(j,x)} < j$ for *all* $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$, and no solution x^* can exist. Indeed, $DT(\mathcal{D})$ does not exist as

discussed in Remark 2.4. Furthermore, $DT(\mathcal{D})$ is not unique if there exist $d+2$ or more points in $\mathcal{D}$ that lie on the same circumball, since there may be more than one $x^* \in \mathcal{D}$ that minimize $\|\xi^*\|$. For the purpose of interpolation, all of these solutions are equally suitable, and the decision between any number of such candidate solutions can be made arbitrarily. In particular, these cases are resolved by choosing the candidate x^* with the lowest index in $\mathcal{D}$. Pseudocode for this process follows.

Algorithm 2.6.
input
$\mathcal{D}$ contains n d-dimensional data points;
$y \in CH(\mathcal{D})$ is the interpolation point;
output
$\{s^{(1)}, \ldots, s^{(d+1)}\}$ denote the vertices of the seed simplex.
begin
$s^{(1)} \leftarrow \underset{x \in \mathcal{D}}{\arg\min} \|y - x\|$;
$s^{(2)} \leftarrow \underset{\substack{x \in \mathcal{D}, \\ x \neq s^{(1)}}}{\arg\min} \|s^{(1)} - x\|$;
for $j = 2, \ldots, d$ **do**
  initialize $r_{min} \leftarrow \infty$; $x^* \leftarrow null$;
  for all $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$ **do**
    Compute $A^{(j,x)}$ and $b^{(j,x)}$;
    if rank $A^{(j,x)} < j$ **then**
      Skip this point;
    else if rank $A^{(j,x)} = j$ **then**
      Compute the minimum norm solution ξ^* to (2.2);
      if $\|\xi^*\| < r_{min}$ **then**
        $r_{min} \leftarrow \|\xi^*\|$;
        $x^* \leftarrow x$;
      endif
    endif
  enddo
  $s^{(j+1)} \leftarrow x^*$;
enddo
if $x^* \neq null$ **then**
  return $\{s^{(1)}, \ldots, s^{(d+1)}\}$;
else
  return error (points in a lower-dimensional linear manifold);
endif

The dominant costs of the above algorithm are determining the rank of $A^{(j,x)}$ and finding the minimum norm solution ξ^* to (2.2). Both of these computations can be done using a LQ factorization of $A^{(j,x)}$. Such a factorization has a computational complexity of at most $\mathcal{O}(d^3)$ (in the case where $j = d$).

Remark 2.7. To determine the rank of $A^{(j,x)}$ using a LQ factorization, consider the final term on the diagonal of L, L_{jj}. From the construction of $A^{(j,x)}$, in exact arithmetic L_{jj} is the distance from x to the $(j-1)$-dimensional linear manifold defined by $\{s^{(1)}, \ldots, s^{(j)}\}$. To allow for floating point error, x should be at least a distance of ε outside of the manifold. Therefore $A^{(j,x)}$ is considered singular if $|L_{jj}| < \varepsilon$, where ε is a scale/machine dependent constant. In practice, this check is sufficient to avoid flat simplices. Furthermore, because the Euclidean distance from the $(j-1)$-dimensional linear manifold defined by $\{s^{(1)}, \ldots, s^{(j)}\}$ is used as the criterion for singularity, this check is amenable to backward stability in the geometric sense. Specifically, no input point whose Euclidean distance from the manifold is greater than ε is treated as affinely dependent. This is distinctly different from approximating the distance to rank deficiency for the matrix $A^{(j,p)}$. Shroff and Bischof [88] provide an algorithm for approximating the latter after a rank-1 update at the cost of an additional linear solve. In order to save computational expense (the check $|L_{jj}| < \varepsilon$ comes at no additional expense) and guarantee backward stability in the geometric sense, the simpler check $|L_{jj}| < \varepsilon$ is favored here. In pathological cases that are rare in practice, this could result in a matrix $A^{(j,p^*)}$ that is nearly rank deficient in the matrix 2-norm.

Remark 2.8. In each iteration of the inner loop in Algorithm 2.6 (over all $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$), a jth row is appended onto $A^{(j-1,s^{(j)})}$ to construct $A^{(j,x)}$. So, given the LQ factorization of $A^{(j-1,s^{(j)})}$, it is possible to directly compute the minimum norm solution to (2.2) in $\mathcal{O}(d^2)$ time by using a rank-1 update. These updates are always applied to the original LQ factorization of $A^{(j-1,s^{(j)})}$, which is full-rank by construction. Therefore there is no risk of compounding numerical error, and the complexity of the inner loop is $\mathcal{O}(nd^2)$.

Using the rank-1 update described in Remark 2.8, the total complexity of Algorithm 2.6 (for growing a seed d-simplex) is reduced to $\mathcal{O}(nd^3)$.

The correctness of Algorithm 2.6 follows from Lemma 2.9, whose proof is in the appendix.

Lemma 2.9. *Let $\mathcal{D}$ be a set of points in $\mathbb{R}^d$ in general position, and let $\{s^{(1)}, \ldots, s^{(j)}\}$ be the vertex set for a $(j-1)$-face ($j \leq d$) with the following property: the open ball whose boundary is the minimum radius $(d-1)$-sphere passing through $\{s^{(1)}, \ldots, s^{(j)}\}$ has empty intersection with $\mathcal{D}$. Let $x^* \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$ be the minimizer for $\|\xi^*\|$, as in Algorithm 2.5. Then $\{s^{(1)}, \ldots, s^{(j)}, x^*\}$ is the vertex set for a j-face whose minimum radius circumsphere has the same property.*

If $\mathcal{D}$ is in general position (up to the precision ε from Remark 2.7), then Lemma 2.9 guarantees that after each iteration of the outer loop (**for** $j = 2, \ldots, d$ **do**) in Algorithm 2.6 the set $\{s^{(1)}, \ldots, s^{(j+1)}\}$ maintains the property that its minimum radius circumsphere is the boundary for an "empty" ball. Following Definition 2.1, this guarantees that the final set $\{s^{(1)}, \ldots, s^{(d+1)}\}$ is the vertex set for a Delaunay simplex. Even if $\mathcal{D}$ is not in general position, as long as $DT(\mathcal{D})$ exists, Algorithm 2.6 produces the vertex set for a nondegenerate simplex (i.e., nonzero volume) that is within an ε-magnitude perturbation of being a Delaunay simplex.

2.4.2 The Visibility Walk

After constructing the seed simplex, DELAUNAYSPARSE advances on the simplex containing an interpolation point y by following a visibility walk. A facet Γ of a simplex $\mathcal{S}$ is said to be *visible* to y if there exists a point $\rho \in \text{int } \mathcal{S}$ such that the line segment drawn from ρ to y intersects Γ. A visibility walk is a sequence of "flips" that always occur across a facet from which y is visible. Note that each flip in a visibility walk is generally not unique, as it is possible for y to be visible from multiple facets, and a flip across *any* visible facet constitutes a valid step in a visibility walk. Edelsbrunner [45] showed that in a Delaunay triangulation, every visibility walk is acyclic and therefore must converge for any $y \in CH(\mathcal{D})$. A formal statement of Edelsbrunner's theorem and its proof are provided in the appendix. The mechanics of performing each flip in the visibility walk will be detailed in Section 2.4.3. In this section, the processes of identifying each visible facet and terminating the visibility walk will be explored.

For a simplex $\mathcal{S}$ with vertices $s^{(1)}, \ldots, s^{(d+1)}$, define the $d \times d$ matrix

$$A^{(\mathcal{S})} = \left[\left(s^{(2)} - s^{(1)}\right) \quad \cdots \quad \left(s^{(d+1)} - s^{(1)}\right) \right].$$

Let ξ_i denote the ith entry in the d-vector ξ, given by the solution to the linear system

$$A^{(\mathcal{S})}\xi = y - s^{(1)}. \tag{2.3}$$

Then the vector of affine weights $w = [w_1, \ldots, w_{d+1}]^T$ for generating y as a combination of $s^{(1)}, \ldots, s^{(d+1)}$ is given by

$$w = \begin{bmatrix} \left(1 - \sum_{i=1}^{d} \xi_i\right) \\ \xi_1 \\ \vdots \\ \xi_d \end{bmatrix}.$$

If $w_i \geqq 0$ for $i = 1, \ldots, d+1$, then $y \in \mathcal{S}$ and w contains the interpolation weights in (2.1). If any $w_i < 0$, then dropping the corresponding vertex $s^{(i)}$ leaves the vertices of a facet of $\mathcal{S}$ from which y is visible. If $\mathcal{S}$ is a valid Delaunay simplex, $A^{(\mathcal{S})}$ is nonsingular and (2.3) can be solved via LU factorization. In practice, DELAUNAYSPARSE solves (2.3) using a LQ factorization, which can be reused for performing the "flip" operation, as described in Section 2.4.3.

Remark 2.10. To account for floating point errors, the condition that $w_i \geqq 0$ for $i = 1, \ldots, d+1$ should be replaced with $w_i \geqq -\varepsilon$, where ε is a scale/machine dependent constant, similarly as in Remark 2.7.

The cost of a single LQ factorization for solving (2.3) is dominated by the cost of performing a flip, as described in the next section, which requires up to n rank-1 updates. However, the total length of the visibility walk (in number of flips) will be important in determining the computational complexity of the DELAUNAYSPARSE algorithm and is analyzed in Section 2.5.2.

2.4.3 Flipping Across a Facet

Let $\Gamma = CH(\{s^{(1)}, \ldots, s^{(d)}\})$ be a facet of a previously constructed Delaunay simplex from which the interpolation point y is visible. Let $\mathcal{H}(\Gamma)$ denote the hyperplane containing Γ, and let $\mathcal{H}^{(y)}(\Gamma)$ denote the open halfspace (with respect to $\mathcal{H}(\Gamma)$) that contains y. The goal of this section is to "flip toward" y, by constructing a new Delaunay d-simplex with vertices $s^{(1)}, \ldots, s^{(d+1)}$, where $s^{(d+1)} \in \mathcal{D} \cap \mathcal{H}^{(y)}(\Gamma)$.

Recall from Remark 2.2 that a d-simplex $\mathcal{S}$ is Delaunay if and only if $\mathcal{B}^{(\mathcal{S})} \cap \mathcal{D} = \emptyset$. Since at least one Delaunay simplex (of which Γ is a facet) has already been constructed, $DT(\mathcal{D})$ exists. Therefore, if Γ is **not** a facet of $CH(\mathcal{D})$, there must be at least one point $x^* \in \mathcal{D} \cap \mathcal{H}^{(y)}(\Gamma)$ such that the simplex $\mathcal{S}^*$ with vertices $\{s^{(1)}, \ldots, s^{(d)}, x^*\}$ is Delaunay, satisfying $\mathcal{B}^{(\mathcal{S}^*)} \cap \mathcal{D} = \emptyset$. If no such x^* exists, then it can be inferred that $y \notin CH(\mathcal{D})$. So, to perform a "flip" to a new Delaunay simplex closer to y, it suffices to check inside the circumball of the simplex with vertices $s^{(1)}, \ldots, s^{(d)}, x$, for each $x \in \mathcal{D} \cap \mathcal{H}^{(y)}(\Gamma)$. By exploiting the property described in Remark 2.3, this can be done in a single pass over $\mathcal{D}$.

For a facet Γ with vertices $s^{(1)}, \ldots, s^{(d)}$, let the $(d-1) \times d$ matrix

$$A^{(\Gamma)} = \begin{bmatrix} \left(s^{(2)} - s^{(1)}\right)^T \\ \vdots \\ \left(s^{(d)} - s^{(1)}\right)^T \end{bmatrix}.$$

To obtain a normal to $\mathcal{H}(\Gamma)$, it suffices to take any nontrivial vector in the nullspace of $A^{(\Gamma)}$. Since Γ is a Delaunay facet, rank $A^{(\Gamma)} = d - 1$. Therefore, using a LQ factorization with row pivoting, $PA^{(\Gamma)} = LQ$, the unit normal to $\mathcal{H}(\Gamma)$ is given by the last row of $\mathcal{Q}$, $\hat{v}^T = Q_{d\cdot}$.

Given the normal vector $\hat{v}$ for $\mathcal{H}(\Gamma)$, consider the function

$$\Psi^{(\Gamma)}(x) = \operatorname{sgn}((x - s^{(1)})^T \hat{v}). \tag{2.4}$$

For each $x \in \mathcal{D}$, $x \in \mathcal{H}^{(y)}(\Gamma)$ if and only if $\Psi^{(\Gamma)}(x) = \Psi^{(\Gamma)}(y)$.

Remark 2.11. To account for floating point errors, the additional condition that $|(x - s^{(1)})^T \hat{v}| > \varepsilon$ should be imposed, where ε is a scale/machine dependent constant, similarly as in Remarks 2.7 and 2.10. Note that similarly as in Remark 2.7, this is a geometric distance threshold and does not guarantee that the resulting matrix $A^{(d,p)}$ is far from singular for pathological point sets.

Consider now those $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(d)}\}$ such that $\Psi^{(\Gamma)}(x) = \Psi^{(\Gamma)}(y)$, i.e., $x \in \mathcal{H}^{(y)}(\Gamma)$. Similarly as in Section 2.4.1, the center of the circumball about Γ and x is given by $c = \xi^* + s^{(1)}$, and the radius of the circumball is given by $\|\xi^*\|$, where ξ^* is a solution to the system

$$A^{(d,x)}\xi = b^{(d,x)}. \tag{2.5}$$

Since Γ is a valid Delaunay facet, the first $d-1$ columns of $A^{(d,x)}$ must be linearly independent. Furthermore, if the condition described in Remark 2.7 has been satisfied, then it is safe to assume that $A^{(d,x)}$ is full-rank, and (2.5) has a unique solution. Then x^* is any point in $\mathcal{D} \cap \mathcal{H}^{(y)}(\Gamma)$ that satisfies $\mathcal{B}^{(\|\xi^*\|)}(c) \cap \mathcal{D} = \emptyset$, where $\mathcal{B}^{(\|\xi^*\|)}(c)$ denotes the open ball centered at c with radius $\|\xi^*\|$. Pseudocode for this entire process follows.

Remark 2.12. If $\mathcal{D}$ is in general position, then x^* is unique. If there exist $d+2$ or more cospherical points in $\mathcal{D}$, then x^* may not be unique. However, any $x^* \in \mathcal{D} \cap \mathcal{H}^{(y)}(\Gamma)$ that satisfies $\mathcal{B}^{(\|\xi^*\|)}(c) \cap \mathcal{D} = \emptyset$ can be chosen in union with $\{s^{(1)}, \ldots, s^{(d)}\}$ to form the vertex set for a valid simplex in *some* Delaunay triangulation. Such situations can be resolved by choosing the x^* with the greatest index in $\mathcal{D}$.

Algorithm 2.13.
input
$\mathcal{D}$ contains n d-dimensional data points;
$y \in CH(\mathcal{D})$ is the interpolation point;
Γ is the current Delaunay facet;
output
$\{s^{(1)}, \ldots, s^{(d+1)}\}$ denote the vertices of the new simplex;
begin
$\{s^{(1)}, \ldots, s^{(d)}\}$ are given by the vertices of Γ;
Compute the LQ factorization $PA^{(\Gamma)} = LQ$ and set $\hat{v}^T \leftarrow Q_{d\cdot}$;
Compute $\Psi^{(\Gamma)}(y)$ using (2.4);
initialize $r_{min} \leftarrow \infty$; $c_{min} \leftarrow \vec{0}$; $x^* \leftarrow null$;
for all $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(d)}\}$ **do**
  if $\Psi^{(\Gamma)}(x) \neq \Psi^{(\Gamma)}(y)$ **or** $|(x - s^{(1)})^T \hat{v}| < \varepsilon$ **then**
    Skip this point;
  else if $\|x - c_{min}\| < r_{min}$ **then**
    Update $A^{(d,x)}$ and $b^{(d,x)}$ and compute ξ^*, the solution to (2.5);
    $r_{min} \leftarrow \|\xi^*\|$;
    $c_{min} \leftarrow \xi^* + s^{(1)}$;
    $x^* \leftarrow x$;
  endif
enddo
if $x^* = null$ **then**
  return error, $y \notin CH(\mathcal{D})$;
else
  return $s^{(d+1)} \leftarrow x^*$;
endif

The dominant cost for Algorithm 2.13 is repeatedly solving (2.5). Since $A^{(d,x)}$ is always full-rank, each instance of (2.5) could be solved using a LU factorization, with computational complexity $\mathcal{O}(d^3)$. However, similarly as in Remark 2.8, a rank-1 update is applied to the

LQ factorization of $A^{(\Gamma)}$ instead, reducing the complexity per iteration of the loop to $\mathcal{O}(d^2)$. So the worst-case complexity of Algorithm 2.13 is $\mathcal{O}(nd^2)$. Note that Algorithm 2.13 is called once in each iteration of the simplex walk.

The correctness of Algorithm 2.13 is established in Lemma 2.14, whose proof is in the appendix. Again, Lemma 2.14 guarantees that a Delaunay simplex is found when $\mathcal{D}$ is in general position (up to the precision ε). If $DT(\mathcal{D})$ exists but is not unique, a nondegenerate simplex is found that is within an ε-magnitude perturbation of being a Delaunay simplex.

Lemma 2.14. *Let $\mathcal{D}$ be in general position and $y \in CH(\mathcal{D})$, let $\{s^{(1)}, \ldots, s^{(d)}\}$ be the vertex set of a Delaunay facet. Then Algorithm 2.13 computes the vertex set of a Delaunay simplex.*

2.5 Algorithm Analysis

The correctness of DELAUNAYSPARSE immediately follows from Lemmas 2.9 and 2.14 and the acyclicity theorem of Edelsbrunner [45]. When $\mathcal{D}$ is in general position up to perturbations of magnitude ε, Lemmas 2.9 and 2.14 guarantee that every simplex constructed is Delaunay. Then [45] shows that the visibility walk converges to a Delaunay simplex in a finite number of flips. This section further analyzes the sensitivity (when $\mathcal{D}$ is not in general position up to ε magnitude perturbations), time complexity, and approximation accuracy of DELAUNAYSPARSE.

2.5.1 Robustness and Backward Stability

The purpose of ε (as described in Remarks 2.7, 2.10, and 2.11) is to guarantee the robustness of DELAUNAYSPARSE. Remarks 2.7 and 2.11 together guarantee that no simplex can be constructed whose vertex set is nearly coplanar. Remark 2.10 guarantees that any simplex that is within a small perturbation of containing the interpolation point will be accepted, thus preventing DELAUNAYSPARSE from cycling due to perturbations. Together, these remarks guarantee that a nondegenerate simplex is found such that the interpolation point is nearly contained in that simplex, even for degenerate and nearly degenerate data sets. Considering Remarks 2.7, 2.10, and 2.11 along with the standard floating point error, the computed Delaunay simplices are actually elements of a perturbed triangulation, $DT(\hat{\mathcal{D}})$.

An appropriate value of ε should be larger than the backward error due to floating point error (in practice the square root of the unit round-off is enough). Since Remarks 2.7 and 2.11 use ε as a geometric distance threshold, $\|x^{(i)} - \hat{x}^{(i)}\|_2 < \varepsilon$ for all $\hat{x}^{(i)} \in \hat{\mathcal{D}}$ corresponding to $x^{(i)} \in \mathcal{D}$. These conditions can be interpreted as backward stability guarantees.

2.5.2 Time Complexity

The time complexity of DELAUNAYSPARSE depends on the length of the visibility walk. From Section 2.4.1, the time complexity for growing the seed simplex is $\mathcal{O}(nd^3)$. From Sections 2.4.2 and 2.4.3, the time complexity for performing each flip is $\mathcal{O}(nd^2)$. Therefore, the overall time complexity for locating a single interpolation point is $\mathcal{O}(nd^3 + \ell nd^2)$, where ℓ is the length of the visibility walk. Since ℓ is typically much greater than d, this can be simplified to $\mathcal{O}(\ell nd^2)$.

The maximal length of the visibility walk ℓ is an open problem. Bowyer [15] claimed without proof that when starting a visibility walk from the center of a Delaunay triangulation, ℓ is $\mathcal{O}(n^{1/d})$. Mücke et al. [74] proved that in up to three-dimensions, a stronger claim can be made for some variations of the standard visibility walk.

Table 2.1 shows empirically that for uniformly distributed data points, when starting from a simplex grown off the nearest data point to y, ℓ tends to grow polynomially with dimension and has no dependence on n for large values of n. For larger values of d, ℓ appears to have some small dependence on n but this is most likely because $32,000$ data points do not saturate a 32- or 64-dimensional space. The data in Table 2.1 is based on the average of 20 simplex walks by DELAUNAYSPARSE for each dimension/problem size combination.

Table 2.1: Average number (with a sample size of 20) of Delaunay simplices computed in a simplex walk from the simplex built off the nearest neighbor to y for n uniform randomly generated points in d dimensions.

	$n = 2K$	$n = 8K$	$n = 16K$	$n = 32K$
$d = 2$	3.05	2.90	3.25	3.10
$d = 8$	23.75	24.75	24.30	23.10
$d = 32$	95.25	125.60	131.85	150.10
$d = 64$	171.95	221.85	248.35	280.60

If ℓ is truly independent of n, then the overall time complexity of DELAUNAYSPARSE is linear in n. However, because of the relationship with linear programming, proving this would be equivalent to finding a strongly polynomial time algorithm for solving linear programming problems [71], which is Smale's 9th open problem in *Mathematical Problems for the Next Century* [89]. One likely possibility is that ℓ is very short in practice, but that the visibility walk could visit every simplex in $DT(\mathcal{D})$ in the worst case, as demonstrated by Klee and Minty [61] for the Dantzig simplex algorithm.

2.5.3 Approximation Accuracy

The approximation accuracy of simplicial interpolation schemes is analyzed in [68] and [86]. In particular, Lux et al. [68] give an error bound when using $T(\mathcal{D})$ to interpolate a once

differentiable scalar function f with γ-Lipschitz continuous gradient in the 2-norm. Consider an interpolation point $y^{(0)} \in \mathbb{R}^d$, whose containing simplex is $\mathcal{S}^{(0)} \in T(\mathcal{D})$. Suppose $\mathcal{S}^{(0)}$ contains $x^{(0)}$ in its vertex set, has a maximum edge length of ν, and has a barycentric transformation matrix whose smallest singular value is λ_d. Then

$$|f(y^{(0)}) - \hat{f}_T(y^{(0)})| \leqq \frac{\gamma \|y^{(0)} - x^{(0)}\|_2^2}{2} + \frac{\sqrt{d}\gamma\nu^2}{2\lambda_d}\|y^{(0)} - x^{(0)}\|_2.$$

Since the Delaunay property tends to minimize circumball radii [85] and produces "fat" well-conditioned simplices, it follows that $DT(\mathcal{D})$ is optimal for interpolation in comparison with other triangulations. The analytic error bounds that are readily available for Delaunay interpolation are generalized to produce interpretable models for artificial intelligence and machine learning applications in [11] and [64].

By applying DELAUNAYSPARSE, Lux et al. [68] provide a detailed comparison between Delaunay interpolation and other approximation techniques for a wide variety of high-dimensional data science problems. For these problems, Delaunay interpolation is shown to be competitive with widely used modern and classical approximation techniques such as neural network regressors, support vector regressors, spline interpolation, and inverse distance weighting. There is no clear "best algorithm" for high-dimensional approximation in general because the underlying structures of problems vary. However, Delaunay interpolants are accompanied by the broadly applicable uniform error bound above and often outperform other techniques in terms of prediction accuracy. The empirical errors in predictions made by the Delaunay interpolant also seem to agree with the theoretical error bound given above. The success of Delaunay interpolation on these problems demonstrates that DELAUNAYSPARSE is a valuable addition to any data scientist's toolkit.

2.6 Serial Implementation

The serial subroutine `DELAUNAYSPARSES` is implemented in ISO Fortran 2003 [56]. For efficient numerically stable linear algebra, `DELAUNAYSPARSES` uses LAPACK [4]. The complete code is provided in the appendix.

2.6.1 Handling Multiple Interpolation Points

The serial subroutine `DELAUNAYSPARSES` performs interpolation at m points $\mathcal{Q} = \{y^{(1)}, \ldots, y^{(m)}\}$ using Algorithms 2.5, 2.6, and 2.13, as described in Sections 2.3 and 2.4. By default, `DELAUNAYSPARSES` will perform these interpolations sequentially with no modification to Algorithms 2.5, 2.6, 2.13. However, an optional argument can be set to "daisy chain" the visibility walks, i.e., for the ith interpolation point $y^{(i)}$ where $i > 1$, the last constructed

Delaunay simplex (typically the simplex containing $y^{(i-1)}$) is used as the first simplex for walking to $y^{(i)}$, replacing Algorithm 2.6. In general, this behavior can greatly increase the length of each visibility walk and is not recommended. However, if the interpolation points $\mathcal{Q}$ are tightly clustered in a relatively small region of $DT(\mathcal{D})$ or if the size of $DT(\mathcal{D})$ is relatively small, this behavior can slightly improve performance by avoiding the expense of Algorithm 2.6.

Additionally, note that after computing the LU factorization of $A^{(\mathcal{S})}$ for solving (2.3), it requires relatively little additional computation to check whether $\mathcal{S}$ contains any future interpolation points. Thus if $\mathcal{S}$ is a simplex with vertices $s^{(1)}$, ..., $s^{(d+1)}$ that has been constructed during the visibility walk to an interpolation point $y^{(i)}$, `DELAUNAYSPARSES` will check whether $y^{(j)} \in \mathcal{S}$, where $i < j \leqq n$ and $y^{(j)}$ has not already been found in some simplex of $DT(\mathcal{D})$, by solving

$$A^{(\mathcal{S})}\xi = y^{(j)} - s^{(1)}, \tag{2.6}$$

and using the same criterion described in Section 2.4.2. If any $y^{(j)} \in \mathcal{S}$, then $s^{(1)}$, ..., $s^{(d+1)}$ and the corresponding interpolation weights for (2.1) are saved, and $y^{(j)}$ is marked as found and will not be considered again. Because of its cost effectiveness, this behavior is always active during an execution of `DELAUNAYSPARSES`. However, it is most effective for tightly clustered interpolation points.

2.6.2 Extrapolation

Often, it is reasonable to make predictions for extrapolation points that are *slightly* outside $CH(\mathcal{D})$. In these cases, the most reasonable solution is to project each extrapolation point onto $CH(\mathcal{D})$ and interpolate the projection, provided the residual is small. Let z be an extrapolation point, and let E be a $d \times n$ matrix whose columns are points in $\mathcal{D}$. Then the projection $\hat{z}$ of z onto $CH(\mathcal{D})$ is given by $\hat{z} = E\xi^*$, where ξ^* is the solution to the linearly constrained least squares problem

$$\min_{\xi \in \mathbb{R}^n} \|E\xi - z\| \quad \text{subject to} \quad \xi \geq 0 \quad \text{and} \quad \sum_{i=1}^{n} \xi_i = 1. \tag{2.7}$$

Hanson and Haskell [50] provide an efficient solution to (2.7) based on a slack variable formulation. The most recent version of their subroutine `DWNNLS` is available in the SLATEC software package [97], and included with the DELAUNAYSPARSE files.

Remark 2.15. Note that the visibility walk described in Section 2.4.2 is only guaranteed to converge for $y \in CH(\mathcal{D})$. In particular, if a projection $\hat{z}$ is left within floating point error of $CH(\mathcal{D})$, and the matrix $A^{(\mathcal{S})}$ for a nearby Delaunay simplex has smallest singular value $\mathcal{O}(\varepsilon)$, then it is possible for a visibility walk to fail by repeatedly calling for a flip that would lead outside of the convex hull. This is an extremely rare situation. However, in these situations, `DELAUNAYSPARSE` will first try to flip in other potential directions (i.e.,

by dropping different negatively weighted vertices). Then, if no "good" direction can be found, the correct interpolation weights must ultimately be computed by a second `DWNNLS` projection onto the current simplex.

Given the above solution, the residual is given by $r = \|z - \hat{z}\|$. When r is small with respect to the scale of the data, it is reasonable to perform extrapolation at z by interpolating at $\hat{z}$. However, when r is large it is impossible to make any reasonable prediction for $F(z)$. By default, when `DELAUNAYSPARSES` encounters an extrapolation point z, it computes the projection $\hat{z}$ and residual r using `DWNNLS`. If r is smaller than some percentage of the diameter of $\mathcal{D}$, `DELAUNAYSPARSES` resumes interpolation using $y = \hat{z}$. If r is greater than that percentage of the diameter, the extrapolation point z is skipped and an appropriate error is returned.

By default, the threshold for extrapolation is 10% of the diameter of $\mathcal{D}$, but this percentage can be adjusted using an optional input argument. Furthermore, setting this optional value to 0% of the diameter of $\mathcal{D}$ will short-circuit the extrapolation process, preventing $\hat{z}$ and r from ever being computed and preventing any `DWNNLS` work arrays from being allocated. Note that the time and space demands of `DWNNLS` can be significantly greater than those of `DELAUNAYSPARSES`. So, for large problems or in cases where computational resources are limited, it is often appropriate to turn off extrapolation by setting the extrapolation threshold to 0% of the diameter of $\mathcal{D}$.

Remark 2.16. For similar reasons as in Remark 2.15, it is possible that for poorly spaced data points $\mathcal{D}$, `DELAUNAYSPARSES` may incorrectly call for a projection of an interpolation point that is within floating point error of the boundary of $CH(\mathcal{D})$. After unnecessarily performing the projection and finding that $r = 0$, such situations are easily detected retrospectively. However, if the extrapolation threshold is set to 0% of the data diameter, the projection will short circuit and an interpolation point that is within ε of the convex hull $CH(\mathcal{D})$ could be incorrectly skipped. The conditions that could lead to such an error are pathological, but might still occur.

2.6.3 Data Scaling

Recall from Remarks 2.7, 2.10, 2.11, and 2.15 that a small scale/machine dependent constant $\varepsilon > 0$ is used to account for floating point error. Affine operations do not affect the Delaunay triangulation or interpolation results, so to account for scaling, `DELAUNAYSPARSES` rescales and shifts the data points $\mathcal{D}$ and the interpolation points $\mathcal{Q}$ on input. First, the points in $\mathcal{D}$ are shifted so that their barycenter is at the origin, then they are rescaled so that they are contained in the unit hypersphere. This ensures that ε can be chosen without accounting for data scale. The interpolation points $\mathcal{Q}$ must then be shifted and scaled by the same amounts to maintain relative positions.

After scaling, the default value ε can be chosen based only on machine precision. By default, ε is the square root of the unit round-off. This is the minimum appropriate value of ε for

most applications, and an optional argument can be used to increase ε where appropriate.

2.6.4 Memory Usage

`DELAUNAYSPARSES` uses assumed-shape arrays where appropriate. To ensure expected behavior, the dimensions of each dummy array are checked against user-specified values of d, n, and m on input. Due to the size of $DT(\mathcal{D})$ in high dimensions, the space complexity of any Delaunay triangulation algorithm is equally as important as its time complexity. The computational operations described in Section 2.4 do not require any work arrays larger than $\mathcal{O}(d^2)$, making DELAUNAYSPARSE a space efficient algorithm.

However, to take full advantage of LAPACK code optimizations, one larger work array is required. Therefore, `DELAUNAYSPARSES` uses one allocatable work array, whose size is determined at runtime based on LAPACK queries. Other allocatable work arrays of size $\mathcal{O}(nd)$ are required by `DWNNLS` for extrapolation, but are only allocated if an extrapolation is performed.

2.6.5 The Cost of Robustness and Correctness

`DELAUNAYSPARSES` is designed to be robust for a wide variety of use cases and usage errors. In particular, `DELAUNAYSPARSES` uses the diameter of $\mathcal{D}$ to judge extrapolation residuals, as discussed in Section 2.6.2. Also, the minimum pairwise distance between points in $\mathcal{D}$ is used to catch bad inputs, since after rescaling, any two points that are closer than ε will be indistinguishable from the perspective of `DELAUNAYSPARSES` and could cause hard to find bugs. The computation of the diameter and minimum pairwise distance is performed while the points are being rescaled, as discussed in Section 2.6.3. The computational complexity of these distance computations is $\mathcal{O}(n^2 d)$. Recall from Section 2.5.2 that the computational complexity of the DELAUNAYSPARSE algorithm is $\mathcal{O}(\ell n d^2)$, where ℓ appears to be independent of n for uniformly spaced $\mathcal{D}$. So the cost of interpolating at m points is $\mathcal{O}(m\ell n d^2)$. Therefore in situations where $n^2 d$ is significantly larger than $m\ell n d^2$, the complexity of `DELAUNAYSPARSES` can be dominated by nonessential distance computations, used only for robustness and extrapolation checks.

Since these computations are not necessary for the DELAUNAYSPARSE algorithm, it is possible to "turn off" exact extrapolation and error checking by setting an optional input argument. When d is relatively small and $m < n$, this can greatly improve the time complexity of the algorithm. However, doing so uses a diameter approximation of twice the distance from the barycenter of $\mathcal{D}$ to the farthest point, which could be off by up to a factor of two. Additionally, it will be possible for duplicate points to go undetected, causing difficult to find bugs and irregularities in results. For this reason, it is recommended that users only switch off these computations once they have already carefully cleaned their data

of duplicate points and when the extrapolation cutoff is flexible.

2.7 Parallel Implementation

The parallel subroutine `DELAUNAYSPARSEP` is based on the serial subroutine `DELAUNAYSPARSES` and shares the implementation details discussed in Sections 2.6.1, 2.6.2, 2.6.3, 2.6.4, and 2.6.5. `DELAUNAYSPARSEP` uses OpenMP 4.5 [79] to implement a shared memory paradigm. It is also possible to achieve distributed parallelism by breaking up the interpolation points $\mathcal{Q}$ into β separate batches $\mathcal{Q} = \mathcal{Q}^{(1)} \cup \ldots \cup \mathcal{Q}^{(\beta)}$, then distributing these batches $\mathcal{Q}^{(i)}$ across available nodes (along with copies of the data points $\mathcal{D}$). Each batch can then be evaluated independently on its corresponding node.

Remark 2.17. Recall from Section 2.6.1 that code optimizations for handling multiple interpolation points are most effective when the points are clustered in $DT(\mathcal{D})$. Therefore, optimal distributed memory performance is achieved when each $Q^{(i)}$ represents a cluster of interpolation points from $\mathcal{Q}$. Note that clustering is an open problem, and the above described distributed memory parallelism can be implemented trivially using separate calls to `DELAUNAYSPARSES` or `DELAUNAYSPARSEP`. Therefore, a distributed implementation with clustering of $\mathcal{Q}$ is not provided, and left entirely to the user.

For the OpenMP shared memory implementation, two levels of parallelism are targeted. The first level of parallelism is the loop over all m interpolation points $\mathcal{Q}$. The second level of parallelism applies to the various loops over all n data points $\mathcal{D}$ and the loop over all unresolved interpolation points, as computed by (2.6) and described in Section 2.6.1. The details for exploiting the first and second levels of parallelism are given in Sections 2.7.1 and 2.7.2, respectively. There is also a loop over the n data points for computing the scale factor (as discussed in Section 2.6.3) and a pair of nested loops over the n data points for computing the diameter and minimum pairwise distance of $\mathcal{D}$ (as discussed in Section 2.6.5). These loops can be parallelized independently of either level of parallelism using a static scheduler.

If m is small with respect to the number of available processors, level one parallelism will not saturate the available computational resources. In the extreme case where $m = 1$, level one parallelism is not available. However, when available, level one parallelism is significantly more efficient than level two parallelism. Therefore, the default behavior for `DELAUNAYSPARSEP` is to exploit level one parallelism whenever it is available (if $m > 1$), and to exploit level two parallelism otherwise (if $m = 1$). If this is not the desired behavior, the type of parallelism can be set manually via an optional argument, and for advanced users, both levels can be activated at the same time resulting in nested parallelism.

The complete code is provided in the appendix.

2.7.1 Level 1 Parallelism

The first level of parallelism is the loop over m interpolation points in $\mathcal{Q}$: **for all** $y \in \mathcal{Q}$ **do**, from Algorithm 2.5. Since the variation in the length ℓ of each visibility walk could be large, `DELAUNAYSPARSEP` uses a dynamic scheduler with a chunk size of one to parallelize this $\mathcal{Q}$ loop. The only dependencies between iterations of the $\mathcal{Q}$ loop occur when implementing the code optimizations described in Section 2.6.1.

First, when "checking ahead" for future interpolation points $y^{(j)} \in \mathcal{Q}$, there is a possible race condition since another thread could already be performing a visibility walk toward $y^{(j)}$. Since these dependencies are minimal, an OpenMP `CRITICAL` lock is used with minimal modification to the serial code to ensure safe sequentially consistent memory accesses. Once any thread of `DELAUNAYSPARSEP` has begun constructing the first simplex in the walk toward $y^{(j)}$, no other threads will continue to test $y^{(j)}$ when "checking ahead." Additionally, if daisy chaining is activated, each thread in the team must maintain a private copy of the seed simplex. Therefore, only previously constructed simplices of the active thread are considered for seeding future visibility walks.

Other than the minor issues discussed above, level one parallelism is dependency free and dynamically load balanced. Therefore, under ideal conditions, `DELAUNAYSPARSEP` is capable of weak scaling with respect to the problem dimension m, with negligible overhead.

2.7.2 Level 2 Parallelism

The second level of parallelism applies primarily to the various loops over n data points in $\mathcal{D}$: **for all** $x \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}(s^{(d)})\}$, as appear in Algorithms 2.6 and 2.13. Note that these $\mathcal{D}$ loops are not totally free of dependencies. Therefore, to parallelize the $\mathcal{D}$ loops, private copies of certain variables (such as r_{min} and x^*) must be maintained. Then, after completing these loops in parallel and producing private solutions, a reduction can be done to determine the global solutions. Note that this will result in some redundant computations. There is relatively little room for performance variation between iterations of the $\mathcal{D}$ loops, so `DELAUNAYSPARSEP` parallelizes them with a static scheduler and a fixed chunk size of $\lceil n/t^{(2)} \rceil$, where $t^{(2)}$ denotes the number of threads in each level two team.

The one exception to the above methodology is the loop over all future interpolation points, as described in Section 2.6.1. This loop is dependency free and can be parallelized using a static scheduler with a fixed chunk size of $\lceil (m-i)/t^{(2)} \rceil$, where i is the index of the current interpolation point. However, if level one parallelism is active, parallelizing this loop results in significant conflict. Therefore, in the case of nested (level one and two) parallelism, this loop executes serially within each level one thread.

Remark 2.18. There is no true dependency between iterations of the above loop over remaining interpolation points. However, the OpenMP `CRITICAL` directive used in Section

Problem Sizes (d, n)		Quickhull	Delaunay Graph
5 dimensions,	time:	3.2 sec.	58 sec.
2,000 points	space:	52 MB	10.1 MB
5 dimensions,	time:	76 sec.	1463.46 sec.
32,000 points	space:	973 MB	106 MB
6 dimensions,	time:	swap	28,296 sec.
32,000 points	space:	swap	267 MB

Table 2.2: Time and space requirements (as reported by Boissonnat et al. [14]) for computing the complete Delaunay triangulation using Quickhull and the Delaunay Graph algorithm. Entries containing the word "swap" indicate that the process exceeded RAM limitations.

2.7.1 locks code segments, as opposed to variable addresses and cannot distinguish between loop iterations, inducing a "false" conflict. As mentioned above, when level one parallelism is available, it is recommended that all available threads be devoted to level one parallelism. Therefore, in the recommended use case, this loop would not offer significant parallelism, and serializing it is no significant loss.

Due to interloop dependencies, exploiting level two parallelism can significantly increase the total number of computations performed by DELAUNAYSPARSE. Furthermore, there are significant regions of serial code separating each level two parallel block. So, the parallel efficiency of `DELAUNAYSPARSEP` with level two parallelism can be poor.

2.8 Performance

This section will focus on the runtimes of `DELAUNAYSPARSES` and `DELAUNAYSPARSEP`. For reference, first consider Table 2.2, which presents performance data (as reported in [14]) in up to six dimensions for computing the complete Delaunay triangulation of uniform randomly distributed data points in the unit cube using Quickhull [9] and the Delaunay Graph algorithm used in CGAL [14]. Boissonnat et al. gathered this data using a 2.6 GHz Intel processor with 6MB of level 2 cache and 4 GB of DDR2 RAM. The purpose of including Table 2.2 is not for direct comparison, as the problem of computing the complete Delaunay triangulation is significantly harder than that of locating a single interpolation point. Indeed, recall that standard Delaunay triangulation algorithms are not capable of scaling past six or seven dimensions for sufficiently large problems. Rather, this data is intended to clarify the issues addressed by DELAUNAYSPARSE, and inform users on when DELAUNAYSPARSE is an appropriate choice over algorithms that compute the complete Delaunay triangulation.

	Problem dimension (d)				
Problem size (n)	2	8	32	64	128
250	0.005	0.013	0.150	3.404	27.078
500	0.021	0.042	0.325	6.479	59.511
1000	0.083	0.152	0.791	14.020	124.320
2000	0.344	0.583	2.230	28.984	242.066
4000	1.314	2.284	7.165	62.494	502.620
8000	5.580	9.027	26.210	151.177	905.711
16,000	22.086	35.725	109.448	386.596	2190.362
32,000	82.915	145.115	421.934	1097.060	5024.675

Table 2.3: Serial runtimes (in seconds) for computing the Delaunay interpolant at a single interpolation point. Values shown represent the average over 20 independent trials with n pseudo-randomly generated data points in the d-dimensional unit hypercube.

2.8.1 Serial Performance

To test the performance of `DELAUNAYSPARSES`, runtimes have been gathered on AMD CPUs @2.3 GHz. Table 2.3 presents runtimes for interpolating at a single interpolation point (the center of the unit hypercube) using `DELAUNAYSPARSES`, for various problem sizes (n) and dimensions (d) with uniform randomly distributed data in the unit hypercube. To account for performance variance, each runtime represents an average over 20 independent runs of `DELAUNAYSPARSES`, each with a different data set of the same size and dimension. Note that in the higher dimensions, the data points ($\mathcal{D}$) are extremely sparse, even for large values of n. For such problems, it is typical to employ some intelligent experimental design. Therefore, in the higher dimensions, the uniform randomly spaced data used for testing becomes increasingly unrepresentative of real-world data. However, this data is sufficient for discussing how the runtime of `DELAUNAYSPARSES` scales with n and d and is comparable to the data used to generate Table 2.2. Since extrapolation presents additional computational complexities, any data set that does not contain the interpolation point ($y = [0.5, 0.5, \ldots, 0.5]^T$) in its convex hull is discarded and regenerated (an unlikely occurrence for the problem sizes shown). Note that the distance computations discussed in Section 2.6.5 cause an overall computational complexity of $\mathcal{O}(n^2)$ in the lower dimensions. However, in higher dimensions, the cost of the DELAUNAYSPARSE algorithm dominates, and the runtimes approach linear growth with respect to the number of data points n, as predicted in Section 2.4.

In general, users should expect these costs to grow linearly with m, the number of interpolation points. However, in the case where all the interpolation points are tightly clustered, the cost for interpolating at m points can be bounded by a constant times the cost for interpolating at a single point. See Tables 2 and 3 in [23]. The improved scaling for clustered interpolation points is directly caused by the code optimizations introduced in Section 2.6.1.

2.8.2 Parallel Performance

To compare the performance of `DELAUNAYSPARSEP` at all levels of parallelism against the performance of `DELAUNAYSPARSES`, runtimes have been gathered over a cluster of eight NUMA nodes with four AMD cores per node @2.3 GHz (each core identical to when timing `DELUNAYSPARSES`). Figures 2.3, 2.4, and 2.5 plot the average elapsed wallclock time (in seconds) for 20 independent runs of `DELAUNAYSPARSEP` against the number of cores used by OpenMP. The data for these experiments was generated using a randomized Latin hypercube design, as described in [3]. Then the interpolation points were generated from random convex combinations of $d + 1$ randomly selected points from the design.

Figure 2.3 presents runtimes and parallel speedup factors for interpolating at 1024 points in a 10-dimensional design with 1000 data points, reflecting workloads where m is large. Figure 2.4 presents runtimes and parallel speedup factors for interpolating at 64 points in a 10-dimensional design with 20,000 data points, reflecting workloads where n is large. Figure 2.5 presents runtimes and parallel speedup factors for interpolating at 64 points in a 50-dimensional design with 500 data points, reflecting workloads where d is large. For nested parallel runs with four, eight, 32 total active cores, the number of level one (two) threads is two, four, eight (two, four, four), respectively.

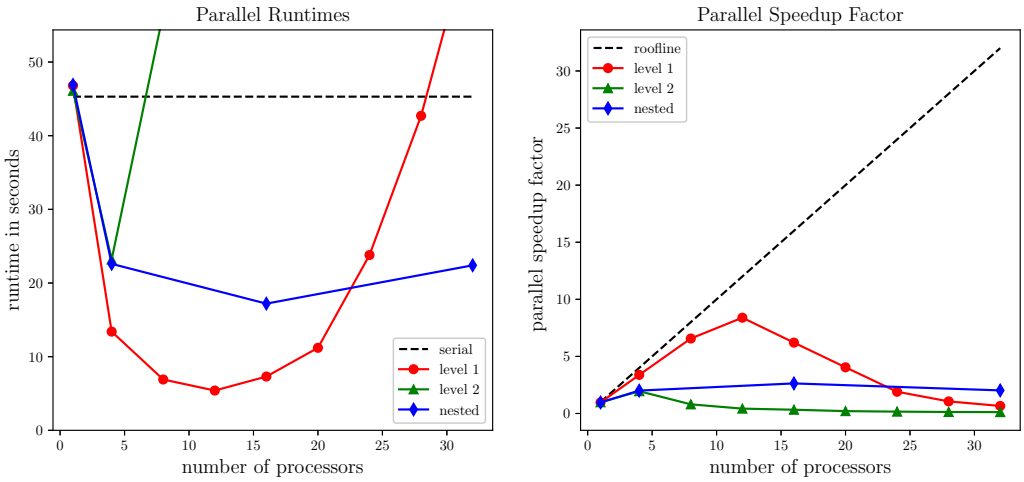

Figure 2.3: Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 1000$ and $m = 1024$.

In all three figures, level one parallelism achieves the best parallel speedup factors, as expected. In particular, for the most expensive problem (Figure 2.5), level one parallelism hugs tightly against the strong scaling roofline. It is somewhat surprising to observe that level one parallelism did not scale to more than 12 processors for the problem size used in Figure 2.3 ($d = 10$, $n = 1000$, $m = 1024$), given that the opportunities for level one parallelism are maximized for large values of m. In fact, when the cost of performing each individual

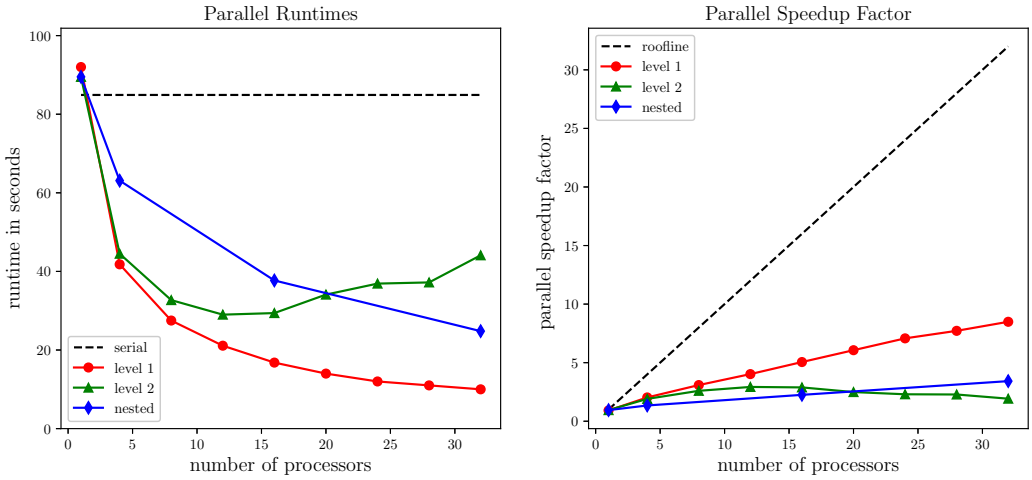

Figure 2.4: Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 10-dimensional problem, with $n = 20,000$ and $m = 64$.

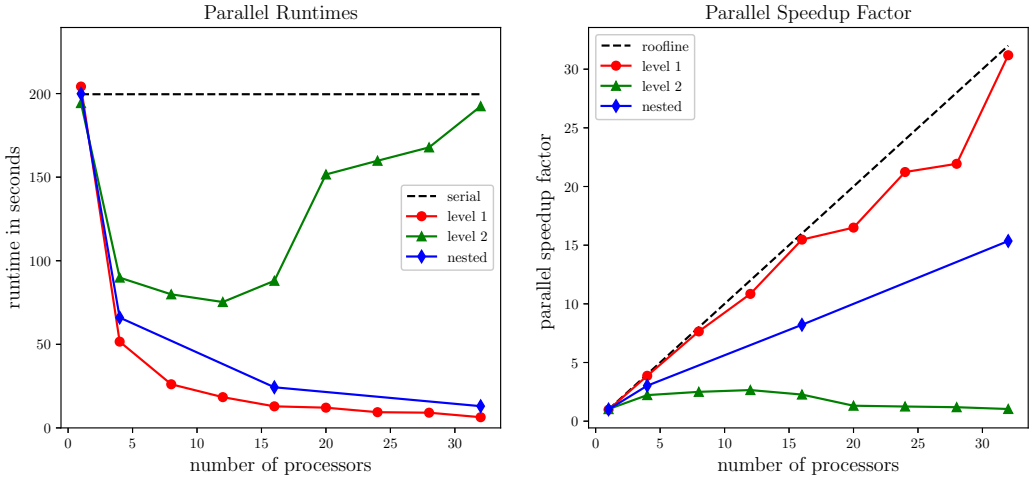

Figure 2.5: Average time to compute the Delaunay interpolant (left) and average parallel speedup factor (right), for a 50-dimensional problem, with $n = 500$ and $m = 64$.

flip is relatively small (when d and n are small), the number of interpolation points (m) is relatively large, and the thread count per contention group is relatively high, level one threads can be blocked to the point of serialization by `CRITICAL` locks during the loop to "check ahead," as described in Section 2.7.1. This presents a trade-off since checking ahead does not result in significant conflict for relatively large values of n and d and could offer significant performance benefits when $\mathcal{Q}$ is clustered (as shown in [23]). In the cases where the cost of each flip is small, it is recommended that users partition the interpolation points into β batches, $\mathcal{Q} = \mathcal{Q}^{(1)} \cup \ldots \cup \mathcal{Q}^{(\beta)}$. Then each $\mathcal{Q}^{(i)}$ can be handled by a separate call to `DELAUNAYSPARSEP` with level one parallelism and an appropriate thread count. For example, in the case of Figure 2.3, for optimal performance over 32 active cores, `DELAUNAYSPARSEP` could be called four times ($\beta = 4$), with 8 threads and 256 interpolation points per call. In general, the optimal choices for β, the thread count per batch, and the physical partition are highly problem dependent and beyond the scope of this work.

Level two parallelism provides some performance improvements for large values of n (although less than level one), but does not appear to scale well to large numbers of processors. This is due to the redundant computations that are introduced with each additional level two thread. As a usage example, a user might prefer level two parallelism in a hybrid distributed/shared memory setting. First, the interpolation points ($\mathcal{Q}$) could be broken up into extremely small batches and distributed across a large number of nodes, as suggested in Section 2.7. Then level two parallelism could be applied within each individual node, which may offer a limited number of shared memory processors.

In most cases, nested parallelism seems to offer a parallel speedup factor between that of level one and level two parallelism. Nested parallelism has a relatively limited usage case, providing a middle ground when there are multiple interpolation points (so that level one parallelism is available), but there are not enough interpolation points to fully saturate the system, making pure level one parallelism inefficient.

Chapter 3

Multiobjective Optimization and VTMOP

3.1 Introduction

This chapter formally introduces and solves the blackbox MOP. Parts of this chapter are based on work that has been published in [26].

Remark 3.1. It may be useful to review the notation in Section 1.3 before reading this chapter.

The standard form for a MOP is a real vector-valued minimization problem. The multiobjective cost function F is of the form $F : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} \subset \mathbb{R}^d$ is the *feasible design space* and $\mathcal{Y} \subset \mathbb{R}^p$ is the *feasible objective space*. This chapter considers MOPs where $\mathcal{X}$ is a *simply bounded set*, meaning that $\mathcal{X} = [L, U] = \{x \in \mathbb{R}^d : L \leqq x \leqq U\}$ for some $L, U \in \mathbb{R}^d$ with $L < U$. Then $\mathcal{Y}$ is the image of $[L, U]$ under F. Conceptually, F can be decomposed into p scalar cost functions $f_i : \mathcal{X} \to \mathbb{R}$, $i = 1, \ldots, p$ such that $F(x) = (f_1(x), \ldots, f_p(x))^T$.

The solution to a MOP is defined by the partial ordering $\leq$ on $\mathcal{Y}$. An objective value $F(x^*)$ is said to be *nondominated* if $F(x) \not\leq F(x^*)$ for all $x \in \mathcal{X}$. If $F(x^*)$ is nondominated, then x^* is said to be *efficient* and the pair $(x^*, F(x^*))$ is *Pareto optimal*. The *Pareto front* is given by the set of all nondominated objective points, and often accompanied by the set of all efficient designs, called the *efficient set*. The Pareto front has dimensionality of at most one less than the number of objectives and could be nonsmooth or even discontinuous. Further reading on MOPs can be found in [46].

A scalar function $g : \mathbb{R}^d \to \mathbb{R}$ is said to be a *blackbox* when g can only be evaluated to obtain function values, and no additional information (such as gradients) can be obtained. Often, blackbox functions are the result of computationally expensive numerical simulations, requiring substantial computing resources and time for each evaluation. The multiobjective cost function F is a blackbox if each component function f_i is a blackbox function. In some cases, this could imply that each component f_i must be evaluated separately at great computational expense. In other cases, all components of F may be obtained through a single integrated simulation.

This chapter presents the VTMOP software package for solving computationally expensive

blackbox MOPs, subject to simple bound constraints. Distinctive features of VTMOP are that it (1) handles any $p \geqq 2$, (2) is derivative free by default but could be adapted to use derivative information, and (3) devotes considerable effort to produce well-distributed points over the Pareto front. VTMOP also features a solver subroutine that can be used to solve MOPs when F is available as a Fortran callable function; a "return-to-caller" interface that can be used when F must be decoupled from the optimization algorithm; both parallel and serial execution paradigms; a checkpointing system for storing and recovering function evaluation and iteration data; and support for user-defined local optimization and surrogate modeling procedures.

The chapter is organized as follows. Section 3.2 provides some background on multiobjective optimization algorithms (MOAs) and widely distributed software packages. Section 3.3 outlines the VTMOP algorithm and worker subroutines. Section 3.4 presents two interfaces for solving blackbox MOPs using VTMOP: a return-to-caller interface and a driver subroutine, both of which use the worker subroutines outlined in Section 3.3. Section 3.5 describes how VTMOP is integrated with the `libEnsemble` library for coordinating resource utilization for extreme scale computing. Section 3.6 provides performance results for VTMOP on several analytic test problems. Section 3.7 provides details on the parallel performance of several variations of VTMOP.

3.2 Literature Review

MOPs arise in many disciplines of science and engineering. The field of multidisciplinary design optimization (MDO) [90] is an excellent motivating example. In MDO problems, a multidisciplinary team of engineers must collaborate to design a large complex system, such as an aircraft, balancing design tradeoffs spanning many different subfields of engineering. In this context, the Pareto front describes the interactions between several conflicting design criteria. Understanding the shape of the Pareto front allows the design engineers to make an informed decision on how to balance design tradeoffs *a posteriori*, i.e., based on the results of the design optimization. For specific examples of MOPs in engineering, [19] presents a ship hull design problem and [102] describes a chemical engineering problem.

This section presents a review of MOAs and techniques that can be used to solve blackbox MOPs of the form

$$\min_{x \in [L,U]} F(x), \tag{3.1}$$

where the minimization is understood as Pareto optimality. In general, the solution to (3.1) is an uncountably infinite set, describing a $(p-1)$-dimensional manifold in the objective space. The MOAs presented in this section produce a set of approximately nondominated objective points (and corresponding efficient designs), which provide a discrete approximation to the Pareto front.

3.2.1 Multiobjective Optimization Techniques and Algorithms

There are three fundamentally different approaches to solving MOPs. The first approach consists of *a priori* methods, where the decision makers are able to express some preference about the tradeoff **before** viewing the results of the optimization. In these cases, the decision maker typically supplies a preference function, which reduces the MOP to a scalar optimization problem. The second class of methods consist of *a posteriori* methods, where the decision maker is not able to express any sort of preference until **after** viewing the results of the optimization procedure. This is the most general class of methods and the subject of this chapter. The final class of methods consists of "human in the loop" algorithms, where a decision maker is able to progressively provide feedback on preference while the algorithm is running. A full survey summarizing all three of these techniques can be found in [70]. A recent detailed survey of *a posteriori* techniques for computationally expensive blackbox MOPs is given by [55].

The classical *a posteriori* technique for solving blackbox MOPs is to use multiple *scalarization* functions. Scalarization reduces a MOP to a scalar optimization problem by composing a *scalarization function* $G : \mathbb{R}^p \to \mathbb{R}$ with F. The resulting function $G \circ F$ can be minimized by using a scalar blackbox optimization solver to find a single (approximately) nondominated point. Optimizing different scalarizations produces a discrete set of approximately nondominated points, thus approximating the Pareto front. Further reading on general scalarization schemes can be found in Chapter 4 of [46]. One common scalarization scheme is the weighted sum method, which uses a vector of nonnegative weights $w = (w_1, \ldots, w_p)$ to produce a "weighted average" cost function

$$G_w(F(x)) = \sum_{i=1}^{p} w_i f_i(x). \tag{3.2}$$

When $w > 0$, the solution to (3.2) is Pareto optimal. However, when the Pareto front manifold is nonconvex, not all Pareto points can be achieved by (3.2). Figure 3.1 illustrates this limitation in a simple example with $p = 2$. Furthermore, when applied naïvely to a set of uniformly distributed weights, (3.2) tends to produce objective values that form clusters and leave gaps along the Pareto front, giving a poor understanding of its true shape. Therefore, effective weighted sum scalarization depends on an *adaptive weighting scheme*. Further reading on weighted sum scalarization can be found in Chapter 3 of [46].

One issue with scalarization techniques is that each subproblem must be solved individually as a blackbox scalar optimization problem. Therefore, the cost of solving the MOP is many times that of solving each scalar subproblem. When F is computationally expensive to evaluate, this can be prohibitive. The response surface methodology (RSM) can be used to mitigate these expenses [76]. In multiobjective RSM, a computationally cheap (to evaluate) surrogate function is fit to each objective function using experimental designs. Then multiple scalarizations of these surrogates are optimized, each producing a candidate design point. If the design of experiments is thorough and the surrogates are accurate, then evaluating

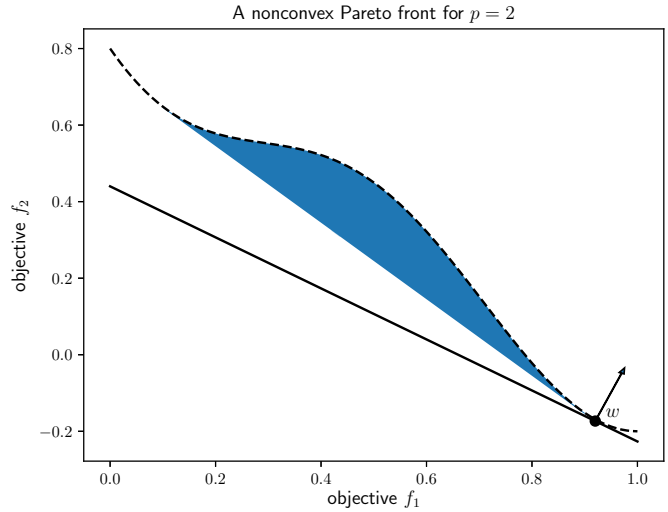

Figure 3.1: An illustration of one of the drawbacks of weighted sum scalarization. The dashed line depicts a nonconvex Pareto front for $p = 2$, The solid line depicts the supporting hyperplane (in this case a line) whose normal is w, and the solution to (3.2) is marked at the point of tangency between the hyperplane and the Pareto front. Note that there is no vector w that can attain solutions in the shaded region.

these candidate designs will produce numerous points approximately on the Pareto front. Thus many Pareto points can be found for little more than the cost of evaluating a single experimental design.

In modern industrial settings, evolutionary MOAs are perhaps the most widely used class of MOAs. These algorithms extend evolutionary algorithms to the multiobjective case by using a fitness sorting criterion based on the partial ordering $\leq$. An *elitist* evolutionary MOA also maintains previously observed nondominated solutions, using them to ensure that convergence to the true Pareto front is monotone. Further reading on evolutionary MOAs can be found in [1]. Evolutionary MOAs have an advantage over other MOAs in that they are capable of converging to the entire Pareto front concurrently. However, the function evaluation budget requirement for evolutionary MOAs tends to be too large for computationally expensive problems. For the test problems presented in [39], the recommended budget for an evolutionary algorithm is between 20,000–50,000 function evaluations. For a computationally expensive problem, each function evaluation could require minutes, hours, or days and occupy massive parallel resources. In these situations, such a budget would be prohibitively large. For computationally expensive problems, evolutionary MOAs are often combined with RSM to reduce computational expense [47, 75].

Another prevalent class of algorithms is direct search MOAs. These MOAs extend scalar direct search methods, such as the algorithm dividing rectangles (DIRECT) [58] and mesh

adaptive direct search (MADS) [5]. Two examples of these algorithms are multiobjective DIRECT (MODIR) and multiobjective MADS (MultiMADS). The algorithm MODIR extends DIRECT to the multiobjective case [19]. MODIR offers the same global convergence guarantees as DIRECT, but requires an unreasonably large number of function evaluations in practice. Therefore, [19] recommends combining MODIR with a locally convergent MOA, such as the derivative free multiobjective line search algorithm of [65]. MultiMADS was proposed by Audet et al. [6] and extends MADS to the multiobjective case. MultiMADS does this by combining MADS with the normal boundary intersection scalarization technique of [34], which sets evenly spaced goal points on the Pareto front.

This thesis implements the MOA of Deshpande et al. [41], which combines an adaptive weighting scheme for weighted sum scalarization, RSM, and trust-region methods (TRMs). TRMs are widely used in scalar optimization [77]. In the context of RSM, each local trust region (LTR) defines a region of the design space in which the current response surface approximation is accurate enough for usage as a surrogate. In the context of MOPs, these LTRs can be used to control the spacing of points on the Pareto front and force solutions in nonconvex portions of the Pareto front when combined with weighted sum scalarization [87].

3.2.2 Multiobjective Optimization Software

Of the above techniques and algorithms, there are relatively few widely distributed software packages for solving MOPs. When it comes to computationally expensive problems, many simulation packages support multiobjective design optimization by solving a single scalarized subproblem, producing a single Pareto optimal design. For example, NASA's FUN3D package performs fluid dynamics based design simulations and provides support for multiobjective analyses by using the design drivers KSOPT, PORT, or SNOPT to solve a single scalarized subproblem, as described in Section 9.9 of [12]. This approach falls under the *a priori* techniques mentioned in Section 3.2.1. While this approach is reasonable for the extremely limited evaluation budget of most computational fluid dynamics based analyses [12] recommends a budget of just 20 simulations), it results in no understanding of the complex design tradeoffs and is less general than the *a posteriori* approaches favored here.

When it comes to *a posteriori* methods, the optimizer `BiMADS` is a variation of MultiMADS that is publicly available and widely used for blackbox engineering problems with exactly $p = 2$ objectives (the biobjective case). While there is currently no publicly available implementation of the algorithm MultiMADS, the `BiMADS` solver is available in the NOMAD software package [63]. `BiMADS` takes advantage of the natural ordering of the Pareto front in the case where $p = 2$, when the Pareto front is a one-dimensional curve embedded in the objective space. However, this solution does not generalize when $p > 2$.

For blackbox MOPs with $p > 2$ objectives, the widely used software packages implement variations of evolutionary MOAs. One widely used and representative MOA is the non-

dominated sorting genetic algorithm NSGA-II [38], an elitist evolutionary MOA with low iteration complexity. Because of its large function evaluation requirement but low iteration costs, NSGA-II is most effective when the multiobjective cost function is a blackbox but is not computationally expensive.

There are several other software packages that are worth mentioning, but not immediately relevant to this work. The Python package `PyMOSO` [32] solves multiobjective simulation optimization problems on an integer lattice and provides a Python framework for implementing new multiobjective simulation optimization algorithms. Additionally, [2] provide a Matlab toolbox for applying surrogate modeling to MOPs.

At the time of this survey, there are no widely available production quality solvers that

- generalize to an arbitrary number $p \geqq 2$ of objectives;
- scale to computationally expensive blackbox problems;
- produce well-distributed points on the Pareto front; and
- take an *a posteriori* approach.

Thus, VTMOP fills a need for a scalable multiobjective software package capable of solving computationally expensive blackbox MOPs.

3.3 A Framework for Multiobjective Optimization

The MOA of Deshpande et al. [41] applies RSM in a sequence of LTRs, each centered at a design point corresponding to an "isolated" objective value from the current set of approximately nondominated points. The algorithm begins from the zeroth iteration, and the kth iteration of the algorithm can be summarized by the following four step process. Detailed descriptions of Steps 1, 2, 3, and 4 are presented in Sections 3.3.1, 3.3.2, 3.3.3, and 3.3.4 (respectively). Note that evaluations of F are only required during Steps 2 and 4.

1. If $k > 0$, update the current set of nondominated points, then identify an isolated point $F(\tilde{x}^{(k)})$ in the current set of nondominated points; center the kth LTR $\Delta^{(k)}$ about $\tilde{x}^{(k)}$; and assign the kth set $\mathcal{W}^{(k)}$ of adaptive weights based on objective values near $F(\tilde{x}^{(k)})$. If $k = 0$, then there is no current nondominated set; $\Delta^{(0)} = [L, U]$ and $\mathcal{W}^{(0)}$ is assigned predetermined values.

2. Perform the kth exploration by sampling F within $\Delta^{(k)}$; if $k = 0$, then a large exploration budget may be necessary.

3. Fit p surrogates, $\hat{f}_i^{(k)} \approx f_i$ for $i = 1, \ldots, p$, using the data gathered from Step 2 (plus any data available from previous iterations if $k > 0$). Apply each weight vector in $\mathcal{W}^{(k)}$ to

$\hat{f}_1^{(k)}, \ldots, \hat{f}_p^{(k)}$, and minimize the resulting surrogate problems using a local optimization strategy. This produces the kth batch $\mathcal{C}^{(k)}$ of candidate design points.

4. Evaluate $F(z)$ for all $z \in \mathcal{C}^{(k)}$. Increment k. Update the current database of function values, check for termination conditions, and if no termination conditions have been triggered, proceed to the next iteration.

3.3.1 Computing the Local Trust Region & Choosing Adaptive Weights

In order to construct the kth LTR and select the adaptive weights when $k > 0$, VTMOP must identify an "isolated point" in the current nondominated point set. VTMOP centers the kth LTR at a design point, in the current approximation to the efficient set, whose image is "far away" from any neighbors in the objective space. This enables VTMOP to spend function evaluations where they are needed most and fill in gaps on the current approximation to the Pareto front. When $p = 2$, the Pareto front is a 1-dimensional curve, which admits a natural (left-to-right) ordering. Ryu et al. [87] identify an isolated point by considering the Euclidean distance from each point in the nondominated set to its left and right neighbors. However, this approach does not generalize to $p > 2$.

Deshpande et al. [41] generalize this approach to arbitrary dimension using the Delaunay graph. Let $\mathcal{D}^{(k)} = \{x^{(k,1)}, \ldots, x^{(k,n^{(k)})}\}$ denote the set of $n^{(k)}$ design points that have been evaluated at the start of iteration k, and let $\mathcal{F}^{(k)} = \{F(x^{(k,1)}), \ldots, F(x^{(k,n^{(k)})})\}$. Then the kth approximation to the Pareto front is

$$\mathcal{P}^{(k)} = \{Y \in \mathcal{F}^{(k)} : X \not\leq Y \text{ for all } X \in \mathcal{F}^{(k)}\}, \tag{3.3}$$

and the kth approximation to the efficient set is

$$\mathcal{E}^{(k)} = \{x^{(k,i)} : F(x^{(k,i)}) \in \mathcal{P}^{(k)} \text{ and } F(x^{(k,i)}) = F(x^{(k,j)}) \Rightarrow i \leqq j\}. \tag{3.4}$$

Because the Pareto front is (generically) a $(p-1)$-dimensional manifold, the first step is to project $\mathcal{P}^{(k)}$ into its natural dimension. Let $\mathcal{P}^{(k)} = \{Y^{(k,1)}, \ldots, Y^{(k,m^{(k)})}\}$ where $m^{(k)}$ is the number of unique objective points in the kth approximately nondominated set; let $Y_1^{(k,i)}$, $\ldots$, $Y_p^{(k,i)}$ denote the components of $Y^{(k,i)}$; and let $s^{(k)}$ be a constant such that $s^{(k)} < Y_p^{(k,i)}$ for $i = 1, \ldots, m^{(k)}$. Then the $(p-1)$-dimensional projected set is given by

$$\Pi^{(k)} = \{Z^{(k,1)}, \ldots, Z^{(k,m^{(k)})}\},\ Z^{(k,i)} = \left(\frac{Y_1^{(k,i)}}{Y_p^{(k,i)} - s^{(k)}}, \ldots, \frac{Y_{p-1}^{(k,i)}}{Y_p^{(k,i)} - s^{(k)}}\right). \tag{3.5}$$

Remark 3.2. In order to avoid division by zero or a small number when computing (3.5), $s^{(k)}$ must be chosen small enough so that $Y_p^{(k,i)} - s^{(k)}$ is significantly greater than 0 for all $i = 1, \ldots, m^{(k)}$. In VTMOP, this is done by choosing $s^{(k)} = \min_{Y \in \mathcal{P}^{(k)}} Y_p - 1$.

Remark 3.3. Duplicate values are determined using distance in the projected space in order to prevent $\Pi^{(k)}$ from containing duplicate values, which would present problems in the coming computations. Specifically, VTMOP computes (3), (4), and (5) at the same time, and for any $Z^{(k,i)}$ that satisfies $\|Z^{(k,i)} - Z^{(k,j)}\|_2 < \epsilon$, the corresponding values $Y^{(k,i)}$ and $Y^{(k,j)}$ are treated as duplicates. In particular, if $j < i$, then $Z^{(k,i)}$ is omitted from $\Pi^{(k)}$, $Y^{(k,i)}$ is omitted from $\mathcal{P}^{(k)}$, and $F^{-1}(Y^{(k,i)})$ is omitted from $\mathcal{E}^{(k)}$. Here ϵ is the *objective space tolerance*, a small scale and machine dependent constant that is an optional input for VTMOP.

After computing the projected set $\Pi^{(k)}$, its *Delaunay graph* $DG(\Pi^{(k)})$ is used to infer a neighborhood structure. Further details on the Delaunay graph and a novel algorithm for computing it, based on DELAUNAYSPARSE, are provided in the next subsection. From $DG(\Pi^{(k)})$, an isolation score is computed for each $Y^{(k,i)} \in \mathcal{P}^{(k)}$. Let $\mathcal{N}^{(k,i)} = \{Y^{(k,j)} : Z^{(k,j)}$ is a neighbor of $Z^{(k,i)}$ in $DG(\Pi^{(k)})\}$. Then the isolation score for $Y^{(k,i)}$ is

$$\delta(Y^{(k,i)}) = \sum_{Y \in \mathcal{N}^{(k,i)}} \frac{\|Y^{(k,i)} - Y\|_2}{|\mathcal{N}^{(k,i)}|}. \tag{3.6}$$

In general, the kth LTR is centered at a design point $\tilde{x}^{(k)}$ that is chosen so that $F(\tilde{x}^{(k)}) = \arg\max_{Y \in \mathcal{P}^{(k)}} \delta(Y)$, whose neighborhood $\tilde{\mathcal{N}}^{(k)}$ is the corresponding $\mathcal{N}^{(k,i)}$. In case the Y from the arg max is not unique, choose the $Y^{(k,i)}$ with smallest i.

Remark 3.4. In rare cases, it is possible that $m^{(k)} = 1$. Then $\tilde{x}^{(k)}$ is given by the only element of $\mathcal{E}^{(k)}$ and $\tilde{\mathcal{N}}^{(k)} = \emptyset$.

Let $\tilde{r}^{(k)} \in (0, 1)$ denote the kth LTR radius (as a fraction of $U - L$). Then the kth LTR is given by

$$\Delta^{(k)} = [\ \tilde{x}^{(k)} - \tilde{r}^{(k)}(U - L),\ \tilde{x}^{(k)} + \tilde{r}^{(k)}(U - L)\ ] \cap [L, U]. \tag{3.7}$$

Note that $\Delta^{(k)}$ is a simply bounded set.

Remark 3.5. Ideally, performing the kth iteration within $\Delta^{(k)}$ will generate design data in the neighborhood of $\tilde{x}^{(k)}$ and fill in the Pareto front in the neighborhood of $F(\tilde{x}^{(k)})$. However, when the Pareto front is nonsmooth or discontinuous, VTMOP may fail to produce any approximately nondominated points that are near $F(\tilde{x}^{(k)})$ in the objective space. In these cases, $\tilde{x}^{(k)}$ could be a reoccurrence of a previous LTR center. Every time $\|\tilde{x}^{(k)} - \tilde{x}^{(k')}\|_2 < \mu$ for some $k' < k$, the fraction $\tilde{r}^{(k)}$ is decayed using $\tilde{r}^{(k)} = \tau\, \tilde{r}^{(k^*)}$, where $0 < \tau < 1$, and k^* is the largest such k'. Here μ is the *design space tolerance*, a small scale and machine dependent constant. If $\tilde{r}^{(k)}$ drops below some predetermined tolerance, then $\tilde{x}^{(k)}$ is skipped and the next most isolated point is used instead. For every **new** value of $\tilde{x}^{(k)}$, $\tilde{r}^{(k)}$ is initialized to a default value, regardless of whether the LTR radius has been decayed in previous iterations. The default trust region radius fraction, the decay factor τ, the trust region tolerance, and μ are all optional inputs to VTMOP.

The zeroth iteration is a special case. VTMOP allows users to (optionally) supply a database of previously evaluated designs. However, often there is no data available at the start of the zeroth iteration, and therefore there is no $\mathcal{E}^{(0)}$ or $\mathcal{P}^{(0)}$. Instead, $\Delta^{(0)} = [L, U]$ resulting in an exploration of the entire design space. When F is nonconvex, the global convergence of VTMOP is entirely dependent on a thorough design space exploration during the zeroth iteration.

After computing $\Delta^{(k)}$, the kth set of adaptive weights is selected. When $k = 0$ or if $|\tilde{\mathcal{N}}^{(k)}| < 1$, then the adaptive weights are given by $\mathcal{W}^{(k)} = \{e^{(1)}, \ldots, e^{(p)}, \sum_{i=1}^{p} e^{(i)}/p\}$ where $e^{(i)}$ is the ith standard basis vector in $\mathbb{R}^p$. Otherwise, $|\mathcal{W}^{(k)}| = p + |\tilde{\mathcal{N}}^{(k)}|$, and the values of each weight vector are determined as described in a subsequent subsection. The entire process of identifying an isolated point, constructing the kth LTR, and choosing the adaptive weights is summarized in Algorithm 3.6.

Algorithm 3.6.
input
k is the current iteration of VTMOP;
$[L, U]$ is the feasible design space;
$\mathcal{D}^{(k)}$ is the kth design database, containing $n^{(k)}$ design points that were evaluated in previous iterations of VTMOP (if $k = 0$, then $\mathcal{D}^{(0)}$ could contain designs that were evaluated during previous analyses or $\mathcal{D}^{(0)} = \emptyset$);
$\mathcal{F}^{(k)} = \{F(x) : x \in \mathcal{D}^{(k)}\}$;
$\rho^{(0)}$ is the default trust region radius (as a fraction of $U - L$);
$\rho^{(1)}$ is the trust region tolerance (as a fraction of $U - L$);
τ is the decay factor for the trust region radius;
μ is the design space tolerance;
If $k > 1$, then $\tilde{r}^{(1)}, \ldots, \tilde{r}^{(k-1)}$ are the previous trust region radii fractions;
If $k > 1$, then $\tilde{x}^{(1)}, \ldots, \tilde{x}^{(k-1)}$ are the previous trust region centers;
begin
if $k = 0$ **then**
  $\Delta^{(k)} \leftarrow [L, U]$;
  $\mathcal{W}^{(k)} \leftarrow \{e^{(1)}, \ldots, e^{(p)}, \sum_{i=1}^{p} e^{(i)}/p\}$;
else
  $\mathcal{P}^{(k)} \leftarrow \{Y \in \mathcal{F}^{(k)} : X \not\leq Y \text{ for all } X \in \mathcal{F}^{(k)}\} = \{Y^{(k,i)} = F(y^{(k,i)}) \mid 1 \leqq i \leqq m^{(k)}\}$ as in (3.3);
  $\mathcal{E}^{(k)} \leftarrow$ subset of $\mathcal{D}^{(k)}$ as in (4);
  Compute $\Pi^{(k)}$ using (3.5);
  $m^{(k)} \leftarrow |\Pi^{(k)}|$; **comment** The cardinality $|\mathcal{P}^{(k)}| = |\mathcal{E}^{(k)}| = |\Pi^{(k)}|$ may have changed.
  Compute $DG(\Pi^{(k)})$ using Algorithm 3.7;
  for $i = 1, \ldots, m^{(k)}$ **do**
    Compute $\mathcal{N}^{(k,i)}$ based on $DG(\Pi^{(k)})$;
    Compute $\delta(Y^{(k,i)})$ using (3.6);
  enddo

initialize $J^{(k)} \leftarrow \{1, \ldots, m^{(k)}\}$; $\tilde{x}^{(k)} \leftarrow null$;
while $\tilde{x}^{(k)} = null$ **and** $J^{(k)} \neq \emptyset$ **do**
$i^* = \min \arg\max_{i \in J^{(k)}} \delta(Y^{(k,i)})$;
if $\|\tilde{x}^{(k')} - y^{(k,i^*)}\|_2 < \mu$ for any $k' < k$ **then**
$k^* \leftarrow \max\{k' \mid k' < k \text{ and } \|\tilde{x}^{(k')} - y^{(k,i^*)}\|_2 < \mu\}$;
if $\tau\, \tilde{r}^{(k^*)} > \rho^{(1)}$ **then**
$\tilde{r}^{(k)} \leftarrow \tau\, \tilde{r}^{(k^*)}$;
$\tilde{x}^{(k)} \leftarrow y^{(k,i^*)}$;
$\tilde{\mathcal{N}}^{(k)} \leftarrow \mathcal{N}^{(k,i^*)}$;
else
$J^{(k)} \leftarrow J^{(k)} \setminus \{i^*\}$;
endif
else
$\tilde{r}^{(k)} \leftarrow \rho^{(0)}$;
$\tilde{x}^{(k)} \leftarrow y^{(k,i^*)}$;
$\tilde{\mathcal{N}}^{(k)} \leftarrow \mathcal{N}^{(k,i^*)}$;
endif
enddo
if $\tilde{x}^{(k)} = null$ **then**
All candidate LTR centers have been maximally refined, which is a termination condition for VTMOP;
return an appropriate termination message;
endif
$\Delta^{(k)} \leftarrow [\tilde{x}^{(k)} - \tilde{r}^{(k)}e,\, \tilde{x}^{(k)} + \tilde{r}^{(k)}e] \cap [L, U]$ as in (3.7);
if $|\tilde{\mathcal{N}}^{(k)}| < 1$ **then**
$\mathcal{W}^{(k)} \leftarrow \{e^{(1)}, \ldots, e^{(p)}, \sum_{i=1}^{p} e^{(i)}/p\}$;
else
Compute $\mathcal{W}^{(k)}$ using (3.8);
endif
endif
return $\Delta^{(k)}, \mathcal{W}^{(k)}$;

The dominant costs for Algorithm 3.6 are computing $\mathcal{P}^{(k)}$ using (3.3) and computing $DG(\Pi^{(k)})$ using Algorithm 3.7. The $\mathcal{O}$ time complexity for computing (3.3) is a quadratic function of $n^{(k)}$. In Algorithm 3.7 (below), a novel algorithm is proposed for computing the $DG(\Pi^{(k)})$ by using DELAUNAYSPARSE in a nontraditional way. The $\mathcal{O}$ time complexity of this algorithm is a cubic function of $m^{(k)}$ in practice. When F is an analytic function, these could be significant iteration costs. However, when F is a computationally expensive blackbox function, these costs are insignificant. In the package VTMOP, the subroutine `VTMOP_LTR` implements Algorithm 3.6.

