Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2020

Project Title:	**Static Optimisation in the Julia Programming Language**
Student:	**Edward P. Stables**
CID:	**01220379**
Course:	**EEE4**
Project Supervisor:	**Dr Eric Kerrigan**
Second Marker:	**Dr Fei Teng**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, electronic copies of my final year project report to both Blackboard and the EEE coursework submission system.

I affirm that the two copies of the report are identical.

I affirm that I have provided explicit references for all material in my Final Report which is not authored by me and represented as my own work.

I would like to thank my supervisor Dr Eric Kerrigan, as well as Ian McInerney and Omar Faqir for their constant help and guidance throughout this project.

To my parents, thank you for always supporting me, especially over the last four years.

To my friends, thank you for ensuring I never got any work done in the labs, and that my cup of tea was never empty.

And lastly to Fangfang, for everything over the last four years.

This project concerns the design and implementation of a pair of numerical optimisation software packages that focus on being very simple to extend, modify, and experiment with. The classes of algorithms in question are row action methods, and direct search methods. The intention of the project is to deliver software that utilises the high performance characteristics of Julia to provide good optimisation performance, while also remaining simple to understand, extend, and modify. Each package defines a framework for the implementation of algorithms within the target class, allowing an algorithm designer to quickly build running code within an environment that abstracts many of the more technical or difficult aspects of the software design process. The software as a whole, as well as any algorithms within it, is evaluated on both its correctness and performance, with comparisons made to other existing algorithm implementations and software packages.

Contents

1 Introduction

The field of mathematical optimisation faces a problem in that the vast majority of available software packages are designed in a manner that disallows easy modification or adaption of the software internals. This is due to these packages using compiled languages that do not lend themselves to easy modification, as well as being highly optimised for performance. While these are desirable characteristics for deployed software, or circumstances where a specified algorithm is required, it is problematic in an developmental setting where the ability to experiment with aspects of an algorithm is generally more desirable than the performance of the software.

This project intends to tackle this issue by implementing a pair of optimisation software packages which are designed in such a way that experimentation and modification of algorithms is simple, but performance is still competitive with traditional software packages. The increasing popularity of the Julia programming language [1] has inspired this project as the software paradigm it offers is unique. Julia offers a familiar mathematical syntax, a code structure that is perfectly suited to the aim of the project, and better performance than the vast majority of its contemporaries [2].

To the best of our knowledge there are no equivalent optimisation software packages that prioritise this algorithmic flexibility, let alone ones that also maintain the majority of the performance of their highly optimised counterparts.

The two packages being developed in this project each focus on a different subset of optimisation algorithms. The first focuses on row action methods, and the second on direct-search methods. These are named *RowActionMethods.jl* (RAM) and *DirectSearch.jl* (DS) respectively (these names follow the Julia convention of appending '*.jl*' to signify package names).

Row action methods are a class of algorithm that are characterised by only accessing a single row of a problem matrix at a time, and have shown good performance when applied to large and sparse problems [3]. One of the motivations of using these algorithms is the promise that they have shown when utilised in a distributed computing environment [4]. It is not believed that there are any alternative software packages that focus solely on these kinds of methods.

Direct search methods are a broad class of derivative free optimisation algorithms, with many different algorithms fitting within the classification [5]. The subset of direct search algorithms being focused on in this project is the Mesh Adaptive Direct Search (MADS) family of algorithms [6]. MADS requires no analytical knowledge about the objective function or constraints of a problem. It has been shown to be effective when the objective function is very expensive to calculate (therefore a minimum number of evaluations is desirable), as well as for non-smooth functions (making the application of solvers that have a requirement for derivative information or smoothness ineffective).

The software and algorithms developed in this project will be extensively tested to ensure that they are free of error, and benchmarked to demonstrate that they offer very similar performance to algorithms that do not offer the same degree of flexibility.

2 Algorithms

2.1 Row Action Methods

Row action methods can be applied to a variety of smooth objective functions, with each algorithm in the class differing in its particular target function. These algorithms show good performance on huge, sparse matrices which have no detectable structure that may be exploited by other algorithms. A collection of such methods is put forwards by [3], which characterises row action methods as methods that:

- Make no changes to the original problem matrix.
- Perform no operations on the matrix as a whole.
- Access only a single row of the matrix for each iteration.
- When calculating the result of an iterative step, the only other iterate needed is the result of the current iteration's predecessor.

This class of algorithms is not presented as a well defined set, but rather as a broad category that algorithms may be adapted to fit within. The target row action method for this project is Hildreth's algorithm and as such, a detailed description of the algorithm is given. This description will be referenced later in the report when discussing the implementation of the package.

Most row action methods can be understood as a series of projections. Methods differ in how this is performed, for example Kaczmarz's method [4][7] solves a system of linear equations with a series of orthogonal projections onto the problem equations. This is shown in Figure 1, where the H_i are the equations, and on each iteration the incumbent point $x^{(k)}$ is projected onto the equations in order, bringing it closer to the solution.

2.1.1 Hildreth's Algorithm

Hildreth published his algorithm as a general procedure for solving quadratic programming problems [8]. However its easy adaption to a row action formulation is noted by Censor's paper on these methods. In an additional publication, Lent and Censor describe extensions to this method that provide improved performance of the algorithm [9]. While this improved method is not implemented in this project, their formulation of the algorithm is also used here, and their proof that cyclic control does not affect convergence is critical in the adaption of the algorithm to a multi-threaded environment (Section 5.8)

The projection process is more complicated when compared to Kaczmarz's algorithm, and the possible cases are shown in Figure 2. Firstly, rather than solving a system of linear equations, the algorithm uses the inequality constraints of a quadratic programming problem. In the example the bold lines in are the borders of the the halfspace that each constraint defines, with the dashes showing the valid halfspace. The example assumes that the pictured boundary is the one under consideration for the current iteration. The three cases are as follows:

- If the incumbent point $x^{(k)}$ lies on the boundary, then the point is unchanged.
- If $x^{(k)}$ is outside the feasible halfspace then it is updated to its orthogonal projection on the boundary.
- If the point is within the feasible region then the algorithm defines a distance d that it should move along its orthogonal projection onto the boundary. If a distance of d would

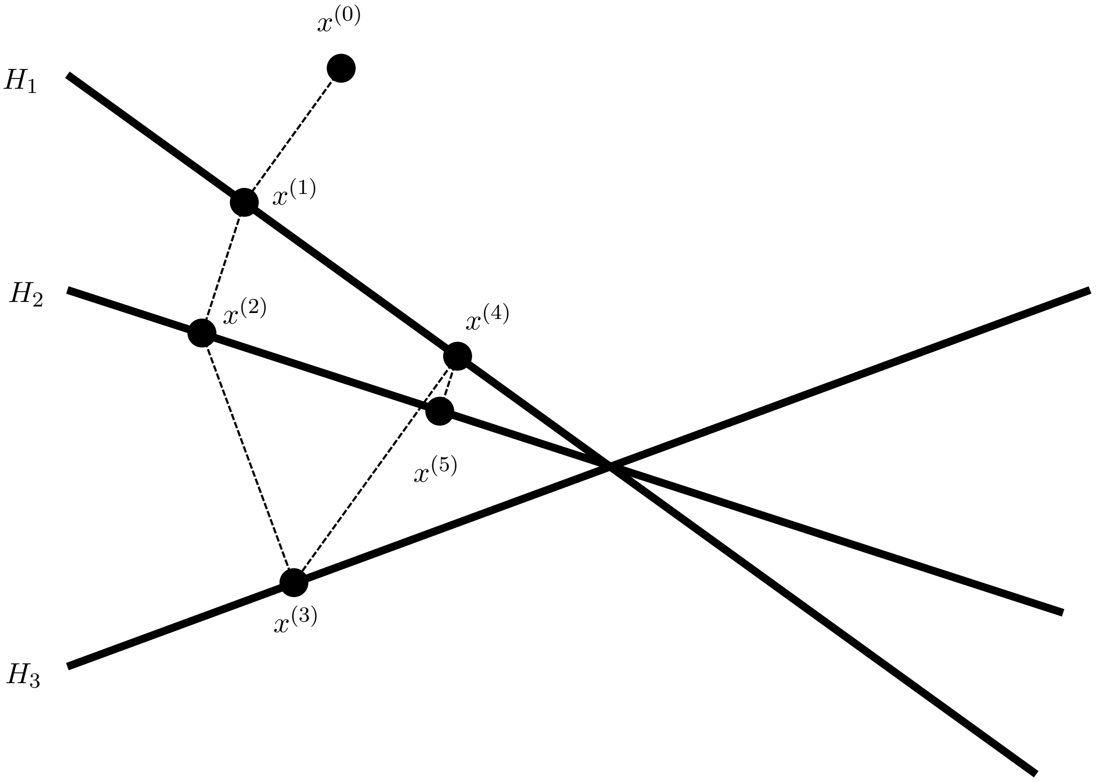

Figure 1: Example of Orthogonal Projection in Kaczmarz's Algorithm (Figure adapted from [3])

make the point invalid, then the movement is limited to keep the point on the boundary (point $x_2^{(k+1)}$ in the example), otherwise it lies distance d along the orthogonal projection direction (point $x_1^{(k+1)}$).

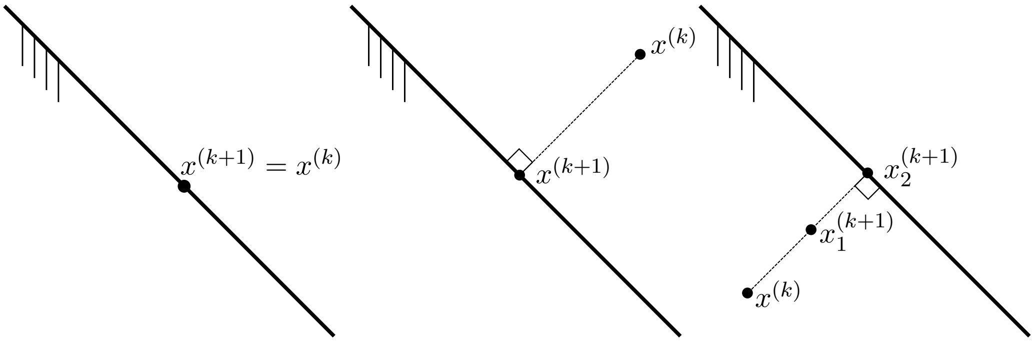

Figure 2: Three Possible Cases of Hildreth's Algorithm (Figure adapted from [9])

Formulation

The core algorithm presented by Hildreth solves the quadratic problem with n variables and m

constraints,

$$\min_y \frac{1}{2}\langle By, y\rangle + \langle y, d\rangle, \tag{1a}$$

$$\text{s.t. } Gy \leq h, \tag{1b}$$

where $y \in \mathbb{R}^n$, $d \in \mathbb{R}^n$, $B \in \mathbb{R}^{n\times n}$, $G \in \mathbb{R}^{m\times n}$, and $h \in \mathbb{R}^m$.

By defining,

$$B = D^T D, \tag{2a}$$

$$y = D^{-1}x - B^{-1}d, \tag{2b}$$

where 2a is a Cholesky decomposition, this can be reformulated as,

$$\min_x \frac{1}{2}\|x\|^2 \tag{3a}$$

$$\text{s.t. } Ax \leq b. \tag{3b}$$

Where $A = GD^{-1}$ and $b = h + GB^{-1}d$. In this formulation, $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, and $b \in \mathbb{R}^m$. As before, n is the number of decision variables, and m is the number of constraints.

In addition, the m-dimensional vector of dual variables, z, is defined. This is initialised to an arbitrary value within the non-negative orthant of $\mathbb{R}^m$ (ie, every entry is greater or equal to zero). Finally, the matrix $\Delta = AA^T \in \mathbb{R}^{m\times m}$ is defined.

Iteration

A single iteration performs the following operation on each index of z (subscripts indicate indexes, and superscripts in parenthesis indicate iteration numbers). Note that the update rule (4b) is not required within the iteration, therefore can be ignored until the final iteration.

$$z_i^{(k+1)} = \max\Big(0, z_i^{(k)} - \frac{1}{\Delta_{ii}}\big(b_i + \sum_{j=1}^{m} \Delta_{ij} z_j^{(k)}\big)\Big) \tag{4a}$$

$$x^{(k)} = -A^T z^{(k)} \tag{4b}$$

This expressions shows that Hildreth's algorithm meets the requirements for being considered a row action method.

- The only modification made on each iteration is to the vector of dual variables
- No matrix operations are performed (excluding the update rule)
- An iteration that updates variable i will only access row i of the problem matrix
- The iteration requires only information from the previous iteration, as well as the problem definition

2.2 Mesh Adaptive Direct Search

MADS is a blackbox optimisation algorithm, and is suitable in cases where analytical knowledge of the objective function and constraints are not available. MADS is also suited to non-smooth functions. The MADS algorithm [6] was first presented as an improvement on the General

Pattern Search (GPS) class of algorithms [10]. GPS is now defined as a special case of MADS. The algorithm has also been shown to be effective on noisy or expensive problems.

The chief of advantage of MADS is that it is a blackbox algorithm, therefore no analytical knowledge of objective function or the constraints is needed whatsoever. The objective function can be a black box that takes a point input and returns a cost, and the constraints can just return a yes/no answer for any trial point. This allows the algorithm to be used in many situations that more traditional optimisation algorithms that need analytical knowledge would not be able to solve.

MADS describes a general class of algorithm with several published variations, and the algorithm is able to be configured to use each variation interchangeably. The initial publication of MADS utilised a method of direction generation named Lower Triangular MADS (LTMADS), due to its use of semi-random lower triangular matrices [6]. A subsequent version named Orthogonal MADS (OrthoMADS) is an alternative that guarantees orthogonality in the generated directions, and is also deterministic [11]. Both LTMADS and OrthoMADS have been implemented in this project, and will be discussed in Sections 2.2.2 and 2.2.3 respectively.

The initial version of MADS implemented constraints by only considering points that are valid for each constraint, these are defined as extreme barrier constraints. Later versions presented a modified algorithm that uses relaxable constraints [12]. This version of the algorithm works to minimise both the objective function, but also the violation of the constraints. These two constraint formulations are discussed in Sections 2.2.4 and 2.2.5 respectively.

2.2.1 MADS Structure

Overview

Each iteration of MADS has three main steps, *search*, *poll*, and *update*. The first two steps are concerned with producing trial points that may offer an improvement over the current incumbent solution by decreasing cost (note that this is slightly complicated by progressive barrier constraints, discussed in Section 2.2.5). The final step will update the internal stage of the algorithm based on the outcome of the evaluation of the trial points. This structure is shown by Algorithm 1.

Mesh

The mesh is the core structure within MADS, for iteration k it is defined as the set,

$$M_k = \bigcup_{x \in S_k} \{x + \Delta_k^m D z : z \in \mathbb{N}^{n_D}\}, \tag{5}$$

where S_k is the set of points at which the objective function has been evaluated so far, D is a finite set of n_D directions ($D \subset \mathbb{R}^n$), Δ^m is a scalar named the mesh size parameter, and n_D is the number of columns in D (i.e. the number of unique directions in the set)[6]. M_k is not constructed by the algorithm, but MADS guarantees that all points generated will lie on the mesh, and therefore the definition of the mesh is able to be used in proofs of convergence.

It is simple to ensure that this requirement for all points to be on the mesh is met. All $x \in S_k$ by definition must belong on the mesh, Δ^m and D define the mesh, leaving the only requirement to be that z is an integer vector (making the newly generated point x plus a linear combination of the directions in D, and scaled by Δ^m).

Algorithm 1 MADS Algorithm High-Level Overview

```
  $\Omega$: The set of feasible points
  $f$: Objective function
  $k$: Iteration counter
  $x^{(k)}$: Incumbent point for iteration $k$
  $\Delta_k^m$: Mesh size parameter for iteration $k$
Require: Initial point $x^{(0)} \in \Omega$, $k = 1$, $\Delta_1^m = 1$
  repeat
    $T$: Set of trial points from search algorithm.
    $x^{(k)} = x^{(k-1)}$
    for $t$ in $T$ do
      if $t \in \Omega$ and $f(t) < f(x^{(k)})$ then
        $x^{(k)} = t$
      end if
    end for
    $D$: Set of trial directions from poll algorithm.
    for $d$ in $D$ do
      $p = x^{(k-1)} + \Delta_k^m d$
      if $p \in \Omega$ and $f(p) < f(x^{(k)})$ then
        $x^{(k)} = p$
      end if
    end for
    if $f(x^{(k)}) == f(x^{(k-1)})$ then
      $\Delta_{k+1}^m = \Delta_k^m \div 4$
    else if $f(x^{(k)}) < f(x^{(k-1)})$ and $\Delta_k^m \leq 0.25$ then
      $\Delta_{k+1}^m = \Delta_k^m \times 4$
    else
      $\Delta_{k+1}^m = \Delta_k^m$
    end if
  until a stopping condition is met
```

MADS defines an additional parameter, the poll size parameter $\Delta^p \in \mathbb{R}_+$. This defines the the magnitude of the maximum distance between the incumbent point and the new trial points.

The poll parameter is subject to a pair of conditions:

- $\Delta_k^m \leq \Delta_k^p \forall k$
- $\lim\limits_{k \in K} \Delta_k^m = 0$ if and only if $\lim\limits_{k \in K} \Delta_k^p = 0$ for any infinite subset of indicies K.

The maximum distance around the incumbent point defined by the poll size parameter is defined as the poll frame. This is illustrated in Figure 3. The bold lines represent the frame defined by Δ_k^p. The intersection of all other lines defines the current mesh (where the distance between points is given by Δ_k^m). As can be seen, the trial poll points p_i all lie on the mesh on points within the poll frame. This freedom in positioning within the frame allows for many directions to be explored.

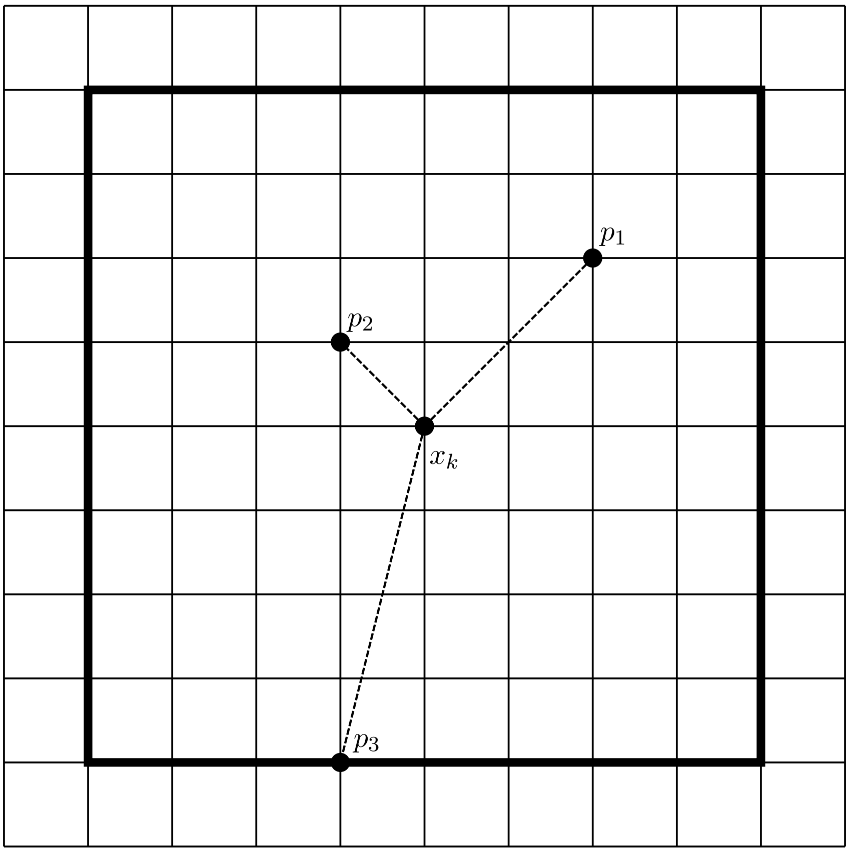

Figure 3: An illustration of poll points p_i in a frame around incumbent point x_k. In this case $\Delta_k^m = \frac{\Delta_k^p}{4}$ (Figure adapted from [6])

Search

The first step of the MADS algorithm is named the search step. This concerns the process of generating trial points and evaluating them. There is no requirements for how these are generated, apart from them being on the mesh.

This flexibility allows for the search stage to be tailored to the particular problem (e.g. exploiting a known structure of the problem), use a generic step (e.g. generating points along the direction that the previous iteration travelled), or ignoring the step completely.

The flexibility of this step is problematic for proving the convergence properties of MADS, however that is outside the scope of this report.

If an improved point is generated by the search step it is possible to skip the following poll step entirely, or continue to attempt to find a further improved point.

Poll

The poll step defines an exploration of the area directly surrounding the current incumbent point by generating a set of directions to investigate.

On iteration k the poll algorithm returns a set of poll directions D_k. A set of trial points, P_k, is generated from D_k with,

$$P_k = \left\{x + \Delta_k^m d : d \in D_k\right\} \subset M_k, \tag{6}$$

where the mesh size parameter Δ_k^m sets the distance between the current incumbent point and the trial point.

The poll step must meet a set of requirements to ensure that the trial directions will result in a point that is on the mesh, and that the generated directions will allow the algorithm to. This is the reason that an algorithm such as LTMADS must be used rather than a simpler technique (e.g. random direction sampling). Note that these other techniques may still be effective, it would just not fit the definition (and therefore convergence proofs) of MADS.

Update

The 'adaptive' term within the name MADS refers to the ability of the algorithm to change the resolution of the mesh by modifying the value of the mesh and poll size parameters. As these parameters define the distance between points on the mesh, it essentially controls how fine or coarse the mesh is. It is intuitive to see how a finer mesh would allow improvements to be found when the objective function is on a plateau (for example, when close to a local minimum), while a coarse mesh would allow for fast convergence (for example, when on a steep slope of the objective function. However if these situations were reversed, the algorithm would behave poorly (a small update on a steep slope would give slow convergence, and a large update on a plateau would be unlikely to find a minimum). The update to the mesh size algorithm to adapt to the objective function at the current point.