Computing the Delaunay Graph

$DG(\Pi^{(k)})$ is defined in terms of the Delaunay triangulation. The node set for $DG(\Pi^{(k)})$ is given by the set of all Delaunay vertices, and the edge list is given by connecting all vertices that share a 1-face (i.e., an edge) in $DT(\Pi^{(k)})$.

In the literature, the Delaunay graph is computed as a biproduct of the Delaunay triangulation [14]. However, due to the exponential size of Delaunay triangulations in high-dimensional spaces (as discussed in Chapter 2), this approach becomes prohibitively expensive for $p > 6$ objectives. Another strategy that has been investigated during this thesis is to compute each Voronoi polytope individually [23], allowing for each node's neighborhood to be constructed separately and in parallel. However, this approach ultimately requires more total computations than when constructing the complete Delaunay triangulation, and is not robust when the input data exhibits geometrical degeneracies. Instead, VTMOP implements a novel strategy for constructing the connectivity matrix for the Delaunay graph; this strategy is capable of scaling to large values of p (i.e., $p \approx 100$ objectives). The pseudo code is given in Algorithm 3.7.

Algorithm 3.7.
input
$\Pi^{(k)} = \{Z^{(k,1)}, \ldots, Z^{(k,m^{(k)})}\}$ with each $Z^{(k,i)}$ defined as in (3.5);
begin
$M^{(k)}$ is a Boolean valued matrix of dimensions $m^{(k)} \times m^{(k)}$;
initialize

$$M^{(k)} \leftarrow \begin{bmatrix} FALSE & \ldots & FALSE \\ \vdots & & \vdots \\ FALSE & \ldots & FALSE \end{bmatrix};$$

for $i = 1, \ldots, m^{(k)}$ **do**
  for $j = i + 1, \ldots, m^{(k)}$ **do**
    $\bar{Z}^{(k,i,j)} \leftarrow$ the midpoint between $Z^{(k,i)}$ and $Z^{(k,j)}$;
    $\mathcal{V}^{(k,i,j)} \leftarrow$ the vertex set for a Delaunay simplex containing $\bar{Z}^{(k,i,j)}$;
    if $Z^{(k,i)} \in \mathcal{V}^{(k,i,j)}$ **and** $Z^{(k,j)} \in \mathcal{V}^{(k,i,j)}$ **then**
      $M^{(k)}_{i,j} \leftarrow TRUE$;
      $M^{(k)}_{j,i} \leftarrow TRUE$;
    endif
  enddo
enddo
return $M^{(k)}$

The key cost for Algorithm 3.7 is that for computing $\mathcal{V}^{(k,i,j)}$. VTMOP uses DELAUNAYSPARSE to compute each of these vertex sets. Since the input to DELAUNAYSPARSE in this use case is the projected set, one should replace "d" with "$p-1$" and "n" with "$m^{(k)}$"

when considering the analysis in Section 2.5.2, resulting in a time complexity of $\mathcal{O}(\ell m^{(k)} p^2)$ per simplex constructed. Assuming that ℓ is independent of $m^{(k)}$ in the average case, the execution time for Algorithm 3.7 is a cubic function of $m^{(k)}$, regardless of the number of objectives p.

Algorithm 3.7 correctly computes the Delaunay graph when $\Pi^{(k)}$ is in general position. This is established in Theorem 3.8 whose proof is in the appendix.

Theorem 3.8. *If $\Pi^{(k)}$ is in general position, then Algorithm 3.7 computes the connectivity matrix for $DG(\Pi^{(k)})$.*

Remark 3.9. If $DG(\Pi^{(k)})$ is not unique, then Algorithm 3.7 may fail to produce a connectivity structure that is consistent with any Delaunay triangulation. However, the *Gabriel graph* is a subgraph of every Delaunay triangulation, and therefore the Gabriel graph will always be embedded in the connectivity structure computed by Algorithm 3.7, regardless of uniqueness.

Remark 3.10. If the Delaunay triangulation does not exist, then $\Pi^{(k)}$ is contained in a $(p-2)$-dimensional linear manifold. Typically, this only occurs when $m^{(k)} < p$. Then all points in $\Pi^{(k)}$ are considered neighbors, and the most isolated point corresponds to the projected point that is farthest away from other points in $\Pi^{(k)}$. One extremely rare case is that $m^{(k)} \geq p$, but $\Pi^{(k)}$ is contained in a $(p-2)$-dimensional linear manifold due to the Pareto front being a $(p-2)$- or lower-dimensional manifold embedded in the objective space. In this case, a principle component analysis is performed, by using the `DGESVD` subroutine from LAPACK [4] to compute the singular value decomposition of the matrix whose columns are points in $\Pi^{(k)}$ after they have been shifted so that their barycenter is the origin. After computing the decomposition, $\Pi^{(k)}$ is projected onto the span of the left singular vectors whose corresponding singular values are greater than the objective space tolerance ϵ (from Remark 3.3). Then $DG(\Pi^{(k)})$ is given by computing the Delaunay graph in this reduced dimensional space.

Algorithm 3.7 will most likely be of interest independently of VTMOP. Current state-of-the-art algorithms [14] and software [53] for computing the Delaunay graph do not scale past 12 dimensions. Therefore, a standalone subroutine `DELAUNAYGRAPH` is provided, implementing Algorithm 3.7.

Choosing the Adaptive Weights

Deshpande et al. [41] assign a set $\mathcal{W}^{(k)}$ of $p + |\tilde{\mathcal{N}}^{(k)}|$ weight vectors in the kth iteration, given that $k > 0$ and $|\tilde{\mathcal{N}}^{(k)}| > 0$. The first p of these weight vectors are always the standard basis vectors $e^{(1)}, \ldots, e^{(p)}$. By searching for the individual minimum of each component function in every iteration, VTMOP expands the coverage of its approximation to the Pareto front.

The remaining $|\tilde{\mathcal{N}}^{(k)}|$ weight vectors are assigned based on the objective scaling in order to fill in gaps in the current approximation to the Pareto front. For each $\tilde{Y} \in \tilde{\mathcal{N}}^{(k)}$, the corresponding adaptive weight vector $\tilde{W}$ is given by

$$\tilde{W}_i = \begin{cases} 0, & \text{if } |f_i(\tilde{x}^{(k)}) - \tilde{Y}_i| < \epsilon, \\ \dfrac{\tilde{c}}{|f_i(\tilde{x}^{(k)}) - \tilde{Y}_i|}, & \text{otherwise,} \end{cases} \tag{3.8}$$

where $\tilde{c}$ is a normalizing constant such that $\tilde{W}_1 + \ldots + \tilde{W}_p = 1$.

If $k = 0$ or $|\tilde{\mathcal{N}}^{(k)}| = 0$, then there are no "gaps" to fill in. So, VTMOP focuses on expansion, optimizing each individual component function by applying pure weight vectors $e^{(1)}, \ldots, e^{(p)}$. Additionally, an equal weighting of all objectives $\sum_{i=1}^{p} e^{(i)}/p$ is applied.

Remark 3.11. Certain components of $\tilde{W}$ in (3.8) and the basis vectors $e^{(1)}, \ldots, e^{(p)}$ contain zero values. However, recall that (3.2) is only guaranteed to produce a Pareto optimal point when $w > 0$. Therefore, after computing the weights in this section, all zero valued components are replaced by a small machine dependent "fudge factor" and the resulting weights are renormalized. The magnitude of this fudge factor is an optional input to VTMOP.

3.3.2 Searching the Design Space and Trust Regions

After Algorithm 3.6, VTMOP performs an exploration of $\Delta^{(k)}$, which is the first step in the multiobjective RSM framework. Depending on whether $k = 0$, $\Delta^{(k)}$ could either be the entire feasible design space $[L, U]$ or the kth LTR, as computed in (3.7).

Remark 3.12. The number of cost function evaluations used during this exploration should be much larger when $k = 0$ than when $k > 0$. There are two reasons for this. First, the feasible design space could be very large, so a large number of function evaluations may be required for a thorough exploration. Second, when F is nonconvex, the global convergence of VTMOP is entirely dependent on an effective initial search. In future iterations, every function evaluation takes place within a local trust region that is centered at a design point whose image is part of the current approximation to the Pareto front.

VTMOP provides two options for the search phase. The first option is an adaptive search technique based on the algorithm DIRECT [58]. This technique is attractive due to the global convergence guarantees of DIRECT, and often achieves the best performance with a fixed function evaluation budget. The second option is a static search technique, which uses a Latin hypercube design to generate a batch of well distributed design points within the simply bounded search region. This technique does not make usage of function values when generating the design points, which allows them to be evaluated in a large batch. This can be advantageous in settings where (1) it is convenient to decouple the evaluation of F from VTMOP or (2) enough computing resources are available to support many concurrent evaluations of F.

Adaptive Search using DIRECT

Deshpande et al. [41] propose using DIRECT to perform an adaptive global search. DIRECT is applied in $\Delta^{(k)}$ at least p times per iteration, once to each component function $f_i(x) = (e^{(i)})^T F(x)$. This strategy produces additional data in the neighborhood of each f_i's minimum design point. Because the zeroth iteration requires additional data, one additional DIRECT search is performed in the zeroth iteration, using an equal weighting of all f_i. The implementation of DIRECT in VTDIRECT95 is used for the adaptive search [52].

Remark 3.13. Similarly as in Remark 3.11, a fudge factor is applied to each zero valued component of $e^{(1)}, \ldots, e^{(p)}$ and the weights are renormalized. This ensures that each instance of the driver subroutine `VTdirect` from VTDIRECT95 will converge to a Pareto optimal point in the limit.

One important detail of DIRECT is that it samples on an implicit mesh. Since p or $p+1$ instances of the driver `VTdirect` are run in each iteration, this will surely result in redundant design point evaluations. To avoid wasting precious function evaluations, before evaluating F, VTMOP queries its internal database. If the requested design has already been evaluated (up to the design tolerance μ) then that design is not re-evaluated, and the corresponding objective value from the database is returned.

Static Search using Latin Hypercube Design

As a static experimental design, the Latin hypercube design is extremely common and is one of the recommendations of [76]. VTMOP uses the subroutine `LATINDESIGN` from the package QNSTOP [3]. Note that Latin hypercube designs are stochastic by nature, whereas the search via DIRECT from Section 3.3.2 is deterministic. Redundant design points are an unlikely occurrence when using the Latin hypercube design. However, to ensure that no function evaluations are wasted, VTMOP still checks all design points against its internal database before evaluating.

3.3.3 Solving the Surrogate Optimization Problem

The kth batch of surrogate optimization problems takes place within $\Delta^{(k)}$. Let $\mathcal{D}^{(k,*)}$ contain all $x \in \mathcal{D}^{(k)}$ plus all the points that were evaluated during the kth search phase from Section 3.3.2, and let $\mathcal{F}^{(k,*)} = \{F(x) : x \in \mathcal{D}^{(k,*)}\}$. First, p surrogates $\hat{f}_1 \approx f_1, \ldots, \hat{f}_p \approx f_p$ are fit using the data in $\mathcal{D}^{(k,*)}$ and $\mathcal{F}^{(k,*)}$. VTMOP supports the usage of a user defined surrogate, matching the provided interface. However, when no user surrogate is supplied, VTMOP uses the subroutine `LSHEP` from `SHEPPACK` [94], which implements a linear modified Shepard method. `LSHEP` was chosen as the default surrogate in VTMOP based on `LSHEP`'s

demonstrated performance as a surrogate in [40] and the recommendation of Deshpande et al. [41].

Remark 3.14. DELAUNAYSPARSE is not appropriate for usage as a surrogate model in this context. This is because the Delaunay interpolant will not make predictions outside the range of values observed in $\mathcal{F}^{(k,*)}$. This is often a desirable property in the context of MIPs. However, in the context of surrogate optimization, the surrogate must be able to extrapolate on local trends.

Remark 3.15. A user defined surrogate is desirable when F is not a true blackbox, and the user is able to exploit domain knowledge to design a better surrogate. For example, if the gradient is available for any f_i, then this information could be used to design a higher order surrogate. One could integrate such information into the linear modified Shepard approximation by replacing the local linear fit at each data point with the true tangent. For another example, if any component f_i is computationally cheap to evaluate, then surrogate modeling of f_i can be "turned off" by setting $\hat{f}_i = f_i$.

After fitting the surrogates, the kth batch of surrogate optimization problem is solved to obtain the kth batch of candidate design points

$$\mathcal{C}^{(k)} = \{ \arg\min_{z \in \Delta^{(k)}} [\hat{f}_1(z), \ldots, \hat{f}_p(z)]\tilde{W} : \tilde{W} \in \mathcal{W}^{(k)} \}. \tag{3.9}$$

To solve (3.9), VTMOP allows the usage of a user defined "local optimization" subroutine. By default, VTMOP uses the algorithm GPSMADS [5]. It should be noted that (3.9) involves analytic surrogates whose derivative information is available, but GPSMADS is an algorithm for blackbox optimization. The rational for using GPSMADS to solve (3.9) is as follows.

- MADS algorithms are locally convergent even for nonsmooth functions, which improves the robustness of VTMOP.
- When applied to analytic functions, the practical difference between first order algorithms and blackbox (zeroth order) algorithms is that first order algorithms typically converge at a faster rate. However, solving (3.9) does not require any evaluations of F, so it is an inconsequential expense in the context of blackbox optimization.

Rather than using the widely distributed NOMAD software package [63] for its implementation of GPSMADS, VTMOP offers its own lightweight Fortran implementation for the following reasons.

- Portability: NOMAD is written in C++, which creates additional dependencies for VTMOP. Specifically, using NOMAD would require users to provide a C++ compiler (in addition to a Fortran compiler) and the correct version of the C++ standard libraries.
- Package overhead: NOMAD is a large software package designed for solving blackbox optimization problems, and includes many features that are irrelevant to the surrogate

problem (3.9). Separating the used and unused features of NOMAD is not practical and including all of these features would greatly increase the size of the package VTMOP.

The implementation of GPSMADS in VTMOP is very simple. In each iteration, GPSMADS polls in every direction along an axis aligned mesh, steps in the direction of steepest descent, and if no descent direction is found, then decays the mesh size by a factor of two down to a minimum of μ. The process of fitting the surrogates and solving (3.9) is summarized in Algorithm 3.17.

Remark 3.16. VTMOP checks the current database for each new design point before evaluating. However, in the return-to-caller interface described in Section 3.4.1, the user is required to evaluate $F(z)$ for $z \in \mathcal{C}^{(k)}$. Therefore, Algorithm 3.17 eliminates any redundant or previously evaluated candidate design points from $\mathcal{C}^{(k)}$ before returning. In practice, this could mean that $|\mathcal{C}^{(k)}| < |\mathcal{W}^{(k)}|$. This is especially common when F is nonconvex, since different weight vectors could produce the same solution to (3.9).

Algorithm 3.17.
input
$\mathcal{D}^{(k,*)}$ is a database of evaluated designs, including both $\mathcal{D}^{(k)}$ and additional designs that were evaluated during the kth search (Section 3.3.2);
$\mathcal{F}^{(k,*)} = \{F(x) : x \in \mathcal{D}^{(k,*)}\}$;
$\Delta^{(k)}$ is the kth LTR;
$\mathcal{W}^{(k)}$ is the kth set of adaptive weights;
$\hat{f}_1, \ldots, \hat{f}_p$ are user supplied surrogates;
μ is the design space tolerance;
begin
initialize $\mathcal{C}^{(k)} = \emptyset$;
for $i = 1, \ldots, p$ **do**
  Fit $\hat{f}_i \approx f_i$ using $\mathcal{D}^{(k,*)}$ and $\mathcal{F}^{(k,*)}$;
enddo
for $\tilde{W} \in \mathcal{W}^{(k)}$ **do**
  $\tilde{z} \leftarrow \arg\min_{z \in \Delta^{(k)}} [\hat{f}_1(z), \ldots, \hat{f}_p(z)]\tilde{W}$;
  if $\|\tilde{z} - x\|_2 > \mu$ for all $x \in \mathcal{D}^{(k,*)}$ **and** $\|\tilde{z} - z\|_2 > \mu$ for all $z \in \mathcal{C}^{(k)}$ **then**
    $\mathcal{C}^{(k)} \leftarrow \mathcal{C}^{(k)} \cup \{\tilde{z}\}$;
  endif
enddo
return $\mathcal{C}^{(k)}$

The time complexity of Algorithm 3.17 is dependent on the cost for fitting the surrogates, the cost for evaluating the surrogates, and the number of surrogate evaluations. By default, VTMOP performs 2500 iterations of `GPSMADS` (each requiring $2d$ evaluations of the surrogate) per $\tilde{W} \in \mathcal{W}^{(k)}$. In the package VTMOP, the subroutine `VTMOP_OPT` implements Algorithm 3.17.

3.3.4 Evaluating Candidate Designs and Iterating

The final step in the kth iteration is to evaluate all $z \in \mathcal{C}^{(k)}$ with F. After all evaluations have finished, the $(k+1)$st databases $\mathcal{D}^{(k+1)}$ and $\mathcal{F}^{(k+1)}$ have been completed. In the next iteration, `VTMOP_LTR` will compute $\mathcal{P}^{(k+1)}$ and $\mathcal{E}^{(k+1)}$, so there is no need to do so in this stage unless a termination condition has been met. The iteration counter k is incremented at this point.

There are three potential termination conditions for VTMOP. The first condition is that the budget for evaluations of F has been spent. If this limit is reached mid-iteration, the rest of the iteration aborts. The second condition is that the iteration limit has been exceeded. This condition is checked before the beginning of each iteration. The third condition is that every point in the current approximation to the efficient set has been used as the center of a previous LTR, whose trust region radius fraction has been decayed below the trust region tolerance. This condition is checked by `VTMOP_LTR`, as described in Algorithm 3.6.

When any termination condition occurs at iteration k, the current database of evaluated design points and corresponding function values is used to compute the final nondominated and approximately efficient point sets, which are returned with the termination indicator.

Remark 3.18. Unlike in Section 3.3.1, upon termination, VTMOP returns every observed objective vector that is nondominated and its corresponding efficient design. In particular, this may include duplicate nondominated points. In the literature, these are referred to as the (approximately) weakly nondominated and efficient sets.

3.4 Solvers for Multiobjective Optimization Problems

VTMOP offers two interfaces for solving blackbox MOPs based on the framework laid out in Section 3.3. The first is a return-to-caller interface and the second is the driver subroutine `VTMOP_SOLVE`. In many situations, F can easily be wrapped in a Fortran subroutine, and the driver subroutine is preferred due to its ease of use. However, the return-to-caller interface offers a flexible alternative for situations where the computing environment makes wrapping F in a Fortran subroutine difficult or inefficient. Both interfaces are implemented in ISO Fortran 2008 [57] and support some form of OpenMP parallelism. The complete code is provided in the appendix.

3.4.1 The Return-To-Caller Interface

In many real-world blackbox optimization problems, special purpose computing environments [84] and parallel resource coordination libraries [54] require the evaluation of F to be decoupled from the optimization algorithm. In these situations, it is convenient to offer a

return-to-caller interface, where F is only evaluated from outside of any Fortran subroutine's call stack.

Here it is necessary to preserve the entire state of the algorithm VTMOP during Steps 2 and 4 (Section 3.3), where F is evaluated in batches (presumably in parallel). The state is recorded in a Fortran derived data type `VTMOP_TYPE`, created by the subroutine `VTMOP_INIT`, and stored between invocations of components of `VTMOP`. The cycle is: $k \leftarrow 0$; `call VTMOP_INIT` (which creates a `VTMOP_TYPE` instance holding the problem parameters and VTMOP state), store the state; `STEP 1`: `call VTMOP_LTR`; `STEP 2`: evaluate batch of $F(x)$ values where x is in the trust region returned by `VTMOP_LTR` (trust region search); `STEP 3`: `call VTMOP_OPT`; `STEP 4`: evaluate batch of $F(z)$ values at candidate design points $z \in \mathcal{C}^{(k)}$; `STEP 5`: $k \leftarrow k+1$; check termination conditions, and either terminate with `call VTMOP_FINALIZE` (compute $\mathcal{P}$, $\mathcal{E}$, deallocate all `VTMOP_TYPE` storage, and exit) or go to `STEP 1`. Precede each `STEP` by "read the state" and follow each `STEP` by "update and store the state," if necessary.

3.4.2 The Driver Subroutine

The driver subroutine `VTMOP_SOLVE` implements the same process as in Section 3.4.1, but without reading and writing the state, and using a Fortran implementation of F. The search phase is either performed via DIRECT as in Section 3.3.2 or via Latin hypercube design as in Section 3.3.2, with the choice being specified as an optional input. For the search via DIRECT, the "search budget" inputs are used to specify the iteration limit for `VTDIRECT95`. For the search via Latin hypercube design, the search budgets are used to specify the number of points in the design.

In addition, `VTMOP_SOLVE` offers a checkpointing system and a parallel option described in Section 3.4.3.

Remark 3.19. The checkpointing system in VTMOP separately saves iteration data (such as the sequence of LTRs and parameter settings) and function evaluation data. This is done so that the iteration data can still be recovered when the return-to-caller interface is used, although the function evaluation data would need to be tracked separately. As a side effect, the checkpoint files do not track what design point evaluations have been requested but not fulfilled. So, if an instance of `VTMOP_SOLVE` terminates midway through Step 2 or 4 from Section 3.3, reloading the last checkpoint may result in a new sequence of evaluations.

3.4.3 Parallel Implementation

There are two sources of parallelism in VTMOP. First, the iteration tasks in the worker subroutines `VTMOP_LTR` and `VTMOP_OPT` can be parallelized. This includes computing $DG(\Pi^{(k)})$ in parallel using the parallel driver `DELAUNAYSPARSEP` from DELAUNAYSPARSE, fitting the

p surrogates in parallel, and solving the $|\mathcal{W}^{(k)}|$ surrogate problems (3.9) in parallel. Iteration level parallelism is provided via OpenMP 4.5 [79], and is available through either the return-to-caller interface or the driver subroutine.

The second source of parallelism is concurrent evaluations of F. In the context of blackbox optimization, this is the more impactful source of parallelism. Since the return-to-caller interface does not evaluate F, this form of parallelism is only available through the driver subroutine (which offers a choice between iteration task parallelism, concurrent function evaluations, or both). Recall that VTMOP only requires function evaluations while searching $\Delta^{(k)}$ as described in Section 3.3.2 and when evaluating candidate designs from $\mathcal{C}^{(k)}$ as described in Section 3.3.4. In both cases, concurrent evaluations are achieved by spawning a new OpenMP task for each evaluation of F.

Remark 3.20. Within a single evaluation of F, there may be ample opportunities for parallelism, which is problem dependent and left to the user. In particular, in many real-world applications, each evaluation of F could be the result of a multi-node distributed simulation. VTMOP does not distribute calls to F, but provides the ability for concurrent evaluations of F via OpenMP task-based parallelism. This places the burden of distributing calls to F on the user, but allows for greater flexibility.

When evaluating $\mathcal{C}^{(k)}$ as in Section 3.3.4 and when using the Latin hypercube design during the search phase, all candidates/design points can be evaluated in parallel if there are enough available computing resources. For the search via DIRECT, the situation is more complicated since each batch of potentially optimal boxes in a DIRECT iteration can be divided in parallel, and each instance of `VTdirect` can be run asynchronously. `VTDIRECT95` offers a parallel driver `pVTdirect` for distributing the division of boxes, but the distributed memory paradigm is not appropriate for this use case. Instead, a slightly modified implementation of `VTdirect` called `bVTdirect` is provided, which uses OpenMP tasks to perform concurrent evaluations of boxes. During the kth search, VTMOP makes either $p+1$ (if $k=0$) or p (if $k>0$) asynchronous calls to `bVTdirect`, resulting in nested parallelism. The code for `bVTdirect` is provided in the appendix.

Remark 3.21. The following considerations only apply when using the DIRECT based search in the driver `VTMOP_SOLVE`. As described in Section 3.3.2, running asynchronous instances of `bVTdirect` will result in requests for redundant design point evaluations, which VTMOP handles by using OpenMP task dependency. This means that when using DIRECT during the search phase, many OpenMP tasks will be waiting on the completion of other tasks. As a result, the total number of OpenMP threads should be greater than the desired number of concurrent function evaluations so that tasks that are waiting will not block other designs from being evaluated. A general recommendation is to set the number of OpenMP level one threads to the desired number of concurrent function evaluations $t^{(1)}$. This number will also be used for determining the number of concurrent evaluations when evaluating the candidate designs and for determining the number of iteration tasks that can be done in parallel. For example, if each evaluation of F requires a single node, then $t^{(1)}$ would be

the number of available nodes. If $t^{(1)} \leqq p$, then the number of level two threads $t^{(2)}$ should typically be two or three. If $t^{(1)} > p$, a larger value of $t^{(2)}$ may be needed.

Remark 3.22. Each serial run of `VTMOP_SOLVE` is deterministic when using the search via DIRECT, `LSHEP` surrogate, and `GPSMADS` optimization options. However, for grid aligned data, `LSHEP` can be sensitive to the ordering of function evaluations in the databases $\mathcal{D}^{(k,*)}$ and $\mathcal{F}^{(k,*)}$. Since DIRECT samples on an implicit mesh, using concurrent function evaluations in a run of `VTMOP_SOLVE` with the above settings could produce nondeterministic (but still reasonable) results.

3.4.4 Use Cases for VTMOP

Algorithms 3.6 and 3.17 make very mild assumptions about the cost function F, which are appropriate for nearly all real-world blackbox problems. Specifically, in order to use adaptive weighting schemes and trust region methods, one must assume that the Pareto front consists of finitely many continuous regions (with possible discontinuities between them) and that F satisfies a Lipschitz condition. Additional conditions may be required depending on the global search technique, surrogate, and local optimizer that are used, but for the search via DIRECT, `LSHEP`, and `GPSMADS`, the above assumptions are sufficient.

3.5 Integration with libEnsemble

`libEnsemble` is a library for coordinating the concurrent evaluation of dynamic ensembles of calculations [54]. The library is developed at Argonne National Laboratory as a part of the DOE Exascale Computing Project [72], and it is designed to use massively parallel resources to accelerate the solution of design, decision, and inference problems and to expand the class of problems that can benefit from increased levels of concurrency.

The motivation for integrating VTMOP with `libEnsemble` is to solve next-generational problems in the exascale environment. Recall from Section 1.2 that although this thesis targets problems where minutes are required per evaluation of F, extremely expensive blackbox MOPs could require hours or days per evaluation of F, and each evaluation may occupy hundreds or even thousands of CPUs [12, 16]. With modern hardware, this means that the 1000–2000 evaluation budget targeted by VTMOP is still prohibitively expensive for extremely expensive MOPs. However, on next generation hardware such as the exascale machine Aurora [95], it may be possible to perform hundreds of these simulations in parallel making VTMOP feasible for a broader class of problems.

3.5.1 The libEnsemble Library

`libEnsemble` employs a manager-worker scheme that is controlled by user-defined simulation, generation, and allocation functions (each of which could be using parallel computing resources). The generation function produces parameter values to be evaluated by the simulation function; the simulation function uses the specified parameters to run a (potentially expensive) numerical calculation; and the allocation function decides when a simulation or generation function should be called and with what resources. If one is using `libEnsemble` to solve MOPs, the generation function is a MOA, the simulation function evaluates the blackbox function F, and the allocation function calls the generation function when the simulation function has evaluated all of the previously requested set of parameter values.

`libEnsemble` employs a manager process to allocate work to multiple worker processes. A `libEnsemble` worker is the smallest indivisible unit that can perform calculations. For some cases, a worker can be given a single core to generate a sample of points; in other cases, a worker can be given many nodes to perform a complex numerical simulation. While one can allocate different resources to different workers throughout a given `libEnsemble` run, all instances in this section give each worker equal computational resources. The `libEnsemble` manager-worker communications can utilize various communication media, including MPI, multiprocessing, and TCP. Interfacing with user-provided executables is also supported. Each worker can control and monitor any level of work, from small subnode jobs to huge many-node simulations.

In this section VTMOP is integrated into `libEnsemble` as a generation function. For integration with VTMOP, a persistent generation function is used to produce points to be evaluated: the generation function is initiated at the beginning of the `libEnsemble` run when it provides a batch of points to be evaluated. Only after points in a batch have been evaluated is the generation function informed of the function values so that it can produce the next batch of points. Hence, workers (and their computational resources) will be idle when a worker completes a given function evaluation and all remaining points in the batch are currently being evaluated.

For more details on `libEnsemble`, readers are referred to the user's manual [54].

3.5.2 Implementing VTMOP as a libEnsemble Generator

To integrate VTMOP with `libEnsemble`, Algorithms 3.6 and 3.17 are implemented as `libEnsemble` generation functions. Since `libEnsemble` handles evaluations of F in a separate environment, this requires the return-to-caller interface from Section 3.4.1. Naïvely, this could be achieved by using a Latin hypercube design for each search and without altering any fundamental details of VTMOP. This naïve approach would result in the following methodology, which uses an input file for both VTMOP and `libEnsemble` to share data.

In the initial call to the generator, `VTMOP_INIT` is called to create a new `VTMOP_TYPE`. Then `LATINDESIGN` is called with a large budget to generate samples in $[L, U]$. The state of the `VTMOP_TYPE` is saved for future calls using VTMOP's built in checkpointing system and the requested design points in the Latin hypercube design are written to `libEnsemble`'s input file. In each subsequent call to the generator the following steps are performed.

- The VTMOP problem state is reloaded from the last checkpoint and the current database $\mathcal{D}^{(k)}$ and $\mathcal{F}^{(k)}$ or $\mathcal{D}^{(k,*)}$ and $\mathcal{F}^{(k,*)}$ is read in from VTMOP's input file.

- Either `VTMOP_LTR` or `VTMOP_OPT` is called (depending on the stage of the iteration).

- If `VTMOP_OPT` has been called, then the batch of design points $\mathcal{C}^{(k)}$ is written to an input file for `libEnsemble` and the iteration counter is incremented. If `VTMOP_LTR` has been called, then a Latin hypercube design is generated in $\Delta^{(k)}$ and these points are written to the input file for `libEnsemble`.

- The checkpoint file is updated for tracking iteration data before exiting.

After each call to the generator, `libEnsemble` coordinates the concurrent simulation evaluations. Encapsulating each call to VTMOP is a Python script that implements VTMOP as a `libEnsemble` generator function. Prior to each call to the VTMOP generator described above, this script writes the results from previous `libEnsemble` simulation evaluations to VTMOP's input file. After each call to the VTMOP generator, this script reads the batch of requested evaluations from the input file (containing either $\mathcal{C}^{(k)}$ or points in a Latin hypercube design) and adds them into `libEnsemble`'s internal list of simulations to be evaluated.

The above approach will result in a reasonable integration between VTMOP and `libEnsemble` without significantly modifying any of the details from Section 3.3. However, one of the goals of `libEnsemble` is to make efficient usage of extreme scale resources, and as described above, there is no mechanism to control the size of each batch of simulations, thus leading to poor load balancing when implemented over statically allocated HPC resources. For calls to the generator that use `VTMOP_LTR` and a Latin hypercube design, there is an easy fix. Suppose that `libEnsemble` is run on an HPC system with sufficient resources for $t^{(b)}$ concurrent evaluations of F. When the size of each Latin hypercube design is large relative to $t^{(b)}$, then the machine utilization should always be relatively high. However, when the size of the Latin hypercube design is small relative to $t^{(b)}$, then this size should be a multiple of $t^{(b)}$.

The second modification allows users to control the batch size for generating candidate design points at the end of Algorithm 3.17 in order to match $t^{(b)}$. Note that as described in Algorithm 3.17, `VTMOP_OPT` produces variable-sized batches of candidate designs $\mathcal{C}^{(k)}$, which is not conducive to load balancing. Every candidate design for one of the adaptive weightings must be evaluated, in order to ensure that new solutions are found surrounding the current isolated point **before** resuming iteration. In particular, if $|\mathcal{C}^{(k)}| < t^{(b)}$ then this will result in poor system utilization.

Suppose that $t^{(a)}$ additional candidates are needed to achieve a multiple of $t^{(b)}$. In order to find $t^{(a)}$ additional candidates to pad out the current batch, a large number (greater than $t^{(a)}$) of **additional** convex weight vectors are randomly generated and added to $\mathcal{W}^{(k)}$ before beginning Algorithm 3.17. After solving all the surrogate optimization problems, these random weight vectors produce additional candidate designs within the current LTR. When the component functions $f_1, \ldots, f_p$ are smooth, these candidate designs will yield new solution points that are in the neighborhood of the current isolated point in the objective space. It is not always possible to find $t^{(a)}$ additional candidates, however, especially when $t^{(b)}$ is large. The reason is that different weight vectors can yield the same candidate solutions (particularly when F is nonconvex). Therefore, a fixed number of additional random weights is always generated and used to produce a pool of additional candidates. If this pool of additional candidates is larger than $t^{(a)}$, then the first $t^{(a)}$ candidates are added to pad out the batch returned by the `libEnsemble` generator function. If fewer than $t^{(a)}$ additional candidates are generated, then all additional candidates are added, padding out the batch as much as possible.

Note that generating this additional pool of candidates to pad out each batch significantly increases the iteration time for Algorithm 3.17 since many additional surrogate optimization problems must be solved. However, in the context of extremely expensive blackbox MOPs, these additional costs will be negligible.

The code for initializing VTMOP within `libEnsemble`, the code for generating a batch of evaluations for `libEnsemble`, and the Python script that encapsulates each call to VTMOP are provided in the appendix.

3.6 Approximation Performance

Recall that the original algorithm of Deshpande et al. [41] used the search based on DIRECT. This algorithm was previously compared against `BiMADS` from NOMAD [63], using a budget of just 200 function evaluations [41]. It was also compared against a private implementation of MultiMADS with a budget of 30,000 evaluations of F (as recommended by [39]). The results in [41] suggest that using VTMOP with a search via DIRECT is competitive with `BiMADS` and `MultiMADS` at those budgets. In particular, the algorithm implemented in VTMOP produces better distributions of points on the Pareto front, and therefore a better approximation to its true shape.

In this section, two test problems are solved using VTMOP with budgets that are in the target range of 1000–2000 evaluations.

3.6.1 Description of Test Problems

Let $d > p$, and let $e^{(i)}$ denote the ith standard basis vector in $\mathbb{R}^p$. Then the first test problem is defined by

$$F^{(c)}(x) = \left(\|x - 0.6e^{(1)}\|_2^2, \ldots, \|x - 0.6e^{(p)}\|_2^2\right), \quad x \in [0,1]^d.$$

The Pareto front for $F^{(c)}$ is a portion of a rotated parabola in $\mathbb{R}^p$. Because this problem is convex, it is an easier problem for MOAs.

The second test problem is the nonconvex problem `DTLZ2`, as described by [39]. Its Pareto front is the unit sphere in the positive orthant. This Pareto front is concave, which generally presents a problem for adaptive weighting schemes. The equation for each component function of `DTLZ2` is

$$\begin{aligned}
f_1^{(D2)}(x) &= (1 + g^{(D2)}(x)) \cos(\pi x_1/2) \cos(\pi x_2/2) \ldots \cos(\pi x_{p-2}/2) \cos(\pi x_{p-1}/2) \\
f_2^{(D2)}(x) &= (1 + g^{(D2)}(x)) \cos(\pi x_1/2) \cos(\pi x_2/2) \ldots \cos(\pi x_{p-2}/2) \sin(\pi x_{p-1}/2) \\
f_3^{(D2)}(x) &= (1 + g^{(D2)}(x)) \cos(\pi x_1/2) \cos(\pi x_2/2) \ldots \sin(\pi x_{p-2}/2) \\
&\vdots \qquad\qquad \vdots \\
f_{p-1}^{(D2)}(x) &= (1 + g^{(D2)}(x)) \cos(\pi x_1/2) \sin(\pi x_2/2) \\
f_p^{(D2)}(x) &= (1 + g^{(D2)}(x)) \sin(\pi x_1/2)
\end{aligned}$$

where $g^{(D2)}(x) = \sum_{i=p}^{d} (x_i - 0.6)^2$.

For all problems considered, the design dimension is $d = 5$. The objective dimensions considered are $p = 2$, $p = 3$, and $p = 4$, and the function evaluation budgets of sizes 1,000, 1,500, and 2,000 are considered.

Evaluation of how well points evaluated by a MOA approximate the true Pareto front is an open problem [7]. Three criteria to measure the quality of an approximate Pareto surface are considered:

- the number of nondominated solution points identified,
- the distance between points on the approximate and true Pareto front,
- the degree to which those points are spread evenly across the entire Pareto front.

Specifically, the cardinality, the root mean squared error (RMSE), and the *star discrepancy* [41] of the solution set are used to measure these three criteria, respectively. Note that the RMSE can be easily computed because the test functions have analytic Pareto fronts. In order to compute the star discrepancy, unions of path-connected Delaunay simplices are used as a family of Lebesgue measurable sets. The Delaunay discrepancy, approximating the star discrepancy, is computed as the maximum absolute value of the difference between the fraction of points in a region of the Pareto front and the fraction of that region's volume relative to the total volume of all the Delaunay simplices, taken over growing unions

of path-connected Delaunay simplices. The Delaunay discrepancy approaches zero when evaluated for uniformly spaced points. Since computation of the Delaunay discrepancy requires the complete Delaunay triangulation, this is not an appropriate use case for DELAUNAYSPARSE. Therefore, the Delaunay discrepancy is evaluated based on the triangulation returned by `SciPy 1.0` [98], which uses Quickhull.

3.6.2 Algorithm Variations and Parameter Settings

Both test problems are optimized with each of the three variations of VTMOP:

- the "DIR" variant uses VTMOP with a search based on DIRECT;
- the "LH" variant uses VTMOP with a Latin hypercube search; and
- the "LIBE" variant uses the variation of VTMOP that was integrated with `libEnsemble`.

For all runs, `LSHEP` is used as the surrogate, and GPSMADS is run for 2500 iterations per surrogate problem. ϵ (Remark 3.3, μ (Remark 3.5), and the "fudge factor" (Remarks 3.11 and 3.13) are set to the fourth root of the unit round-off. The problem's working precision is set to the square root of the unit round-off. The initial trust region radius fraction is set to 0.2 and decayed by a factor of 0.5 down to a minimum tolerance of 0.02.

For the DIR variant, `bVTdirect` is given a budget of ten iterations for the zeroth search (over the entire feasible design space) and five iterations in each subsequent search (over the kth LTR).

For the LH variant, the initial search consists of a 500 point Latin hypercube design and each subsequent search consists of a 72 point design.

For the LIBE variant, 36 workers are allocated indicating that the preferred batch size should either be very large or a multiple of 36. Similarly as above, the zeroth Latin hypercube search consists of a 500 point design, and all subsequent searches consist of 72 point designs. The preferred batch size for each batch of candidates is $t^{(b)} = 36$ and 54 random weight vectors are generated to pad out batches.

3.6.3 Results

The performance statistics in this section are gathered by running all three variations of VTMOP (as described in Section 3.6.2) on the test problems described in Section 3.6.1. For the LH and LIBE variants (stochastic) results are averaged over five repeated trials.

Tables 3.1, 3.2, and 3.3 show the (average) number of solution points, RMSE, and Delaunay discrepancy, respectively, for VTMOP using the DIR, LH, and LIBE variants. Statistics are shown for budgets of 1,000, 1,500, and 2,000 in a comma separated list. For the LH and

Table 3.1: Number of solutions found by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
$F^{(c)}$ / DIR	44, 59, 66	110, 166, 187	197, 313, 399
$F^{(c)}$ / LH	31, 66, 90	98, 160, 229	171, 295, 416
$F^{(c)}$ / LIBE	70, 122, 136	139, 247, 304	240, 404, 534
`DTLZ2` / DIR	44, 60, 80	83, 148, 248	138, 247, 393
`DTLZ2` / LH	51, 80, 112	108, 171, 249	233, 341, 485
`DTLZ2` / LIBE	29, 38, 64	78, 166, 205	248, 311, 467

Table 3.2: RMSE reported by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
$F^{(c)}$ / DIR	.0509, .0501, .0506	.121, .121, .119	.191, .194, .192
$F^{(c)}$ / LH	.0595, .0614, .0585	.123, .120, .192	.197, .194, .193
$F^{(c)}$ / LIBE	.0192, .00631, .00508	.0110, .00799, .00697	.0125. .00962, .00833
`DTLZ2` / DIR	.0101, .00710, .00616	.0170, .0142, .0115	.0156, .0151, .0134
`DTLZ2` / LH	.0246, .0179, .0137	.111, .0616, .0400	.107, .0834, .0678
`DTLZ2` / LIBE	.0548, .0250, .0192	.0928, .0626, .0438	.0780, .0856, .0780

LIBE variants, the average number of solution points is rounded to the nearest integer. For all three variations, the (average) RMSE and discrepancies are rounded to three significant figures.

For both problems, the DIR and LH variants offer similar performance for the convex problem $F^{(c)}$. However, for the nonconvex problem `DTLZ2`, the DIR variant significantly outperforms the LH variant in terms of RMSE. This is not surprising since the search via DIRECT utilizes function evaluation information to improve performance during each search phase. However, it is somewhat surprising that both variations achieve significantly lower RMSE for the nonconvex problem than in the convex case. This is most likely because the entire efficient set for $F^{(c)}$ lies on the boundary of the box $[0, 1]^d$, which implies that every efficient point could only be on the boundary of each LTR. The LIBE variant performs similarly to the LH variant for the nonconvex problem, most likely due to the fact that it is not able to pad out the batches of candidates in the nonconvex case and therefore is equivalent to the LH variant. However, the LIBE variant produces significantly more approximately nondominated points with lower RMSE for $F^{(c)}$ than any of the other variations. This is a clear indication that the surrogate optimization problems are being solved to high accuracy since the only difference

Table 3.3: Discrepancy reported by VTMOP using DIRECT (DIR variant), VTMOP using Latin hypercube search (LH variant), and VTMOP with `libEnsemble` (LIBE variant) for the test problems $F^{(c)}$ and `DTLZ2` with budgets of 1000, 1500, and 2000 evaluations given in a comma separated list.

Problem/Method	$p = 2$	$p = 3$	$p = 4$
$F^{(c)}$ / DIR	.104, .188, .172	.154, .0822, .0482	.169, .194, .192
$F^{(c)}$ / LH	.216, .210, .250	.125, .199, .202	.194, .172, .158
$F^{(c)}$ / LIBE	.204, .210, .186	.178, .111, .0812	.229, .165, .252
`DTLZ2` / DIR	.211, .100, .173	.233, .146, .0784	.112, .0930, .0666
`DTLZ2` / LH	.230, .243, .282	.111, .105, .0600	.221, .106, .0756
`DTLZ2` / LIBE	.141, .140, .122	.285, .0800, .0971	.130, .193, .129

between the LIBE and LH variants is that LIBE solves many additional surrogate problems.

Overall, there is no clear best algorithm for both problems when considering all criteria. As seen in the next section, the improved parallel efficiency of the LH and LIBE variants would generally allow for more evaluations to be completed with the same resources in an extreme scale HPC setting.

3.7 Parallel Performance

In this section, the parallel performance is analyzed for each of the three VTMOP implementations from Section 3.6.2.

3.7.1 Description of Hardware and Problem Expense

Experiments have been performed on the HPC system `Bebop` at the Laboratory Computing Resource Center at Argonne National Laboratory. Each Broadwell node has an Intel Xeon E5-2695v4 CPU, with 36 cores and 128 GB of DDR4 RAM. A single node has been dedicated for each problem instance run and both timing and convergence data was collected. In the context of a true simulation-based MOP, this methodology models the availability of computational resources to perform up to 36 concurrent evaluations of F. In order to evaluate runtime performance, both wall time and CPU time have been measured.

In order to emulate the expense of a blackbox simulation problem, an artificial runtime is simulated using the CPU clock. For both $F^{(c)}$ and `DTLZ2`, two problem variations are created for timing purposes. In the first variation, the runtime is set to one second; in the second variation, the runtime is a random value drawn uniformly from the interval $[0.5s, 1.5s]$. This tests the four VTMOP variations for their ability to load balance function evaluation times that vary.

Additionally, recall from Remark 3.21 that when using `bVTdirect` to perform the search via DIRECT in parallel, certain OpenMP threads will end up waiting on other tasks to complete. In a true HPC environment, users should overload the master node with OpenMP threads, but only allow a distributed simulation to launch when sufficient resources for performing that simulation become available. In this section, two subvariations of the DIR variant from the previous section are run for timing purposes.

The variation "DIR1" demonstrates the true performance of performing a search via DIRECT using `bVTdirect` on a single node shared-memory system, as the total number of OpenMP threads is limited to 36 (the number of cores on a single `Bebop` compute node). This implies that less than 36 threads can be working during each run of `bVTdirect`, since numerous threads will spend time waiting on other tasks to complete. In the variation "DIR2," the maximum number of OpenMP threads is unlimited, but the number of concurrent evaluations of F is capped at 36 by implementing an internal counter in each of the test problems. This simulates a distributed-memory setting where sufficient resources for 36 concurrent evaluations are available.

Unfortunately, for the variation DIR2, one cannot accurately estimate the total CPU time. This is because there is no way to distinguish threads that are waiting for tasks to complete (and therefore should not count toward the total CPU time), threads that are waiting for resources to become available (and therefore should also not count toward the total CPU time), and threads that are "performing computations," i.e., whose timer is running. Furthermore, because function evaluations are dynamically assigned to OpenMP tasks, there is no way to enforce CPU affinity, so multiple threads may be able to sit on a single core and run their timers (for enforcing the function evaluation time) concurrently. However, if it were possible to gather this information, the total CPU time for the DIR2 variant should be very similar to the CPU time for the DIR1 variant.

3.7.2 Runtime Performance Results

The performance statistics in this section are gathered by averaging over five repeated trials.

Table 3.4 shows the average wall times and CPU times in seconds for solving both $F^{(c)}$ and `DTLZ2` with and without performance variability with a budget of 2,000 function evaluations. Summary statistics are reported for $p = 2$, $p = 3$, and $p = 4$ objectives, using the DIR1, DIR2, LH, and LIBE variations. CPU times and wall times are given in seconds, rounded to two decimal places. Note that the total amount of CPU time required to perform all 2,000 evaluations is (approximately) 2,000 seconds for all problems, without considering iteration tasks. Also, recall that the CPU times could not be accurately computed for the variation DIR2; these entries are labeled "NA."

For all the problems, the variations LIBE and LH appear to perform best in terms of CPU time over wall time. Since the variation DIR performed better in Section 3.6.3, this illustrates

Table 3.4: Runtime performance summary (CPU time / wall time) for four variations of VTMOP. Times are given in seconds.

p	method	$F^{(c)}$, no var	$F^{(c)}$, w/ var	DTLZ2, no var	DTLZ2, w/ var
2	DIR1	2008.15 / 1037.60	2007.22 / 1039.40	2007.74 / 1093.20	2004.87 / 1082.22
	DIR2	NA / 170.58	NA / 239.09	NA / 175.28	NA / 240.15
	LH	2015.46 / 88.64	2016.78 / 107.32	2027.76 / 89.38	2011.80 / 109.09
	LIBE	2051.96 / 112.32	2070.76 / 142.66	2060.57 / 111.98	2064.93 / 143.50
3	DIR1	2012.50 / 717.86	2012.79 / 719.34	2021.66 / 797.24	2018.79 / 797.70
	DIR2	NA / 137.08	NA / 207.48	NA / 165.54	NA / 237.07
	LH	2023.13 / 94.05	2033.94 / 116.28	2039.76 / 95.31	2023.30 / 116.91
	LIBE	2077.04 / 133.10	2066.47 / 144.46	2054.04 / 99.34	2057.03 / 126.77
4	DIR1	2026.62 / 582.50	2029.23 / 586.49	2177.44 / 807.59	2149.43 / 782.86
	DIR2	NA / 134.42	NA / 208.84	NA / 280.75	NA / 348.38
	LH	2041.90 / 107.90	2044.53 / 127.45	2176.07 / 199.70	2200.73 / 280.49
	LIBE	2134.58 / 190.23	2124.67 / 186.57	2182.78 / 227.51	2185.34 / 257.72

the inherent tradeoff between the convergence rate of VTMOP and its ability to utilize parallel resources.

However, also notice that the CPU time required by the LIBE variant is slightly greater than that required by the variations DIR1, DIR2, and LH. This is because of the additional computations required for padding out the batches of candidate points as described in Section 3.5.2. Also notice that based on the apparent CPU utilization, none of the variations appear to make full utilization of all 36 cores.

In the case of DIR1 and DIR2, this lack of full utilization is caused by a lack of opportunity for increased levels of concurrency. Notice that the CPU utilization for DIR1 and DIR2 significantly improves as p increases. This is because the opportunities for parallelism naturally increase with the number of objectives, as indicated in Remark 3.21. Figure 3.2 shows how the CPU is utilized over time in the three-objective case for all four problem variations in a single run with the DIR1 variant of VTMOP. Because of the issues discussed above, it was not possible to produce this plot for the DIR2 variation. As seen in Figure 3.2, the DIR1 variant is simply not able to make effective utilization of resources for 36 concurrent evaluations for the given problems. With larger values of p, it is likely that this utilization would improve. However, the limited search via DIRECT, as used here, is clearly better suited for situations where resources for only a few concurrent evaluations are available. This is in contrast with the standard usage case for VTDIRECT95 – a many iteration single objective run, which is capable of achieving reasonable overall resource utilization with up to 99 concurrent evaluations [51]. Note that due to the resource requirement of many expensive simulations, this is not an uncommon scenario.

For the LH variant, the lack of full CPU utilization is due to the evaluation of small batches of candidate designs. Figure 3.3 shows how the CPU is utilized over time in the three-

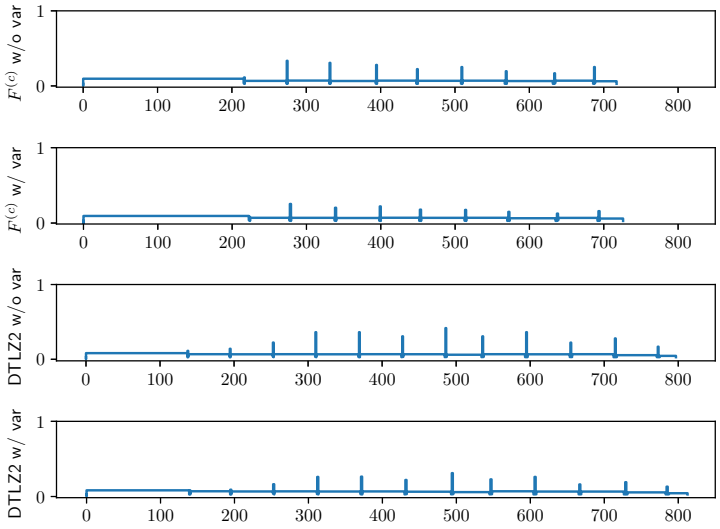

Figure 3.2: CPU utilization over time for a single run of VTMOP using the search via DIRECT with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation.

objective case for all four problem variations in a single run of the LH variant of VTMOP. The performance peaks correspond to instances when many simulations are being performed in parallel, and the performance valleys correspond to iteration tasks, which were done serially in this experiment.

For the LIBE variant, the lack of full CPU utilization can be mostly attributed to the increased iteration costs. Figure 3.4 shows how the CPU is utilized over time in the three-objective case for all four problem variations in a single run of the LIBE variant of VTMOP. Notice that unlike in Figure 3.3, almost all of the peaks in Figure 3.4 give very high CPU utilization, especially for the convex problem $F^{(c)}$. There are still several instance when solving `DTLZ2` where less than 50% utilization is achieved over certain batches of evaluations. This is due to the nonconvexity of `DTLZ2`, which makes finding $t^{(a)}$ additional candidates to pad out each batch impossible.

When considering the results from this section, note that in real-world functions involving numerical simulations, the evaluation times are often significantly greater than one second and would overwhelm these heightened iteration costs. Also, it is interesting to notice the significant drop in peak CPU utilization (for all algorithms) once performance variability is introduced. This observation will be referenced again in Chapter 4.

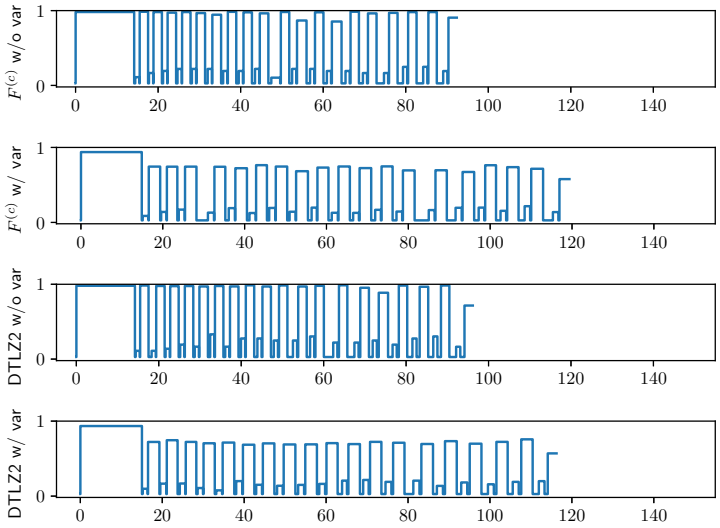

Figure 3.3: CPU utilization over time for a single run of VTMOP using the Latin hypercube search with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation. Note that the performance valleys correspond to iteration tasks, which were done serially in this experiment.

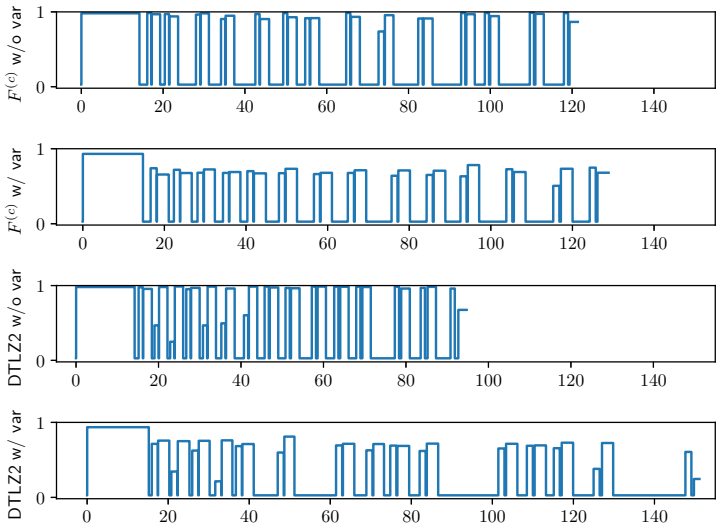

Figure 3.4: CPU utilization over time for a single run of VTMOP using `libEnsemble` with $p = 3$ objectives on two test problems, with and without runtime variance. The y-axis shows proportion of CPU resources being utilized and the x-axis shows the time in seconds since the beginning of the computation. Note that the performance valleys correspond to iteration tasks, which were done serially in this experiment.

Chapter 4

Applications

"*The final test of a theory is its capacity to solve the problems which originated it.*"
— George B. Dantzig (Preface to [33])

4.1 The Variability Problem

In this chapter, DELAUNAYSPARSE and VTMOP are applied to solve real-world problems in computer system performance modeling. The applications in this section are motivated by the VarSys project, as described in Section 1.2. The results in this chapter are based on work that has been published in [23] and is currently under review for publication in *the 2020 Winter Simulation Conference* [25].

The rest of this chapter is organized as follows. Section 4.2 provides a brief literature review and further details motivating the study of performance variability. Section 4.3 presents an application of DELAUNAYSPARSE for making performance variability predictions. Section 4.4 presents an application of VTMOP for balancing performance variability against mean performance.

4.2 Literature Review

Performance variability—the fluctuation or "jitter" in the observed performance of a computing system—is a well-known issue in both cloud [96] and high-performance computing (HPC) systems [62]. In practice, these fluctuations need not be normally distributed; they can follow a variety of distributions, including multimodal distributions [101]. Accounting for such distributions often results in analysis that requires the use of nonparametric statistic [67]. Performance variability can have many sources, including interference from system- and OS-level processes [35, 83], filesystem performance [20], and application-level parameters [49].

When left unchecked, performance variability can result in poor load balancing [37] and a significant degradation in performance [10]. In particular, recall the CPU utilization plots in Figures 3.2, 3.3, and 3.4 from Section 3.7.2. When performance variability was intro-

duced into these problems, the CPU utilization of all three variations of VTMOP suffered a significant drop.

Several works have focused on modeling and analyzing I/O throughput variability for memory-bound tasks [18, 69]. Notably, Cameron et al. [18] propose the MOANA methodology for modeling and analyzing performance variability. Recent works have also presented datasets and thorough analyses of performance variability for compute-bound settings [81]. This work in computational throughput variability is of great interest for its relevance to the problem of load balancing ensembles of expensive numerical simulations with `libEnsemble`, as in Section 3.7.2.

4.3 Using DELAUNAYSPARSE to Predict I/O Throughput Variance

In this section, DELAUNAYSPARSE is used in the modeling and analysis (MOANA) framework proposed by [18], which uses predictive analysis to better understand performance variability.

A central component of the MOANA methodology is the construction of a *variability map*, which is a function whose inputs are system and application level configuration parameters and whose output is a metric for performance variability. The variability map is constructed by applying approximation techniques to performance data that has been collected on real-world HPC systems. Cameron et al. [18] propose the usage of several nonparametric approximation techniques to construct the variability map.

Within the MOANA methodology, the Delaunay interpolant has been shown to be an effective model for predicting both I/O throughput variance [18] and mean I/O throughput [66]. This includes significantly out-performing other popular prediction techniques such as neural network regression, support vector regression, inverse distance weighting, and multivariate splines [66]. Delaunay interpolation has also been used to directly predict the cumulative distribution functions for I/O throughput performance (which are closed under convex combinations) in order to obtain nonparametric performance distribution predictions [67].

In this section, DELAUNAYSPARSE is applied in the MOANA framework in order to predict the I/O throughput variance of the `IOzone` benchmark application [21].

4.3.1 Data Collection

Data for the I/O throughput variance prediction problem has been gathered at Virginia Tech on a homogeneous cluster of shared-memory nodes running Ubuntu 14.04 LTS on a

dedicated 2TB HDD. Each node is a 2 socket, 4 core hyperthreaded Intel Xeon E5-2623 v3 (Haswell) processor with 32 GB DDR4. Data has been gathered by running the `IOzone` benchmark application. `IOzone` measures read/write throughputs by reading and writing files of configurable size, broken up into configurable record sizes, utilizing a configurable number of threads. For more information on `IOzone`, see [21]. Up to 13 different variations of read and write tasks can be tested, but in this section only the `fread` task is considered. To generate each data point, the `IOzone` benchmark application is run independently κ times for each parameter configuration, and the variance is computed for the maximum throughput of the `fread` task using

$$(\hat{\sigma}^{(\kappa)})^2 = \frac{\sum_{i=1}^{\kappa}(q^{(i)} - \hat{\mu}^{(\kappa)})^2}{\kappa - 1} \tag{4.1}$$

where $q^{(1)}, \ldots, q^{(\kappa)}$ are the observed throughputs and $\hat{\mu}^{(\kappa)}$ is the mean observed throughput over all κ runs.

The variance is modeled as a function of four system and application parameters, chosen because of their relevance to `IOzone`. These parameters are thread count, CPU frequency, file size, and record size. Note that the parameters thread count, file size, and record size are specifiable as `IOzone` inputs, while CPU frequency must be manually set for each run using system tools. To avoid biasing the data, the CPU cache has been purged between each run of an `IOzone` test. For further details on this dataset and methodology, see [18].

For each parameter, several values have been chosen spanning a range of reasonable values, and the observed variance has been calculated for each combination of settings using (4.1). For the data presented, the values chosen are in Table 4.1. Note that some combinations of parameters are not viable (specifically, the record size cannot be greater than the file size). These combinations have been omitted when collecting data. Every valid combination of settings from Table 4.1 has been evaluated to generate the dataset. Observe that there are 6 valid combinations of file and record size, 9 thread counts, and 16 distinct CPU frequencies. This results in $6 \times 9 \times 16 = 864$ total data points.

Table 4.1: System and `IOzone` configuration settings that are used to build the variability map with DELAUNAYSPARSE. Note, the frequency 3.001 GHz results from overclocking.

	values
file size (KB)	64, 256, 1024
record size (KB)	32, 128, 512
thread count	1, 2, 4, 8, 16, 32, 64, 128, 256
frequency (GHz)	1.2, 1.4, 1.5, 1.6, 1.8, 1.9, 2.0, 2.1, 2.3, 2.4, 2.5, 2.7, 2.8, 2.9, 3.0, 3.001

In the next section, predictions are made for selected points from the full dataset to show how the expected accuracy of the Delaunay interpolant scales in the MOANA framework with respect to the amount of data.

4.3.2 Model Evaluation

From the collected data, the following points in (4.2) have been selected for testing the Delaunay prediction.

$$\begin{aligned} \hat{x}^{(1)} &= (\text{fsize} = 256, \text{rsize} = 128, \text{threads} = 2, \text{freq} = 1.9) \\ \hat{x}^{(2)} &= (\text{fsize} = 256, \text{rsize} = 128, \text{threads} = 2, \text{freq} = 2.9) \\ \hat{x}^{(3)} &= (\text{fsize} = 1024, \text{rsize} = 32, \text{threads} = 4, \text{freq} = 2.9) \end{aligned} \tag{4.2}$$

Note that because of the restriction on valid combinations of file and record size, it is not possible to select points strictly inside the convex hull. Consequently, all the points in (4.2) are on the boundary of the convex hull. Various percentages of the remaining points are used by the Delaunay interpolant to predict the value of the throughput variance for all points in (4.2). For each of these "training percentages," up to 200 Delaunay interpolations are calculated using DELAUNAYSPARSE with different pseudo-random samplings from the complete set of data points, with a bias toward uniformly distributed samplings. These samplings are constrained in that the prediction of $\hat{F}_{DT}(\hat{x}^{(i)})$ cannot be based off a sampling that includes $\hat{x}^{(i)}$, and $\hat{x}^{(i)}$ cannot be outside the convex hull of the selected points. Also, repeated use of the same sampling in a single batch of 200 samplings has been forbidden. Note that because of the constraints, in some cases, less than 200 samplings could be gathered.

Figures 4.1, 4.2, and 4.3 show box plots of the relative errors observed when using the Delaunay interpolant to predict $\hat{x}^{(1)}$, $\hat{x}^{(2)}$, and $\hat{x}^{(3)}$ at each training percentage. Note that for the large training percentages, the boxes in Figures 4.1, 4.2, and 4.3 are very narrow. This is because large samplings have a relatively low probability for dropping a vertex from the Delaunay simplex containing $\hat{x}^{(i)}$, and unless some vertex of the containing Delaunay simplex is dropped, the interpolated value will not change for the Delaunay interpolant.

Using 90% of the 863 remaining data points (after excluding the point $\hat{x}^{(i)}$ that is being interpolated at) the Delaunay interpolant is fairly accurate, taking into consideration the difficulty of the problem. The convergence trend demonstrated in all three figures suggests that with more data, the Delaunay interpolant could provide an even better estimate of variance.

4.3.3 Other Applications

Building the variability map in the MOANA framework is just one of several known real-world applications of DELAUNAYSPARSE.

In other known applications, DELAUNAYSPARSE has been used to predict oblique shocks solutions during supersonic flow [59] and is currently being used for research in machine learning model validation at Lawrence Livermore National Laboratory.

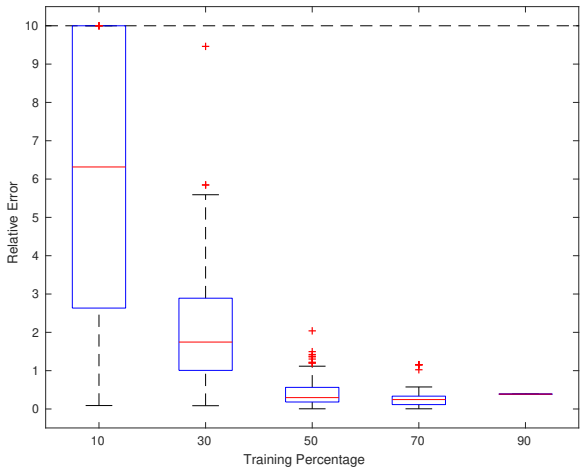

Figure 4.1: Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(1)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated.

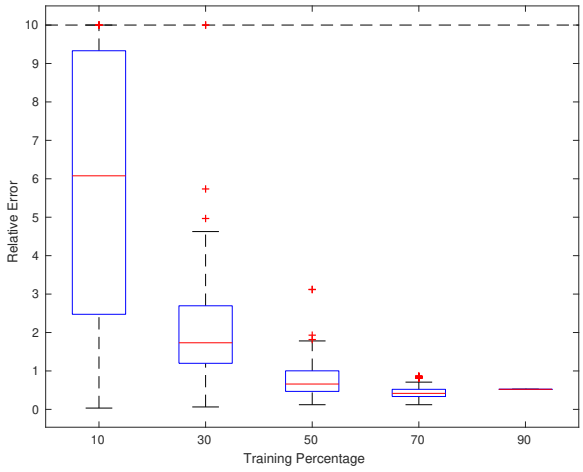

Figure 4.2: Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(2)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data. Note: Relative errors greater than 10 (1000%) have been truncated.

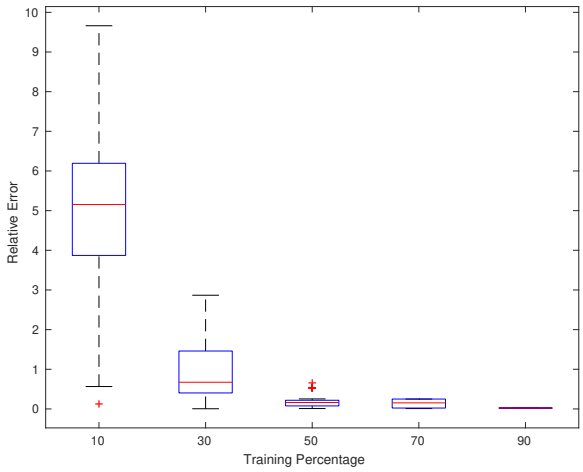

Figure 4.3: Box plot of the relative error for up to 200 variance predictions at $\hat{x}^{(3)}$ (4.2) using the Delaunay interpolant with various percentages of the total available data.

4.4 Tuning the HPL Benchmark Solver with VTMOP

In this section, toward the goal of managing computational throughput variability, the problem of optimizing the configuration parameters of the `HPL` benchmark solver is considered.

4.4.1 The HPL Benchmark Problem

As a problem that is representative of dense linear algebra workloads, consider the High-Performance LINPACK benchmark (HPLB) [43] problem. The TOP500 list [91] defines the HPLB as the problem of solving a dense system of linear equations (of any size) using an LU factorization with partial pivoting. Submissions to the TOP500 list are allowed to present results of solving the HPLB of arbitrary size N and using any linear system solver, as long as the implementation has a floating-point operation count of $\frac{2}{3}N^3 + \mathcal{O}(N^2)$ and solves the problem in 64-bit precision. The software package `HPL` [82] is a portable linear system solver that can utilize massive parallel resources to solve the HPLB. `HPL` uses a block-cyclic data distributing, recursive panel factoring, right-looking variant of the LU-decomposition algorithm.

`HPL` accepts an input file specifying numerous parameters to adjust the underlying algorithm. Properly identifying the parameters that maximize the observed throughput of `HPL` is a notoriously difficult task but has a significant impact on the observed computational throughput [93]. Traditionally, `HPL` is optimized by experimenting with algorithmic parameters and following the recommendations of [82]. When submitting performance results to the TOP500 list, researchers may run `HPL` many times with various settings and report only

the maximum observed throughput. In previous work, genetic algorithms have been applied to the problem of optimizing HPL, treating this task as a blackbox optimization problem [44].

This section investigates the effects of throughput variability on performance when solving a dense linear system using HPL and models the inherent tradeoff between HPL's mean throughput and throughput standard deviation. The fundamental algorithm that is implemented by HPL is similar to others, such as that used by the PDGESV driver from ScaLAPACK, which also uses parallel resources to solve a dense linear system of equations [13]. Therefore, as an example of a dense linear algebra workload, HPL can be used to gain insight into the effects of performance variability on parallel computational linear algebra tasks. A similar problem of interest is the usage of blackbox optimization to empirically optimize the basic linear algebra subroutines (BLAS) for maximum performance on a given system [100]. However, previous works in linear algebra library tuning generally have not investigated the effects of performance variability.

In this section, HPL is optimized on an HPC system by using VTMOP to simultaneously minimize throughput standard deviation and maximize the mean throughput. This produces approximations to the Pareto optimal parameters for HPL, which produce throughput distributions along the tradeoff curve between mean throughput and throughput standard deviation. The goals of these experiments are to

- understand the shape of the tradeoff curve between throughput mean and standard deviation when solving large dense linear algebra problems on parallel resources and
- identify parameters for HPL that balance this tradeoff.

Two experiments are performed. The first is a single-node analysis, where HPL is optimized for a fixed problem size on a single computing node. The second is a four-node study, where HPL is optimized on a significantly larger problem size with access to distributed computing resources. The second study requires much more computation time but is a more accurate representation of a large distributed simulation's workload.

4.4.2 The HPL Solver

The driver for HPL solves a linear system $A\hat{\xi} = b$, where $A \in \mathbb{R}^{N\times N}$ and $b \in \mathbb{R}^N$. This is done by decomposing $PA = LU$, where L is lower triangular, U is upper triangular, and P is a permutation matrix reflecting row-interchanges. The vector $\hat{\zeta} = L^{-1}Pb$ is computed during the factorization. Then the system $U\hat{\xi} = \hat{\zeta}$ is solved by using back-substitution. Before computing the decomposition, the parameter $EQUIL$ specifies whether A will be equilibrated so that its rows and columns have approximately equal magnitudes. Also, $ALIGN$ specifies the byte alignment for double precision numbers. To compute the decomposition, A is decomposed into $NB \times NB$ blocks, which are cyclically distributed over a two-dimensional grid of $PG \times QG$ processors. The parameter $PMAP$ specifies whether these blocks are mapped

in row- or column-major order.

The main loop of HPL's algorithm works rightward from the leftmost columns of A. In each iteration, a panel of NB columns is factored by recursively dividing each panel into $NDIVS$ subpanels until each subpanel has a size of less than or equal to $NBMIN$. The parameters $RFACT$ and $PFACT$ specify the algorithm variants (right-looking, left-looking, or Crout's method) for recursively dividing each panel and solving each $NBMIN$-sized subpanel, respectively. After a panel has been factored, each processor must broadcast updates, using the broadcast topology $BCAST$. Six broadcast topologies are available, but the modified increasing ring is strongly recommended. The broadcasts are used to update the lower right submatrix. While the broadcast operation is being completed, it is possible to begin updating the next column of the submatrix using a lookahead pipe, whose depth is $DEPTH$. For each update, users have the choice between two update algorithms ($SWAP$): binary exchange or spread-roll. Binary exchange is preferred when the number of columns in the submatrix is small, and spread-roll is preferred when it is large; a mix of the two can be used when a threshold SN for swapping between the two algorithms is provided. Each panel's lower factors $L^{(1)}$ and the final upper factor U can be stored in either transposed or non-transposed format, as specified by $TL^{(1)}$ and TU, respectively. After completing the linear solve, the residual is checked against an error tolerance ET to determine whether the HPLB has been solved to an acceptable precision.

The parameters covered in the preceding paragraphs are summarized in Table 4.2, along with their recommended values/ranges for achieving the maximum overall throughput [82].

Table 4.2: Tuning Parameters for HPL

Parameter	**Meaning**	**Value(s)**	**Recommendation**
N	problem size	positive integer	80% of RAM
NB	block size	positive integer	$32, 33, \ldots, 256$
$PMAP$	process mapping	row- or col.-major	row-major
PG	1st dim. of process grid	positive integer	$PG = QG$
QG	2nd dim. of process grid	positive integer	$QG = PG$
ET	error tolerance	real number	16.0
$PFACT$	LU variant for $NBMIN$-sized panels	right, left, Crout	Crout
$NBMIN$	recursion stopping criterion	positive integer	4 or 8
$NDIVS$	divisions per level of recursion	positive integer	2
$RFACT$	LU variant for panel recursion	right, left, Crout	right
$BCAST$	broadcast topology	6 choices	mod. incr.-ring
$DEPTH$	lookahead pipe depth	nonnegative integer	1
$SWAP$	update algorithm	bin-ex, spread-roll, mix	mix
SN	swapping threshold (for mix)	positive integer	NB
$TL^{(1)}$	transpose $L^{(1)}$	Yes or No	Yes
TU	transpose U	Yes or No	Yes
$EQUIL$	equilibrate A	Yes or No	Yes
$ALIGN$	double alignment in bytes	positive integer	8

4.4.3 Integrating HPL as a VTMOP Objective Function

Not all of the parameters in Table 4.2 can be optimized by using VTMOP, and not all of these parameters are desirable to adjust: for example, adjusting the error tolerance ET could allow for problems to be solved at undesirable precision. For the single-node study, `HPL` is optimized for the problem size $N = 10,000$, and the 6 integer-valued parameters NB, PG, $NBMIN$, $NDIVS$, $DEPTH$, and SN are adjusted by VTMOP. Note that only one dimension (PG) of the process grid is adjusted. Since the process grid's dimensions should match the number of available processors (36 for this study), QG is inferred using $QG = \lceil 36/PG \rceil$. The bound constraints for the six adjustable variables are summarized in Table 4.3. All other parameters are fixed to the values recommended by [82] as shown in Table 4.3.

Table 4.3: Bounds for Adjustable Inputs When Tuning `HPL`

Parameter	**Lower Bound**	**Upper Bound**
NB	1	256
PG	1	36
$NBMIN$	1	256
$NDIVS$	2	36
$DEPTH$	0	4
SN	1	256

For each parameter configuration that is supplied to `HPL`, the mean overall throughput and the throughput standard deviation are estimated. Similarly as in Section 4.3, this estimate is achieved by running `HPL` κ times for each configuration and computing the observed sample mean $\hat{\mu}^{(\kappa)}$ and the unbiased estimate for the standard deviation $\hat{\sigma}^{(\kappa)}$. The primary difference between here and Section (4.3) is that `HPL` task is compute bound and the throughputs $q^{(i)}$ from (4.1) now represent computational throughput (in Gflops) as opposed to I/O throughput values. Another key difference is that rather than using the variance estimate $(\hat{\sigma}^{(\kappa)})^2$, this section uses the standard deviation $\hat{\sigma}^{(\kappa)}$, which is the square-root of the variance.

Similarly as in Section 4.3, $\hat{\mu}^{(\kappa)}$ and $\hat{\sigma}^{(\kappa)}$ are computed after a sample of $\kappa = 40$ runs of `HPL`. On a single Broadwell node of `Bebop`, the total time for 40 runs of `HPL` with a problem size of $N = 10,000$ is more than a minute for all parameter configurations considered; certain suboptimal configurations can take much longer. The following steps are taken in order to prevent VTMOP from spending too much time performing 40 sample evaluations of parameters that are clearly suboptimal. For every parameter x, the mean and standard deviation estimates are computed after just $\kappa = 5$ runs of `HPL`. Let $\hat{\mu}'$ and $\hat{\sigma}'$ be the early termination thresholds. If the five sample estimates for the throughput mean and standard deviation satisfy

$$\hat{\mu}^{(5)} < \hat{\mu}' \quad \textbf{and} \quad \hat{\sigma}^{(5)} > \hat{\sigma}', \tag{4.3}$$

then the current run is aborted, and the five sample estimates are immediately returned.

In order to determine the values of $\hat{\mu}'$ and $\hat{\sigma}'$ to use in (4.3), `HPL` was run 40 times with the recommended settings of $NB = 128$, $PG = 6$, $NBMIN = 8$, $NDIVS = 2$, $DEPTH = 1$, and $SN = 128$. The resulting estimates for throughput mean and standard deviation (rounded to three decimal places) are $\hat{\mu}^{(40)} = 613.041$ and $\hat{\sigma}^{(40)} = 3.800$, and the early termination thresholds are assigned values $\hat{\mu}' = 300$ and $\hat{\sigma}' = 8$.

VTMOP can be initialized with a database of precomputed function evaluations. Before starting the single-node study, VTMOP is given an initial database containing the observed mean throughput and throughput standard deviation from running `HPL` with the recommended parameter evaluation. This inclusion serves as a sanity check since any parameter configurations whose result is dominated by the recommended parameters will not appear in the solution set.

As discussed in Section 3, VTMOP solves MOPs that are real-valued minimization problems. As posed above, `HPL` is an integer-valued min/max problem. The following adjustments to VTMOP are made for compatibility.

- The parameter space tolerance (an optional input to VTMOP) is set to 0.99. This prevents VTMOP from evaluating any two points in the parameter space whose Euclidean distance is less than or equal to 0.99 and prevents VTDIRECT95 from dividing any box whose diameter is less than 0.99.
- During the surrogate model optimization phase, the GPSMADS mesh size is restricted to an integer value. This ensures that each batch of candidate parameters values will be spaced on an integer lattice, offset by the position of the current trust region center.
- Before each set of parameters requested by VTMOP is evaluated (by running `HPL`), all of its components are "binned" by rounding to the nearest integer.
- Tuning `HPL` is posed as a MOP with the objective $F(x) = (-\hat{\mu}^{(\kappa)}, \hat{\sigma}^{(\kappa)})$.

4.4.4 Tuning HPL on a Single Node with VTMOP

The single-node optimization of `HPL` takes place on an Intel Broadwell node of the HPC system `Bebop` at Argonne National Laboratory. Each Broadwell node is a 36-core Intel Xeon E5-2695v4 processor with 128 GB of DDR4 RAM. The `HPL` executable is built by using the Intel 17.0.4 C compiler and the corresponding Intel Parallel Studio implementation of MPI and Intel Math Kernel Library (MKL) implementation of the BLAS. VTMOP is built by using the Intel 17.0.4 Fortran compiler.

Figure 4.4 shows an approximation to the tradeoff curve between the mean throughput and throughput standard deviation in 40 runs `HPL` on a single node with $N = 10,000$. These results were attained by using VTMOP with a budget of 2,000 evaluations. Note that

the scale of the mean throughput is two orders of magnitude larger than the scale of the throughput standard deviation.