In general the update rule is relatively arbitrary (within the requirements previously outlined) and depends on the poll step of the algorithm. For both poll steps considered in this project (LTMADS and OrthoMADS) the update rule to the mesh size is the same. The *mesh index* variable, l_k, is defined, and initialised as $l_0 = 0$. This is then updated on each iteration as,

$$l_{k+1} = \begin{cases} l_k - 1 & \text{on a successful iteration,} \\ l_k + 1 & \text{on an unsuccessful iteration.} \end{cases} \tag{7}$$

The mesh index is then used to define the mesh and poll size parameters,

$$\Delta_k^m = \min\{1, 4^{-l_k}\} \quad \text{and} \quad \Delta_k^p = 2^{-l_k}. \tag{8}$$

Therefore, when a reduced cost point is found (a successful iteration) the value of Δ^m increases by a factor of 4 (up to a maximum of 1). This encourages fast convergence, as larger steps are taken while new points are being found. When unsuccessful (no improved points were found), Δ^m is reduced by a factor of 4. This increased resolution allows a nearer search space to the incumbent point, hopefully allowing further improvements. This is likewise the case with Δ^p, but with the variation being by a factor of 2, and no maximum size limit. This update rule can be seen at the bottom of Algorithm 1.

Note that a minimum size of the mesh is not limited by the algorithm, but is limited by the numerical precision of the platform the algorithms runs on.

2.2.2 LTMADS

LTMADS is the poll method proposed in the original paper for MADS [6]. This method utilises a semi-randomly generated lower triangular matrix to create a basis, from which poll directions are extracted.

LTMADS requires a direction to be maintained for each mesh size that will always be present whenever that mesh size is used. The generation of this direction is given in Algorithm 2. This procedure returns the vector b_l and the index, $\hat{i}_l$, that indicates the position of the absolute maximum entry in b_l.

Algorithm 2 LTMADS initial direction generation

```
$l$: The current mesh index
$n$: The problem dimension
$b_l$: The saved direction for index $l$
$\hat{i}_l$: The index of the absolute max value of $b_l$
if $b_l$ exists for $l$ then
    return $(b_l, \hat{i}_l)$
else
    Let $\hat{i}_l \in N$ where $N = \{1, \dots, n\}$
    Set $b_l[\hat{i}_l] = \pm 2^l$
    Set $b_l[i]$ to a value in $[-2^l + 1, 2^l - 1]$ for $i \in N \setminus \{\hat{i}_l\}$
    return $(b_l, \hat{i}_l)$
end if
```

From the vector b_l a positive basis of directions is generated with the procedure given in Algorithm 3.

This set of directions is then utilised to generate trial points with (6).

The randomness of the basis generation within LTMADS is both a benefit and a downside. Each time the algorithm is run, a different set of directions will be produced, and therefore a different path the optimum solution will be found. Depending on the directions that are generated the algorithm can converge very quickly, or can take a long time.

2.2.3 OrthoMADS

OrthoMADS was designed after LTMADS with the intention of providing a guarantee of orthogonality to the generated poll directions, as well as being deterministic in its results [11]. In many tests OrthoMADS has been demonstrated to provide superior results than the medium result of several LTMADS runs. OrthoMADS will also produce the same directions every time it is run, this allows only a single run to be performed, where previously LTMADS may need several to find a good minimum.

Algorithm 3 LTMADS Direction Basis Generation

```
  $l$: The current mesh index
  $n$: The problem dimension
  $b$: The saved direction for index $l$
  $\hat{i}$: The index of the absolute max value of $b_l$
Require: $b$ and $\hat{i}$ from Algorithm 2
  {Construct basis in $\mathbb{R}^{n-1}$}
  Let $L$ be an $(n-1) \times (n-1)$ LT matrix
  Set diagonal entries of $L$ to $\pm 2^l$
  Set each value in lower triangle $L$ to a random value from the range $[-2^l + 1, 2^l - 1]$
  {Construct basis in $\mathbb{R}^n$}
  Let matrix $B$ be formed from a random permutation of the rows of $L$
  Insert a row of 0s at index $\hat{i}$ of $B$
  Append $b$ as a column of $B$
  Randomly permute the columns of $B$
  {Completion to a positive base}
  if a minimal basis is desired ($n+1$ directions) then
    return $\begin{bmatrix} B & d \end{bmatrix}$ where $d_i = -\sum_{j \in n} B'_{ij}$
  else if a maximal basis is desired ($2n$ directions) then
    return $\begin{bmatrix} B & -B \end{bmatrix}$
  end if
```

It should be noted that LTMADS has the potential to outperform OrthoMADS due to its random selection of directions. However it can take many runs to achieve this superior result.

Again, this section will give a brief overview of the steps required to generate the poll directions used in OrthoMADS. See the OrthoMADS paper for a far more in-depth analysis [11].

Halton Sequences

OrthoMADS is based off the use of *Halton Sequences* [13]. These are a group of quasi-random vector sequences that have deterministic generation procedures.

OrthoMADS utilises the simplest Halton sequence. The t^{th} index of this sequence is described by the vector u_t,

$$u_t = \begin{bmatrix} u_{t,p_1} & \dots & u_{t,p_n} \end{bmatrix}^T, \quad u_{t,p} = \sum_{r=0}^{\infty} \frac{a_{t,r,p}}{p^{1+r}}, \tag{9}$$

where p_i is the ith prime number, and where $a_{t,r,p}$ contains the coefficients that define the base p expansion of t. t is the parameter that OrthoMADS uses to control the generated directions.

t is the Halton index (i.e. the index within the sequence), and the problem dimension, n, defines the maximum prime number, p_n, as well as the dimension of each Halton entry. Halton sequences show a degree of linear dependence between their first few entries, to avoid this the sequence is started with $t = p_n$, which is labelled the *Halton seed*, and denoted as t_0.

Adjusted Halton Sequence

On their own, Halton sequences do not meet the requirements for being poll directions (for instance, they are non-integer vectors). Therefore an adjustment step is performed to scale, translate, and round each u_t.

Each vector u_t is transformed to a corresponding function $q_t(\alpha)$,

$$q_t(\alpha) = \text{round}\Big(\alpha \frac{2u_t - e}{\|2u_t - e\|}\Big), \tag{10}$$

where $e \in \mathbb{R}^n$ has every element valued as 1. α is then selected as the scalar argument that satisfies,

$$\underset{\alpha \in \mathbb{R}_+}{\arg\max} \, \|q_t(\alpha)\| \tag{11a}$$

$$\text{s.t.} \quad \|q_t(\alpha)\| \leq 2^{\frac{|l|}{2}}. \tag{11b}$$

Orthogonal Integer Basis Construction

The final step in is constructing an orthogonal integer basis from the vector q_t. This is performed with the symmetric, scaled, Household transform [14].

$$H = \|q\|^2(I_n - 2vv^T), \quad v = \frac{q}{\|q\|}. \tag{12}$$

Where I_n is the identity matrix in $\mathbb{R}^n$.

Finally the basis is formed by setting,

$$D_k = \begin{bmatrix} H & -H \end{bmatrix}, \tag{13}$$

and trial points are formulated with (6).

2.2.4 Extreme Barrier Constraints

So far the discussion of the MADS algorithm only considered the problem,

$$\min_x f(x), \quad x \in \mathbb{R}^n. \tag{14}$$

However MADS is an algorithm designed for solving constrained optimisation problems. Therefore the objective function is redefined as,

$$f_\Omega(x) = \begin{cases} f(x) & x \in \Omega, \\ \infty & \text{otherwise,} \end{cases} \tag{15}$$

where Ω is the feasible subset of $\mathbb{R}^n$. This ensures that only feasible values of x are ever considered, as f_Ω will always evaluate to ∞ otherwise. The constraints that define the feasible set Ω are named *extreme barrier constraints* and the algorithm that uses them is named MADS Extreme Barrier (MADS-EB).

The effect of these constraints is that the algorithm will not even consider evaluating any points that are not in the feasible set, and will immediately discard them.

2.2.5 Progressive Barrier Constraints

The most recent aspect of the MADS algorithm to be considered in this project is the *progressive barrier constraint* formulation [12], or MADS Progressive Barrier (MADS-PB). This is an additional kind of constraint that allows infeasible points to be considered, but modifies the algorithm to select for new incumbent points that minimise the constraint violation.

There are multiple advantages to this, with some of the most interesting being that this method removes the requirement for having a feasible initial point, as well as potentially allowing for a more direct route to the feasible optimum, therefore needing less objective function evaluations.

This formulation requires an alteration to the definition of a constraint for MADS. The constraints may still be blackbox functions, but must now return a value indicating the amount a trial point violates the constraint. A trial point that violates the constraint should return a positive number, and vice versa.

As before, this problem considers the minimisation of the function,

$$\min_{x \in \Omega} f(x). \tag{16}$$

Subject to a set of J constraints and the feasible set Ω,

$$\Omega = \{x \in X : c_j(x) \leq 0, j \in J\}, \tag{17}$$

where $c_j(x)$ is the constraint violation function of the jth constraint, and $X \subset \mathbb{R}^n$ is the set of points defined by constraints that must always be satisfied. This makes Ω the set of points that satisfy every constraint. MADS-PB requires that $x \in \Omega$ is enforced only for the final considered point.

Overall Constraint Violation Function

All constraints are combined to form the overall constraint violation function,

$$h(x) = \begin{cases} \sum\limits_{j \in J} (\max(c_j(x), 0))^2 & x \in X, \\ \infty & \text{otherwise.} \end{cases} \tag{18}$$

This ensures that any points that do not satisfy any unrelaxable constraints evaluate to a cost of ∞, a point that satisfies every constraint (i.e. $x \in \Omega$) provides a cost of 0, and points that satisfy unrelaxable constraints but not relaxable ones provide a cost in the range $(0, \infty)$.

The barrier threshold $h_k^{\max}$ is now introduced. This variable sets a limit on the maximum value of $h(x)$, meaning that the total constraint violation amount can be controlled. The core of MADS-PB is that this threshold is reduced until a point is found that is both optimal and feasible.

Iteration Results

In previous algorithms the result of an iteration was either defined to be a success or a failure. MADS-PB complicates this slightly due to now attempting to select for points that are both feasible and an improvement. To this end, three iteration outcomes are defined: *dominating*, *improving*, and *unsuccessful*.

For the following description, we borrow the notation $y \prec x$ from [12] to indicate that either $h(y) < h(x)$ and $f(y) \leq f(x)$ or that $h(y) \leq h(x)$ and $f(y) < f(x)$.

- *Dominating* iterations are iterations in which a trial point, x_{k+1}, is found that has,

$$h(x_{k+1}) = 0 \quad \text{and} \quad f(x_{k+1}) < f(x_k), \tag{19}$$

  or has,

$$h(x_{k+1}) > 0 \quad \text{and} \quad x_{k+1} \prec x_k. \tag{20}$$

- *Improving* iterations do not satisfy the requirements of a dominating iteration, but do find an infeasible solution for which the constraint violation is reduced,

$$0 < h(x_{k+1}) < h(x_k) \quad \text{and} \quad f(x_{k+1}) > f(x_k). \tag{21}$$

- *Unsuccessful* iterations generate no points that are dominating or improving.

Ideally every generated point would be dominating, resulting in a decrease in either the objective cost or constraint violation while leaving the other constant (or also decreasing). Realistically this is not possible on every iteration. But by allowing the improving iterations as an outcome, the algorithm ensures that generated points will move towards being feasible at the cost of optimiality.

Update

As the update rules for the mesh and poll size parameters (8) depend on the outcome of the iteration (7), a modification to these update rules is required for MADS-PB. The mesh index is redefined as:

$$l_{k+1} = \begin{cases} l_k - 1 & \text{on a dominating iteration,} \\ l_k & \text{on an improving iteration,} \\ l_k + 1 & \text{on an unsuccessful iteration.} \end{cases} \tag{22}$$

Geometrically this may be interpreted as shrinking the mesh only when a dominating point is found, as the objective function is improving. As the improving iteration is, by definition, not moving in an improved direction on the objective function it is not desirable to shrink the mesh despite the iteration counting as a success.

Finally h^{max} also must be updated to continue convergence towards feasible solutions.

$$h_{k+1}^{max} = \begin{cases} \max\limits_{x \in V}\{h(x) : h(x) < h^I\} & \text{if the iteration is improving,} \\ h_k^I & \text{otherwise.} \end{cases} \tag{23}$$

Where V is the set of all considered points, and h^I is the selected new infeasible incumbent point. Therefore this update reduces h^{max} to ensure future results become closer to feasible. In the case that the iteration is improving (i.e. the new infeasible incumbent point increases the cost but improves feasibility) then a further reduced value of h^{max} is selected.

3 Background

3.1 Julia

Julia is a relatively new programming language, with development beginning in 2009 and the 1.0.0 version being released in 2018. The following subsection will discuss some of the language features that set Julia apart from its contemporaries, and the interesting design patterns that the language allows for. Further reading on using Julia in a high-performance computing role can be found in [1]. Julia's most notable features are multiple dispatch, a powerful type system, and high performance. Each of these features will be discussed, and it will be demonstrated why these make Julia suitable for this project.

3.1.1 Multiple Dispatch

Julia's main design paradigm is based on multiple-dispatch, but the language also takes aspects of procedural and functional languages into its design. Multiple dispatch is the ability for the language to have multiple definitions of the same function (with the individual implementations being named methods), and call the most appropriate for given set of arguments. This gives the ability to design software in a very generic way, with the flexibility to extend it later with new functionality for new data types without changing any of the existing codebase. No additional selection or parsing logic needs adding, as Julia itself is able to call the appropriate function.

A simple example of multiple dispatch is shown in Listing 1. Function `my_add_function` has been given three methods, differentiated by their argument types. The first takes two `Int`s and sums them. The second concatenates two `String`s (`*` is used for string concatenation in Julia). And the third takes a `String` and an `Int`, converts the `Int` to a string and concatenates them. Listing 2 shows the output of calling these functions. It is clear that a different function is called by providing different arguments.

```julia
function my_add_function(a::Int, b::Int)
    return a + b
end

function my_add_function(a::String, b::String)
    return a * b
end

function my_add_function(a::String, b::Int)
    return a * string(b)
end
```

Listing 1: Three implementations of the same function

```julia
my_add_function(1, 2)
> 3
my_add_function("hello ", "world")
> "hello world"
my_add_function("hello ", 2)
> "hello 2"
```

Listing 2: Output of the multiple dispatch example

In this project multiple dispatch is used in several different ways, with the main one being Application Programming Interface (API) design. Each package defines APIs using specified functions. Algorithms within the packages then implement the API via providing their own definitions of these functions. These are then automatically compatible with the rest of the package. This is used extensively, and is the main design pattern used throughout the project.

3.1.2 Type System

A type system is a programming language's way of recognising the internal representation of data within a program. C requires each variable to have its type explicitly defined whereas Python doesn't have a mechanism for specifying types, instead interpreting types from the code directly. In many ways Julia can operate like Python, with types of variables being interpreted. However, the type system of Julia is actually highly complex, and one of the most useful features of the language.

Julia defines its types in a tree, with each type inheriting from another. At the top of this tree is the `Any` type. A highly simplified diagram of this type hierarchy is shown in Figure 4. The `Any`, `Number`, and `Int` nodes are named abstract types, with the others being named concrete types. Abstract types cannot be directly instantiated (i.e. they do not describe a pattern of bits that represents data within Julia). However, they can be used as a 'catchall' description. For example, in Listing 1 the functions with `Int` arguments are able to accept variables of types `Int16`, `Int32`, or `Int64`. Likewise, a function could have an argument of type `Number`, and be valid for both `Float` and `Int` types.

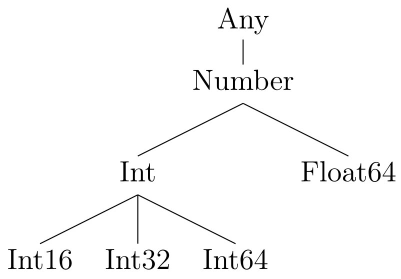

Figure 4: Simplified Type Hierarchy

Users of Julia are able to define their own types, which are treated with the same level of precedence as the default types of the language. This is crucial for using multiple dispatch within a codebase. A user can define an abstract type with the `abstract type` keyword, or a concrete type with the `struct` keyword. Example of custom types that define a tree are shown in Listing 3. The abstract type `Node` has three subtypes, `BinaryNode`, `NaryNode`, and `EndNode` (the syntax `<:` is used to denote a subtype). These types can now be used in exactly the same way as any default type in Julia. Note how the children of `BinaryNode` and `NaryNode` are of type `Node`. This allows any kind of node to be a child, and a new kind of node could easily be defined and used with no changes to the existing types.

```julia
abstract type Node end

struct BinaryNode <: Node
    left_child::Node
    right_child::Node
end
```

```julia

struct NaryNode <: Node
    children::Vector{Node}
end

struct EndNode <: Node
    val::Int
end
```

Listing 3: Custom types in Julia

As an example of using these types (and as another showcase of multiple dispatch) see Listing 4. The `TraverseTree` function has an implementation for each of the appropriate node types, and Julia is able to dispatch on the custom types just as it would with any default type.

```julia
function TraverseTree(t::BinaryNode)
    TraverseTree(t.left_child)
    TraverseTree(t.right_child)
end

function TraverseTree(t::NaryNode)
    map(TraverseTree, t.children)
end

function TraverseTree(t::EndNode)
    println(t.val)
end
```

Listing 4: Usage of custom types

As is shown in `BinaryNode`, a custom type is able to contain more than a single value. This is used extensively in both RAM and DS, which both use a single type to define the current state of an optimisation problem. Listing 5 details a moderately sized example from the DirectSearch.jl codebase.

This example shows how the state of an OrthoMADS process is stored. This example also shows two other important concepts, constructors and parametric types. The function that is shown inside this struct is known as its constructor, this allows operations to be done on input data, and default values used. For example, in this case, a call to `OrthoMADS` need only specify the dimension of the problem (`N`), and the constructor sets all the other required variables.

The `{T}` syntax shown here shows that `OrthoMADS` is a parametric type. This means that the types noted with `T` have a variable type that is decided when the struct is instantiated. For example, specifying `OrthoMADS{Float32,Int32}(4)` will describe a 4 dimensional OrthoMADS instance that uses 32-bit numbers internally. An additional constructor function above the main is used to give a default value of `Float64` and `Int64` for the parametric types, as this would be the desired value for the majority of users. The parametric nature is included as it will make it simpler to adapt this code to run in a more specialised environment, for example as embedded code running on a microcontroller.

```julia
mutable struct OrthoMADS{T,F} <: AbstractPoll
    l::F
    Δᵖmin::T
    t₀::F
    t::F
    tmax::F
    OrthoMADS(N::Int) = OrthoMADS{Float64,Int64}(N)
    function OrthoMADS{T,F}(N::F) where T,F
        M = new()
        #Initialise as the Nth prime
        M.tmax = M.t = M.t₀ = prime(N)
        M.l = 0
        M.Δᵖmin = 1.0
        return M
    end
end
```

Listing 5: Composite type that describes OrthoMADS

3.1.3 Performance of Julia

While the performance of a programming language will vary depending on the usecase, it is simple to show that Julia is able to provide performance that is, in many cases, comparable to C. [2] provides a dataset of microbenchmarks provided by the Julia developers that test a variety of common mathematical operations.

Note that these performance results date from 2018, and as such are not strictly representative of current performance numbers of Julia and the other included languages. However the overall trend is still correct as of the writing of this report.

An interesting difference between Julia and other high-level languages is the use of code vectorisation. Languages such as Python are slow at doing operations themselves, and use precompiled high performance code for intensive operations (for example, NumPy for Python). Vectorisation is the process of formulating a problem as an input to these high performance routines.

Julia is a compiled language, unlike Python, meaning that the vectorised and unvectorised versions of code eventually compile to the same set of instructions. While, in some cases, vectorised code is easier to read, it is not a requirement for fast code. This is a huge benefit as many operations do not vectorise well. In most cases, Julia is able to run at the same speed whether using vectorised or unvectorised code [1].

3.2 Optimisation in Julia

The Julia community has multiple organisations that are based around the development of optimisation libraries, solvers, and utilities. The largest two of these are JuliaOpt [15], and JuliaSmoothOptimizers [16], with several smaller contributors that focus on more specific areas (e.g. the package BlackBoxOptim.jl focuses solely on derivative free optimisation). JuliaOpt contains a large collection of packages, most of which are interfaces to existing solvers (such as Gurobi, or GLPK), as well as the utility packages JuMP [17][18] and *MathOptInterface.jl* (MOI) [19]. These last two packages are most relevant to this project.

3.2.1 MathOptInterface

MOI is a Julia package that defines a standard interface to communicate with optimisation software packages. In addition it has the ability to identify the structure of a problem, the structures supported by a solver, and reformulate problems to be compatible. Discussions on our implementation of an MOI wrapper can be found in Section 5.6.

It is possible to define optimisation problems with the syntax offered by MOI, Listing 6 shows an MOI definition of the objective function of the problem,

$$\begin{aligned} \max_{x,y} 2x + 7y&, \\ -5 \leq x \leq 2&, \\ y \leq 30&, \\ 2x + 8y \geq 3&. \end{aligned} \tag{24}$$

```
x = MOI.VariableIndex(1)
y = MOI.VariableIndex(2)
a = MOI.ScalarAffineVariable(2, VariableIndex(1))
b = MOI.ScalarAffineVariable(7, VariableIndex(2))
f = MOI.ScalarAffineFunction([a,b], 0)
```

Listing 6: MOI definition of an optimisation problem

It is clear that the MOI interface is very verbose, declaring a new variable and type for each aspect of the function. This approach is necessary to ensure a well defined internal structure for the package, however it results in a unwieldy and complex system for manual use.