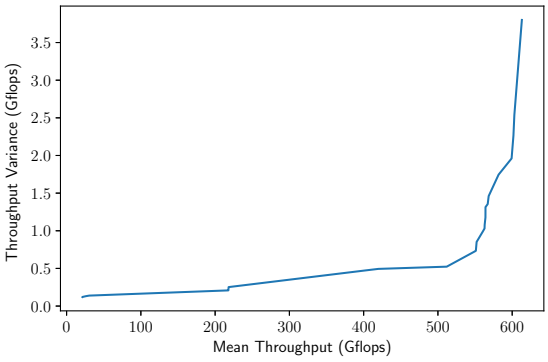

Figure 4.4: Tradeoff curve between mean throughput and throughput standard deviation in 40 runs of `HPL` when using the approximate Pareto optimal configurations on a single node with $N = 10,000$.

VTMOP found 24 approximate Pareto optimal configurations, as shown in Figure 4.4 and whose values are given in Table 4.4 listed in ascending order by mean throughput/throughput standard deviation. Note that the four configurations in Table 4.4 with the lowest overall mean/standard deviation have dismal throughputs and are probably not of interest to most readers. Also note that whenever $NBMIN \geq NB$, no recursion takes place, and therefore the variable $NDIVS$ is unused and meaningless.

For the 20 configurations that offer reasonable throughputs, the trends in Table 4.4 indicate that the "simple" variations of the `HPL` algorithm (i.e., no lookahead and no recursion) produce significantly lower standard deviations at a steep cost to the mean throughput. On the other hand, using lookahead and allowing for many levels of recursion (i.e., $NBMIN$ and $NDIVS$ small) result in maximal mean throughput, at the cost of increased variability. Each of these trends has several discontinuities, which could indicate that either the underlying Pareto front/efficient set is discontinuous or VTMOP did not fully converge in certain regions of the Pareto front with the given budget. With the exception of the four extremely low throughput settings, all of the approximate Pareto optimal configurations have block sizes that are close to the recommended value of 128. The recommended configuration that was supplied in the initial database is Pareto optimal and achieves the highest mean throughput; it is the last row in Table 4.4.

In order to further understand the throughput distributions that are shown in Figure 4.4 and Table 4.4, the 12 configuration with the highest mean throughputs were used for 100 runs of `HPL`, and the resulting histograms are shown in Figure 4.5. The distributions are ordered from left to right, top to bottom in ascending order by mean/standard deviation.

Figure 4.5 shows that each of the throughput distributions has a few outliers that are far

Table 4.4: Approximate Pareto Optimal Set for Single-Node Runs of HPL with $N = 10,000$

$\hat{\mu}$	$\hat{\sigma}$	NB	PG	$NBMIN$	$NDIVS$	$DEPTH$	SN
21.31	0.120	2	19	48	27	3	122
21.35	0.121	2	19	48	27	3	123
25.89	0.131	3	25	47	28	2	122
30.48	0.139	3	20	47	28	3	122
217.82	0.208	129	1	129	19	0	129
218.15	0.252	129	1	256	2	0	1
400.37	0.471	128	1	16	10	0	128
419.73	0.494	129	1	1	2	0	1
511.87	0.523	214	4	15	33	0	72
551.07	0.734	204	4	3	33	0	62
552.12	0.852	204	4	25	35	0	62
560.54	0.991	204	4	15	23	0	185
562.78	1.030	204	4	6	22	0	185
562.84	1.053	204	4	6	33	0	66
562.95	1.080	204	4	6	23	0	182
564.01	1.177	204	4	6	22	0	195
564.06	1.314	204	4	6	22	0	191
567.05	1.355	204	4	9	22	0	191
568.34	1.461	133	3	9	3	2	128
581.69	1.746	128	3	9	3	2	123
599.21	1.961	128	3	4	3	1	128
601.69	2.264	128	3	9	9	1	123
602.93	2.539	128	3	9	6	1	124
613.04	3.800	128	6	8	2	1	128

below the median value. When running numerous numerical simulations in a batch, the presence of a few "low" outliers could slow the entire computation [37]. For a few of the distributions, there appears to be a cluster of lower throughput values, indicating a multimodal distribution. This mirrors the findings of [101] for the distribution of I/O throughputs.

The findings in this section provide some insight into parameters for HPL that can be used to control the tradeoff between mean throughput and throughput variability. However, since HPL is intended for usage in a distributed memory environment, single-node runs are less interesting than multi-node runs. In the Section 4.4.5, this study is expanded to the multinode case.

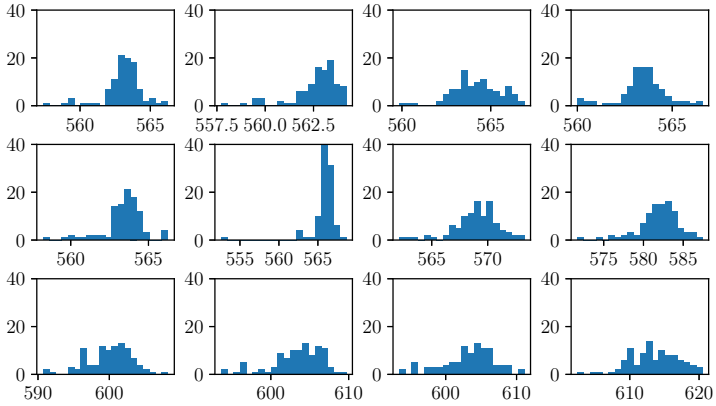

Figure 4.5: Histograms of observed throughputs when running `HPL` on a single node with $N = 10,000$, for the 12 highest mean throughput configurations. Distributions are ordered from left to right, top to bottom in ascending order by mean/standard deviation.

4.4.5 Tuning HPL on Multiple Nodes

In this section, `HPL` is optimized for four 36-core Intel Broadwell nodes, each of which is as described in Section 4.4.4. Thus, there are 144 total processors available and 512 GB of distributed RAM. In order to ensure that there is enough work for all 144 processors, the problem size considered in this section is increased to $N = 20,000$. The operation count for `HPL` grows cubically with the problem size, so doubling N results in each run of `HPL` requiring roughly eight times as many operations as the runs from Section 4.4.4.

The experiment is configured as in Section 4.4.4, with the following adjustments.

- Because evaluating the objectives is more expensive, a sample of only $\kappa = 30$ runs of `HPL` is used to compute each configuration's mean and standard deviation, and VTMOP is given a budget of only 1,000 evaluations of F (half the budget from Section 4.4.4).
- Because the budget for evaluating F has been lowered, the variable NB is eliminated in order to reduce the size of the search space. Based on the recommendations of [82] and the results from Section 4.4.4, $NB = 128$. This results in a five-variable problem ($d = 5$).
- Because the processor count is now 144, the upper bound for PG (as previously listed in Table 4.3) is increased to 144, and QG is inferred by using $QG = \lceil 144/PG \rceil$.
- The five-sample thresholds in (4.3) are $\hat{\mu}' = 1000$ and $\hat{\sigma}' = 40$, based on 40 runs with the recommended settings of $PG = 12$, $NBMIN = 8$, $NDIVS = 2$, $DEPTH = 1$, and $SN = 128$, which produced the estimates $\hat{\mu}^{(40)} = 2236.558$ and $\hat{\sigma}^{(40)} = 21.362$. As in Section 4.4.4, VTMOP is initialized with these precomputed values in its database.

Figure 4.6 shows an approximation to the tradeoff curve between the mean throughput and

throughput standard deviation in 30 runs of `HPL` on four nodes with $N = 20,000$. The shape of the tradeoff curve in Figure 4.6 is somewhat similar to the shape of the curve from Figure 4.4, but is less smooth, and the scale of the mean throughput is three orders of magnitude larger than the scale of the throughput standard deviation.

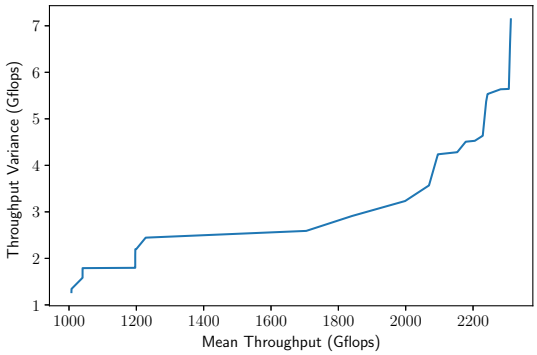

Figure 4.6: Tradeoff curve between mean throughput and throughput standard deviation in 30 runs of `HPL` when using the approximate Pareto optimal configurations on four nodes with $N = 20,000$.

Again, 24 approximate Pareto optimal configurations are identified, as shown in Figure 4.6. Their values are given in Table 4.5, listed in ascending order by mean throughput/throughput standard deviation. Recall that $NBMIN = 128$ for all of these configurations, so only one of the configurations in Table 4.5 ($PG = 8$, $NBMIN = 129$, $NDIVS = 20$, $DEPTH = 0$, $SN = 138$) results in no recursion, and every other configuration results in exactly one level of recursion.

Unlike in Section 4.4.4, the recommended configuration of $PG = 12$, $NBMIN = 8$, $NDIVS = 2$, $DEPTH = 1$, and $SN = 128$, which produced the observations $\hat{\mu}^{(40)} = 2236.558$ and $\hat{\sigma}^{(40)} = 21.362$, is **not** one of the approximate Pareto optimal configurations listed in Table 4.5. Notice that the recommended configuration's mean throughput is relatively close to the fastest mean throughputs in Table 4.5, but the standard deviation of the recommended configuration is about three times the largest standard deviation in Table 4.5 and more than four times that of configurations with similar mean throughputs.

The values of $DEPTH$ and SN that produce the highest mean throughputs are identical or very similar to their recommendations. The value of PG that produces the highest mean throughput is $PG = 9$ (which implies $QG = 16$). Although the recommended setting used in this experiment was $PG = 12$, the recommendation of [82] is actually to use either $PG = QG$ **or** PG slightly less than QG, so $PG = 9$ is still in line with this recommendation. However, the values $NBMIN > 100$ and $NDIVS > 20$ seem to strongly contrast with the recommendations of [82] that $NBMIN = 4$ or 8 and $NDIVS = 2$. On closer inspection, with a fixed block size of $NB = 128$, using $NDIVS = 21, \ldots, 26$ (as in the highest mean configurations in Table 4.5) results in an effective minimum block size of between 4 and 6

Table 4.5: Approximate Pareto Optimal Set for Four-Node Runs of `HPL` with $N = 20,000$, $NB = 128$

$\hat{\mu}$	$\hat{\sigma}$	PG	$NBMIN$	$NDIVS$	$DEPTH$	SN
1007.1933	1.2741281	3	123	26	3	123
1007.4800	1.3432847	3	123	26	3	118
1040.2800	1.5831875	3	117	31	3	123
1040.3267	1.7903830	3	117	31	3	118
1196.7100	1.7968075	3	117	21	2	123
1196.7167	2.0589725	3	117	21	2	121
1197.0267	2.1971664	3	117	23	2	123
1199.9100	2.2000549	3	116	21	2	123
1227.7600	2.4454885	3	112	31	2	123
1704.5900	2.5906130	8	129	20	0	138
1840.7733	2.9114439	6	117	26	3	121
1998.3433	3.2330922	7	117	26	3	123
2069.1700	3.5682653	9	124	21	3	134
2095.6033	4.2372310	9	114	21	3	134
2153.0533	4.2825011	6	123	21	1	127
2178.2900	4.5079508	6	117	21	1	121
2205.2133	4.5274438	6	112	15	1	121
2228.8300	4.6360767	9	112	21	2	123
2238.9800	5.3728821	9	107	15	2	123
2243.0733	5.5329068	7	114	21	1	123
2281.7467	5.6351565	9	119	23	1	129
2306.3233	5.6427973	9	114	23	1	123
2309.9733	6.6279416	9	107	21	1	123
2312.3500	7.1394364	9	107	26	1	118

after a single level of recursion. This *partially* agrees with the recommendations of [82], who recommend a similar minimum block size after many more levels of recursion.

Based on these results, one reasonable hypothesis is that the settings of $PG = 9$, $NBMIN \in [4, 8]$, $DEPTH = 1$, and $SN = 128$ produce the highest mean throughputs, and a large number of divisions per level (resulting in very few levels of recursion before achieving $NBMIN$) produces low variance. In order to check this hypothesis, 4 additional configurations are evaluated with a budget of 100, whose results are given in Table 4.6. Of the configurations in Table 4.6, the two configurations with $NB = 4$ are nondominated with respect to other configurations in Table 4.5, exhibiting significantly higher mean throughputs than the configurations found by VTMOP at the cost of a steep increase in the throughput standard deviation. These results somewhat support the hypothesis that the number of levels of recursion is correlated with higher throughput standard deviation. However, the last few entries of Table 4.5 offer significantly lower standard deviations, which suggest that the configurations $NDIVS = 21, \ldots, 26$ and $SN = 118, \ldots, 123$ may further affect the mean throughput and throughput standard deviation.

Table 4.6: Additional Evaluations of `HPL` with $N = 20,000$, $NB = 128$

$\hat{\mu}$	$\hat{\sigma}$	PG	$NBMIN$	$NDIVS$	$DEPTH$	SN
2330.69	15.178	9	4	2	1	128
2319.27	14.011	9	8	2	1	128
2323.22	14.302	9	4	16	1	128
2310.85	13.302	9	8	32	1	128

Figure 4.7 (left) shows histograms of observed throughputs in 100 runs of `HPL` with the 12 configurations from Table 4.5 with the highest mean throughput, listed from left to right, top to bottom in ascending order by mean/standard deviation. Figure 4.7 (right) shows histograms of observed throughputs in 100 runs of `HPL` with the two nondominated configurations from Table 4.6. The maximum throughput distribution (from Table 4.6) has a flatter shape, associated with its higher standard deviation, while other high-throughput configurations are either normally distributed or left-skewed.

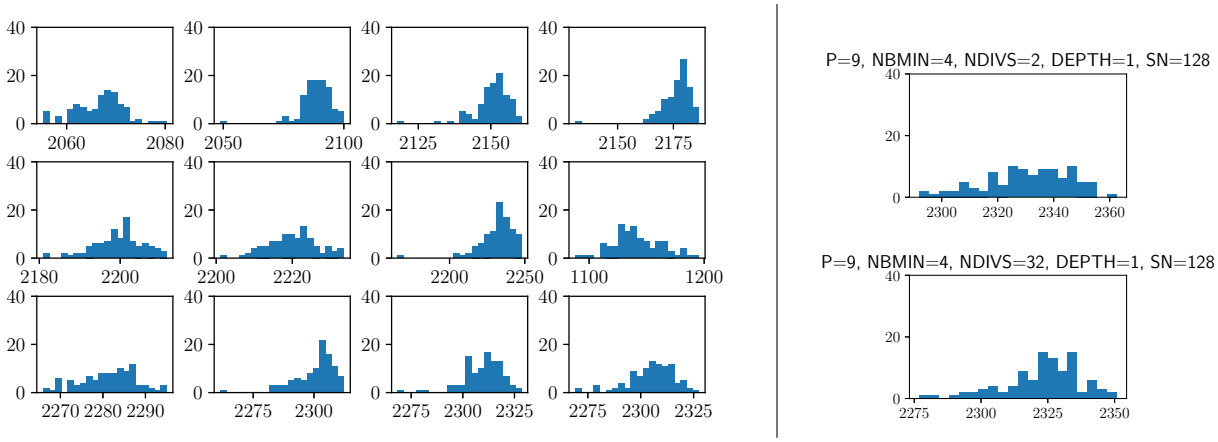

Figure 4.7: Histograms of observed throughputs when running `HPL` on four nodes with $N = 20,000$, $NB = 128$. The throughput distributions for the 12 highest mean throughput configurations are from Table 4.5 (left) and the two nondominated configurations are from Table 4.6 (right).

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, two high-quality numerical software packages were produced to solve uniquely challenging problems in computational science. The first is DELAUNAYSPARSE, which solves multivariate interpolation problems and has already seen significant usage in its short life. The second is VTMOP, which solves blackbox multiobjective optimization problems. Both of these packages were demonstrated on a real-world HPC system performance problem.

Key takeaways are as follows.

- Delaunay interpolation is a viable tool in the modern data science setting and offers the benefit of years of rigorous theory. The package DELAUNAYSPARSE provides access to Delaunay interpolation for high-dimensional data science applications.

- Using DELAUNAYSPARSE in a nontraditional way, it is possible to construct the Delaunay graph in high (i.e., 100+) dimensions.

- Blackbox MOPs of medium computational expense can be solved by using the most advanced techniques from modern literature, such as response surface modeling, adaptive weighting schemes, and trust region methods. The package VTMOP provides access to these techniques within a flexible framework.

- Computer system performance variability can be effectively modeled and controlled using the techniques described in this thesis.

5.2 Future Work

As with any dissertation work, these conclusions inspire many new and exciting directions for future work.

5.2.1 The Future of DELAUNAYSPARSE

For DELAUNAYSPARSE, several interesting avenues for future research are as follows.

- Based on the discussion in Section 2.3.2, it will be interesting to further investigate the connection between Delaunay interpolation and linear programming. Specifically, it would be interesting to investigate whether the theory of linear programming can be used to further inform the theory of Delaunay interpolation, or vice versa.

- Note that when using the ReLU activation function and optimizing to zero training error, a neural network regressor becomes a piecewise linear interpolant. Because Delaunay interpolation is often considered to be an "optimal" piecewise linear interpolant [27, 68, 85], it would be interesting to study the Delaunay interpolant's relationship with widely used pretrained neural network architectures, many of which use ReLU and achieve zero training error.

- In other work, Delaunay triangulations have been defined for an arbitrary metric space [27]. It would also be interesting to investigate algorithms for computing the interpolant in these spaces, using similar techniques as DELAUNAYSPARSE.

- In order to achieve truly distributed memory parallelism, it would be ideal to distribute not just the interpolation points $\mathcal{Q}$, but also the data points $\mathcal{D}$. Fortune [48] proposed a sweepline algorithm for computing two-dimensional Delaunay triangulations that uses a transformation of the input set to easily determine whether each point affects the triangulation before the current position of the sweepline. Perhaps this strategy could be extended to higher dimensions and used to quickly filter out points that will clearly not contribute to the computation of $\hat{F}_{DT}(y)$, in order to distribute $\mathcal{D}$ based on the values in $\mathcal{Q}$.

- Finally, it would be interesting to investigate high-dimensional applications for the Delaunay graph, which can now be computed using DELAUNAYSPARSE based on the novel algorithm in Chapter 3.

5.2.2 The Future of VTMOP

In the case of VTMOP, opportunities for future work are as follows.

- VTMOP can take advantage of derivative information, when available, to fit higher-order surrogates. In particular, the FUN3D CFD analysis package [12] uses adjoint methods to offer derivatives for sensitivity analysis. Derivative information could be incorporated into `LSHEP` (as described in Remark 3.15) or into the Delaunay interpolant by applying DELAUNAYSPARSE to blend linear basis functions that are tangent planes (similarly to the usage of DELAUNAYSPARSE in [67]). It would be interesting to see how the usage

of derivatives affects the convergence rate of VTMOP.

- In Section 4.4 VTMOP was used to solve a blackbox multiobjective integer programming problem. One of the modifications for this use case was that GPSMADS was constrained to sample on an integer lattice. However, this lattice could be offset due to VTDIRECT95's lack of support for integer programming. By investigating search strategies that only sample on an integer lattice, VTMOP could be extended to handle multiobjective integer programming problems, without resorting to any "binning" operations.

- The resource utilization of VTMOP in the `libEnsemble` setting could be drastically improved by allowing for asynchronous batch evaluations. In particular, by allowing VTMOP to begin the next iteration before a batch of candidate designs has finished evaluation, VTMOP could take full advantage of the services provided by `libEnsemble` and fully utilize all compute nodes at all times. In order to achieve this, it would be necessarry to predict the "next most isolated point." This is a nontrivial problem since points that are neighboring the current LTR center will become "less isolated" after evaluating all candidate designs.

- There are many existing techniques for approximating the Delaunay triangulation and graph, many of which offer better scaling with the dimension [30]. Although computing the Delaunay graph tends to be a lower cost operation than solving the surrogate problems, it would be interesting to see how much iteration cost is saved by using an approximation to the Delaunay graph instead of computing the true Delaunay graph with Algorithm 3.7. This would need to be considered against the loss in accuracy compared to using the true Delaunay graph, which manifests in the spacing of the approximately nondominated point set.

- Finally, it would be interested to investigate further applications for multiobjective blackbox optimization. For the specific applications mentioned in Chapter 4, a natural next step is to consider more performance factors in the multiobjective analysis, such as energy consumption. Other possibilities include using VTMOP for training an "ensemble" of predictive models, which lie along the tradeoff curve between model complexity and training accuracy (often referred to as the bias-variance tradeoff curve in machine learning literature).

5.3 The Big Picture

The original goal of this work, as stated in Chapter 1, was to solve computationally expensive blackbox optimization problems. In this work, a first step was taken in that direction, solving problems that require many minutes to evaluate. Extensions to extremely expensive problems have been considered, by utilizing massive levels of concurrency on exascale hardware. However, the most expensive problems are still somewhat elusive, since they allow for

far too few function evaluations (i.e., less than 100 total evaluations). To perform multi-objective optimization with these problems will most likely require a marriage between the techniques used herein and many other modern techniques such as multifidelity modeling, robust optimization, and uncertainty quantification. Additionally, this will almost certainly require algorithms that "open the blackbox" in order to leverage domain specific knowledge. Because of the flexible nature of VTMOP, it can be used as a general framework in these settings, working toward solutions to extremely expensive blackbox problems.

The limitations listed above should not detract from the contributions made here. VTMOP and DELAUNAYSPARSE are both first-of-their-kind packages, offering meaningful solutions to a variety of real-world problems.

Bibliography

[1] Ajith Abraham, Lakhmi Jain, and Robert Goldberg, editors. *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications*. Advanced Information and Knowledge Processing Series. Springer Verlag, London, UK, 2005.

[2] Abdullah Al-Dujaili and S. Suresh. A Matlab toolbox for surrogate-assisted multiobjective optimization: a preliminary study. In *Companion to the Proceedings of the 2016 Genetic and Evolutionary Computation Conference (GECCO '16)*, pages 1209–1216, Denver, CO, USA, 2016. ACM.

[3] Brandon D. Amos, David R. Easterling, Layne T. Watson, William I. Thacker, Brent S. Castle, and Michael W. Trosset. Algorithm XXX: QNSTOP: Quasi-Newton algorithm for stochastic optimization. *ACM Transactions on Mathematical Software*, to appear, 2014.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philidelphia, PA, USA, 3 edition, 1999.

[5] Charles Audet and John E. Dennis. Mesh adaptive direct search algorithms for constrainted optimization. *SIAM Journal on Optimization*, 17(1):188–217, 2006.

[6] Charles Audet, Gilles Savard, and Walid Zghal. A mesh adaptive direct search algorithm for multiobjective optimization. *European Journal of Operational Research*, 204 (3):545–556, 2010.

[7] Charles Audet, Jean Bigeon, Dominique Cartier, and Sébastien Le Digabel. *European Journal of Operational Research*, submitted:39 pages, 2018. URL http://www.optimization-online.org/DB_FILE/2018/10/6887.pdf.

[8] Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. *Voronoi diagrams and Delaunay triangulations*. World Scientific Publishing Co., Hackensack, NJ, USA, 2013.

[9] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[10] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[11] Mikhail Belkin, Daniel Hsu, and Partha P. Mitra. Overfitting or perfect fitting? Risk bounds for classification and regression rules that interpolate. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, pages 2306–2317, Montréal, Canada, 2018. Curran Associates Inc.

[12] Robert T. Biedron, Jan Renee Carlson, Joseph M. Derlaga, Peter A. Gnoffo, Dana P. Hammond, William T. Jones, Bill Kleb, Elizabeth M. Lee-Rausch, Eric J. Nielson, Michael A. Park, Christopher L. Rumsey, James L. Thomas, Kyle B. Thompson, and William A. Wood. FUN3D Manual: 13.6. NASA Technical Memorandum (TM) 2019-220416, NASA Langley Research Center, Hampton, VA, USA, October 2019.

[13] L. Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*, volume 4. SIAM, 1997.

[14] Jean-Daniel Boissonnat, Olivier Devillers, and Samuel Hornus. Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension. In *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry (SCG '09)*, pages 208–216, Aarhus, Denmark, 2009. ACM.

[15] Adrian Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2): 162–166, 1981.

[16] Susan Burgee, Anthony A. Giunta, Vladimir Balabanov, Bernard Grossman, William H. Mason, Robert Narducci, Raphael T. Haftka, and Layne T. Watson. A coarse-grained parallel variable-complexity multidisciplinary optimization paradigm. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(4):269–299, 1996.

[17] Kirk W. Cameron, Layne T. Watson, Ali R. Butt, Danfeng Yao, and Yili Hong. VarSys: Managing variability in high-performance computing systems, 2018. NSF award number CNS-1838271.

[18] Kirk W. Cameron, Ali Anwar, Yue Cheng, Li Xu, Uday Li, Bo Ananth, Jon Bernard, Chandler Jearls, Thomas Lux, Yili Hong, Layne T. Watson, and Ali R. Butt. MOANA: Modeling and analyzing I/O variability in parallel system experimental design. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1843–1856, 2019.

[19] Emilio Fortunato Campana, Matteo Diez, Giampaolo Liuzzi, Stefano Lucidi, Riccardo Pellegrini, Veronica Piccialli, Francesco Rinaldi, and Andrea Serani. A multi-objective DIRECT algorithm for ship hull optimization. *Computational Optimization and Applications*, 71(1):53–72, 2018.

[20] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 329–344, Vancouver, Canada, 2017. USENIX Association.

[21] Don Capps, Carol Capps, Darren Sawyer, Jerry Lohr, George Dowding, Gary Little, Terry Capps, Robin Miller, Sorin Faibish, Raymond Wang, Tanmay Waghmare, Yansheng Zhang, Vernon Miller, Nick Principe, Zach Jones, Udayan Bapat, William Norcott, Isom Crawford, Kirby Collins, Al Slater, Scott Rhine, Mike Wisner, Ken Goss, Steve Landherr, Brad Smith, Mark Kelly, Alain CYR, Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner, Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone, Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root, Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer, Vangel Bojaxhi, Ben England, Vikentsi Lapa, and Alexey Skidanov. IOzone filesystem benchmark, January 2016. URL www.iozone.org. accessed 2016.

[22] Tyler H. Chang, Layne T. Watson, Thomas C. H. Lux, Jon Bernard, Bo Li, Li Xu, Godmar Back, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Predicting system performance by interpolation using a high-dimensional Delaunay triangulation. In *Proceedings of the 2018 Spring Simulation Conference (SpringSim 2018), the 26th High Performance Computing Symposium (HPC '18)*, page article no. 2, Baltimore, MD, USA, 2018. SCS.

[23] Tyler H. Chang, Layne T. Watson, Thomas C. H. Lux, Bo Li, Li Xu, Ali R. Butt, Kirk W. Cameron, and Yili Hong. A polynomial time algorithm for multivariate interpolation in arbitrary dimension via the Delaunay triangulation. In *Proceedings of the 2018 ACM Southeast Conference (ACMSE '18)*, page article no. 12, Richmond, KY, USA, 2018. ACM.

[24] Tyler H. Chang, Layne T. Watson, Thomas C. H. Lux, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Algorithm XXX: DELAUNAYSPARSE: Interpolation via a sparse subset of the Delaunay triangulation in medium to high dimensions. *ACM Transactions on Mathematical Software*, submitted, 2019.

[25] Tyler H. Chang, Jeffrey Larson, and Layne T. Watson. Multiobjective optimization of the variability of the high-performance LINPACK solver. In *Proceedings of the 2020 Winter Simulation Conference (WSC 2020)*, page submitted, Orlando, FL, USA, 2020. IEEE.

[26] Tyler H. Chang, Jeffrey Larson, Layne T. Watson, and Thomas C. H. Lux. Managing computationally expensive blackbox multiobjective optimization problems using libEnsemble. In *Proceedings of the 2020 Spring Simulation Conference (SpringSim 2020), the 28th High Performance Computing Symposium (HPC '20)*, page to appear, Fairfax, VA, USA, 2020. SCS.

[27] Long Chen and Jin-chao Xu. Optimal Delaunay triangulations. *Journal of Computational Mathematics*, 22(2):299–308, 2004.

[28] Elliott W. Cheney and William A. Light. *A Course in Approximation Theory*. Graduate Studies in Mathematics. AMS, Providence, RI, USA, 2009.

[29] Siu-Wing Cheng, Tamal K. Dey, and Jonathan R. Shewchuk. *Delaunay Mesh Generation*. Computer and Information Science Series. CRC Press, Boca Raton, FL, USA, 2012.

[30] Aruni Choudhary, Michael Kerber, and Sharath Raghvendra. Polynomial-sized topological approximations using the permutahedron. *Discrete & Computational Geometry*, 61(1):42–80, 2019.

[31] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. DeWall: A fast divide and conquer Delaunay triangulation algorithm in $\mathbb{E}^d$. *Computer-Aided Design*, 30(5):333–341, 1998.

[32] Kyle Cooper and Susan R. Hunter. PyMOSO: Software for multi-objective simulation optimization with R-PERLE and R-MinRLE. *INFORMS Journal on Computing*, to appear, 2020. doi: 10.1287/ijoc.2019.0902.

[33] George B. Dantzig. *Linear Programming and Extensions*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, Princeton, NJ, USA, 11 edition, 1998.

[34] Indraneel Das and John E. Dennis. Normal-boundary intersection: A new method for generating the Pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, 1998.

[35] Pradipta De, Ravi Kothari, and Vijay Mann. A trace-driven emulation framework to predict scalability of large clusters in presence of OS jitter. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 232–241, Tsukuba, Japan, 2008. IEEE.

[36] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, TELOS, Santa Clara, CA, USA, 3 edition, 2008.

[37] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[38] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwel, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[39] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable multi-objective optimization test problems. In *Proceedings of the 2002 IEEE Congress on Evolutionary Computation (CEC '02)*, volume 1, pages 825–830, Honolulu, HI, USA, 2002. IEEE.

[40] Shubhangi Deshpande, Layne T. Watson, Jiang Shu, and Naren Ramakrishnan. Data driven surrogate-based optimization in the problem solving environment WBCSim. *Engineering with Computers*, 27(3):211–223, 2011.

[41] Shubhangi Deshpande, Layne T. Watson, and Robert A. Canfield. Multiobjective optimization using an adaptive weighting scheme. *Optimization Methods and Software*, 31(1):110–133, 2016.

[42] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry (SCG '01)*, pages 106–114, Medford, MA, USA, 2001. ACM.

[43] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present, and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[44] Dominic Dunlop, Sebastien Varrette, and Pascal Bouvry. On the use of a genetic algorithm in high performance computing benchmark tuning. In *Proceedings of the 2008 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 105–113, Edinburgh, UK, 2008. IEEE.

[45] Herbert Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. In *Proceedings of the Fifth Annual Symposium on Computational Geometry (SCG '89)*, pages 145–151, Saarbruchen, West Germany, 1989. ACM.

[46] Matthias Ehrgott. *Multicriteria Optimization*. Lecture Notes in Economics and Mathematical Systems Series. Springer Science & Business Media, Heidelberg, Germany, 2 edition, 2005.

[47] Yukun Feng, Zuogang Chen, Yi Dai, Fei Wang, Jiqiang Cai, and Zhixin Shen. Multidisciplinary optimization of an offshore aquaculture vessel hull form based on the support vector regression surrogate model. *Ocean Engineering*, 166:145–158, 2018.

[48] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1): 153–174, 1987.

[49] Adam Hammouda, Andrew R. Siegel, and Stephen F. Siegel. Noise-tolerant explicit stencil computations for nonuniform process execution rates. *ACM Transactions on Parallel Computing*, 2(1):7:1–7:33, 2015.

[50] Richard J. Hanson and Karen H. Haskell. Algorithm 587: Two algorithms for the linearly constrained least squares problem. *ACM Transactions on Mathematical Software*, 8(3):323–333, 1982.

[51] Jian He, Alex Verstak, Layne T. Watson, and Masha Sosonkina. Performance modeling and analysis of a massively parallel DIRECT – part 1. *The International Journal of High Performance Computing Applications*, 23(1):14–28, 2009.

[52] Jian He, Layne T. Watson, and Masha Sosonkina. Algorithm 897: VTDIRECT95: Serial and parallel codes for the global optimization algorithm DIRECT. *ACM Transactions on Mathematical Software*, 36(3):17, 2009.

[53] Susan Hert and Michael Seel. dD convex hulls and Delaunay triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.0.2 edition, 2020. URL https://doc.cgal.org/5.0.2/Manual/packages.html#PkgConvexHullD.

[54] Stephen Hudson, Jeffrey Larson, Stefan M. Wild, David Bindel, and John-Luke Navarro. libEnsemble users manual. Technical Report Revision 0.6.0, Argonne National Laboratory, 2019. URL https://buildmedia.readthedocs.org/media/pdf/libensemble/latest/libensemble.pdf.

[55] Susan R. Hunter, Eric A. Applegate, Viplove Arora, and Bryan Chong. An introduction to multiobjective simulation optimization. *ACM Transactions on Modeling and Computer Simulation*, 29(1):1–36, 2019.

[56] ISO/IEC 1539-1:2004(E). Information technology – Programming languages – Fortran – Part 1: Base Language. Standard, International Organization for Standardization, Geneva, Switzerland, November 2004.

[57] ISO/IEC 1539-1:2010(E). Information technology – Programming languages – Fortran – Part 1: Base Language. Standard, International Organization for Standardization, Geneva, Switzerland, October 2010.

[58] Donald R. Jones, Cary D. Perttunen, and Bruce E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.

[59] Mohamed Jrad, Rakesh K. Kapania, Joseph A. Schetz, and Layne T. Watson. Self-learning, adaptive software for aerospace engineering applications: Example of oblique shocks in supersonic flow. In *AIAA Scitech 2019 Forum*, San Diego, CA, USA, 2019. AIAA.

[60] Victor Klee. On the complexity of d-dimensional Voronoi diagrams. *Archiv der Mathematik*, 34(1):75–80, 1980.

[61] Victor Klee and George J. Minty. How good is the simplex algorithm? *Inequalities*, III:159–175, 1972.

[62] William T. Kramer and Clint Ryan. Performance variability of highly parallel architectures. In *Proceedings of the International Conference on Computational Science*, pages 560–569, Melbourne, Australia, 2003. Springer.

[63] Sébastien Le Digabel. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software*, 37(4):44, 2011.

[64] Yehong Liu and Guosheng Yin. Nonparametric functional approximation with Delaunay triangulation learner. In *Proceedings of the 2019 IEEE International Conference on Big Knowledge (ICBK)*, pages 167–174, Beijing, China, 2019. IEEE.

[65] Giampaolo Liuzzi, Stefano Lucidi, and Francesco Rinaldi. A derivative-free approach to constrained multiobjective nonsmooth optimization. *SIAM Journal on Optimization*, 26(4):2744–2774, 2016.

[66] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Jon Bernard, Bo Li, Li Xu, Godmar Back, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Predictive modeling of I/O characteristics in high performance computing systems. In *Proceedings of the 2018 Spring Simulation Conference (SpringSim 2018), the 26th High Performance Computing Symposium (HPC '18)*, page article no. 8, Baltimore, MD, USA, 2018. SCS.

[67] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Jon Bernard, Bo Li, Xiadong Yu, Li Xu, Godmar Back, Ali R. Butt, Kirk W. Cameron, and Yili Hong. Nonparametric distribution models for predicting and managing computational performance variability. In *Proceedings of IEEE SoutheastCon 2018*, pages 1–7, St. Petersburg, FL, USA, 2018. IEEE.

[68] Thomas C. H. Lux, Layne T. Watson, Tyler H. Chang, Yili Hong, and Kirk W. Cameron. Interpolation of sparse high-dimensional data. *Numerical Algorithms*, submitted, 2019.

[69] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, USA, 2018. USENIX Association.

[70] R. Timothy Marler and Jasbir S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[71] Nimrod Megiddo. On finding primal- and dual-optimal bases. *ORSA Journal on Computing*, 3(1):63–65, 1991.

[72] Paul Messina. The exascale computing project. *Computing in Science Engineering*, 19 (3):63–67, 2017.

[73] G. Miller, T. Phillips, and D. Sheehy. Linear-size meshes. In *Proceedings of the 20th Canadian Conference on Computational Geometry (CCCG 2008)*, pages 175–178, Montréal, Québec, 2008.

[74] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry*, 12(1):63–83, 1999.

[75] Juliane Müller. SOCEMO: Surrogate optimization of computationally expensive multiobjective problems. *INFORMS Journal on Computing*, 29(4):581–596, 2017.

[76] Raymond H. Myers, Douglas C. Montgomery, and Christine M. Anderson-Cook. *Response Surface Methodology: Process and Design Optimization Using Designed Experiments*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 4 edition, 2016.

[77] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer Science & Business Media, Heidelberg, Germany, 2 edition, 2006.

[78] Stephen M. Omohundro. Geometric learning algorithms. *Physica D: Nonlinear Phenomena*, 42(1):307–321, 1990.

[79] OpenMP 4.5. OpenMP Application Programming Interface version 4.5. Standard, OpenMP Architecture Review Board (ARB), November 2015.

[80] George Papazafeiropoulos. MATLAB computational geometry toolbox version 1.2. Technical report, MathWorks, November 2014. URL https://www.mathworks.com/matlabcentral/fileexchange/48509-computational-geometry-toolbox. accessed April 27, 2020.

[81] Tapasya Patki, Jayaraman J. Thiagarajan, Alexis Ayala, and Tanzima Z. Islam. Performance optimality or reproducibility: that is the question. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–30, Dallas, TX, USA, 2019. ACM.

[82] Antoine Petitet, R. Clint Whaley, Jack Dongarra, and Andy Cleary. *HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. Version 2.3. https://www.netlib.org/benchmark/hpl, accessed April 5, 2020, 2018.

[83] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, pages 55–55, Phoenix, AZ, USA, 2003. ACM.

[84] Chaitra Raghunath, Tyler H. Chang, Layne T. Watson, Mohamad Jrad, Rakesh K. Kapania, and Raymond M. Kolonay. Global deterministic and stochastic optimization in a service oriented architecture. In *Proceedings of the 2017 Spring Simulation Conference (SpringSim 2017), the 25th High Performance Computing Symposium (HPC '17)*, page article no. 7, Virginia Beach, VA, USA, 2017. SCS.

[85] V. T. Rajan. Optimality of the Delaunay triangulation in $\mathbb{R}^d$. *Discrete & Computational Geometry*, 12(2):189–202, 1994.

[86] Rommel G. Regis. The calculus of simplex gradients. *Optimization Letters*, 9(5): 845–865, 2015.

[87] Jong-hyun Ryu, Sujin Kim, and Hong Wan. Pareto front approximation with adaptive weighted sum method in multiobjective simulation optimization. In *Proceedings of the 2009 Winter Simulation Conference (WSC '09)*, pages 623–633, Austin, TX, USA, 2009. IEEE.

[88] Gautam M. Shroff and Christian H. Bischof. Adaptive condition estimation for rank-one updates of qr factorizations. *SIAM Journal on Matrix Analysis and Applications*, 13(4):1264–1278, 1992.

[89] Steve Smale. Mathematical problems for the next century. *The Mathematical Intelligencer*, 20(2):7–15, 1998.

[90] Jaroslaw Sobieszczanski-Sobieski, Alan Morris, and Michel Van Tooren. *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*. John Wiley & Sons, Ltd., Chichester, UK, 2015.

[91] Eric Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *The Top 500 List*. https://www.top500.org, accessed April 18, 2020, November 2019.

[92] Peter Su and Robert L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry*, 7(5):361–385, 1997.

[93] Tua Zea Tan, Rick Sow Mong Goh, Verdi March, and Simon See. Data mining analysis to validate performance tuning practices for hpl. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, New Orlean, LA, USA, 2009. IEEE.

[94] William I. Thacker, Jingwei Zhang, Layne T. Watson, Jeffrey B. Birch, Manjula A. Iyer, and Michael W. Berry. Algorithm 905: SHEPPACK: Modified Shepard algorithm for interpolation of scattered multivariate data. *ACM Transactions on Mathematical Software*, 37(3):34, 2010.

[95] U.S. Department of Energy, Argonne National Laboratory. Press release: U.S. Department of Energy and Intel to deliver first exascale supercomputer, March 2019. URL https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer. accessed April 28, 2020.

[96] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *Proc. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 513–527, Santa Clara, CA, USA, 2020. USENIX Association.

[97] Walter H. Vandevender and Karen H. Haskell. The SLATEC mathematical subroutine library. *SIGNUM Newsletter*, 17(3):16–21, 1982.

[98] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.

[99] David F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[100] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[101] Li Xu, Yueyao Wang, Thomas Lux, Tyler Chang, Jon Bernard, Bo Li, Yili Hong, Kirk Cameron, and Layne Watson. Modeling I/O performance variability in high-performance computing systems using mixture distributions. *Journal of Parallel and Distributed Computing*, 139:87–89, 2020.

[102] Amy K. Y. Yee, Ajay K. Ray, and G. P. Rangaiah. Multiobjective optimization of an industrial styrene reactor. *Computers & Chemical Engineering*, 27(1):111–130, 2003.

Appendices

Appendix A

Proofs

A.1 Proof of Lemma 2.9

Lemma 2.9. Let $\mathcal{D}$ be a set of points in $\mathbb{R}^d$ in general position, and let $\{s^{(1)}, \ldots, s^{(j)}\}$ be the vertex set for a $(j-1)$-face ($j \leqq d$) with the following property: the open ball whose boundary is the minimum radius $(d-1)$-sphere passing through $\{s^{(1)}, \ldots, s^{(j)}\}$ has empty intersection with $\mathcal{D}$. Let $x^* \in \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$ be the minimizer for $\|\xi^*\|$, as in Algorithm 2.5. Then $\{s^{(1)}, \ldots, s^{(j)}, x^*\}$ is the vertex set for a j-face whose minimum radius circumsphere has the same property.

Proof. The set of centers of all $(d-1)$-spheres through $\{s^{(1)}, \ldots, s^{(j)}\}$ is a $(d+1-j)$-dimensional linear manifold $\mathcal{M}$ (the solution set of $j-1$ linear equations). Let $\mathcal{B}^*$ be the open ball with center c^* and radius r^*, whose boundary is the minimum radius $(d-1)$-sphere through $\{s^{(1)}, \ldots, s^{(j)}, x^*\}$.

If there were a $x^\dagger \in \mathcal{B}^* \cap \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(j)}\}$, then by continuity of $\mathcal{M}$, $\|x^\dagger - c^\dagger\|_2 \leqq \|x^\dagger - c^*\|_2 < r^*$, where $c^\dagger$ is the center of the minimum radius circumsphere passing through $\{s^{(1)}, \ldots, s^{(j)}, x^\dagger\}$. Hence there exists an open ball $\mathcal{B}^\dagger$ centered at $c^\dagger$ whose boundary passes through $\{s^{(1)}, \ldots, s^{(j)}, x^\dagger\}$ and whose radius is less than r^*, which contradicts the definition of c^*, r^*, and $\mathcal{B}^*$. Therefore, $\mathcal{B}^* \cap \mathcal{D} = \emptyset$. □

A.2 Definition and Proof of Acyclicity

Consider the following definition of acyclicity and summary Edelsbrunner's proof.

Definition A.1. Let $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}$ be simplices in $DT(\mathcal{D})$. Then $\mathcal{S}^{(1)} <_y \mathcal{S}^{(2)}$, (read "$y$ is visible to $\mathcal{S}^{(1)}$ with respect to $\mathcal{S}^{(2)}$"), if every ray $\mathcal{R}$ drawn from y that has a nonempty intersection with both $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$ intersects $\mathcal{S}^{(1)}$ "before" $\mathcal{S}^{(2)}$, i.e., for all $z^{(1)} \in \mathcal{S}^{(1)} \cap \mathcal{R}$ and for all $z^{(2)} \in \mathcal{S}^{(2)} \cap \mathcal{R}$, $z^{(1)}$ is between y and $z^{(2)}$ on $\mathcal{R}$.

Remark A.2. A visibility walk is defined by a sequence of simplices $\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(\ell)}$ that satisfy $\mathcal{S}^{(\ell)} <_y \mathcal{S}^{(\ell-1)} <_y \ldots <_y \mathcal{S}^{(1)}$.

Definition A.3. A triangulation $T(\mathcal{D})$ is said to be acyclic with respect to some viewpoint y if for all $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \mathcal{S}^{(3)} \in T(\mathcal{D})$, $\mathcal{S}^{(1)} <_y \mathcal{S}^{(2)}$ and $\mathcal{S}^{(2)} <_y \mathcal{S}^{(3)}$ implies $\mathcal{S}^{(1)} <_y \mathcal{S}^{(2)}$.

Theorem A.4 (Edelsbrunner)**.** *Delaunay triangulations are acyclic with respect to a fixed viewpoint y.*

Proof. If there exists a function $\Phi^{(y)} : DT(\mathcal{D}) \to \mathbb{R}$ such that $\mathcal{S}^{(1)} <_y \mathcal{S}^{(2)} \Rightarrow \Phi^{(y)}(\mathcal{S}^{(1)}) < \Phi^{(y)}(\mathcal{S}^{(2)})$, then the proof follows trivially since it cannot be that

$$\Phi^{(y)}(\mathcal{S}^{(1)}) < \Phi^{(y)}(\mathcal{S}^{(2)}) < \ \ldots \ < \Phi^{(y)}(\mathcal{S}^{(i)}) < \Phi^{(y)}(\mathcal{S}^{(1)}).$$

Define $\Phi^{(y)}(\mathcal{S}) = \|y - c^{(\mathcal{S})}\|_2^2 - (r^{(\mathcal{S})})^2$, where $c^{(\mathcal{S})}$ and $r^{(\mathcal{S})}$ are the center and radius of the $(d-1)$-sphere circumscribing $\mathcal{S}$. Assume $\mathcal{S}^{(1)}, \mathcal{S}^{(2)} \in DT(\mathcal{D})$ such that $\mathcal{S}^{(1)} <_y \mathcal{S}^{(2)}$. Then any ray originating at y passes through $\mathcal{S}^{(1)}$ before $\mathcal{S}^{(2)}$. Since $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$ have mutually empty circumballs (with respect to each others' vertices), it follows that $\Phi^{(y)}(\mathcal{S}^{(1)}) < \Phi^{(y)}(\mathcal{S}^{(2)})$. □

A.3 Proof of Lemma 2.14

Lemma 2.14. Let $\mathcal{D}$ be in general position and $y \in CH(\mathcal{D})$, let $\{s^{(1)}, \ldots, s^{(d)}\}$ be the vertex set of a Delaunay facet. Then Algorithm 2.13 computes the vertex set of a Delaunay simplex.

Proof. Let x^* be the new vertex found by Algorithm 2.13. By Remark 2.3, x^* is in *every* open ball whose boundary passes through $s^{(1)}, \ldots, s^{(d)}$, and x for all $x \in \mathcal{H}^{(y)}(\Gamma) \cap \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(d)}, x^*\}$. So, x^* is the *only* point in $\mathcal{H}^{(y)}(\Gamma) \cap \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(d)}\}$ such that $\{s^{(1)}, \ldots, s^{(d)}, x^*\}$ could be the vertex set for a Delaunay simplex. Since $\{s^{(1)}, \ldots, s^{(d)}\}$ is the vertex set for a Delaunay facet, $y \in CH(\mathcal{D})$, and the Delaunay triangulation exists, there must be a point in $\mathcal{H}^{(y)}(\Gamma) \cap \mathcal{D} \setminus \{s^{(1)}, \ldots, s^{(d)}\}$ whose union with $\{s^{(1)}, \ldots, s^{(d)}\}$ is the vertex set for a Delaunay simplex. Therefore, $\{s^{(1)}, \ldots, s^{(d)}, x^*\}$ is the vertex set for a Delaunay simplex. □

A.4 Proof of Theorem 3.8

Theorem 3.8. If $\Pi^{(k)}$ is in general position, then Algorithm 3.7 computes the connectivity matrix for $DG(\Pi^{(k)})$.

Proof. Suppose that $\Pi^{(k)}$ is in general position, and let $\mathcal{V}^{(k,i,j)}$, $Z^{(k,i)}$, $Z^{(k,j)}$, and $\bar{Z}^{(k,i,j)}$ be as defined in Algorithm 3.7. If $\mathcal{V}^{(k,i,j)}$ contains both $Z^{(k,i)}$ and $Z^{(k,j)}$, then clearly $Z^{(k,i)}$ and $Z^{(k,j)}$ are connected in the Delaunay graph. Otherwise, since the simplex $CH(\mathcal{V}^{(k,i,j)})$ contains the midpoint $\bar{Z}^{(k,i,j)}$, at least one facet of $CH(\mathcal{V}^{(k,i,j)})$ must separate $Z^{(k,i)}$ and $Z^{(k,j)}$. Therefore, $Z^{(k,i)}$ and $Z^{(k,j)}$ cannot be connected in the Delaunay graph. □