3.2.2 JuMP

Rather than interface with MOI directly, the JuMP package can be used to interface with MOI, which then translates the problem into a format used by solvers. JuMP offers expressions that operate at a much higher level of abstraction than MOI, giving the freedom to input expressions directly.

The entire problem shown in (24) can be formulated, solved, and queried in JuMP as shown in Listing 7 (using the GLPK solver for Julia [20]). What took five lines in MOI is accomplished in one with JuMP, line 9 of the listing. This expression is directly translated into the MOI expression, but in a layer hidden from the user.

```
1  using GLPK
2  using JuMP
3
4  p = Model(GLPK.Optimizer)
5
6  @variable(p, x)
7  @variable(p, y)
8
9  @objective(p, Max, 2x + 7y)
10
11 @constraint(p, -5 ≤ x ≤ 2)
12 @constraint(p, y ≤ 30)
13 @constraint(p, 2x + 8y ≥ 3)
14
15 optimize!(p)
16 @show value(x) # = 2.0
17 @show value(y) # = 30.0
```

Listing 7: Optimisation problem definition in JuMP

This arrangement leads to a very flexible system, with a problem defined in JuMP easily being applied to multiple solvers without even needing to be re-entered into Julia.

The RAM package has been given an MOI interface, meaning that the solver be used like any other optimisation package in Julia. DS has not been given an interface to MOI, as it is intended for use with blackbox functions which are not able to be expressed with MOI or JuMP.

3.3 Existing Algorithm Implementations

3.3.1 Row Action Methods and Hildreth's Algorithm

For row action methods it is commonly the case that algorithms are put forward in papers, but software resources are not also provided. Or if they are, they are not in a form that is easily utilised as a standalone software package.

There are exceptions to this, for example, several algorithms in the AIR Tools II MATLAB package [21] meet the requirements for classification as row action methods.

Hildreth's algorithm, as far as is clear, is not available in any software packages, and there does appear to be any row action methods available for Julia.

3.3.2 Mesh Adaptive Direct Search

MADS and its associated sub-algorithms (MADS-PB, OrthoMADS, etc.) are all offered within the NOMAD software package from the authors of MADS [22]. This software will be used as a reference implementation for analysing the performance of the DirectSearch.jl package.

The reason that creating the DS package is not just replicating work that has been done in the creation of NOMAD is that NOMAD is designed as a standalone piece of software and is not easy to modify. The case for the development of DirectSearch.jl is that it is easy to understand the design of the software, and to make modifications and extensions that fit a target problem.

4 Requirements

4.1 Initial Project Specification

After the commencement of the project, a meeting was held where the exact focus of the project was discussed, with the initial aim being to select a set of solvers to implement in Julia. It was important to choose solvers that were representative of a broad range of algorithm types, but also to not replicate any work that had already been done to implement solvers in native Julia.

In order to showcase derivative-free, first-order, and second-order solvers the decision was made to implement the MADS [6], OSQP [23], and PIPS-NLP [24] algorithms, respectively. In addition, an implementation of Hildreth's Algorithm [8] would be included to provide an implementation of row-action method algorithms.

During the implementation of Hildreth's algorithm it was remarked that the design of the package would be very easy to generalise to an entire category of solvers, with Julia's multiple dispatch paradigm allowing new row action methods to be easily added. As such, we decided to transition the aim of this sub-project to the design a framework in which any row-action method can easily be implemented. It was also discussed whether this idea could also be applied to the MADS algorithm sub-project, by implementing a framework for direct-search methods rather than just the MADS algorithm itself.

The increase in scope of these two packages decreased the time available to focus on the OSQP and PIPS-NLP algorithms. It was decided to remove these from the project aims, and focus on developing the row action methods and direct search packages to be as high quality as possible.

This gives the final high-level project specification: The development of two optimisation packages in Julia, one that implements a framework for the development of row action method algorithms, and the other a framework for the development of direct search methods. The highest priority of these packages is to facilitate simple algorithm modification, with a secondary priority being to ensure the code is high performance.

4.2 Shared Requirements

Each of the packages share a set of requirements that will ensure they are developed to a high standard, and are able to provide a useful utility.

4.2.1 Code Style

Ensuring a consistent naming convention and use of programming idioms helps to make code understandable. For the most part the project follows the Julia style guide [25]. The recommendations in this document allow for both a programming style in common with the majority of the Julia codebase, but also helps to avoid pitfalls that lead to overly complex/unmaintainable code, or performance problems.

An exception to this is when working with external package that do not follow these guidelines. For example, it is common in Julia to append a ! symbol to the name of functions that modify their arguments. However, MOI (Section 3.2.1) forgoes this convention due to the majority of its functions modifying the arguments. As RAM makes use of MOI extensively, this convention was adopted.

4.2.2 Documentation

Documentation is crucial for both the user and maintainer of a package to understand how the software works. Julia has a flexible documentation system in the form of ‘docstrings’. These define a format of strings that are placed above the definition of a function or type and is taken as documentation of that object.

Listing 8 demonstrates the formatting of a docstring. Docstrings support markdown formatting, as well as LaTeX equation formatting. This allows for usage examples to be included within the documentation.

```julia
"""
    BumpIterationLimit(p::DSProblem, val::Int=100)

Increase the iteration limit by `i`.
"""
function BumpIterationLimit(p::DSProblem; i::Int=100)
    p.iteration_limit += i
end
```

Listing 8: A docstring in Julia

When queried within Julia, the docstring is immediately formatted and printed to the command line (Listing 9). This shows the utility of docstrings, immediately being printed when a user needs them.

```
help?> BumpIterationLimit
search: BumpIterationLimit

  BumpIterationLimit(p::DSProblem, val::Int=100)


  Increase the iteration limit by i.
```

Listing 9: A formatted docstring in the Julia command line

It is also possible to create longer form documentation for a Julia package using the Documenter.jl package [26]. This package provides a mechanism for standalone documentation pages to be written (for example, as a guide on how to use the package), and also have pages that automatically import documentation from docstrings.

This solution allows for accurate detailed information on each function to be very easily incorporated with higher level descriptions of a package. For this reason, having detailed documentation for the packages is a high requirement of this project.

4.2.3 Testing

Testing to ensure that the code actually performs its intended function is just as important as the functionality of the code itself. As with documentation, Julia contains powerful testing utilities that make it simple to create tests alongside source code.

A unit test is defined as a set of tests that validate that a ‘unit’ of software is behaving as intended. In most cases this unit maps to a single function within the codebase. Each test within the unit test is typically small, testing the result of a function call against a known correct solution. For good unit testing, several inputs that cover the main operation range of the function should be given, as well as several tests on the edge-cases (e.g. if a function is valid for only positive integers then it should be tested for a zero-valued input). In addition, testing to ensure that the function fails correctly for invalid inputs is also useful (e.g. providing a negative input to the previous example).

Julia's system for unit testing consists of three macros, `@test`, `@testset`, and `@test_throws`. `@test` evaluates an equality test that follows it, and accumulates all such tests into a report. `@testset` defines a group of `@tests` and other `@testsets` with a common name, and is useful for organising tests. `@test_throws` is used to check that a function fails in a defined way, but otherwise behave identically to `@test`.

```julia
@testset "SetInitialPoint" begin
    p = DSProblem(2)
    @test_throws ErrorException SetInitialPoint(p, [1.0, 1.0, 1.0])
    SetInitialPoint(p, [5.0, 5.0])
    @test p.user_initial_point == [5.0, 5.0]
end
```

Listing 10: A unit test in Julia

An example of a small unit test is given in Listing 10. This tests the behaviour of the `SetInitialPoint` function from the DS package. Firstly setup is done, defining the problem as a two dimensional direct search problem. It is then verified that an error is thrown when trying to set an initial point of a different dimension to the problem. A valid initial point is then set, and it is verified that the corresponding action the function implements has been performed (that being setting the `user_initial_point` value within the direct search problem. Note that this is a subset of the tests actually done on this function in DS.

As a complex piece of software can have many errors introduced when being implemented, it is critical that these tests are included to verify the software behaves as intended. In addition, it is necessary to test the behaviour of algorithms to ensure the understood behaviour is the same as that defined by the algorithm designers.

For the implementation of MADS, the paper authors have included many examples of the output of their algorithms, this allows simple testing where the outputs are compared for a given input configuration. An example of these results can be seen in *Example 4.1* of [6].

This is more complex for RAM as the same kinds of examples are not given. A solution to this is to implement the algorithms separately using a different method to the package's implementation and and record the outputs for a wide set of inputs. The package can then be checked against these results. This does not protect against a fundamental misunderstanding of the algorithm, but does test against mistakes in implementation. Separate tests can then be made on the performance of the algorithm to ensure it converges to a correct value in an expected number of iterations to confirm that the understood algorithm is behaving as intended.

Testing on the behaviour of algorithms (e.g. number of iterations to converge on a known problem) is also useful to ensure that a change to the code has not introduced problems that are not shown by other tests. This can be verified by recording the state following a run of a trusted implementation of the algorithm, and comparing it to the state following new implementations.

4.2.4 Continuous Integration

Another aspect of software development that integrates well with documentation and testing is continuous integration. At its core, continuous integration is the process of running scripts to automate tasks whenever new code is added to the repository.

The main use of this is to run all the defined tests on new code additions. While the developer should have run the tests before adding code, the continuous integration allows for the same tests to be run in a variety of environments. For example, multiple versions of software (e.g. Julia versions 1.2, 1.3, and 1.4 are all in common use while producing this project), and on multiple operating systems (it is relatively common to find that code that runs well on Linux or MacOS has problems on Windows, and vice versa).

Continuous integration also allows for the automation of generating and publishing documentation. The technique in common use within Julia is to define the contents of the documentation with markdown files. These are then automatically rendered by the continuous integration service when added to the project and published to a documentation website.

For this project, Travis [27] is used for continuous integration, GitHub [28] is used for code hosting, and GitHub Pages [29] is used for documentation deployment.

4.3 Requirements of RowActionMethods.jl

The design and structure of the package should make it simple to add new row action algorithms. This requires an API that allows for many different kinds of algorithms to be added and work well within the package, without adding performance problems.

The kinds of problems required for the example algorithm, Hildreth's algorithm, makes use of a quadratic objective function and linear constraints. Therefore the software must be able to represent these kinds of problems, and be designed in such a way that it is simple to add other kinds of objective functions and constraint types in future.

As discussed in the overview of row action methods and Hildreth's algorithn (Section 2.1), the main advantages of these methods are their application to sparse and large problems. Therefore the package should take advantage of this structure by using appropriate sparse data storage types.

The package should be able to offer performance comparable to that of other quadratic solvers, such as OSQP [23], and superior performance to more generic nonlinear solvers such as IPOPT [30].

Finally, one of the advantages of the row action methods is the ability to compute them in a very parallel manner (as the calculations in each iteration can be made independent of each other). Theoretically this allows an N-fold increase in the number of processors to scale to N-times the performance (although this is limited by the number of rows in the problem matrices). An implementation of this package should allow this kind of scaling to be observed, with the understanding that a certain penalty will be paid for the overhead in distributing and collecting these operations.

4.4 Requirements of DirectSearch.jl

The package should implement the core MADS algorithms: MADS-EB, MADS-PB, LTMADS, and OrthoMADS.

The package should require a similar number of objective function evaluations as NOMAD. As it is expected that the target problems will be far more costly than MADS itself, this ensures that no extra wasteful computations are performed.

The package should be able to offer performance similar to that of NOMAD (when NOMAD is configured to use the same algorithms as DS). This ensures that no additional overhead is being introduced.

Rather than swapping out entire algorithms, as with RAM, the modularity of the MADS family allows for different parts to be combined to define an algorithm. Therefore the package should be designed in such a way that it is simple to design and integrate different parts of the MADS algorithm.

Finally, the package should be designed such that it is simple to define a new sub-algorithm that can be used with the existing algorithms of the package. This extends to poll steps, search steps, and constraints.

5 Design - RowActionMethods.jl

This section will cover the design decisions made in the development of the RowActionMethods.jl package, as well as details of its implementation.

Most of the design decisions discussed in this section are based off prototyping and experimenting within testbenches based on Jupyter notebooks. This method allowed for rapidly iterating over designs, maintaining multiple designs for cross-comparison, and benchmarking the performance of each. Selected designs were chosen due to having the best performance for the given requirements, although in some cases performance has been sacrificed to improve usability.

5.1 Structure

A constant factor during the design of this package is the balance of control between the package itself, and algorithm designers. Giving full control to the algorithm designer essentially removes any complex functionality from the package, and would result in additional algorithms differing greatly in their internal designs. On the other hand, being overly restrictive disallows flexibility in algorithm design and potentially enforces poor performance across the board.

To resolve this, the package has its own definitions for the objective function and constraints that algorithm designers must use. This ensures that the package is able to make assumptions about the data that the algorithms expect to receive and optimise accordingly. These data types are designed to be relatively simple, have efficient functions to access them, and precalculate many actions for the user. The objective function and constraints are discussed in Sections 5.3 and 5.4 respectively.

To balance this aspect of prescriptive design, the users are given a simple API, and the freedom to define their own internal representation of the problem. Experimentation with more complicated designs showed that the API became overly complex and introduced significant bottlenecks. For example, for a time it was designed such that a user would ‘register’ their problem matrices with the package (which would itself store the matrix in a sparse format), and the user would store a reference to this matrix. However, this design resulted in the user having to store and manage the same number of variables, as well as a more complex system for accessing and updating variables, and gave no performance benefits. The only advantage was that it attempted to enforce storage in a sparse format. As users should be able to judge if a variable is best stored in a sparse format or not, this design was dropped and the simpler construction was adopted.

5.2 Problem Construction

The core of the package is the `RAMProblem` type. This contains all the information that defines the problem, including the description of the problem, and the solver. This custom type is shown in Listing 11.

The struct is parameterised with the `T` and `F` types that fulfil the role of float and integer types respectively. This is done to allow for this package to easily be adapted to an environment that does not contain the standard `Float` and `Int` types. By default, this type is constructed with `Float64` and `Int64`, as this should be suitable in the majority of computing environments the package is used in.

```julia
mutable struct RAMProblem{T,F}
    #== Variables ==#
    variable_count::F

    #== Constraints ==#
    constraints::Constraints{T,F}

    #== Problem Description ==#
    objective::AbstractObjective
    result::Union{SparseVector{T},Nothing}

    #== General ==#
    status::AbstractStatus
    iterations::F
    method::ModelFormulation

    #== Threading ==#
    threads::Bool

    statistics::Statistics{T}
end
```

Listing 11: `RAMProblem` struct

`variable_count` is an integer number that records the number of variables the current problem has. The `constraints` and `objective` variables are covered in their own sections.

`result` will store the result of the algorithm as soon as it is calculated. This was done so that the package is able to store a solution once it is found, and not rely on the internal stage of any of the algorithms.

The `status` entry is of type `AbstractStatus`. Subtypes of this correspond to a particular outcome or state that the solver can be in. The default statuses are given in listing 12. The initial type is set as `OPTIMIZE_NOT_CALLED`, with this updated whenever iteration is stopped with the most appropriate entry. Custom statuses can easily be defined and set by custom stopping conditions (Section 5.7).

```
abstract type AbstractStatus end

struct OPTIMIZE_NOT_CALLED              <: AbstractStatus end
struct OPTIMAL                          <: AbstractStatus end
struct INFEASIBLE                       <: AbstractStatus end
struct ITERATION_LIMIT                  <: AbstractStatus end
struct TIME_LIMIT                       <: AbstractStatus end
struct UNKNOWN_TERMINATION_CONDITION    <: AbstractStatus end
```

Listing 12: Status types

`iterations` records the number of iterations that have been performed of the current algorithm, and `method` contains the algorithm's struct itself.

The final entries are the boolean `threads`, which will distribute iterations across multiple processors, and `statistics` which records timing information for various stages of the algorithm.

The final entry in `RAMProblem` is the `statistics` entry. This was added during testing as an internal tracker for useful information. At the current stage of the project it consists of a struct that lists runtimes for parts of the algorithm. However this will be easy to expand with more information in future if required.

5.3 Objective

While row action methods are applicable to multiple kinds of objective functions, the target algorithm for this project, Hildreth's method, makes use of quadratic objective functions, therefore this is the kind of objective function that has been implemented. However Julia's type system and multiple dispatch means that the additional of other objective functions in future is simple. Listing 11 shows that the objective is of type `AbstractObjective` this is an abstract type that all other objectives must be a child of .

5.3.1 Objective Definition

A problem with n dimensions has a quadratic objective function in the form,

$$\frac{1}{2}\langle Qy, y\rangle + \langle y, F\rangle, \tag{25}$$

where Q is an $n \times n$ matrix and F and y are n dimensional vectors. This requires storing the matrix Q and vector F. This was initially implemented in the form shown by Listing 13. Note that the entries are given a sparse format due to this package being aimed at problems characterised by a high degree of sparsity.

```julia
struct SparseQuadraticObjective{T} <: AbstractObjective
    Q::SparseMatrixCSC{T}
    F::SparseVector{T}
end
```

Listing 13: Original Quadratic Objective Type

It was then noted that the matrix Q was most often needed when solving systems of linear equations in the form $Qa = b$. The default methods Julia uses to solve these systems will calculate an LU or Cholesky factorisation of the matrix. To avoid repeated calculation of the matrix inverse, it is logical to precompute this value and store it within the objective. Therefore the type was updated to the form shown in Listing 14.

```julia
struct SparseQuadraticObjective{T} <: AbstractObjective
    Q::SparseMatrixCSC{T}
    Qf::Cholesky{T}
    F::SparseVector{T}
end
```

Listing 14: Updated Quadratic Objective Type

Unfortunately there is a type conflict between the linear algebra package (the package that solves the system of equations) and SuiteSparse (the package that defines the sparse matrices). If SuiteSparse is used to perform the factorisation (with the input and output both being sparse matrices), the resultant type is incompatible with the generic linear algebra operations. The alternative is used here, where the cholesky factorisation from the linear algebra package is used, but does not use sparse storage.

There are a pair of downsides to this. Firstly, the storage of the factorisation will require more memory due to the dense storage. Secondarily, any calculations that may be able to take advantage of a sparse objective factorisation is forced to use the dense matrix, resulting in potentially slower computations.

5.3.2 Objective Accessor Functions

It is possible to access the objective values directly given a reference to the `RAMProblem` they are contained within. However, this introduces multiple design dependencies within the package. For instance, if the name of a variable within an objective type is changed then anywhere this value is accessed must also be updated. The alternative is to create a set of accessor (or getter) functions that return the required information. This ensures that only these functions need to be maintained, rather than multiple other references from elsewhere in the package.

For the objective function a pair of accessor functions were defined: `GetObjective` and `GetObjectiveFactorised`. Respectively these return the tuples `(Q, F)` and `(Qf, F)`.

This accessor function pattern is repeated throughout the project for the majority of commonly needed values within algorithm/problem description structs.

5.4 Constraints

A problem of dimension n and with m linear constraints defines the constraints as,

$$Gy \leq h, \tag{26}$$

where G is an $m \times n$ matrix, and y and h are n and m dimensional vectors respectively. While the use of constraints depends on the algorithm in question, the package assumes that the matrix G will be both large and sparse.

5.4.1 Constraint Storage

The `Constraints` type, Listing 15, stores the constraint of the problem. The `Functions` and `Limits` entries store the matrix G and the vector h respectively.

```
mutable struct Constraints{T,F}
    Functions::SparseMatrixCSC{T}
    Limits::SparseVector{T}

    #Maps constraint index to actual vector index
    constraint_indexes::Dict{F, F}

    #Tracks largest constraint to ensure a unique new index
    max_constraint_index::F
    #Track number of constraints
    constraint_count::F

    Constraints() = Constraints{Float64,Int64}()
    function Constraints{T,F}() where {T,F}
        c = new()
        c.constraint_indexes = Dict{F,F}()
        c.max_constraint_index = 0
        c.constraint_count = 0
        return c
    end
end
```

Listing 15: `Constraints` Type

To define the constraints, the `AddConstraint` function is used, Listing 16. If no constraints are currently present then the function will initialise the sparse matrix and vector. Otherwise it will define a new array that is the concatenation of the existing constraints and the new ones (as Julia does not support resizing matrices in-place).

```julia
function AddConstraint(model::RAMProblem{T}, M_row::SparseVector{T}, lim::T
                      )::Int where T
    c = model.constraints

    if c.constraint_count == 0
        c.Functions = M_row
        c.Limits = sparse([lim])
    else
        c.Functions = hcat(c.Functions, M_row)
        c.Limits = vcat(c.Limits, lim)
    end

    c.constraint_count += 1

    #Ensures unique constraint index
    new_index = c.max_constraint_index + 1

    #Update largest index
    c.constraint_indexes[new_index] = c.constraint_count

    return new_index
end
```

Listing 16: `AddConstraint` function

It should be noted that the constraints of a problem are defined as rows of the matrix G. This is slightly problematic, as Julia stores matrices with column based ordering (meaning that consecutive values in a column are next to each other in memory, but consecutive values in rows are not). This results in appending or accessing columns of a matrix being far faster than rows. This also extends to sparse data storage.

Therefore the `Functions` entry in the constraints struct is actually the transpose of the matrix G. This is documented in both the struct itself, as well as by the accessor functions.

5.4.2 Constraint Modification

The additional variables within `Constraints` and the additional functionality within `AddConstraint` is related to being able to remove constraints that have already been added. This functionality is in a prototype stage at the time of submission, but has heavily influenced the design of the constraints and therefore is still included.

A unique index is generated for each constraint that is added. For simplicity this is just set at the absolute number of constraints that have ever been added, meaning there is no possibility of repeated indexes. This value is tracked by the `max_constraint_index`. The dictionary `constraint_indexes` is then used to map between these unique indexes and the row index within the constraint matrix/vector.

If a constraint is removed (and the matrix/vector are changed accordingly) this mapping is no longer valid, but can be easily updated to reflect the new position of constraints.

A disadvantage of this kind of constraint modification is that it will require rebuilding the problem model, which is a very expensive operation. Another addition that could be done is to create an API for in-place modification of problem constraints where rebuilding is not required.