Appendix B Code for DELAUNAYSPARSE

B.1 delsparse.f90

```
MODULE REAL_PRECISION  ! HOMPACK90 module for 64-bit arithmetic.
INTEGER, PARAMETER:: R8=SELECTED_REAL_KIND(13)
END MODULE REAL_PRECISION

MODULE DELSPARSE_MOD
! This module contains the REAL_PRECISION R8 data type for 64-bit arithmetic
! and interface blocks for the DELAUNAYSPARSES and DELAUNAYSPARSEP
! subroutines for computing the Delaunay simplices containing interpolation
! points Q in R^D given data points PTS.
USE REAL_PRECISION
PUBLIC

INTERFACE
   ! Interface for serial subroutine DELAUNAYSPARSES.
   SUBROUTINE DELAUNAYSPARSES( D, N, PTS, M, Q, SIMPS, WEIGHTS, IERR,     &
                               INTERP_IN, INTERP_OUT, EPS, EXTRAP, RNORM, &
                               IBUDGET, CHAIN, EXACT                      )
      USE REAL_PRECISION, ONLY : R8
      INTEGER, INTENT(IN) :: D, N
      REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:)
      INTEGER, INTENT(IN) :: M
      REAL(KIND=R8), INTENT(INOUT) :: Q(:,:)
      INTEGER, INTENT(OUT) :: SIMPS(:,:)
      REAL(KIND=R8), INTENT(OUT) :: WEIGHTS(:,:)
      INTEGER, INTENT(OUT) :: IERR(:)
      REAL(KIND=R8), INTENT(IN), OPTIONAL:: INTERP_IN(:,:)
      REAL(KIND=R8), INTENT(OUT), OPTIONAL :: INTERP_OUT(:,:)
      REAL(KIND=R8), INTENT(IN), OPTIONAL:: EPS, EXTRAP
      REAL(KIND=R8), INTENT(OUT), OPTIONAL :: RNORM(:)
      INTEGER, INTENT(IN), OPTIONAL :: IBUDGET
      LOGICAL, INTENT(IN), OPTIONAL :: CHAIN
      LOGICAL, INTENT(IN), OPTIONAL :: EXACT
   END SUBROUTINE DELAUNAYSPARSES

   ! Interface for parallel subroutine DELAUNAYSPARSEP.
   SUBROUTINE DELAUNAYSPARSEP( D, N, PTS, M, Q, SIMPS, WEIGHTS, IERR,     &
                               INTERP_IN, INTERP_OUT, EPS, EXTRAP, RNORM, &
                               IBUDGET, CHAIN, EXACT, PMODE               )
      USE REAL_PRECISION, ONLY : R8
      INTEGER, INTENT(IN) :: D, N
      REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:)
      INTEGER, INTENT(IN) :: M
      REAL(KIND=R8), INTENT(INOUT) :: Q(:,:)
      INTEGER, INTENT(OUT) :: SIMPS(:,:)
      REAL(KIND=R8), INTENT(OUT) :: WEIGHTS(:,:)
      INTEGER, INTENT(OUT) :: IERR(:)
      REAL(KIND=R8), INTENT(IN), OPTIONAL:: INTERP_IN(:,:)
      REAL(KIND=R8), INTENT(OUT), OPTIONAL :: INTERP_OUT(:,:)
      REAL(KIND=R8), INTENT(IN), OPTIONAL:: EPS, EXTRAP
      REAL(KIND=R8), INTENT(OUT), OPTIONAL :: RNORM(:)
      INTEGER, INTENT(IN), OPTIONAL :: IBUDGET
      LOGICAL, INTENT(IN), OPTIONAL :: CHAIN
      LOGICAL, INTENT(IN), OPTIONAL :: EXACT
      INTEGER, INTENT(IN), OPTIONAL :: PMODE
   END SUBROUTINE DELAUNAYSPARSEP

   ! Interface for SLATEC subroutine DWNNLS.
   SUBROUTINE DWNNLS( W, MDW, ME, MA, N, L, PRGOPT, X, RNORM,  &
                                MODE, IWORK, WORK )
      USE REAL_PRECISION, ONLY : R8
      INTEGER :: IWORK(*), L, MA, MDW, ME, MODE, N
      REAL(KIND=R8) :: PRGOPT(*), RNORM, W(MDW,*), WORK(*), X(*)
   END SUBROUTINE DWNNLS

END INTERFACE

END MODULE DELSPARSE_MOD

SUBROUTINE DELAUNAYSPARSES( D, N, PTS, M, Q, SIMPS, WEIGHTS, IERR, &
  INTERP_IN, INTERP_OUT, EPS, EXTRAP, RNORM, IBUDGET, CHAIN, EXACT )
! This is a serial implementation of an algorithm for efficiently performing
! interpolation in R^D via the Delaunay triangulation. The algorithm is fully
! described and analyzed in
!
! T. H. Chang, L. T. Watson,  T. C.H. Lux, B. Li, L. Xu, A. R. Butt, K. W.
! Cameron, and Y. Hong. 2018. A polynomial time algorithm for multivariate
! interpolation in arbitrary dimension via the Delaunay triangulation. In
! Proceedings of the ACMSE 2018 Conference (ACMSE '18). ACM, New York, NY,
! USA. Article 12, 8 pages.
!
!
! On input:
!
! D is the dimension of the space for PTS and Q.
!
! N is the number of data points in PTS.
!
! PTS(1:D,1:N) is a real valued matrix with N columns, each containing the
```

```
!    coordinates of a single data point in R^D.
!
! M is the number of interpolation points in Q.
!
! Q(1:D,1:M) is a real valued matrix with M columns, each containing the
!    coordinates of a single interpolation point in R^D.
!
!
! On output:
!
! PTS and Q have been rescaled and shifted. All the data points in PTS
!    are now contained in the unit hyperball in R^D, and the points in Q
!    have been shifted and scaled accordingly in relation to PTS.
!
! SIMPS(1:D+1,1:M) contains the D+1 integer indices (corresponding to columns
!    in PTS) for the D+1 vertices of the Delaunay simplex containing each
!    interpolation point in Q.
!
! WEIGHTS(1:D+1,1:M) contains the D+1 real valued weights for expressing each
!    point in Q as a convex combination of the D+1 corresponding vertices
!    in SIMPS.
!
! IERR(1:M) contains integer valued error flags associated with the
!    computation of each of the M interpolation points in Q. The error
!    codes are:
!
! 00 : Succesful interpolation.
! 01 : Succesful extrapolation (up to the allowed extrapolation distance).
! 02 : This point was outside the allowed extrapolation distance; the
!      corresponding entries in SIMPS and WEIGHTS contain zero values.
!
! 10 : The dimension D must be positive.
! 11 : Too few data points to construct a triangulation (i.e., N < D+1).
! 12 : No interpolation points given (i.e., M < 1).
! 13 : The first dimension of PTS does not agree with the dimension D.
! 14 : The second dimension of PTS does not agree with the number of points N.
! 15 : The first dimension of Q does not agree with the dimension D.
! 16 : The second dimension of Q does not agree with the number of
!      interpolation points M.
! 17 : The first dimension of the output array SIMPS does not match the number
!      of vertices needed for a D-simplex (D+1).
! 18 : The second dimension of the output array SIMPS does not match the
!      number of interpolation points M.
! 19 : The first dimension of the output array WEIGHTS does not match the
!      number of vertices for a a D-simplex (D+1).
! 20 : The second dimension of the output array WEIGHTS does not match the
!      number of interpolation points M.
! 21 : The size of the error array IERR does not match the number of
!      interpolation points M.
! 22 : INTERP_IN cannot be present without INTERP_OUT or vice versa.
! 23 : The first dimension of INTERP_IN does not match the first
!      dimension of INTERP_OUT.
! 24 : The second dimension of INTERP_IN does not match the number of
!      data points PTS.
! 25 : The second dimension of INTERP_OUT does not match the number of
!      interpolation points M.
! 26 : The budget supplied in IBUDGET does not contain a positive
!      integer.
! 27 : The extrapolation distance supplied in EXTRAP cannot be negative.
! 28 : The size of the RNORM output array does not match the number of
!      interpolation points M.
!
! 30 : Two or more points in the data set PTS are too close together with
!      respect to the working precision (EPS), which would result in a
!      numerically degenerate simplex.
! 31 : All the data points in PTS lie in some lower dimensional linear
!      manifold (up to the working precision), and no valid triangulation
!      exists.
! 40 : An error caused DELAUNAYSPARSES to terminate before this value could
!      be computed. Note: The corresponding entries in SIMPS and WEIGHTS may
!      contain garbage values.
!
! 50 : A memory allocation error occurred while allocating the work array
!      WORK.
!
! 60 : The budget was exceeded before the algorithm converged on this
!      value. If the dimension is high, try increasing IBUDGET. This
!      error can also be caused by a working precision EPS that is too
!      small for the conditioning of the problem.
!
! 61 : A value that was judged appropriate later caused LAPACK to encounter a
!      singularity. Try increasing the value of EPS.
!
! 70 : Allocation error for the extrapolation work arrays.
! 71 : The SLATEC subroutine DWNNLS failed to converge during the projection
!      of an extrapolation point onto the convex hull.
! 72 : The SLATEC subroutine DWNNLS has reported a usage error.
!
!      The errors 72, 80--83 should never occur, and likely indicate a
!      compiler bug or hardware failure.
! 80 : The LAPACK subroutine DGEQP3 has reported an illegal value.
! 81 : The LAPACK subroutine DGETRF has reported an illegal value.
```

```
! 82 : The LAPACK subroutine DGETRS has reported an illegal value.
! 83 : The LAPACK subroutine DORMQR has reported an illegal value.
!
!
! Optional arguments:
!
! INTERP_IN(1:IR,1:N) contains real valued response vectors for each of
!    the data points in PTS on input. The first dimension of INTERP_IN is
!    inferred to be the dimension of these response vectors, and the
!    second dimension must match N. If present, the response values will
!    be computed for each interpolation point in Q, and stored in INTERP_OUT,
!    which therefore must also be present. If both INTERP_IN and INTERP_OUT
!    are omitted, only the containing simplices and convex combination
!    weights are returned.
!
! INTERP_OUT(1:IR,1:M) contains real valued response vectors for each
!    interpolation point in Q on output. The first dimension of INTERP_OUT
!    must match the first dimension of INTERP_IN, and the second dimension
!    must match M. If present, the response values at each interpolation
!    point are computed as a convex combination of the response values
!    (supplied in INTERP_IN) at the vertices of a Delaunay simplex containing
!    that interpolation point.  Therefore, if INTERP_OUT is present, then
!    INTERP_IN must also be present.  If both are omitted, only the
!    simplices and convex combination weights are returned.
!
! EPS contains the real working precision for the problem on input. By default,
!    EPS is assigned \sqrt{\mu\} where \mu denotes the unit roundoff for the
!    machine. In general, any values that differ by less than EPS are judged
!    as equal, and any weights that are greater than -EPS are judged as
!    nonnegative.  EPS cannot take a value less than the default value of
!    \sqrt{\mu}. If any value less than \sqrt{\mu} is supplied, the default
!    value will be used instead automatically.
!
! EXTRAP contains the real maximum extrapolation distance (relative to the
!    diameter of PTS) on input. Interpolation at a point outside the convex
!    hull of PTS is done by projecting that point onto the convex hull, and
!    then doing normal Delaunay interpolation at that projection.
!    Interpolation at any point in Q that is more than EXTRAP * DIAMETER(PTS)
!    units outside the convex hull of PTS will not be done and an error code
!    of 2 will be returned. Note that computing the projection can be
!    expensive. Setting EXTRAP=0 will cause all extrapolation points to be
!    ignored without ever computing a projection. By default, EXTRAP=0.1
!    (extrapolate by up to 10% of the diameter of PTS).
!
! RNORM(1:M) contains the real unscaled projection (2-norm) distances from
!    any projection computations on output. If not present, these distances
!    are still computed for each extrapolation point, but are never returned.
!
! IBUDGET on input contains the integer budget for performing flips while
!    iterating toward the simplex containing each interpolation point in
!    Q. This prevents DELAUNAYSPARSES from falling into an infinite loop when
!    an inappropriate value of EPS is given with respect to the problem
!    conditioning.  By default, IBUDGET=50000. However, for extremely
!    high-dimensional problems and pathological inputs, the default value
!    may be insufficient.
!
! CHAIN is a logical input argument that determines whether a new first
!    simplex should be constructed for each interpolation point
!    (CHAIN=.FALSE.), or whether the simplex walks should be "daisy-chained."
!    By default, CHAIN=.FALSE. Setting CHAIN=.TRUE. is generally not
!    recommended, unless the size of the triangulation is relatively small
!    or the interpolation points are known to be tightly clustered.
!
! EXACT is a logical input argument that determines whether the exact
!    diameter should be computed and whether a check for duplicate data
!    points should be performed in advance. When EXACT=.FALSE., the
!    diameter of PTS is approximated by twice the distance from the
!    barycenter of PTS to the farthest point in PTS, and no check is
!    done to find the closest pair of points, which could result in hard
!    to find bugs later on. When EXACT=.TRUE., the exact diameter is
!    computed and an error is returned whenever PTS contains duplicate
!    values up to the precision EPS. By default EXACT=.TRUE., but setting
!    EXACT=.FALSE. could result in significant speedup when N is large.
!    It is strongly recommended that most users leave EXACT=.TRUE., as
!    setting EXACT=.FALSE. could result in input errors that are difficult
!    to identify. Also, the diameter approximation could be wrong by up to
!    a factor of two.
!
!
! Subroutines and functions directly referenced from BLAS are
!      DDOT, DGEMV, DNRM2, DTRSM,
! and from LAPACK are
!      DGEQP3, DGETRF, DGETRS, DORMQR.
! The SLATEC subroutine DWNNLS is directly referenced. DWNNLS and all its
! SLATEC dependencies have been slightly edited to comply with the Fortran
! 2008 standard, with all print statements and references to stderr being
! commented out. For a reference to DWNNLS, see ACM TOMS Algorithm 587
! (Hanson and Haskell).  The module REAL_PRECISION from HOMPACK90 (ACM TOMS
! Algorithm 777) is used for the real data type. The REAL_PRECISION module,
! DELAUNAYSPARSES, and DWNNLS and its dependencies comply with the Fortran
! 2008 standard.
!
```

```
! Primary Author: Tyler H. Chang
! Last Update: March, 2020
!
USE REAL_PRECISION, ONLY : R8
IMPLICIT NONE

! Input arguments.
INTEGER, INTENT(IN) :: D, N
REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:) ! Rescaled on output.
INTEGER, INTENT(IN) :: M
REAL(KIND=R8), INTENT(INOUT) :: Q(:,:) ! Rescaled on output.
! Output arguments.
INTEGER, INTENT(OUT) :: SIMPS(:,:)
REAL(KIND=R8), INTENT(OUT) :: WEIGHTS(:,:)
INTEGER, INTENT(OUT) :: IERR(:)
! Optional arguments.
REAL(KIND=R8), INTENT(IN), OPTIONAL:: INTERP_IN(:,:)
REAL(KIND=R8), INTENT(OUT), OPTIONAL :: INTERP_OUT(:,:)
REAL(KIND=R8), INTENT(IN), OPTIONAL:: EPS, EXTRAP
REAL(KIND=R8), INTENT(OUT), OPTIONAL :: RNORM(:)
INTEGER, INTENT(IN), OPTIONAL :: IBUDGET
LOGICAL, INTENT(IN), OPTIONAL :: CHAIN
LOGICAL, INTENT(IN), OPTIONAL :: EXACT

! Local copies of optional input arguments.
REAL(KIND=R8) :: EPSL, EXTRAPL
INTEGER :: IBUDGETL
LOGICAL :: CHAINL, EXACTL

! Local variables.
INTEGER :: I, J, K ! Loop iteration variables.
INTEGER :: IEXTRAPS ! Extrapolation budget.
INTEGER :: ITMP, JTMP ! Temporary variables for swapping, looping, etc.
INTEGER :: LWORK ! Size of the work array.
INTEGER :: MI ! Index of current interpolation point.
REAL(KIND=R8) :: CURRRAD ! Radius of the current circumsphere.
REAL(KIND=R8) :: MINRAD ! Minimum circumsphere radius observed.
REAL(KIND=R8) :: PTS_DIAM ! Scaled diameter of data set.
REAL(KIND=R8) :: PTS_SCALE ! Data scaling factor.
REAL(KIND=R8) :: RNORML ! Euclidean norm of the projection residual.
REAL(KIND=R8) :: SIDE1, SIDE2 ! Signs (+/-1) denoting sides of a facet.

! Local arrays, requiring O(d^2) additional memory.
INTEGER :: IPIV(D) ! Pivot indices.
INTEGER :: SEED(D+1) ! Copy of the SEED simplex. Only used if CHAIN = .TRUE.
REAL(KIND=R8) :: AT(D,D) ! The transpose of A, the linear coefficient matrix.
REAL(KIND=R8) :: B(D) ! The RHS of a linear system.
REAL(KIND=R8) :: CENTER(D) ! The circumcenter of a simplex.
REAL(KIND=R8) :: LQ(D,D) ! Holds LU or QR factorization of AT.
REAL(KIND=R8) :: PLANE(D+1) ! The hyperplane containing a facet.
REAL(KIND=R8) :: PRGOPT_DWNNLS(1) ! Options array for DWNNLS.
REAL(KIND=R8) :: PROJ(D) ! The projection of the current iterate.
REAL(KIND=R8) :: TAU(D) ! Householder reflector constants.
REAL(KIND=R8) :: X(D) ! The solution to a linear system.

! Extrapolation work arrays are only allocated if DWNNLS is called.
INTEGER, ALLOCATABLE :: IWORK_DWNNLS(:) ! Only for DWNNLS.
REAL(KIND=R8), ALLOCATABLE :: W_DWNNLS(:,:) ! Only for DWNNLS.
REAL(KIND=R8), ALLOCATABLE :: WORK(:) ! Allocated with size LWORK.
REAL(KIND=R8), ALLOCATABLE :: WORK_DWNNLS(:) ! Only for DWNNLS.
REAL(KIND=R8), ALLOCATABLE :: X_DWNNLS(:) ! Only for DWNNLS.

! External functions and subroutines.
REAL(KIND=R8), EXTERNAL :: DDOT  ! Inner product (BLAS).
REAL(KIND=R8), EXTERNAL :: DNRM2 ! Euclidean norm (BLAS).
EXTERNAL :: DGEMV  ! General matrix vector multiply (BLAS)
EXTERNAL :: DGEQP3 ! Perform a QR factorization with column pivoting (LAPACK).
EXTERNAL :: DGETRF ! Perform a LU factorization with partial pivoting (LAPACK).
EXTERNAL :: DGETRS ! Use the output of DGETRF to solve a linear system (LAPACK).
EXTERNAL :: DORMQR ! Apply householder reflectors to a matrix (LAPACK).
EXTERNAL :: DTRSM  ! Perform a triangular solve (BLAS).
EXTERNAL :: DWNNLS ! Solve an inequality constrained least squares problem
                   ! (SLATEC).

! Check for input size and dimension errors.
IF (D < 1) THEN ! The dimension must satisfy D > 0.
   IERR(:) = 10; RETURN; END IF
IF (N < D+1) THEN ! Must have at least D+1 data points.
   IERR(:) = 11; RETURN; END IF
IF (M < 1) THEN ! Must have at least one interpolation point.
   IERR(:) = 12; RETURN; END IF
IF (SIZE(PTS,1) .NE. D) THEN ! Dimension of PTS array should match.
   IERR(:) = 13; RETURN; END IF
IF (SIZE(PTS,2) .NE. N) THEN ! Number of data points should match.
   IERR(:) = 14; RETURN; END IF
IF (SIZE(Q,1) .NE. D) THEN ! Dimension of Q should match.
   IERR(:) = 15; RETURN; END IF
IF (SIZE(Q,2) .NE. M) THEN ! Number of interpolation points should match.
   IERR(:) = 16; RETURN; END IF
IF (SIZE(SIMPS,1) .NE. D+1) THEN ! Need space for D+1 vertices per simplex.
   IERR(:) = 17; RETURN; END IF
IF (SIZE(SIMPS,2) .NE. M) THEN  ! There will be M output simplices.
```

```
   IERR(:) = 18; RETURN; END IF
IF (SIZE(WEIGHTS,1) .NE. D+1) THEN ! There will be D+1 weights per simplex.
   IERR(:) = 19; RETURN; END IF
IF (SIZE(WEIGHTS,2) .NE. M) THEN ! One vector of weights per simplex.
   IERR(:) = 20; RETURN; END IF
IF (SIZE(IERR) .NE. M) THEN ! An error flag for each interpolation point.
   IERR(:) = 21; RETURN; END IF

! Check for optional arguments.
IF (PRESENT(INTERP_IN) .NEQV. PRESENT(INTERP_OUT)) THEN
   IERR(:) = 22; RETURN; END IF
IF (PRESENT(INTERP_IN)) THEN ! Sizes must agree.
   IF (SIZE(INTERP_IN,1) .NE. SIZE(INTERP_OUT,1)) THEN
      IERR(:) = 23 ; RETURN; END IF
   IF(SIZE(INTERP_IN,2) .NE. N) THEN
      IERR(:) = 24; RETURN; END IF
   IF (SIZE(INTERP_OUT,2) .NE. M) THEN
      IERR(:) = 25; RETURN; END IF
   INTERP_OUT(:,:) = 0.0_R8 ! Initialize output to zeros.
END IF
EPSL = SQRT(EPSILON(0.0_R8)) ! Get the machine unit roundoff constant.
IF (PRESENT(EPS)) THEN
   IF (EPSL < EPS) THEN ! If the given precision is too small, ignore it.
      EPSL = EPS
   END IF
END IF
IF (PRESENT(IBUDGET)) THEN
   IBUDGETL = IBUDGET ! Use the given budget if present.
   IF (IBUDGETL < 1) THEN
      IERR(:) = 26; RETURN; END IF
ELSE
   IBUDGETL = 50000 ! Default value for budget.
END IF
IF (PRESENT(EXTRAP)) THEN
   EXTRAPL = EXTRAP
   IF (EXTRAPL < 0) THEN ! Check that the extrapolation distance is legal.
      IERR(:) = 27; RETURN; END IF
ELSE
   EXTRAPL = 0.1_R8 ! Default extrapolation distance (for normalized points).
END IF
IF (PRESENT(RNORM)) THEN
   IF (SIZE(RNORM,1) .NE. M) THEN ! The length of the array must match.
      IERR(:) = 28; RETURN; END IF
   RNORM(:) = 0.0_R8 ! Initialize output to zeros.
END IF
IF (PRESENT(CHAIN)) THEN
   CHAINL = CHAIN ! Turn chaining on, if necessarry.
   SEED(:) = 0 ! Initialize SEED in case it is needed.
ELSE
   CHAINL = .FALSE.
END IF
IF (PRESENT(EXACT)) THEN
   EXACTL = EXACT ! Set error checking and exact diameter computations.
ELSE
   EXACTL = .TRUE.
END IF

! Scale and center the data points and interpolation points.
CALL RESCALE(MINRAD, PTS_DIAM, PTS_SCALE)
IF (MINRAD < EPSL) THEN ! Check for degeneracies in points spacing.
   IERR(:) = 30; RETURN; END IF

! Query DGEQP3 for optimal work array size (LWORK).
LWORK = -1
CALL DGEQP3(D,D,LQ,D,IPIV,TAU,B,LWORK,IERR(1))
LWORK = INT(B(1)) ! Compute the optimal work array size.
ALLOCATE(WORK(LWORK), STAT=I) ! Allocate WORK to size LWORK.
IF (I .NE. 0) THEN ! Check for memory allocation errors.
   IERR(:) = 50; RETURN; END IF

! Initialize all error codes to "TBD" values.
IERR(:) = 40

! Outer loop over all interpolation points (in Q).
OUTER : DO MI = 1, M

   ! Check if this interpolation point was already found.
   IF (IERR(MI) .EQ. 0) CYCLE OUTER

   ! Initialize the projection and reset the residual.
   PROJ(:) = Q(:,MI)
   RNORML = 0.0_R8

   ! Check if extrapolation is enabled.
   IF (EXTRAPL < EPSL) THEN
      IEXTRAPS = -1 ! If not, set the extrapolation budget negative.
   ELSE
      IEXTRAPS = 1 ! Allow for exactly one projection for this point.
   END IF

   ! If there is no useable seed or if chaining is turned off, then make a new
   ! simplex.
```

```
IF( (.NOT. CHAINL) .OR. SEED(1) .EQ. 0) THEN
   CALL MAKEFIRSTSIMP()
   IF(IERR(MI) .NE. 0) CYCLE OUTER
! Otherwise, use the seed.
ELSE
   ! Copy the seed to the current simplex.
   SIMPS(:,MI) = SEED(:)
   ! Rebuild the linear system.
   DO J=1,D
      AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
      B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
   END DO
END IF

! Inner loop searching for a simplex containing the point Q(:,MI).
INNER : DO K = 1, IBUDGETL

   ! If chaining is on, save each good simplex as the next seed.
   IF (CHAINL) SEED(:) = SIMPS(:,MI)

   ! Check if the current simplex contains Q(:,MI).
   IF (PTINSIMP()) EXIT INNER
   IF (IERR(MI) .NE. 0) CYCLE OUTER ! Check for an error flag.

   ! Swap out the least weighted vertex, but save its value in case it
   ! needs to be restored later.
   JTMP = MINLOC(WEIGHTS(1:D+1,MI), DIM=1)
   ITMP = SIMPS(JTMP,MI)
   SIMPS(JTMP,MI) = SIMPS(D+1,MI)

   ! If the least weighted vertex (index JTMP) is not the first vertex,
   ! then just drop row (JTMP-1) from the linear system (corresponding
   ! to column (JTMP-1) of A^T).
   IF(JTMP .NE. 1) THEN
      AT(:,JTMP-1) = AT(:,D); B(JTMP-1) = B(D)
   ! However, if JTMP = 1, then both A^T and B must be reconstructed.
   ELSE
      DO J=1,D
         AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
         B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
      END DO
   END IF

   ! Compute the next simplex (do one flip).
   CALL MAKESIMPLEX()
   IF (IERR(MI) .NE. 0) CYCLE OUTER

! If no vertex was found, then this is an extrapolation point.
IF (SIMPS(D+1,MI) .EQ. 0) THEN

   ! If extrapolation is not allowed (EXTRAP=0), do not proceed.
   IF (IEXTRAPS < 0) THEN
      SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
      ! Set the error flag and skip this point.
      IERR(MI) = 2; CYCLE OUTER

   ! If extrapolation is allowed (EXTRAP>0), check the budget.
   ELSE IF (IEXTRAPS .EQ. 0) THEN
      ! A second projection has been attempted. This code is rarely
      ! called, except in extreme cases involving nearly singular
      ! simplices near the convex hull of P.

      ! Swap the weights to match the simplex indices, and zero the
      ! most negative weight.
      WEIGHTS(JTMP,MI) = WEIGHTS(D+1,MI)
      WEIGHTS(D+1,MI) = 0.0_R8
      ! Loop through all the remaining facets from which Q(:,MI) is
      ! visible, and attempt to flip across each one.
      DO WHILE (SIMPS(D+1,MI) .EQ. 0)
         ! Restore the previous simplex and linear system.
         SIMPS(D+1,MI) = ITMP
         AT(:,D) = PTS(:,ITMP) - PTS(:,SIMPS(1,MI))
         B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
         ! Find the next most negative weight.
         JTMP = MINLOC(WEIGHTS(1:D+1,MI), DIM=1)
         ! Check if WEIGHTS(JTMP,MI) .GE. 0.
         IF (WEIGHTS(JTMP,MI) .GE. -EPSL) THEN
            ! There is no other direction to flip, so Q(:,MI) must be
            ! within EPSL of the current simplex.
            ! Project Q(:,MI) onto the current simplex.

            ! Since at least one projection has already been done,
            ! the work arrays have already been allocated.
            PRGOPT_DWNNLS(1) = 1.0_R8
            IWORK_DWNNLS(1) = 6*D + 6
            IWORK_DWNNLS(2) = 2*D + 2
            ! Set equality constraint.
            W_DWNNLS(1,1:D+2) = 1.0_R8
            ! Populate LS coefficient matrix and RHS.
            FORALL (I=1:D+1) W_DWNNLS(2:D+1,I) = PTS(:,SIMPS(I,MI))
            W_DWNNLS(2:D+1,D+2) = PROJ(:)
            ! Project onto the current simplex.
```

```
         CALL DWNNLS(W_DWNNLS, D+1, 1, D, D+1, 0, PRGOPT_DWNNLS, &
            WEIGHTS(:,MI), WORK(1), IERR(MI), IWORK_DWNNLS,      &
            WORK_DWNNLS)
         IF (IERR(MI) .EQ. 1) THEN ! Failure to converge.
            IERR(MI) = 71; CYCLE OUTER
         ELSE IF (IERR(MI) .EQ. 2) THEN ! Illegal input detected.
            IERR(MI) = 72; CYCLE OUTER
         END IF
         ! A solution has been found; return it.
         EXIT INNER
      END IF
      ! Otherwise, swap the vertices.
      ITMP = SIMPS(JTMP,MI)
      SIMPS(JTMP,MI) = SIMPS(D+1,MI)
      ! Swap the weights to match, and zero the most negative weight.
      WEIGHTS(JTMP,MI) = WEIGHTS(D+1,MI)
      WEIGHTS(D+1,MI) = 0.0_R8
      ! If the least weighted vertex (index JTMP) is not the first
      ! vertex, then just drop row (JTMP-1) from the linear system
      ! (corresponding to column (JTMP-1) of A^T).
      IF (JTMP .NE. 1) THEN
         AT(:,JTMP-1) = AT(:,D); B(JTMP-1) = B(D)
      ! However, if JTMP=1, then both A^T and B must be reconstructed.
      ELSE
         DO J=1,D
            AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
            B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
         END DO
      END IF
      ! Compute another simplex (try to flip again).
      CALL MAKESIMPLEX(); IF (IERR(MI) .NE. 0) CYCLE OUTER
   END DO
   ! If the loop terminates, then a good direction was found.
   ! Resume the visibility walk as normal.
   CYCLE INNER
END IF

! Otherwise, project the extrapolation point onto the convex hull.
CALL PROJECT()
IF (IERR(MI) .NE. 0) CYCLE OUTER

! Check the value of RNORML for over-extrapolation.
IF (RNORML > EXTRAPL * PTS_DIAM) THEN
   SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
   ! If present, record the unscaled RNORM output.
   IF (PRESENT(RNORM)) RNORM(MI) = RNORML*PTS_SCALE
      ! Set the error flag and skip this point.
      IERR(MI) = 2; CYCLE OUTER
   END IF

   ! Otherwise, restore the previous simplex and continue with the
   ! projected value.
   SIMPS(D+1,MI) = ITMP
   AT(:,D) = PTS(:,ITMP) - PTS(:,SIMPS(1,MI))
   B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
   IEXTRAPS = IEXTRAPS - 1 ! Decrement the budget.
  END IF

! End of inner loop for finding each interpolation point.
END DO INNER

! Check for budget violation conditions.
IF (K > IBUDGETL) THEN
   SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
   ! Set the error flag and skip this point.
   IERR(MI) = 60; CYCLE OUTER
END IF

! If the residual is nonzero, set the extrapolation flag.
IF (RNORML > EPSL) IERR(MI) = 1

! If present, record the RNORM output.
IF (PRESENT(RNORM)) RNORM(MI) = RNORML*PTS_SCALE

END DO OUTER  ! End of outer loop over all interpolation points.

! If INTERP_IN and INTERP_OUT are present, compute all values f(q).
IF (PRESENT(INTERP_IN)) THEN
   ! Loop over all interpolation points.
   DO MI = 1, M
      ! Check for errors.
      IF (IERR(MI) .LE. 1) THEN
         ! Compute the weighted sum of vertex response values.
         DO K = 1, D+1
            INTERP_OUT(:,MI) = INTERP_OUT(:,MI) &
             + INTERP_IN(:,SIMPS(K,MI)) * WEIGHTS(K,MI)
         END DO
      END IF
   END DO
END IF

! Free dynamic work arrays.
```

```
DEALLOCATE(WORK)
IF (ALLOCATED(IWORK_DWNNLS)) DEALLOCATE(IWORK_DWNNLS)
IF (ALLOCATED(WORK_DWNNLS))  DEALLOCATE(WORK_DWNNLS)
IF (ALLOCATED(W_DWNNLS))     DEALLOCATE(W_DWNNLS)
IF (ALLOCATED(X_DWNNLS))     DEALLOCATE(X_DWNNLS)

RETURN

CONTAINS    ! Internal subroutines and functions.

SUBROUTINE MAKEFIRSTSIMP()
! Iteratively construct the first simplex by choosing points that
! minimize the radius of the smallest circumball. Let P_1, P_2, ..., P_K
! denote the current set of vertices for the simplex. Let P* denote the
! candidate vertex to be added to the simplex. Let CENTER denote the
! circumcenter of the simplex.  Then
!
! X = CENTER - P_1
!
! is given by the minimum norm solution to the underdetermined linear system
!
! A X = B, where
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_K - P_1, P* - P_1 ] and
! B = [ <A_{1.},A_{1.}>/2, <A_{2.},A_{2.}>/2, ..., <A_{K.},A_{K.}>/2 ]^T.
!
! Then the radius of the smallest circumsphere is CURRRAD = \| X \|,
! and the next vertex is given by P_{K+1} = argmin_{P*} CURRRAD, where P*
! ranges over points in PTS that are not already a vertex of the simplex.
!
! On output, this subroutine fully populates the matrix A^T and vector B,
! and fills SIMPS(:,MI) with the indices of a valid Delaunay simplex.

! Find the first point, i.e., the closest point to Q(:,MI).
SIMPS(:,MI) = 0
MINRAD = HUGE(0.0_R8)
DO I = 1, N
   ! Check the distance to Q(:,MI).
   CURRRAD = DNRM2(D, PTS(:,I) - PROJ(:), 1)
   IF (CURRRAD < MINRAD) THEN; MINRAD = CURRRAD; SIMPS(1,MI) = I; END IF
END DO
! Find the second point, i.e., the closest point to PTS(:,SIMPS(1,MI)).
MINRAD = HUGE(0.0_R8)
DO I = 1, N
   ! Skip repeated vertices.
   IF (I .EQ. SIMPS(1,MI)) CYCLE
   ! Check the diameter of the resulting circumsphere.
   CURRRAD = DNRM2(D, PTS(:,I)-PTS(:,SIMPS(1,MI)), 1)
   IF (CURRRAD < MINRAD) THEN; MINRAD = CURRRAD; SIMPS(2,MI) = I; END IF
END DO
IF (MINRAD < EPSL) THEN ! Check for degeneracies in points spacing.
   IERR(MI) = 30; RETURN; END IF
! Set up the first row of the linear system.
AT(:,1) = PTS(:,SIMPS(2,MI)) - PTS(:,SIMPS(1,MI))
B(1) = DDOT(D, AT(:,1), 1, AT(:,1), 1) / 2.0_R8
! Loop to collect the remaining D-1 vertices for the first simplex.
DO I = 2, D
   ! For numerical stability, refactor A^T P = Q R for the next iteration.
   LQ(:,1:I-1) = AT(:,1:I-1)
   CALL DGEQP3(D, I-1, LQ, D, IPIV, TAU, WORK, LWORK, IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 80; RETURN
   END IF
   ! Set the RHS to P^T B.
   FORALL (ITMP = 1:I-1) X(ITMP) = B(IPIV(ITMP))
   ! Solve R^T Q^T X = P^T B for Q^T X, and save for later.
   CALL DTRSM('L', 'U', 'T', 'N', I-1, 1, 1.0_R8, LQ, D, X, D)
   ! Make a copy for computing the current center.
   CENTER(1:I-1) = X(1:I-1)
   CENTER(I:D) = 0.0_R8
   ! Apply Q from the left.
   CALL DORMQR('L', 'N', D, 1, I-1, LQ, D, TAU, CENTER, D, WORK, &
     LWORK, IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 83; RETURN
   END IF
   CENTER = CENTER + PTS(:,SIMPS(1,MI))
   ! Re-initialize the radius for each iteration.
   MINRAD = HUGE(0.0_R8)
   ! Check each point P* in PTS.
   DO J = 1, N
      ! Check that this point is not already in the simplex.
      IF (ANY(SIMPS(:,MI) .EQ. J)) CYCLE
      ! If PTS(:,J) is more than twice MINRAD from CENTER, do a quick skip.
      IF (DNRM2(D, CENTER - PTS(:,J), 1) > 2.0_R8 * MINRAD) CYCLE
      ! Perform a rank-1 update to the current QR factorization of A^T by
      ! rotating PTS(:,I) - PTS(:,SIMPS(1,MI)) by Q^T and storing in the
      ! final column of R.
      LQ(:,I) = PTS(:,J) - PTS(:,SIMPS(1,MI))
      CALL DORMQR('L', 'T', D, 1, I-1, LQ(:,1:I-1), D, TAU, LQ(:,I), D, &
        WORK, LWORK, IERR(MI))
      IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
```

```
         IERR(MI) = 83; RETURN
      END IF
      ! Implicitly apply the next Householder reflector.
      LQ(I,I) = DNRM2(D+1-I, LQ(I:D,I), 1)
      IF (LQ(I,I) < EPSL) THEN ! A is rank-deficient.
         CYCLE ! If rank-deficient, skip this point.
      END IF
      ! Update the current radius by \| Q^T X \| = \| X \|.
      WORK(1:I-1) = (LQ(1:I-1,I) / 2.0_R8) - X(1:I-1)
      WORK(I) = LQ(I,I) / 2.0_R8
      X(I) = DDOT(I, LQ(1:I,I), 1, WORK(1:I), 1) / LQ(I,I)
      CURRRAD = DNRM2(I, X(1:I), 1)
      ! Compare the last component of Q^T X to the current minimum.
      IF (CURRRAD < MINRAD) THEN; MINRAD = CURRRAD; SIMPS(I+1,MI) = J; END IF
   END DO
   ! Check that a point was found. If not, then all the points must lie in a
   ! lower dimensional linear manifold (error case).
   IF (SIMPS(I+1,MI) .EQ. 0) THEN; IERR(MI) = 31; RETURN; END IF
   ! If all operations were successful, add the best P* to the linear system.
   AT(:,I) = PTS(:,SIMPS(I+1,MI)) - PTS(:,SIMPS(1,MI))
   B(I) = DDOT(D, AT(:,I), 1, AT(:,I), 1) / 2.0_R8
END DO
IERR(MI) = 0 ! Set error flag to 'success' for a normal return.
RETURN
END SUBROUTINE MAKEFIRSTSIMP

SUBROUTINE MAKESIMPLEX()
! Given a Delaunay facet F whose containing hyperplane does not contain
! Q(:,MI), complete the simplex by adding a point from PTS on the same 'side'
! of F as Q(:,MI). Assume SIMPS(1:D,MI) contains the vertex indices of F
! (corresponding to data points P_1, P_2, ..., P_D in PTS), and assume the
! matrix A(1:D-1,:)^T and vector B(1:D-1) are filled appropriately (similarly
! as in MAKEFIRSTSIMP()). Then for any P* (not in the hyperplane containing
! F) in PTS, let CENTER denote the circumcenter of the simplex with vertices
! P_1, P_2, ..., P_D, P*. Then
!
! X = CENTER - P_1
!
! is given by the solution to the nonsingular linear system
!
! A X = B where
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1, P* - P_1 ] and
! B = [ <A_{1.},A_{1.}>/2, <A_{2.},A_{2.}>/2, ..., <A_{D.},A_{D.}>/2 ]^T.
!
! Then CENTER = X + P_1 and RADIUS = \| X \|.  P_{D+1} will be given by the
! candidate P* that satisfies both of the following:
!
! 1) Let PLANE denote the hyperplane containing F. Then P_{D+1} and Q(:,MI)
! must be on the same side of PLANE.
!
! 2) The circumball about CENTER must not contain any points in PTS in its
! interior (Delaunay property).
!
! The above are necessary and sufficient conditions for flipping the
! Delaunay simplex, given that F is indeed a Delaunay facet.
!
! On input, SIMPS(1:D,MI) should contain the vertex indices (column indices
! from PTS) of the facet F.  Upon output, SIMPS(:,MI) will contain the vertex
! indices of a Delaunay simplex closer to Q(:,MI).  Also, the matrix A^T and
! vector B will be updated accordingly. If SIMPS(D+1,MI)=0, then there were
! no points in PTS on the appropriate side of F, meaning that Q(:,MI) is an
! extrapolation point (not a convex combination of points in PTS).

! Compute the hyperplane PLANE.
CALL MAKEPLANE()
IF(IERR(MI) .NE. 0) RETURN ! Check for errors.
! Compute the sign for the side of PLANE containing Q(:,MI).
SIDE1 =  DDOT(D,PLANE(1:D),1,PROJ(:),1) - PLANE(D+1)
SIDE1 = SIGN(1.0_R8,SIDE1)
! Initialize the center, radius, and simplex.
SIMPS(D+1,MI) = 0
CENTER(:) = 0.0_R8
MINRAD = HUGE(0.0_R8)
! If D=1, just check for the closest point on SIDE1 of PTS(:,SIMPS(1,MI)).
IF (D .EQ. 1) THEN
   ! Loop through all points P* in PTS.
   DO I = 1, N
      ! Check that P* is on the appropriate halfspace.
      SIDE2 = (PTS(1,I) - PLANE(2)) * SIDE1
      IF (SIDE2 < EPSL .OR. SIMPS(1,MI) .EQ. I) CYCLE
      ! Check that P* is closer than the current solution.
      IF (SIDE2 > MINRAD) CYCLE
      ! Update the minimum distance and save the index I.
      MINRAD = SIDE2
      SIMPS(2,MI) = I
   END DO
   IERR(MI) = 0 ! Reset the error flag to 'success' code.
   ! Check for extrapolation condition.
   IF(SIMPS(2,MI) .EQ. 0) RETURN
   ! Add new point to the linear system.
   AT(1,1) = PTS(1,SIMPS(2,MI)) - PTS(1,SIMPS(1,MI))
```

```
   B(1) = (AT(1,1) ** 2.0_R8) / 2.0_R8
   RETURN
END IF
! Set the RHS to P^T B.
FORALL (ITMP = 1:D-1) X(ITMP) = B(IPIV(ITMP))
! Solve R^T Q^T X = P^T B for Q^T X.
CALL DTRSM('L', 'U', 'T', 'N', D-1, 1, 1.0_R8, LQ, D, X, D)
! Loop through all points P* in PTS.
DO I = 1, N
   ! Check that P* is inside the current ball.
   IF (DNRM2(D, PTS(:,I) - CENTER(:), 1) > MINRAD) CYCLE ! If not, skip.
   ! Check that P* is on the appropriate halfspace.
   SIDE2 = DDOT(D,PLANE(1:D),1,PTS(:,I),1) - PLANE(D+1)
   IF (SIDE1*SIDE2 < EPSL .OR. ANY(SIMPS(:,MI) .EQ. I)) CYCLE ! If not, skip.
   ! Perform a rank-1 update to the current QR factorization of A^T by
   ! rotating PTS(:,I) - PTS(:,SIMPS(1,MI) by Q^T and storing in the
   ! final column of R.
   LQ(:,D) = PTS(:,I) - PTS(:,SIMPS(1,MI))
   CALL DORMQR('L', 'T', D, 1, D-1, LQ(:,1:D-1), D, TAU, LQ(:,D), D, WORK, &
     LWORK, IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 83; RETURN
   END IF
   ! Update the last element of Q^T X.
   WORK(1:D-1) = (LQ(1:D-1,D) / 2.0_R8) - X(1:D-1)
   WORK(D) = LQ(D,D) / 2.0_R8
   CENTER(1:D-1) = X(1:D-1)
   CENTER(D) = DDOT(D, LQ(:,D), 1, WORK(1:D), 1) / LQ(D,D)
   ! Get the center by applying Q to the solution.
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, CENTER, D, WORK, LWORK, &
      IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 83; RETURN
   END IF
   ! Update the new radius, center, and simplex.
   MINRAD = DNRM2(D, CENTER, 1)
   CENTER(:) = CENTER(:) + PTS(:,SIMPS(1,MI))
   SIMPS(D+1,MI) = I
END DO
IERR(MI) = 0 ! Reset the error flag to 'success' code.
! Check for extrapolation condition.
IF(SIMPS(D+1,MI) .EQ. 0) RETURN
! Add new point to the linear system.
AT(:,D) = PTS(:,SIMPS(D+1,MI)) - PTS(:,SIMPS(1,MI))
B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
RETURN
END SUBROUTINE MAKESIMPLEX

SUBROUTINE MAKEPLANE()
! Construct a hyperplane c^T x = \alpha containing the first D vertices indexed
! in SIMPS(:,MI). The plane is determined by its normal vector c and \alpha.
! Let P_1, P_2, ..., P_D be the vertices indexed in SIMPS(1:D,MI). A normal
! vector is any nonzero vector in ker A, where the matrix
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1 ].
!
! Since rank A = D-1, dim ker A = 1, and ker A can be found from a QR
! factorization of A^T:  A^T P = QR, where P permutes the columns of A^T.
! Then the last column of Q is orthogonal to the range of A^T, and in ker A.
!
! Upon output, PLANE(1:D) contains the normal vector c and PLANE(D+1)
! contains \alpha defining the plane. Also, LQ, IPIV, and TAU define a QR
! factorizaton of the first D-1 columns of A^T.

IF (D > 1) THEN ! Check that D-1 > 0, otherwise the plane is trivial.
   ! Compute the QR factorization.
   IPIV=0
   LQ = AT
   CALL DGEQP3(D, D-1, LQ, D, IPIV, TAU, WORK, LWORK, IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 80; RETURN
   END IF
   ! The nullspace is given by the last column of Q.
   PLANE(1:D-1) = 0.0_R8
   PLANE(D) = 1.0_R8
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, PLANE, D, WORK, &
    LWORK, IERR(MI))
   IF(IERR(MI) < 0) THEN ! LAPACK illegal input error.
      IERR(MI) = 83; RETURN
   END IF
   ! Calculate the constant \alpha defining the plane.
   PLANE(D+1) = DDOT(D,PLANE(1:D),1,PTS(:,SIMPS(1,MI)),1)
ELSE ! Special case where D=1.
   PLANE(1) = 1.0_R8
   PLANE(2) = PTS(1,SIMPS(1,MI))
END IF
RETURN
END SUBROUTINE MAKEPLANE

FUNCTION PTINSIMP() RESULT(TF)
! Determine if any interpolation points are in the current simplex, whose
! vertices P_1, P_2, ..., P_{D+1} are indexed by SIMPS(:,MI). These
```

```
! vertices determine a positive cone with generators V_I = P_{I+1} - P_1,
! I = 1, ..., D. For each interpolation point Q* in Q, Q* - P_1 can be
! expressed as a unique linear combination of the V_I.  If all these linear
! weights are nonnegative and sum to less than or equal to 1.0, then Q* is
! in the simplex with vertices {P_I}_{I=1}^{D+1}.
!
! If any interpolation points in Q are contained in the simplex whose
! vertices are indexed by SIMPS(:,MI), then those points are marked as solved
! and the values of SIMPS and WEIGHTS are updated appropriately. On output,
! WEIGHTS(:,MI) contains the affine weights for producing Q(:,MI) as an
! affine combination of the points in PTS indexed by SIMPS(:,MI). If these
! weights are nonnegative, then PTINSIMP() returns TRUE.

! Initialize the return value and local variables.
LOGICAL :: TF ! True/False value.
TF = .FALSE.

! Compute the LU factorization of the matrix A^T, whose columns are
! P_{I+1} - P_1.
LQ = AT
CALL DGETRF(D, D, LQ, D, IPIV, IERR(MI))
IF (IERR(MI) < 0) THEN ! LAPACK illegal input.
   IERR(MI) = 81; RETURN
ELSE IF (IERR(MI) > 0) THEN ! Rank-deficiency detected.
   IERR(MI) = 61; RETURN
END IF
! Solve A^T w = WORK to get the affine weights for Q(:,MI) or its projection.
WORK(1:D) = PROJ(:) - PTS(:,SIMPS(1,MI))
CALL DGETRS('N', D, 1, LQ, D, IPIV, WORK(1:D), D, IERR(MI))
IF (IERR(MI) < 0) THEN ! LAPACK illegal input.
   IERR(MI) = 82; RETURN
END IF
WEIGHTS(2:D+1,MI) = WORK(1:D)
WEIGHTS(1,MI) = 1.0_R8 - SUM(WEIGHTS(2:D+1,MI))
! Check if the weights for Q(:,MI) are nonnegative.
IF (ALL(WEIGHTS(:,MI) .GE. -EPSL)) TF = .TRUE.

! Compute the affine weights for the rest of the interpolation points.
DO I = MI+1, M
   ! Check that no solution has already been found.
   IF (IERR(I) .NE. 40) CYCLE
   ! Solve A^T w = WORK to get the affine weights for Q(:,I).
   WORK(2:D+1) = Q(:,I) - PTS(:,SIMPS(1,MI))
   CALL DGETRS('N', D, 1, LQ, D, IPIV, WORK(2:D+1), D, ITMP)
   IF (ITMP < 0) CYCLE ! Illegal input error that should never occurr.
   ! Check if the weights define a convex combination.
   WORK(1) = 1.0_R8 - SUM(WORK(2:D+1))
   IF (ALL(WORK(1:D+1) .GE. -EPSL)) THEN
      ! Copy the simplex indices and weights then flag as complete.
      SIMPS(:,I) = SIMPS(:,MI)
      WEIGHTS(:,I) = WORK(1:D+1)
      IERR(I) = 0
   END IF
END DO
RETURN
END FUNCTION PTINSIMP

SUBROUTINE PROJECT()
! Project a point outside the convex hull of the point set onto the convex hull
! by solving an inequality constrained least squares problem. The solution to
! the least squares problem gives the projection as a convex combination of the
! data points. The projection can then be computed by performing a matrix
! vector multiplication.

! Allocate work arrays.
IF (.NOT. ALLOCATED(IWORK_DWNNLS)) THEN
   ALLOCATE(IWORK_DWNNLS(D+1+N), STAT=IERR(MI))
   IF(IERR(MI) .NE. 0) THEN; IERR(MI) = 70; RETURN; END IF
END IF
IF (.NOT. ALLOCATED(WORK_DWNNLS)) THEN
   ALLOCATE(WORK_DWNNLS(D+1+N*5), STAT=IERR(MI))
   IF(IERR(MI) .NE. 0) THEN; IERR(MI) = 70; RETURN; END IF
END IF
IF (.NOT. ALLOCATED(W_DWNNLS)) THEN
   ALLOCATE(W_DWNNLS(D+1,N+1), STAT=IERR(MI))
   IF(IERR(MI) .NE. 0) THEN; IERR(MI) = 70; RETURN; END IF
END IF
IF (.NOT. ALLOCATED(X_DWNNLS)) THEN
   ALLOCATE(X_DWNNLS(N), STAT=IERR(MI))
   IF(IERR(MI) .NE. 0) THEN; IERR(MI) = 70; RETURN; END IF
END IF

! Initialize work array and settings values.
PRGOPT_DWNNLS(1) = 1.0_R8
IWORK_DWNNLS(1) = D+1+5*N
IWORK_DWNNLS(2) = D+1+N
W_DWNNLS(1, :) = 1.0_R8          ! Set convexity (equality) constraint.
W_DWNNLS(2:D+1,1:N) = PTS(:,:)   ! Copy data points.
W_DWNNLS(2:D+1,N+1) = PROJ(:)    ! Copy extrapolation point.
! Compute the solution to the inequality constrained least squares problem to
! get the projection coefficients.
CALL DWNNLS(W_DWNNLS, D+1, 1, D, N, 0, PRGOPT_DWNNLS, X_DWNNLS, RNORML, &
```

```
   IERR(MI), IWORK_DWNNLS, WORK_DWNNLS)
IF (IERR(MI) .EQ. 1) THEN ! Failure to converge.
   IERR(MI) = 71; RETURN
ELSE IF (IERR(MI) .EQ. 2) THEN ! Illegal input detected.
   IERR(MI) = 72; RETURN
END IF
! Zero all weights that are approximately zero and renormalize the sum.
WHERE (X_DWNNLS < EPSL) X_DWNNLS = 0.0_R8
X_DWNNLS(:) = X_DWNNLS(:) / SUM(X_DWNNLS)
! Compute the actual projection via matrix vector multiplication.
CALL DGEMV('N', D, N, 1.0_R8, PTS, D, X_DWNNLS, 1, 0.0_R8, PROJ, 1)
RNORML = DNRM2(D, PROJ(:) - Q(:,MI), 1)
RETURN
END SUBROUTINE PROJECT

SUBROUTINE RESCALE(MINDIST, DIAMETER, SCALE)
! Rescale and transform data to be centered at the origin with unit
! radius. This subroutine has O(n^2) complexity.
!
! On output, PTS and Q have been rescaled and shifted. All the data
! points in PTS are centered with unit radius, and the points in Q
! have been shifted and scaled in relation to PTS.
!
! MINDIST is a real number containing the (scaled) minimum distance
!    between any two data points in PTS.
!
! DIAMETER is a real number containing the (scaled) diameter of the
!    data set PTS.
!
! SCALE contains the real factor used to transform the data and
!    interpolation points: scaled value = (original value -
!    barycenter of data points)/SCALE.

! Output arguments.
REAL(KIND=R8), INTENT(OUT) :: MINDIST, DIAMETER, SCALE

! Local variables.
REAL(KIND=R8) :: PTS_CENTER(D) ! The center of the data points PTS.
REAL(KIND=R8) :: DISTANCE ! The current distance.

! Initialize local values.
MINDIST = HUGE(0.0_R8)
DIAMETER = 0.0_R8
SCALE = 0.0_R8

! Compute barycenter of all data points.
PTS_CENTER(:) = SUM(PTS(:,:), DIM=2)/REAL(N, KIND=R8)
! Center the points.
FORALL (I = 1:N) PTS(:,I) = PTS(:,I) - PTS_CENTER(:)
! Compute the scale factor (for unit radius).
DO I = 1, N ! Cycle through all points again.
   DISTANCE = DNRM2(D, PTS(:,I), 1) ! Compute the distance from the center.
   IF (DISTANCE > SCALE) THEN ! Compare to the current radius.
      SCALE = DISTANCE
   END IF
END DO
! Scale the points to unit radius.
PTS = PTS / SCALE
! Also transform Q similarly.
FORALL (I = 1:M) Q(:,I) = (Q(:,I) - PTS_CENTER(:)) / SCALE
! Compute the minimum and maximum distances.
IF (EXACTL) THEN
   ! If exact error error checking is turned on, then compute the DIAMETER
   ! and MINDIST values.
   DO I = 1, N ! Cycle through all pairs of points.
      DO J = I + 1, N
         DISTANCE = DNRM2(D, PTS(:,I) - PTS(:,J), 1) ! Compute the distance.
         IF (DISTANCE > DIAMETER) THEN ! Compare to the current diameter.
            DIAMETER = DISTANCE
         END IF
         IF (DISTANCE < MINDIST) THEN ! Compare to the current minimum distance.
            MINDIST = DISTANCE
         END IF
      END DO
   END DO
ELSE
   ! If exact error checking is turned off, then the diameter is approximately
   ! 2.0 after rescaling and centering the points. The MINDIST is not computed.
   DIAMETER = 2.0_R8
   MINDIST = 1.0_R8
END IF
RETURN
END SUBROUTINE RESCALE

END SUBROUTINE DELAUNAYSPARSES


SUBROUTINE DELAUNAYSPARSEP( D, N, PTS, M, Q, SIMPS, WEIGHTS, IERR,  &
  INTERP_IN, INTERP_OUT, EPS, EXTRAP, RNORM, IBUDGET, CHAIN, EXACT, &
  PMODE )
! This is a parallel implementation of an algorithm for efficiently performing
! interpolation in R^D via the Delaunay triangulation. The algorithm is fully
```

```
! described and analyzed in
!
! T. H. Chang, L. T. Watson,  T. C.H. Lux, B. Li, L. Xu, A. R. Butt, K. W.
! Cameron, and Y. Hong. 2018. A polynomial time algorithm for multivariate
! interpolation in arbitrary dimension via the Delaunay triangulation. In
! Proceedings of the ACMSE 2018 Conference (ACMSE '18). ACM, New York, NY,
! USA. Article 12, 8 pages.
!
!
! On input:
!
! D is the dimension of the space for PTS and Q.
!
! N is the number of data points in PTS.
!
! PTS(1:D,1:N) is a real valued matrix with N columns, each containing the
!    coordinates of a single data point in R^D.
!
! M is the number of interpolation points in Q.
!
! Q(1:D,1:M) is a real valued matrix with M columns, each containing the
!    coordinates of a single interpolation point in R^D.
!
!
! On output:
!
! PTS and Q have been rescaled and shifted. All the data points in PTS
!    are now contained in the unit hyperball in R^D, and the points in Q
!    have been shifted and scaled accordingly in relation to PTS.
!
! SIMPS(1:D+1,1:M) contains the D+1 integer indices (corresponding to columns
!    in PTS) for the D+1 vertices of the Delaunay simplex containing each
!    interpolation point in Q.
!
! WEIGHTS(1:D+1,1:M) contains the D+1 real valued weights for expressing each
!    point in Q as a convex combination of the D+1 corresponding vertices
!    in SIMPS.
!
! IERR(1:M) contains integer valued error flags associated with the
!    computation of each of the M interpolation points in Q. The error
!    codes are:
!
! 00 : Succesful interpolation.
! 01 : Succesful extrapolation (up to the allowed extrapolation distance).
! 02 : This point was outside the allowed extrapolation distance; the
!      corresponding entries in SIMPS and WEIGHTS contain zero values.
!
! 10 : The dimension D must be positive.
! 11 : Too few data points to construct a triangulation (i.e., N < D+1).
! 12 : No interpolation points given (i.e., M < 1).
! 13 : The first dimension of PTS does not agree with the dimension D.
! 14 : The second dimension of PTS does not agree with the number of points N.
! 15 : The first dimension of Q does not agree with the dimension D.
! 16 : The second dimension of Q does not agree with the number of
!      interpolation points M.
! 17 : The first dimension of the output array SIMPS does not match the number
!      of vertices needed for a D-simplex (D+1).
! 18 : The second dimension of the output array SIMPS does not match the
!      number of interpolation points M.
! 19 : The first dimension of the output array WEIGHTS does not match the
!      number of vertices for a a D-simplex (D+1).
! 20 : The second dimension of the output array WEIGHTS does not match the
!      number of interpolation points M.
! 21 : The size of the error array IERR does not match the number of
!      interpolation points M.
! 22 : INTERP_IN cannot be present without INTERP_OUT or vice versa.
! 23 : The first dimension of INTERP_IN does not match the first
!      dimension of INTERP_OUT.
! 24 : The second dimension of INTERP_IN does not match the number of
!      data points PTS.
! 25 : The second dimension of INTERP_OUT does not match the number of
!      interpolation points M.
! 26 : The budget supplied in IBUDGET does not contain a positive
!      integer.
! 27 : The extrapolation distance supplied in EXTRAP cannot be negative.
! 28 : The size of the RNORM output array does not match the number of
!      interpolation points M.
!
! 30 : Two or more points in the data set PTS are too close together with
!      respect to the working precision (EPS), which would result in a
!      numerically degenerate simplex.
! 31 : All the data points in PTS lie in some lower dimensional linear
!      manifold (up to the working precision), and no valid triangulation
!      exists.
! 40 : An error caused DELAUNAYSPARSEP to terminate before this value could
!      be computed. Note: The corresponding entries in SIMPS and WEIGHTS may
!      contain garbage values.
!
! 50 : A memory allocation error occurred while allocating the work array
!      WORK.
!
! 60 : The budget was exceeded before the algorithm converged on this
```

```
!      value. If the dimension is high, try increasing IBUDGET. This
!      error can also be caused by a working precision EPS that is too
!      small for the conditioning of the problem.
!
! 61 : A value that was judged appropriate later caused LAPACK to encounter a
!      singularity. Try increasing the value of EPS.
!
! 70 : Allocation error for the extrapolation work arrays.
! 71 : The SLATEC subroutine DWNNLS failed to converge during the projection
!      of an extrapolation point onto the convex hull.
! 72 : The SLATEC subroutine DWNNLS has reported a usage error.
!
!      The errors 72, 80--83 should never occur, and likely indicate a
!      compiler bug or hardware failure.
! 80 : The LAPACK subroutine DGEQP3 has reported an illegal value.
! 81 : The LAPACK subroutine DGETRF has reported an illegal value.
! 82 : The LAPACK subroutine DGETRS has reported an illegal value.
! 83 : The LAPACK subroutine DORMQR has reported an illegal value.
!
! 90 : The value of PMODE is not valid.
!
!
! Optional arguments:
!
! INTERP_IN(1:IR,1:N) contains real valued response vectors for each of
!    the data points in PTS on input. The first dimension of INTERP_IN is
!    inferred to be the dimension of these response vectors, and the
!    second dimension must match N. If present, the response values will
!    be computed for each interpolation point in Q, and stored in INTERP_OUT,
!    which therefore must also be present. If both INTERP_IN and INTERP_OUT
!    are omitted, only the containing simplices and convex combination
!    weights are returned.
!
! INTERP_OUT(1:IR,1:M) contains real valued response vectors for each
!    interpolation point in Q on output. The first dimension of INTERP_OU
!    must match the first dimension of INTERP_IN, and the second dimension
!    must match M. If present, the response values at each interpolation
!    point are computed as a convex combination of the response values
!    (supplied in INTERP_IN) at the vertices of a Delaunay simplex containing
!    that interpolation point.  Therefore, if INTERP_OUT is present, then
!    INTERP_IN must also be present.  If both are omitted, only the
!    simplices and convex combination weights are returned.
!
! EPS contains the real working precision for the problem on input. By
!    default, EPS is assigned \sqrt{\mu} where \mu denotes the unit roundoff
!    for the machine. In general, any values that differ by less than EPS
!    are judged as equal, and any weights that are greater than -EPS are
!    judged as nonnegative.  EPS cannot take a value less than the default
!    value of \sqrt{\mu}. If any value less than \sqrt{\mu} is supplied,
!    the default value will be used instead automatically.
!
! EXTRAP contains the real maximum extrapolation distance (relative to the
!    diameter of PTS) on input. Interpolation at a point outside the convex
!    hull of PTS is done by projecting that point onto the convex hull, and
!    then doing normal Delaunay interpolation at that projection.
!    Interpolation at any point in Q that is more than EXTRAP * DIAMETER(PTS)
!    units outside the convex hull of PTS will not be done and an error code
!    of 2 will be returned. Note that computing the projection can be
!    expensive. Setting EXTRAP=0 will cause all extrapolation points to be
!    ignored without ever computing a projection. By default, EXTRAP=0.1
!    (extrapolate by up to 10% of the diameter of PTS).
!
! RNORM(1:M) contains the real unscaled projection (2-norm) distances from
!    any projection computations on output. If not present, these distances
!    are still computed for each extrapolation point, but are never returned.
!
! IBUDGET on input contains the integer budget for performing flips while
!    iterating toward the simplex containing each interpolation point in Q.
!    This prevents DELAUNAYSPARSEP from falling into an infinite loop when
!    an inappropriate value of EPS is given with respect to the problem
!    conditioning.  By default, IBUDGET=50000. However, for extremely
!    high-dimensional problems and pathological inputs, the default value
!    may be insufficient.
!
! CHAIN is a logical input argument that determines whether a new first
!    simplex should be constructed for each interpolation point
!    (CHAIN=.FALSE.), or whether the simplex walks should be "daisy-chained."
!    By default, CHAIN=.FALSE. Setting CHAIN=.TRUE. is generally not
!    recommended, unless the size of the triangulation is relatively small
!    or the interpolation points are known to be tightly clustered.
!
! EXACT is a logical input argument that determines whether the exact
!    diameter should be computed and whether a check for duplicate data
!    points should be performed in advance. When EXACT=.FALSE., the
!    diameter of PTS is approximated by twice the distance from the
!    barycenter of PTS to the farthest point in PTS, and no check is
!    done to find the closest pair of points, which could result in hard
!    to find bugs later on. When EXACT=.TRUE., the exact diameter is
!    computed and an error is returned whenever PTS contains duplicate
!    values up to the precision EPS. By default EXACT=.TRUE., but setting
!    EXACT=.FALSE. could result in significant speedup when N is large.
!    It is strongly recommended that most users leave EXACT=.TRUE., as
```

```
!    setting EXACT=.FALSE. could result in input errors that are difficult
!    to identify. Also, the diameter approximation could be wrong by up to
!    a factor of two.
!
! PMODE is an integer specifying the level of parallelism to be exploited.
!    If PMODE = 1, then parallelism is exploited at the level of the loop
!    over all interpolation points (Level 1 parallelism).
!    If PMODE = 2, then parallelism is exploited at the level of the loops
!    over data points when constructing/flipping simplices (Level 2
!    parallelism).
!    If PMODE = 3, then parallelism is exploited at both levels. Note: this
!    implies that the total number of threads active at any time could be up
!    to OMP_NUM_THREADS^2.
!    By default, PMODE is set to 1 if there is more than 1 interpolation
!    point and 2 otherwise.
!
!
! Subroutines and functions directly referenced from BLAS are
!      DDOT, DGEMV, DNRM2, DTRSM,
! and from LAPACK are
!      DGEQP3, DGETRF, DGETRS, DORMQR.
! The SLATEC subroutine DWNNLS is directly referenced. DWNNLS and all its
! SLATEC dependencies have been slightly edited to comply with the Fortran
! 2008 standard, with all print statements and references to stderr being
! commented out. For a reference to DWNNLS, see ACM TOMS Algorithm 587
! (Hanson and Haskell).  The module REAL_PRECISION from HOMPACK90 (ACM TOMS
! Algorithm 777) is used for the real data type. The REAL_PRECISION module,
! DELAUNAYSPARSEP, and DWNNLS and its dependencies comply with the Fortran
! 2008 standard.
!
! Primary Author: Tyler H. Chang
! Last Update: March, 2020
!
USE REAL_PRECISION, ONLY : R8
IMPLICIT NONE

! Input arguments.
INTEGER, INTENT(IN) :: D, N
REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:) ! Rescaled on output.
INTEGER, INTENT(IN) :: M
REAL(KIND=R8), INTENT(INOUT) :: Q(:,:) ! Rescaled on output.
! Output arguments.
INTEGER, INTENT(OUT) :: SIMPS(:,:)
REAL(KIND=R8), INTENT(OUT) :: WEIGHTS(:,:)
INTEGER, INTENT(OUT) :: IERR(:)
! Optional arguments.
REAL(KIND=R8), INTENT(IN), OPTIONAL:: INTERP_IN(:,:)
REAL(KIND=R8), INTENT(OUT), OPTIONAL :: INTERP_OUT(:,:)
REAL(KIND=R8), INTENT(IN), OPTIONAL:: EPS, EXTRAP
REAL(KIND=R8), INTENT(OUT), OPTIONAL :: RNORM(:)
INTEGER, INTENT(IN), OPTIONAL :: IBUDGET, PMODE
LOGICAL, INTENT(IN), OPTIONAL :: CHAIN
LOGICAL, INTENT(IN), OPTIONAL :: EXACT

! Local copies of optional input arguments.
REAL(KIND=R8) :: EPSL, EXTRAPL
INTEGER :: IBUDGETL
LOGICAL :: CHAINL, EXACTL, PLVL1, PLVL2

! Local variables.
LOGICAL :: PTINSIMP ! Tells if Q(:,MI) is in SIMPS(:,MI).
INTEGER :: I, J, K ! Loop iteration variables.
INTEGER :: IEXTRAPS ! Extrapolation budget.
INTEGER :: IERR_PRIV ! Private copy of the error flag.
INTEGER :: ITMP, JTMP ! Temporary variables for swapping, looping, etc.
INTEGER :: LWORK ! Size of the work array.
INTEGER :: MI ! Index of current interpolation point.
INTEGER :: VERTEX_PRIV ! Private copy of next vertex to add.
REAL(KIND=R8) :: CURRRAD ! Radius of the current circumsphere.
REAL(KIND=R8) :: MINRAD ! Minimum circumsphere radius observed.
REAL(KIND=R8) :: MINRAD_PRIV ! Private copy of MINRAD.
REAL(KIND=R8) :: PTS_DIAM ! Scaled diameter of data set.
REAL(KIND=R8) :: PTS_SCALE ! Data scaling factor.
REAL(KIND=R8) :: RNORML ! Euclidean norm of the projection residual.
REAL(KIND=R8) :: SIDE1, SIDE2 ! Signs (+/-1) denoting sides of a facet.

! Local arrays, requiring O(d^2) additional memory.
INTEGER :: IPIV(D) ! Pivot indices.
INTEGER :: SEED(D+1) ! Copy of the SEED simplex. Only used if CHAIN = .TRUE.
REAL(KIND=R8) :: AT(D,D) ! The transpose of A, the linear coefficient matrix.
REAL(KIND=R8) :: B(D) ! The RHS of a linear system.
REAL(KIND=R8) :: CENTER(D) ! The circumcenter of a simplex.
REAL(KIND=R8) :: CENTER_PRIV(D) ! Private copy of CENTER.
REAL(KIND=R8) :: LQ(D,D) ! Holds LU or QR factorization of AT.
REAL(KIND=R8) :: PLANE(D+1) ! The hyperplane containing a facet.
REAL(KIND=R8) :: PRGOPT_DWNNLS(1) ! Options array for DWNNLS.
REAL(KIND=R8) :: PROJ(D) ! The projection of the current iterate.
REAL(KIND=R8) :: TAU(D) ! Householder reflector constants.
REAL(KIND=R8) :: X(D) ! The solution to a linear system.

! Extrapolation work arrays are only allocated if DWNNLS is called.
INTEGER, ALLOCATABLE :: IWORK_DWNNLS(:) ! Only for DWNNLS.
```

```
REAL(KIND=R8), ALLOCATABLE :: W_DWNNLS(:,:) ! Only for DWNNLS.
REAL(KIND=R8), ALLOCATABLE :: WORK(:) ! Allocated with size LWORK.
REAL(KIND=R8), ALLOCATABLE :: WORK_DWNNLS(:) ! Only for DWNNLS.
REAL(KIND=R8), ALLOCATABLE :: X_DWNNLS(:) ! Only for DWNNLS.

! External functions and subroutines.
REAL(KIND=R8), EXTERNAL :: DDOT  ! Inner product (BLAS).
REAL(KIND=R8), EXTERNAL :: DNRM2 ! Euclidean norm (BLAS).
EXTERNAL :: DGEMV  ! General matrix vector multiply (BLAS)
EXTERNAL :: DGEQP3 ! Perform a QR factorization with column pivoting (LAPACK).
EXTERNAL :: DGETRF ! Perform a LU factorization with partial pivoting (LAPACK).
EXTERNAL :: DGETRS ! Use the output of DGETRF to solve a linear system (LAPACK).
EXTERNAL :: DORMQR ! Apply householder reflectors to a matrix (LAPACK).
EXTERNAL :: DTRSM  ! Perform a triangular solve (BLAS).
EXTERNAL :: DWNNLS ! Solve an inequality constrained least squares problem
                   ! (SLATEC).

! Check for input size and dimension errors.
IF (D < 1) THEN ! The dimension must satisfy D > 0.
   IERR(:) = 10; RETURN; END IF
IF (N < D+1) THEN ! Must have at least D+1 data points.
   IERR(:) = 11; RETURN; END IF
IF (M < 1) THEN ! Must have at least one interpolation point.
   IERR(:) = 12; RETURN; END IF
IF (SIZE(PTS,1) .NE. D) THEN ! Dimension of PTS array should match.
   IERR(:) = 13; RETURN; END IF
IF (SIZE(PTS,2) .NE. N) THEN ! Number of data points should match.
   IERR(:) = 14; RETURN; END IF
IF (SIZE(Q,1) .NE. D) THEN ! Dimension of Q should match.
   IERR(:) = 15; RETURN; END IF
IF (SIZE(Q,2) .NE. M) THEN ! Number of interpolation points should match.
   IERR(:) = 16; RETURN; END IF
IF (SIZE(SIMPS,1) .NE. D+1) THEN ! Need space for D+1 vertices per simplex.
   IERR(:) = 17; RETURN; END IF
IF (SIZE(SIMPS,2) .NE. M) THEN  ! There will be M output simplices.
   IERR(:) = 18; RETURN; END IF
IF (SIZE(WEIGHTS,1) .NE. D+1) THEN ! There will be D+1 weights per simplex.
   IERR(:) = 19; RETURN; END IF
IF (SIZE(WEIGHTS,2) .NE. M) THEN ! One vector of weights per simplex.
   IERR(:) = 20; RETURN; END IF
IF (SIZE(IERR) .NE. M) THEN ! An error flag for each interpolation point.
   IERR(:) = 21; RETURN; END IF

! Check for optional arguments.
IF (PRESENT(INTERP_IN) .NEQV. PRESENT(INTERP_OUT)) THEN
   IERR(:) = 22; RETURN; END IF
IF (PRESENT(INTERP_IN)) THEN ! Sizes must agree.
   IF (SIZE(INTERP_IN,1) .NE. SIZE(INTERP_OUT,1)) THEN
      IERR(:) = 23 ; RETURN; END IF
   IF(SIZE(INTERP_IN,2) .NE. N) THEN
      IERR(:) = 24; RETURN; END IF
   IF (SIZE(INTERP_OUT,2) .NE. M) THEN
      IERR(:) = 25; RETURN; END IF
   INTERP_OUT(:,:) = 0.0_R8 ! Initialize output to zeros.
END IF
EPSL = SQRT(EPSILON(0.0_R8)) ! Get the machine unit roundoff constant.
IF (PRESENT(EPS)) THEN
   IF (EPSL < EPS) THEN ! If the given precision is too small, ignore it.
      EPSL = EPS
   END IF
END IF
IF (PRESENT(IBUDGET)) THEN
   IBUDGETL = IBUDGET ! Use the given budget if present.
   IF (IBUDGETL < 1) THEN
      IERR(:) = 26; RETURN; END IF
ELSE
   IBUDGETL = 50000 ! Default value for budget.
END IF
IF (PRESENT(EXTRAP)) THEN
   EXTRAPL = EXTRAP
   IF (EXTRAPL < 0) THEN ! Check that the extrapolation distance is legal.
      IERR(:) = 27; RETURN; END IF
ELSE
   EXTRAPL = 0.1_R8 ! Default extrapolation distance (for normalized points).
END IF
IF (PRESENT(RNORM)) THEN
   IF (SIZE(RNORM,1) .NE. M) THEN ! The length of the array must match.
      IERR(:) = 28; RETURN; END IF
   RNORM(:) = 0.0_R8 ! Initialize output to zeros.
END IF
IF (PRESENT(CHAIN)) THEN
   CHAINL = CHAIN ! Turn chaining on, if necessarry.
   SEED(:) = 0 ! Initialize SEED in case it is needed.
ELSE
   CHAINL = .FALSE.
END IF
IF (PRESENT(EXACT)) THEN
   EXACTL = EXACT ! Set error checking and exact diameter computations.
ELSE
   EXACTL = .TRUE.
END IF
! Set the PMODE.
```

```
PLVL1 = .FALSE.
PLVL2 = .FALSE.
IF (PRESENT(PMODE)) THEN ! Check PMODE for legal values.
   IF (PMODE .EQ. 1) THEN
      PLVL1 = .TRUE.
   ELSE IF (PMODE .EQ. 2) THEN
      PLVL2 = .TRUE.
   ELSE IF (PMODE .EQ. 3) THEN
      PLVL1 = .TRUE.; PLVL2 = .TRUE.
   ELSE
      IERR(:) = 90; RETURN
   END IF
ELSE ! The default setting for PMODE is level 1 parallelism if M > 1.
   IF (M > 1) THEN
      PLVL1 = .TRUE.
   ELSE
      PLVL2 = .TRUE.
   END IF
END IF

! Scale and center the data points and interpolation points.
CALL RESCALE(MINRAD, PTS_DIAM, PTS_SCALE)
IF (MINRAD < EPSL) THEN ! Check for degeneracies in points spacing.
   IERR(:) = 30; RETURN; END IF

! Query DGEQP3 for optimal work array size (LWORK).
LWORK = -1
CALL DGEQP3(D,D,LQ,D,IPIV,TAU,B,LWORK,IERR(1))
LWORK = INT(B(1)) ! Compute the optimal work array size.
ALLOCATE(WORK(LWORK), STAT=I) ! Allocate WORK to size LWORK.
IF (I .NE. 0) THEN ! Check for memory allocation errors.
   IERR(:) = 50; RETURN; END IF

! Initialize PRGOPT_DWNNLS in case of extrapolation.
PRGOPT_DWNNLS(1) = 1.0_R8

! Initialize all error codes to "TBD" values.
IERR(:) = 40

! Begin level 1 parallel region (over all interpolation points in Q).
!$OMP PARALLEL &
!
! The FIRSTPRIVATE list specifies initialized variables, of which each
! thread has a private copy.
!$OMP& FIRSTPRIVATE(SEED), &
!
! The PRIVATE list specifies uninitialized variables, of which each
! thread has a private copy.
!$OMP& PRIVATE(I, J, K, IEXTRAPS, ITMP, JTMP, CURRRAD, MI, MINRAD,        &
!$OMP&         RNORML, SIDE1, SIDE2, IERR_PRIV, VERTEX_PRIV, MINRAD_PRIV, &
!$OMP&         PTINSIMP, IPIV, AT, B, CENTER, CENTER_PRIV, LQ, PLANE,     &
!$OMP&         PROJ, TAU, WORK, X, IWORK_DWNNLS, W_DWNNLS, WORK_DWNNLS,   &
!$OMP&         X_DWNNLS),                                                 &
!
! Any variables not explicitly listed above receive the SHARED scope
! by default and are visible across all threads.
!$OMP& DEFAULT(SHARED), &
!
!$OMP& IF(PLVL1)
!$OMP DO SCHEDULE(DYNAMIC)
OUTER : DO MI = 1, M
   !$OMP CRITICAL(CHECK_IERR)
   ! Check if this interpolation point was already found.
   IF (IERR(MI) .EQ. 40) THEN
      IERR(MI) = 0
      IERR_PRIV = 0
   ELSE
      IERR_PRIV = -1
   END IF
   !$OMP END CRITICAL(CHECK_IERR)
   IF(IERR_PRIV .EQ. -1) CYCLE OUTER

   ! Initialize the projection and reset the residual.
   PROJ(:) = Q(:,MI)
   RNORML = 0.0_R8

   ! Check if extrapolation is enabled.
   IF (EXTRAPL < EPSL) THEN
      IEXTRAPS = -1 ! If not, set the extrapolation budget negative.
   ELSE
      IEXTRAPS = 1 ! Allow for exactly one projection for this point.
   END IF

   ! If there is no useable seed or if chaining is turned off, then make a new
   ! simplex.
   IF( (.NOT. CHAINL) .OR. SEED(1) .EQ. 0) THEN
!      CALL MAKEFIRSTSIMP(); IF(IERR_PRIV .NE. 0) CYCLE OUTER


!*******************************************************************************
! Due to OpenMP's handling of variable scope, the parallel implementation of
! the subroutine MAKEFIRSTSIMP() has been in-lined here.
```

```
!
! SUBROUTINE MAKEFIRSTSIMP()
!
! Iteratively construct the first simplex by choosing points that
! minimize the radius of the smallest circumball. Let P_1, P_2, ..., P_K
! denote the current list of vertices for the simplex. Let P* denote the
! candidate vertex to be added to the simplex. Let CENTER denote the
! circumcenter of the simplex.  Then
!
! X = CENTER - P_1
!
! is given by the minimum norm solution to the underdetermined linear system
!
! A X = B, where
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_K - P_1, P* - P_1 ] and
! B = [ <A_{1.},A_{1.}>/2, <A_{2.},A_{2.}>/2, ..., <A_{K.},A_{K.}>/2 ]^T.
!
! Then the radius of the smallest circumsphere is CURRRAD = \| X \|,
! and the next vertex is given by P_{K+1} = argmin_{P*} CURRRAD, where P*
! ranges over points in PTS that are not already a vertex of the simplex.
!
! On output, this subroutine fully populates the matrix A^T and vector B,
! and fills SIMPS(:,MI) with the indices of a valid Delaunay simplex.

! Initialize simplex and shared variables.
SIMPS(:,MI) = 0
MINRAD_PRIV = HUGE(0.0_R8)
MINRAD = HUGE(0.0_R8)

! Below is a Level 2 parallel region over N points in PTS to find the
! first and second vertices SIMPS(1,MI) and SIMPS(2,MI).
!$OMP PARALLEL &
!
! The FIRSTPRIVATE list specifies initialized variables, of which each
! thread has a private copy.
!$OMP& FIRSTPRIVATE(MINRAD_PRIV), &
!
! The PRIVATE list specifies uninitialized variables, of which each
! thread has a private copy.
!$OMP& PRIVATE(I, CURRRAD, VERTEX_PRIV), &
!
! Any variables not explicitly listed above receive the SHARED scope
! by default and are visible across all threads.
!$OMP& DEFAULT(SHARED), &
!
!$OMP& IF(PLVL2)
! Find the first point, i.e., the closest point to Q(:,MI).
!$OMP DO SCHEDULE(STATIC)
DO I = 1, N
   ! Check the distance to Q(:,MI)
   CURRRAD = DNRM2(D, PTS(:,I) - PROJ(:), 1)
   IF (CURRRAD < MINRAD_PRIV) THEN
      MINRAD_PRIV = CURRRAD; VERTEX_PRIV = I;
   END IF
END DO
!$OMP END DO
!$OMP CRITICAL(REDUC_1)
IF (MINRAD_PRIV < MINRAD) THEN
   MINRAD = MINRAD_PRIV; SIMPS(1,MI) = VERTEX_PRIV;
END IF
!$OMP END CRITICAL(REDUC_1)
! Find the second point, i.e., the closest point to PTS(:,SIMPS(1,MI)).
MINRAD_PRIV = HUGE(0.0_R8)
!$OMP BARRIER
!$OMP SINGLE
MINRAD = HUGE(0.0_R8)
!$OMP END SINGLE
!$OMP DO SCHEDULE(STATIC)
DO I = 1, N
   ! Skip repeated vertices.
   IF (I .EQ. SIMPS(1,MI)) CYCLE
   ! Check the diameter of the resulting circumsphere.
   CURRRAD = DNRM2(D, PTS(:,I)-PTS(:,SIMPS(1,MI)), 1)
   IF (CURRRAD < MINRAD_PRIV) THEN
      MINRAD_PRIV = CURRRAD; VERTEX_PRIV = I
   END IF
END DO
!$OMP END DO
!$OMP CRITICAL(REDUC_2)
IF (MINRAD_PRIV < MINRAD) THEN
   MINRAD = MINRAD_PRIV; SIMPS(2,MI) = VERTEX_PRIV
END IF
!$OMP END CRITICAL(REDUC_2)
!$OMP END PARALLEL
! This is the end of the Level 2 parallel block.
IF (MINRAD < EPSL) THEN ! Check for degeneracies in points spacing.
   IERR(MI) = 30; CYCLE OUTER; END IF

! Set up the first row of the system A X = B.
AT(:,1) = PTS(:,SIMPS(2,MI)) - PTS(:,SIMPS(1,MI))
B(1) = DDOT(D, AT(:,1), 1, AT(:,1), 1) / 2.0_R8
```

```
! Loop to collect the remaining D-1 vertices for the first simplex.
DO I = 2, D
   ! Compute A^T P = Q R for the current matrix A^T.
   LQ(:,1:I-1) = AT(:,1:I-1)
   CALL DGEQP3(D, I-1, LQ, D, IPIV, TAU, WORK, LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 80
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! Set the RHS to P^T B.
   FORALL (ITMP = 1:I-1) X(ITMP) = B(IPIV(ITMP))
   ! Solve R^T Q^T X = P^T B for Q^T X, and save for later.
   CALL DTRSM('L', 'U', 'T', 'N', I-1, 1, 1.0_R8, LQ, D, X, D)
   ! Make a copy for computing the current center.
   CENTER(1:I-1) = X(1:I-1)
   CENTER(I:D) = 0.0_R8
   ! Apply Q from the left.
   CALL DORMQR('L', 'N', D, 1, I-1, LQ, D, TAU, CENTER, D, WORK, &
     LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 83
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   CENTER = CENTER + PTS(:,SIMPS(1,MI))
   ! Re-initialize the radius for each iteration.
   MINRAD = HUGE(0.0_R8)
   MINRAD_PRIV = HUGE(0.0_R8)
   VERTEX_PRIV = 0

   ! This is another Level 2 parallel block over N points in PTS.
   !$OMP PARALLEL &
   !
   ! The FIRSTPRIVATE list specifies initialized variables, of which each
   ! thread has a private copy.
   !$OMP& FIRSTPRIVATE(LQ, MINRAD_PRIV, VERTEX_PRIV, X), &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(J, CURRRAD, WORK), &
   !
   ! The REDUCTION clause specifies a PRIVATE variable that will retain
   ! some value (i.e., max, min, sum, etc.) upon output.
   !$OMP& REDUCTION(MAX:IERR_PRIV), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   !$OMP& IF(PLVL2)

   ! Initialize the error flag.
   IERR_PRIV = 0
   !$OMP DO SCHEDULE(STATIC)
   DO J = 1, N
      IF (IERR_PRIV .NE. 0) CYCLE ! If an error occurs, skip to the end.
      ! Check that this point is not already in the simplex.
      IF (ANY(SIMPS(:,MI) .EQ. J)) CYCLE
      ! If PTS(:,J) is more than twice MINRAD_PRIV from CENTER, do a quick skip.
      IF (DNRM2(D, CENTER - PTS(:,J), 1) > 2.0_R8 * MINRAD_PRIV) CYCLE
      ! Perform a rank-1 update to the current QR factorization of A^T by
      ! rotating PTS(:,I) - PTS(:,SIMPS(1,MI) by Q^T and storing in the
      ! final column of R.
      LQ(:,I) = PTS(:,J) - PTS(:,SIMPS(1,MI))
      CALL DORMQR('L', 'T', D, 1, I-1, LQ(:,1:I-1), D, TAU, LQ(:,I), D, &
        WORK, LWORK, IERR_PRIV)
      IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
         IERR_PRIV = 83; CYCLE
      END IF
      ! Implicitly apply the next Householder reflector.
      LQ(I,I) = DNRM2(D+1-I, LQ(I:D,I), 1)
      IF (LQ(I,I) < EPSL) THEN ! A is rank-deficient.
         CYCLE ! If rank-deficient, skip this point.
      END IF
      ! Update the current radius by \| Q^T X \| = \| X \|.
      WORK(1:I-1) = (LQ(1:I-1,I) / 2.0_R8) - X(1:I-1)
      WORK(I) = LQ(I,I) / 2.0_R8
      X(I) = DDOT(I, LQ(1:I,I), 1, WORK(1:I), 1) / LQ(I,I)
      CURRRAD = DNRM2(I, X(1:I), 1)
      ! Compare the last component of Q^T X to the current minimum.
      IF (CURRRAD < MINRAD_PRIV) THEN
         MINRAD_PRIV = CURRRAD; VERTEX_PRIV = J
      END IF
   END DO
   !$OMP END DO
   !$OMP CRITICAL(REDUC_3)
   IF (MINRAD_PRIV < MINRAD) THEN
      MINRAD = MINRAD_PRIV; SIMPS(I+1,MI) = VERTEX_PRIV
```

```
   END IF
   !$OMP END CRITICAL(REDUC_3)
   !$OMP END PARALLEL
   ! End of Level 2 parallel block.

   ! Check the final error flag.
   IF (IERR_PRIV .NE. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = IERR_PRIV
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! Check that a point was found. If not, then all the points must lie in a
   ! lower dimensional linear manifold (error case).
   IF (SIMPS(I+1,MI) .EQ. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 31
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! If all operations were successful, add the best P* to the linear system.
   AT(:,I) = PTS(:,SIMPS(I+1,MI)) - PTS(:,SIMPS(1,MI))
   B(I) = DDOT(D, AT(:,I), 1, AT(:,I), 1) / 2.0_R8
END DO
! RETURN
! END SUBROUTINE MAKEFIRSTSIMP
! This marks the end of the in-lined MAKEFIRSTSIMP() subroutine call.
!*******************************************************************************


   ! Otherwise, use the seed.
   ELSE
      ! Copy the seed to the current simplex.
      SIMPS(:,MI) = SEED(:)
      ! Rebuild the linear system.
      DO J=1,D
         AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
         B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
      END DO
   END IF

   ! Inner loop searching for a simplex containing the point Q(:,MI).
   INNER : DO K = 1, IBUDGETL

      ! If chaining is on, save each good simplex as the next seed.
      IF (CHAINL) SEED(:) = SIMPS(:,MI)


!*******************************************************************************
! Due to OpenMP's handling of variable scope, the parallel implementation of
! the subroutine PTINSIMP() has been in-lined here.
!
! FUNCTION PTINSIMP() RESULT(TF)
! Determine if any interpolation points are in the current simplex, whose
! vertices (P_1, P_2, ..., P_{D+1}) are indexed by SIMPS(:,MI). These
! vertices determine a positive cone with generators V_I = P_{I+1} - P_1,
! I = 1, ..., D. For each interpolation point Q* in Q, Q* - P_1 can be
! expressed as a unique linear combination of the V_I.  If all these linear
! weights are nonnegative and sum to less than or equal to 1.0, then Q* is
! in the simplex with vertices {P_I}_{I=1}^{D+1}.
!
! If any interpolation points in Q are contained in the simplex whose
! vertices are indexed by SIMPS(:,MI), then those points are marked as solved
! and the values of SIMPS and WEIGHTS are updated appropriately. On output,
! WEIGHTS(:,MI) contains the affine weights for producing Q(:,MI) as an
! affine combination of the points in PTS indexed by SIMPS(:,MI). If these
! weights are nonnegative, then PTINSIMP() returns TRUE.

! Initialize the return value and local variables.
PTINSIMP = .FALSE.

! Compute the LU factorization of the matrix A^T, whose columns are
! P_{I+1} - P_1.
LQ = AT
CALL DGETRF(D, D, LQ, D, IPIV, IERR_PRIV)
IF (IERR_PRIV < 0) THEN ! LAPACK illegal input.
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = 81
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
ELSE IF (IERR_PRIV > 0) THEN ! Rank-deficiency detected.
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = 61
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
END IF
! Solve A^T w = WORK to get the affine weights for Q(:,MI) or its projection.
WORK(1:D) = PROJ(:) - PTS(:,SIMPS(1,MI))
```

```
CALL DGETRS('N', D, 1, LQ, D, IPIV, WORK(1:D), D, IERR_PRIV)
IF (IERR_PRIV < 0) THEN ! LAPACK illegal input.
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = 82
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
END IF
WEIGHTS(2:D+1,MI) = WORK(1:D)
WEIGHTS(1,MI) = 1.0_R8 - SUM(WEIGHTS(2:D+1,MI))
! Check if the weights for Q(:,MI) are nonnegative.
IF (ALL(WEIGHTS(:,MI) .GE. -EPSL)) PTINSIMP = .TRUE.

! If Level 1 parallelism is active, do not parallelize this loop.
IF (PLVL1) THEN
   ! Loop over all remaining unsolved interoplation points. Uses PLANE(:)
   ! as a work array.
   DO I = MI+1, M
      ! Check that no solution has already been found.
      !$OMP CRITICAL(CHECK_IERR)
      ITMP = IERR(I)
      !$OMP END CRITICAL(CHECK_IERR)
      IF (ITMP .NE. 40) CYCLE
      ! Solve A^T w = PLANE to get the affine weights for Q(:,I).
      PLANE(2:D+1) = Q(:,I) - PTS(:,SIMPS(1,MI))
      CALL DGETRS('N', D, 1, LQ, D, IPIV, PLANE(2:D+1), D, ITMP)
      IF (ITMP < 0) CYCLE ! Illegal input error that should never occurr.
      ! Check if the weights define a convex combination.
      PLANE(1) = 1.0_R8 - SUM(PLANE(2:D+1))
      IF (ALL(PLANE(1:D+1) .GE. -EPSL)) THEN
         !$OMP CRITICAL(CHECK_IERR)
         IF(IERR(I) .EQ. 40) THEN
            ! Copy the simplex indices and weights then flag as complete.
            SIMPS(:,I) = SIMPS(:,MI)
            WEIGHTS(:,I) = PLANE(1:D+1)
            IERR(I) = 0
         END IF
         !$OMP END CRITICAL(CHECK_IERR)
      END IF
   END DO
! If Level 1 parallelism is not active, there will be no conflicts for
! parallelizing this loop.
ELSE
   ! Level 2 parallel block over all remaining unsolved interoplation
   ! points. Uses PLANE(:) as a work array.
   !$OMP PARALLEL DO &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(I, PLANE, ITMP), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   !$OMP& SCHEDULE(STATIC), &
   !$OMP& IF(PLVL2)
   DO I = MI+1, M
      ! Check that no solution has already been found.
      IF (IERR(I) .NE. 40) CYCLE
      ! Solve A^T w = PLANE to get the affine weights for Q(:,I).
      PLANE(2:D+1) = Q(:,I) - PTS(:,SIMPS(1,MI))
      CALL DGETRS('N', D, 1, LQ, D, IPIV, PLANE(2:D+1), D, ITMP)
      IF (ITMP < 0) CYCLE ! Illegal input error that should never occurr.
      ! Check if the weights define a convex combination.
      PLANE(1) = 1.0_R8 - SUM(PLANE(2:D+1))
      IF (ALL(PLANE(1:D+1) .GE. -EPSL)) THEN
         ! Copy the simplex indices and weights then flag as complete.
         SIMPS(:,I) = SIMPS(:,MI)
         WEIGHTS(:,I) = PLANE(1:D+1)
         IERR(I) = 0
      END IF
   END DO
   !$OMP END PARALLEL DO
END IF
! End of Level 2 parallel block.
! RETURN
! END FUNCTION PTINSIMP
! This marks the end of the in-lined PTINSIMP() subroutine call.
!*******************************************************************************


      ! Check if the current simplex contains Q(:,MI).
      IF (PTINSIMP) EXIT INNER

      ! Swap out the least weighted vertex, but save its value in case it
      ! needs to be restored later.
      JTMP = MINLOC(WEIGHTS(1:D+1,MI), DIM=1)
      ITMP = SIMPS(JTMP,MI)
      SIMPS(JTMP,MI) = SIMPS(D+1,MI)

      ! If the least weighted vertex (index JTMP) is not the first vertex,
```

```
      ! then just drop row (JTMP-1) from the linear system (corresponding
      ! to column (JTMP-1) of A^T).
      IF(JTMP .NE. 1) THEN
         AT(:,JTMP-1) = AT(:,D); B(JTMP-1) = B(D)
      ! However, if JTMP = 1, then both A^T and B must be reconstructed.
      ELSE
         DO J=1,D
            AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
            B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
         END DO
      END IF

      ! Compute the next simplex (do one flip).
!      CALL MAKESIMPLEX(); IF (IERR_PRIV .NE. 0) CYCLE OUTER


!*******************************************************************************
! Due to OpenMP's handling of variable scope, the parallel implementation of
! the subroutine MAKESIMPLEX() has been in-lined here.
!
! SUBROUTINE MAKESIMPLEX()
! Given a Delaunay facet F whose containing hyperplane does not contain
! Q(:,MI), complete the simplex by adding a point from PTS on the same 'side'
! of F as Q(:,MI). Assume SIMPS(1:D,MI) contains the vertex indices of F
! (corresponding to data points P_1, P_2, ..., P_D in PTS), and assume the
! matrix A(1:D-1,:)^T and vector B(1:D-1) are filled appropriately (similarly
! as in MAKEFIRSTSIMP()). Then for any P* (not in the hyperplane containing
! F) in PTS, let CENTER denote the circumcenter of the simplex with vertices
! P_1, P_2, ..., P_D, P*. Then
!
! X = CENTER - P_1
!
! is given by the solution to the nonsingular linear system
!
! A X = B where
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1, P* - P_1 ] and
! B = [ <A_{1.},A_{1.}>/2, <A_{2.},A_{2.}>/2, ..., <A_{D.},A_{D.}>/2 ]^T.
!
! Then CENTER = X + P_1 and RADIUS = \| X \|.  P_{D+1} will be given by the
! candidate P* that satisfies both of the following:
!
! 1) Let PLANE denote the hyperplane containing F. Then P_{D+1} and Q(:,MI)
! must be on the same side of PLANE.
!
! 2) The circumball about CENTER must not contain any points in PTS in its
! interior (Delaunay property).
!
! The above are necessary and sufficient conditions for flipping the
! Delaunay simplex, given that F is indeed a Delaunay facet.
!
! On input, SIMPS(1:D,MI) should contain the vertex indices (column indices
! from PTS) of the facet F.  Upon output, SIMPS(:,MI) will contain the vertex
! indices of a Delaunay simplex closer to Q(:,MI).  Also, the matrix A^T and
! vector B will be updated accordingly. If SIMPS(D+1,MI)=0, then there were
! no points in PTS on the appropriate side of F, meaning that Q(:,MI) is an
! extrapolation point (not a convex combination of points in PTS).

! Construct a hyperplane c^T x = \alpha containing the first D vertices indexed
! in SIMPS(:,MI). The plane is determined by its normal vector c and \alpha.
! Let P_1, P_2, ..., P_D be the vertices indexed in SIMPS(1:D,MI). A normal
! vector is any nonzero vector in ker A, where the matrix
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1 ].
!
! Since rank A = D-1, dim ker A = 1, and ker A can be found from a QR
! factorization of A^T:  A^T P = QR, where P permutes the columns of A^T.
! Then the last column of Q is orthogonal to the range of A^T, and in ker A.
IF (D > 1) THEN ! Check that D-1 > 0, otherwise the plane is trivial.
   ! Compute the QR factorization.
   IPIV=0
   LQ = AT
   CALL DGEQP3(D, D-1, LQ, D, IPIV, TAU, WORK, LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 80
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! The nullspace is given by the last column of Q.
   PLANE(1:D-1) = 0.0_R8
   PLANE(D) = 1.0_R8
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, PLANE, D, WORK, &
    LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 83
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
```

```
! Calculate the constant \alpha defining the plane.
PLANE(D+1) = DDOT(D,PLANE(1:D),1,PTS(:,SIMPS(1,MI)),1)
! Compute the sign for the side of PLANE containing Q(:,MI).
SIDE1 =  DDOT(D,PLANE(1:D),1,PROJ(:),1) - PLANE(D+1)
SIDE1 = SIGN(1.0_R8,SIDE1)

! Set the RHS to P^T B.
FORALL (ITMP = 1:D-1) X(ITMP) = B(IPIV(ITMP))
! Solve R^T Q^T X = P^T B for Q^T X.
CALL DTRSM('L', 'U', 'T', 'N', D-1, 1, 1.0_R8, LQ, D, X, D)

! Initialize the center, radius, simplex, and OpenMP variabls.
SIMPS(D+1,MI) = 0
CENTER(:) = 0.0_R8
CENTER_PRIV(:) = 0.0_R8
MINRAD = HUGE(0.0_R8)
MINRAD_PRIV = HUGE(0.0_R8)
VERTEX_PRIV = 0

! Begin Level 2 parallel loop over N points in PTS.
!$OMP PARALLEL &
!
! The FIRSTPRIVATE list specifies initialized variables, of which each
! thread has a private copy.
!$OMP& FIRSTPRIVATE(CENTER_PRIV, LQ, MINRAD_PRIV, VERTEX_PRIV), &
!
! The PRIVATE list specifies uninitialized variables, of which each
! thread has a private copy.
!$OMP& PRIVATE(I, SIDE2, WORK), &
!
! The REDUCTION clause specifies a PRIVATE variable that will retain
! some value (i.e., max, min, sum, etc.) upon output.
!$OMP& REDUCTION(MAX:IERR_PRIV), &
!
! Any variables not explicitly listed above receive the SHARED scope
! by default and are visible across all threads.
!$OMP& DEFAULT(SHARED), &
!
!$OMP& IF(PLVL2)

! Initialize the error flag.
IERR_PRIV = 0
!$OMP DO SCHEDULE(STATIC)
DO I = 1, N
   IF(IERR_PRIV .NE. 0) CYCLE ! If an error occurs, skip to the end.
   ! Check that P* is inside the current ball.
   IF (DNRM2(D, PTS(:,I) - CENTER_PRIV(:), 1) > MINRAD_PRIV) CYCLE
   ! Check that P* is on the appropriate halfspace.
   SIDE2 = DDOT(D,PLANE(1:D),1,PTS(:,I),1) - PLANE(D+1)
   IF (SIDE1*SIDE2 < EPSL .OR. ANY(SIMPS(:,MI) .EQ. I)) CYCLE
   ! Perform a rank-1 update to the current QR factorization of A^T by
   ! rotating PTS(:,I) - PTS(:,SIMPS(1,MI) by Q^T and storing in the
   ! final column of R.
   LQ(:,D) = PTS(:,I) - PTS(:,SIMPS(1,MI))
   CALL DORMQR('L', 'T', D, 1, D-1, LQ(:,1:D-1), D, TAU, LQ(:,D), D, WORK, &
     LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      IERR_PRIV = 83; CYCLE
   END IF
   ! Update the last element of Q^T X.
   WORK(1:D-1) = (LQ(1:D-1,D) / 2.0_R8) - X(1:D-1)
   WORK(D) = LQ(D,D) / 2.0_R8
   CENTER_PRIV(1:D-1) = X(1:D-1)
   CENTER_PRIV(D) = DDOT(D, LQ(:,D), 1, WORK(1:D), 1) / LQ(D,D)
   ! Get the center by applying Q to the solution.
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, CENTER_PRIV, D, &
      WORK, LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      IERR_PRIV = 83; CYCLE
   END IF
   ! Update the new radius, center, and simplex.
   MINRAD_PRIV = DNRM2(D, CENTER_PRIV, 1)
   CENTER_PRIV(:) = CENTER_PRIV(:) + PTS(:,SIMPS(1,MI))
   VERTEX_PRIV = I
END DO
!$OMP END DO
!$OMP CRITICAL(REDUC_4)
! Check if PTS(:,VERTEX_PRIV) is inside the circumball.
IF (VERTEX_PRIV .NE. 0) THEN
   IF (DNRM2(D, PTS(:,VERTEX_PRIV) - CENTER(:), 1) < MINRAD) THEN
      MINRAD = MINRAD_PRIV
      CENTER(:) = CENTER_PRIV(:)
      SIMPS(D+1,MI) = VERTEX_PRIV
   END IF
END IF
!$OMP END CRITICAL(REDUC_4)
!$OMP END PARALLEL
! End level 2 parallel region.

! Check for error flags.
IF(IERR_PRIV .NE. 0) THEN
   ! Store the error code.
```

```
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = IERR_PRIV
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! Check for extrapolation condition.
   IF(SIMPS(D+1,MI) .NE. 0) THEN
      ! Add new point to the linear system.
      AT(:,D) = PTS(:,SIMPS(D+1,MI)) - PTS(:,SIMPS(1,MI))
      B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
   END IF
ELSE ! Special case where D=1.
   PLANE(1) = 1.0_R8
   PLANE(2) = PTS(1,SIMPS(1,MI))
   SIDE1 = SIGN(1.0_R8, PROJ(1) - PLANE(2))
   ! Initialize the radius, simplex, and OpenMP variabls.
   SIMPS(2,MI) = 0
   MINRAD = HUGE(0.0_R8)
   MINRAD_PRIV = HUGE(0.0_R8)
   VERTEX_PRIV = 0
   ! Begin Level 2 parallel loop over N points in PTS.
   !$OMP PARALLEL &
   !
   ! The FIRSTPRIVATE list specifies initialized variables, of which each
   ! thread has a private copy.
   !$OMP& FIRSTPRIVATE(MINRAD_PRIV, VERTEX_PRIV), &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(I, SIDE2), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   !$OMP& IF(PLVL2)

   !$OMP DO SCHEDULE(STATIC)
   DO I = 1, N
      ! Check that P* is on the appropriate halfspace.
      SIDE2 = (PTS(1,I) - PLANE(2)) * SIDE1
      IF (SIDE2 < EPSL .OR. SIMPS(1,MI) .EQ. I) CYCLE
      ! Check that P* is closer than the current solution.
      IF (SIDE2 > MINRAD) CYCLE
      ! Update the minimum distance and save the index I.
      MINRAD_PRIV = SIDE2
      VERTEX_PRIV = I
   END DO
   !$OMP END DO
   !$OMP CRITICAL(REDUC_4)
   ! Check if PTS(:,VERTEX_PRIV) is inside the circumball.
   IF (VERTEX_PRIV .NE. 0) THEN
      IF (MINRAD_PRIV < MINRAD) THEN
         MINRAD = MINRAD_PRIV
         SIMPS(2,MI) = VERTEX_PRIV
      END IF
   END IF
   !$OMP END CRITICAL(REDUC_4)
   !$OMP END PARALLEL
   ! Check for extrapolation condition.
   IF(SIMPS(2,MI) .NE. 0) THEN
      ! Add new point to the linear system.
      AT(1,1) = PTS(1,SIMPS(2,MI)) - PTS(1,SIMPS(1,MI))
      B(1) = (AT(1,1) ** 2.0_R8) / 2.0_R8
   END IF
END IF
! RETURN
! END SUBROUTINE MAKESIMPLEX
! End of in-lined code for MAKESIMPLEX().
!*******************************************************************************


      ! If no vertex was found, then this is an extrapolation point.
      IF (SIMPS(D+1,MI) .EQ. 0) THEN
         ! If extrapolation is not allowed (EXTRAP=0), do not proceed.
         IF (IEXTRAPS < 0) THEN
            SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
            ! Set the error flag and skip this point.
            !$OMP CRITICAL(CHECK_IERR)
            IERR(MI) = 2
            !$OMP END CRITICAL(CHECK_IERR)
            CYCLE OUTER

         ! If extrapolation is allowed (EXTRAP>0), check the budget.
         ELSE IF (IEXTRAPS .EQ. 0) THEN
            ! A second projection has been attempted. This code is rarely
            ! called, except in extreme cases involving nearly singular
            ! simplices near the convex hull of P.

            ! Swap the weights to match the simplex indices, and zero the
            ! most negative weight.
            !$OMP CRITICAL(CHECK_IERR)
```

```
WEIGHTS(JTMP,MI) = WEIGHTS(D+1,MI)
WEIGHTS(D+1,MI) = 0.0_R8
!$OMP END CRITICAL(CHECK_IERR)
! Loop through all the remaining facets from which Q(:,MI) is
! visible, and attempt to flip across each one.
DO WHILE (SIMPS(D+1,MI) .EQ. 0)
   ! Restore the previous simplex and linear system.
   SIMPS(D+1,MI) = ITMP
   AT(:,D) = PTS(:,ITMP) - PTS(:,SIMPS(1,MI))
   B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
   ! Find the next most negative weight.
   JTMP = MINLOC(WEIGHTS(1:D+1,MI), DIM=1)
   ! Check if WEIGHTS(JTMP,MI) .GE. 0.
   IF (WEIGHTS(JTMP,MI) .GE. -EPSL) THEN
      ! There is no other direction to flip, so Q(:,MI) must be
      ! within EPSL of the current simplex.
      ! Project Q(:,MI) onto the current simplex.

      ! Since at least one projection has already been done,
      ! the work arrays have already been allocated.
      PRGOPT_DWNNLS(1) = 1.0_R8
      IWORK_DWNNLS(1) = 6*D + 6
      IWORK_DWNNLS(2) = 2*D + 2
      ! Set equality constraint.
      W_DWNNLS(1,1:D+2) = 1.0_R8
      ! Populate LS coefficient matrix and RHS.
      FORALL (I=1:D+1) W_DWNNLS(2:D+1,I) = PTS(:,SIMPS(I,MI))
      W_DWNNLS(2:D+1,D+2) = PROJ(:)
      ! Project onto the current simplex.
      CALL DWNNLS(W_DWNNLS, D+1, 1, D, D+1, 0, PRGOPT_DWNNLS, &
         WEIGHTS(:,MI), WORK(1), IERR_PRIV, IWORK_DWNNLS,     &
         WORK_DWNNLS)
      IF (IERR_PRIV .EQ. 1) THEN ! Failure to converge.
         !$OMP CRITICAL(CHECK_IERR)
         IERR(MI) = 71
         !$OMP END CRITICAL(CHECK_IERR)
         CYCLE OUTER
      ELSE IF (IERR_PRIV .EQ. 2) THEN ! Illegal input detected.
         !$OMP CRITICAL(CHECK_IERR)
         IERR(MI) = 72
         !$OMP END CRITICAL(CHECK_IERR)
         CYCLE OUTER
      END IF
      ! A solution has been found; return it.
      EXIT INNER
   END IF
                  ! Otherwise, swap the vertices.
                  ITMP = SIMPS(JTMP,MI)
                  SIMPS(JTMP,MI) = SIMPS(D+1,MI)
                  ! Swap the weights to match, and zero the most negative weight.
                  !$OMP CRITICAL(CHECK_IERR)
                  WEIGHTS(JTMP,MI) = WEIGHTS(D+1,MI)
                  WEIGHTS(D+1,MI) = 0.0_R8
                  !$OMP END CRITICAL(CHECK_IERR)
                  ! If the least weighted vertex (index JTMP) is not the first
vertex,
                  ! then just drop row (JTMP-1) from the linear system
                  ! (corresponding to the JTMP-1st column of A^T).
                  IF (JTMP .NE. 1) THEN
                     AT(:,JTMP-1) = AT(:,D); B(JTMP-1) = B(D)
                  ! However, if JTMP=1, then both A^T and B must be reconstructed.
                  ELSE
                     DO J=1,D
                        AT(:,J) = PTS(:,SIMPS(J+1,MI)) - PTS(:,SIMPS(1,MI))
                        B(J) = DDOT(D, AT(:,J), 1, AT(:,J), 1) / 2.0_R8
                     END DO
                  END IF
                  ! Compute another simplex (try to flip again).
!                  CALL MAKESIMPLEX(); IF (IERR(MI) .NE. 0) CYCLE OUTER


!*****************************************************************************
! Due to OpenMP's handling of variable scope, the parallel implementation of
! the subroutine MAKESIMPLEX() has been in-lined here.
!
! SUBROUTINE MAKESIMPLEX()
! Given a Delaunay facet F whose containing hyperplane does not contain
! Q(:,MI), complete the simplex by adding a point from PTS on the same 'side'
! of F as Q(:,MI). Assume SIMPS(1:D,MI) contains the vertex indices of F
! (corresponding to data points P_1, P_2, ..., P_D in PTS), and assume the
! matrix A(1:D-1,:)^T and vector B(1:D-1) are filled appropriately (similarly
! as in MAKEFIRSTSIMP()). Then for any P* (not in the hyperplane containing
! F) in PTS, let CENTER denote the circumcenter of the simplex with vertices
! P_1, P_2, ..., P_D, P*. Then
!
! X = CENTER - P_1
!
! is given by the solution to the nonsingular linear system
!
! A X = B where
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1, P* - P_1 ] and
```

```
! B = [ <A_{1.},A_{1.}>/2, <A_{2.},A_{2.}>/2, ..., <A_{D.},A_{D.}>/2 ]^T.
!
! Then CENTER = X + P_1 and RADIUS = \| X \|.  P_{D+1} will be given by the
! candidate P* that satisfies both of the following:
!
! 1) Let PLANE denote the hyperplane containing F. Then P_{D+1} and Q(:,MI)
! must be on the same side of PLANE.
!
! 2) The circumball about CENTER must not contain any points in PTS in its
! interior (Delaunay property).
!
! The above are necessary and sufficient conditions for flipping the
! Delaunay simplex, given that F is indeed a Delaunay facet.
!
! On input, SIMPS(1:D,MI) should contain the vertex indices (column indices
! from PTS) of the facet F.  Upon output, SIMPS(:,MI) will contain the vertex
! indices of a Delaunay simplex closer to Q(:,MI).  Also, the matrix A^T and
! vector B will be updated accordingly. If SIMPS(D+1,MI)=0, then there were
! no points in PTS on the appropriate side of F, meaning that Q(:,MI) is an
! extrapolation point (not a convex combination of points in PTS).

! Construct a hyperplane c^T x = \alpha containing the first D vertices indexed
! in SIMPS(:,MI). The plane is determined by its normal vector c and \alpha.
! Let P_1, P_2, ..., P_D be the vertices indexed in SIMPS(1:D,MI). A normal
! vector is any nonzero vector in ker A, where the matrix
!
! A^T = [ P_2 - P_1, P_3 - P_1, ..., P_D - P_1 ].
!
! Since rank A = D-1, dim ker A = 1, and ker A can be found from a QR
! factorization of A^T:  A^T P = QR, where P permutes the columns of A^T.
! Then the last column of Q is orthogonal to the range of A^T, and in ker A.
IF (D > 1) THEN ! Check that D-1 > 0, otherwise the plane is trivial.
   ! Compute the QR factorization.
   IPIV=0
   LQ = AT
   CALL DGEQP3(D, D-1, LQ, D, IPIV, TAU, WORK, LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 80
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! The nullspace is given by the last column of Q.
   PLANE(1:D-1) = 0.0_R8
   PLANE(D) = 1.0_R8
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, PLANE, D, WORK, &
    LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 83
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
   ! Calculate the constant \alpha defining the plane.
   PLANE(D+1) = DDOT(D,PLANE(1:D),1,PTS(:,SIMPS(1,MI)),1)
   ! Compute the sign for the side of PLANE containing Q(:,MI).
   SIDE1 =  DDOT(D,PLANE(1:D),1,PROJ(:),1) - PLANE(D+1)
   SIDE1 = SIGN(1.0_R8,SIDE1)
   ! Set the RHS to P^T B.
   FORALL (ITMP = 1:D-1) X(ITMP) = B(IPIV(ITMP))
   ! Solve R^T Q^T X = P^T B for Q^T X.
   CALL DTRSM('L', 'U', 'T', 'N', D-1, 1, 1.0_R8, LQ, D, X, D)
   ! Initialize the center, radius, simplex, and OpenMP variabls.
   SIMPS(D+1,MI) = 0
   CENTER(:) = 0.0_R8
   CENTER_PRIV(:) = 0.0_R8
   MINRAD = HUGE(0.0_R8)
   MINRAD_PRIV = HUGE(0.0_R8)
   VERTEX_PRIV = 0

   ! Begin Level 2 parallel loop over N points in PTS.
   !$OMP PARALLEL &
   !
   ! The FIRSTPRIVATE list specifies initialized variables, of which each
   ! thread has a private copy.
   !$OMP& FIRSTPRIVATE(CENTER_PRIV, LQ, MINRAD_PRIV, VERTEX_PRIV), &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(I, SIDE2, WORK), &
   !
   ! The REDUCTION clause specifies a PRIVATE variable that will retain
   ! some value (i.e., max, min, sum, etc.) upon output.
   !$OMP& REDUCTION(MAX:IERR_PRIV), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   !$OMP& IF(PLVL2)
```

```
! Initialize the error flag.
IERR_PRIV = 0
!$OMP DO SCHEDULE(STATIC)
DO I = 1, N
   IF(IERR_PRIV .NE. 0) CYCLE ! If an error occurs, skip to the end.
   ! Check that P* is inside the current ball.
   IF (DNRM2(D, PTS(:,I) - CENTER_PRIV(:), 1) > MINRAD_PRIV) CYCLE
   ! Check that P* is on the appropriate halfspace.
   SIDE2 = DDOT(D,PLANE(1:D),1,PTS(:,I),1) - PLANE(D+1)
   IF (SIDE1*SIDE2 < EPSL .OR. ANY(SIMPS(:,MI) .EQ. I)) CYCLE
   ! Perform a rank-1 update to the current QR factorization of A^T by
   ! rotating PTS(:,I) - PTS(:,SIMPS(1,MI) by Q^T and storing in the
   ! final column of R.
   LQ(:,D) = PTS(:,I) - PTS(:,SIMPS(1,MI))
   CALL DORMQR('L', 'T', D, 1, D-1, LQ(:,1:D-1), D, TAU, LQ(:,D), D, WORK, &
     LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      IERR_PRIV = 83; CYCLE
   END IF
   ! Update the last element of Q^T X.
   WORK(1:D-1) = (LQ(1:D-1,D) / 2.0_R8) - X(1:D-1)
   WORK(D) = LQ(D,D) / 2.0_R8
   CENTER_PRIV(1:D-1) = X(1:D-1)
   CENTER_PRIV(D) = DDOT(D, LQ(:,D), 1, WORK(1:D), 1) / LQ(D,D)
   ! Get the center by applying Q to the solution.
   CALL DORMQR('L', 'N', D, 1, D-1, LQ, D, TAU, CENTER_PRIV, D, &
      WORK, LWORK, IERR_PRIV)
   IF(IERR_PRIV < 0) THEN ! LAPACK illegal input error.
      IERR_PRIV = 83; CYCLE
   END IF
   ! Update the new radius, center, and simplex.
   MINRAD_PRIV = DNRM2(D, CENTER_PRIV, 1)
   CENTER_PRIV(:) = CENTER_PRIV(:) + PTS(:,SIMPS(1,MI))
   VERTEX_PRIV = I
END DO
!$OMP END DO
!$OMP CRITICAL(REDUC_4)
! Check if PTS(:,VERTEX_PRIV) is inside the circumball.
IF (VERTEX_PRIV .NE. 0) THEN
   IF (DNRM2(D, PTS(:,VERTEX_PRIV) - CENTER(:), 1) < MINRAD) THEN
      MINRAD = MINRAD_PRIV
      CENTER(:) = CENTER_PRIV(:)
      SIMPS(D+1,MI) = VERTEX_PRIV
   END IF
END IF
!$OMP END CRITICAL(REDUC_4)
!$OMP END PARALLEL
! End level 2 parallel region.

! Check for error flags.
IF(IERR_PRIV .NE. 0) THEN
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = IERR_PRIV
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
END IF
! Check for extrapolation condition.
IF(SIMPS(D+1,MI) .NE. 0) THEN
   ! Add new point to the linear system.
   AT(:,D) = PTS(:,SIMPS(D+1,MI)) - PTS(:,SIMPS(1,MI))
   B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
END IF
ELSE ! Special case where D=1.
   PLANE(1) = 1.0_R8
   PLANE(2) = PTS(1,SIMPS(1,MI))
   SIDE1 = SIGN(1.0_R8, PROJ(1) - PLANE(2))
   ! Initialize the radius, simplex, and OpenMP variabls.
   SIMPS(2,MI) = 0
   MINRAD = HUGE(0.0_R8)
   MINRAD_PRIV = HUGE(0.0_R8)
   VERTEX_PRIV = 0
   ! Begin Level 2 parallel loop over N points in PTS.
   !$OMP PARALLEL &
   !
   ! The FIRSTPRIVATE list specifies initialized variables, of which each
   ! thread has a private copy.
   !$OMP& FIRSTPRIVATE(MINRAD_PRIV, VERTEX_PRIV), &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(I, SIDE2), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   !$OMP& IF(PLVL2)

   !$OMP DO SCHEDULE(STATIC)
   DO I = 1, N
```

```
      ! Check that P* is on the appropriate halfspace.
      SIDE2 = (PTS(1,I) - PLANE(2)) * SIDE1
      IF (SIDE2 < EPSL .OR. SIMPS(1,MI) .EQ. I) CYCLE
      ! Check that P* is closer than the current solution.
      IF (SIDE2 > MINRAD) CYCLE
      ! Update the minimum distance and save the index I.
      MINRAD_PRIV = SIDE2
      VERTEX_PRIV = I
   END DO
   !$OMP END DO
   !$OMP CRITICAL(REDUC_4)
   ! Check if PTS(:,VERTEX_PRIV) is inside the circumball.
   IF (VERTEX_PRIV .NE. 0) THEN
      IF (MINRAD_PRIV < MINRAD) THEN
         MINRAD = MINRAD_PRIV
         SIMPS(2,MI) = VERTEX_PRIV
      END IF
   END IF
   !$OMP END CRITICAL(REDUC_4)
   !$OMP END PARALLEL
   ! Check for extrapolation condition.
   IF(SIMPS(2,MI) .NE. 0) THEN
      ! Add new point to the linear system.
      AT(1,1) = PTS(1,SIMPS(2,MI)) - PTS(1,SIMPS(1,MI))
      B(1) = (AT(1,1) ** 2.0_R8) / 2.0_R8
   END IF
END IF
! RETURN
! END SUBROUTINE MAKESIMPLEX
! End of in-lined code for MAKESIMPLEX().
!*******************************************************************************


            END DO
            ! If the loop terminates, then a good direction was found.
            ! Resume the visibility walk as normal.
            CYCLE INNER
         END IF

         ! Otherwise, project the extrapolation point onto the convex hull.
!         CALL PROJECT(); IF (IERR_PRIV .NE. 0) CYCLE OUTER


!*******************************************************************************
! Due to OpenMP's handling of variable scope, the parallel (identical to serial)
! implementation of the subroutine PROJECT() has been in-lined here.
!
! SUBROUTINE PROJECT()
! Project a point outside the convex hull of the point set onto the convex hull
! by solving an inequality constrained least squares problem. The solution to
! the least squares problem gives the projection as a convex combination of the
! data points. The projection can then be computed by performing a matrix
! vector multiplication.

! Allocate work arrays.
IF (.NOT. ALLOCATED(IWORK_DWNNLS)) THEN
   ALLOCATE(IWORK_DWNNLS(D+1+N), STAT=IERR_PRIV)
   IF(IERR_PRIV .NE. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 70
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
END IF
IF (.NOT. ALLOCATED(WORK_DWNNLS)) THEN
   ALLOCATE(WORK_DWNNLS(D+1+N*5), STAT=IERR_PRIV)
   IF(IERR_PRIV .NE. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 70
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
END IF
IF (.NOT. ALLOCATED(W_DWNNLS)) THEN
   ALLOCATE(W_DWNNLS(D+1,N+1), STAT=IERR_PRIV)
   IF(IERR_PRIV .NE. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 70
      !$OMP END CRITICAL(CHECK_IERR)
      CYCLE OUTER
   END IF
END IF
IF (.NOT. ALLOCATED(X_DWNNLS)) THEN
   ALLOCATE(X_DWNNLS(N), STAT=IERR_PRIV)
   IF(IERR_PRIV .NE. 0) THEN
      ! Store the error code.
      !$OMP CRITICAL(CHECK_IERR)
      IERR(MI) = 70
      !$OMP END CRITICAL(CHECK_IERR)
```

```
      CYCLE OUTER
   END IF
END IF

! Initialize work array and settings values.
IWORK_DWNNLS(1) = D+1+5*N
IWORK_DWNNLS(2) = D+1+N
W_DWNNLS(1, :) = 1.0_R8          ! Set convexity (equality) constraint.
W_DWNNLS(2:D+1,1:N) = PTS(:,:)   ! Copy data points.
W_DWNNLS(2:D+1,N+1) = PROJ(:)    ! Copy extrapolation point.
! Compute the solution to the inequality constrained least squares problem to
! get the projection coefficients.
CALL DWNNLS(W_DWNNLS, D+1, 1, D, N, 0, PRGOPT_DWNNLS, X_DWNNLS, RNORML, &
   IERR_PRIV, IWORK_DWNNLS, WORK_DWNNLS)
IF (IERR_PRIV .EQ. 1) THEN ! Failure to converge.
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = 71
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
ELSE IF (IERR(MI) .EQ. 2) THEN ! Illegal input detected.
   ! Store the error code.
   !$OMP CRITICAL(CHECK_IERR)
   IERR(MI) = 72
   !$OMP END CRITICAL(CHECK_IERR)
   CYCLE OUTER
END IF
! Compute the actual projection via matrix vector multiplication.
CALL DGEMV('N', D, N, 1.0_R8, PTS, D, X_DWNNLS, 1, 0.0_R8, PROJ, 1)
! Zero all weights that are approximately zero and renormalize the sum.
WHERE (X_DWNNLS < EPSL) X_DWNNLS = 0.0_R8
X_DWNNLS(:) = X_DWNNLS(:) / SUM(X_DWNNLS)
! Compute the actual projection via matrix vector multiplication.
CALL DGEMV('N', D, N, 1.0_R8, PTS, D, X_DWNNLS, 1, 0.0_R8, PROJ, 1)
RNORML = DNRM2(D, PROJ(:) - Q(:,MI), 1)
! RETURN
! END SUBROUTINE PROJECT
! End of in-lined code for PROJECT().
!***********************************************************************


         ! Check the value of RNORML for over-extrapolation.
         IF (RNORML > EXTRAPL * PTS_DIAM) THEN
            SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
            ! If present, record the unscaled RNORM output.
            IF (PRESENT(RNORM)) RNORM(MI) = RNORML*PTS_SCALE
               ! Set the error flag and skip this point.
               !$OMP CRITICAL(CHECK_IERR)
               IERR(MI) = 2
               !$OMP END CRITICAL(CHECK_IERR)
               CYCLE OUTER
            END IF

            ! Otherwise, restore the previous simplex and continue with the
            ! projected value.
            SIMPS(D+1,MI) = ITMP
            AT(:,D) = PTS(:,ITMP) - PTS(:,SIMPS(1,MI))
            B(D) = DDOT(D, AT(:,D), 1, AT(:,D), 1) / 2.0_R8
            IEXTRAPS = IEXTRAPS - 1 ! Decrement the budget.
         END IF

      ! End of inner loop for finding each interpolation point.
      END DO INNER

      ! Check for budget violation conditions.
      IF (K > IBUDGETL) THEN
         SIMPS(:,MI) = 0; WEIGHTS(:,MI) = 0  ! Zero all output values.
         ! Set the error flag and skip this point.
         !$OMP CRITICAL(CHECK_IERR)
         IERR(MI) = 60
         !$OMP END CRITICAL(CHECK_IERR)
         CYCLE OUTER
      END IF

      ! If the residual is nonzero, set the extrapolation flag.
      IF (RNORML > EPSL) THEN
         !$OMP CRITICAL(CHECK_IERR)
         IERR(MI) = 1
         !$OMP END CRITICAL(CHECK_IERR)
      END IF

      ! If present, record the RNORM output.
      IF (PRESENT(RNORM)) RNORM(MI) = RNORML*PTS_SCALE

   END DO OUTER  ! End of outer loop over all interpolation points.
   !$OMP END DO

   ! If INTERP_IN and INTERP_OUT are present, compute all values f(q).
   IF (PRESENT(INTERP_IN)) THEN
      ! Level 1 parallel loop over all interpolation points.
      !$OMP DO SCHEDULE(STATIC)
      DO MI = 1, M
```

```
      ! Check for errors.
      IF (IERR(MI) .LE. 1) THEN
         ! Compute the weighted sum of vertex response values.
         DO K = 1, D+1
            INTERP_OUT(:,MI) = INTERP_OUT(:,MI) &
             + INTERP_IN(:,SIMPS(K,MI)) * WEIGHTS(K,MI)
         END DO
      END IF
   END DO
   !$OMP END DO
END IF

! Free optional work arrays.
IF (ALLOCATED(IWORK_DWNNLS)) DEALLOCATE(IWORK_DWNNLS)
IF (ALLOCATED(WORK_DWNNLS))  DEALLOCATE(WORK_DWNNLS)
IF (ALLOCATED(W_DWNNLS))     DEALLOCATE(W_DWNNLS)
IF (ALLOCATED(X_DWNNLS))     DEALLOCATE(X_DWNNLS)
!$OMP END PARALLEL
! End of Level 1 parallel region.

! Free dynamic work arrays.
DEALLOCATE(WORK)

RETURN

CONTAINS    ! Internal subroutines and functions.

SUBROUTINE RESCALE(MINDIST, DIAMETER, SCALE)
! Rescale and transform data to be centered at the origin with unit
! radius.
!
! The parallel implementation of this subroutine exploits parallelism
! over loops of length N. For nested loops, this subroutine follows
! the OpenMP recommendation of a static schedule with a fixed chunk
! size (of 100).
!
! On output, PTS and Q have been rescaled and shifted. All the data
! points in PTS are centered with unit radius, and the points in Q
! have been shifted and scaled in relation to PTS.
!
! MINDIST is a real number containing the (scaled) minimum distance
!    between any two data points in PTS.
!
! DIAMETER is a real number containing the (scaled) diameter of the
!    data set PTS.
!
! SCALE contains the real factor used to transform the data and
!    interpolation points: scaled value = (original value -
!    barycenter of data points)/SCALE.

! Output arguments.
REAL(KIND=R8), INTENT(OUT) :: MINDIST, DIAMETER, SCALE

! Local variables.
REAL(KIND=R8) :: PTS_CENTER(D) ! The center of the data points PTS.
REAL(KIND=R8) :: DISTANCE ! The current distance.

! Initialize local values.
MINDIST = HUGE(0.0_R8)
DIAMETER = 0.0_R8
SCALE = 0.0_R8

! Compute barycenter of all data points.
PTS_CENTER(:) = SUM(PTS(:,:), DIM=2)/REAL(N, KIND=R8)
! Center the points.
FORALL (I = 1:N) PTS(:,I) = PTS(:,I) - PTS_CENTER(:)
! Compute the scale factor (for unit radius).
!$OMP PARALLEL DO &
!$OMP& PRIVATE(I, DISTANCE), &
!$OMP& REDUCTION(MAX:SCALE), &
!$OMP& SCHEDULE(STATIC),     &
!$OMP& DEFAULT(SHARED)
DO I = 1, N ! Cycle through all points again.
   DISTANCE = DNRM2(D, PTS(:,I), 1) ! Compute the distance from the center.
   IF (DISTANCE > SCALE) THEN ! Compare to the current radius.
      SCALE = DISTANCE
   END IF
END DO
!$OMP END PARALLEL DO
! Scale the points to unit radius.
PTS = PTS / SCALE
! Also transform Q similarly.
FORALL (I = 1:M) Q(:,I) = (Q(:,I) - PTS_CENTER(:)) / SCALE
! Compute the minimum and maximum distances.
IF (EXACTL) THEN
   ! If exact error error checking is turned on, then compute the DIAMETER
   ! and MINDIST values.
   !$OMP PARALLEL DO &
   !$OMP& PRIVATE(I, DISTANCE),    &
   !$OMP& REDUCTION(MAX:DIAMETER), &
   !$OMP& REDUCTION(MIN:MINDIST),  &
   !$OMP& SCHEDULE(STATIC, 100),   &
```

```
   !$OMP& DEFAULT(SHARED)
   DO I = 1, N ! Cycle through all pairs of points.
      DO J = I + 1, N
         DISTANCE = DNRM2(D, PTS(:,I) - PTS(:,J), 1) ! Compute the distance.
         IF (DISTANCE > DIAMETER) THEN ! Compare to the current diameter.
            DIAMETER = DISTANCE
         END IF
         IF (DISTANCE < MINDIST) THEN ! Compare to the current minimum distance.
            MINDIST = DISTANCE
         END IF
      END DO
   END DO
   !$OMP END PARALLEL DO
ELSE
   ! If exact error checking is turned off, then the diameter is approximately
   ! 2.0 after rescaling and centering the points. The MINDIST is not computed.
   DIAMETER = 2.0_R8
   MINDIST = 1.0_R8
END IF
RETURN
END SUBROUTINE RESCALE

END SUBROUTINE DELAUNAYSPARSEP
```