This operation would differ between algorithms, but has the potential to be quite efficient. For example, the constraint limit for the dual formulation of Hildreth's algorithm is defined as $b = h + GB^{-1}d$, where h is the constraint limit of the primal formulation, with this being the only point that h is used. Therefore a modification to the primal constraints could be very efficiently copied to the dual formulation, requiring only a vector addition.

These kinds of constraint modifications were unfortunately not included in RAM at the time of submission, but it is hoped that this is a feature that can be added at a later date.

5.4.3 Constraint Accessor Functions

As with the objective function, it is necessary to access the constraints in a safe manner by using accessor, or getter, functions. Two of these are defined for the constraints: `GetConstraintMatrixTransposed` and `GetConstraintVector`. The names of the functions should imply their individual use.

It was a deliberate decision to not supply a function that transposes the constraint function back to its 'traditional' form. It was felt like this may give the impression to a user that it is an efficient function to call, when in reality it is just performing its own transposition. With only the transposed function, it is very clear to the user that a transposition is necessary to get the original data.

It is possible in future that a non-transposed constraint matrix and corresponding function may be added, but this does come at the cost of memory.

5.5 Model Formulation

The model formulation is the area of the package design that is the most 'open' in implementation. This area is where the algorithm designer is free to take the APIs implemented in the package to form their algorithm. The main two APIs they have access to are the objective and constraint APIs, discussed previously. The designer must also themselves implement the Model API, discussed in the next subsection.

The model formulation is defined by a struct that is a subtype of the `ModelFormulation` abstract type and stored in the `method` entry of the problem definition (Listing 11). This should include all variables required for the algorithm to function. As an example, Listing 17 shows the model formulation for Hildreth's method.

```julia
mutable struct Hildreth{T} <: ModelFormulation
    A::SparseMatrixCSC{T}
    b::SparseVector{T}

    Δ::SparseMatrixCSC{T}
    n::Int

    z::SparseVector{T}
    x::SparseVector{T}

    user_initial_point::Union{Nothing,Vector{T}}

    function Hildreth{T}(;kwargs...) where T
        m = new()
        m.user_initial_point =
            haskey(kwargs, :initial) ? kwargs[:initial] : nothing
        return m
    end
end
```

Listing 17: Hildreth's Method Model Formulation

The user is able to define custom arguments to configure a model and pass them in as keyword arguments when defining the problem. For the `Hildreth` type, the constructor can be passed the argument `initial` to set the `user_initial_point` variable. This is useful when the method has different possible configuration options, as further options can easily be added that are specific to the solver.

5.5.1 Model API

With the aim of providing a flexible structure for algorithms to be developed in, the necessary API for a designer to implement is deliberately minimal. Apart from the model formulation struct, there are three kinds of functions that define the API: configuration, necessary, and optional.

Configuration Functions

The configuration functions are very small, consisting of returning a single type or value that will indicate to the rest of the package what the requirements of the algorithm are. Of these, the only one that must be implemented is the `ObjectiveType` function, which indicates the kinds of problems that the algorithm can solve. Other configuration options have defaults implemented such that multiple dispatch will cause the default to be called if the solver doesn't implement its own version. An example would be the `SupportsDeleteConstraint` function.

Necessary Functions

There are four necessary functions that are instrumental to the use of an algorithm, and will be discussed in the order that they are called.

`Build` is called when the package believes the objective and all constraints have been defined, and is intended for the algorithm to build its internal representation from the objective and constraint matrices.

`Iterate` implements the main iteration routine of the algorithm. As this is the most frequently called function in the entire package it should be implemented to be as efficient as possible. To this end, it should avoid defining any new variables (as allocating memory is a major performance bottlenecks in Julia). If working variables are needed for this stage they should be preallocated during the build stage within the model formulation.

`Resolve` is called after optimisation has finished, and is useful for calculating primal results from a dual representation that may have been used during solving.

Finally `GetVariables` is used whenever the resultant optimal point is queried.

It is possible that an algorithm may not have a need for one of these steps (e.g. no dual formulation is used, meaning that the resolve step isn't required). In this case, the necessary functions must still be implemented, but can be left empty.

Optional Functions

The final kind of function within the API are the optional functions. These are included depending on the features that the algorithm supports. For example, if an algorithm is able to support the deletion of a constraint (and they indicate this with the corresponding configuration option) then they must also implement the `DeleteConstraint` function to recalculate the internal variables that would be affected by the removal of a constraint.

Other kinds of functions that fit this category are the functions responsible for multithreaded operation. By default the package will assume an algorithm does not support multithreading, but implementing the appropriate optional functions allows for multithreading to be used.

5.6 MathOptInterface Wrapper

As was covered in Section 3.2.1, MathOptInterface is a standard interface that optimisation solvers can implement to allow compatibility with JuMP, and other pieces of optimisation software. MathOptInterface defines a large number of functions for accessing and setting variables, as well as running behaviours within the software. Packages that are compatible with MOI define a 'wrapper' between the MOI types and functions, and those internal to the package itself. It is useful to think of this as a translator between two languages, one that is very general (MOI) and one that is more specialised (RAM).

The wrapper has two sections. The first is a composite type that contains values related to the state of the wrapper and the solver (for example, the configuration settings), as well as the struct that defines the actual solver (for this package this is the RAMProblem type). The second section is a large collection of functions that translate between the MOI format and the format used internally.

A large benefit of the way MOI is designed is that the wrapper can implement as much or as little is needed for the solver to function. Having a functioning package with only a handful of wrapper functions for setting up and solving a problem is totally possible. If this was the case, then it would not be overly problematic to allow users of the package to create their own representations of objective functions and constraints. However, if a larger subset of the MOI interface is implemented then it becomes far more complex. This complexity is specifically related to problem modification.

MOI defines a large number of ways to modify a problem. For example, modifying the coefficients within constraints, or adding variables to the problem. If each algorithm provided their own implementation of an objective and constraint then each would require a custom implementation of problem modification. This is a behaviour that requires careful design and knowledge of MOI to implement correctly, as well as extensive tests to verify correctness. This therefore shows another reason for enforcing preset objective and constraint definitions.

5.7 Stopping Conditions

An important usability feature of optimisation software is being able to define conditions at which it will be able to stop running, and report what reason for stopping. Julia's multiple dispatch system offers a highly extensible system for implementing this. The package is designed with a small collection of commonly needed conditions (for example number of iterations, or runtime), but the system can easily be extended with custom stopping conditions.

The stopping condition system has three distinct parts. Firstly is the `StoppingCondition` abstract type. Any stopping condition must be a subtype of this. An example of this is the `IterationStop` struct in Listing 18. This type contains any configuration values for the stopping condition, and also performs selection of the correct evaluation via multiple dispatch.

```julia
struct IterationStop <: StoppingCondition
    value::Int64
end

StopConditionStatus(::IterationStop) = ITERATION_LIMIT()
```

Listing 18: Stopping Condition Implementation

Secondly is the `stopcondition` function, Listing 19. This function is called on each iteration of the algorithm and should return a boolean value to indicate if the condition has been met or not. This function has an implementation for every stopping condition, with the specific stopping condition selected via multiple dispatch.

```julia
function stopcondition(model::RAMProblem, iterations_limit::IterationStop)::Bool
    return model.iterations >= iterations_limit.value
end
```

Listing 19: Stopping Condition Function

Finally is the `StopConditionStatus` status function (also in Listing 18). Again this is a single function that has an implementation for each stopping condition and is selected with multiple dispatch. This is called if a stopping condition indicates that it has been met, and the status function is then used to set the status of the overall problem. This function can be skipped, and the status will be updated to `UnknownStoppingCondition`.

This method allows for new stopping conditions to be very easily added, only requiring the new type and function to be impemented and with no modifications made to existing code. This pattern is possible thanks to Julia's use of multiple dispatch.

Multiple stopping conditions can be given to RAM by combining the desired conditions into a vector which is then checked in turn on each iteration. While this may give the impression of being inefficient, if the stopping conditions are small operations, Julia's compiler is able to inline the functions (meaning that their content is moved to the caller's scope), resulting in highly efficient code.

Listing 20 shows the function `check_stopcondition`. This function is called on each iteration to check indicate if any stopping conditions have been met. It simply iterates through each of the configured stopping conditions until one evaluates as true, otherwise returning false and allowing the iteration to continue.

```julia
function check_stopcondition(model::RAMProblem,
                             conditions::Vector{S}
                            )::Bool where {S<:StoppingCondition}

    for c in conditions
        stopcondition(model, c) && return true
    end
    return false
end
```

Listing 20: `check_stopcondition` Function Implementation

In addition to the illustrated iteration stopping condition, a condition that stops iteration after a configured number of seconds is included.

5.8 Distributed Computing

It is illustrated in [4] that row action methods have the potential to be very efficiently parallelised, theoretically allowing perfect scaling. A requirement of the project is to demonstrate this scaling ability, and make it simple for algorithms to be made parallel.

5.8.1 Threading

The manner of distributed processing selected for implementation in RAM was local threading within Julia on a single machine. This is less extensible than a distributed computing platform such as MPI [31], as the maximum number of concurrent operations is limited to the amount of logical cores that available on the host machine. However, threading is simpler to implement than a full MPI interface, as well as being potentially more portable.

Multithreaded applications are able to create multiple concurrent computations that each are run on their own logical core within a computer, each computation is known as a thread. These threads have shared memory, meaning that communication between them is fast but can easily lead to issues where the shared memory is written to at the same time by different threads, which can result in an unknown program state.

The alternative to multithreading is multiprocessing. This is the process of spawning an entirely new operation system process that has its own program state and memory, with data being passed between the two via the operation system. Typically multiprocessing is slower than multithreading when many small operations are needed, but is much easier to manage for a more complex operations.

Multithreading has been selected for use in this package as each single operation is relatively simple, and updates only a single value in the shared memory. The distribution of operations can be implemented in such a way that there is no possibility of a value being written to by two threads at the same time.

In addition Julia's threading package has also been shown to be stable and resilient to crashing due to concurrent accessing of shared memory. It is worth noting that the threading module is marked as Beta by the Julia developers, but this is due to possible future changes in its API rather than problems with stability of the code.

5.8.2 Almost Cyclic Control Of Hildreth's Algorithm

Hildreth's algorithm is the only target algorithm for implementation in this project, and its original formulation [8] assumes each iteration to be performed subsequently. This is not directly parallelisable. However, [9] extends the proof of convergence of Hildreth's algorithm to cover cases where the iteration is defined by almost cyclic control, and also show that this control can deliver superior convergence results.

To define an almost-cyclic sequence, let $I = \{1, 2, \ldots, m\}$ be a finite set. A sequence $\{i_k\}_{k=0}^{\infty}$ is almost cyclic on I if:

- $i_k \in I$ for all $k \geq 0$
- There exists integer C such that for $\forall k \geq 0, I \subseteq \{i_{k+1}, \ldots, i_{k+}\}$

The parameter C ensures that after at most C iterations, every index will have been visited. Almost cyclic control applies an almost-cyclic sequence to control the order that rows are considered.