Appendix C Code for VTMOP

C.1 vtmop.f90

```
! Module and subroutines implementing an adaptive weighting scheme for
! generating uniformly spaced points on the Pareto front for multiobjective
! optimization problems.
!
! Last Update : May, 2020 by Tyler Chang
MODULE VTMOP_MOD
! Use the R8 data type from HOMPACK90 for approximately 64-bit precision on
! all known machines.
USE REAL_PRECISION, ONLY : R8
USE OMP_LIB

! The default scope for VTMOP_MOD is public.
PUBLIC
! The following module arrays and pointers are private.
PRIVATE :: VTMOP_MOD_WEIGHTS, VTMOP_MOD_SURROGATES
! The following LSHEP (ACM TOMS Algorithm 905) parameters are private.
PRIVATE :: LSHEP_D, LSHEP_P, LSHEP_N_PTS, LSHEP_N_TABOO, LSHEP_A, &
           LSHEP_DES_TOL, LSHEP_FVALS, LSHEP_RW, LSHEP_SCALE,     &
           LSHEP_SHIFT, LSHEP_TABOO, LSHEP_XVALS
! The following auxiliary subroutines are private.
PRIVATE :: SCALAR_FUNC, SURROGATE_FUNC

! The derived data VTMOP_TYPE type carries metadata about the multiobjective
! optimization problem between iterations of the algorithm.
TYPE VTMOP_TYPE
   ! Contents of the VTMOP_TYPE data object.
   INTEGER :: D, P ! Problem dimensions.
   INTEGER :: ITERATE ! Total number of iterations elapsed.
   INTEGER :: LCLIST ! Length of CLIST(:,:) array.
   INTEGER :: LOPT_BUDGET ! Budget for the local optimizer.
   LOGICAL :: CHKPT ! Checkpointing mode.
   LOGICAL :: PMODE ! Parallel execution mode.
   REAL(KIND=R8) :: DECAY ! Rate of decay for the local trust region (LTR).
   REAL(KIND=R8) :: DES_TOL ! Design point tolerance.
   REAL(KIND=R8) :: EPS ! Working precision for the problem.
   REAL(KIND=R8) :: EPSW ! Fudge factor for zero weights.
   REAL(KIND=R8) :: MIN_RADF ! Minimum LTR radius as a fraction of UB-LB.
   REAL(KIND=R8) :: OBJ_TOL ! Objective point tolerance.
   REAL(KIND=R8) :: TRUST_RADF ! Initial LTR radius as a fraction of UB-LB.
```

```
   REAL(KIND=R8), ALLOCATABLE :: CLIST(:,:) ! Previously used LTR centers.
   REAL(KIND=R8), ALLOCATABLE :: LB(:), UB(:) ! Lower and upper bounds.
   REAL(KIND=R8), ALLOCATABLE :: WEIGHTS(:,:) ! Adaptive weights.

   ! Pointers to procedures that are called by the subroutine VTMOP_OPT.
   ! Subroutine to evaluate surrogate models.
   PROCEDURE(VTMOP_MOD_SEVAL_INT), NOPASS, POINTER :: EVAL_SURROGATES
   ! Subroutine to fit surrogate models.
   PROCEDURE(VTMOP_MOD_SFIT_INT), NOPASS, POINTER :: FIT_SURROGATES
   ! Subroutine to perform local optimization over surrogate models.
   PROCEDURE(VTMOP_MOD_LOCAL_INT), NOPASS, POINTER :: LOCAL_OPT
END TYPE VTMOP_TYPE

! Module variables.
! Module variables for checkpointing.
CHARACTER(LEN=20) :: VTMOP_CHKPTFILE = "vtmop.chkpt" ! Checkpoint file name.
CHARACTER(LEN=20) :: VTMOP_DATAFILE = "vtmop.dat" ! Database file name.
! LSHEP (ACM TOMS Algorithm 905) surrogate model parameters.
INTEGER :: LSHEP_D, LSHEP_N_PTS, LSHEP_N_TABOO, LSHEP_P
REAL(KIND=R8) :: LSHEP_DES_TOL
! Module variables for checkpointing.
INTEGER :: VTMOP_CHKPTUNIT = 11 ! Iteration data checkpoint unit.
INTEGER :: VTMOP_DATAUNIT = 12 ! Function evaluation database unit.
! Module variables containing problem metadata.
INTEGER :: VTMOP_MOD_BB_BUDGET ! The blackbox function evaluation budget.
INTEGER :: VTMOP_MOD_D ! Number of design variables.
INTEGER :: VTMOP_MOD_DBN ! Size of the databases.
INTEGER :: VTMOP_MOD_P ! Number of objectives.
! OpenMP lock for synchronizing parallel access to the database.
INTEGER(KIND=OMP_LOCK_KIND) :: VTMOP_MOD_DBLCK
! Module variables for checkpointing.
LOGICAL :: VTMOP_MOD_CHKPT ! Global copy of the checkpointing status.
! Module variables containing problem metadata.
REAL(KIND=R8) :: VTMOP_MOD_DES_TOL ! Design space tolerance.

! Dynamic module arrays.
! OpenMP locks for synchronizing parallel access to the database.
INTEGER(KIND=OMP_LOCK_KIND), ALLOCATABLE :: VTMOP_MOD_DB_BUSY(:)
! Arrays for the LSHEP (ACM TOMS Algorithm 905) surrogate models.
REAL(KIND=R8), ALLOCATABLE :: LSHEP_A(:,:,:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_FVALS(:,:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_RW(:,:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_TABOO(:,:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_SCALE(:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_SHIFT(:)
REAL(KIND=R8), ALLOCATABLE :: LSHEP_XVALS(:,:)
! Module arrays for maintaining the database and weights.
REAL(KIND=R8), ALLOCATABLE :: VTMOP_MOD_DBF(:,:) ! Database of objective values.
REAL(KIND=R8), ALLOCATABLE :: VTMOP_MOD_WEIGHTS(:) ! Scalarization weights.
REAL(KIND=R8), ALLOCATABLE :: VTMOP_MOD_DBX(:,:) ! Database of design points.

! Pointers to module procedures for the objective function and its surrogate.
PROCEDURE(VTMOP_MOD_OBJ_INT), POINTER :: VTMOP_MOD_OBJ_FUNC
PROCEDURE(VTMOP_MOD_SEVAL_INT), POINTER :: VTMOP_MOD_SURROGATES

! The THREADPRIVATE list specifies a list of public variables that will be
! initialized PRIVATE to each thread and persist outside of PARALLEL blocks.
!$OMP THREADPRIVATE(VTMOP_MOD_WEIGHTS)

! Interfaces for external procedures.
INTERFACE
   ! DELAUNAYGRAPH interface.
   SUBROUTINE DELAUNAYGRAPH( D, N, PTS, GRAPH, IERR, EPS, IBUDGET, PMODE )
      USE REAL_PRECISION
      ! Input arguments.
      INTEGER, INTENT(IN) :: D, N
      REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:)
      ! Output arguments.
      LOGICAL, INTENT(OUT) :: GRAPH(:,:)
      INTEGER, INTENT(OUT) :: IERR
      ! Optional arguments.
      REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPS
      INTEGER, OPTIONAL, INTENT(IN) :: IBUDGET
      LOGICAL, OPTIONAL, INTENT(IN) :: PMODE
   END SUBROUTINE DELAUNAYGRAPH

   ! Scalarized objective function interface.
   FUNCTION VTMOP_MOD_SCALAR_INT(C, IERR) RESULT(F)
      USE REAL_PRECISION, ONLY : R8
      REAL(KIND=R8), INTENT(IN) :: C(:)
      INTEGER, INTENT(OUT) :: IERR
      REAL(KIND=R8) :: F
   END FUNCTION VTMOP_MOD_SCALAR_INT

   ! Local optimization subroutine interface.
   SUBROUTINE VTMOP_MOD_LOCAL_INT(D, X, LB, UB, OBJ_FUNC, BUDGET, TOL, IERR)
      USE REAL_PRECISION, ONLY : R8
      INTEGER, INTENT(IN) :: D
      REAL(KIND=R8), INTENT(INOUT) :: X(:)
      REAL(KIND=R8), INTENT(IN) :: LB(:)
      REAL(KIND=R8), INTENT(IN) :: UB(:)
      INTERFACE
```

```
      FUNCTION OBJ_FUNC(C, IERR) RESULT(F)
         USE REAL_PRECISION, ONLY : R8
         REAL(KIND=R8), INTENT(IN) :: C(:)
         INTEGER, INTENT(OUT) :: IERR
         REAL(KIND=R8) :: F
      END FUNCTION OBJ_FUNC
   END INTERFACE
   INTEGER, INTENT(IN) :: BUDGET
   REAL(KIND=R8), INTENT(IN) :: TOL
   INTEGER, INTENT(OUT) :: IERR
END SUBROUTINE VTMOP_MOD_LOCAL_INT

! Objective subroutine interface.
SUBROUTINE VTMOP_MOD_OBJ_INT(C, V, IERR)
   USE REAL_PRECISION, ONLY : R8
   REAL(KIND=R8), INTENT(IN) :: C(:)
   REAL(KIND=R8), INTENT(OUT) :: V(:)
   INTEGER, INTENT(OUT) :: IERR
END SUBROUTINE VTMOP_MOD_OBJ_INT

! Surrogate function evaluation interface.
SUBROUTINE VTMOP_MOD_SEVAL_INT(C, V, IERR)
   USE REAL_PRECISION, ONLY : R8
   ! Parameters.
   REAL(KIND=R8), INTENT(IN) :: C(:)
   REAL(KIND=R8), INTENT(OUT) :: V(:)
   INTEGER, INTENT(OUT) :: IERR
END SUBROUTINE VTMOP_MOD_SEVAL_INT

! Surrogate function fitting interface.
SUBROUTINE VTMOP_MOD_SFIT_INT(D, P, N, X_VALS, Y_VALS, FIRST, PARALLEL, &
                              DES_TOL, SCALE_FACT, SHIFT_FACT, IERR)
   USE REAL_PRECISION, ONLY : R8
   ! Parameters.
   INTEGER, INTENT(IN) :: D
   INTEGER, INTENT(IN) :: P
   INTEGER, INTENT(IN) :: N
   REAL(KIND=R8), INTENT(IN) :: X_VALS(:,:)
   REAL(KIND=R8), INTENT(IN) :: Y_VALS(:,:)
   LOGICAL, INTENT(IN) :: FIRST
   LOGICAL, INTENT(IN) :: PARALLEL
   REAL(KIND=R8), INTENT(IN) :: DES_TOL
   REAL(KIND=R8), INTENT(IN) :: SCALE_FACT(:)
   REAL(KIND=R8), INTENT(IN) :: SHIFT_FACT(:)
   INTEGER, INTENT(OUT) :: IERR
END SUBROUTINE VTMOP_MOD_SFIT_INT
END INTERFACE

CONTAINS

! The following public subroutines are referenced by the driver VTMOP_SOLVE
! program and could be used individually by an advanced user for a
! return-to-caller interface.

SUBROUTINE VTMOP_INIT( VTMOP, D, P, LB, UB, IERR, LOPT_BUDGET, DECAY,     &
                       DES_TOL, EPS, EPSW, OBJ_TOL, MIN_RADF, TRUST_RADF, &
                       LOCAL_OPT, FIT_SURROGATES, EVAL_SURROGATES, PMODE, &
                       ICHKPT )
! This subroutine initializes a VTMOP object for tracking the adaptive
! weighting scheme described in
!
! Deshpande, Shubhangi, Layne T. Watson, and Robert A. Canfield.
! "Multiobjective optimization using an adaptive weighting scheme."
! Optimization Methods and Software 31.1 (2016): 110-133.
!
!
! On input:
!
! D is the dimension of the design space.
!
! P is the dimension of the objective space.
!
! LB(1:D) is the real vector of lower bound constraints for the
!    D design variables.
!
! UB(1:D) is the real vector of upper bound constraints for the
!    D design variables.
!
!
! On output:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem.
!
! IERR is an integer error flag.
!
! Hundreds digit:
!  000 : Normal output. Successful initialization of VTMOP object.
!
!  1xx : Errors detected.
!   Tens digit:
!     11x : The input parameters contained illegal dimensions or values.
```

```
!       Ones digit:
!         110 : D (design dimension) must be a positive integer.
!         111 : P (objective dimension) must be at least two.
!         112 : The lead dimension of LB(:) must match D.
!         113 : The lead dimension of UB(:) must match D.
!         114 : LB(:) must be elementwise strictly less than UB(:) - DES_TOL.
!     12x : The optional dummy arguments contained illegal values.
!       Ones digit:
!         123 : LOPT_BUDGET must be positive.
!         124 : DECAY must be in the range (EPS, 1-EPS).
!         125 : TRUST_RADF must be larger than or equal to MIN_RADF.
!         128 : If either FIT_SURROGATES or EVAL_SURROGATES are present,
!               then both must be present.
!     13x : A memory allocation error has occurred.
!       Ones digit:
!         130 : A memory allocation error occurred.
!         131 : A memory deallocation error occurred.
!
!  9xx : A checkpointing error has occurred.
!    901 : WARNING: the VTMOP object was successfully recovered from the
!          checkpoint but does not match the input data.
!
!  The following error codes are returned by VTMOP_CHKPT_NEW. Further details
!  can be found in the header for VTMOP_CHKPT_NEW.
!    91x : The VTMOP passed to the checkpoint was invalid.
!    92x : Error creating the checkpoint file.
!
!  The following error codes are returned by VTMOP_CHKPT_RECOVER, when
!  VTMOP_INIT is called in recovery mode. Further details can be found in
!  the header for VTMOP_CHKPT_RECOVER.
!    95x : Error reading data from the checkpoint file.
!    96x : A memory management error occurred during recovery.
!
!
! Optional input arguments.
!
! LOPT_BUDGET is an integer input, which specifies the budget for the
!    local optimization subroutine. The default value for LOPT_BUDGET is
!    2500 surrogate evaluations. Note that this value is not saved during
!    checkpointing, and must be reset by the user when recovery mode is
!    active, whenever a non-default value is desired.
!
! DECAY is a real input specifying the decay rate for the local
!    trust region (LTR) radius. This value affects how many times an
!    isolated point can be the center of a LTR before it is discarded.
!    By default, DECAY = 0.5.
!
! DES_TOL is the tolerance for the design space. A design point that
!    is within DES_TOL of an evaluated design point will not be reevaluated.
!    The default value for DES_TOL is the square-root of the working precision
!    EPS. Note that any value that is smaller than the working precsion EPS
!    will be ignored and EPS will be used.
!
! EPS is a real input, which specifies the working precision of the
!    machine. The default value for EPS is SQRT(EPSILON), where EPSILON
!    is the unit roundoff. Note that if the value supplied is smaller than
!    the default value then the default value will be used.
!
! EPSW is a small positive number, which is used as the fudge factor for
!    zero-valued weights. A zero-valued weight does not guarantee Pareto
!    optimality. Therefore, all zero weights are set to EPSW. The appropriate
!    value of EPSW is problem dependent. By default, EPSW is the fourth root
!    of EPSILON (the unit roundoff). Note that any value that is smaller
!    than SQRT(EPSILON) is ignored and SQRT(EPSILON) will be used.
!
! OBJ_TOL is the tolerance for the objective space. An objective point
!    that is within OBJ_TOL of being dominated by another objective point
!    will be treated as such. The default value of OBJ_TOL is the
!    square-root of EPS. Note that any value that is smaller than the
!    working precsion EPS will be ignored and EPS will be used.
!
! MIN_RADF is the smallest value for the fraction r defining the trust region
!    box dimensions r * (UB - LB), before an isolated point is abandoned.
!    By default, MIN_RADF = 0.1 * TRUST_RADF, and is also set to this default
!    value if it is less than DES_TOL. After MIN_RADF and TRUST_RADF are set,
!    MIN_RADF < TRUST_RADF must hold.
!
! TRUST_RADF defines the initial trust region centered at an isolated
!    point X as [X - TRUST_RADF * (UB - LB), X + TRUST_RADF * (UB - LB)]
!    intersected with [LB, UB].  By default, TRUST_RADF = 0.2, and is also set
!    to this value if the value given is outside the interval
!    (DES_TOL, 1 - DES_TOL).
!
! LOCAL_OPT is a SUBROUTINE, whose interface matches that of
!    VTMOP_MOD_LOCAL_INT. LOCAL_OPT is used to optimize the surrogate model.
!    The default value for LOCAL_OPT is GPSMADS, a lightweight Fortran
!    implementation of the algorithm GPS MADS from NOMAD (ACM TOMS Alg. 909).
!    Note that this value is not saved during checkpointing, and must be
!    reset by the user when recovery mode is active, whenever a non-default
!    value is desired.
!
! FIT_SURROGATES is a module subroutine that fits P surrogate models, by
```

```
!    setting variables in its module. The interface for FIT_SURROGATES
!    must match that of VTMOP_MOD_SFIT_INT. By default, FIT_SURROGATES is
!    LSHEP_FIT. Note that this value is not saved during checkpointing, and
!    must be reset by the user when recovery mode is active, whenever a
!    non-default value is desired.
!
! EVAL_SURROGATES is a module subroutine that evaluates the P surrogate
!    models fit by FIT_SURROGATES. The interface for EVAL_SURROGATES must
!    match that of VTMOP_MOD_SEVAL_INT. By default, EVAL_SURROGATES is
!    LSHEP_EVAL. Note that this value is not saved during checkpointing, and
!    must be reset by the user when recovery mode is active, whenever a
!    non-default value is desired.
!
! PMODE is a logical input that specifies whether or not iteration tasks
!    should be performed in parallel. By default, PMODE = .FALSE. Note
!    that this value is not saved during checkpointing, and must be reset
!    by the user when recovery mode is active, whenever a non-default value
!    is desired.
!
! ICHKPT is an integer that specifies the checkpointing status. The
!    checkpoint file and checkpoint unit are "vtmop.chkpt" and 10 by
!    default, but can be adjusted by setting the module variables
!    VTMOP_CHKPTFILE and VTMOP_CHKPTUNIT. Possible values are:
!
!    ICHKPT = 0 : No checkpointing (default setting).
!    ICHKPT < 0 : Recover from the last checkpoint.
!    ICHKPT > 0 : Begin a new checkpoint file.
!
! In recovery mode the inputs D, P, LB, and UB are still referenced
!    (for sanity checks). Also, the procedure arguments are still
!    needed to recover the procedure settings. No other optional
!    arguments are referenced.
!
IMPLICIT NONE
! Input parameters.
INTEGER, INTENT(IN) :: D ! Dimension of design space.
INTEGER, INTENT(IN) :: P ! Dimension of objective space.
REAL(KIND=R8), INTENT(IN) :: LB(:) ! Lower bound constraints.
REAL(KIND=R8), INTENT(IN) :: UB(:) ! Upper bound constraints.
! Output parameters.
TYPE(VTMOP_TYPE), INTENT(OUT) :: VTMOP ! Data struct containing problem info.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Optional parameters.
INTEGER, OPTIONAL, INTENT(IN) :: LOPT_BUDGET ! Local optimizer budget.
LOGICAL, OPTIONAL, INTENT(IN) :: PMODE ! Parallel execution mode.
INTEGER, OPTIONAL, INTENT(IN) :: ICHKPT ! Checkpointing mode.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DECAY ! Decay rate for LTR.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DES_TOL ! Design space tolerance.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPS ! Working precision.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPSW ! Fudge factor for zero weights.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: OBJ_TOL ! Objective space tolerance.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: MIN_RADF ! Minimum LTR radius fraction.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: TRUST_RADF ! Initial LTR radius fraction.
! Optional procedure arguments.
! Locally convergent optimization procedure for solving surrogate problem.
PROCEDURE(VTMOP_MOD_LOCAL_INT), OPTIONAL :: LOCAL_OPT
! Procedure for fitting the surrogate models.
PROCEDURE(VTMOP_MOD_SFIT_INT), OPTIONAL :: FIT_SURROGATES
! Procedure for evaluating the surrogate models.
PROCEDURE(VTMOP_MOD_SEVAL_INT), OPTIONAL :: EVAL_SURROGATES

! Local copy of optional variable.
INTEGER :: ICHKPTL
! External BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2

! Check for illegal input dimensions and values.
IF (D < 1) THEN ! Illegal design space dimension.
   IERR = 110; RETURN; END IF
IF (P < 2) THEN ! Illegal objective space dimension.
   IERR = 111; RETURN; END IF
IF (SIZE(LB,1) .NE. D) THEN ! Lower bounds dimension must match D.
   IERR = 112; RETURN; END IF
IF (SIZE(UB,1) .NE. D) THEN ! Upper bounds dimension must match D.
   IERR = 113; RETURN; END IF
! Get optional inputs.
ICHKPTL = 0
IF (PRESENT(ICHKPT)) ICHKPTL = ICHKPT

! If in checkpoint recovery mode, read the VTMOP object in.
IF (ICHKPTL < 0) THEN
   ! Load problem status from last checkpoint.
   CALL VTMOP_CHKPT_RECOVER(VTMOP, IERR)
   IF (IERR .NE. 0) RETURN
   ! Perform the final sanity check.
   IF ( (VTMOP%D .NE. D) .OR. (VTMOP%P .NE. P) .OR. &
        ANY(VTMOP%LB(:) .NE. LB(:)) .OR. ANY(VTMOP%UB(:) .NE. UB(:)) ) THEN
      IERR = 901; END IF
   ! Set the checkpointing flag.
   VTMOP%CHKPT = .TRUE.

! Otherwise, perform a normal initialization.
```

```
ELSE
   ! Initialize the VTMOP structure to maintain the status of the problem.
   VTMOP%D = D
   VTMOP%P = P
   VTMOP%LCLIST = 20
   ALLOCATE(VTMOP%LB(D), VTMOP%UB(D), VTMOP%CLIST(D+1,VTMOP%LCLIST), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 130; RETURN; END IF
   VTMOP%ITERATE = 0
   VTMOP%LB(:) = LB(:)
   VTMOP%UB(:) = UB(:)

   ! Check for optional inputs.
   ! Initialize the working precision.
   VTMOP%EPS = SQRT(EPSILON(0.0_R8))
   IF(PRESENT(EPS)) THEN
      ! The default value of EPS cannot be decreased. Ignore such inputs.
      IF(EPS > VTMOP%EPS) VTMOP%EPS = EPS
   END IF
   ! Initialize the fudge factor for zero weights.
   VTMOP%EPSW = EPSILON(0.0_R8) ** 0.25_R8
   IF(PRESENT(EPSW)) THEN
      ! Ignore EPSW if it is less than the square-root of the machine EPSILON.
      IF(EPSW .GE. SQRT(EPSILON(0.0_R8))) VTMOP%EPSW = EPSW
   END IF
   ! Initialize the design space and objective space tolerances.
   VTMOP%DES_TOL = SQRT(VTMOP%EPS)
   IF (PRESENT(DES_TOL)) THEN
      IF(DES_TOL > VTMOP%EPS) THEN
         VTMOP%DES_TOL = DES_TOL
      ELSE
         VTMOP%DES_TOL = VTMOP%EPS
      END IF
   END IF
   VTMOP%OBJ_TOL = SQRT(VTMOP%EPS)
   IF (PRESENT(OBJ_TOL)) THEN
      IF(OBJ_TOL > VTMOP%EPS) THEN
         VTMOP%OBJ_TOL = OBJ_TOL
      ELSE
         VTMOP%OBJ_TOL = VTMOP%EPS
      END IF
   END IF
   ! Initialize the decay rate.
   VTMOP%DECAY = 0.5_R8
   IF(PRESENT(DECAY)) THEN
      ! The decay rate must be between EPS and 1-EPS.
      IF(DECAY > 1.0_R8 - VTMOP%EPS .OR. DECAY < VTMOP%EPS) THEN
         IERR = 124; RETURN; END IF
      VTMOP%DECAY = DECAY
   END IF
   ! Initialize the LTR radius fraction and minimum LTR radius fraction.
   VTMOP%TRUST_RADF = 0.2_R8
   IF(PRESENT(TRUST_RADF)) THEN
      ! The LTR radius fraction must be greater than the design space tolerance.
      IF(TRUST_RADF .GE. VTMOP%DES_TOL .AND. TRUST_RADF < 1.0_R8) THEN
         VTMOP%TRUST_RADF = TRUST_RADF; END IF
   END IF
   ! The minimum LTR radius fraction must be greater than the design
   ! space tolerance. By default, tolerate decay down to 10% of the
   ! initial LTR radius.
   VTMOP%MIN_RADF = MAX(0.1_R8 * VTMOP%TRUST_RADF, VTMOP%DES_TOL)
   IF(PRESENT(MIN_RADF)) THEN
      IF(MIN_RADF .GE. VTMOP%DES_TOL) VTMOP%MIN_RADF = MIN_RADF
   END IF
   ! Check that the size of the LTR radius is appropriate.
   IF(VTMOP%MIN_RADF > VTMOP%TRUST_RADF) THEN
      IERR = 125; RETURN; END IF
   ! Lower bounds must be elementwise strictly less than upper bounds.
   ! Use the design space tolerance to check.
   IF(ANY(LB(:) .GE. UB(:) - VTMOP%DES_TOL)) THEN
      IERR = 114; RETURN; END IF
   ! Initialize the checkpointing mode.
   VTMOP%CHKPT = .FALSE.
   ! If ICHKPT > 0, initialize the checkpoint file and activate checkpointing.
   ! Check whether checkpointing is enabled.
   IF (ICHKPTL .NE. 0) THEN
      ! Initialize the checkpoint file and save the initialized VTMOP.
      CALL VTMOP_CHKPT_NEW(VTMOP, IERR)
      IF (IERR .NE. 0) RETURN
      ! Activate checkpointing mode.
      VTMOP%CHKPT = .TRUE.
   END IF
END IF

! The remaining optional inputs are not saved when checkpointing is active.
! Therefore, if non-default values are desired, these values must be re-set
! by the user, even in recovery mode.

! Initialize the local optimization budget.
VTMOP%LOPT_BUDGET = 2500
IF(PRESENT(LOPT_BUDGET)) THEN
   ! The budget must be at least 1, return an error for nonpositive values.
```

```
   IF(LOPT_BUDGET < 1) THEN
      IERR = 123; RETURN; END IF
   VTMOP%LOPT_BUDGET = LOPT_BUDGET
END IF
! Set the parallel execution mode.
VTMOP%PMODE = .FALSE.
IF(PRESENT(PMODE)) VTMOP%PMODE = PMODE
! Set the procedure arguments.
VTMOP%LOCAL_OPT => GPSMADS ! Default optimizer is GPSMADS.
IF(PRESENT(LOCAL_OPT)) THEN
   VTMOP%LOCAL_OPT => LOCAL_OPT; END IF
VTMOP%FIT_SURROGATES => LSHEP_FIT ! Default fit is LSHEP_FIT.
IF(PRESENT(FIT_SURROGATES)) THEN
   IF (.NOT. PRESENT(EVAL_SURROGATES)) THEN
      IERR = 128; RETURN; END IF
   VTMOP%FIT_SURROGATES => FIT_SURROGATES
END IF
VTMOP%EVAL_SURROGATES => LSHEP_EVAL ! Default evaluation is LSHEP_EVAL.
IF(PRESENT(EVAL_SURROGATES)) THEN
   IF (.NOT. PRESENT(FIT_SURROGATES)) THEN
      IERR = 128; RETURN; END IF
   VTMOP%EVAL_SURROGATES => EVAL_SURROGATES
END IF

RETURN
END SUBROUTINE VTMOP_INIT

SUBROUTINE VTMOP_LTR( VTMOP, DES_PTS, OBJ_PTS, LTR_LB, LTR_UB, IERR )
! This subroutine identifies the most isolated point, builds a local
! trust region (LTR), and chooses the adaptive weights, as described in
!
! Deshpande, Shubhangi, Layne T. Watson, and Robert A. Canfield.
! "Multiobjective optimization using an adaptive weighting scheme."
! Optimization Methods and Software 31.1 (2016): 110-133.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem. VTMOP is created using VTMOP_INIT.
!
! DES_PTS(1:D,1:N) is a real matrix of all design points in the feasible
!    design space [LB, UB], stored by column. The second dimension of
!    DES_PTS(:,:) (N) is assumed based on the shape and must be at least D+1
!    to build an accurate surrogate model. In the special case of the zeroth
!    iteration, DES_PTS need not be allocated.
!
! OBJ_PTS(1:P,1:N) is a real matrix of objective values corresponding
!    to the design points in DES_PTS(:,:), stored by column: for cost
!    function F, OBJ_PTS(:,I) = F(DES_PTS(:,I)). In the special case of the
!    zeroth iteration, OBJ_PTS need not be allocated.
!
!
! On output:
!
! LTR_LB(1:D) is a real array of lower bounds for the LTR.
!
! LTR_UB(1:D) is a real array of upper bounds for the LTR.
!
! IERR is an integer error flag.
!
! Hundreds digit:
!  0xx : Normal output.
!    Ones digit:
!      000 : Successfully constructed a new LTR and selected adaptive weights.
!      003 : Maximal accuracy has already been achieved, no isolated points
!            can be further refined for the problem tolerance.
!
!  2xx : Error detected in input, or during VTMOP_INIT (initialization) code.
!   Tens digit:
!     21x : The input parameters contained illegal dimensions or values.
!       Ones digit:
!         210 : The VTMOP object appears to be uninitialized.
!         211 : The VTMOP object is initialized, but its dimensions either
!               do not agree or contain illegal values. This is likely the
!               result of an undetected segmentation fault.
!         212 : The lead dimension of LTR_LB(:) must match the design
!               dimension D, stored in VTMOP.
!         213 : The lead dimension of LTR_UB(:) must match the design
!               dimension D, stored in VTMOP.
!         214 : The lead dimension of DES_PTS(:,:) must match D.
!         215 : The lead dimension of OBJ_PTS(:,:) must match P.
!         216 : The second dimensions of DES_PTS and OBJ_PTS must match.
!     22x : A memory error occurred while managing the dynamic memory in VTMOP.
!         220 : A memory allocation error while copying the history.
!         221 : A memory deallocation error occurred while freeing temp arrays.
!         222 : A memory allocation error while allocating the adaptive weights.
!     23x : A memory error occurred while managing the local arrays.
!       Ones digit:
!         230 : A memory allocation error occurred.
!         231 : A memory deallocation error occurred.
!     240 : Detected a set of unused adaptive weights. This subroutine
```

```
!           may have been called out of sequence.
!
!  5xx : Error thrown by DELAUNAYSPARSE.
!    Tens and ones digits carry the exact error code from DELAUNAYSPARSE,
!    passed by the subroutine DELAUNAYGRAPH.
!
!  9xx : A checkpointing error was thrown.
!    91x : The VTMOP passed to the checkpoint was invalid.
!    93x : Error writing iteration information to the checkpoint file.
!
USE IEEE_ARITHMETIC
IMPLICIT NONE
! Input parameters.
TYPE(VTMOP_TYPE), INTENT(INOUT) :: VTMOP ! Data struct containing problem info.
REAL(KIND=R8), INTENT(IN) :: DES_PTS(:,:) ! Table of precomputed design pts.
REAL(KIND=R8), INTENT(IN) :: OBJ_PTS(:,:) ! Table of objective values.
! Output parameters.
REAL(KIND=R8), INTENT(OUT) :: LTR_LB(:) ! LTR lower bound constraints.
REAL(KIND=R8), INTENT(OUT) :: LTR_UB(:) ! LTR upper bound constraints.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Problem dimensions.
INTEGER :: D ! Design space dimension.
INTEGER :: M ! Cardinality of the current Pareto set.
INTEGER :: N ! Number of design/objective points in the database.
INTEGER :: P ! Number of objectives.
! Local variables.
LOGICAL :: ACCEPT ! Acceptance condition for new center point.
LOGICAL :: FOUND ! Indicate whether the box center was found.
INTEGER :: I, J, K ! Loop indexing / temp variables.
INTEGER :: MAXIND ! The index of the most isolated point.
REAL(KIND=R8) :: BOX(VTMOP%D+1) ! Center and radius of next LTR.
REAL(KIND=R8) :: MINVAL_P ! Minimum value taken by the Pth objective.
REAL(KIND=R8) :: TMP(VTMOP%P-1) ! Temporary array of length P-1.
! Local dynamic arrays.
LOGICAL, ALLOCATABLE :: DELGRAPH(:,:) ! Delaunay graph.
REAL(KIND=R8), ALLOCATABLE :: CLIST_TMP(:,:) ! Temp array for expanding CLIST.
REAL(KIND=R8), ALLOCATABLE :: DISCREP(:) ! List of star discrepancies.
REAL(KIND=R8), ALLOCATABLE :: EFFICIENT_SET(:,:) ! Efficient point set.
REAL(KIND=R8), ALLOCATABLE :: HOMOGENEOUS_PF(:,:) ! Homogeneous Pareto front.
REAL(KIND=R8), ALLOCATABLE :: PARETO_SET(:,:) ! Pareto front.
! External BLAS procedures.
REAL(KIND=R8), EXTERNAL :: DNRM2 ! Euclidean distance (BLAS).

! Retrieve problem dimensions from input data.
D = VTMOP%D; P = VTMOP%P; N = SIZE(DES_PTS,2)
! Check for illegal problem dimensions.
IF ( (.NOT. ALLOCATED(VTMOP%LB)) .OR. (.NOT. ALLOCATED(VTMOP%UB)) &
     .OR. (.NOT. ALLOCATED(VTMOP%CLIST)) ) THEN
   IERR = 210; RETURN; END IF
IF ((D < 1) .OR. (P < 2) .OR. (SIZE(VTMOP%LB,1) .NE. D) .OR. &
    (SIZE(VTMOP%UB,1) .NE. D)) THEN
   IERR = 211; RETURN; END IF
IF (SIZE(LTR_LB,1) .NE. D) THEN ! Lower bounds dimension does not match D.
   IERR = 212; RETURN; END IF
IF (SIZE(LTR_UB,1) .NE. D) THEN ! Upper bounds dimension does not match D.
   IERR = 213; RETURN; END IF
! If the adaptive weights array is already allocated, then VTMOP_OPT has not
! been called since the last call to VTMOP_LTR.
IF (ALLOCATED(VTMOP%WEIGHTS)) THEN
   IERR = 240; RETURN; END IF

! In the zeroth iteration, use static weights on the entire design space.
IF (VTMOP%ITERATE .EQ. 0) THEN
   ! Set the LTR bounds.
   LTR_LB(:) = VTMOP%LB(:)
   LTR_UB(:) = VTMOP%UB(:)
   ! Allocate the initial weights.
   ALLOCATE(VTMOP%WEIGHTS(P,P+1), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 222; RETURN; END IF
   ! Set the individual weights.
   DO I = 1, P
      ! Set zero-valued weights to VTMOP%EPS to avoid pathological cases.
      VTMOP%WEIGHTS(:,I) = VTMOP%EPSW
      VTMOP%WEIGHTS(I,I) = 1.0_R8 - VTMOP%EPSW*REAL(P-1,KIND=R8)
   END DO
   ! Set the adaptive weights.
   VTMOP%WEIGHTS(:,P+1) = 1.0_R8 / REAL(P, KIND=R8)

! Otherwise, for the Kth iteration (K > 0), perform a general iteration.
ELSE
   ! Check that the dimensions of DES_PTS and OBJ_PTS match.
   IF (SIZE(DES_PTS,1) .NE. D) THEN
      IERR = 214; RETURN; END IF
   IF (SIZE(OBJ_PTS,1) .NE. P) THEN
      IERR = 215; RETURN; END IF
   IF (SIZE(OBJ_PTS,2) .NE. N) THEN
      IERR = 216; RETURN; END IF

   ! If CLIST is at capacity, then reallocate CLIST.
   IF (VTMOP%ITERATE > VTMOP%LCLIST) THEN
      ! Allocate the temporary array to copy CLIST.
```

```
   ALLOCATE(CLIST_TMP(D+1,VTMOP%LCLIST), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 220; RETURN; END IF
   CLIST_TMP = VTMOP%CLIST
   ! Reallocate CLIST to twice its current size.
   DEALLOCATE(VTMOP%CLIST, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 221; RETURN; END IF
   ALLOCATE(VTMOP%CLIST(D+1,VTMOP%LCLIST*2), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 220; RETURN; END IF
   ! Restore values back into CLIST and free the temporary array.
   VTMOP%CLIST(:,1:VTMOP%LCLIST) = CLIST_TMP(:,:)
   DEALLOCATE(CLIST_TMP, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 221; RETURN; END IF
   ! Update the size of VTMOP%LCLIST.
   VTMOP%LCLIST = VTMOP%LCLIST * 2
END IF

! Allocate the dynamic arrays for storing both copies of the Pareto front.
ALLOCATE(PARETO_SET(P,N), EFFICIENT_SET(D,N), HOMOGENEOUS_PF(P-1,N), &
         STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 230; RETURN; END IF
! Compute the minimum value obtained by the Pth objective. This value is
! used to shift the points in the Pth dimension, so that they are above
! the hyperplane X(P) = 1. This prevents division by zero.
MINVAL_P = MINVAL(OBJ_PTS(P,:)) - 1.0_R8
! Get the current Pareto front (in both the objective space and in the
! homogeneous coordinates).
M = 0 ! Count the cardinality of the Pareto front.
OUTER : DO I = 1, N
   ! Skip NAN values.
   IF (ANY(IEEE_IS_NAN(OBJ_PTS(:,I)))) CYCLE OUTER
   INNER : DO J = 1, N
      ! Skip index I.
      IF (I .EQ. J) CYCLE INNER
      ! Compute the homogeneous coordinates.
      TMP(:) = (OBJ_PTS(1:P-1,I) / (OBJ_PTS(P,I) - MINVAL_P)) &
               - (OBJ_PTS(1:P-1,J) / (OBJ_PTS(P,J) - MINVAL_P))
      ! Check whether OBJ_PTS(:,I) and OBJ_PTS(:,J) are equal in the
      ! homogeneous coordinate system (up to the working precision).
      IF (DNRM2(P-1, TMP, 1) < VTMOP%OBJ_TOL) THEN
         IF ( I > J ) THEN
            CYCLE OUTER ! Only store the first occurrence of a duplicate.
         ELSE
            CYCLE INNER ! This is the first occurrence of a duplicate.
         END IF
      END IF
      ! Check whether OBJ_PTS(:,J) dominates OBJ_PTS(:,I).
      IF (ALL(OBJ_PTS(:,J) .LE. OBJ_PTS(:,I) + VTMOP%OBJ_TOL)) CYCLE OUTER
   END DO INNER
   ! Increment the counter and update both Pareto front arrays.
   M = M + 1
   PARETO_SET(:,M) = OBJ_PTS(:,I)
   EFFICIENT_SET(:,M) = DES_PTS(:,I)
   HOMOGENEOUS_PF(:,M) = OBJ_PTS(1:P-1,I) / (OBJ_PTS(P,I) - MINVAL_P)
END DO OUTER

! Allocate the remaining dynamic arrays.
ALLOCATE(DELGRAPH(M,M), DISCREP(M), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 230; RETURN; END IF

! Use DELAUNAYGRAPH to compute the Delaunay graph (serially or in parallel).
CALL DELAUNAYGRAPH(P-1, M, HOMOGENEOUS_PF(:,1:M), DELGRAPH, IERR, &
                   EPS=VTMOP%EPS, PMODE=VTMOP%PMODE)
IF (IERR < 10) THEN ! Normal execution.
   IERR = 0
ELSE ! An irrecoverable error occurred.
   IERR = IERR + 500
   RETURN
END IF

! Identify the most isolated point using the star discrepancy.
DO I = 1, M
   ! Compute the star discrepancy at index I using DELGRAPH(:,I).
   DISCREP(I) = 0.0_R8
   K = 0
   DO J = 1, M
      IF(DELGRAPH(J,I)) THEN
         DISCREP(I) = DISCREP(I) + &
                      DNRM2(P, PARETO_SET(:,J)-PARETO_SET(:,I), 1)
         K = K + 1
      END IF
   END DO
   IF (K > 0) THEN
      ! In general, point at index I has at least one Delaunay neighbor.
      DISCREP(I) = DISCREP(I) / REAL(K, KIND=R8)
   ELSE
      ! In rare cases, the point at index I has no neighbors. This
```

```
      ! can only occur when the Pareto front consists of a single point.
      DISCREP(I) = 1.0_R8
   END IF
END DO

! Loop until an acceptable center is found.
ACCEPT = .FALSE.
DO WHILE(.NOT. ACCEPT)
   ! Identify the largest discrepancy, except negative/zero values.
   MAXIND = MAXLOC(DISCREP, DIM=1, MASK=(DISCREP > VTMOP%EPS))
   ! If no values with positive discrepancy remain, terminate.
   IF (MAXIND .EQ. 0) EXIT
   ! Otherwise, set BOX(1:D) to the corresponding design point.
   BOX(1:D) = EFFICIENT_SET(:,MAXIND)
   ! Check whether the current design point has been used before.
   FOUND = .FALSE.
   DO I = VTMOP%ITERATE-1, 1, -1
      IF ( DNRM2(D, VTMOP%CLIST(1:D,I) - BOX(1:D), 1) < VTMOP%DES_TOL ) THEN
         FOUND = .TRUE.
         EXIT
      END IF
   END DO
   ! A previous entry in VTMOP%CLIST(:,:) matches.
   IF (FOUND) THEN
      ! Copy the item and decay the LTR radius.
      BOX = VTMOP%CLIST(:,I)
      BOX(D+1) = BOX(D+1) * VTMOP%DECAY
      ! The minimum tolerance has not yet been exceeded.
      IF (BOX(D+1) > VTMOP%MIN_RADF) THEN
         ! Append BOX(:) to CLIST(:,:).
         VTMOP%CLIST(:,VTMOP%ITERATE) = BOX(:)
         ACCEPT = .TRUE.
      ! The minimum tolerance has been exceeded, set the
      ! discrepancy to a negative number.
      ELSE
         DISCREP(MAXIND) = -1.0_R8
      END IF
   ! No match found.
   ELSE
      ! Append BOX(:) to CLIST(:,:).
      BOX(D+1) = VTMOP%TRUST_RADF
      VTMOP%CLIST(:,VTMOP%ITERATE) = BOX(:)
      ACCEPT = .TRUE.
   END IF
END DO

! If no point was accepted, terminate. It must be that the Pareto front
! has been approximated to the maximum tolerance.
IF (.NOT. ACCEPT) THEN
   IERR = 3;
   RETURN
END IF

! Build the LTR. It is the intersection over the current box and the
! bound constraints.
DO I = 1, D
   LTR_LB(I) = MAX(BOX(I)-BOX(D+1)*(VTMOP%UB(I)-VTMOP%LB(I)), VTMOP%LB(I))
   LTR_UB(I) = MIN(BOX(I)+BOX(D+1)*(VTMOP%UB(I)-VTMOP%LB(I)), VTMOP%UB(I))
END DO

! Count the number of Delaunay neighbors of PARETO_SET(:,MAXIND) using
! DELGRAPH(:,MAXIND).
K = 0
DO I = 1, M
   IF(DELGRAPH(I, MAXIND)) K = K + 1
END DO
! Construct the adaptive weights. First, consider the special case where
! there are insufficiently many Delaunay neighbors.
IF (K < 1) THEN
   ! Allocate the adaptive weights.
   ALLOCATE(VTMOP%WEIGHTS(P,P+1), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 222; RETURN; END IF
   ! Set the individual weights.
   DO I = 1, P
      ! Set zero-valued weights to VTMOP%EPSW to avoid pathological cases.
      VTMOP%WEIGHTS(:,I) = VTMOP%EPSW
      VTMOP%WEIGHTS(I,I) = 1.0_R8 - VTMOP%EPSW*REAL(P-1,KIND=R8)
   END DO
   ! Set the adaptive weights.
   VTMOP%WEIGHTS(:,P+1) = 1.0_R8 / REAL(P, KIND=R8)

! In the general case, use the Delaunay neighborhood.
ELSE
   ! Allocate the adaptive weights.
   ALLOCATE(VTMOP%WEIGHTS(P,P+K), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 222; RETURN; END IF
   ! Set the individual weights.
   DO I = 1, P
      ! Set zero-valued weights to VTMOP%EPSW to avoid pathological cases.
      VTMOP%WEIGHTS(:,I) = VTMOP%EPSW
```

```
         VTMOP%WEIGHTS(I,I) = 1.0_R8 - VTMOP%EPSW*REAL(P-1,KIND=R8)
      END DO
      ! Set the adaptive weights.
      K = 1
      DO I = 1, M
         ! Check that I and MAXIND are Delaunay neighbors.
         IF (DELGRAPH(I,MAXIND)) THEN
            ! Set the adaptive weights.
            VTMOP%WEIGHTS(:,P+K) = ABS(PARETO_SET(:,I) - PARETO_SET(:,MAXIND))
            ! Invert when greater than or equal to zero.
            DO J = 1, P
               IF (VTMOP%WEIGHTS(J,P+K) < VTMOP%EPS) THEN
                  VTMOP%WEIGHTS(J,P+K) = VTMOP%EPSW
               ELSE
                  VTMOP%WEIGHTS(J,P+K) = 1.0_R8 / VTMOP%WEIGHTS(J,P+K)
               END IF
            END DO
            ! Normalize to make VTMOP%WEIGHTS(:,P+K) convex.
            VTMOP%WEIGHTS(:,P+K) = VTMOP%WEIGHTS(:,P+K) / &
                                   SUM(VTMOP%WEIGHTS(:,P+K))
            K = K + 1
         END IF
      END DO
   END IF
   ! Free heap memory.
   DEALLOCATE(DELGRAPH, DISCREP, PARETO_SET, HOMOGENEOUS_PF, STAT=IERR)
   IF (IERR .NE. 0) IERR = 231
END IF

! If CHKPT is set, then save to the checkpoint file.
IF (VTMOP%CHKPT) THEN
   ! Save the updated VTMOP object to the checkpoint file.
   CALL VTMOP_CHKPT(VTMOP, IERR)
   IF (IERR .NE. 0) RETURN
END IF
RETURN
END SUBROUTINE VTMOP_LTR

SUBROUTINE VTMOP_OPT( VTMOP, LTR_LB, LTR_UB, DES_PTS, OBJ_PTS, CAND_PTS, IERR )
! This subroutine fits and optimizes P surrogate models within the current
! local trust region (LTR), over the adaptive weights in the VTMOP object,
! as described in
!
! Deshpande, Shubhangi, Layne T. Watson, and Robert A. Canfield.
! "Multiobjective optimization using an adaptive weighting scheme."
! Optimization Methods and Software 31.1 (2016): 110-133.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem. VTMOP is created using VTMOP_INIT.
!
! LTR_LB(1:D) is the real vector of lower bounds for the LTR.
!
! LTR_UB(1:D) is the real vector of upper bounds for the LTR.
!
! DES_PTS(1:D,1:N) is a real matrix of all design points in the feasible
!    design space [LB, UB], stored by column. The second dimension of
!    DES_PTS(:,:) (N) is assumed based on the shape and must be at least
!    D+1 to build an accurate surrogate model.
!
! OBJ_PTS(1:P,1:N) is a real matrix of objective values corresponding
!    to the design points in DES_PTS(:,:), stored by column: for
!    cost function F, OBJ_PTS(:,I) = F(DES_PTS(:,I)).
!
! CAND_PTS(:,:) is an ALLOCATABLE real array, which need not be allocated
!    on input. If allocated, any contents of CAND_PTS are lost, and CAND_PTS
!    are reallocated on output.
!
!
! On output:
!
! CAND_PTS(1:D,1:M) is a list of candidate design points to be evaluated
!    before the next iteration of the algorithm. CAND_PTS contains
!    no redundant design points.
!
! IERR is an integer error flag. IERR=0 signifies a successful iteration.
!
! Hundreds digit:
!  000 : Normal output. Successful iteration, and list CAND_PTS obtained.
!
!  3xx : Errors detected.
!   Tens digit:
!     31x : The input parameters contained illegal dimensions or values.
!       Ones digit:
!         310 : Either the LB(:) or UB(:) array is not allocated.
!         311 : Either P or D contains an illegal value, or does not
!               agree with input sizes.
!         312 : The lead dimension of LB(:) must match D.
!         313 : The lead dimension of UB(:) must match D.
!         314 : LB(:) must be elementwise strictly less than UB(:) - TOL.
```

```
!         315 : The lead dimension of DES_PTS(:,:) must match D.
!         316 : The lead dimension of OBJ_PTS(:,:) must match P.
!         317 : The second dimensions of DES_PTS and OBJ_PTS must match.
!     32x : An irregularity was detected in the supplied VTMOP data type.
!       Ones digit:
!         320 : The adaptive weights are not allocated. Check that
!               the last subroutine called was VTMOP_LTR.
!         321 : The adaptive weights array is allocated, but its lead
!               dimension does not match the number of objectives P. This
!               is most likely the result of an undetected segmentation
!               fault.
!         322 : There are too few adaptive weights. This is most likely the
!               result of an undetected segmentation fault.
!         323 : One of the adaptive weights contains a negative value. This
!               is most likely the result of an undetected segmentation
!               fault.
!     33x : A memory error has occurred.
!       Ones digit:
!         330 : A memory allocation error occurred in the local memory.
!         331 : A memory deallocation error occurred in the local memory.
!         332 : A memory deallocation error occurred while freeing the
!               adaptive weights for the next iteration.
!     340 : Too few points were supplied in DES_PTS and OBJ_PTS to construct
!           an accurate surrogate. At least D+1 points are required.
!
!  6xx : Error thrown by FIT_SURROGATES.
!    Tens and ones digits carry the error code from FIT_SURROGATES subroutine.
!    Note, this assumes that FIT_SURROGATES returns an error code less than
!    100.
!
!  7xx : Error thrown by LOCAL_OPT.
!    Tens and ones digits carry the error code from LOCAL_OPT subroutine.
!    Note, this assumes that LOCAL_OPT returns an error code less than 100.
!
!  9xx : A checkpointing error was thrown.
!    91x : The VTMOP passed to the checkpoint was invalid.
!    93x : Error writing iteration information to the checkpoint file.
!
IMPLICIT NONE
! Input parameters.
TYPE(VTMOP_TYPE), INTENT(INOUT) :: VTMOP ! Data struct containing problem info.
REAL(KIND=R8), INTENT(IN) :: LTR_LB(:) ! Lower bound constraints.
REAL(KIND=R8), INTENT(IN) :: LTR_UB(:) ! Upper bound constraints.
REAL(KIND=R8), INTENT(IN) :: DES_PTS(:,:) ! Table of design points.
REAL(KIND=R8), INTENT(IN) :: OBJ_PTS(:,:) ! Table of objective values.
! Output parameters.
REAL(KIND=R8), INTENT(OUT), ALLOCATABLE :: CAND_PTS(:,:) ! Efficient set.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Problem dimensions.
INTEGER :: D ! Design space dimension.
INTEGER :: L ! Number of adaptive weights for the current iteration.
INTEGER :: M ! Number of candidate design points to be returned.
INTEGER :: N ! Number of design/objective points in the database.
INTEGER :: P ! Number of objectives.
! Local variables.
INTEGER :: I, J ! Loop indexing variables.
REAL(KIND=R8) :: MIN_X(VTMOP%D,SIZE(VTMOP%WEIGHTS,2)) ! Potential candidate pts.
! External BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2

! Get problem dimensions from input data.
D = VTMOP%D; P = VTMOP%P; N = SIZE(DES_PTS,2); L = SIZE(VTMOP%WEIGHTS,2)
! Check for illegal problem dimensions.
IF ((.NOT. ALLOCATED(VTMOP%LB)) .OR. (.NOT. ALLOCATED(VTMOP%UB))) THEN
   IERR = 310; RETURN; END IF
IF ((D < 1) .OR. (P < 2) .OR. (SIZE(VTMOP%LB,1) .NE. D) .OR. &
    (SIZE(VTMOP%UB,1) .NE. D)) THEN
   IERR = 311; RETURN; END IF
IF (SIZE(LTR_LB,1) .NE. D) THEN ! Lower bound dimension does not match D.
   IERR = 312; RETURN; END IF
IF (SIZE(LTR_UB,1) .NE. D) THEN ! Upper bound dimension does not match D.
   IERR = 313; RETURN; END IF
IF (ANY(LTR_LB .GE. LTR_UB - VTMOP%DES_TOL) .OR. &
    ANY(LTR_LB < VTMOP%LB - VTMOP%EPS) .OR.      &
    ANY(LTR_UB > VTMOP%UB + VTMOP%EPS)) THEN
   IERR = 314; RETURN; END IF
! Check that the dimensions of DES_PTS and OBJ_PTS match.
IF (SIZE(DES_PTS,1) .NE. D) THEN
   IERR = 315; RETURN; END IF
IF (SIZE(OBJ_PTS,1) .NE. P) THEN
   IERR = 316; RETURN; END IF
IF (SIZE(OBJ_PTS,2) .NE. N) THEN
   IERR = 317; RETURN; END IF
! Check the adaptive VTMOP%WEIGHTS array.
IF(.NOT. ALLOCATED(VTMOP%WEIGHTS)) THEN
   IERR = 320; RETURN; END IF
IF(SIZE(VTMOP%WEIGHTS,1) .NE. P) THEN
   IERR = 321; RETURN; END IF
IF(L < P+1) THEN
   IERR = 322; RETURN; END IF
IF(ANY(VTMOP%WEIGHTS < -VTMOP%EPS)) THEN
   IERR = 323; RETURN; END IF
```

```
! Too few points to fit a surrogate model.
IF (N < D+1) THEN
   IERR = 340; RETURN; END IF

! Fit P surrogate models.
IF (VTMOP%ITERATE .EQ. 0) THEN
   CALL VTMOP%FIT_SURROGATES(D, P, N, DES_PTS, OBJ_PTS, .TRUE., VTMOP%PMODE, &
                             VTMOP%DES_TOL, VTMOP%UB(:)-VTMOP%LB(:),         &
                             VTMOP%LB(:), IERR)
   IF (IERR .NE. 0) THEN
      IERR = IERR + 600; RETURN; END IF
ELSE
   CALL VTMOP%FIT_SURROGATES(D, P, N, DES_PTS, OBJ_PTS, .FALSE., VTMOP%PMODE, &
                             VTMOP%DES_TOL, VTMOP%UB(:)-VTMOP%LB(:),          &
                             VTMOP%LB(:), IERR)
   IF (IERR .NE. 0) THEN
      IERR = IERR + 600; RETURN; END IF
END IF

! Initialize the module variables containing the problem dimensions.
VTMOP_MOD_D = D
VTMOP_MOD_P = P
! Set the module surrogate functions.
VTMOP_MOD_SURROGATES => VTMOP%EVAL_SURROGATES

! Optimize the surrogate models for all adaptive weightings.
!$OMP PARALLEL &
!
! The PRIVATE list specifies uninitialized variables, of which each
! thread has a private copy.
!$OMP& PRIVATE(I), &
!
! The REDUCTION clause specifies a PRIVATE variable that will retain
! some value (i.e., max, min, sum, etc.) upon output.
!$OMP& REDUCTION(MAX:IERR), &
!
! Any variables not explicitly listed above receive the SHARED scope
! by default and are visible across all threads.
!$OMP& DEFAULT(SHARED), &
!
! Only use this level of parallelism if PMODE is .TRUE.
!$OMP& IF(VTMOP%PMODE)
! Initialize the error flag.
IERR = 0
! Allocate the adaptive weights array.
IF(ALLOCATED(VTMOP_MOD_WEIGHTS)) THEN
   DEALLOCATE(VTMOP_MOD_WEIGHTS, STAT=IERR)
   IF (IERR .NE. 0) IERR = 331
END IF
ALLOCATE(VTMOP_MOD_WEIGHTS(P), STAT=IERR)
IF (IERR .NE. 0) IERR = 330
! Optimize L adaptive weightings of the P surrogate models.
!$OMP DO SCHEDULE(STATIC)
DO I = 1, L
   ! If an error occurs, skip to the end.
   IF (IERR .NE. 0) CYCLE
   ! Set the module weights.
   VTMOP_MOD_WEIGHTS(:) = VTMOP%WEIGHTS(:,I)
   ! Call the local optimizer.
   MIN_X(:,I) = (LTR_LB(:) + LTR_UB(:)) / 2.0_R8
   CALL VTMOP%LOCAL_OPT( D, MIN_X(:,I), LTR_LB(:), LTR_UB(:),              &
                         SURROGATE_FUNC, VTMOP%LOPT_BUDGET, VTMOP%DES_TOL, &
                         IERR )
END DO
!$OMP END DO
!$OMP END PARALLEL
IF (IERR .NE. 0) THEN
   IERR = IERR + 700; RETURN; END IF

! Post processing loop to filter out redundant evaluation points.
M = L ! Track the number of distinct candidate points to return.
I = 1 ! Track the current iterate.
FILTER_LOOP : DO WHILE(.TRUE.)
   ! If the end of the reduced list is reached, exit the loop.
   IF (I > M) EXIT FILTER_LOOP
   ! If MIN_X(:,I) is already in DES_PTS(:,:), don't add to CAND_PTS(:,:).
   DO J = 1, SIZE(DES_PTS(:,:), 2)
      IF (DNRM2(D, DES_PTS(:,J)-MIN_X(:,I), 1) < VTMOP%DES_TOL) THEN
         ! Perform a deletion by overwriting the current entry with the last.
         MIN_X(:,I) = MIN_X(:,M)
         M = M - 1
         ! Skip forward.
         CYCLE FILTER_LOOP
      END IF
   END DO
   ! If MIN_X(:,J) = MIN_X(:,I), for J <= I, don't add to CAND_PTS(:,:).
   DO J = 1, I - 1
      IF (DNRM2(D, MIN_X(:,J)-MIN_X(:,I), 1) < VTMOP%DES_TOL) THEN
         ! Perform a deletion by overwriting the current entry with the last.
         MIN_X(:,I) = MIN_X(:,M)
         M = M - 1
         ! Skip forward.
```

```
         CYCLE FILTER_LOOP
      END IF
   END DO
   ! If all the tests passed, advance the iteration index.
   I = I + 1
END DO FILTER_LOOP

! Increment the iteration counter.
VTMOP%ITERATE = VTMOP%ITERATE + 1
! Free CAND_PTS if they are already allocated.
IF(ALLOCATED(CAND_PTS)) THEN
   DEALLOCATE(CAND_PTS, STAT=IERR)
   IF(IERR .NE. 0) THEN
      IERR = 331; RETURN; END IF
END IF
! Allocate and fill CAND_PTS(:,:) to return.
ALLOCATE(CAND_PTS(D,M), STAT=IERR)
IF(IERR .NE. 0) THEN
   IERR = 330; RETURN; END IF
CAND_PTS(:,:) = MIN_X(:,1:M)
! Free the WEIGHTS array for next iteration.
DEALLOCATE(VTMOP%WEIGHTS, STAT=IERR)
IF(IERR .NE. 0) IERR = 332
! Free the VTMOP_MOD_WEIGHTS array.
DEALLOCATE(VTMOP_MOD_WEIGHTS, STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 332; RETURN; END IF

! If CHKPT is set, save to the checkpoint file.
! Check whether checkpointing is enabled.
IF (VTMOP%CHKPT) THEN
   ! Save the updated VTMOP object to the checkpoint file.
   CALL VTMOP_CHKPT(VTMOP, IERR)
   IF (IERR .NE. 0) RETURN
END IF
RETURN
END SUBROUTINE VTMOP_OPT

SUBROUTINE VTMOP_FINALIZE( VTMOP, DES_PTS, OBJ_PTS, M, PARETO_F, EFFICIENT_X, &
                           IERR )
! This subroutine finalizes a multiobjective optimization problem, by
! computing the entire weakly Pareto set, and freeing all dynamic memory
! allocated to the VTMOP object.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem. VTMOP is created using VTMOP_INIT.
!
! DES_PTS(1:D,1:N) is a real matrix of all design points in
!    the feasible design space [LB, UB], stored in column major order.
!    The second dimension of DES_PTS(:,:) (N) is assumed based on the shape
!    and must be at least D+1 to build an accurate surrogate model.
!
! OBJ_PTS(1:P,1:N) is a real matrix of objective values corresponding
!    to the design points in DES_PTS(:,:), stored in column major order.
!    I.e., for cost function F, OBJ_PTS(:,I) = F(DES_PTS(:,I)).
!
! PARETO_F(:,:) is an ALLOCATABLE real array. PARETO_F need not be allocated
!    on input. If allocated, any contents of PARETO_F are lost, as PARETO_F
!    is reallocated on output.
!
! EFFICIENT_X(:,:) is an ALLOCATABLE real array. EFFICIENT_X need not be
!    allocated on input. If allocated, any contents of EFFICIENT_X are
!    lost, as EFFICIENT_X is reallocated on output.
!
!
! On output:
!
! M is the cardinality of the weakly Pareto set.
!
! EFFICIENT_X(1:D,1:M) contains the entire weakly efficient set, stored in
!    column major ordering, with corresponding objective values in PARETO_F.
!
! PARETO_F(1:P,1:M) contains the entire weakly nondominated set. Note,
!    PARETO_F may contain duplicate values since the entire weakly nondominated
!    set is returned.
!
! IERR is an integer error flag.
!
! Hundreds digit:
!  000 : Normal output. Successful iteration, and list CAND_PTS obtained.
!
!  4xx : Errors detected.
!   Tens digit:
!     41x : The input parameters contained illegal dimensions or values.
!       Ones digit:
!         410 : The VTMOP object contains illegal problem dimensions. This
!               is most likely the result of an undetected segmentation
!               fault.
!         411 : The lead dimension of DES_PTS(:,:) must match D.
```

```
!         412 : The lead dimension of OBJ_PTS(:,:) must match P.
!         413 : The second dimensions of DES_PTS and OBJ_PTS must match.
!     42x : A memory error occurred while managing the output arrays.
!       Ones digit:
!         420 : A memory allocation error occurred while allocating an
!               output array PARETO_F and EFFICIENT_X.
!         421 : A memory deallocation error occurred while freeing an
!               output array PARETO_F or EFFICIENT_X, which was already
!               allocated on input.
!     43x : A memory error has occurred while managing internal memory.
!       Ones digit:
!         430 : A memory allocation error occurred in the local memory.
!         431 : A memory deallocation error occurred in the local memory.
!     441 : A memory error has occurred while freeing the VTMOP object or
!           module memory. The output arrays EFFICIENT_X and PARETO_F
!           should be unaffected.
!
USE IEEE_ARITHMETIC
IMPLICIT NONE
! Input parameters.
TYPE(VTMOP_TYPE), INTENT(INOUT) :: VTMOP ! Data struct containing problem info.
REAL(KIND=R8), INTENT(IN) :: DES_PTS(:,:) ! Table of precomputed design pts.
REAL(KIND=R8), INTENT(IN) :: OBJ_PTS(:,:) ! Table of objective values.
! Output parameters.
INTEGER, INTENT(OUT) :: M ! Cardinality of the weakly Pareto set.
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: PARETO_F(:,:)
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: EFFICIENT_X(:,:)
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Problem dimensions.
INTEGER :: D ! Design space dimension.
INTEGER :: N ! Cardinality of the weakly Pareto set.
INTEGER :: P ! Number of objectives.
! Local variables.
INTEGER :: I, J ! Loop indexing variables.
! Local dynamic arrays.
REAL(KIND=R8), ALLOCATABLE :: PARETO_SET(:,:) ! Pareto front.
REAL(KIND=R8), ALLOCATABLE :: EFFICIENT_SET(:,:) ! Efficient set.
! External BLAS procedures.
REAL(KIND=R8), EXTERNAL :: DNRM2 ! Euclidean distance (BLAS).

! Get problem dimensions from input data.
D = VTMOP%D; P = VTMOP%P; N = SIZE(DES_PTS,2)
! Check for illegal problem dimensions.
IF ((D < 1) .OR. (P < 2)) THEN
   IERR = 410; RETURN; END IF
IF (SIZE(DES_PTS,1) .NE. D) THEN
   IERR = 411; RETURN; END IF
IF (SIZE(OBJ_PTS,1) .NE. P) THEN
   IERR = 412; RETURN; END IF
IF (SIZE(OBJ_PTS,2) .NE. N) THEN
   IERR = 413; RETURN; END IF

! Allocate the dynamic arrays for storing both copies of the Pareto front.
ALLOCATE(PARETO_SET(P,N), EFFICIENT_SET(D,N), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 430; RETURN; END IF

! Get the weakly Pareto and efficient sets.
M = 0 ! Count the cardinality of the Pareto set.
OUTER : DO I = 1, N
   ! Skip NAN values.
   IF (ANY(IEEE_IS_NAN(OBJ_PTS(:,I)))) CYCLE OUTER
   INNER : DO J = 1, N
      ! Skip this point.
      IF (I .EQ. J) CYCLE INNER
      ! Check whether OBJ_PTS(:,I) and OBJ_PTS(:,J) are equal.
      IF (DNRM2(P, OBJ_PTS(:,I) - OBJ_PTS(:,J), 1) < VTMOP%OBJ_TOL) THEN
         CYCLE INNER ! Consider all weakly Pareto points.
      END IF
      ! Check whether OBJ_PTS(:,J) dominates OBJ_PTS(:,I).
      IF (ALL(OBJ_PTS(:,J) .LE. OBJ_PTS(:,I) + VTMOP%OBJ_TOL)) CYCLE OUTER
   END DO INNER
   ! Increment the counter and update both Pareto and efficient set.
   M = M + 1
   PARETO_SET(:,M) = OBJ_PTS(:,I)
   EFFICIENT_SET(:,M) = DES_PTS(:,I)
END DO OUTER

! Reallocate the output arrays.
IF (ALLOCATED(PARETO_F)) THEN
   DEALLOCATE(PARETO_F, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 421; RETURN; END IF
END IF
IF (ALLOCATED(EFFICIENT_X)) THEN
   DEALLOCATE(EFFICIENT_X, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 421; RETURN; END IF
END IF
ALLOCATE(EFFICIENT_X(D,M), PARETO_F(P,M), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 420; RETURN; END IF
```

```
! Populate the output arrays.
EFFICIENT_X(:,:) = EFFICIENT_SET(:,1:M)
PARETO_F(:,:) = PARETO_SET(:,1:M)
! Free the local memory.
DEALLOCATE(EFFICIENT_SET, PARETO_SET, STAT=IERR)
IF (IERR .NE. 0) THEN
    IERR = 431; RETURN; END IF
! Free the VTMOP data structure's memory.
IF (ALLOCATED(VTMOP%WEIGHTS)) THEN
   DEALLOCATE(VTMOP%WEIGHTS, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 441; RETURN; END IF
END IF
IF (ALLOCATED(VTMOP%LB)) THEN
   DEALLOCATE(VTMOP%LB, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 441; RETURN; END IF
END IF
IF (ALLOCATED(VTMOP%UB)) THEN
   DEALLOCATE(VTMOP%UB, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 441; RETURN; END IF
END IF
IF (ALLOCATED(VTMOP%CLIST)) THEN
   DEALLOCATE(VTMOP%CLIST, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 441; RETURN; END IF
END IF
RETURN
END SUBROUTINE VTMOP_FINALIZE

SUBROUTINE VTMOP_GENERATE(VTMOP, DES_PTS, OBJ_PTS, LBATCH, BATCHX, IERR, NB)
! The VTMOP_GENERATE subroutine produces static batches of candidate points
! using VTMOP_MOD. Each call to VTMOP_GENERATE performs a half iteration
! of the VTMOP_SOLVE algorithm, using a Latin hypercube design of size 4*D
! (where D is the design dimension) to explore each LTR and a Latin
! hypercube design of size (4*D)**2 to explore the entire design space.
!
! VTMOP_INITIALIZE must still be used to initialize the VTMOP object, and
! VTMOP_FINALIZE must be used to terminate and post process the results.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem. VTMOP is created using VTMOP_INIT.
!
! DES_PTS(D,N) is the current list of design points.
!
! OBJ_PTS(P,N) is the current list of objective points.
!
!
! On output:
!
! LBATCH returns the size of the next batch.
!
! BATCHX(D,LBATCH) is the next batch of design points to evaluate.
!
! IERR is an integer error flag. Error codes carried from worker
!    subroutines. In general:
!
!  000 : successful iteration.
!  003 : stopping criterion 3 achieved.
!
!  1xx : illegal input error.
!   Tens digit:
!    11x : The VTMOP object contained illegal dimensions or values, and
!          may not have been properly initialized.
!       Ones digit:
!         110 : The problem dimensions D and P are not legal.
!         111 : The internal arrays have not been allocated.
!         112 : The internal arrays have been initialized, but to invalid
!               dimensions.
!         113 : The lower and upper bound constraints contain illegal values,
!               subject to the design space tolerance.
!    12x : Illegal values in other inputs.
!         120 : The lead dimension of DES_PTS does not match the data in VTMOP.
!         121 : The lead dimension of OBJ_PTS does not match the data in VTMOP.
!         122 : The second dimensions of DES_PTS and OBJ_PTS do not match.
!         123 : The preferred batch size NB must be nonnegative.
!
!    2xx, 3xx : Error in VTMOP_LTR or VTMOP_OPT, respectively.
!
!    5xx, 6xx, 7xx, 8xx : Error in DELAUNAYGRAPH, FIT_SURROGATES, LOCAL_OPT,
!                         or LH_DESIGN, respectively. See documentation for
!                         those procedures for more details.
!
!    9xx : Checkpointing error. See checkpointing subroutine documentation.
!
!
! Optional input arguments.
!
```

```
! NB is the preferred batch size. If NB is present, then it is used as the
!    size of the Latin hypercube design in the search phase.
!
IMPLICIT NONE
! Parameter list.
TYPE(VTMOP_TYPE), INTENT(INOUT) :: VTMOP ! VTMOP object.
REAL(KIND=R8), INTENT(IN) :: DES_PTS(:,:) ! Design point database.
REAL(KIND=R8), INTENT(IN) :: OBJ_PTS(:,:) ! Objective point database.
INTEGER, INTENT(OUT) :: LBATCH ! Requested batch size.
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: BATCHX(:,:) ! Batch of requested pts.
INTEGER, INTENT(OUT) :: IERR ! Integer error flag.
! Optional inputs.
INTEGER, OPTIONAL, INTENT(IN) :: NB ! Preferred batch size.
! Local variables.
INTEGER :: D, P ! Problem dimensions.
INTEGER :: I, J, K ! Loop indexing variables.
INTEGER :: NW ! Number of adaptive weights.
REAL(KIND=R8) :: LTR_LB(SIZE(VTMOP%LB,1)) ! LTR lower bound constraints.
REAL(KIND=R8) :: LTR_UB(SIZE(VTMOP%UB,1)) ! LTR upper bound constraints.
REAL(KIND=R8), ALLOCATABLE :: TMP(:,:) ! Temporary array for filtering data.
REAL(KIND=R8), ALLOCATABLE :: WEIGHTS(:,:) ! Temporary array for weights.
! BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2
! Get problem dimensions.
D = VTMOP%D; P = VTMOP%P
! Check for illegal input parameter values.
IF ( D < 1 .OR. P < 2 ) THEN
   IERR = 110; RETURN; END IF
IF ( .NOT. ( ALLOCATED(VTMOP%LB) .AND. ALLOCATED(VTMOP%UB) .AND. &
             ALLOCATED(VTMOP%CLIST) ) ) THEN
   IERR = 111; RETURN; END IF
IF ( SIZE(VTMOP%LB,1) .NE. D .OR. SIZE(VTMOP%UB,1) .NE. D .OR. &
     SIZE(VTMOP%CLIST,1) .NE. D+1)  THEN
   IERR = 112; RETURN; END IF
IF ( ANY(VTMOP%LB(:) .GE. VTMOP%UB(:) - VTMOP%DES_TOL) ) THEN
   IERR = 113; RETURN; END IF
! Check for illegal dimensions to DES_PTS and OBJ_PTS.
IF ( D .NE. SIZE(DES_PTS,1) ) THEN
   IERR = 120; RETURN; END IF
IF (  P .NE. SIZE(OBJ_PTS,1) ) THEN
   IERR = 121; RETURN; END IF
IF ( SIZE(DES_PTS,2) .NE. SIZE(OBJ_PTS,2) ) THEN
   IERR = 122; RETURN; END IF
IF (PRESENT(NB)) THEN
   IF (NB < 0) THEN
      IERR = 123; RETURN; END IF
END IF

! Perform a half-iteration and request the next batch, based on the state of
! the VTMOP object.

! There are two cases.
IF (.NOT. ALLOCATED(VTMOP%WEIGHTS)) THEN
   ! Execute the search phase.

   ! Choose the most isolated point(s), and construct the LTR(s).
   CALL VTMOP_LTR( VTMOP, DES_PTS, OBJ_PTS, LTR_LB, LTR_UB, IERR )
   IF (IERR .NE. 0) RETURN

   ! Get the batch size, using a special rule for the first iteration
   ! and respecting the preferred batch size NB.
   IF (VTMOP%ITERATE .EQ. 0) THEN
      ! In first iteration, generate a much larger LH_DESIGN.
      LBATCH = (4*D)**2
   ELSE
      ! In later iterations, generate a smaller LH_DESIGN.
      LBATCH = 4*D
   END IF
   ! Correct to NB, when present.
   IF (PRESENT(NB)) LBATCH = NB
   ! Request a static exploration of the LTR.
   CALL LH_DESIGN(D, LTR_LB, LTR_UB, LBATCH, TMP, IERR, VTMOP%DES_TOL)
   IF (IERR .NE. 0) THEN
      IERR = IERR + 800; RETURN; END IF
   ! Allocate the output array.
   ALLOCATE(BATCHX(D,LBATCH), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 820; RETURN; END IF
   ! Filter out any redundant points.
   K = 0
   OUTER : DO I = 1, LBATCH
      INNER : DO J = 1, SIZE(DES_PTS,2)
         ! Check for any repeated design points.
         IF (DNRM2(D, TMP(:,I) - DES_PTS(:,J), 1) < VTMOP%DES_TOL) CYCLE OUTER
      END DO INNER
      ! If the point is not repeated, add it to the output array.
      K = K + 1
      BATCHX(:,K) = TMP(:,I)
   END DO OUTER
   ! Check the size of the output array.
   IF (K .EQ. LBATCH) THEN
      ! No resizing is required, free the temporary array (typical case).
```

```
      DEALLOCATE(TMP, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 821; RETURN; END IF
   ELSE
      ! Resize the output array (rare case).
      TMP(:,1:K) = BATCHX(:,1:K)
      DEALLOCATE(BATCHX, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 821; RETURN; END IF
      ALLOCATE(BATCHX(D,K), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      BATCHX(:,:) = TMP(:,1:K)
      ! Free the temporary array.
      DEALLOCATE(TMP, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 821; RETURN; END IF
   END IF
ELSE
   ! Execute the optimization phase.

   ! Recover the last LTR.
   IF (VTMOP%ITERATE .EQ. 0) THEN
      LTR_LB(:) = VTMOP%LB(:)
      LTR_UB(:) = VTMOP%UB(:)
   ELSE
      DO I = 1, D
         LTR_UB(I) = MIN( VTMOP%CLIST(I,VTMOP%ITERATE) +   &
                          VTMOP%CLIST(D+1,VTMOP%ITERATE) * &
                          (VTMOP%UB(I) - VTMOP%LB(I)),     &
                          VTMOP%UB(I) )
         LTR_LB(I) = MAX( VTMOP%CLIST(I,VTMOP%ITERATE) -   &
                          VTMOP%CLIST(D+1,VTMOP%ITERATE) * &
                          (VTMOP%UB(I) - VTMOP%LB(I)),     &
                          VTMOP%LB(I) )
      END DO
   END IF

   ! Add additional weights at random in order to match NB.
   IF (PRESENT(NB)) THEN
      NW = SIZE(VTMOP%WEIGHTS,2)
      ! Generate twice as many points as needed, in case several points
      ! produce redundant solutions.
      ALLOCATE(TMP(P,FLOOR(NB*1.5)), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      CALL RANDOM_NUMBER(TMP)
      DO I = 1, SIZE(TMP, 2)
         TMP(:,I) = TMP(:,I) / SUM(TMP(:,I))
      END DO
      ! Make a copy of VTMOP%WEIGHTS.
      ALLOCATE(WEIGHTS(P,NW), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      WEIGHTS(:,:) = VTMOP%WEIGHTS(:,:)
      ! Reallocate the VTMOP%WEIGHTS array.
      DEALLOCATE(VTMOP%WEIGHTS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 821; RETURN; END IF
      ALLOCATE(VTMOP%WEIGHTS(P,NW+SIZE(TMP,2)), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      ! Copy in the full list of weights.
      VTMOP%WEIGHTS(:,1:NW) = WEIGHTS(:,:)
      VTMOP%WEIGHTS(:,NW+1:NW+SIZE(TMP,2)) = TMP(:,:)
      ! Free the temporary arrays.
      DEALLOCATE(WEIGHTS, TMP, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 821; RETURN; END IF
   END IF

   ! Optimize the surrogates in the LTR.
   CALL VTMOP_OPT( VTMOP, LTR_LB, LTR_UB, DES_PTS, OBJ_PTS, TMP, IERR )
   IF (IERR .NE. 0) RETURN

   ! If less than NW + NB - (NW mod NB) points were returned, return all the
   ! candidates.
   IF (SIZE(TMP,2) .LE. NW+NB-MOD(NW,NB)) THEN
      ALLOCATE(BATCHX(D, SIZE(TMP,2)), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      BATCHX(:,:) = TMP(:,:)
   ! Otherwise, return the first NW + NB - (NW mod NB) candidates.
   ELSE
      ALLOCATE(BATCHX(D, NW+NB-MOD(NW,NB)), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 820; RETURN; END IF
      BATCHX(:,:) = TMP(:,1:NW+NB-MOD(NW,NB))
   END IF

   ! Free the temporary array.
   DEALLOCATE(TMP, STAT=IERR)
```

```
   IF (IERR .NE. 0) THEN
      IERR = 821; RETURN; END IF
   ! Get the size of BATCHX.
   LBATCH = SIZE(BATCHX, 2)
END IF
RETURN
END SUBROUTINE VTMOP_GENERATE

! The following subroutines are for performing checkpointing.

SUBROUTINE VTMOP_CHKPT_NEW(VTMOP, IERR)
! Create a new checkpoint file for a given instance of VTMOP.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries metadata
!    about the multiobjective problem.
!
!
! On output:
!
! IERR is an integer error flag.
!
!  000 : Normal output. Successful initialization of a new VTMOP checkpoint.
!
!  9xx : Errors detected.
!     91x : The input parameters contained illegal dimensions or values.
!         910 : VTMOP does not appear to have been properly allocated.
!         911 : VTMOP has been allocated, but appears to contain corrupted
!               or inconsistent data.
!     92x : A file I/O error was detected.
!         920 : An error occurred while opening the checkpoint file.
!         921 : An error occurred while writing data to the checkpoint file.
!         922 : An error occurred while closing the checkpoint file.
!
IMPLICIT NONE
! Input parameters.
TYPE(VTMOP_TYPE), INTENT(IN) :: VTMOP ! Data structure containing problem info.
! Output parameters.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Check for uninitialized values.
IF ( (.NOT. ALLOCATED(VTMOP%LB)) .OR. (.NOT. ALLOCATED(VTMOP%UB)) ) THEN
   IERR = 910; RETURN; END IF
! Check for illegal/mismatched values.
IF ( VTMOP%D < 1 .OR. VTMOP%P < 2) THEN
   IERR = 911; RETURN; END IF
IF (SIZE(VTMOP%LB, 1) .NE. VTMOP%D .OR. SIZE(VTMOP%UB) .NE. VTMOP%D) THEN
   IERR = 911; RETURN; END IF
! Open the checkpoint file, using unformatted write.
OPEN(VTMOP_CHKPTUNIT, FILE=VTMOP_CHKPTFILE, FORM="unformatted", &
     ACTION="write", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 920; RETURN; END IF
! Write unformatted VTMOP metadata to the checkpoint file.
WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%D, VTMOP%P
IF (IERR .NE. 0) THEN
   IERR = 921; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%DECAY, VTMOP%DES_TOL, VTMOP%EPS,     &
                                    VTMOP%EPSW, VTMOP%OBJ_TOL, VTMOP%MIN_RADF, &
                                    VTMOP%TRUST_RADF
IF (IERR .NE. 0) THEN
   IERR = 921; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%LB(1:VTMOP%D), VTMOP%UB(1:VTMOP%D)
IF (IERR .NE. 0) THEN
   IERR = 921; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Close the checkpoint file.
CLOSE(VTMOP_CHKPTUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 922; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_CHKPT_NEW

SUBROUTINE VTMOP_CHKPT(VTMOP, IERR)
! Save VTMOP's iteration data to an existing checkpoint file.
!
!
! On input:
!
! VTMOP is an object of derived data type VTMOP_TYPE, which carries meta data
!    about the multiobjective problem.
!
!
! On output:
!
! IERR is an integer error flag.
!
!  000 : Normal output. Iteration data saved to the checkpoint file.
!
!  9xx : Errors detected.
!     91x : The input parameters contained illegal dimensions or values.
!         910 : VTMOP does not appear to have been properly allocated.
```

```
!         911 : VTMOP has been allocated, but appears to contain corrupted
!               or inconsistent data.
!     93x : A file I/O error was detected.
!         930 : The checkpoint file could not be opened, check whether
!               CHKPTFILE has been properly initialized.
!         931 : An error occurred while writing data to the checkpoint file.
!         932 : An error occurred while closing the checkpoint file.
!
IMPLICIT NONE
! Input parameters.
TYPE(VTMOP_TYPE), INTENT(IN) :: VTMOP ! Data structure containing problem info.
! Output parameters.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Check for uninitialized values.
IF ( (.NOT. ALLOCATED(VTMOP%LB)) .OR. (.NOT. ALLOCATED(VTMOP%UB)) ) THEN
   IERR = 910; RETURN; END IF
! Check for illegal/mismatched values.
IF ( VTMOP%D < 1 .OR. VTMOP%P < 2) THEN
   IERR = 911; RETURN; END IF
IF (SIZE(VTMOP%LB, 1) .NE. VTMOP%D .OR. SIZE(VTMOP%UB) .NE. VTMOP%D) THEN
   IERR = 911; RETURN; END IF
! Open the checkpoint file, using unformatted append.
OPEN(VTMOP_CHKPTUNIT, FILE=VTMOP_CHKPTFILE, FORM="unformatted", ACTION="write", &
     POSITION="append", STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 930; RETURN; END IF
! The status of VTMOP%WEIGHTS tells the phase of the VTMOP algorithm.
IF (ALLOCATED(VTMOP%WEIGHTS)) THEN
   IF (VTMOP%ITERATE > 0) THEN ! Don't write CLIST for the zeroth iteration.
      ! Write unformatted VTMOP iteration data to the checkpoint file.
      WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%CLIST(1:VTMOP%D+1,VTMOP%ITERATE)
      IF (IERR .NE. 0) THEN
         IERR = 931; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
   END IF
   ! Write unformatted adaptive weights to the checkpoint file.
   WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) SIZE(VTMOP%WEIGHTS, 2)
   IF (IERR .NE. 0) THEN
      IERR = 931; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
   WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%WEIGHTS(:,:)
   IF (IERR .NE. 0) THEN
      IERR = 931; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
ELSE
   ! Write the iteration counter.
   WRITE(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%ITERATE
   IF (IERR .NE. 0) THEN
      IERR = 931; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
END IF
! Close the checkpoint file.
CLOSE(VTMOP_CHKPTUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 932; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_CHKPT

SUBROUTINE VTMOP_CHKPT_RECOVER(VTMOP, IERR)
! Recover VTMOP's progress from a checkpoint file.
!
!
! On output:
!
! The status of VTMOP is as specified in CHKPTFILE.
!
! IERR is an integer error flag.
!
!  000 : Normal output. Iteration data saved to the checkpoint file.
!
!  9xx : Errors detected.
!     94x : A file I/O error was detected.
!         940 : The checkpoint file could not be opened, check whether
!               CHKPTFILE has been properly initialized.
!         941 : An error occurred while writing data to the checkpoint file.
!         942 : An error occurred while closing the checkpoint file.
!     95x : A memory allocation error occurred.
!         950 : A memory allocation error occurred.
!         951 : A memory deallocation error occurred.
!     960 : Failed the sanity check. Either the checkpoint feature was
!           implemented improperly, or VTMOP_CHKPTFILE was corrupted.
!
IMPLICIT NONE
! Output parameters.
TYPE(VTMOP_TYPE), INTENT(OUT) :: VTMOP ! Data structure containing problem info.
INTEGER, INTENT(OUT) :: IERR ! Error flag arrays.
! Temporary arrays.
INTEGER :: NW
REAL(KIND=R8), ALLOCATABLE :: TMP(:,:)
! Open the checkpoint file, using unformatted write.
OPEN(VTMOP_CHKPTUNIT, FILE=VTMOP_CHKPTFILE, FORM="unformatted", ACTION="read", &
     STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 940; RETURN; END IF
! Read in the problem dimensions.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%D, VTMOP%P
```

```
IF (IERR .NE. 0) THEN
   IERR = 941; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Read in the problem parameters.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%DECAY, VTMOP%DES_TOL, VTMOP%EPS,     &
                                   VTMOP%EPSW, VTMOP%OBJ_TOL, VTMOP%MIN_RADF, &
                                   VTMOP%TRUST_RADF
IF (IERR .NE. 0) THEN
   IERR = 941; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Allocate the bound constraints.
ALLOCATE(VTMOP%LB(VTMOP%D), VTMOP%UB(VTMOP%D), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Read in the bound constraints.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%LB(1:VTMOP%D), VTMOP%UB(1:VTMOP%D)
IF (IERR .NE. 0) THEN
   IERR = 941; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Initialize the iteration data.
VTMOP%ITERATE = 0
VTMOP%LCLIST = 20
ALLOCATE(VTMOP%CLIST(VTMOP%D+1,VTMOP%LCLIST), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF

! Recreate the 0th iteration.

! Read the size of the next batch of adaptive weights.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) NW
IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
   IERR = 0; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Allocate VTMOP%WEIGHTS accordingly.
ALLOCATE(VTMOP%WEIGHTS(VTMOP%P, NW), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Read in the next batch of adaptive weights.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%WEIGHTS(:,:)
IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
   IERR = 0; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Next checkpoint tells that the search phase has completed.
READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) NW
IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
   IERR = 0; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Update the iteration counter, and do a sanity check.
VTMOP%ITERATE = VTMOP%ITERATE + 1
IF (NW .NE. VTMOP%ITERATE) THEN
   IERR = 960; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
! Free the adaptive weights for the next iteration.
DEALLOCATE(VTMOP%WEIGHTS, STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 951; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF

! Read data into VTMOP%CLIST(:,:) until the end of file.
READ_LOOP : DO WHILE (.TRUE.)
   ! Check if VTMOP%CLIST(:,:) needs to be resized.
   IF (VTMOP%ITERATE .EQ. VTMOP%LCLIST) THEN
      ! Allocate the temporary array.
      ALLOCATE(TMP(VTMOP%D+1,VTMOP%ITERATE), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
      ! Make a temporary copy.
      TMP(:,:) = VTMOP%CLIST(:,:)
      ! Update the size of VTMOP%LCLIST.
      VTMOP%LCLIST = VTMOP%LCLIST * 2
      ! Reallocate VTMOP%CLIST to twice its current size.
      DEALLOCATE(VTMOP%CLIST, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 951; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
      ALLOCATE(VTMOP%CLIST(VTMOP%D+1, VTMOP%LCLIST), STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
      ! Restore values back into CLIST and free the temporary array.
      VTMOP%CLIST(:,1:VTMOP%ITERATE) = TMP(:,:)
      DEALLOCATE(TMP, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 951; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
   END IF
   ! Now, read in the next point in the center list from the file.
   READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%CLIST(1:VTMOP%D+1, VTMOP%ITERATE)
   IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
      IERR = 0; EXIT READ_LOOP; END IF
   ! Read the size of the next batch of adaptive weights.
   READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) NW
   IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
      IERR = 0; EXIT READ_LOOP; END IF
   ! Allocate VTMOP%WEIGHTS accordingly.
   ALLOCATE(VTMOP%WEIGHTS(VTMOP%P, NW), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 950; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
   ! Read in the next batch of adaptive weights.
   READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) VTMOP%WEIGHTS(:,:)
   IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
      IERR = 0; EXIT READ_LOOP; END IF
   ! Next checkpoint tells that the search phase has completed.
```

```
   READ(VTMOP_CHKPTUNIT, IOSTAT=IERR) NW
   IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
      IERR = 0; EXIT READ_LOOP; END IF
   ! Update the iteration counter, and do a sanity check.
   VTMOP%ITERATE = VTMOP%ITERATE + 1
   IF (NW .NE. VTMOP%ITERATE) THEN
      IERR = 960; EXIT READ_LOOP; END IF
   ! Free the adaptive weights for the next iteration.
   DEALLOCATE(VTMOP%WEIGHTS, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 951; CLOSE(VTMOP_CHKPTUNIT); RETURN; END IF
END DO READ_LOOP
! Close the checkpoint file.
CLOSE(VTMOP_CHKPTUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 942; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_CHKPT_RECOVER

SUBROUTINE VTMOP_NEW_DATA(D, P, IERR)
! Create a new checkpoint data file to store VTMOP's function evaluation data.
!
!
! On input:
!
! D is the dimension of the design space.
!
! P is the dimension of the objective space.
!
!
! On output:
!
! IERR is an integer error flag.
!
!  000 : Normal output. Successful initialization of a new data checkpoint
!        file.
!
!  97x : Errors detected.
!     97x : A data I/O error was detected.
!         970 : An error occurred while opening the data file.
!         971 : An error occurred while writing data to the data file.
!         972 : An error occurred while closing the data file.
!
IMPLICIT NONE
! Input parameters.
INTEGER, INTENT(IN) :: D, P ! Problem dimensions.
! Output parameters.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Create a new data file, using unformatted write.
OPEN(VTMOP_DATAUNIT, FILE=VTMOP_DATAFILE, FORM="unformatted", ACTION="write", &
     IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 970; RETURN; END IF
! Write the problem dimensions.
WRITE(VTMOP_DATAUNIT, IOSTAT=IERR) D, P
IF (IERR .NE. 0) THEN
   IERR = 971; CLOSE(VTMOP_DATAUNIT); RETURN; END IF
! Close the data file.
CLOSE(VTMOP_DATAUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 972; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_NEW_DATA

SUBROUTINE VTMOP_SAVE_DATA(DES_PT, OBJ_PT, IERR)
! Save VTMOP's function evaluation data to an existing checkpoint file.
!
!
! On input:
!
! DES_PT(:) is a design point to save.
!
! OBJ_PT(:) is a corresponding objective value to save.
!
!
! On output:
!
! IERR is an integer error flag.
!
!  000 : Normal output. Successful initialization of a new VTMOP checkpoint.
!
!  9xx : Errors detected.
!     98x : A data I/O error was detected.
!         980 : An error occurred while opening the data file.
!         981 : An error occurred while writing data to the data file.
!         982 : An error occurred while closing the data file.
!
IMPLICIT NONE
! Input parameters.
REAL(KIND=R8), INTENT(IN) :: DES_PT(:), OBJ_PT(:)
! Output parameters.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
```

```
! Open an existing data file, using unformatted append.
OPEN(VTMOP_DATAUNIT, FILE=VTMOP_DATAFILE, FORM="unformatted", ACTION="write", &
     POSITION="append", STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 980; RETURN; END IF
! Write the design point and objective value.
WRITE(VTMOP_DATAUNIT, IOSTAT=IERR) DES_PT(:), OBJ_PT(:)
IF (IERR .NE. 0) THEN
   IERR = 981; CLOSE(VTMOP_DATAUNIT); RETURN; END IF
! Close the checkpoint file.
CLOSE(VTMOP_DATAUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 982; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_SAVE_DATA

SUBROUTINE VTMOP_RECOVER_DATA(DBN, DBX, DBF, IERR, DB_SIZE)
! Recover VTMOP's function evaluation database from an existing checkpoint
! file.
!
!
! On input:
!
! DBX(:,:) and DBF(:,:) are unallocated, allocatable arrays.
!
! On output:
!
! DBN is the integer counter specifying the final length of DB{X|F}
!
! DBX(:,:) is allocated to the problem dimensions and contains the
!    recovered database of design points.
!
! DBF(:,:) is allocated to the problem dimensions and contains the
!    recovered database of objective points.
!
! IERR is an integer error flag.
!
!  000 : Normal output. Successful initialization of a new VTMOP checkpoint.
!
!  99x : Errors detected.
!     99x : A data I/O error was detected.
!         990 : An error occurred while opening the data file.
!         991 : An error occurred while reading data from the data file.
!         992 : An error occurred while closing the data file.
!         993 : There was an issue allocating the local memory.
!         994 : The number of recovered data points exceeds the budget.
!
!
! Optional arguments:
!
! DB_SIZE, when present, specifies the amount of memory to allocate for
!    DBX and DBF.  By default, DBX and DBF are allocated to a length of
!    1000.
!
IMPLICIT NONE
! Output parameters.
INTEGER, INTENT(OUT) :: DBN ! The size of the final database.
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: DBX(:,:) ! Recovered design points.
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: DBF(:,:) ! Recovered objective pts.
INTEGER, INTENT(OUT) :: IERR ! Error flag arrays.
INTEGER, OPTIONAL, INTENT(IN) :: DB_SIZE ! Size of the database.
! Local variables.
INTEGER :: D, P ! Problem dimensions.
INTEGER :: DB_SIZEL ! Local copy of database size parameters.
REAL(KIND=R8), ALLOCATABLE :: DES_PT(:), OBJ_PT(:) ! Temp arrays.
! Read in the optional inputs.
DB_SIZEL = 1000
IF (PRESENT(DB_SIZE)) THEN
   IF (DB_SIZE > 0) DB_SIZEL = DB_SIZE
END IF
! Create a new data file, using unformatted write.
OPEN(VTMOP_DATAUNIT, FILE=VTMOP_DATAFILE, FORM="unformatted", ACTION="read", &
     STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 990; RETURN; END IF
! Read in the problem dimensions.
READ(VTMOP_DATAUNIT, IOSTAT=IERR) D, P
IF (IERR .NE. 0) THEN
   IERR = 991; CLOSE(VTMOP_DATAUNIT); RETURN; END IF
! Allocate the output array memory.
IF (ALLOCATED(DBX)) THEN
   DEALLOCATE(DBX, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 993; RETURN; END IF
END IF
IF (ALLOCATED(DBF)) THEN
   DEALLOCATE(DBF, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 993; RETURN; END IF
END IF
ALLOCATE(DBX(D,DB_SIZEL), DBF(P,DB_SIZEL), STAT=IERR)
IF (IERR .NE. 0) THEN
```

```
   IERR = 993; RETURN; END IF
! Allocate the local memory.
ALLOCATE(DES_PT(D), OBJ_PT(P), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 993; RETURN; END IF
! Read until all data points are recovered.
DBN = 0
DO WHILE(.TRUE.)
   READ(VTMOP_DATAUNIT, IOSTAT=IERR) DES_PT, OBJ_PT
   IF (IERR .NE. 0) THEN ! A read error occurred. This must be EOF.
      IERR = 0; EXIT; END IF
   ! If the database is at capacity, return an error.
   IF (DBN .GE. DB_SIZEL) THEN
      IERR = 994; RETURN; END IF
   ! Add DES_PT and OBJ_PT to the module database.
   DBN = DBN + 1
   DBX(:,DBN) = DES_PT(:)
   DBF(:,DBN) = OBJ_PT(:)
END DO
! Close the data file.
CLOSE(VTMOP_DATAUNIT, IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 992; RETURN; END IF
RETURN
END SUBROUTINE VTMOP_RECOVER_DATA

! The following module procedures are used internally.

SUBROUTINE MOP_EVALUATE(C, V, IERR)
! MOP_EVALUATE is a wrapper procedure for VTMOP_MOD_OBJ_FUNC. Before
! each evaluation, MOP_EVALUATE checks VTMOP_MOD_DBX(:,:) to prevent
! redundant function evaluations. If the evaluation point is already
! in the VTMOP_MOD_DBX(:,:), then the corresponding entry from
! VTMOP_MOD_DBF(:,:) gets returned. Otherwise, a true evaluation of
! VTMOP_MOD_OBJ_FUNC is performed, and VTMOP_MOD_DB{X|F|N} get updated.
!
! This is a serial implementation of MOP_EVALUATE and is not threadsafe.
!
!
! On input:
!
! C(:) contains the real design point to evaluate.
!
!
! On output:
!
! V(:) contains the evaluated objective value.
!
! IERR is an integer error code. Any nonzero code indicates an
!    illegal/missing value, and V(:) is filled with IEEE NAN values.
!
USE IEEE_ARITHMETIC
IMPLICIT NONE
! Parameter list.
REAL(KIND=R8), INTENT(IN) :: C(:) ! Design point input.
REAL(KIND=R8), INTENT(OUT) :: V(:) ! Objective point output.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Local variables.
INTEGER :: I ! Use for identifying the index of any duplicate points.
! External BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2
! If the budget has been exceeded, do nothing.
IF (VTMOP_MOD_DBN .GE. VTMOP_MOD_BB_BUDGET) THEN
   V = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
   IERR = -1
   RETURN
END IF
! Check if already evaluated.
DO I = VTMOP_MOD_DBN, 1, -1
   IF(DNRM2(VTMOP_MOD_D, VTMOP_MOD_DBX(:,I) - C(:), 1) < VTMOP_MOD_DES_TOL) EXIT
END DO
IF (I .EQ. 0) THEN
   ! Get vector-valued function output.
   CALL VTMOP_MOD_OBJ_FUNC(C, V, IERR)
   IF (IERR .NE. 0) THEN
      ! If a nonzero error is returned, store a NAN.
      V = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
      VTMOP_MOD_DBN = VTMOP_MOD_DBN + 1
      VTMOP_MOD_DBX(:, VTMOP_MOD_DBN) = C(:)
      VTMOP_MOD_DBF(:, VTMOP_MOD_DBN) = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
   ELSE
      ! If an IEEE NAN is returned, report a missing value.
      IF ( ANY(IEEE_IS_NAN(V(:))) ) THEN
         V = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN); END IF
      ! Update the design and objective datasets.
      VTMOP_MOD_DBN = VTMOP_MOD_DBN + 1
      VTMOP_MOD_DBX(:, VTMOP_MOD_DBN) = C(:)
      VTMOP_MOD_DBF(:, VTMOP_MOD_DBN) = V(:)
   END IF
   ! If checkpointing is active, then save to the data file.
   IF (VTMOP_MOD_CHKPT) THEN
      CALL VTMOP_SAVE_DATA(C, V, IERR)
```

```
      IF (IERR .NE. 0) RETURN
   END IF
ELSE
   ! Retrieve function value from table.
   V(:) = VTMOP_MOD_DBF(:,I)
END IF
IF ( ANY(IEEE_IS_NAN(V(:))) ) IERR = -1
RETURN
END SUBROUTINE MOP_EVALUATE

SUBROUTINE MOP_P_EVALUATE(C, V, IERR)
! This is a parallel threadsafe implementation of MOP_EVALUATE, assuming
! that the objective function referenced by VTMOP_MOD_OBJ_FUNC is also
! threadsafe.
!
! Before each evaluation, MOP_P_EVALUATE checks VTMOP_MOD_DBX(:,:), acquiring
! an OpenMP lock to ensure sequential consistency. If the evaluation point
! is already in the VTMOP_MOD_DBX(:,:), then the corresponding entry from
! VTMOP_MOD_DBF(:,:) gets returned. Otherwise, a true evaluation of
! VTMOP_MOD_OBJ_FUNC is performed, and VTMOP_MOD_DB{X|F|N} get updated.
! An array of OpenMP locks is used to control access to entries that are
! mid-evaluation.
!
!
! On input:
!
! C(:) contains the real design point to evaluate.
!
!
! On output:
!
! V(:) contains the evaluated objective value.
!
! IERR is an integer error code. Any nonzero code indicates an
!    illegal/missing value, and V(:) is filled with IEEE NAN values.
!
USE IEEE_ARITHMETIC
IMPLICIT NONE
! Parameter list.
REAL(KIND=R8), INTENT(IN) :: C(:) ! Design point input.
REAL(KIND=R8), INTENT(OUT) :: V(:) ! Objective point output.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Local variables.
INTEGER :: I, J ! Use for identifying the index of any duplicate points.
! External BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2
! Acquire a lock on the entire databse, to prevent race conditions.
CALL OMP_SET_LOCK(VTMOP_MOD_DBLCK)
! If the budget has been exceeded, do nothing.
IF (VTMOP_MOD_DBN .GE. VTMOP_MOD_BB_BUDGET) THEN
   V = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
   IERR = -1
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DBLCK)
   RETURN
END IF
! Check if C(:) was already evaluated.
DO I = VTMOP_MOD_DBN, 1, -1
   IF(DNRM2(VTMOP_MOD_D,VTMOP_MOD_DBX(:,I)-C(:),1) < VTMOP_MOD_DES_TOL) EXIT
END DO
! If an evaluation is being done, update the table and launch a new task.
IF (I .EQ. 0) THEN
   ! Increment the counter and add C to the database.
   VTMOP_MOD_DBN = VTMOP_MOD_DBN + 1
   J = VTMOP_MOD_DBN
   VTMOP_MOD_DBX(:, J) = C(:)
   ! Acquire a lock on the database entry.
   CALL OMP_SET_LOCK(VTMOP_MOD_DB_BUSY(J))
   ! Release the lock on the entire database.
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DBLCK)
   ! Evaluate the objective function asynchronously.
   CALL VTMOP_MOD_OBJ_FUNC(C, V, IERR)
   ! Reacquire the database lock and write the function values into the table.
   CALL OMP_SET_LOCK(VTMOP_MOD_DBLCK)
   IF ((IERR .NE. 0) .OR. ANY(IEEE_IS_NAN(V(:)))) THEN
      ! If a nonzero error flag was returned, store a NAN.
      V = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
      VTMOP_MOD_DBF(:, J) = IEEE_VALUE(0.0_R8, IEEE_QUIET_NAN)
   ELSE
      ! Update the objective dataset.
      VTMOP_MOD_DBF(:, J) = V(:)
   END IF
   ! Release the lock on the specific entry.
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DB_BUSY(J))
   ! If checkpointing is active, then save data to the chkpt file.
   IF (VTMOP_MOD_CHKPT) THEN
      CALL VTMOP_SAVE_DATA(C, V, IERR)
   END IF
   ! Release the lock on the entire database.
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DBLCK)
ELSE
   ! Release the database lock and wait for the function value lock.
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DBLCK)
```

```
   CALL OMP_SET_LOCK(VTMOP_MOD_DB_BUSY(I))
   ! Read the function value from the database and release the lock.
   V(:) = VTMOP_MOD_DBF(:,I)
   CALL OMP_UNSET_LOCK(VTMOP_MOD_DB_BUSY(I))
END IF
IF ( ANY(IEEE_IS_NAN(V(:))) ) IERR = -1
RETURN
END SUBROUTINE MOP_P_EVALUATE

FUNCTION SCALAR_FUNC(C, IERR) RESULT(F)
! This is a public module procedure that uses the private module array
! VTMOP_MOD_WEIGHTS to scalarize the output of MOP_EVALUATE. SCALAR_FUNC
! matches the interface of VTMOP_MOD_SCALAR_INT and can be passed as
! input to a generic single objective optimization procedure.
!
! On input:
!
! C(:) is a design point to be evaluated.
!
!
! On output:
!
! IERR is an integer error flag. Any nonzero value indicates an
!    illegal/missing value, as reported by MOP_EVALUATE.
!
! F is a scalar output, as determined by the scalarization weights in
!    VTMOP_MOD_WEIGHTS(:).
!
IMPLICIT NONE
! Parameters.
REAL(KIND=R8), INTENT(IN) :: C(:) ! Design space input.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
REAL(KIND=R8) :: F ! Scalar output value.
! Local variables.
REAL(KIND=R8) :: V(VTMOP_MOD_P) ! Nonscalarized output vector.
! BLAS function for computing inner products.
REAL(KIND=R8), EXTERNAL :: DDOT
! Get vector-valued objective.
CALL MOP_EVALUATE(C, V, IERR)
IF (IERR .NE. 0) RETURN
! Return weighted objective value.
F = DDOT(VTMOP_MOD_P, VTMOP_MOD_WEIGHTS(:), 1, V(:), 1)
RETURN
END FUNCTION SCALAR_FUNC

FUNCTION P_SCALAR_FUNC(C, WEIGHTS, IERR) RESULT(F)
! This is a threadsafe variation of SCALAR_FUNC. Because the module
! weights cannot be passed through nested OpenMP parallel regions
! without risking error, P_SCALAR_FUNC explicitly receives the
! scalarization weights array.
!
! On input:
!
! C(:) is a design point to be evaluated.
!
! WEIGHTS(:) is a scalarizing weight vector.
!
!
! On output:
!
! IERR is an integer error flag. Any nonzero value indicates an
!    illegal/missing value, as reported by MOP_EVALUATE.
!
! F is a scalar output, as determined by WEIGHTS(:).
!
IMPLICIT NONE
! Parameters.
REAL(KIND=R8), INTENT(IN) :: C(:) ! Design space input.
REAL(KIND=R8), INTENT(IN) :: WEIGHTS(:) ! Scalarizing weights.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
REAL(KIND=R8) :: F ! Scalar output value.
! Local variables.
REAL(KIND=R8) :: V(VTMOP_MOD_P) ! Nonscalarized output vector.
! BLAS function for computing inner products.
REAL(KIND=R8), EXTERNAL :: DDOT
! Get vector-valued objective.
CALL MOP_P_EVALUATE(C, V, IERR)
IF (IERR .NE. 0) RETURN
! Return weighted objective value.
F = DDOT(VTMOP_MOD_P, WEIGHTS(:), 1, V(:), 1)
RETURN
END FUNCTION P_SCALAR_FUNC

FUNCTION SURROGATE_FUNC(C, IERR) RESULT(F)
! This module procedure uses the private module array VTMOP_MOD_WEIGHTS
! to scalarize the output of VTMOP_MOD_SURROGATES, matches the interface
! of VTMOP_MOD_SCALAR_INT, and can be passed as input to a generic single
! objective optimization procedure.
!
!
! On input:
!
```

```
! C(:) is a design point to be evaluated.
!
!
! On output:
!
! IERR is an integer error flag. Any nonzero value indicates an
!    illegal/missing value, as reported by VTMOP_MOD_SURROGATES.
!
! F is a scalar output, as determined by VTMOP_MOD_WEIGHTS(:).
!
IMPLICIT NONE
! Parameters.
REAL(KIND=R8), INTENT(IN) :: C(:)
INTEGER, INTENT(OUT) :: IERR
REAL(KIND=R8) :: F
! Local variables.
REAL(KIND=R8) :: V(VTMOP_MOD_P)
! BLAS function for computing inner products.
REAL(KIND=R8), EXTERNAL :: DDOT
! Evaluate the surrogates.
CALL VTMOP_MOD_SURROGATES(C, V, IERR)
IF (IERR .NE. 0) RETURN
! Compute the weighted sum.
F = DDOT(VTMOP_MOD_P, VTMOP_MOD_WEIGHTS, 1, V, 1)
RETURN
END FUNCTION SURROGATE_FUNC

! The following procedures define the default surrogate, using
! the module LINEAR_SHEPARD from SHEPPACK (ACM TOMS Alg. 905).

SUBROUTINE LSHEP_FIT(D, P, N, X_VALS, Y_VALS, FIRST, PARALLEL, DES_TOL, &
                     SCALE_FACT, SHIFT_FACT, IERR)
! This subroutine fits all P surrogate using the module LINEAR_SHEPARD
! from SHEPPACK.
!
! Thacker, William I., J. Zhang, L. T. Watson, J. B. Birch, M. A. Iyer, and
! M. W. Berry. Algorithm 905: SHEPPACK: Modified Shepard algorithm for
! interpolation of scattered multivariate data. ACM Trans. Math. Softw. (TOMS)
! 37.3 (2010): 34.
!
!
! On input:
!
! D is the dimension of the design space.
!
! P is the dimension of the objective space.
!
! N is the current size of the internal database.
!
! X_VALS(1:D,1:N) is a real vector containing the current database
!    of evaluated design points.
!
! Y_VALS(1:P,1:N) is a real vector containing the current database
!    of corresponding objective values.
!
! FIRST is a logical type that specifies whether this is the first iteration
!    of the algorithm. In the first iteration, since data is sparse, the
!    radius of influence for each design point is doubled.
!
! PARALLEL is a logical type that specifies whether the P LSHEP models should
!    be fit in parallel, using OpenMP.
!
! DES_TOL is a real type that specifies the design space tolerance.
!
! SCALE_FACT(1:D) is a real type that specifies the rescale factor for each
!    dimension of the input data. In particular, each point in X_VALS is
!    transformed by (X_VALS(:,I) - SHIFT_FACT(:)) / SCALE_FACT(:).
!
! SHIFT_FACT(1:D) is a real type that specifies the shift factor for each
!    dimension of the input data. In particular, each point in X_VALS is
!    transformed by (X_VALS(:,I) - SHIFT_FACT(:)) / SCALE_FACT(:).
!
!
! On output:
!
! IERR is an integer error flag.
!
!  00 : Normal output. Successfully fit the P LSHEP models.
!
!  1x : An illegal input was supplied.
!       11 : The problem dimensions D, P, or N contain illegal values.
!       12 : The sizes of the databases X_VALS(:,:) and Y_VALS(:,:) do not
!            match the problem dimensions
!  2x : Error reported by the LSHEP fit subroutine.
!       21 : The number of non-NaN values in the database was not enough
!            to fit the LSHEP models. Try increasing the search budget,
!            or consider whether the cost function is defined within
!            the given bound constraints.
!  3x : A memory allocation error occurred.
!       30 : A memory allocation error occurred.
!       31 : A memory deallocation error occurred.
!
```

```
!
! The following module variables are also altered on output.
!
! The P local linear fits are stored in the private module array
! LSHEP_A(:,:,1:P) and the radii of influence in the private module array
! LSHEP_RW(:,1:P). Also, this subroutine makes copies of the current dataset
! in LSHEP_XVALS(:,:) and LSHEP_FVALS(:,:), along with problem dimensions
! in LSHEP_D, LSHEP_P, and LSHEP_N_PTS. LSHEP_DES_TOL is set to the value
! DES_TOL. LSHEP_SCALE and LSHEP_SHIFT are set to SCALE_FACT and SHIFT_FACT,
! respectively.
!
! Finally, a taboo list LSHEP_TABOO(:,:) is created with length LSHEP_N_TABOO.
! If there are any missing values (marked as NaN values) in Y_VALS, then
! these entries appear in LSHEP_TABOO(:,:).
!
!
USE LINEAR_SHEPARD_MOD
USE IEEE_ARITHMETIC
IMPLICIT NONE
! Parameters.
INTEGER, INTENT(IN) :: D ! The dimension of the design space.
INTEGER, INTENT(IN) :: P ! The dimension of the objective space.
INTEGER, INTENT(IN) :: N ! The number of points in X_VALS/Y_VALS.
REAL(KIND=R8), INTENT(IN) :: X_VALS(:,:) ! N points in the design space.
REAL(KIND=R8), INTENT(IN) :: Y_VALS(:,:) ! Corresponding objective values.
LOGICAL, INTENT(IN) :: FIRST ! FIRST is .TRUE. in the 0th iteration.
LOGICAL, INTENT(IN) :: PARALLEL ! Fit the P surrogate models in parallel.
REAL(KIND=R8), INTENT(IN) :: DES_TOL ! Design space tolerance.
REAL(KIND=R8), INTENT(IN) :: SCALE_FACT(:) ! Scale factor for data points.
REAL(KIND=R8), INTENT(IN) :: SHIFT_FACT(:) ! Shift factor for data points.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Local variables.
INTEGER :: I ! Loop indexing variable.
REAL(KIND=R8) :: TMP_FVALS(P,N) ! Temporary list of F values.
REAL(KIND=R8) :: TMP_TABOO(D,N) ! Temporary taboo list.
REAL(KIND=R8) :: TMP_XVALS(D,N) ! Temporary list of x values.
! Check for illegal input dimensions.
IF (D < 1 .OR. P < 2 .OR. N < D+2) THEN
   IERR = 11; RETURN; END IF
! Check that the database dimensions match the input dimensions.
IF ( SIZE(X_VALS, 1) .NE. D .OR.    &
     SIZE(X_VALS, 2) .NE. N .OR.    &
     SIZE(Y_VALS, 1) .NE. P .OR.    &
     SIZE(Y_VALS, 2) .NE. N .OR.    &
     SIZE(SCALE_FACT,1) .NE. D .OR. &
     SIZE(SHIFT_FACT,1) .NE. D ) THEN
   IERR = 12; RETURN; END IF
! Set the LSHEP problem dimensions.
LSHEP_D = D
LSHEP_P = P
LSHEP_DES_TOL = DES_TOL
! Get the SCALE and SHIFT factors.
IF (ALLOCATED(LSHEP_SCALE)) THEN
   DEALLOCATE(LSHEP_SCALE, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 31; RETURN; END IF
END IF
IF (ALLOCATED(LSHEP_SHIFT)) THEN
   DEALLOCATE(LSHEP_SHIFT, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 31; RETURN; END IF
END IF
ALLOCATE(LSHEP_SCALE(D), LSHEP_SHIFT(D), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 30; RETURN; END IF
LSHEP_SCALE(:) = SCALE_FACT(:)
LSHEP_SHIFT(:) = SHIFT_FACT(:)
! Populate the temporary arrays with correct values.
LSHEP_N_PTS = 0
LSHEP_N_TABOO = 0
DO I = 1, N
   ! Check for NAN values.
   IF (ANY(IEEE_IS_NAN(Y_VALS(:,I)))) THEN
      ! Build the taboo list.
      LSHEP_N_TABOO = LSHEP_N_TABOO + 1
      TMP_TABOO(:,LSHEP_N_TABOO) = (X_VALS(:,I)-LSHEP_SHIFT(:))/LSHEP_SCALE(:)
   ELSE
      ! Build the LSHEP surrogate data set.
      LSHEP_N_PTS = LSHEP_N_PTS + 1
      TMP_XVALS(:,LSHEP_N_PTS) = (X_VALS(:,I)-LSHEP_SHIFT(:))/LSHEP_SCALE(:)
      TMP_FVALS(:,LSHEP_N_PTS) = Y_VALS(:,I)
   END IF
END DO
! Check that a reasonable number of non-NaN points were found.
IF (LSHEP_N_PTS > D+1) THEN
   ! Free the module arrays for resizing.
   IF (ALLOCATED(LSHEP_A)) THEN
      DEALLOCATE(LSHEP_A, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 31; RETURN; END IF
   END IF
   IF (ALLOCATED(LSHEP_RW)) THEN
```

```
      DEALLOCATE(LSHEP_RW, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 31; RETURN; END IF
   END IF
   IF (ALLOCATED(LSHEP_XVALS)) THEN
      DEALLOCATE(LSHEP_XVALS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 31; RETURN; END IF
   END IF
   IF (ALLOCATED(LSHEP_FVALS)) THEN
      DEALLOCATE(LSHEP_FVALS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 31; RETURN; END IF
   END IF
   IF (ALLOCATED(LSHEP_TABOO)) THEN
      DEALLOCATE(LSHEP_TABOO, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 31; RETURN; END IF
   END IF
   ! Reallocate LSHEP_A, LSHEP_RW, LSHEP_FVALS, and LSHEP_XVALS.
   ALLOCATE( LSHEP_A(D, LSHEP_N_PTS, P), LSHEP_RW(LSHEP_N_PTS, P),   &
             LSHEP_XVALS(D,LSHEP_N_PTS), LSHEP_FVALS(LSHEP_N_PTS,P), &
             STAT=IERR )
   IF (IERR .NE. 0) THEN
      IERR = 30; RETURN; END IF
! If not enough non-NAN points were found, LSHEP_FIT cannot proceed.
ELSE
   IERR = 21; RETURN; END IF
! Populate the LSHEP arrays.
LSHEP_XVALS(:,:) = TMP_XVALS(:,1:LSHEP_N_PTS)
LSHEP_FVALS(:,:) = TRANSPOSE(TMP_FVALS(:,1:LSHEP_N_PTS))
! If LSHEP_N_TABOO > 0, allocate the taboo list.
IF (LSHEP_N_TABOO > 0) THEN
   ALLOCATE(LSHEP_TABOO(D,LSHEP_N_TABOO), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 30; RETURN; END IF
   LSHEP_TABOO(:,:) = TMP_TABOO(:,1:LSHEP_N_TABOO)
END IF
! This is the beginning of an iteration task parallelism block. This block
! fits P surrogate models in parallel and stores the weights/radii in A and RW.
!$OMP PARALLEL DO SCHEDULE(STATIC) &
!
! The PRIVATE list specifies uninitialized variables, of which each
! thread has a private copy.
!$OMP& PRIVATE(I, IERR), &
!
! Any variables not explicitly listed above receive the SHARED scope
! by default and are visible across all threads.
!$OMP& DEFAULT(SHARED), &
!
! Only execute if in parallel mode.
!$OMP& IF(PARALLEL)
DO I = 1, LSHEP_P
   ! Perform the LSHEP fit for each of the P surrogate models.
   ! There is no need to consider IERR, since the only serious error
   ! case has already been handled.
   CALL LSHEP( LSHEP_D, LSHEP_N_PTS, LSHEP_XVALS, LSHEP_FVALS(:,I), &
               LSHEP_A(:,:,I), LSHEP_RW(:,I), IERR )
END DO
!$OMP END PARALLEL DO
! Reset the error flag to zero to indicate a successful output.
IERR = 0
! Double the radii of influence in the 0th iteration.
IF (FIRST) LSHEP_RW(:,:) = LSHEP_RW(:,:) * 2.0_R8
RETURN
END SUBROUTINE LSHEP_FIT

SUBROUTINE LSHEP_EVAL(C, V, IERR)
! This subroutine evaluates the P surrogate using the module LINEAR_SHEPARD
! from SHEPPACK.
!
! Thacker, William I., J. Zhang, L. T. Watson, J. B. Birch, M. A. Iyer, and
! M. W. Berry. Algorithm 905: SHEPPACK: Modified Shepard algorithm for
! interpolation of scattered multivariate data. ACM Trans. Math. Softw. (TOMS)
! 37.3 (2010): 34.
!
! This subroutine uses the private module variables and arrays LSHEP_D,
! LSHEP_P, LSHEP_N_PTS, LSHEP_XVALS, LSHEP_FVALS, LSHEP_A, LSHEP_DES_TOL,
! and LSHEP_RW, as set by the subroutine LSHEP_FIT.
!
! Also, this subroutine checks the taboo list
! LSHEP_TABOO(1:LSHEP_D,1:LSHEP_N_TABOO)
! for missing values.
!
!
! On input:
!
! C(:) is a real point in the design space.
!
!
! On output:
!
```

```
! V(:) is the objective value predicted at C(:), according to the LSHEP
!    surrogate functions.
!
! IERR is an integer error flag. Any nonzero value indicates an
!    illegal/missing value, as recorded in the taboo list LSHEP_TABOO.
!
USE LINEAR_SHEPARD_MOD
IMPLICIT NONE
! Parameters.
REAL(KIND=R8), INTENT(IN) :: C(:) ! Input point (from the design space).
REAL(KIND=R8), INTENT(OUT) :: V(:) ! Output point (from the objective space).
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Local variables.
INTEGER I ! Loop indexing variable.
REAL(KIND=R8) :: CL(SIZE(C,1)) ! Rescaled input point.
! BLAS function for Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2
! Rescale the input point and store in CL(:).
CL(:) = (C(:)-LSHEP_SHIFT(:)) / LSHEP_SCALE(:)
! First check the taboo list.
DO I = 1, LSHEP_N_TABOO
   IF (DNRM2(LSHEP_D, CL(:) - LSHEP_TABOO(:,I), 1) < LSHEP_DES_TOL) THEN
      IERR = -1; RETURN; END IF
END DO
! Evaluate the surrogate models.
DO I = 1, LSHEP_P
   V(I) = LSHEPVAL( CL, LSHEP_D, LSHEP_N_PTS, LSHEP_XVALS, LSHEP_FVALS(:,I), &
                    LSHEP_A(:,:,I), LSHEP_RW(:,I), IERR )
   IF (IERR .GE. 10) RETURN
END DO
! Reset error flag to success code.
IERR = 0
RETURN
END SUBROUTINE LSHEP_EVAL

! The following is the optimization procedure. It is a lightweight native
! Fortran implementation of the algorithm GPS MADS from the NOMAD software
! package (ACM TOMS Alg. 909).

SUBROUTINE GPSMADS(D, X, LB, UB, OBJ_FUNC, BUDGET, TOL, IERR)
! This is a lightweight implementation of MADS designed for usage with
! computationally cheap surrogate functions, based on the algorithm GPS MADS
! described in
!
! Le Digabel, Sbastien. Algorithm 909: NOMAD: Nonlinear Optimization with
! the MADS Algorithm. ACM Trans. Math. Softw. (TOMS) 37.4 (2011): 15.
!
! All features not relevant for local optimization of computationally cheap
! surrogates, such as quadratic surrogate models for poll ordering, the global
! search phase, the variable neighborhood search, and the taboo list are
! omitted.
!
!
! On input:
!
! D is the dimension of the design space.
!
! X(1:D) contains a design point from which to start the local optimization
!    procedure.
!
! LB(1:D) contains the lower bound constraints for the design space or the
!    current local trust region (LTR).
!
! UB(1:D) contains the upper bound constraints for the design space or the
!    current LTR.
!
! OBJ_FUNC is a subroutine, whose interface matches VTMOP_MOD_SCALAR_INT.
!    OBJ_FUNC returns a scalarization of the surrogate models. Any
!    missing/illegal values requested should trigger OBJ_FUNC to return
!    a nonzero error flag.
!
! BUDGET is the iteration budget for iterations of the algorithm GPS MADS.
!
! TOL is the tolerance for the design space. Once the mesh fineness reaches
!    TOL the algorithm terminates, regardless of the value of BUDGET.
!
!
! On output:
!
! X(:) is a local minimizer of the scalarized surrogate functions, described
!    by OBJ_FUNC.
!
! IERR is an integer error flag.
!
!  00 : Normal output. Successfully converged on a local minimizer, or
!       BUDGET iterations of GPS MADS.
!
!  1x : An illegal input was supplied.
!      10 : The problem dimensions D, P, or N contain illegal values.
!      11 : The dimension of the bound constraint arrays LB(:) or UB(:) do
!           not match the design dimension.
!      12 : LB(:) must be elementwise strictly less than UB(:) - TOL.
```

```
!      13 : X(:) must be a point in the specified bound constraints [LB, UB].
!
IMPLICIT NONE
! Parameter list.
INTEGER, INTENT(IN) :: D ! The dimension of the design space.
REAL(KIND=R8), INTENT(INOUT) :: X(:) ! The starting point for GPSMADS.
REAL(KIND=R8), INTENT(IN) :: LB(:) ! The lower bound constraints.
REAL(KIND=R8), INTENT(IN) :: UB(:) ! The upper bound constraints.
PROCEDURE(VTMOP_MOD_SCALAR_INT) :: OBJ_FUNC ! The scalarized objective function.
INTEGER, INTENT(IN) :: BUDGET ! The iteration budget.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
REAL(KIND=R8), INTENT(IN) :: TOL ! The design space tolerance.
! Local variables.
INTEGER :: I, J ! Loop index variables.
INTEGER :: MIN_POLL ! Minimum poll index.
REAL(KIND=R8) :: X_VAL ! Current X value.
REAL(KIND=R8) :: POLLS(D,2*D) ! List of poll directions.
REAL(KIND=R8) :: POLL_VALS(2*D) ! List of poll values.
REAL(KIND=R8) :: MESH_GPS(D,2*D) ! The GPS mesh.
REAL(KIND=R8) :: MESH_SIZE ! The current mesh size.
REAL(KIND=R8) :: RESCALE(D) ! Rescale factors for bounding box.
! Check for bad inputs.
IF (SIZE(X, 1) .NE. D) THEN
   IERR = 10; RETURN; END IF
IF ( (SIZE(LB, 1) .NE. D) .OR. (SIZE(UB,1) .NE. D) ) THEN
   IERR = 11; RETURN; END IF
IF ( ANY(LB(:) .GE. UB(:) - TOL) ) THEN
   IERR = 12; RETURN; END IF
IF ( ANY(X(:) .GE. UB(:) + TOL) .OR. ANY(X(:) .LE. LB(:) - TOL) ) THEN
   IERR = 13; RETURN; END IF
! Initialize the mesh fineness and rescale factors.
RESCALE(:) = (UB(:) - LB(:)) / 2.0_R8
MESH_SIZE = 1.0_R8
! Generate the GPS mesh.
DO I = 1, D
   MESH_GPS(:,2*I-1) = 0.0_R8
   MESH_GPS(I,2*I-1) = 1.0_R8
   MESH_GPS(:,2*I) = 0.0_R8
   MESH_GPS(I,2*I) = -1.0_R8
END DO
! Initialize the center value.
X_VAL = OBJ_FUNC(X, IERR)
! Avoid missing values.
IF (IERR .NE. 0) THEN
   X_VAL = HUGE(0.0_R8)
   IERR = 0
END IF
! Loop until the iteration budget is exhausted.
DO I = 1, BUDGET ! Stopping condition 1: budget exhausted.
   DO J = 1, 2*D
      ! Get the next poll point.
      POLLS(:,J) = X(:) + (MESH_GPS(:,J) * RESCALE(:) * MESH_SIZE)
      ! Now predict the objective value for the poll.
      IF ( ANY(POLLS(:,J) > UB(:)) .OR. &
           ANY(POLLS(:,J) < LB(:)) ) THEN
         ! Use an extreme barrier approach for bound violations.
         POLL_VALS(J) = HUGE(0.0_R8)
      ELSE
         ! For legal values, query the objective surrogate.
         POLL_VALS(J) = OBJ_FUNC(POLLS(:,J), IERR)
         ! Avoid missing values.
         IF (IERR .NE. 0) THEN
            X_VAL = HUGE(0.0_R8)
            IERR = 0
         END IF
      END IF
   END DO
   ! Check all poll directions for the best result.
   MIN_POLL = MINLOC(POLL_VALS, 1)
   IF (POLL_VALS(MIN_POLL) < X_VAL) THEN
      ! If the result is an improvement, then move to the new poll location.
      X(:) = POLLS(:,MIN_POLL)
      X_VAL = POLL_VALS(MIN_POLL)
   ELSE
      ! Otherwise, decay the mesh size.
      MESH_SIZE = MESH_SIZE * 0.5_R8
      ! If the mesh size has reached its limit, then exit.
      IF (MESH_SIZE < TOL) EXIT ! Stop cond 2: mesh tolerance reached.
   END IF
END DO
RETURN
END SUBROUTINE GPSMADS

! The following are possible global search options. The VTDIRECT_SEARCH
! is adaptive and uses VTDIRECT95 (ACM TOMS Alg. 897), as proposed
! by Deshpande et al. LH_DESIGN is static and returns a batch of function
! evaluations. In general, VTDIRECT_SEARCH makes more effective use of
! each function evaluation. On the other hand, LH_DESIGN has better properties
! for load balancing and can sometimes produce more evenly spaced points
! when the function evaluation budget is small.

SUBROUTINE VTDIRECT_SEARCH( D, P, LB, UB, MAXITERS, FIRST, IERR, &
```

```
                                EPSW, TOL, PARALLEL )
! This is a wrapper for the VTDIRECT95 code, used for performing an adaptive
! global search. VTDIRECT_SEARCH uses VTDIRECT95 to perform a search of the
! design space or the current local trust region (LTR) by running either P or
! P+1 instances of VTDIRECT95.
!
! The VTDIRECT95 code is described in
!
! He, Jian, Layne T. Watson, and Masha Sosonkina. Algorithm 897:
! VTDIRECT95: serial and parallel codes for the global optimization algorithm
! DIRECT. ACM Trans. Math. Softw. (TOMS) 36.3 (2009): 17.
!
! This wrapper function uses the module array VTMOP_MOD_WEIGHTS(:,:) and
! the module subroutines SCALAR_FUNC and P_SCALAR_FUNC to guide the search.
! The resulting function evaluations are stored in the internal module
! database. This subroutine is not suitable for usage outside of VTVTMOP.
!
!
! On input:
!
! D is the dimension of the design space.
!
! P is the dimension of the objective space.
!
! LB(1:D) contains the lower bound constraints for the design space or the
!    current LTR.
!
! UB(1:D) contains the upper bound constraints for the design space or the
!    current LTR.
!
! MAXITERS is the iteration budget for iterations of the algorithm DIRECT.
!
! FIRST is a logical switch, which specifies whether this is the
!    zeroth iteration of the algorithm. If FIRST=.FALSE., then each objective
!    is optimized individually. If FIRST=.TRUE., one additional scalarization
!    is applied, which equally weights all P objective functions.
!
! IERR is an integer error flag, which relays error messages from
!    VTdirect.
!
!  00 : Normal output. VTdirect has successfully run for MAXITERS iterations,
!       or until some other termination condition, for each of the weightings
!       in the ADAPTIVE_WEIGHTS(:,:) array.
!
!  1x : Input data error.
!     10 : D < 2.
!     11 : Assumed shape array L, U, W, or X does not have size D.
!     12 : Some lower bound is >= the corresponding upper bound.
!     13 : MIN_DIA, OBJ_CONV, or EPS is invalid or below the roundoff level.
!     14 : None of MAX_EVL, MAX_ITER, MIN_DIA, and OBJ_CONV are specified;
!          there is no stopping rule.
!     15 : Invalid SWITCH value.
!     16 : SWITCH = 0 and EPS > 0 are incompatible.
!  2x : Memory allocation error or failure.
!     20 : BoxMatrix type allocation.
!     21 : BoxLink or BoxLine type allocation.
!     22 : int_vector or real_vector type allocation.
!     23 : HyperBox type allocation.
!     24 : BOX_SET is allocated with a wrong problem dimension.
!
!  3x : The following errors are not reported by VTdirect, and are specific
!       to this usage case.
!     30 : P cannot be less than 2.
!     31 : A memory allocation error occurred.
!     32 : A memory deallocation error occurred.
!
!
! Optional input arguments:
!
! EPSW is the tolerance for zero-valued weights. All zero-valued weights
!    (for the single objective problems) are instead set to EPSW.
!    By default, EPSW is the fourth-root of the machine epsilon.
!    EPSW cannot be decreased below the square-root of the machine epsilon.
!
! TOL is the tolerance for the design space. VTDIRECT95 will not divide a
!    box whose diameter is less than 2.0 * TOL. By default, TOL is the
!    square-root of the machine epsilon. TOL cannot be decreased below its
!    default value.
!
! PARALLEL is a logical flag, which specifies whether or not
!
USE SVTDIRECT_MOD, ONLY : SVTDIRECT ! Modified serial code for VTdirect95.
USE BVTDIRECT_MOD, ONLY : BVTDIRECT ! Batched parallel code for Vtdirect.f95.
IMPLICIT NONE
! Parameter list.
INTEGER, INTENT(IN) :: D, P ! The dimension of the design and objective spaces.
REAL(KIND=R8), INTENT(IN) :: LB(:), UB(:) ! Lower and upper bound constraints.
INTEGER, INTENT(IN) :: MAXITERS ! Maximum number of iterations for VTdirect95.
LOGICAL, INTENT(IN) :: FIRST ! FIRST = .TRUE. for the 0th iteration.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Optional parameters.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPSW ! Fudge factor for zero weights.
```

```
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: TOL ! Design space tolerance.
LOGICAL, OPTIONAL, INTENT(IN) :: PARALLEL ! Parallel function evaluations.
! Local variables.
INTEGER :: I, J ! Loop indices and temporary variables.
INTEGER :: IERR_PRIV(P+1) ! Local array of IERR flags.
INTEGER :: ITMP ! Temporary integer variable.
INTEGER :: MAXITERSL ! Iteration limit.
REAL(KIND=R8) :: ADAPTIVE_WEIGHTS(P,P+1) ! Adaptive weight vectors.
REAL(KIND=R8) :: EPS ! Box tolerance.
REAL(KIND=R8) :: EPSWL ! Weight fudge factor.
REAL(KIND=R8) :: RTMP ! Temporary real variable.
REAL(KIND=R8) :: FMIN, X(D) ! Dummy variables required by VTDIRECT.
LOGICAL :: PARALLEL_L ! Local copy of PARALLEL.
! Check for illegal problem dimensions.
IF (P < 2) THEN
   IERR = 30; RETURN; END IF
! Set the minimum box diameter to twice the tolerance.
EPS = 2.0_R8 * SQRT(EPSILON(0.0_R8))
IF (PRESENT(TOL)) THEN
   IF (TOL > EPS / 2.0_R8) EPS = 2.0_R8 * TOL
END IF
! Set the zero weight fudge factor to the fourth root of machine EPSILON.
EPSWL = EPSILON(0.0_R8) ** 0.25_R8
IF (PRESENT(EPSW)) THEN
   ! The fudge factor must be at least the square root of EPSILON.
   IF (EPSW .GE. SQRT(EPSILON(0.0_R8))) EPSWL = EPSW
END IF
! Set the parallel execution mode.
PARALLEL_L = .FALSE.
IF (PRESENT(PARALLEL)) PARALLEL_L = PARALLEL
! Set the iteration limit.
MAXITERSL = MAXITERS
VTMOP_MOD_P = P
! Call different implementations of VTDIRECT95, depending on whether the
! execution mode is parallel.
IF (PARALLEL_L) THEN
   ! Generate P objective weights.
   J = P
   DO I = 1, P
      ADAPTIVE_WEIGHTS(:,I) = EPSWL
      ADAPTIVE_WEIGHTS(I,I) = 1.0_R8 - (REAL(P-1,KIND=R8) * EPSWL)
   END DO
   ! The (P+1)th weight vector will only be used in the 0th iteration.
   IF (FIRST) THEN
      J = P+1
      ADAPTIVE_WEIGHTS(:,P+1) = 1.0_R8 / REAL(P, KIND=R8)
   END IF
   ! Call VTDIRECT once for each objective function.
   !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
   !
   ! Any variables not explicitly listed above receive the SHARED scope
   ! by default and are visible across all threads.
   !$OMP& DEFAULT(SHARED), &
   !
   ! The PRIVATE list specifies uninitialized variables, of which each
   ! thread has a private copy.
   !$OMP& PRIVATE(FMIN, I, ITMP, RTMP, X)
   DO I = 1, J
      ITMP = MAXITERSL
      RTMP = EPS
      ! Perform global optimization.
      CALL BVTDIRECT( D, P, LB, UB, ADAPTIVE_WEIGHTS(:,I),  &
                      P_SCALAR_FUNC, X, FMIN, IERR_PRIV(I), &
                      MIN_DIA=RTMP, MAX_ITER=ITMP )
   END DO
   !$OMP END PARALLEL DO
   ! Post process the array of error flags.
   DO I = 1, J
      IF (IERR_PRIV(I) .GE. 10) THEN
         IERR = IERR_PRIV(I); RETURN; END IF
   END DO
ELSE
   ! Execute the serial version.
   ! Reallocate the adaptive weights array to the correct size.
   IF(ALLOCATED(VTMOP_MOD_WEIGHTS)) THEN
      DEALLOCATE(VTMOP_MOD_WEIGHTS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 32; RETURN; END IF
   END IF
   ALLOCATE(VTMOP_MOD_WEIGHTS(P), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 31; RETURN; END IF
   ! Call VTDIRECT once for each objective function.
   DO I = 1, P
      VTMOP_MOD_WEIGHTS(:) = EPSWL
      VTMOP_MOD_WEIGHTS(I) = 1.0_R8 - (REAL(P-1,KIND=R8) * EPSWL)
      ITMP = MAXITERSL
      RTMP = EPS
      ! Perform global optimization.
      CALL SVTDIRECT( D, LB, UB, SCALAR_FUNC, X(:), FMIN, IERR,  &
                      MIN_DIA=RTMP, MAX_ITER=ITMP )
      IF (IERR .GE. 10) RETURN
```

```
   END DO
   ! If this is the 0th iteration, call VTDIRECT once over all objectives.
   IF (FIRST) THEN
      VTMOP_MOD_WEIGHTS(:) = 1.0_R8 / REAL(P, KIND=R8)
      ITMP = MAXITERSL
      RTMP = EPS
      ! Perform global optimization.
      CALL SVTDIRECT( D, LB, UB, SCALAR_FUNC, X(:), FMIN, IERR,  &
                      MIN_DIA=RTMP, MAX_ITER=ITMP )
      IF (IERR .GE. 10) RETURN
   END IF
   ! Free VTMOP_MOD_WEIGHTS, which contains no useful information on output.
   DEALLOCATE(VTMOP_MOD_WEIGHTS, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 32; RETURN; END IF
END IF
RETURN
END SUBROUTINE VTDIRECT_SEARCH

SUBROUTINE LH_DESIGN(D, LB, UB, N_PTS, CAND_PTS, IERR, TOL, XI)
! This wrapper generates a Latin hypercube design of experiment using the
! QNSTOP subroutine LATINDESIGN, as implemented in
!
! Amos, B. D., D. R. Easterling, L. T. Watson, W. I. Thacker, B. S. Castle,
! and M. W. Trosset. Algorithm XXX: QNSTOP -- Quasi-Newton Algorithm for
! Stochastic Optimization. Tech Report. Virginia Polytechnic Institute and
! State University (2014).
!
! This design can be used to perform an exploration of the entire design
! space, or the current local trust region (LTR).
!
!
! On input:
!
! D is the dimension of the design space.
!
! LB(1:D) contains the lower bound constraints for the design space or the
!    current LTR.
!
! UB(1:D) contains the upper bound constraints for the design space or the
!    current LTR.
!
! N_PTS is an integer specifying the size of the requested design.
!
!
! On output:
!
! CAND_PTS(:,:) is a real allocatable array. On output, CAND_PTS(:,:)
!    is allocated to size D by N_PTS, and contains a Latin hypercube design.
!
! IERR is an integer error flag, which relays error messages from
!    VTdirect.
!
!  00 : Normal output. VTdirect has successfully run for MAXITERS iterations,
!       or until some other termination condition, for each of the weightings
!       in the ADAPTIVE_WEIGHTS(:,:) array.
!
!  1x : Input data error.
!     10 : D does not match the size of LB(:) or UB(:).
!     11 : LB(:) must be componentwise less than UB(:) - TOL.
!     12 : When present, the size of XI(:) must match with D.
!     13 : When present, XI(:) must be within the bound constraints [LB, UB].
!  20 : A memory allocation error has occurred.
!
!
! Optional input arguments:
!
! TOL is the tolerance for the design space. This value is only used for
!    sanity checks. By default, TOL is the square-root of the machine
!    epsilon.
!
! XI(1:D) is an initial point in the design space, to include in the Latin
!    hypercube design. When supplied, N_PTS additional points are generated,
!    with the understanding that XI has already been evaluated.
!
USE QNSTOPS_MOD, ONLY : LATINDESIGN
IMPLICIT NONE
! Parameter list.
INTEGER, INTENT(IN) :: D ! The dimension of the design space.
REAL(KIND=R8), INTENT(IN) :: LB(:), UB(:) ! Lower and upper bound constraints.
INTEGER, INTENT(IN) :: N_PTS ! Number of points in the design.
REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: CAND_PTS(:,:) ! The output design.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Optional Parameters.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: TOL ! Design space tolerance.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: XI(:) ! An initial design point.
! Local variables.
REAL(KIND=R8) :: DES_PTS(D,N_PTS+1) ! Set of new candidate points.
REAL(KIND=R8) :: TOL_L ! Local copy of the tolerance.
REAL(KIND=R8) :: XIL(D) ! Local copy of initial point.
! Check for bad input dimensions.
IF (SIZE(LB,1) .NE. D .OR. SIZE(UB,1) .NE. D) THEN
```

```
   IERR = 10; RETURN; END IF
! Get optional inputs.
TOL_L = SQRT(EPSILON(0.0_R8))
IF (PRESENT(TOL)) THEN
   IF (TOL > TOL_L) TOL_L = TOL
END IF
! Check that the bounds are appropriate.
IF (ANY(UB(:) - LB(:) < TOL_L)) THEN
   IERR = 11; RETURN; END IF
! Different rules depending on whether an initial point is supplied.
IF (PRESENT(XI)) THEN
   ! Check that XI is of the correct dimension.
   IF (SIZE(XI,1) .NE. D) THEN
      IERR = 12; RETURN; END IF
   ! Check that XI is within the bounds.
   IF (ANY(XI(:) > UB(:) + TOL_L) .OR. ANY(XI(:) < LB(:) - TOL_L)) THEN
      IERR = 13; RETURN; END IF
   ! Create a design with N_PTS+1 points, so that XI(:) is included in the
   ! design but N_PTS new points are generated.
   IF (N_PTS > 0) THEN
      CALL LATINDESIGN(D, N_PTS+1, LB, UB, XI, DES_PTS)
   ELSE
      DES_PTS(1:D,1) = XI(:)
   END IF
ELSE
   ! Otherwise, randomly generate XIL.
   CALL RANDOM_NUMBER(XIL(:))
   ! Rescale to bounds.
   XIL(:) = XIL(:) * (UB(:) - LB(:)) + LB(:)
   ! Create a design with N_PTS points.
   IF (N_PTS > 1) THEN
      CALL LATINDESIGN(D, N_PTS, LB, UB, XIL, DES_PTS(1:D,1:N_PTS))
   ELSE
      DES_PTS(1:D,1) = XIL(:)
   END IF
END IF
! Allocate the output array.
ALLOCATE(CAND_PTS(D,N_PTS), STAT=IERR)
IF (IERR .NE. 0) THEN
   IERR = 20; RETURN; END IF
! Copy the points into the output array.
CAND_PTS(:,:) = DES_PTS(:,1:N_PTS)
RETURN
END SUBROUTINE LH_DESIGN

END MODULE VTMOP_MOD
```

```
! The following module contains public interfaces for driver and worker
! subroutines.
MODULE VTMOP_LIB
USE VTMOP_MOD
! The default scope is private.
PRIVATE
! The following data types and structures are public.
PUBLIC :: VTMOP_TYPE, R8
! The following VTMOP_MOD interfaces are public.
PUBLIC :: VTMOP_MOD_OBJ_INT, VTMOP_MOD_SCALAR_INT, VTMOP_MOD_LOCAL_INT, &
          VTMOP_MOD_SFIT_INT, VTMOP_MOD_SEVAL_INT, DELAUNAYGRAPH
! The following VTMOP_MOD worker and driver subroutines are public.
PUBLIC :: VTMOP_INIT, VTMOP_LTR, VTMOP_OPT, VTMOP_FINALIZE, VTMOP_SOLVE
! The following auxiliary subroutines are public.
PUBLIC :: GPSMADS, LSHEP_FIT, LSHEP_EVAL, LH_DESIGN
! The following checkpointing subroutines are public.
PUBLIC :: VTMOP_CHKPT_NEW, VTMOP_CHKPT, VTMOP_CHKPT_RECOVER, VTMOP_NEW_DATA, &
          VTMOP_SAVE_DATA, VTMOP_RECOVER_DATA

! Public interface for the driver subroutine VTMOP_SOLVE, which approximates the
! Pareto optimal set for the Fortran subroutine OBJ_FUNC using VTMOP_MOD.
INTERFACE
   SUBROUTINE VTMOP_SOLVE(D, P, LB, UB, OBJ_FUNC, EFFICIENT_X, PARETO_F, IERR, &
                          ADAPTIVE_SEARCH, BB_BUDGET, MAXITERS, SEARCH_BUDGET, &
                          INITIAL_SBUDGET, LOPT_BUDGET, DECAY, DES_TOL, EPS,   &
                          EPSW, OBJ_TOL, MIN_RADF, TRUST_RADF, DES_PTS,        &
                          OBJ_PTS, LOCAL_OPT, FIT_SURROGATES, EVAL_SURROGATES, &
                          PFLAG, ICHKPT)
      USE REAL_PRECISION, ONLY : R8
      INTEGER, INTENT(IN) :: D
      INTEGER, INTENT(IN) :: P
      REAL(KIND=R8), INTENT(IN) :: LB(:)
      REAL(KIND=R8), INTENT(IN) :: UB(:)
      REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: EFFICIENT_X(:,:)
      REAL(KIND=R8), ALLOCATABLE, INTENT(OUT) :: PARETO_F(:,:)
      INTEGER, INTENT(OUT) :: IERR
      LOGICAL, OPTIONAL, INTENT(IN) :: ADAPTIVE_SEARCH
      INTEGER, OPTIONAL, INTENT(IN) :: BB_BUDGET
      INTEGER, OPTIONAL, INTENT(IN) :: MAXITERS
      INTEGER, OPTIONAL, INTENT(IN) :: SEARCH_BUDGET
      INTEGER, OPTIONAL, INTENT(IN) :: INITIAL_SBUDGET
      INTEGER, OPTIONAL, INTENT(IN) :: LOPT_BUDGET
      REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DECAY
      REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DES_TOL
      REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPS
```

```
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPSW
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: OBJ_TOL
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: MIN_RADF
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: TRUST_RADF
REAL(KIND=R8), ALLOCATABLE, OPTIONAL, INTENT(INOUT) :: DES_PTS(:,:)
REAL(KIND=R8), ALLOCATABLE, OPTIONAL, INTENT(INOUT) :: OBJ_PTS(:,:)
OPTIONAL :: LOCAL_OPT
OPTIONAL :: FIT_SURROGATES
OPTIONAL :: EVAL_SURROGATES
INTEGER, OPTIONAL, INTENT(IN) :: PFLAG
INTEGER, OPTIONAL, INTENT(IN) :: ICHKPT
INTERFACE
   SUBROUTINE OBJ_FUNC(C, V, IERR)
      USE REAL_PRECISION, ONLY : R8
      REAL(KIND=R8), INTENT(IN) :: C(:)
      REAL(KIND=R8), INTENT(OUT) :: V(:)
      INTEGER, INTENT(OUT) :: IERR
   END SUBROUTINE OBJ_FUNC

   SUBROUTINE LOCAL_OPT(D, X, LB, UB, OBJ_FUNC, BUDGET, TOL, IERR)
      USE REAL_PRECISION, ONLY : R8
      INTEGER, INTENT(IN) :: D
      REAL(KIND=R8), INTENT(INOUT) :: X(:)
      REAL(KIND=R8), INTENT(IN) :: LB(:)
      REAL(KIND=R8), INTENT(IN) :: UB(:)
      INTERFACE
         FUNCTION OBJ_FUNC(C, IERR) RESULT(F)
            USE REAL_PRECISION, ONLY : R8
            REAL(KIND=R8), INTENT(IN) :: C(:)
            INTEGER, INTENT(OUT) :: IERR
            REAL(KIND=R8) :: F
         END FUNCTION OBJ_FUNC
      END INTERFACE
      INTEGER, INTENT(IN) :: BUDGET
      REAL(KIND=R8), INTENT(IN) :: TOL
      INTEGER, INTENT(OUT) :: IERR
   END SUBROUTINE LOCAL_OPT

   SUBROUTINE FIT_SURROGATES( D, P, N, X_VALS, Y_VALS, FIRST, &
                              PARALLEL, DES_TOL, SCALE_FACT,  &
                              SHIFT_FACT, IERR )
      USE REAL_PRECISION, ONLY : R8
      ! Parameters.
      INTEGER, INTENT(IN) :: D
      INTEGER, INTENT(IN) :: P
      INTEGER, INTENT(IN) :: N
      REAL(KIND=R8), INTENT(IN) :: X_VALS(:,:)
      REAL(KIND=R8), INTENT(IN) :: Y_VALS(:,:)
      LOGICAL, INTENT(IN) :: FIRST
      LOGICAL, INTENT(IN) :: PARALLEL
      REAL(KIND=R8), INTENT(IN) :: DES_TOL
      REAL(KIND=R8), INTENT(IN) :: SCALE_FACT(:)
      REAL(KIND=R8), INTENT(IN) :: SHIFT_FACT(:)
      INTEGER, INTENT(OUT) :: IERR
   END SUBROUTINE FIT_SURROGATES

   SUBROUTINE EVAL_SURROGATES(C, V, IERR)
      USE REAL_PRECISION, ONLY : R8
      ! Parameters.
      REAL(KIND=R8), INTENT(IN) :: C(:)
      REAL(KIND=R8), INTENT(OUT) :: V(:)
      INTEGER, INTENT(OUT) :: IERR
   END SUBROUTINE EVAL_SURROGATES
 END INTERFACE
END SUBROUTINE VTMOP_SOLVE
END INTERFACE

END MODULE VTMOP_LIB

! The following subroutine is the main driver/solver for VTMOP.

SUBROUTINE VTMOP_SOLVE( D, P, LB, UB, OBJ_FUNC, EFFICIENT_X, PARETO_F, IERR, &
                        ADAPTIVE_SEARCH, BB_BUDGET, MAXITERS, SEARCH_BUDGET, &
                        INITIAL_SBUDGET, LOPT_BUDGET, DECAY, DES_TOL, EPS,   &
                        EPSW, OBJ_TOL, MIN_RADF, TRUST_RADF, DES_PTS,        &
                        OBJ_PTS, LOCAL_OPT, FIT_SURROGATES, EVAL_SURROGATES, &
                        PFLAG, ICHKPT )
! This is the driver subroutine for the adaptive weighting scheme described in
!
! Deshpande, Shubhangi, Layne T. Watson, and Robert A. Canfield.
! "Multiobjective optimization using an adaptive weighting scheme."
! Optimization Methods and Software 31.1 (2016): 110-133.
!
!
! On input:
!
! D is the dimension of the design space.
!
! P is the dimension of the objective space.
!
! LB(1:D) is the real vector of lower bound constraints for the
!    D design variables.
```

```
!
! UB(1:D) is the real vector of upper bound constraints for the
!    D design variables.
!
! OBJ_FUNC is a subroutine, defining the objective function F(X), with
!    signature OBJ_FUNC(C, F, IERR). When called, OBJ_FUNC returns F(C),
!    with IERR=0 for a successful call and IERR/=0 for an unsuccessful
!    call.
!
! EFFICIENT_X(:,:) is an uninitialized ALLOCATABLE matrix of type REAL.
!
! PARETO_F(:,:) is an uninitialized ALLOCATABLE matrix of type REAL.
!
!
! On output:
!
! EFFICIENT_X(1:D,1:N) is a vector containing efficient design points
! corresponding to the objective values in PARETO_F.
!
! PARETO_F(1:P,1:N) is a vector containing N objective points on the
! Pareto front.
!
! IERR is an integer error flag. The error codes are as follows:
!
! Hundreds digit:
!  0xx : Successful computation, normal stopping criterion triggered.
!    Tens digit:
!      00x : Stopping conditions triggered.
!       Ones digit:
!         001 : Stopping criterion 1: budget exhausted.
!         002 : Stopping criterion 2: max iterations exceeded.
!         003 : Stopping criterion 3: maximum accuracy attained.
!
!  1xx : Error detected during input or initialization.
!   Tens digit:
!     11x : The input parameters contained illegal dimensions or values.
!       Ones digit:
!         110 : D (design dimension) must be a positive integer.
!         111 : P (objective dimension) must be at least two.
!         112 : The lead dimension of LB(:) must match D.
!         113 : The lead dimension of UB(:) must match D.
!         114 : LB(:) must be elementwise strictly less than UB(:) - DES_TOL.
!     12x : The optional dummy arguments contained illegal values.
!       Ones digit:
!         120 : BB_BUDGET must be a positive number.
!         121 : MAXITERS must be positive.
!         122 : SEARCHBUDGET must be at least 1, and INITIAL_SBUDGET must
!               be nonnegative.
!         123 : LOPT_BUDGET must be positive.
!         124 : DECAY must be in the range (EPS, 1-EPS).
!         125 : TRUST_RADF must be larger than MIN_RADF.
!         126 : If either DES_PTS or OBJ_PTS are present, then
!               both DES_PTS and OBJ_PTS must be present.
!         127 : If either DES_PTS or OBJ_PTS are allocated on input, then both
!               DES_PTS and OBJ_PTS must be allocated and their dimensions
!               must agree with the problem dimensions.
!         128 : If either FIT_SURROGATES or EVAL_SURROGATES are present,
!               then both must be present.
!     13x : A memory allocation error has occurred.
!       Ones digit:
!         130 : A memory allocation error occurred.
!         131 : A memory deallocation error occurred.
!
!  2xx : Error detected during VTMOP_LTR procedure.
!    See VTMOP_LTR definition for further details.
!
!  3xx : Error detected during VTMOP_OPT procedure.
!    See VTMOP_OPT definition for further details.
!
!  4xx : Error detected during VTMOP_FINALIZE procedure.
!    See VTMOP_FINALIZE definition for further details.
!
!  5xx : Error thrown by DELAUNAYSPARSE, while building DELAUNAYGRAPH.
!    Tens and ones digit carry the specific error code from DELAUNAYSPARSE.
!    See delsparse.f90 for details.
!  599 : Error thrown by LAPACK subroutine DGESVD while performing PCA,
!    after DELAUNAYSPARSE reported error code 31, meaning the current
!    Pareto set lies in a lower-dimensional affine subspace.
!
!  6xx : Error thrown by FIT_SURROGATES.
!    Tens and ones digits carry the error code from the FIT_SURROGATES
!    subroutine.
!
!  7xx : Error thrown by LOCAL_OPT.
!    Tens and ones digits carry the error code from the LOCAL_OPT subroutine.
!
!  8xx : Error thrown by GLOBAL_SEARCH.
!    Tens and ones digits carry the error code from the GLOBAL_SEARCH
!    subroutine.
!
!  9xx : A checkpointing error was thrown.
!    901 : The checkpoint was recovered, but does not match the given
```

```
!          problem dimensions.
!    91x : Error while checkpointing: The VTMOP data object was invalid.
!          This is rare and was likely caused by a hardware failure or
!          segmentation fault.
!    92x : Error while checkpointing: There was an error while creating a
!          new checkpoint file.
!    93x : Error while checkpointing: There was an error while writing
!          iteration information to the checkpoint file.
!          This is rare and was likely caused by a hardware failure or
!          segmentation fault.
!    94x : Error while recovering: There was an error while reading data
!          in from the checkpoint file.
!    95x : Error while recovering: A memory management error occurred during
!          recovery.
!    960 : A sanity check failed during recovery, indicating that the
!          checkpoint file may have been corrupted.
!
!
! Optional input and output arguments:
!
! ADAPTIVE_SEARCH is a Boolean value specifying whether to perform each
!    search adaptively using VTdirect95 (.TRUE.), or by Latin hypercube
!    design of experiment (.FALSE.). By default, ADAPTIVE_SEARCH = .TRUE.
!
! BB_BUDGET is an integer input specifying the total blackbox function
!    evaluation budget. Note that when DES_PTS and OBJ_PTS are present,
!    the initial database counts toward the budget. By default, BB_BUDGET
!    is 1000.
!
! MAXITERS is an integer input specifying the maximum number of iterations
!    over the outer loop. By default, MAXITERS is unlimited.
!
! SEARCH_BUDGET is an integer input for controlling the budget of the
!    search phase. If ADAPTIVE_SEARCH=.TRUE., then SEARCH_BUDGET specifies
!    the number of iterations for each instance of VTdirect95. By default,
!    5 iterations are allowed. If ADAPTIVE_SEARCH=.FALSE., then SEARCH_BUDGET
!    specifies the size of each experimental design. By default, there are
!    8*D points in each design. In rare cases, SEARCH_BUDGET could be set
!    to zero, in which case no exploration is done within each local trust
!    region (LTR). Instead, data from a previous iteration is used to fit
!    the surrogate models. Skipping the exploration phase is not recommended,
!    but could save computations when an extremely thorough initial search
!    has been performed.
!
! INITIAL_SBUDGET is an integer input specifying the value of SEARCH_BUDGET
!    for the zeroth iteration. By default, when ADAPTIVE_SEARCH=.TRUE.,
!    INITIAL_SBUDGET=10, and when ADAPTIVE_SEARCH=.FALSE.,
!    INITIAL_SBUDGET=16*D*D. It is recommended that INITIAL_SBUDGET be
!    significantly larger than SEARCH_BUDGET, since the initial search drives
!    the global convergence of VTMOP_MOD. However, if a large well-distributed
!    initial database is supplied, then INITIAL_SBUDGET could be small.
!    In extreme cases where DES_PTS and OBJ_PTS are supplied and contain a
!    large initial database, INITIAL_SBUDGET could be set to zero. In most
!    cases, however, setting INITIAL_SBUDGET too low will cause poor
!    performance and could lead to an error.
!
! LOPT_BUDGET is an integer input specifying the number of surrogate model
!    evaluations allowed during the local optimization phase. Since the
!    local optimization subroutine does not directly query the blackbox
!    function, this budget can be made as large as necessary to guarantee
!    convergence. By default, LOPT_BUDGET=2500.
!
! DECAY is a real input specifying the decay rate for the LTR
!    radius. This value affects how many times an isolated point can
!    be the center of a LTR before it is discarded. By default, DECAY = 0.5.
!
! DES_TOL is the tolerance for the design space. A design point that
!    is within DES_TOL of an evaluated design point will not be reevaluated.
!    The default value for DES_TOL is the square-root of the working precision
!    EPS. Note that any value that is smaller than the working precsion EPS
!    will be ignored and EPS will be used.
!
! EPS is a real input, which specifies the working precision of the
!    machine. The default value for EPS is SQRT(EPSILON), where EPSILON
!    is the unit roundoff. Note that if the value supplied is smaller than
!    the default value then the default value will be used.
!
! EPSW is a small positive number, which is used as the fudge factor for
!    zero-valued weights. A zero-valued weight does not guarantee Pareto
!    optimality. Therefore, all zero weights are set to EPSW. The appropriate
!    value of EPSW is problem dependent. By default, EPSW is the fourth root
!    of EPSILON (the unit roundoff). Note that any value that is smaller
!    than SQRT(EPSILON) is ignored and SQRT(EPSILON) will be used.
!
! OBJ_TOL is the tolerance for the objective space. An objective point
!    that is within OBJ_TOL of being dominated by another objective point
!    will be treated as such. The default value of OBJ_TOL is the
!    square-root of EPS. Note that any value that is smaller than the
!    working precsion EPS will be ignored and EPS will be used.
!
! MIN_RADF is the smallest value for the fraction r defining the trust region
!    box dimensions r * (UB - LB), before an isolated point is abandoned.
```

```
!    By default, MIN_RADF = 0.1 * TRUST_RADF, and is also set to this default
!    value if it is less than DES_TOL. After MIN_RADF and TRUST_RADF are set,
!    MIN_RADF < TRUST_RADF must hold.
!
! TRUST_RADF defines the initial trust region centered at an isolated
!    point X as [X - TRUST_RADF * (UB - LB), X + TRUST_RADF * (UB - LB)]
!    intersected with [LB, UB].  By default, TRUST_RADF = 0.2, and is also set
!    to this value if the value given is outside the interval
!    (DES_TOL, 1 - DES_TOL).
!
! DES_PTS(D,:) and OBJ_PTS(P,:) are real allocatable arrays. If
!    either DES_PTS or OBJ_PTS is present, then both must appear.
!    If DES_PTS or OBJ_PTS is allocated on input, then both must be
!    allocated and their second dimensions must match. They should
!    contain a database of pre-evaluated design points and corresponding
!    objective values. This initial database is used to improve the quality
!    of the surrogate models, and avoid redundant function evaluations.
!
!    On output, both DES_PTS and OBJ_PTS are reallocated so that their
!    second dimensions match the size of the final database. They contain
!    the full database of all design points and corresponding objective
!    values that were evaluated by VTVTMOP.
!
! LOCAL_OPT is a subroutine, whose interface matches VTMOP_MOD_LOCAL_INT.
!    LOCAL_OPT is used to optimize the surrogate model. The default value
!    for LOCAL_OPT = GPSMADS, a lightweight Fortran implementation of the
!    algorithm GPS MADS from NOMAD (ACM TOMS Alg. 909).
!
! FIT_SURROGATES is a module subroutine that fits P surrogate models, by
!    setting its internal module variables. The interface for FIT_SURROGATES
!    must match VTMOP_MOD_SFIT_INT. By default, FIT_SURROGATES = LSHEP_FIT.
!
! EVAL_SURROGATES is a module subroutine that evaluates the P surrogate
!    models fit by FIT_SURROGATES. The interface for EVAL_SURROGATES must
!    match VTMOP_MOD_SEVAL_INT. By default, EVAL_SURROGATES = LSHEP_EVAL.
!
! PFLAG is an integer input that specifies whether or not iteration tasks
!    and function evaluations should be performed in parallel. Possible
!    values are:
!
!    PFLAG = 1 : Perform function evaluations asynchronously, but do not
!                parallelize iteration tasks.
!    PFLAG = 2 : Parallelize iteration tasks, but perform function evaluations
!                serially.
!    PFLAG = 3 : Perform both function evaluations and iteration tasks in
!                parallel.
!    OTHER     : For all other values of PFLAG, no parallelism is used.
!
! ICHKPT is an integer that specifies the checkpointing status. The
!    checkpoint file and checkpoint unit are "vtmop.chkpt" and 10 by
!    default, but can be adjusted by setting the module variables
!    VTMOP_CHKPTFILE and VTMOP_CHKPTUNIT. Possible values are:
!
!    ICHKPT = 0 : No checkpointing (default setting).
!    ICHKPT < 0 : Recover from the last checkpoint.
!    ICHKPT > 0 : Begin a new checkpoint file.
!
! In recovery mode, the inputs D, P, LB, and UB are still referenced and
!    used for sanity checks. The optional inputs DECAY, DES_TOL, EPS,
!    EPSW, OBJ_TOL, MIN_RADF, and TRUST_RADF are recovered from the previous
!    run and not referenced, even if present. The optional inputs
!    ADAPTIVE_SEARCH, BB_BUDGET, MAXITERS, SEARCH_BUDGET, INITIAL_SBUDGET,
!    and LOPT_BUDGET are not recovered and must be re-set, and thus could
!    be changed from their initial values.
!
USE VTMOP_MOD
IMPLICIT NONE
! Input parameters.
INTEGER, INTENT(IN) :: D ! Dimension of design space.
INTEGER, INTENT(IN) :: P ! Dimension of objective space.
REAL(KIND=R8), INTENT(IN) :: LB(:) ! Lower bound constraints.
REAL(KIND=R8), INTENT(IN) :: UB(:) ! Upper bound constraints.
PROCEDURE(VTMOP_MOD_OBJ_INT) :: OBJ_FUNC ! Vector-valued objective function.
! Output parameters.
REAL(KIND=R8), INTENT(OUT), ALLOCATABLE :: EFFICIENT_X(:,:) ! Efficient set.
REAL(KIND=R8), INTENT(OUT), ALLOCATABLE :: PARETO_F(:,:) ! Pareto set.
INTEGER, INTENT(OUT) :: IERR ! Error flag.
! Optional parameters.
LOGICAL, OPTIONAL, INTENT(IN) :: ADAPTIVE_SEARCH ! Adaptive search option.
INTEGER, OPTIONAL, INTENT(IN) :: BB_BUDGET ! Blackbox budget.
INTEGER, OPTIONAL, INTENT(IN) :: LOPT_BUDGET ! Local optimizer budget.
INTEGER, OPTIONAL, INTENT(IN) :: MAXITERS ! Maximum number of iterations.
INTEGER, OPTIONAL, INTENT(IN) :: SEARCH_BUDGET ! Global search budget.
INTEGER, OPTIONAL, INTENT(IN) :: INITIAL_SBUDGET ! First search budget.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DECAY ! Trust region decay rate.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: DES_TOL ! Design space tolerance.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPS ! Working precision.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPSW ! Fudge factor for zero weights.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: OBJ_TOL ! Objective space tolerance.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: MIN_RADF ! Minimum LTR fraction.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: TRUST_RADF ! Initial LTR fraction.
! {DES|OBJ}_PTS is a database of design points and corresponding objective
```

```
! values that have been pre-evaluated.
REAL(KIND=R8), ALLOCATABLE, OPTIONAL, INTENT(INOUT) :: DES_PTS(:,:)
REAL(KIND=R8), ALLOCATABLE, OPTIONAL, INTENT(INOUT) :: OBJ_PTS(:,:)
INTEGER, OPTIONAL, INTENT(IN) :: PFLAG ! Parallel execution mode.
INTEGER, OPTIONAL, INTENT(IN) :: ICHKPT ! Checkpointing mode.
! Optional procedures.
PROCEDURE(VTMOP_MOD_LOCAL_INT), OPTIONAL :: LOCAL_OPT
PROCEDURE(VTMOP_MOD_SFIT_INT), OPTIONAL :: FIT_SURROGATES
PROCEDURE(VTMOP_MOD_SEVAL_INT), OPTIONAL :: EVAL_SURROGATES
! Local copies of optional parameters.
LOGICAL :: ADAPTIVE_SEARCHL ! Adaptive search option.
INTEGER :: BB_BUDGETL ! Budget limit.
INTEGER :: LOPT_BUDGETL ! Local optimizer budget.
INTEGER :: MAXITERSL ! Maximum number of iterations allowed.
INTEGER :: SEARCH_BUDGETL ! Search budget.
INTEGER :: INITIAL_SBUDGETL ! Initial search budget.
INTEGER :: PFLAGL ! Parallel execution mode.
INTEGER :: ICHKPTL ! Checkpointing mode.
REAL(KIND=R8) :: DECAYL ! Decay factor for the LTR radius.
REAL(KIND=R8) :: DES_TOL_L ! Design space tolerance.
REAL(KIND=R8) :: EPSL ! Working precision.
REAL(KIND=R8) :: EPSWL ! Fudge factor for zero weights.
REAL(KIND=R8) :: OBJ_TOL_L ! Objective space tolerance.
REAL(KIND=R8) :: MIN_RADFL ! Minimum LTR radius allowed.
REAL(KIND=R8) :: TRUST_RADFL ! Initial LTR radius.
! Pointers to local copies of procedures.
PROCEDURE(VTMOP_MOD_LOCAL_INT), POINTER :: LOCAL_OPTL
PROCEDURE(VTMOP_MOD_SFIT_INT), POINTER :: FIT_SURROGATESL
PROCEDURE(VTMOP_MOD_SEVAL_INT), POINTER :: EVAL_SURROGATESL
! Local variables.
TYPE(VTMOP_TYPE) :: VTMOP ! Multiobjective optimization problem metadata.
INTEGER :: I, J ! Loop indexing variables.
INTEGER :: M, N ! Array dimensions.
LOGICAL :: PEVALS ! Do function evaluations in parallel.
LOGICAL :: PMODE ! Parallel flag for VTMOP_INIT.
REAL(KIND=R8) :: LTR_UB(D), LTR_LB(D) ! LTR upper and lower bounds.
REAL(KIND=R8) :: F_VAL(P) ! Dummy variable for storing function value.
REAL(KIND=R8), ALLOCATABLE :: CAND_X(:,:) ! Potentially efficient design point.
! External BLAS function for computing Euclidean distance.
REAL(KIND=R8), EXTERNAL :: DNRM2

! *** Preprocess input arguments and initialize the module memory *** !

! Check for illegal input dimensions and values.
IF (D < 1) THEN ! Illegal design space dimension.
   IERR = 110; RETURN; END IF
IF (P < 2) THEN ! Illegal objective space dimension.
   IERR = 111; RETURN; END IF
IF (SIZE(LB,1) .NE. D) THEN ! Lower bounds dimension must match D.
   IERR = 112; RETURN; END IF
IF (SIZE(UB,1) .NE. D) THEN ! Upper bounds dimension must match D.
   IERR = 113; RETURN; END IF

! Check optional inputs.
! Check whether the search will be adaptive.
ADAPTIVE_SEARCHL = .TRUE.
IF(PRESENT(ADAPTIVE_SEARCH)) THEN
   ADAPTIVE_SEARCHL = ADAPTIVE_SEARCH; END IF
! Set the budget for the global search algorithm over each LTR.
IF(ADAPTIVE_SEARCHL) THEN
   SEARCH_BUDGETL = 5
ELSE
   SEARCH_BUDGETL = 8*D; END IF
IF(PRESENT(SEARCH_BUDGET)) THEN
   ! Search budget must be nonnegative.
   IF(SEARCH_BUDGET < 0) THEN
      IERR = 122; RETURN; END IF
   SEARCH_BUDGETL = SEARCH_BUDGET
END IF
! Set the budget for the initial global search, over the entire design space.
IF(ADAPTIVE_SEARCHL) THEN
   INITIAL_SBUDGETL = 10
ELSE
   INITIAL_SBUDGETL = 16 * D**2; END IF
IF(PRESENT(INITIAL_SBUDGET)) THEN
   ! Any nonnegative value is allowed, but if too few data points are
   ! available for the zeroth optimization phase (between the initial
   ! database and initial search) then the surrogate models will fail.
   IF(INITIAL_SBUDGET < 0) THEN
      IERR = 122; RETURN; END IF
   INITIAL_SBUDGETL = INITIAL_SBUDGET
END IF
! Set the budget for the total number of blackbox function evaluations.
BB_BUDGETL = 1000
IF(PRESENT(BB_BUDGET)) THEN
   ! The budget must be positive.
   IF(BB_BUDGET < 1) THEN
      IERR = 120; RETURN; END IF
   BB_BUDGETL = BB_BUDGET
END IF
! Set the budget on the maximum number of iterations. By default, this
! value is unlimited.
```

```
MAXITERSL = HUGE(0)
IF(PRESENT(MAXITERS)) THEN
  ! The iteration limit must allow for at least one iteration.
   IF(MAXITERS < 1) THEN
      IERR = 121; RETURN; END IF
   MAXITERSL = MAXITERS
END IF
! Set the budget on the number of iterations of the local optimization
! subroutine.
LOPT_BUDGETL = 2500
IF(PRESENT(LOPT_BUDGET)) THEN
   ! The budget must be at least 1, return an error for nonpositive values.
   IF(LOPT_BUDGET < 1) THEN
      IERR = 123; RETURN; END IF
   LOPT_BUDGETL = LOPT_BUDGET
END IF
! Initialize the working precision.
EPSL = SQRT(EPSILON(0.0_R8))
IF(PRESENT(EPS)) THEN
   ! The default value of EPS cannot be decreased. Simply ignore such inputs.
   IF(EPS > EPSL) EPSL = EPS
END IF
! Initialize the fudge factor for zero weights.
EPSWL = EPSILON(0.0_R8) ** 0.25_R8
IF(PRESENT(EPSW)) THEN
   ! Ignore EPSW if it is less than the square-root of the machine EPSILON.
   IF(EPSW .GE. SQRT(EPSILON(0.0_R8))) EPSWL = EPSW
END IF
! Initialize the design space and objective space tolerances.
DES_TOL_L = SQRT(EPSL)
IF (PRESENT(DES_TOL)) THEN
   IF(DES_TOL .GE. EPSL) THEN
      DES_TOL_L = DES_TOL
   ELSE
      DES_TOL_L = EPSL
   END IF
END IF
OBJ_TOL_L = SQRT(EPSL)
IF (PRESENT(OBJ_TOL)) THEN
   IF(OBJ_TOL .GE. EPSL) THEN
      OBJ_TOL_L = OBJ_TOL
   ELSE
      OBJ_TOL_L = EPSL
   END IF
END IF
! Initialize the decay rate.
DECAYL = 0.5_R8
IF(PRESENT(DECAY)) THEN
   ! The decay rate must be between 0 and 1, up to the working precision EPS.
   IF(DECAY > 1.0_R8 - EPSL .OR. DECAY < EPSL) THEN
      IERR = 124; RETURN; END IF
   DECAYL = DECAY
END IF
! Initialize the LTR radius fraction and minimum LTR radius fraction.
TRUST_RADFL = 0.2_R8
IF(PRESENT(TRUST_RADF)) THEN
   ! The LTR radius fraction must be greater than the design space tolerance.
   IF(TRUST_RADF .GE. DES_TOL_L .AND. TRUST_RADF < 1.0_R8) THEN
      TRUST_RADFL = TRUST_RADF; END IF
END IF
! The minimum fraction for the LTR radius must be greater than the design
! space tolerance. By default, tolerate decay down to 10% of the
! initial LTR radius or the design space tolearnce (whichever is larger).
MIN_RADFL = MAX(0.1_R8 * TRUST_RADFL, DES_TOL_L)
IF(PRESENT(MIN_RADF)) THEN
   IF(MIN_RADF .GE. DES_TOL_L) MIN_RADFL = MIN_RADF
END IF
! Check that the size of the LTR radius is appropriate.
IF(MIN_RADFL > TRUST_RADFL) THEN
   IERR = 125; RETURN; END IF
! Set the parallel execution mode.
PFLAGL = 0
PMODE = .FALSE.
PEVALS = .FALSE.
IF (PRESENT(PFLAG)) THEN
   PFLAGL = PFLAG; END IF
! Use the PFLAG codes to set PEVALS and PMODE.
IF (PFLAGL .EQ. 1 .OR. PFLAGL .EQ. 3) PEVALS = .TRUE.
IF (PFLAGL .EQ. 2 .OR. PFLAGL .EQ. 3) PMODE = .TRUE.

! Set the checkpoint mode.
ICHKPTL = 0
IF (PRESENT(ICHKPT)) THEN
   ICHKPTL = ICHKPT; END IF
! Set the procedure arguments.
LOCAL_OPTL => GPSMADS ! Default optimizer is GPSMADS.
IF(PRESENT(LOCAL_OPT)) THEN
   LOCAL_OPTL => LOCAL_OPT
END IF
FIT_SURROGATESL => LSHEP_FIT ! Default fit is LSHEP_FIT.
IF(PRESENT(FIT_SURROGATES)) THEN
   IF (.NOT. PRESENT(EVAL_SURROGATES)) THEN
```

```
      IERR = 128; RETURN; END IF
   FIT_SURROGATESL => FIT_SURROGATES
END IF
EVAL_SURROGATESL => LSHEP_EVAL ! Default evaluation is LSHEP_EVAL.
IF(PRESENT(EVAL_SURROGATES)) THEN
   IF (.NOT. PRESENT(FIT_SURROGATES)) THEN
      IERR = 128; RETURN; END IF
   EVAL_SURROGATESL => EVAL_SURROGATES
END IF
! Lower bounds must be elementwise strictly less than upper bounds.
IF (ANY(LB(:) .GE. UB(:) - DES_TOL_L)) THEN
   IERR = 114; RETURN; END IF

! Set the necessary module variables.
VTMOP_MOD_D = D; VTMOP_MOD_P = P
VTMOP_MOD_BB_BUDGET = BB_BUDGETL
IF (ICHKPTL .NE. 0) THEN
   VTMOP_MOD_CHKPT = .TRUE.
ELSE
   VTMOP_MOD_CHKPT = .FALSE.; END IF
VTMOP_MOD_DES_TOL = DES_TOL_L
VTMOP_MOD_OBJ_FUNC => OBJ_FUNC
! Free the module databases, if they are already allocated.
IF(ALLOCATED(VTMOP_MOD_DBX)) THEN
   DEALLOCATE(VTMOP_MOD_DBX, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 131; RETURN; END IF
END IF
IF(ALLOCATED(VTMOP_MOD_DBF)) THEN
   DEALLOCATE(VTMOP_MOD_DBF, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 131; RETURN; END IF
END IF
! If parallel function evaluations are active, initialize the OpenMP locks.
IF (PEVALS) THEN
   ! Reallocate the lock array.
   IF (ALLOCATED(VTMOP_MOD_DB_BUSY)) THEN
      DEALLOCATE(VTMOP_MOD_DB_BUSY, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF
   END IF
   ALLOCATE(VTMOP_MOD_DB_BUSY(BB_BUDGETL), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 130; RETURN; END IF
   ! Initialize the locks.
   CALL OMP_INIT_LOCK(VTMOP_MOD_DBLCK)
   DO I = 1, BB_BUDGETL
      CALL OMP_INIT_LOCK(VTMOP_MOD_DB_BUSY(I))
   END DO
END IF

! *** Execute the VTMOP algorithm, as described by Deshpande et al. *** !

! Begin the initialization phase. A different initialization is required when
! recovery mode is activated.

! If recovery mode is activated, read from file then complete the current
! iteration.
IF( ICHKPTL < 0) THEN
   ! If DES_PTS or OBJ_PTS were supplied, check that both are present.
   IF (PRESENT(DES_PTS) .OR. PRESENT(OBJ_PTS)) THEN
      ! The presence of DES_PTS without OBJ_PTS, or vice versa, is an error.
      IF (.NOT. (PRESENT(DES_PTS) .AND. PRESENT(OBJ_PTS))) THEN
         IERR = 126; RETURN; END IF
   END IF

   ! Recover the VTMOP object.
   CALL VTMOP_INIT( VTMOP, D, P, LB, UB, IERR, LOPT_BUDGET=LOPT_BUDGETL, &
                  LOCAL_OPT=LOCAL_OPTL, FIT_SURROGATES=FIT_SURROGATESL,  &
                  EVAL_SURROGATES=EVAL_SURROGATESL, PMODE=PMODE,         &
                  ICHKPT=ICHKPTL )
   IF (IERR .NE. 0) RETURN

   ! Recover the VTMOP data.
   CALL VTMOP_RECOVER_DATA( VTMOP_MOD_DBN, VTMOP_MOD_DBX, VTMOP_MOD_DBF, IERR, &
                            DB_SIZE=BB_BUDGETL )
   IF (IERR .NE. 0) RETURN

   ! The status of VTMOP%WEIGHTS(:,:) determines whether or not to immediately
   ! execute a search/optimize phase.
   IF (ALLOCATED(VTMOP%WEIGHTS)) THEN

      IF (VTMOP%ITERATE .EQ. 0) THEN
         ! Build the initial LTR, it is the entire design space.
         LTR_LB(:) = VTMOP%LB(:)
         LTR_UB(:) = VTMOP%UB(:)
      ELSE
         ! Rebuild the Kth LTR. It is the intersection over the current
         ! box and the bound constraints.
         DO I = 1, D
            LTR_UB(I) = MIN( VTMOP%CLIST(I,VTMOP%ITERATE) +   &
                             VTMOP%CLIST(D+1,VTMOP%ITERATE) * &
```

```
                     (VTMOP%UB(I) - VTMOP%LB(I)),     &
                     VTMOP%UB(I) )
      LTR_LB(I) = MAX( VTMOP%CLIST(I,VTMOP%ITERATE) -   &
                     VTMOP%CLIST(D+1,VTMOP%ITERATE) * &
                     (VTMOP%UB(I) - VTMOP%LB(I)),     &
                     VTMOP%LB(I) )
   END DO
END IF

! Perform the search phase.
IF (VTMOP%ITERATE .EQ. 0) THEN
   ! Special search rules for the 0th iteration.
   IF (ADAPTIVE_SEARCHL) THEN
      ! Search the entire design space using VTdirect95.
      CALL VTDIRECT_SEARCH( D, P, LTR_LB, LTR_UB, INITIAL_SBUDGETL,  &
                            .TRUE., IERR, EPSW=EPSWL, TOL=DES_TOL_L, &
                            PARALLEL=PEVALS )
      IF (IERR .NE. 0) THEN
         IERR = IERR + 800; RETURN; END IF
   ELSE
      ! Generate an experimental design for the entire space.
      CALL LH_DESIGN( D, LTR_LB, LTR_UB, INITIAL_SBUDGETL, &
                      CAND_X, IERR, TOL=DES_TOL_L )
      IF (IERR .NE. 0) THEN
         IERR = IERR + 800; RETURN; END IF
      IF (PEVALS) THEN
         ! Do parallel threadsafe evaluations of candidate design points.
         !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
         !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
         DO I = 1, SIZE(CAND_X, 2)
            CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
         !$OMP END PARALLEL DO
      ELSE
         ! Evaluate all candidate design points.
         DO I = 1, SIZE(CAND_X, 2)
            ! If IERR and F_VAL are dummy arguments here.
            CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
      END IF
      ! Free the candidate point set.
      DEALLOCATE(CAND_X, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF
   END IF
ELSE
   ! Search the Kth LTR, using the standard search budget.
   IF (ADAPTIVE_SEARCHL) THEN
      ! Search the Kth LTR using VTdirect95.
      CALL VTDIRECT_SEARCH( D, P, LTR_LB, LTR_UB, SEARCH_BUDGETL,     &
                            .FALSE., IERR, EPSW=EPSWL, TOL=DES_TOL_L, &
                            PARALLEL=PEVALS )
      IF (IERR .NE. 0) THEN
         IERR = IERR + 800; RETURN; END IF
   ELSE
      ! Generate an experimental design for the Kth LTR.
      CALL LH_DESIGN( D, LTR_LB, LTR_UB, SEARCH_BUDGETL, &
                      CAND_X, IERR, TOL=DES_TOL_L,       &
                      XI=VTMOP%CLIST(1:D,VTMOP%ITERATE) )
      IF (IERR .NE. 0) THEN
         IERR = IERR + 800; RETURN; END IF
      IF (PEVALS) THEN
         ! Do parallel threadsafe evaluations of candidate design points.
         !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
         !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
         DO I = 1, SIZE(CAND_X, 2)
            CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
         !$OMP END PARALLEL DO
      ELSE
         ! Evaluate all candidate design points.
         DO I = 1, SIZE(CAND_X, 2)
            ! The outputs of IERR and F_VAL are not used.
            CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
      END IF
      ! Free the candidate point set.
      DEALLOCATE(CAND_X, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF
   END IF
END IF

! Run the local optimizer over the surrogates in the Kth LTR.
CALL VTMOP_OPT( VTMOP, LTR_LB, LTR_UB, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN), &
                VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), CAND_X, IERR )
IF (IERR .NE. 0) RETURN

IF (PEVALS) THEN
   ! Do parallel threadsafe evaluations of candidate design points.
   !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
```

```
         !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
         DO I = 1, SIZE(CAND_X, 2)
            ! The outputs of IERR and F_VAL are not used.
            CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
         !$OMP END PARALLEL DO
      ELSE
         ! Evaluate all candidate design points.
         DO I = 1, SIZE(CAND_X, 2)
            ! The outputs of IERR and F_VAL are not used.
            CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
         END DO
      END IF
      ! Free the candidate point set.
      DEALLOCATE(CAND_X, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF

   END IF

! Otherwise, recovery mode is inactive. Perform a normal initialization.
ELSE

   ! If checkpointing is activated, initialize a new unformatted data file.
   IF (ICHKPTL > 0) THEN
      CALL VTMOP_NEW_DATA(D, P, IERR)
      IF (IERR .NE. 0) RETURN
   END IF

   ! Initialize the VTMOP object.
   CALL VTMOP_INIT( VTMOP, D, P, LB, UB, IERR,      &
                    LOPT_BUDGET=LOPT_BUDGETL,       &
                    DECAY=DECAYL,                   &
                    DES_TOL=DES_TOL_L,              &
                    EPS=EPSL, EPSW=EPSWL,           &
                    OBJ_TOL=OBJ_TOL_L,              &
                    MIN_RADF=MIN_RADFL,             &
                    TRUST_RADF=TRUST_RADFL,         &
                    LOCAL_OPT=LOCAL_OPTL,           &
                    FIT_SURROGATES=FIT_SURROGATESL, &
                    EVAL_SURROGATES=EVAL_SURROGATESL, &
                    PMODE=PMODE, ICHKPT=ICHKPTL )
   IF (IERR .NE. 0) RETURN

   ! Reallocate the module databases to the size of the budget.
   ALLOCATE( VTMOP_MOD_DBX(D,BB_BUDGETL), VTMOP_MOD_DBF(P,BB_BUDGETL), &
             STAT=IERR )
   IF (IERR .NE. 0) THEN
      IERR = 130; RETURN; END IF
   VTMOP_MOD_DBN = 0

   ! If DES_PTS and OBJ_PTS were supplied, fill in the initial database.
   IF (PRESENT(DES_PTS) .OR. PRESENT(OBJ_PTS)) THEN
      ! The presence of DES_PTS without OBJ_PTS, or vice versa, is an error.
      IF (.NOT. (PRESENT(DES_PTS) .AND. PRESENT(OBJ_PTS))) THEN
         IERR = 126; RETURN; END IF
      ! If DES_PTS or OBJ_PTS are allocated, then they are assumed to
      ! contain an initial input database.
      IF (ALLOCATED(DES_PTS) .OR. ALLOCATED(OBJ_PTS)) THEN
         ! The allocation status of DES_PTS and OBJ_PTS must agree.
         IF (.NOT. (ALLOCATED(DES_PTS) .AND. ALLOCATED(OBJ_PTS))) THEN
            IERR = 127; RETURN; END IF
         ! The dimensions of DES_PTS and OBJ_PTS must match.
         N = SIZE(DES_PTS,2)
         IF (SIZE(DES_PTS,1) .NE. D .OR. SIZE(OBJ_PTS,1) .NE. P .OR. &
             SIZE(OBJ_PTS,2) .NE. N) THEN
            IERR = 127; RETURN; END IF
         ! Loop over DES_PTS/OBJ_PTS and populate the initial database.
         DO I = 1, N
            ! Ignore design points that are too close.
            DO J = VTMOP_MOD_DBN, 1, -1
               IF (DNRM2(D, VTMOP_MOD_DBX(:,J)-DES_PTS(:,I), 1) < DES_TOL_L) &
                  EXIT
            END DO
            ! If no duplicates were found, then proceed to insert.
            IF (J .EQ. 0) THEN
               ! Update the design and objective datasets.
               VTMOP_MOD_DBN = VTMOP_MOD_DBN + 1
               VTMOP_MOD_DBX(:,VTMOP_MOD_DBN) = DES_PTS(:,I)
               VTMOP_MOD_DBF(:,VTMOP_MOD_DBN) = OBJ_PTS(:,I)
               ! If checkpointing is active, then save to the data file.
               IF (ICHKPTL > 0) THEN
                  CALL VTMOP_SAVE_DATA(DES_PTS(:,I), OBJ_PTS(:,I), IERR)
                  IF (IERR .NE. 0) RETURN
               END IF
            END IF
         END DO
      END IF
   END IF
END IF

! End of the initialization phase. Enter the main iteration loop.
```

```
! Loop until an exit condition is triggered.
DO WHILE (VTMOP%ITERATE .LE. MAXITERSL)

   ! Check whether the function evaluation budget is exhausted.
   IF (VTMOP_MOD_DBN .GE. BB_BUDGETL) EXIT

   ! Generate the Kth LTR.
   CALL VTMOP_LTR( VTMOP, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN),        &
                   VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), LTR_LB, LTR_UB, &
                   IERR )
   IF (IERR .EQ. 3) THEN ! Acceptance loop failed to produce a new LTR.
      EXIT
   ELSE IF (IERR .NE. 0) THEN ! Other error codes.
      RETURN
   END IF

   ! Perform the search phase.
   IF (VTMOP%ITERATE .EQ. 0) THEN
      ! Special search rules for the 0th iteration.
      IF (ADAPTIVE_SEARCHL) THEN
         ! Search the entire design space using VTdirect95.
         CALL VTDIRECT_SEARCH( D, P, LTR_LB, LTR_UB, INITIAL_SBUDGETL,  &
                               .TRUE., IERR, EPSW=EPSWL, TOL=DES_TOL_L, &
                               PARALLEL=PEVALS )
         IF (IERR .NE. 0) THEN
            IERR = IERR + 800; RETURN; END IF
      ELSE
         ! Generate an experimental design for the entire space.
         CALL LH_DESIGN( D, LTR_LB, LTR_UB, INITIAL_SBUDGETL, &
                         CAND_X, IERR, TOL=DES_TOL_L )
         IF (IERR .NE. 0) THEN
            IERR = IERR + 800; RETURN; END IF
         IF (PEVALS) THEN
            ! Do parallel threadsafe evaluations of candidate design points.
            !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
            !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
            DO I = 1, SIZE(CAND_X, 2)
               ! The outputs of IERR and F_VAL are not used.
               CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
            END DO
            !$OMP END PARALLEL DO
         ELSE
            ! Evaluate all candidate design points.
            DO I = 1, SIZE(CAND_X, 2)
               ! The outputs of IERR and F_VAL are not used.
               CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
            END DO
         END IF
         ! Free the candidate point set.
         DEALLOCATE(CAND_X, STAT=IERR)
         IF (IERR .NE. 0) THEN
            IERR = 131; RETURN; END IF
      END IF

   ELSE
      ! Search the Kth LTR, using the standard search budget.
      IF (ADAPTIVE_SEARCHL) THEN
         ! Search the Kth LTR using VTdirect95.
         CALL VTDIRECT_SEARCH( D, P, LTR_LB, LTR_UB, SEARCH_BUDGETL,     &
                               .FALSE., IERR, EPSW=EPSWL, TOL=DES_TOL_L, &
                               PARALLEL=PEVALS )
         IF (IERR .NE. 0) THEN
            IERR = IERR + 800; RETURN; END IF
      ELSE
         ! Generate an experimental design for the Kth LTR.
         CALL LH_DESIGN( D, LTR_LB, LTR_UB, SEARCH_BUDGETL, &
                         CAND_X, IERR, TOL=DES_TOL_L,       &
                         XI=VTMOP%CLIST(1:D,VTMOP%ITERATE) )
         IF (IERR .NE. 0) THEN
            IERR = IERR + 800; RETURN; END IF
         IF (PEVALS) THEN
            ! Do parallel threadsafe evaluations of candidate design points.
            !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
            !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
            DO I = 1, SIZE(CAND_X, 2)
               ! The outputs of IERR and F_VAL are not used.
               CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
            END DO
            !$OMP END PARALLEL DO
         ELSE
            ! Evaluate all candidate design points.
            DO I = 1, SIZE(CAND_X, 2)
               ! The outputs of IERR and F_VAL are not used.
               CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
            END DO
         END IF
         ! Free the candidate point set.
         DEALLOCATE(CAND_X, STAT=IERR)
         IF (IERR .NE. 0) THEN
            IERR = 131; RETURN; END IF
      END IF
```

```
   END IF

   ! Check whether the function evaluation budget is exhausted.
   IF (VTMOP_MOD_DBN .GE. BB_BUDGETL) EXIT

   ! Run the local optimizer over surrogates in the Kth LTR.
   CALL VTMOP_OPT( VTMOP, LTR_LB, LTR_UB, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN), &
                   VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), CAND_X, IERR )
   IF (IERR .NE. 0) RETURN

   IF (PEVALS) THEN
      ! Do parallel threadsafe evaluations of candidate design points.
      !$OMP PARALLEL DO SCHEDULE(DYNAMIC), &
      !$OMP& DEFAULT(SHARED), PRIVATE(I, F_VAL, IERR)
      DO I = 1, SIZE(CAND_X, 2)
         ! The outputs of IERR and F_VAL are not used.
         CALL MOP_P_EVALUATE(CAND_X(:,I), F_VAL, IERR)
      END DO
      !$OMP END PARALLEL DO
   ELSE
      ! Evaluate all candidate design points.
      DO I = 1, SIZE(CAND_X, 2)
         ! The outputs of IERR and F_VAL are not used.
         CALL MOP_EVALUATE(CAND_X(:,I), F_VAL, IERR)
      END DO
   END IF
   ! Free the candidate point set.
   DEALLOCATE(CAND_X, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 131; RETURN; END IF

END DO

! If DES_PTS and OBJ_PTS are present, then copy out the module database.
IF (PRESENT(DES_PTS) .AND. PRESENT(OBJ_PTS)) THEN
   ! Deallocate DES_PTS and OBJ_PTS.
   IF (ALLOCATED(DES_PTS)) THEN
      DEALLOCATE(DES_PTS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF
   END IF
   IF (ALLOCATED(OBJ_PTS)) THEN
      DEALLOCATE(OBJ_PTS, STAT=IERR)
      IF (IERR .NE. 0) THEN
         IERR = 131; RETURN; END IF
   END IF
   ! Reallocate DES_PTS and OBJ_PTS.
   ALLOCATE(DES_PTS(D, VTMOP_MOD_DBN), OBJ_PTS(P, VTMOP_MOD_DBN), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 130; RETURN; END IF
   ! Copy out the data.
   DES_PTS(:,:) = VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN)
   OBJ_PTS(:,:) = VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN)
END IF

! Finalize the results, and check stopping condition.
IF (VTMOP_MOD_DBN .GE. BB_BUDGETL) THEN ! Stopping condition 1.
   CALL VTMOP_FINALIZE( VTMOP, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN), &
                        VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), M,       &
                        PARETO_F, EFFICIENT_X, IERR )
   IF (IERR .NE. 0) RETURN
   IERR = 1
ELSE IF (VTMOP%ITERATE .GE. MAXITERSL) THEN ! Stopping condition 2.
   CALL VTMOP_FINALIZE( VTMOP, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN), &
                        VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), M,       &
                        PARETO_F, EFFICIENT_X, IERR )
   IF (IERR .NE. 0) RETURN
   IERR = 2
ELSE ! Stopping condition 3.
   CALL VTMOP_FINALIZE( VTMOP, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN), &
                        VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), M,       &
                        PARETO_F, EFFICIENT_X, IERR )
   IF (IERR .NE. 0) RETURN
   IERR = 3
END IF

! Free the module databases.
DEALLOCATE(VTMOP_MOD_DBX, VTMOP_MOD_DBF)
! If parallel function evaluations are active, destroy the OpenMP locks.
IF (PEVALS) THEN
   ! Initialize the locks.
   CALL OMP_DESTROY_LOCK(VTMOP_MOD_DBLCK)
   DO I = 1, BB_BUDGETL
      CALL OMP_DESTROY_LOCK(VTMOP_MOD_DB_BUSY(I))
   END DO
   ! Deallocate the lock array.
   DEALLOCATE(VTMOP_MOD_DB_BUSY, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 131; RETURN; END IF
END IF
RETURN
END SUBROUTINE VTMOP_SOLVE
```

```
! The subroutine DELAUNAYGRAPH is an external procedure, as it may be of
! interest independently.

SUBROUTINE DELAUNAYGRAPH(D, N, PTS, GRAPH, IERR, EPS, IBUDGET, PMODE)
! This subroutine produces the Delaunay graph for a set of N points in R^D
! in polynomial time using DELAUNAYSPARSE. If DELAUNAYSPARSE reports that PTS
! is embedded in a lower-dimensional linear manifold (error code 31), then
! PTS is projected onto the span of its left singular vectors with nonzero
! singular values (i.e., principle component analysis).
!
!
! On input:
!
! D is the dimension of the space for PTS.
!
! N is the number of data points in PTS.
!
! PTS(1:D,1:N) is a real matrix with N columns, each containing the
!    coordinates of a single data point in R^D.
!
!
! On output:
!
! PTS has been rescaled and shifted. All the data points in PTS are now
!    contained in the unit hyperball in R^D.
!
! GRAPH(1:N,1:N) is a symmetric matrix of type LOGICAL containing the
!    Delaunay graph structure. If GRAPH(I,J) is .TRUE., then the vertices
!    PTS(:,I) and PTS(:,J) are Delaunay neighbors in some Delaunay
!    triangulation. Otherwise, if GRAPH(I,J) is .FALSE., then PTS(:,I) and
!    PTS(:,J) are not neighbors in some Delaunay triangulation.
!
! IERR is an integer error flag. For the most part, IERR relays
!    error codes from DELAUNAYSPARSE{S|P}.
!    The error codes are:
!
! 00 : Successfully interpolated all midpoints and constructed the Delaunay
!      graph structure.
! 01 : Too few points were provided to compute a complete triangulation.
!      The Delaunay graph was still computed, under the assumption that
!      all points are Delaunay neighbors.
! 02 : The input set PTS is embedded in a lower dimensional linear manifold.
!      The Delaunay graph was still computed for a dimension reduced copy of
!      PTS.
!
! 10 : The dimension D must be positive.
! 12 : The supplied LOGICAL matrix GRAPH(:,:) is not of dimension N x N.
! 13 : The first dimension of PTS does not agree with the dimension D.
! 14 : The second dimension of PTS does not agree with the number of points N.
!
! 26 : The budget supplied in IBUDGET does not contain a positive
!      integer.
!
! 30 : Two or more points in the data set PTS are too close together with
!      respect to the working precision (EPS), which would result in a
!      numerically degenerate simplex.
!
! 40 : An error caused DELAUNAYSPARSE to terminate before the entire
!      Delaunay graph could be computed.
!
! 50 : A memory allocation error occurred while allocating the internal
!      work array for DELAUNAYSPARSE.
! 51 : A memory allocation error occurred while allocating the work arrays
!      for DGESVD, for performing dimension reduction.
! 52 : A memory deallocation error occurred while freeing work arrays for
!      DGESVD.
!
! 60 : The budget was exceeded before the algorithm converged for at least
!      one of the simplices. If the dimension is high, try increasing IBUDGET.
!      This error can also be caused by a working precision EPS that is too
!      small for the conditioning of the problem.
!
! 61 : A value that was judged appropriate later caused LAPACK to encounter a
!      singularity. Try increasing the value of EPS.
!
! 70 : Allocation error for the extrapolation work arrays.
! 71 : The SLATEC subroutine DWNNLS failed to converge during the projection
!      of an extrapolation point onto the convex hull.
! 72 : The SLATEC subroutine DWNNLS has reported a usage error.
! 73 : At least one of the midpoints was ruled significantly outside the
!      convex hull of PTS by DELAUNAYSPARSE. This is a numerical precision
!      issue that should never occur. Consider increasing EPS.
!
!      The errors 72, 80--83, and 90 should never occur, and likely indicate
!      a compiler bug or hardware failure.
! 80 : The LAPACK subroutine DGEQP3 has reported an illegal value.
! 81 : The LAPACK subroutine DGETRF has reported an illegal value.
! 82 : The LAPACK subroutine DGETRS has reported an illegal value.
! 83 : The LAPACK subroutine DORMQR has reported an illegal value.
!
! 90 : The LAPACK subroutine DGESVD has reported an illegal value.
```

```
! 91 : The LAPACK subroutine DGESVD has failed to converge.
!
!
! Optional arguments:
!
! EPS contains the working precision for the problem on input. By default,
!    EPS is assigned \sqrt{\mu} where \mu denotes the unit roundoff for the
!    machine. In general, any values that differ by less than EPS are judged
!    as equal, and any weights that are greater than -EPS are judged as
!    nonnegative.  EPS cannot take a value less than the default value of
!    \sqrt{\mu}. If any value less than \sqrt{\mu} is supplied, the default
!    value will be used instead automatically.
!
! IBUDGET contains the integer budget for performing flips while
!    iterating toward the simplex containing each interpolation point in Q.
!    This prevents DelaunayFan from falling into an infinite loop when
!    supplied with degenerate or near degenerate data.  By default,
!    IBUDGET=50000. However, for extremely high-dimensional problems and
!    pathological data sets, the default value may be insufficient.
!
! PMODE is a logical value specifying whether to compute the Delaunay graph
!    serially (using DELAUNAYSPARSES) or in parallel (using DELAUNAYSPARSEP).
!
USE DELSPARSE_MOD
IMPLICIT NONE

! Input arguments.
INTEGER, INTENT(IN) :: D, N
REAL(KIND=R8), INTENT(INOUT) :: PTS(:,:)
! Output arguments.
LOGICAL, INTENT(OUT) :: GRAPH(:,:)
INTEGER, INTENT(OUT) :: IERR
! Optional arguments.
REAL(KIND=R8), OPTIONAL, INTENT(IN) :: EPS
INTEGER, OPTIONAL, INTENT(IN) :: IBUDGET
LOGICAL, OPTIONAL, INTENT(IN) :: PMODE

! Local variables.
INTEGER :: IBUDGETL ! Local copy of IBUDGET.
INTEGER :: I, J, K ! Loop iteration variables.
INTEGER :: SIMPS(D+1,N*(N-1)/2) ! Matrix of simplices.
INTEGER :: IERR_LIST(N*(N-1)/2) ! Array of error codes for DELAUNAYSPARSE.
LOGICAL :: PMODEL ! Local copy of PMODE.
REAL(KIND=R8) :: EPSL ! Local copy of EPS.
REAL(KIND=R8) :: WEIGHTS(D+1,N*(N-1)/2) ! Matrix of interpolation weights.
REAL(KIND=R8) :: Q(D,N*(N-1)/2) ! Matrix of interpolation points.
! Work arrays for DGESVD, only referenced if dimension reduction is required.
INTEGER :: LWORK ! Length of the work array.
REAL(KIND=R8), ALLOCATABLE :: LSV(:,:) ! Left singular vectors.
REAL(KIND=R8), ALLOCATABLE :: RED_PTS(:,:) ! Dimension reduced PTS.
REAL(KIND=R8), ALLOCATABLE :: RED_Q(:,:) ! Dimension reduced Q.
REAL(KIND=R8), ALLOCATABLE :: S(:) ! Singular values.
REAL(KIND=R8), ALLOCATABLE :: WORK(:) ! Work array.
REAL(KIND=R8) :: U(1), VT(1) ! Optional outputs, not referenced by DGESVD.

! Check whether GRAPH is a legal size.
IF (SIZE(GRAPH, 1) .NE. N .OR. SIZE(GRAPH, 2) .NE. N) THEN
   IERR = 12
   RETURN
END IF
! Compute the machine precision.
EPSL = SQRT(EPSILON(1.0_R8))
! Check for the optional value EPS, and ensure that EPS is large enough.
IF (PRESENT(EPS)) THEN
   IF(EPSL < EPS) EPSL = EPS
END IF
! Set the budget.
IBUDGETL = 50000
! Check for the optional input IBUDGET, and ensure that it is valid.
IF (PRESENT(IBUDGET)) THEN
   IF (IBUDGET < 1) THEN; IERR = 26; RETURN; END IF
   IBUDGETL = IBUDGET
END IF
! Get optional input PMODE.
PMODEL = .FALSE.
IF (PRESENT(PMODE)) PMODEL = PMODE

! Initialize the interpolation points.
K = 0
DO I = 1, N
   DO J = I+1, N
      ! Interpolate the midpoint between PTS(:,I) and PTS(:,J).
      Q(:,K+J-I) = (PTS(:,I) + PTS(:,J)) / 2.0_R8
   END DO
   ! Track the offset K.
   K = K + N - I
END DO

! Check whether iteration parallelism is turned on.
IF(PMODEL) THEN
   ! Compute the Delaunay simplices containing the midpoints in parallel.
```

```
   CALL DELAUNAYSPARSEP( D, N, PTS, N*(N-1)/2, Q, SIMPS, WEIGHTS, IERR_LIST, &
                         EPS=EPSL, IBUDGET=IBUDGETL, PMODE=1 )
ELSE
   ! Compute the Delaunay simplices containing the midpoints serially.
   CALL DELAUNAYSPARSES( D, N, PTS, N*(N-1)/2, Q, SIMPS, WEIGHTS, IERR_LIST, &
                         EPS=EPSL, IBUDGET=IBUDGETL )
END IF

! Check for errors.
IERR = 0
DO I = 1, N*(N-1)/2
   IF(IERR_LIST(I) > 2) THEN
      IERR = IERR_LIST(I) ! Pass the error code on.
   ELSE IF(IERR_LIST(I) .EQ. 2) THEN
      IERR = 73; END IF ! Error code 2 -> 73.
END DO

! Handle relevant error codes for DELAUNAYGRAPH.
IF (IERR .EQ. 11) THEN
   ! N is too small to triangulate PTS, so all PTS are considered neighbors.
   GRAPH(:,:) = .TRUE.
   FORALL ( I = 1 : N ) GRAPH(I,I) = .FALSE.
   IERR = 1
ELSE IF (IERR .EQ. 31) THEN
   ! PTS lie in a lower dimensional linear manifold. Reduce the dimension
   ! and retry the triangulation.

   ! Allocate the left singular vectors and singular values.
   ALLOCATE(LSV(D,N), S(D), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 51; RETURN; END IF
   ! Query the optimal size for WORK and store in LWORK.
   LWORK = -1
   CALL DGESVD('O', 'N', D, N, LSV, D, S, VT, 1, VT, 1, U, LWORK, IERR)
   IF (IERR < 0) THEN
      IERR = 90; RETURN
   ELSE IF (IERR > 0) THEN
      IERR = 91; RETURN; END IF
   LWORK = INT(U(1))
   ! Allocate the input/output matrix.
   ALLOCATE(WORK(LWORK), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 51; RETURN; END IF
   ! Copy the input array.
   LSV(:,:) = PTS(:,:)
   ! Compute the SVD and overwrite LSV with the left singular vectors.
   ! The dummy arguments U and VT are not referenced for these settings.
   CALL DGESVD('O', 'N', D, N, LSV, D, S, U, 1, VT, 1, WORK, LWORK, IERR)
   IF (IERR < 0) THEN
      IERR = 90; RETURN
   ELSE IF (IERR > 0) THEN
      IERR = 91; RETURN; END IF
   ! Perform the dimension reduction by eliminating directions with small
   ! singular values.
   DO I = 1, D; IF (S(I) < EPSL) EXIT; END DO
   I = I - 1
   ! Free the memory that was used for the SVD.
   DEALLOCATE(S, WORK, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 52; RETURN; END IF
   ! Allocate and initialize the reduced data and interpolation point sets.
   ALLOCATE(RED_PTS(I,N), RED_Q(I,N*(N-1)/2), STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 51; RETURN; END IF
   RED_PTS(:,:) = 0.0_R8
   RED_Q(:,:) = 0.0_R8
   ! Compute the projection RED_PTS(1:I,1:N) = LSV(1:D,1:I)^T PTS(1:D,1:N).
   CALL DGEMM('T', 'N', I, N, D, 1.0_R8, LSV, D, PTS, D, 0.0_R8, RED_PTS, I)
   ! Compute the projection RED_Q(1:I,1:N) = LSV(1:D,1:I)^T Q(1:D,:)
   CALL DGEMM('T', 'N', I, N*(N-1)/2, D, 1.0_R8, LSV, D, Q, D, 0.0_R8, RED_Q, I)
   ! Check whether iteration tasks are parallelized.
   IF (PMODEL) THEN
      ! Compute the reduced dimensional solution, using DELAUNAYSPARSEP.
      CALL DELAUNAYSPARSEP( I, N, RED_PTS, N*(N-1)/2, RED_Q, SIMPS(1:I+1,:), &
                            WEIGHTS(1:I+1,:), IERR_LIST, EPS=EPSL,           &
                            IBUDGET=IBUDGETL, EXACT=.FALSE., PMODE=1 )
   ELSE
      ! Compute the reduced dimensional solution, using DELAUNAYSPARSES.
      CALL DELAUNAYSPARSES( I, N, RED_PTS, N*(N-1)/2, RED_Q, SIMPS(1:I+1,:), &
                            WEIGHTS(1:I+1,:), IERR_LIST, EPS=EPSL,           &
                            IBUDGET=IBUDGETL, EXACT=.FALSE. )
   END IF
   SIMPS(I+2:D+1,:) = 0
   ! Check for errors.
   IERR = 2
   DO I = 1, N
      IF(IERR_LIST(I) > 2) THEN
         IERR = IERR_LIST(I) ! Pass the error code on.
      ELSE IF(IERR_LIST(I) .EQ. 2) THEN
         IERR = 73; END IF ! Error code 2 -> 73.
   END DO
```

```
   ! Free the temporary data matrices used for dimension reduction.
   DEALLOCATE(RED_PTS, RED_Q, LSV, STAT=IERR)
   IF (IERR .NE. 0) THEN
      IERR = 52; RETURN; END IF
END IF

! If an error code persists, quit.
IF (IERR .GE. 10) RETURN

! Initialize the Delaunay graph.
GRAPH(:,:) = .FALSE.
! Construct the Delaunay graph from the resulting simplices.
K = 0
DO I = 1, N
   DO J = I+1, N
      ! Check SIMPS(:,K+J-I) for any edge between PTS(:,I) and PTS(:,J).
      IF (ANY(SIMPS(:,K+J-I) .EQ. I) .AND. ANY(SIMPS(:,K+J-I) .EQ. J)) THEN
         GRAPH(I,J) = .TRUE.; GRAPH(J,I) = .TRUE.;  END IF

      ! If SIMPS(:,K+J-I) does not contain the edge ( PTS(:,I), PTS(:,J) ),
      ! then SIMPS(:,K+J-I) serves as a certificate of separation in some
      ! Delaunay triangulation of PTS.
   END DO
   ! Track the offset K.
   K = K + N - I
END DO

RETURN
END SUBROUTINE DELAUNAYGRAPH
```

C.2 bVTdirect.f90

```
! This file (bVTdirect.f90) contains the module bVTdirect_MOD that
! declares subroutines and functions used by the batched (parallel)
! implementation of VTDIRECT, which is used by VTMOP. Several modifications
! were made to the interface for the serial VTdirect code, allowing
! VTMOP to pass a P-dimensional scalarizing weight vector in a way
! that is threadsafe.
!
MODULE bVTdirect_MOD
USE VTDIRECT_COMMSUB ! Module (shared_modules.f95) for subroutines
  ! commonly used by VTdirect and pVTdirect.
CONTAINS

SUBROUTINE bVTdirect(D, P, L, U, WEIGHTS, OBJ_FUNC, X, FMIN, STATUS, SWITCH, &
                     MAX_ITER, MAX_EVL, MIN_DIA, OBJ_CONV, EPS, &
                     MIN_SEP, W, BOX_SET, NUM_BOX )
IMPLICIT NONE
! This is an OpenMP parallel implementation of the DIRECT global unconstrained
! optimization algorithm described in:
!
!    D.R. Jones, C.D. Perttunen, and B.E. Stuckman, Lipschitzian
!    optimization without the Lipschitz constant, Journal of Optimization
!    Theory and Applications, Vol. 79, No. 1, 1993, pp. 157-181.
!
! The algorithm to minimize f(x) inside the box L <= x <= U is as follows:
!
!    1. Normalize the search space to be the unit hypercube. Let c_1 be
!       the center point of this hypercube and evaluate f(c_1).
!    2. Identify the set S of potentially optimal rectangles.
!    3. For all rectangles j in S:
!       3a. Identify the set I of dimensions with the maximum side length.
!           Let delta equal one-third of this maximum side length.
!       3b. Sample the function at the points c +- delta*e_i for all i
!           in I, where c is the center of the rectangle and e_i is the
!           ith unit vector.
!       3c. Divide the rectangle containing c into thirds along the
!           dimensions in I, starting with the dimension with the lowest
!           value of f(c +- delta*e_i) and continuing to the dimension
!           with the highest f(c +- delta*e_i).
!    4. Repeat 2.-3. until stopping criterion is met.
!
! On input:
!
```

```
! D is the dimension of L, U, and X.
!
! P is the dimension of WEIGHTS.
!
! L(1:D) is a real array giving lower bounds on X.
!
! U(1:D) is a real array giving upper bounds on X.
!
! WEIGHTS(1:P) is a scalarizing weight vector that is passed to OBJ_FUNC.
!
! OBJ_FUNC is the name of the real function procedure defining the
!    objective function w^T f(x) to be minimized. OBJ_FUNC(C, SCAL_W, IFLAG)
!    returns the value SCAL_W^T f(C) with IFLAG=0, or IFLAG/=0 if f(C) is
!    not defined. OBJ_FUNC is precisely defined in the INTERFACE block below.
!
! Optional arguments:
!
! SWITCH =
!    1   select potentially optimal boxes on the convex hull of the
!        (box diameter, function value) points (default).
!    0   select as potentially optimal the box with the smallest function
!        value for each diameter that is above the roundoff level.
!        This is an aggressive selection procedure that generates many more
!        boxes to subdivide.
!
! MAX_ITER is the maximum number of iterations (repetitions of Steps 2-3)
!    allowed; defines stopping rule 1. If MAX_ITER is present but <= 0
!    on input, there is no iteration limit and the number of iterations
!    executed is returned in MAX_ITER.
!
! MAX_EVL is the maximum number of function evaluations allowed; defines
!    stopping rule 2. If MAX_EVL is present but <= 0 on input, there is no
!    limit on the number of function evaluations, which is returned in
!    MAX_EVL.
!
! MIN_DIA is the minimum box diameter allowed; defines stopping rule 3.
!    If MIN_DIA is present but <= 0 on input, a minimum diameter below
!    the roundoff level is not permitted, and the box diameter of the
!    box containing the smallest function value FMIN is returned in
!    MIN_DIA.
!
! OBJ_CONV is the smallest acceptable relative improvement in the minimum
!    objective function value 'FMIN' between iterations; defines
!    stopping rule 4. OBJ_CONV must be positive and greater than the round
!    off level.  If absent, it is taken as zero.
!
! EPS is the tolerance defining the minimum acceptable potential
!    improvement in a potentially optimal box.  Larger EPS values
!    eliminate more boxes from consideration as potentially optimal,
!    and bias the search toward exploration.  EPS must be positive and
!    greater than the roundoff level.  If absent, it is taken as
!    zero.  EPS > 0 is incompatible with SWITCH = 0.
!
! MIN_SEP is the specified minimal (weighted) distance between the
!    center points of the boxes returned in the optional array BOX_SET.
!    If absent or invalid, MIN_SEP is taken as 1/2 the (weighted) diameter
!    of the box [L, U].
!
! W(1:D) is a positive real array.  The distance between two points X
!    and Y is defined as SQRT(SUM( (X-Y)*W*(X-Y) )).  If absent, W is
!    taken as all ones.
!
! BOX_SET is an empty array (TYPE HyperBox) allocated to hold the desired
!    number of boxes.
!
! On output:
!
! X(1:D) is a real vector containing the sampled box center with the
!    minimum objective function value FMIN.
!
! FMIN is the minimum function value.
!
! STATUS is a return status flag. The units decimal digit specifies
!    which stopping rule was satisfied on a successful return. The tens
!    decimal digit indicates a successful return, or an error condition
!    with the cause of the error condition reported in the units digit.
!
! Tens digit =
!  0 Normal return.
!    Units digit =
!     1   Stopping rule 1 (iteration limit) satisfied.
!     2   Stopping rule 2 (function evaluation limit) satisfied.
!     3   Stopping rule 3 (minimum diameter reached) satisfied. The
!         minimum diameter corresponds to the box for which X and
!         FMIN are returned.
!     4   Stopping rule 4 (relative change in 'FMIN') satisfied.
!  1 Input data error.
!    Units digit =
!     0   D < 2.
!     1   Assumed shape array L, U, W, or X does not have size D.
!     2   Some lower bound is >= the corresponding upper bound.
!     3   MIN_DIA, OBJ_CONV, or EPS is invalid or below the roundoff level.
```

```
!     4   None of MAX_EVL, MAX_ITER, MIN_DIA, and OBJ_CONV are specified;
!         there is no stopping rule.
!     5   Invalid SWITCH value.
!     6   SWITCH = 0 and EPS > 0 are incompatible.
!  2 Memory allocation error or failure.
!    Units digit =
!     0   BoxMatrix type allocation.
!     1   BoxLink or BoxLine type allocation.
!     2   int_vector or real_vector type allocation.
!     3   HyperBox type allocation.
!     4   BOX_SET is allocated with a wrong problem dimension.
!
!  For example,
!     03 indicates a normal return (tens digit = 0) with "stopping rule 3
!        satisfied" (units digit = 3), and
!     12 indicates an input error (tens digit = 1) when "some lower bound
!        is >= the corresponding upper bound" (units digit = 2).
!
! Optional arguments:
!
! MAX_ITER (if present) contains the number of iterations.
!
! MAX_EVL (if present) contains the number of function evaluations.
!
! MIN_DIA (if present) contains the diameter of the box associated with
!    X and FMIN.
!
! MIN_SEP (if present) is unchanged if it was a reasonable value on
!    input. Otherwise, it is reset to the default value.
!
! W (if present) is unchanged if it was positive on input. Any
!    non-positive component is reset to one.
!
! BOX_SET (if present) is an array of TYPE (HyperBox) containing the
!    best boxes with centers separated by at least MIN_SEP.
!    The number of returned boxes NUM_BOX <= SIZE(BOX_SET) is as
!    large as possible given the requested separation.
!
! NUM_BOX (if present) is the number of boxes returned in the array
!    BOX_SET(1:).
!
INTEGER, INTENT(IN):: D, P
REAL(KIND = R8), DIMENSION(:), INTENT(IN):: L
REAL(KIND = R8), DIMENSION(:), INTENT(IN):: U
REAL(KIND = R8), DIMENSION(:), INTENT(IN):: WEIGHTS
INTERFACE
  FUNCTION OBJ_FUNC(C, SCAL_W, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN):: C
    REAL(KIND = R8), DIMENSION(:), INTENT(IN):: SCAL_W ! Scalarizing weights.
    INTEGER, INTENT(OUT):: IFLAG
    REAL(KIND = R8):: F
  END FUNCTION OBJ_FUNC
END INTERFACE
REAL(KIND = R8), DIMENSION(:), INTENT(OUT):: X
REAL(KIND = R8), INTENT(OUT):: FMIN
INTEGER, INTENT(OUT):: STATUS
INTEGER, INTENT(IN), OPTIONAL:: SWITCH
INTEGER, INTENT(INOUT), OPTIONAL:: MAX_ITER
INTEGER, INTENT(INOUT), OPTIONAL:: MAX_EVL
REAL(KIND = R8), INTENT(INOUT), OPTIONAL:: MIN_DIA
REAL(KIND = R8), INTENT(IN), OPTIONAL:: OBJ_CONV
REAL(KIND = R8), INTENT(IN), OPTIONAL:: EPS
REAL(KIND = R8), INTENT(INOUT), OPTIONAL:: MIN_SEP
REAL(KIND = R8), DIMENSION(:), INTENT(INOUT), OPTIONAL:: W
TYPE(HyperBox), DIMENSION(:), INTENT(INOUT), OPTIONAL:: BOX_SET
INTEGER, INTENT(OUT), OPTIONAL:: NUM_BOX

! Local variables.
INTEGER:: alloc_err ! Allocation error status.
INTEGER:: b_id ! Box matrix identifier.
INTEGER:: boxset_ind ! BOX_SET array index counter.
INTEGER:: col ! Local column index.
INTEGER:: eval_c ! Function evaluation counter.
INTEGER:: i, j, k ! Loop counters.
INTEGER:: ierr ! Error status for file I/O.
INTEGER:: iflag ! Error flag for subroutine calls.
INTEGER:: i_start ! Records the start index for searching in a node of
  ! 'setInd'.
INTEGER:: lbc ! If 1, LBC (limiting box columns) is enabled (default).
  ! If 0, LBC is disabled when the size of 'BOX_SET' is greater than 1 or
  ! MAX_ITER is not specified.
INTEGER:: stop_rule ! Bits 0, 1, 2, 3 being set correspond to stopping
  ! rules 1 (iteration limit), 2 (function evaluation limit), 3 (minimum
  ! box diameter),4 (relative change in 'FMIN') respectively.
INTEGER:: SWITCH_I ! Local copy of argument SWITCH.
INTEGER:: t ! Iteration counter.
LOGICAL:: do_it ! Sign to process first box in each column of BoxMatrix.
REAL(KIND = R8), DIMENSION(D):: current_center ! Center coordinates of
  ! the best unmarked box in the current box data structure.
REAL(KIND = R8):: dia ! Diameter squared associated with 'FMIN'.
REAL(KIND = R8):: dia_limit ! Minimum diameter permitted.
```

```
REAL(KIND = R8):: EPS_I ! Local copy of argument EPS.
REAL(KIND = R8):: fmin_old ! FMIN backup.
REAL(KIND = R8):: MIN_SEP_I ! Local copy of argument MIN_SEP.
REAL(KIND = R8), DIMENSION(D):: tmp_x ! Temporary variable for 'unit_x'.
REAL(KIND = R8), DIMENSION(D):: UmL ! An array containing U(:)-L(:).
REAL(KIND = R8), DIMENSION(D):: unit_x ! X normalized to unit hypercube.
REAL(KIND = R8), DIMENSION(D):: W_I ! Local copy of weights W.
TYPE(BoxMatrix), POINTER:: m_head ! The first box matrix.
TYPE(BoxLink), POINTER:: p_l ! Pointer to the current box link.
TYPE(BoxMatrix), POINTER:: p_b ! Pointer to box matrix.
TYPE(Hyperbox), POINTER:: p_box ! Box for the removed parent box.
TYPE(HyperBox), POINTER:: p_save ! Pointer to the saved best box.
TYPE(int_vector), POINTER:: p_start ! Pointer to the start node for
  ! searching the column with CONVEX_BIT set in 'setInd'.
TYPE(BoxLine):: setB ! Set of newly sampled boxes.
TYPE(int_vector), POINTER:: p_setInd ! Pointer to a node of 'setInd'.
TYPE(int_vector), POINTER:: setFcol ! A linked list. Each node holds free
  ! column indices in BoxMatrices.
TYPE(int_vector):: setI ! Set I of dimensions with the maximum side length.
TYPE(int_vector), POINTER:: setInd ! A linked list. Each node holds column
  ! indices corresponding to different squared diameters in 'setDia'.
TYPE(real_vector), POINTER:: setDia ! A linked list. Each node holds
  ! current different squared diameters from largest to smallest.
TYPE(ValList):: setW ! Function values for newly sampled center points.

! Perform sanity check of input arguments and set local variables derived
! from input arguments.
STATUS = 0
CALL sanitycheck(STATUS)
IF (STATUS /= 0) RETURN
! Assign 'row_w' and 'col_w' in terms of 'D'.
IF (D <= 10) THEN
  row_w = MAX(10, 2*D)
ELSE
  row_w = 17 + CEILING(LOG(REAL(D))/LOG(2.0))
END IF
IF (lbc == 1) THEN
  ! If LBC is used, limit 'row_w' to be not greater than 'MAX_ITER'.
  ! In case of MAX_ITER==1, force row_w = 2.
  IF (row_w > MAX(MAX_ITER, 2)) row_w = MAX(MAX_ITER, 2)
END IF
col_w = 35*D
! Tolerance for REAL number equality tests.
EPS4N = REAL(4*D, KIND=R8)*EPSILON(1.0_R8)

! Allocate 'setI', 'setB' and 'setW'.
ALLOCATE(setI%elements(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
NULLIFY(setI%flags)
setI%dim = 0
ALLOCATE(setB%Line(2*D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+1;RETURN;END IF
ALLOCATE(setB%dir(2*D),STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+1;RETURN;END IF
DO i = 1, 2*D
  ALLOCATE(setB%Line(i)%c(D), STAT = alloc_err)
  IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+1;RETURN;END IF
  ALLOCATE(setB%Line(i)%side(D), STAT = alloc_err)
  IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+1;RETURN;END IF
END DO
setB%ind = 0
ALLOCATE(setW%val(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
ALLOCATE(setW%dir(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
setW%dim = 0

! Allocate 'setDia', 'setInd', and 'setFcol' for the first box matrix.
ALLOCATE(setDia)
ALLOCATE(setDia%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
NULLIFY(setDia%next)
setDia%id = 1
setDia%dim = 0
ALLOCATE(setInd)
ALLOCATE(setInd%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
ALLOCATE(setInd%flags(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
setInd%flags(:) = 0
NULLIFY(setInd%next)
NULLIFY(setInd%prev)
setInd%id = 1
setInd%dim = 0
ALLOCATE(setFcol)
ALLOCATE(setFcol%elements(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS=ALLOC_ERROR+2;RETURN;END IF
NULLIFY(setFcol%next)
NULLIFY(setFcol%prev)
NULLIFY(setFcol%flags)
setFcol%id = 1
setFcol%dim = 0
```

```
! Allocate p_box.
ALLOCATE(p_box)
ALLOCATE(p_box%c(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS = ALLOC_ERROR+3;RETURN;END IF
ALLOCATE(p_box%side(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS = ALLOC_ERROR+3;RETURN;END IF

! Allocate tempbox.
ALLOCATE(tempbox)
ALLOCATE(tempbox%c(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS = ALLOC_ERROR+3;RETURN;END IF
ALLOCATE(tempbox%side(D), STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS = ALLOC_ERROR+3;RETURN;END IF

! Step 1: Normalization of the search space, and initialization
!         of first hyperbox.
ALLOCATE(m_head, STAT = alloc_err)
IF (alloc_err /= 0) THEN;STATUS = ALLOC_ERROR;RETURN;END IF

! Initialize 't'.
t = 0
! Initialize the data structures with the initial unit box.
iflag = 0
FMIN = HUGE(1.0_R8) ! Just so come compilers can verify INTENT(OUT)
                    ! attribute of FMIN, initialized in 'init'.
CALL init(m_head, iflag)
! Check the returned 'iflag'.
IF (iflag /= 0) THEN;STATUS = iflag;RETURN;END IF

! Initialization for main loop.
t = 1
eval_c = 1
MAIN_LOOP: DO

  ! Step 2: Identify the set S of potentially optimal boxes (rectangles).
  ! Preprocess for identifying potentially optimal hyperboxes. Find and
  ! process the hyperboxes that are on the convex hull if 'SWITCH_I'== 1;
  ! otherwise, process the first box of each column until reaching
  ! the one with 'FMIN'. Set the CONVEX_BIT in 'flags', starting
  ! from the first one in 'setInd', until reaching the column with 'FMIN'.
  p_setInd => setInd
  MARKLOOP: DO
    DO i = 1, p_setInd%dim
      p_setInd%flags(i) = IBSET(p_setInd%flags(i), CONVEX_BIT)
      ! Check if the column has reached the one with 'FMIN' that has box
      ! diameter recorded in 'dia'.
      b_id = (p_setInd%elements(i) - 1)/col_w + 1
      col = MOD(p_setInd%elements(i) - 1, col_w) + 1
      p_b => m_head
      DO j = 1, b_id - 1
        p_b => p_b%child
      END DO
      IF (dia == p_b%M(1,col)%diam) EXIT MARKLOOP
    END DO
    ! Move to the next linked node of 'setInd' if present.
    IF (ASSOCIATED(p_setInd%next)) THEN
      p_setInd => p_setInd%next
    ELSE
      ! Exit when all linked nodes of 'setInd' have been checked.
      EXIT MARKLOOP
    END IF
  END DO MARKLOOP
  IF (SWITCH_I == 1) THEN ! Convex hull processing is on.
    ! Remove the columns that are not on the convex hull of potentially
    ! optimal curve. 'p_setInd' and 'i' point to the box with 'FMIN'.
    ! Pass them to findconvex for terminating the loop of identifying
    ! boxes on convex hull when EPS_I /= 0.
    CALL findconvex(m_head, p_setInd, i, setInd)
  END IF
  ! Process the set of potentially optimal boxes. They are the first
  ! boxes of all columns with CONVEX_BIT set in 'flags' in 'setInd'.
  ! Initialize 'i_start' and 'p_start' in order to search such
  ! columns in 'setInd' starting from the smaller diameters to larger
  ! diameters so sifting up newly generated boxes is not going to override
  ! any convex hull boxes that reside on the first positions of heaps
  ! of box columns (the first boxes on columns with CONVEX_BIT set
  ! in 'flags' are potentially optimal, also called "convex hull boxes").

  ! Locate the last element in the last link of 'setInd'.
  p_start => setInd
  DO WHILE (ASSOCIATED(p_start%next))
    IF (p_start%next%dim > 0) THEN
      p_start => p_start%next
    ELSE ! Exit when the next link is empty.
      EXIT
    END IF
  END DO
  i_start = p_start%dim
  ! If OBJ_CONV is present and not zero, save 'FMIN' in 'fmin_old' to be
  ! compared with the updated 'FMIN' in subroutine sampleF.
  IF (PRESENT(OBJ_CONV)) THEN
```

```
  IF (OBJ_CONV /= 0.0_R8) fmin_old = FMIN
END IF
! Loop processing any boxes in the columns with CONVEX_BIT set in
! 'flags' in 'setInd'.
INNER:  DO
  do_it = .FALSE.
  ! Find such a box column in linked list 'setInd' starting from
  ! position 'i_start' in the node 'p_start'. If found, 'do_it' will
  ! be set TRUE and index 'i' and node 'p_setInd' will be returned.
  p_setInd => findcol(i_start, p_start, i, do_it)
  IF (do_it) THEN
    ! Step 3:
    ! Step 3a: Obtain the 'setI' of dimensions with the maximum
    !          side length for the first box on column
    !          'p_setInd%elements(i)', where 'i' is the index
    !          in 'setInd' for the column holding the hyperbox to
    !          subdivide.
    CALL findsetI(m_head, p_setInd%elements(i), setI)

    ! Step 3b: Sample new center points at c+delta*e_i and
    !          c-delta*e_i for all dimensions in 'setI', where
    !          c is the center of the parent hyperbox being processed,
    !          and e_i is the ith unit vector. Evaluate the objective
    !          function at new center points and keep track of current
    !          global minimum 'FMIN' and its associated 'unit_x'.
    CALL sampleP(p_setInd%elements(i), setI, m_head, setB)

    ! Obtain function evaluations for all new center points.
    CALL sampleF(setB, eval_c, ierr)
    IF (ierr /= 0) THEN;STATUS = ierr;RETURN;END IF

    ! Step 3c: Divide the hyperbox containing c into thirds along the
    !          dimensions in 'setI', starting with the dimension with
    !          the lowest function value of f(c +- delta*e_i) and
    !          continuing to the dimension with the highest function
    !          value f(c +- delta*e_i).
    CALL divide(i, p_setInd%id, m_head, setB, setDia, setInd, setFcol, &
                p_box, setW, setI, iflag)
    IF (iflag /= 0) THEN;STATUS = ALLOC_ERROR+iflag-1;RETURN;END IF
  ELSE
    ! There are no more boxes to divide for this iteration. Release
    ! empty box columns and apply 'LBC' (if enabled) to squeeze out
    ! unnecessary memory storage.
    CALL squeeze()
    EXIT
  END IF
  END DO INNER
  ! Check stop rules:
  ! Stop rule 1: maximum iterations.
  IF (BTEST(stop_rule, 0)) THEN
    IF (t >= MAX_ITER) THEN;STATUS = 1;EXIT MAIN_LOOP;END IF
  END IF
  ! Stop rule 2: maximum evaluations.
  IF (BTEST(stop_rule, 1)) THEN
    IF (eval_c >= MAX_EVL ) THEN;STATUS = 2;EXIT MAIN_LOOP;END IF
  END IF
  ! Stop rule 3: minimum diameter.
  ! Check if minimum diameter has been reached regardless of whether
  ! MIN_DIA was specified.
  IF (sqrt(dia) <= dia_limit) THEN;STATUS = 3;EXIT MAIN_LOOP;END IF
  ! Stop rule 4: objective function convergence
  IF (PRESENT(OBJ_CONV)) THEN
    IF ((OBJ_CONV /= 0.0_R8) .AND. (fmin_old /= FMIN)) THEN
      ! 'FMIN' has been updated.
      IF (fmin_old-FMIN < (1.0_R8 + ABS(fmin_old))*OBJ_CONV) THEN
        STATUS = 4
        EXIT MAIN_LOOP
      END IF
    END IF
  END IF

  ! Update iteration counter.
  t = t + 1
END DO MAIN_LOOP

! Preparation for return to the caller.
! Scale 'unit_x' back to 'X' in original coordinates.
X = L + unit_x*UmL
! Return current diameter of the box with 'FMIN' in the original frame.
IF (PRESENT(MIN_DIA)) MIN_DIA = SQRT(dia)
! Return the total iterations and evaluations.
IF (PRESENT(MAX_ITER)) MAX_ITER = t
IF (PRESENT(MAX_EVL)) MAX_EVL = eval_c

! Find as many as SIZE(BOX_SET) best boxes.
BOXSET: IF (PRESENT(BOX_SET)) THEN
  ! Put the function value and the center coordinates of the best box
  ! into 'BOX_SET'. Initialize the 'current_center' as the best box.
  current_center = unit_x
  boxset_ind = boxset_ind + 1
  BOX_SET(boxset_ind)%val= FMIN
  ! Scale the center coordinates to the original coordinate system.
```

```
BOX_SET(boxset_ind)%c = L + unit_x*UmL
! Loop to find SIZE(BOX_SET)-1 best boxes.
OUTER1: DO k = 1, SIZE(BOX_SET)-1
  ! Set an initial value to be compared with box function values.
  ! Reuse the real variable 'fmin_old' (FMIN backup).
  fmin_old = HUGE(0.0_R8)
  ! Loop over the entire data structure starting from the box matrix
  ! 'm_head'.
  p_b => m_head
  ! Initialize the number of marked boxes (with zero diameter) 't' to be
  ! 0. Reuse the integer variable 't' (loop counter for main loop).
  t = 0
  INNER1: DO WHILE(ASSOCIATED(p_b))
    ! Check all the columns in 'p_b'.
    INNER2: DO i = 1, col_w
      DO j = 1, MOD(p_b%ind(i)-1, row_w) + 1
        ! Only in the first pass, locate and mark the box with 'FMIN'.
        IF (k == 1) THEN
          IF (ALL(unit_x == p_b%M(j,i)%c)) THEN
            ! Fill in the 'side' and 'diam' of the first best box in
            ! BOX_SET. Scale them back to the original coordinate
            ! system. Mark this box by setting its diameter zero and
            ! update 't'.
            BOX_SET(1)%side = p_b%M(j,i)%side*UmL
            BOX_SET(1)%diam = SUM(BOX_SET(1)%side**2)
            p_b%M(j,i)%diam = 0.0_R8
            t = t + 1
            CYCLE
          END IF
        END IF
        ! Process unmarked boxes (with non-zero diameter) and
        ! update 't'.
        IF (p_b%M(j,i)%diam /= 0) THEN
          ! Compute the weighted separation between 'current_center'
          ! and the center of this box 'p_b%M(j,i)'.
          tmp_x = (current_center-p_b%M(j,i)%c)*UmL
          IF (DOT_PRODUCT(tmp_x*W_I, tmp_x) < MIN_SEP_I) THEN
            ! If the separation is less than 'MIN_SEP_I', mark this
            ! box by setting its diameter zero.
            p_b%M(j,i)%diam = 0.0_R8
            t = t + 1
          ELSE
            ! If the separation >= MIN_SEP_I, compare the function value
            ! of the box with the current best value ('fmin_old').
            IF (p_b%M(j,i)%val < fmin_old) THEN
              fmin_old = p_b%M(j,i)%val
              ! Backup the pointer to the current best box.
              p_save => p_b%M(j,i)
            END IF
          END IF
        ELSE
          t = t + 1
        END IF
      END DO
      ! Repeat the above steps if any box link exists.
      IF (p_b%ind(i) > row_w) THEN
        ! There must be box link(s).
        p_l => p_b%sibling(i)%p
        DO WHILE(ASSOCIATED(p_l))
          DO j = 1, p_l%ind
            IF (p_l%Line(j)%diam /= 0) THEN
              tmp_x = (current_center - p_l%Line(j)%c)*UmL
              IF (DOT_PRODUCT(tmp_x*W_I, tmp_x)< MIN_SEP_I) THEN
                p_l%Line(j)%diam = 0.0_R8
                t = t + 1
              ELSE
                IF (p_l%Line(j)%val < fmin_old) THEN
                  fmin_old = p_l%Line(j)%val
                  p_save => p_l%Line(j)
                END IF
              END IF
            ELSE
              t = t + 1
            END IF
          END DO
          ! Move to the next box link, if any.
          p_l => p_l%next
        END DO
      END IF
    END DO INNER2
    ! Move to the next box matrix, if any.
    p_b => p_b%child
  END DO INNER1
  IF (ASSOCIATED(p_save)) THEN
    IF(p_save%diam /= 0) THEN
      ! Found the next best box. Put it into BOX_SET and mark it.
      boxset_ind = boxset_ind + 1
      BOX_SET(boxset_ind) = p_save
      ! Scale the coordinates back to the original coordinate system.
      BOX_SET(boxset_ind)%c = L + BOX_SET(boxset_ind)%c*UmL
      BOX_SET(boxset_ind)%side = BOX_SET(boxset_ind)%side*UmL
      BOX_SET(boxset_ind)%diam = SUM(BOX_SET(boxset_ind)%side**2)
```

```
        p_save%diam = 0
        t = t + 1
        ! Update 'current_center'.
        current_center = p_save%c
      END IF
    ELSE ! Exit when the next best box is not available.
      EXIT OUTER1
    END IF
    ! If all boxes in the data structure are marked, exit.
    IF (eval_c <= t) EXIT OUTER1
  END DO OUTER1
  ! If NUM_BOX or MIN_SEP is specified as an optional input
  ! argument, assign a value to it.
  IF(PRESENT(NUM_BOX)) NUM_BOX = boxset_ind
  IF(PRESENT(MIN_SEP)) MIN_SEP = SQRT(MIN_SEP_I)
END IF BOXSET

! Deallocate all the data structures explicitly allocated, including
! box matrices, box links and setI, setB, setW, setInd, setFcol, setDia,
! and p_box.
CALL cleanup()

RETURN
CONTAINS

SUBROUTINE cleanup()
IMPLICIT NONE
! Cleans up all data structures allocated to prevent a memory leak.
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER:: i, j  ! Loop counters.
TYPE(BoxMatrix), POINTER:: p_b, p_bm
TYPE(BoxLink), POINTER:: p_l
TYPE(int_vector), POINTER:: p_seti
TYPE(real_vector), POINTER:: p_setr

! Deallocate box links and box matrices starting from the first box
! matrix. First deallocate all box links associated with each box
! matrix, and finally deallocate the box matrix.
p_b => m_head
! Check all columns with box links that will be deallocated one by one
! starting from the last box link.
DO WHILE(ASSOCIATED(p_b))
  ! Check all the columns in 'p_b'.
  DO i = 1, col_w
    IF (p_b%ind(i) > row_w) THEN
      ! There must be box link(s). Chase to the last one and start
      ! deallocating them one by one.
      p_l => p_b%sibling(i)%p
      DO WHILE(ASSOCIATED(p_l%next))
       p_l => p_l%next
      END DO
      ! Found the last box link 'p_l'. Trace back and deallocate all
      ! links.
      DO WHILE(ASSOCIATED(p_l))
        IF (ASSOCIATED(p_l%prev)) THEN
          ! 'p_l's previous link is still a box link.
          p_l => p_l%prev
        ELSE
          ! There is no box link before 'p_l'. This is the first box link
          ! of this column.
          DO j = 1, row_w
            DEALLOCATE(p_l%Line(j)%c)
            DEALLOCATE(p_l%Line(j)%side)
          END DO
          DEALLOCATE(p_l%Line)
          DEALLOCATE(p_l)
          EXIT
        END IF
        DO j = 1, row_w
          DEALLOCATE(p_l%next%Line(j)%c)
          DEALLOCATE(p_l%next%Line(j)%side)
        END DO
        DEALLOCATE(p_l%next%Line)
        DEALLOCATE(p_l%next)
      END DO
    END IF
  END DO
  ! Save the pointer of this box matrix for deallocation.
  p_bm => p_b
  ! Before it's deallocated, move to the next box matrix.
  p_b => p_b%child
  ! Deallocate this box matrix with all box links cleaned up.
  DEALLOCATE(p_bm%ind)
  DEALLOCATE(p_bm%sibling)
  DO i = 1, row_w
    DO j = 1, col_w
      DEALLOCATE(p_bm%M(i,j)%c)
```

```
      DEALLOCATE(p_bm%M(i,j)%side)
    END DO
  END DO
  DEALLOCATE(p_bm%M)
  DEALLOCATE(p_bm)
END DO

! Deallocate 'setI', 'setB' and 'setW'.
DEALLOCATE(setI%elements)
DO i = 1, 2*D
  DEALLOCATE(setB%Line(i)%c)
  DEALLOCATE(setB%Line(i)%side)
END DO
DEALLOCATE(setB%Line)
DEALLOCATE(setB%dir)
DEALLOCATE(setW%val)
DEALLOCATE(setW%dir)

! Deallocate nodes of 'setDia', 'setInd' and 'setFcol' starting from
! the last node.
p_setr => setDia
DO WHILE(ASSOCIATED(p_setr%next))
  p_setr => p_setr%next
END DO
! Found the last link pointed to by 'p_setr' of 'setDia', so deallocate
! links one by one until reaching the head node that has a null 'prev'.
DO
  DEALLOCATE(p_setr%elements)
  IF (p_setr%id /= 1) THEN
    p_setr => p_setr%prev
    DEALLOCATE(p_setr%next)
  ELSE
    DEALLOCATE(setDia)
    EXIT
  END IF
END DO
p_seti => setInd
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setInd', so deallocate
! links one by one until reaching the head node that has a null 'prev'.
DO
  DEALLOCATE(p_seti%elements)
  DEALLOCATE(p_seti%flags)
  IF (p_seti%id /= 1) THEN
    p_seti => p_seti%prev
    DEALLOCATE(p_seti%next)
  ELSE
    DEALLOCATE(setInd)
    EXIT
  END IF
END DO
p_seti => setFcol
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setFcol', so deallocate
! links one by one until reaching the head node that has a null 'prev'.
DO
  DEALLOCATE(p_seti%elements)
  IF (p_seti%id /= 1) THEN
    p_seti => p_seti%prev
    DEALLOCATE(p_seti%next)
  ELSE
    DEALLOCATE(setFcol)
    EXIT
  END IF
END DO

! Deallocate p_box.
DEALLOCATE(p_box%c)
DEALLOCATE(p_box%side)
DEALLOCATE(p_box)

! Deallocate tempbox
DEALLOCATE(tempbox%c)
DEALLOCATE(tempbox%side)
DEALLOCATE(tempbox)
RETURN
END SUBROUTINE cleanup

SUBROUTINE divide(parent_i, id, b, setB, setDia, setInd, setFcol, &
                  p_box, setW, setI, iflag)
IMPLICIT NONE
! Divide the first box on a column of one of box matrices 'b', starting
! from the dimension with minimum w to the one with maximum w, where w is
! min{f(c+delta), f(c-delta)}.
!
! On input:
! parent_i - The index in one of nodes of 'setInd' and 'setDia' for the
!            column holding the parent box to divide. Each element in
```

```
!            'setInd' has the same index as the one in 'setDia'.
! id       - The identifier of the node of type 'setInd'.
! b        - The head link of box matrices.
! setB     - A set of 'HyperBox' type structures, each with newly sampled
!            center point coordinates and the corresponding function
!            value. After dividing, it contains complete boxes with
!            associated side lengths and the squared diameters.
! setDia   - A linked list of current different squared diameters of box
!            matrices. It's sorted from the biggest to the smallest.
! setInd   - A linked list of column indices corresponding to the
!            different squared diameters in 'setDia'.
! setFcol  - A linked list of free columns in box matrices.
! p_box    - A 'HyperBox' type structure to hold removed parent box to
!            subdivide.
! setW     - A set of type 'ValList' used to sort wi values, where wi is
!            defined as min{f(c+delta*e_i), f(c-delta*e_i)}, the minimum of
!            function values at the two newly sampled points.
!
! On output:
! b        - 'b' has the parent box removed and contains the newly formed
!            boxes after dividing the parent box.
! setB     - Cleared set of type 'BoxLine'. All newly formed boxes have
!            been inserted to 'b'.
! setDia   - Updated linked list 'setDia' with new squared diameters of
!            boxes, if any.
! setInd   - Updated linked list 'setInd' with new column indices
!            corresponding to newly added squared diameters in 'setDia'.
! setFcol  - Updated linked list 'setFcol' with current free columns in
!           'b'.
! p_box    - A 'HyperBox' structure holding removed parent box to
!            subdivide.
! setW     - 'setW' becomes empty after dividing.
! setI     - A set of dimensions with the order of dimensions for
!            dividing. It is cleared after dividing.
! iflag    - Status to return.
!            0    Normal return.
!            1    Allocation failures.
!
INTEGER, INTENT(IN):: parent_i
INTEGER, INTENT(IN):: id
TYPE(BoxMatrix), INTENT(INOUT),TARGET:: b
TYPE(BoxLine), INTENT(INOUT):: setB
TYPE(real_vector),INTENT(INOUT):: setDia
TYPE(int_vector), INTENT(INOUT), TARGET:: setInd
TYPE(int_vector), INTENT(INOUT):: setFcol
TYPE(HyperBox), INTENT(INOUT):: p_box
TYPE(ValList), INTENT(INOUT):: setW
TYPE(int_vector), INTENT(INOUT):: setI
INTEGER, INTENT(OUT):: iflag

! Local variables.
INTEGER:: b_id, b_j, i, j, k, status
INTEGER, DIMENSION(2*D):: sortInd
TYPE(BoxLink), POINTER:: p_l
TYPE(BoxMatrix), POINTER:: p_b
TYPE(int_vector), POINTER:: p_i, p_setInd
REAL(KIND = R8):: temp

! Initialize 'iflag' for a normal return.
iflag = 0
! Find the desired node of 'setInd'.
p_setInd => setInd
DO i = 1, id - 1
  p_setInd => p_setInd%next
END DO

! Clear the CONVEX_BIT of 'flags' as being processed.
p_setInd%flags(parent_i) = IBCLR(p_setInd%flags(parent_i), CONVEX_BIT)
IF (p_setInd%elements(parent_i) <= col_w) THEN
  ! This column is in the head link of box matrices.
  p_b => b
  b_j = p_setInd%elements(parent_i)
ELSE
  ! Find the box matrix that contains this column.
  b_id = (p_setInd%elements(parent_i)-1)/col_w + 1
  b_j = MOD(p_setInd%elements(parent_i)-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF

! Fill out 'setW'.
DO i = 1, setB%ind, 2
  ! Add minimum 'val' of a pair of newly sampled center points
  ! into 'setW'.
  setW%val((i+1)/2) = MIN(setB%Line(i)%val, setB%Line(i+1)%val)
  setW%dir((i+1)/2) = setB%dir(i)
END  DO
setW%dim = setB%ind/2

! Find the order of dimensions for further dividing by insertion
```

```
! sorting wi's in 'setW'.
DO i = 2, setW%dim
  DO j = i, 2, -1
    IF (setW%val(j) < setW%val(j-1)) THEN
      ! Element j is smaller than element j-1, so swap them. Also,
      ! the associated directions are swapped.
      temp = setW%val(j)
      k = setW%dir(j)
      setW%val(j) = setW%val(j-1)
      setW%dir(j) = setW%dir(j-1)
      setW%val(j-1) = temp
      setW%dir(j-1) = k
    ELSE
      EXIT
    END IF
  END DO
END DO

! Sort the indices of boxes in setB according to the dividing order in
! 'setW%dir'. Record the sorted indices in 'sortInd'.
DO i = 1, setW%dim
  DO j = 1, setB%ind, 2
    IF (setB%dir(j) == setW%dir(i)) THEN
      sortInd(2*i-1) = j
      sortInd(2*i) = j + 1
    END IF
  END DO
END DO
! 'setW%dir' contains the order of dimensions to divide the parent box.
! Loop dividing on all dimensions in 'setW%dir' by setting up the new
! side lengths as 1/3 of parent box side lengths for each newly
! sampled box center.
DO i = 1, setW%dim
  temp = p_b%M(1,b_j)%side(setW%dir(i))/3.0_R8
  DO j = i, setW%dim
   setB%Line(sortInd(2*j-1))%side(setW%dir(i)) = temp
   setB%Line(sortInd(2*j))%side(setW%dir(i)) = temp
  END DO
  ! Modify the parent's side lengths.
  p_b%M(1,b_j)%side(setW%dir(i))= temp
END DO
! Clear 'setW' for next time.
setW%dim = 0

! Remove the parent box from box matrix 'p_b'.
p_box = p_b%M(1,b_j)

! Move the last box to the first position.
IF (p_b%ind(b_j) <= row_w) THEN
  ! There are no box links.
  p_b%M(1,b_j) = p_b%M(p_b%ind(b_j),b_j)
ELSE
  ! There are box links. Chase to the last box link.
  p_l => p_b%sibling(b_j)%p
  DO i = 1, (p_b%ind(b_j)-1)/row_w-1
    p_l => p_l%next
  END DO
  p_b%M(1, b_j) = p_l%Line(p_l%ind)
  p_l%ind = p_l%ind-1
END IF
p_b%ind(b_j) = p_b%ind(b_j)-1

! Adjust this box column to a heap by calling siftdown.
CALL siftdown(p_b, b_j, 1)

! Update 'setDia', 'setInd' and 'setFcol' if this column is empty.
! Find which node 'setInd' is associated with by checking 'setInd%id'.
IF (p_b%ind(b_j) == 0)THEN
  ! This column is empty. Remove this diameter squared from a
  ! corresponding node of 'setDia'.
  CALL rmNode(col_w, p_setInd%id-1, parent_i, setDia)
  ! Push the released column back to top of 'setFcol'.
  IF (setFcol%dim < col_w) THEN
    ! The head node of 'setFcol' is not full.
    CALL insNode(col_w, p_setInd%elements(parent_i), &
                 setFcol%dim + 1, setFcol)
  ELSE
    ! The head node is full. There must be at least one more node
    ! for 'setFcol'. Find the last non-full node of 'setFcol' to
    ! insert the released column.
    p_i => setFcol%next
    DO
      IF (p_i%dim < col_w) THEN
        ! Found it.
        CALL insNode(col_w, p_setInd%elements(parent_i), &
                     p_i%dim + 1, p_i)
        EXIT
      END IF
      ! Go to the next node.
      p_i=> p_i%next
    END DO
  END IF
  ! Remove the column index from a corresponding node of 'setInd'.
```

```
  CALL rmNode(col_w, 0, parent_i, p_setInd)
END IF

! Modify the diameter squared for the parent box temporarily saved in
! 'p_box'. Compute the diameter in the original frame.
p_box%diam = DOT_PRODUCT(p_box%side*UmL, p_box%side*UmL)

! Update 'dia' associated with 'FMIN', which has coordinates in 'unit_x'.
IF (ALL(unit_x == p_box%c)) dia = p_box%diam

! Compute squared diameters for all new boxes in 'setB'.
DO i = 1, setB%ind
  setB%Line(i)%diam = DOT_PRODUCT(setB%Line(i)%side*UmL, &
                                  setB%Line(i)%side*UmL)
  ! Update 'dia' if needed.
  IF (ALL(unit_x == setB%Line(i)%c)) dia = setB%Line(i)%diam
END DO

! Add all new boxes in 'setB' and 'p_box' to 'b' according to different
! squared diameters and different function values.
DO i = 1, setB%ind
  CALL insMat(setB%Line(i), b, setDia, setInd, setFcol, status, 0)
  IF (status /=0) THEN;iflag = status;RETURN;END IF
END DO
CALL insMat(p_box, b, setDia, setInd, setFcol, status, 0)
IF (status /=0) THEN;iflag = status;RETURN;END IF
! Clear 'setB' and 'setI' for calling divide() next time.
setB%ind = 0
setI%dim = 0
RETURN
END SUBROUTINE divide

FUNCTION findcol(i_start, p_start, index, do_it) RESULT(p_iset)
IMPLICIT NONE
! Find the leftmost column (setInd%elements(index)) on the lower convex
! hull, in the plot of (box diameter, function value) points, with
! CONVEX_BIT of 'flags' set in linked list 'setInd', which indicates a
! potentially optimal box to be subdivided.
!
! On input:
! i_start - The index to start searching in node 'p_start'.
! p_start - The pointer to the node at which to start searching.
!
! On output:
! index    - The found index in node 'p_iset' of the linked list 'setInd'.
! do_it    - The returned sign to continue processing or not.
! i_start  - The index at which to resume searching in node 'p_start'
!            next time.
! p_start  - The pointer to the node at which to resume searching next time.
! p_iset   - The returned node, which contains the next box to subdivide.
!
INTEGER, INTENT(INOUT) :: i_start
TYPE(int_vector), POINTER :: p_start
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: do_it
TYPE(int_vector), POINTER :: p_iset

! Local variables.
INTEGER :: i, start
TYPE(int_vector), POINTER :: p_set

do_it = .FALSE.
start = i_start
p_set => p_start
! Search from the last to the first element in the link 'p_set'.
DO WHILE(ASSOCIATED(p_set))
  DO i = start, 1, -1
    ! Find the last column with CONVEX_BIT set in 'flags' of 'p_set'.
    IF (BTEST(p_set%flags(i), CONVEX_BIT)) THEN
      ! Clear the CONVEX_BIT of 'flags' as being processed.
      p_set%flags(i) = IBCLR(p_set%flags(i), CONVEX_BIT)
      do_it = .TRUE.
      index = i
      p_iset => p_set
      ! Save them to i_start and p_start for resuming searching
      ! next time.
      i_start = i
      p_start => p_set
      EXIT
    END IF
  END DO
  ! There are no more box column with CONVEX_BIT set in 'flags' in
  ! this node. Go to the previous one (if any).
  IF (.NOT. do_it) THEN
    IF(ASSOCIATED(p_set%prev)) THEN
      p_set => p_set%prev
      ! Reset 'start' to be the last one for all the following iterations
      ! except the first one which resumed from 'i_start'.
      start = p_set%dim
    ELSE
      EXIT
    END IF
```

```
  ELSE ! Exit when the convex hull box is found.
    EXIT
  END IF
END DO

RETURN
END FUNCTION findcol

SUBROUTINE findconvex(b, p_fmin, i_fmin, setInd)
IMPLICIT NONE
! In 'setInd', clear CONVEX_BIT of columns if the first boxes on these
! columns are not on convex hull. Bit CONVEX_BIT with value 0 indicates
! the first box on the column is not one of potentially optimal boxes.
! This is determined by comparing slopes. If EPS_I is 0, starting from the
! first column, find the maximum slope from the first box on that column
! to the first boxes on all other columns until reaching the box with
! 'FMIN'. Then, starting from the next column with the first box on
! convex hull, repeat the procedure until no more columns before the
! column with 'FMIN' to check. If EPS_I is greater than 0, the outer loop
! breaks out when the maximum slope is less than the value:
! (val-(FMIN-(|FMIN|+1)EPS_I))/diam.
!
! On input:
! b      - The head link of box matrices.
! p_fmin - Pointer to the node holding the column index of the box with
!          'FMIN'.
! i_fmin - Index of the column in the node 'p_fmin'.
! setInd - A linked list holding column indices of box matrices.
!
! On output:
! setInd - 'setInd' has the modified column indices.
!
TYPE(BoxMatrix), INTENT(IN), TARGET:: b
TYPE(int_vector), POINTER:: p_fmin
INTEGER, INTENT(IN):: i_fmin
TYPE(int_vector), INTENT(INOUT), TARGET:: setInd

! Local variables.
INTEGER:: b_id1, b_id2, col1, col2, i, j, k, target_i
LOGICAL:: stop_fmin
REAL(KIND = R8):: slope, slope_max
TYPE(BoxMatrix), POINTER:: p_b1, p_b2
TYPE(int_vector), POINTER:: p_setInd1, p_setInd2, target_set

! Initialize the first node pointer.
p_setInd1 => setInd

! Initialization for outer loop that processes all columns before
! the column containing 'FMIN' in order to find a convex hull curve.
stop_fmin = .FALSE.
i = 1
k = 1
OUTLOOP: DO WHILE((.NOT. stop_fmin) .AND. ASSOCIATED(p_setInd1))
  ! Initialization for inner loop, which computes the slope from the
  ! first box on the fixed column 'i' to the first boxes on all the
  ! other columns, before reaching the column containing a box with
  ! 'FMIN', to locate the target column with maximum slope. Mark off
  ! any columns in between the fixed first column and the target column.
  NULLIFY(target_set)
  slope_max = -HUGE(slope)
  p_setInd2 => p_setInd1
  ! Fix the first convex hull column as column 'i' in 'p_setInd1'.
  ! The second column used to calculate the slope has index 'k' in
  ! 'p_setInd2'.  'k' is incremented up to the column index corresponding
  ! to 'FMIN'.  Find the box matrix 'p_b1' and the local column index
  ! 'col1'.
  b_id1 =(p_setInd1%elements(i)-1)/col_w + 1
  col1 = MOD(p_setInd1%elements(i)-1, col_w) + 1
  p_b1 => b
  DO j = 1, b_id1 - 1
    p_b1 => p_b1%child
  END DO
  ! Check if the first column has reached the column with 'FMIN'. If
  ! so, break out of the outer loop.
  IF (dia == p_b1%M(1,col1)%diam) EXIT OUTLOOP
  k = i + 1
  INLOOP: DO
    IF (k > p_setInd2%dim) THEN
      ! Move to the next node as k increments beyond the maximum
      ! length for each node of 'setInd'.
      p_setInd2 => p_setInd2%next
      IF (.NOT. ASSOCIATED(p_setInd2)) EXIT INLOOP
      IF (p_setInd2%dim == 0) EXIT INLOOP
      k = 1
    END IF
    ! To compute the slope from the first box on column 'i' of 'p_setInd1'
    ! to the first box on column 'k' of 'p_setInd2', find the local
    ! column index 'col2' and the corresponding box matrix 'p_b2'.
    b_id2 = (p_setInd2%elements(k) - 1)/col_w + 1
    col2 = MOD(p_setInd2%elements(k) - 1, col_w) + 1
    p_b2 => b
    DO j = 1, b_id2 - 1
```

```
      p_b2 => p_b2%child
    END DO
    ! Use the slope formula (f1-f2)/(d1-d2), where f1 and f2 are the
    ! function values at the centers of the two boxes with diameters
    ! d1 and d2.
    slope = (p_b1%M(1,col1)%val - p_b2%M(1,col2)%val) / &
            (SQRT(p_b1%M(1,col1)%diam) - SQRT(p_b2%M(1,col2)%diam))
    ! Compare the new slope with the current maximum slope. Keep track
    ! of the target column index and the target node.
    IF (slope > slope_max) THEN
      slope_max = slope
      target_i = k
      target_set => p_setInd2
      ! Check if this target column contains 'FMIN'.
      IF (dia == p_b2%M(1,col2)%diam) stop_fmin = .TRUE.
    END IF
    ! If the target column contains 'FMIN', break out of the inner loop.
    IF (dia == p_b2%M(1,col2)%diam) EXIT INLOOP
    ! Move on to the next column.
    k = k + 1
  END DO INLOOP
  ! Mark off boxes in between.
  IF (ASSOCIATED(target_set)) CALL markoff(i, target_i, p_setInd1, target_set)
  ! Check if EPS_I /= 0. If so, stop if the found 'slope_max' from
  ! the first box is less than the desired accuracy of the solution.
  IF ((EPS_I /= 0) .AND. ASSOCIATED(target_set)) THEN
    IF ((p_b1%M(1,col1)%val - (FMIN - (ABS(FMIN) + 1.0_R8)*EPS_I))/ &
       SQRT(p_b1%M(1,col1)%diam) > slope_max ) THEN
      ! Mark off the first boxes on the columns from the column target_i
      ! to the one with 'FMIN'.
      target_set%flags(target_i) = IBCLR(target_set%flags(target_i), &
                                         CONVEX_BIT)
      CALL markoff(target_i, i_fmin, target_set, p_fmin)
      ! Mark off the first box on the colume with 'FMIN' if it is not marked
      ! off yet as the target_set.
      IF (BTEST(p_fmin%flags(i_fmin), CONVEX_BIT)) p_fmin%flags(i_fmin) = &
          IBCLR(p_fmin%flags(i_fmin), CONVEX_BIT)
      EXIT OUTLOOP
    END IF
  END IF
  ! To start the next pass, the first fixed column jumps to the target column
  ! just found which is the next column on convex hull.
  i = target_i
  p_setInd1 => target_set
END DO OUTLOOP
RETURN
END SUBROUTINE findconvex

SUBROUTINE findsetI(b, col, setI)
IMPLICIT NONE
! Fill out 'setI', holding dimensions with the maximum side length
! of the first box on 'col' in box matrix links 'b'.
!
! On input:
! b   - The head link of box matrices.
! col - The global column index of box matrix links.
!
! On output:
! setI - The set of dimensions with the maximum side length.
!
TYPE(BoxMatrix), INTENT(IN), TARGET:: b
INTEGER, INTENT(IN):: col
TYPE(int_vector), INTENT(INOUT):: setI

! Local variables.
INTEGER:: b_id, i, j
REAL(KIND = R8) :: temp
TYPE(BoxMatrix), POINTER:: p_b

! Find the box matrix link that 'col' is associated with.
IF (col <= col_w) THEN
  p_b => b
  j = col
ELSE
  b_id = (col-1)/col_w + 1
  j = MOD(col-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF
! Search for the maximum side length.
temp = MAXVAL(p_b%M(1,j)%side(:))
! Find all the dimensions with the maximum side length.
DO i = 1, D
  IF ((ABS(p_b%M(1,j)%side(i)-temp)/temp) <= EPS4N) THEN
    ! Add dimension index to 'setI'.
    CALL insNode(D, i, setI%dim + 1, setI)
  END IF
END DO
RETURN
END SUBROUTINE findsetI
```

```
SUBROUTINE init(b, status)
IMPLICIT NONE
! Allocate the arrays and initialize the first center point.
! Evaluates the function value at the center point and initializes
! 'FMIN' and 'unit_x'.
!
! On input: None.
!
! On output:
! b      - The first box matrix to initialize.
! status - Status of return.
!          =0  Successful.
!          >0  Allocation or restart related error.
!
TYPE(BoxMatrix), INTENT(OUT), TARGET:: b
INTEGER, INTENT(OUT):: status

! Local variables.
INTEGER:: i, iflag, j

! Normal status.
status = 0

! Allocate arrays.
ALLOCATE(b%M(row_w, col_w), STAT = iflag)
IF (iflag /= 0) THEN;status = 1;RETURN;END IF
ALLOCATE(b%ind(col_w), STAT = iflag)
IF (iflag /= 0) THEN;status = 1;RETURN;END IF
! Clear the box counter for each column.
b%ind(:) = 0
! Nullify the child link to the next box matrix.
NULLIFY(b%child)
ALLOCATE(b%sibling(col_w), STAT = iflag)
IF (iflag /= 0) THEN;status = 1;RETURN;END IF
DO i = 1, col_w
  NULLIFY(b%sibling(i)%p)
END DO
DO i = 1, row_w
  DO j = 1, col_w
    ALLOCATE(b%M(i,j)%c(D), STAT = iflag)
    IF (iflag /= 0) THEN;status = 1;RETURN;END IF
    ALLOCATE(b%M(i,j)%side(D), STAT = iflag)
    IF (iflag /= 0) THEN;status = 1;RETURN;END IF
  END DO
END DO
! Starting from the last column, push free columns to 'setFcol'.
DO i = 1, col_w-1
  setFcol%elements(i) = col_w-(i-1)
END DO
! Use the first column.
setFcol%dim = col_w-1
! Initialize the center of the first unit hypercube in box matrix 'b'
! and 'unit_x' in the normalized coordinate system.
b%M(1,1)%c(:) = 0.5_R8
b%M(1,1)%side(:) = 1.0_R8
unit_x(:) = 0.5_R8
! Evaluate objective function at 'c'.
! Store the function value and initialize 'FMIN'.
iflag = 0
! For usage with VTMOP, this evaluation is performed before calling
! bVTdirect, and so OBJ_FUNC only querries VTMOP's internal database.
FMIN = OBJ_FUNC(L + b%M(1,1)%c(:)*UmL, WEIGHTS, iflag)
! Check 'iflag' to deal with undefined function values.
IF (iflag /= 0) THEN
  ! Assign a huge value to 'FMIN' for an infeasible region.
  FMIN = HUGE(1.0_R8)
END IF
b%M(1,1)%val = FMIN

! Initialize the diameter squared for this box and 'dia', the diameter
! squared associated with 'FMIN'. Convert 'side' back to the orginal frame
! for computing the real 'dia' that will be compared with 'dia_limit' or
! 'MIN_DIA' given by the user.
dia = DOT_PRODUCT(b%M(1,1)%side*UmL, b%M(1,1)%side*UmL)
b%M(1,1)%diam = dia
! Initialize the 'ind' for the first column and 'id' for this box matrix.
b%ind(1) = 1
b%id = 1
! Initialize 'setDia', 'setInd', and 'setFcol'.
setDia%dim = 1
setDia%elements(1) = b%M(1,1)%diam
! Set the first box as being on convex hull by setting the CONVEX_BIT
! of the 'flags'.
setInd%dim = 1
setInd%elements(1) = 1
setInd%flags(1) = IBSET(setInd%flags(1), CONVEX_BIT)
RETURN
END SUBROUTINE init

SUBROUTINE markoff(i, target_i, p_iset1, target_set)
```

```
IMPLICIT NONE
! Mark off columns in between the column 'i' of 'p_iset1' and column
! 'target_i' of 'target_set' by clearing the CONVEX_BIT in 'flags'.
!
! On input:
! i          - Column index of the first box for computing slope in findconvex.
! target_i   - Column index of the second box for computing slope in
!              findconvex.
! p_iset1  - The node of 'setInd' holding the column 'i'.
! target_set - The node of 'setInd' holding the column 'target_i'.
!
! On output:
! p_iset1  - 'p_iset1' has changed column indices.
! target_set - 'target_set' has changed column indices.
!
INTEGER, INTENT(IN) :: i
INTEGER, INTENT(IN) :: target_i
TYPE(int_vector), INTENT(INOUT), TARGET :: p_iset1
TYPE(int_vector), POINTER :: target_set

! Local variables.
INTEGER :: j
TYPE(int_vector), POINTER :: p_set

! Check if any columns in between.
IF (ASSOCIATED(target_set, p_iset1)) THEN
  ! If 'target_i' is next to column 'i' or no any columns in between, return.
  IF ((i == target_i) .OR. (i+1 == target_i)) RETURN
END IF

! Clear all CONVEX_BITs in 'flags' in between.
j = i
p_set => p_iset1
DO
  j = j + 1
  IF (j > p_set%dim) THEN
    p_set => p_set%next
    IF (.NOT. ASSOCIATED(p_set)) EXIT
    IF (p_set%dim == 0) EXIT
    j = 1
  END IF
  ! Check if at the target node.
  IF (ASSOCIATED(target_set, p_set)) THEN
    ! If 'j' has reached 'target_i', exit.
    IF (j == target_i) EXIT
  END IF
  ! Clear the CONVEX_BIT of 'flags' for column 'j' of 'p_set'.
  p_set%flags(j) = IBCLR(p_set%flags(j), CONVEX_BIT)
END DO
END SUBROUTINE markoff

SUBROUTINE sampleF(setB, eval_c, ierr)
IMPLICIT NONE
! Evaluate the objective function at each newly sampled center point.
! Keep updating 'FMIN' and 'unit_x'.
!
! On input:
! setB   - The set of newly sampled boxes with their center points'
!          coordinates.
! eval_c - The counter of evaluations.
!
! On output:
! setB   - The set of newly sampled boxes with added function values
!          at center points.
! eval_c - The updated counter of evaluations.
! ierr   - Return status.
!          0  Normal return.
!          >0 Error return.
!
TYPE(BoxLine), INTENT(INOUT):: setB
INTEGER, INTENT(INOUT):: eval_c
INTEGER, INTENT(OUT):: ierr

! Local variables.
INTEGER:: i, iflag

! Initialize 'ierr'.
ierr = 0

! Normal evaluation in the original coordinate system.
! Loop evaluating all the new center points of boxes in 'setB'.

! Execute function evaluations in an out-of-order parallel loop.
!$OMP PARALLEL DO SCHEDULE(DYNAMIC), DEFAULT(SHARED), PRIVATE(i, iflag)
DO i = 1, setB%ind
  iflag = 0
  setB%Line(i)%val = OBJ_FUNC(L + setB%Line(i)%c(:)*UmL, WEIGHTS, iflag)
  ! Check 'iflag'.
  IF (iflag /= 0) THEN
    ! Assign a huge value for an infeasible region.
    setB%Line(i)%val = HUGE(1.0_R8)
  END IF
```

```
END DO
!$OMP END PARALLEL DO

! Commit the function evaluations to the database in order.
DO i = 1, setB%ind
  ! Update evaluation counter.
  eval_c = eval_c + 1
  IF (FMIN > setB%Line(i)%val) THEN
    ! Update 'FMIN' and 'unit_x'.
    FMIN = setB%Line(i)%val
    unit_x(:) = setB%Line(i)%c(:)
  END IF
  ! Update FMIN and 'unit_x' in Lexicographical order.
  IF (FMIN == setB%Line(i)%val) THEN
    IF (setB%Line(i)%c .lexLT. unit_x) THEN
      ! The new point is smaller lexicographically than 'unit_x'.
      unit_x(:) = setB%Line(i)%c(:)
    END IF
  END IF
END DO
RETURN
END SUBROUTINE sampleF

SUBROUTINE sampleP(col, setI, b, setB)
IMPLICIT NONE
! In each dimension 'i' in 'setI', samples at the two points c+delta*e_i
! and c-delta*e_i. e_i is the 'i'th unit vector. In 'setB', records all
! the new points as the centers of boxes that will be formed completely
! through subroutines 'sampleF' and 'divide'.
!
! On input:
! col  - The global column index of the box to subdivide.
! setI - The set of dimensions with maximum side length.
! b    - The head link of box matrices.
! setB - The empty set of type 'HyperBox' that will hold newly sampled
!        points as the centers of new boxes.
!
! On output:
! setI - The set of dimensions with maximum side length.
! b    - The head link of box matrices.
! setB - The set of boxes that contains the newly sampled center points.
!
INTEGER, INTENT(IN):: col
TYPE(int_vector), INTENT(INOUT):: setI
TYPE(BoxMatrix), INTENT(INOUT), TARGET:: b
TYPE(BoxLine), INTENT(INOUT):: setB

! Local variables.
INTEGER:: b_id  ! Identifier of the associated box matrix.
INTEGER:: i     ! Loop counters.
INTEGER:: j     ! Local column index converted from the global one 'col'.
INTEGER:: new_i ! Index of new points in setB.
REAL (KIND=R8):: delta ! 1/3 of the maximum side length.
TYPE (BoxMatrix), POINTER:: p_b ! Pointer to the associated box matrix.

! Find the box matrix that 'col' is associated with. Store the pointer
! to box matrix in 'p_b'. The local column index 'j' will be converted from
! 'col'.
IF (col <= col_w) THEN
  p_b => b
  j = col
ELSE
  b_id = (col-1)/col_w + 1
  j = MOD(col-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF

! Find the maximum side length by obtaining the dimension in 'setI'.
! Then, extract the maximum side length from the first box on column 'j'
! of box matrix 'p_b'. Calculate 'delta', 1/3 of the maximum side length.
delta = p_b%M(1,j)%side(setI%elements(setI%dim)) / 3.0_R8

! Loop sampling two new points in all dimensions 'i' in 'setI'.
! c+delta*e_i => newpt_1; c-delta*e_i => newpt_2, where e_i is the 'i'th
! unit vector.
DO i = 1, setI%dim
  new_i = setB%ind + 1
  ! Copy the coordinates of parent box to the two new boxes
  setB%Line(new_i)%c(:) = p_b%M(1, j)%c(:)
  setB%Line(new_i+1)%c(:) = p_b%M(1, j)%c(:)
  ! Assign changed coordinates to the two new points in 'setB'.
  setB%Line(new_i)%c(setI%elements(i)) = &
    p_b%M(1, j)%c(setI%elements(i)) + delta
  setB%Line(new_i+1)%c(setI%elements(i)) = &
  p_b%M(1, j)%c(setI%elements(i))-delta

  ! Record the directions with changes in 'setB%dir' for further
  ! processing to find the dividing order of dimensions.
  setB%dir(new_i) = setI%elements(i)
```

```
  setB%dir(new_i+1) = setI%elements(i)
  ! Update 'ind' of 'setB'.
  setB%ind = setB%ind + 2
  ! Initialize side lengths of new points for further dividing
  ! by copying the sides from the parent box.
  setB%Line(new_i)%side(:) = p_b%M(1,j)%side(:)
  setB%Line(new_i+1)%side(:) = p_b%M(1,j)%side(:)
END DO
! Clear 'setI'.
setI%dim = 0
RETURN
END SUBROUTINE sampleP

SUBROUTINE sanitycheck(iflag)
IMPLICIT NONE
! Check the sanity of the input arguments, and set all local variables
! derived from input arguments.
!
! On input: None.
!
! On output:
! iflag  - The sanity check result.
!
INTEGER, INTENT(INOUT):: iflag

! Initialize 'iflag'.
iflag = 0
!
! Check required arguments.
!
IF (D < 2) THEN
  iflag = INPUT_ERROR
  RETURN
ELSE
  N_I = D
END IF

IF ((SIZE(X) /= D) .OR. (SIZE(L) /= D) .OR. (SIZE(U) /= D)) THEN
  iflag = INPUT_ERROR + 1
  RETURN
END IF
IF ((SIZE(WEIGHTS) /= P)) THEN ! Check WEIGHTS(:) against P.
  iflag = INPUT_ERROR + 1
  RETURN
END IF
IF (ANY(L >= U)) THEN
  iflag = INPUT_ERROR + 2
  RETURN
END IF
IF (ANY(WEIGHTS < 0.0_R8)) THEN ! Check that WEIGHTS(:) is nonnegative.
  iflag = INPUT_ERROR + 2
  RETURN
END IF
!
! Check optional arguments.
!
IF (PRESENT(W)) THEN
  IF (SIZE(W) /= D) THEN
    iflag = INPUT_ERROR + 1
    RETURN
  END IF
END IF
! Default: processing boxes only on convex hull.
SWITCH_I = 1
IF (PRESENT(SWITCH)) THEN
  IF ((SWITCH < 0) .OR. (SWITCH > 1)) THEN
    iflag = INPUT_ERROR + 5
    RETURN
  END IF
  IF (SWITCH == 0) THEN
    IF (PRESENT(EPS)) THEN
      IF (EPS > 0.0_R8) THEN
        iflag = INPUT_ERROR + 6
        RETURN
      END IF
    END IF
  END IF
  ! Assign the local copy 'SWITCH_I'.
  SWITCH_I = SWITCH
END IF
! Enable LBC (limiting box columns) by default.
lbc = 1
stop_rule = 0
! When MAX_ITER <= 0, the number of iterations will be returned on exit.
IF (PRESENT(MAX_ITER)) THEN
  IF (MAX_ITER > 0) THEN
    ! Set bit 0 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE1)
  ELSE ! Disable LBC (limiting box columns).
    lbc = 0
  END IF
ELSE ! Disable LBC (limiting box columns).
```

```
  lbc = 0
END IF
! When MAX_EVL <=0, the number of evaluations will be returned on exit.
IF (PRESENT(MAX_EVL)) THEN
  IF (MAX_EVL > 0) THEN
    ! Set bit 1 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE2)
    ! When 'MAX_ITER' is positive and 'MAX_EVL' is
    ! sufficiently small, disable LBC (limiting box columns).
    IF (PRESENT(MAX_ITER)) THEN
      IF ((MAX_ITER > 0) .AND. (MAX_EVL*(2*D+2) < 2D+6)) lbc = 0
    END IF
  END IF
END IF

! Even if user doesn't specify 'MIN_DIA', a diameter smaller than
! SQRT(SUM(UmL*UmL))*EPSILON(1.0_R8) is not permitted to occur.
! When MIN_DIA <=0, the diameter associated with X and FMIN will be
! returned on exit. Assign 'UmL'.
UmL(:) = U(:) - L(:)
dia_limit =  SQRT(SUM(UmL*UmL))*EPSILON(1.0_R8)
IF (PRESENT(MIN_DIA)) THEN
  IF (MIN_DIA > 0.0_R8) THEN
    IF (MIN_DIA < dia_limit) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      dia_limit = MIN_DIA
      ! Set bit 2 of stop_rule.
      stop_rule = IBSET(stop_rule, STOP_RULE3)
    END IF
  END IF
END IF

! When OBJ_CONV is present a minimum relative change in the minimum
! objective function value will be enforced.
IF (PRESENT(OBJ_CONV)) THEN
  IF (OBJ_CONV /= 0.0_R8) THEN
    IF ((OBJ_CONV < EPSILON(1.0_R8)*REAL(D,KIND=R8))&
       .OR.(OBJ_CONV >= 1.0_R8)) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      ! Set bit 3 of stop_rule.
      stop_rule = IBSET(stop_rule, STOP_RULE4)
    END IF
  END IF
END IF

! When EPS is present a test involving EPS is used to define potentially
! optimal boxes.  The absence of this test is equivalent to EPS=0.
EPS_I = 0.0_R8
IF (PRESENT(EPS)) THEN
  IF (EPS /= 0.0_R8) THEN
    IF ((EPS < EPSILON(1.0_R8)) .OR. (EPS > 1.0_R8)) THEN
      iflag = INPUT_ERROR + 3
      RETURN
    ELSE
      EPS_I = EPS
    END IF
  END IF
END IF

! Check if stop_rule has at least at 1 bit set. Otherwise no stopping rule
! has been given.
IF (stop_rule == 0) THEN;iflag = INPUT_ERROR+4;RETURN;END IF

! Check if MIN_SEP, W, and BOX_SET are correctly set.
IF (PRESENT(BOX_SET)) THEN
  ! Set default weights for distance definition.
  W_I(1:D) = 1.0_R8
  ! Verify and set weights for distance definition.
  IF (PRESENT(W)) THEN
    WHERE (W > 0)
      W_I = W
    ELSEWHERE
      W = 1.0_R8
    END WHERE
  END IF
  ! Compute the weighted diameter of the original design space. Reuse
  ! the variable 'dia' (the diameter squared associated with 'FMIN').
  dia = SQRT(SUM(UmL*W_I*UmL))
  ! Check the optional argument MIN_SEP. Set a default value if MIN_SEP
  !  is not present or not correctly assigned.
  MIN_SEP_I = (0.5_R8*dia)**2
  IF (PRESENT(MIN_SEP)) THEN
    IF ((MIN_SEP < dia*EPSILON(1.0_R8)) .OR. (MIN_SEP > dia)) &
      MIN_SEP = 0.5_R8*dia
    MIN_SEP_I = MIN_SEP**2
  END IF
  ! Check the size of its component arrays c(:) and side(:).
  DO i = 1, SIZE(BOX_SET)
```

```
    IF (ASSOCIATED(BOX_SET(i)%c)) THEN
      IF (SIZE(BOX_SET(i)%c(:)) /= D) THEN
        iflag = ALLOC_ERROR + 4
        RETURN
      END IF
    ELSE ! Allocate component 'c'.
      ALLOCATE(BOX_SET(i)%c(D))
    END IF
    IF (ASSOCIATED(BOX_SET(i)%side)) THEN
      IF (SIZE(BOX_SET(i)%side) /= D) THEN
        iflag = ALLOC_ERROR + 4
        RETURN
      END IF
    ELSE ! Allocate component 'side'.
      ALLOCATE(BOX_SET(i)%side(D))
    END IF
  END DO
  ! Initialize the index counter 'boxset_ind'.
  boxset_ind = 0
  ! Disable LBC (limiting box columns) that removes boxes needed for
  ! finding 'BOX_SET'.
  lbc = 0
END IF

RETURN
END SUBROUTINE sanitycheck

SUBROUTINE squeeze()
IMPLICIT NONE
! Scan through box columns to remove empty box columns and squeeze box
! column lengths to MAX_ITER-t+1 if LBC is enabled.
!
! On input: None.
!
! On output: None.
!
! Local variables.
INTEGER:: b_id ! Box matrix index.
INTEGER:: i, i1, j, j1 ! Loop counters.
INTEGER:: leaf ! Index to a leaf node in a heap(a box column).
INTEGER:: maxid ! Global index to a box with the largest function value.
INTEGER:: maxpos ! Local index of the box (ranked 'maxid' globally in the
  ! heap) in the box matrix or box link.
INTEGER:: mycol ! Local box column index.
TYPE(BoxLink), POINTER:: p_l, p_l1 ! Pointer to a box link.
TYPE(BoxMatrix), POINTER:: p_b ! Pointer to a box matrix.
TYPE(int_vector), POINTER:: p_seti ! Pointer to a link of 'int_vector'.
REAL(KIND = R8):: maxf ! The largest function value in a heap.
REAL(KIND = R8), DIMENSION(D):: maxc ! Coordinates of the box with 'maxf'.

! The scan starts from the first box column indexed in 'setInd'.
p_seti => setInd
DO WHILE(ASSOCIATED(p_seti))
  i = 1
  DO
    ! Exit when reaching the last box column index in the setInd link
    ! pointed to by 'p_seti'.
    IF (i > p_seti%dim) EXIT
    IF (p_seti%elements(i) <= col_w) THEN
      ! This box column is in the head link of the linked list of box
      ! matrices. Let 'p_b' point to the box matrix and find the local
      ! column index.
      p_b => m_head
      mycol = p_seti%elements(i)
    ELSE
      ! Find the box matrix link that contains this box column.
      b_id = (p_seti%elements(i)-1)/col_w + 1
      mycol = MOD(p_seti%elements(i)-1, col_w) + 1
      p_b => m_head
      DO j = 1, b_id-1
        p_b => p_b%child
      END DO
    END IF
    IF (p_b%ind(mycol) == 0)  THEN
      ! This column is empty. Remove this diameter squared from the
      ! corresponding node of 'setDia'.
      CALL rmNode(col_w, p_seti%id-1, i, setDia)
      ! Push the released box column back to top of 'setFcol'.
      IF (setFcol%dim < col_w) THEN
        ! The head node of 'setFcol' is not full.
        CALL insNode(col_w, p_seti%elements(i), setFcol%dim + 1, setFcol)
      ELSE
        ! The head node is full. There must be at least one more node
        ! for 'setFcol'. Find the last non-full node of 'setFcol' to
        ! insert the released column.
        p_seti => setFcol%next
        DO
          IF (p_seti%dim < col_w) THEN
            ! Found it.
            CALL insNode(col_w, p_seti%elements(i), p_seti%dim + 1, p_seti)
            EXIT
          END IF
```

```
      ! Go to the next node.
      p_seti=> p_seti%next
    END DO
  END IF
  ! Remove the column index from a corresponding node of 'setInd'.
  CALL rmNode(col_w, 0, i, p_seti)
  ! Decrease 'i' by 1 to compensate for this removal.
  i = i - 1
ELSE ! This box column is not empty.
  IF (lbc == 1) THEN ! Adjust the box column length to
                     ! MAX(MAX_ITER - t + 1, row_w).
    IF (p_b%ind(mycol) > MAX(MAX_ITER - t + 1, row_w)) THEN
      ! Loop removing the largest element until the goal is reached.
      DO
        IF (p_b%ind(mycol) == MAX(MAX_ITER - t + 1,row_w)) EXIT
        ! Look for the largest element starting from the first leaf.
        ! Initialize and keep update variables ('maxid', 'maxpos',
        ! 'maxf', and 'maxc').
        leaf = p_b%ind(mycol)/2 + 1
        IF (leaf <= row_w) THEN
          ! The first leaf node starts inside 'M', part of the box
          ! matrix.
          maxid = 0
          maxpos = leaf
          maxf = p_b%M(leaf,mycol)%val
          maxc(:) = p_b%M(leaf,mycol)%c(:)
        ELSE ! The first leaf node starts from a box link.
          maxid = (leaf - 1)/row_w
          maxpos = MOD(leaf - 1, row_w) + 1
          p_l => p_b%sibling(mycol)%p
          DO k = 1, maxid - 1
            p_l => p_l%next
          END DO
          maxf = p_l%Line(maxpos)%val
          maxc(:) = p_l%Line(maxpos)%c(:)
        END IF
        IF (maxid == 0) THEN ! Search starts from 'M'.
          DO k = maxpos + 1, row_w ! Compare with the boxes positioned
            ! later than the current 'maxpos' in 'M'.
            ! Exit when reaching the end.
            IF (k > p_b%ind(mycol)) EXIT
            ! Update the variables with the newly found box that has
            ! larger function value or larger lexicographical order.
            IF((p_b%M(k,mycol)%val > maxf) .OR.  &
              ((p_b%M(k,mycol)%val == maxf) .AND.  &
              (maxc .lexLT. p_b%M(k,mycol)%c))) THEN
              maxf = p_b%M(k,mycol)%val
              maxc(:) = p_b%M(k,mycol)%c(:)
              maxpos = k
            END IF
          END DO
          ! Search moves on to the box links (if any) and reset
          ! counters.
          p_l => p_b%sibling(mycol)%p
          k = 0
          j = 0
          i1 = 0
        ELSE
          ! Start search from a box link and initialize the counters.
          k = maxid - 1
          j = maxpos
          i1 = maxid - 1
        END IF
        ! Search through all the box links after the current box with
        ! the largest value, which is stored in the box link indexed
        ! by 'i1' (if 0, the box is in 'M').
        DO j1 = 1, (p_b%ind(mycol) - 1)/row_w - i1
          ! Exit if there are no more box links.
          IF (.NOT. ASSOCIATED(p_l)) EXIT
          IF (p_l%ind == 0) THEN
            ! Deallocate an empty box link and exit.
            DEALLOCATE(p_l)
            EXIT
          END IF
          k = k + 1 ! Increase the global index of a box link.
          DO
            j = j + 1 ! Increase the local counter inside a box link.
            IF (j > p_l%ind) THEN
              ! Reset 'j' at the end of the box link.
              j = 0
              EXIT
            END IF
            ! Update the varaibles with the newly found box that has
            ! larger function valu or larger lexicographical order.
            IF ((p_l%Line(j)%val > maxf) .OR.  &
               ((p_l%Line(j)%val == maxf) .AND.  &
               (maxc .lexLT. p_l%Line(j)%c))) THEN
              maxf = p_l%Line(j)%val
              maxc = p_l%Line(j)%c
              maxid = k
              maxpos = j
            END IF
```

```
  END DO
  p_l => p_l%next ! Move to the next link.
END DO
! Found the largest fval box at link maxid and position maxpos.
! If maxpos is not the last element position in the heap,
! replace position maxpos with the last element of the heap
! and sift it up.
IF (maxpos + maxid*row_w < p_b%ind(mycol)) THEN
  ! Locate the last heap element.
  IF (p_b%ind(mycol) <= row_w) THEN
    ! The last element is inside M.
    p_b%M(maxpos,mycol) = p_b%M(p_b%ind(mycol),mycol)
    ! Update the box counter for the box column.
    p_b%ind(mycol) = p_b%ind(mycol) - 1
    CALL siftup (p_b, mycol, maxpos)
  ELSE ! The last element is inside a box link.
    IF (maxid == 0) THEN
      ! The largest element is inside M.
      p_l1 => p_b%sibling(mycol)%p
      DO k = 1, (p_b%ind(mycol) - 1)/row_w - 1
        p_l1 => p_l1%next
      END DO
      p_b%M(maxpos,mycol) = &
        p_l1%Line(MOD(p_b%ind(mycol) - 1, row_w) + 1)
    ELSE ! The largest element is inside a box link.
      p_l => p_b%sibling(mycol)%p
      DO k = 1, maxid - 1
        p_l => p_l%next
      END DO
      p_l1 => p_l
      DO k = 1, (p_b%ind(mycol) - 1)/row_w - maxid
        p_l1 => p_l1%next
      END DO
      p_l%Line(maxpos) = &
        p_l1%Line(MOD(p_b%ind(mycol) - 1,row_w) + 1)
    END IF
    ! Update counters for the box link and the box column.
    p_l1%ind = p_l1%ind - 1
    p_b%ind(mycol) = p_b%ind(mycol) - 1
    ! Siftup the box that replaced the largest fval box.
    CALL siftup(p_b, mycol, maxid*row_w + maxpos)
  END IF
ELSE ! The largest element is the last element.
  IF (maxid > 0) THEN
    ! Locate the box link that holds the largest element.
    p_l => p_b%sibling(mycol)%p
      DO k = 1, (p_b%ind(mycol) - 1)/row_w - 1
        p_l => p_l%next
      END DO
      p_l%ind = p_l%ind - 1
    END IF
    ! Update the counter.
    p_b%ind(mycol) = p_b%ind(mycol) - 1
  END IF
 END DO
 END IF
 END IF
 END IF
 ! Go to the next box column.
 i = i + 1
 END DO
 ! Go to the next 'setInd' link.
 p_seti => p_seti%next
END DO
RETURN
END SUBROUTINE squeeze

END SUBROUTINE bVTdirect

END MODULE bVTdirect_MOD
```

C.3 Code for VTMOP with libEnsemble

C.3.1 vtmop_initializer.f90

```
PROGRAM VTMOP_INITIALIZER
! Implement a generator function to be used by libEnsemble.
USE ISO_FORTRAN_ENV
USE VTMOP_MOD
IMPLICIT NONE
! Variables.
INTEGER :: D, P ! Problem dimensions.
INTEGER :: LBATCH ! Length of the batch.
INTEGER :: NB ! Preferred batch size.
INTEGER :: IERR ! Error flag.
REAL(KIND=R8) :: TRUST_RAD ! Trust region radius.
REAL(KIND=R8), ALLOCATABLE :: BATCHX(:,:) ! Candidate design points to evaluate.
REAL(KIND=R8), ALLOCATABLE :: LB(:), UB(:) ! Bound constraints.
REAL(KIND=R8), ALLOCATABLE :: DES(:,:), OBJ(:,:)
TYPE(VTMOP_TYPE) :: VTMOP

! Open an existing I/O file using unformatted read and get input dimensions.
OPEN(12, FILE="vtmop.io", FORM="unformatted", ACTION="read", &
     STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT,"(A)") "An error occurred while opening the VTMOP input"
   GO TO 100; END IF
READ(12, IOSTAT=IERR) D, P, NB
ALLOCATE(LB(D), UB(D), STAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT,"(A)") "An error occurred while reading the VTMOP metadata"
   GO TO 100; END IF
ALLOCATE(DES(D,1), OBJ(P,1), STAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT,"(A)") "An error occurred while allocated dummy arrays"
   GO TO 100; END IF
READ(12, IOSTAT=IERR) LB(:), UB(:)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT,"(A)") "An error occurred while reading the constrainst"
   GO TO 100; END IF
READ(12, IOSTAT=IERR) TRUST_RAD
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT,"(A)") "An error occurred while reading the LTR radius"
   GO TO 100; END IF
CLOSE(12)

! Initialize the VTMOP status object.
CALL VTMOP_INIT( VTMOP, D, P, LB, UB, IERR, TRUST_RADF=TRUST_RAD, ICHKPT=1 )
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A,I4)") &
        "An error occurred while initializing. Error code: ", IERR
    GO TO 100
END IF

! Call the generator function and save to the checkpoint.
CALL VTMOP_GENERATE( VTMOP, DES, OBJ, LBATCH, BATCHX, IERR, NB=NB )
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A,I4)") &
        "An error occurred while generating initial batch. Error code: ", &
        IERR
    GO TO 100
END IF

! Open I/O file using unformatted write, and record the output.
OPEN(12, FILE="vtmop.io", FORM="unformatted", ACTION="write", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while opening the output file for the initializer"
   GO TO 100; END IF
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while writing the output file for the initializer"
   GO TO 100; END IF
WRITE(12, IOSTAT=IERR) BATCHX(:,:)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while writing the output file for the initializer"
   GO TO 100; END IF
CLOSE(12)

! End of program marker.
100 CONTINUE

END PROGRAM VTMOP_INITIALIZER
```

C.3.2 vtmop_generator.f90

```
PROGRAM GENERATOR
! Implement a generator function to be used by libEnsemble.
USE ISO_FORTRAN_ENV
USE VTMOP_MOD
IMPLICIT NONE
! Variables.
INTEGER :: D, P ! Problem dimensions.
INTEGER :: LBATCH ! Length of the batch.
INTEGER :: N ! Number of points in database.
INTEGER :: SNB, ONB, NB ! Preferred batch size.
INTEGER :: IERR ! Error flag.
REAL(KIND=R8), ALLOCATABLE :: BATCHX(:,:) ! Candidate design points to evaluate.
REAL(KIND=R8), ALLOCATABLE :: LB(:), UB(:) ! Bound constraints.
TYPE(VTMOP_TYPE) :: VTMOP

! Open an existing I/O file using unformatted read and get input dimensions.
OPEN(12, FILE="vtmop.io", FORM="unformatted", ACTION="read", &
     STATUS="old", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while opening the input file for the generator"
   GO TO 100; END IF
READ(12, IOSTAT=IERR) D, P, N, SNB, ONB
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while reading the input metadata for the generator"
   GO TO 100; END IF
ALLOCATE(LB(D), UB(D), STAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while allocating the bound arrays for the generator"
   GO TO 100; END IF
READ(12, IOSTAT=IERR) LB(:), UB(:)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while reading the constraints for the generator"
   GO TO 100; END IF
CLOSE(12)

! Recover the VTMOP status object.
CALL VTMOP_INIT( VTMOP, D, P, LB, UB, IERR, ICHKPT=-1 )
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A,I4)") &
        "An error occurred while recovering VTMOP object. Error code: ", IERR
   GO TO 100
END IF

! Recover the cost function data.
CALL VTMOP_RECOVER_DATA(VTMOP_MOD_DBN, VTMOP_MOD_DBX, VTMOP_MOD_DBF, IERR, &
                        DB_SIZE=N)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A,I4)") &
        "An error occurred while recovering data. Error code: ", IERR
   GO TO 100
END IF

! Check whether this is an optimization or search phase.
IF (ALLOCATED(VTMOP%WEIGHTS)) THEN
   NB = ONB
ELSE
   NB = SNB
END IF

! Call the generator function and save to the checkpoint.
CALL VTMOP_GENERATE( VTMOP, VTMOP_MOD_DBX(:,1:VTMOP_MOD_DBN),          &
                     VTMOP_MOD_DBF(:,1:VTMOP_MOD_DBN), LBATCH, BATCHX, &
                     IERR, NB=NB )
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A,I4)") &
        "An error occurred while generating candidates. Error code: ", IERR
   GO TO 100
END IF

! Open I/O file using unformatted write, and record the output.
OPEN(12, FILE="vtmop.io", FORM="unformatted", ACTION="write", IOSTAT=IERR)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while opening the output file for the generator"
   GO TO 100; END IF
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while writing the output file for the generator"
   GO TO 100; END IF
WRITE(12, IOSTAT=IERR) BATCHX(:,:)
IF (IERR .NE. 0) THEN
   WRITE(ERROR_UNIT, "(A)") &
        "An error occurred while writing the output file for the generator"
   GO TO 100; END IF
```

```
CLOSE(12)

! End of program marker.
100 CONTINUE

END PROGRAM GENERATOR
```

C.3.3 vtmop.py

```
"""
Wrapper for the VTMOP genfuncs drivers, for solving multiobjective
optimization problems with arbitrary number of objectives.
"""
import numpy as np
from scipy.io import FortranFile  # for reading/writing unformatted binary data
from os import system  # for issuing batch commands


def vtmop_gen(H, persis_info, gen_specs, _):
    """
    This generator function solves multiobjective optimization problems
    with d design variables (subject to simple bound constraints) and
    p objectives using VTMOP.

    This requires that VTMOP be installed and in the system PATH.
    To do so, download VTMOP (contact thchang@vt.edu), and use the
    command

    $ make genfuncs

    to build the generator functions. Next, run the command

    $ export PATH=$PATH:‘pwd‘

    from the VTMOP source/build directory to add VTMOP to your system
    PATH.

    This generator alternates between generating large batches of size
    gen_specs['search_batch_size'] to explore design regions, and small
    batches of size gen_specs['opt_batch_size'] to fill in gaps on the
    Pareto front. An initial search size can also be specified by using
    gen_specs['first_batch_size'].

    gen_specs['ub'] and gen_specs['lb'] must specify upper and lower
    bound constraints on each design variable. The number of design variables
    is inferred from len(gen_specs['ub']). gen_specs['num_obj']
    specifies the number of objectives. The problem dimension is inferred
    based on the length of the gen_specs['lb']. gen_specs['restart']
    specifies whether to reinitialize VTMOP.

    Several unformatted binary files (vtmop.io, vtmop.dat, and vtmop.chkpt)
```

```
    will be generated in the calling directory to pass information between
    libEnsemble and VTMOP.
    """
    # First get the problem dimensions and data
    ub = gen_specs['user']['ub']                  # upper bounds
    lb = gen_specs['user']['lb']                  # lower bounds
    p = gen_specs['user']['num_obj']              # objective dimension
    snb = gen_specs['user']['search_batch_size']  # preferred batch size for
searching
    onb = gen_specs['user']['opt_batch_size']     # preferred batch size for
optimization
    inb = gen_specs['user']['first_batch_size']   # batch size for first
iteration
    tr = gen_specs['user']['trust_rad']           # get the trust region radius
    chkpt_flag = gen_specs['user']['use_chkpt']   # Are we using a checkpoint?

    d = len(lb)                                   # design dimension
    n = np.size(H['f'][:, 0])                     # size of database in the
history array

    if not chkpt_flag:
        # Write initialization data to the vtmop.io file for VTMOP_INIT
        fp1 = FortranFile('vtmop.io', 'w')
        fp1.write_record(np.array([np.int32(d), np.int32(p), np.int32(inb)]))
        fp1.write_record(np.array([np.array(lb, dtype=np.float64),
                        np.array(ub, dtype=np.float64)]))
        fp1.write_record(np.float64(tr))
        fp1.close()
        system("vtmop_initializer")
        gen_specs['user']['use_chkpt'] = True
        # If the initial batch size is zero, run another half iteration
        if inb == 0:
            # Write unformatted problem dimensions to the vtmop.io file
            fp1 = FortranFile('vtmop.io', 'w')
            fp1.write_record(np.array([np.int32(d), np.int32(p), np.int32(n),
                            np.int32(snb), np.int32(onb)]))
            fp1.write_record(np.array([np.array(lb, dtype=np.float64),
                            np.array(ub, dtype=np.float64)]))
            fp1.close()
            # Write unformatted history to the vtmop.dat file
            fp2 = FortranFile('vtmop.dat', 'w')
            fp2.write_record(np.array([np.int32(d), np.int32(p)]))
            for i in range(n):
                toadd = np.zeros(d+p)
                toadd[:d] = np.float64(H['x'][i, :])
                toadd[d:] = np.float64(H['f'][i, :])
                fp2.write_record(toadd)
            fp2.close()
            system("vtmop_generator")
    else:
        # Write unformatted problem dimensions to the vtmop.io file
        fp1 = FortranFile('vtmop.io', 'w')
        fp1.write_record(np.array([np.int32(d), np.int32(p), np.int32(n),
                        np.int32(snb), np.int32(onb)]))
        fp1.write_record(np.array([np.array(lb, dtype=np.float64),
                        np.array(ub, dtype=np.float64)]))
        fp1.close()
        # Write unformatted history to the vtmop.dat file, to be read by VTMOP
        fp2 = FortranFile('vtmop.dat', 'w')
        fp2.write_record(np.array([np.int32(d), np.int32(p)]))
        for i in range(n):
            toadd = np.zeros(d+p)
            toadd[:d] = np.float64(H['x'][i, :])
            toadd[d:] = np.float64(H['f'][i, :])
            fp2.write_record(toadd)
        fp2.close()
        # Call VTMOP from command line
        system("vtmop_generator")

# Read unformatted list of candidates from vtmop.io file
fp1 = FortranFile('vtmop.io', 'r')
cand_pts = fp1.read_record(np.float64)
fp1.close()

# Get the true batch size
b = cand_pts.size // d

# Read record
Out = np.zeros(b, dtype=gen_specs['out'])
for i in range(0, b):
    Out['x'][i] = cand_pts[d*i:d*(i+1)]

return Out, persis_info
```