This result illustrates how applying multithreading to Hildreth's algorithm does not negatively affect the algorithm itself. And as will be shown in Section 8, this control can even result in superior performance.

5.8.3 Threading Implementation

RAM defines an iteration complete when every row in the problem matrix has been considered, and the appropriate variables updated. To maintain this design and avoid having to modify existing code it was decided to allow each operation within a single iteration to be distributed, but that each of these must complete before moving to the following iteration.

For a problem of n dimensions this allows up to n operations to be performed concurrently. While this will result in unused computational power for small dimensioned problems, these problems are not typically the ones that require the acceleration provided by multithreading.

In addition, this requires the definition of a new temporary variable for each thread to access. This is necessary to ensure consistent behaviour for different algorithms. The alternative would be to require algorithm designers to set a threading variable themselves, which could create problems if they chose a type that was not thread-safe.

```
        thread_var = convert(Vector{T}, GetTempVar(model))
        while !check_stopcondition(model, conditions)
            Threads.@threads for i in 1:length(thread_var)
                thread_var[i] = IterateRow(model, i, thread_var)
            end
            model.iterations += 1
            VarUpdate(model, thread_var)
        end
```

Listing 21: RAM Threaded Iteration

The implementation of this operation is given in Listing 21. Initially the temporary data vector is created, this takes the initial values returned from the algorithm and ensures that it is in the required vector format. The stopping conditions are then checked as before by a while loop. The iteration within the while loop utilises the `@threads` macro to distribute each iteration to an available thread. This continues until every iteration is complete, with the maximum number of concurrent operations being controlled by the number of threads Julia is configured to use.

Note that each index of `thread_var` is only ever updated once, this avoids any possibility of multiple writes occuring at the same time. Finally, after each iteration the function `VarUpdate` will ensure that the internal state of the algorithm is kept consistent with the result of the iteration.

There is no guarantee the order in which each iteration of the for loop will be carried out, or the input variables that an iteration will receive. For example, a 40 dimensional problem with eight threads will start the first eight computations with the initial `thread_var` value, and each may not take the other answers into account when calculating their results (depending on the kind of operation they perform).

For Hildreth's algorithm specifically, the out-of-order operation is not problematic (due to the almost-cyclic control). However, this does not offer any guaruntees of convergence for cases where several operations are performed in parallel without taking previous results into account. Testing on this method has not indicated that this is problematic. The results of the algorithm show very good scaling against the number of threads, as well as faster convergence than the non-threaded version (most likely due to the almost-cyclic control). However larger range of tests on a range of problem dimensions needs to be performed to confirm this observation.

5.9 Hildreth's Algorithm Implementation

To create Hildreth's algorithm the model API needs to be implemented. Listing 17 shows the model formulation type, with each variable within it corresponding to the same variables defined in Section 2.1.1. The constructor of this type only takes the initial point as an opertional variable, and does not define the other variables.

5.9.1 Hildreth Build Function

The additional values within the model formulation are set within the `Build` function, Listing 22. As can be seen, the matrices that define the problem are first accessed with the constraint and objective APIs, giving variables `G`, `h`, `B`, and `d`. The interior variables are then calculated.

```
 1  function Build(model::RAMProblem, method::Hildreth)
 2      #Need G to be dense
 3      G = Matrix(GetConstraintMatrixTransposed(model)')
 4      h = GetConstraintVector(model)
 5
 6      B, d = GetObjectiveFactorised(model)
 7      method.A = G/B.U
 8      method.b = h + (G/B)d
 9
10      method.Δ = (method.A * method.A')'
11
12      method.n = length(h)
13      if method.user_initial_point == nothing
14          method.z = rand(Uniform(0,10), method.n)
15      else
16          method.z = method.user_initial_point
17      end
18      method.x = -method.A'method.z
19  end
```

Listing 22: Hildreth's Algorithm Build Stage

While mathematically correct, this function would benefit from several future improvements. As will be demonstrated in Section 8, this build stage takes far longer than the actual optimisation for the majority of problems. This is due to the calculations of the dual variables `A` and `b`, performed on lines 7 and 8 of the function. These calculations are inherently expensive, however this is made worse for sparse problems by the pre-factorised matrix `B` not being sparse. This issue was discussed in 5.3.

Additionally the range of the random selection of the initial point has currently been chosen arbitrarily. While the algorithm only requires that the point be in the non-negative orthant of the problem space, it is likely that certain ranges of this variable will effect the problem. Further experiments would need to be done to see if changing this range will have an effect on the results of the algorithm.

5.9.2 Hildreth Standard Iteration

A single iteration is defined as running over all rows within the problem matrices. This is implemented with the function shown in Listing 23.

```
function Iterate(method::Hildreth)
    for i in 1:method.n
        w = method.Δ[:,i]'*method.z
        w += method.b[i]
        w /= method.Δ[i,i]
        w = method.z[i] - w
        method.z[i] = max(0,w)
    end
end
```

Listing 23: Hildreth's Algorithm Iteration Stage

This series of calculations exactly corresponds to a part of the formulation of Hildreth's algorithm given in Section 2.1.1. While it may appear that the temporary variable `w` is an unnecessary allocation, experimentation showed that removing this and performing the calculation in a single statement had negligible performance impact. This implies that the compiler is able to recognise that this is a temporary variable and removes it. This iteration allocates no new memory, as it only updates the single variable that has already been allocated. Therefore to aid readability, the temporary variable is left in place.

The threaded implementation is very similar to the non-threaded iteration, but only performs an update of a single row, rather than iterating over all rows.

5.9.3 Resolution and Variable Access

The final parts of the necessary API that Hildreth's algorithm must implement is the resolution and variable access functions, shown in Listings 24 and 25 respectively.

```
function Resolve(model::RAMProblem, method::Hildreth)
    method.x = -method.A'*method.z
end
```

Listing 24: Hildreth's Algorithm Resolution stage

The resolution sets the dual variable `x` to its final value from the working variable `z`, this is the operation given in (4b). `GetVariables` implements the operation given in (2b) and returns the initial problem vector.

```
function GetVariables(model::RAMProblem, method::Hildreth)
    B, d = GetObjectiveFactorised(model)
    return B.U\method.x - B\Vector(d)
end
```

Listing 25: Hildreth's Algorithm Variable Access Function

It is worth noting that (2b) is a relatively expensive operation. This is the reason that the result of this is saved in the `result` entry of `RAMProblem` (Listing 11), so that multiple calls to access these variables does not result in additional computation.

6 Design - DirectSearch.jl

The design of DirectSearh.jl (DS) was mostly completed after much of RowActionMethods.jl was implemented, and therefore a lot of the design decisions were made based on lessons learnt during that process. For instance, becoming more familiar with the process of designing custom types and using them for the implementation of highly extensible APIs.

6.1 Structure

DirectSearch.jl has many similarities in its design to RowActionMethods.jl. Both are constructed around a central data structure and utilise multiple dispatch for API implementation. Individual sections differ to a large degree, with the exception of the stopping conditions, which is very similar.

RAM had two internal APIs (the constraint API and the objective API), and a single API for algorithm implementation. As the objective function and all constraints are stored as function handles, it is not possible to offer the poll/search algorithms any analytical information about them, as was the case in RAM.

The alternative internal APIs offered by DS are the cache API and the mesh API. There is also an algorithm API for each of the poll and search stages. The design of each of these parts is discussed in their own following sections.

6.2 Problem Construction

The package is encapsulated by the `DSProblem` type, Listing 26. The `mesh`, `poll`, `search`, `constraints`, `meshscale`, and `cache` variables will each be discussed in their own sections.

`N` describes the dimensions of the problem, `function_evaluations` records the number of times the objective function has been evaluated (as there are generally multiple evaluations per iteration), iteration tracks the number of iterations, and `hmax` is the maximum constraint violation function, and is covered in the constraints section.

```julia
mutable struct DSProblem{T} <: AbstractProblem{T}

    #Solver Config
    mesh::AbstractMesh
    poll::AbstractPoll
    search::AbstractSearch

    #Problem Definition
    objective::Function
    constraints::Constraints
    sense::ProblemSense
    #Feasible Incumbent point
    x::Union{Vector{T},Nothing}
    #Feasible Incumbent point evaluated cost
    x_cost::Union{T,Nothing}

    #Infeasible Incumbent point
    i::Union{Vector{T},Nothing}
    #Infeasible Incumbent point evaluated cost
    i_cost::Union{T,Nothing}

    user_initial_point::Union{Vector{T},Nothing}

    #Problem size
    N::Int

    function_evaluations::Int

    iteration::Int
    iteration_limit::Int
    status::OptimizationStatus

    meshscale::Vector{T}

    cache::PointCache{T}

    num_procs::Int
    max_simultanious_evaluations::Int

    #Barrier threshold
    h_max::T
end
```

Listing 26: `DSProblem` struct

`sense` is a enum that has either the value `Max` or `Min`. As MADS is a minimisation algorithm, this is recorded to multiple any cost evaluations by -1 if a maximisation is required.

`x` and `x_cost` record the current feasible incumbent solution and the associated cost. `i` and `i_cost` are the incumbent infeasible solution and cost, if progressive barrier constraints are being used.

As the feasibility of the initial point supplied by the user is unknown, it is not stored in `x` or `i`. To avoid slowing the process of building the problem in the case of using very expensive constraints (or if not all constraints have been defined), it is not desirable to evaluate the feasibility of the point immediately. Therefore it is temporarily stored in `user_initial_point` until optimisation is started, at which point it will be moved to either `x` or `i`.

As with RAM, a set of problem status is defined that report on the internal state of the solver, stored in the `status` variable. While it was intended to integrate this with the stopping conditions (as it is in RAM), this status is currently defined as an enum that has the values `Unoptimized`, `PrecisionLimit`, or `IterationLimit`.

6.3 Mesh

As all algorithms being implemented within this package use the same definition of a mesh (i.e. the same matrices define its directions, and the same scalar variables define its size) therefore it would be a valid decision to contain these values within the problem definition. However, this reduces the ability for the software to be later expanded if alternative definitions of a mesh are needed for future methods.

6.3.1 Mesh Definition

The definition of the mesh is given in (5). Apart from the point x, this requires the mesh size parameter Δ^m, and the direction set D. In addition, the definition given in [6] defines D with a generating matrix G. LTMADS and OrthoMADS make use of the same default values of these matrices,

$$G = I, \quad D = \begin{bmatrix} I & -I \end{bmatrix}, \tag{27}$$

and do not even explicitly reference G. However future variations of these algorithms could modify these variables. As these are used in the definition of the mesh, they are included in a single data type.

In addition, the value of Δ^m is defined by the mesh index l, and the poll size parameter, Δ^p is also related to the values of Δ^m and is defined by l. Therefore it makes sense to also define these values within the same data type. These values are contained in the `Mesh` data type, Listing 27.

```julia
mutable struct Mesh{T} <: AbstractMesh
    G::Matrix{T}
    D::Matrix{T}
    l::Int
    Δᵐ::T
    Δᵖ::T
    function Mesh{T}(N::Int64) where T
        mesh = new()
        mesh.l = 0
        mesh.Δᵐ = min(1, 4.0^(-mesh.l))
        mesh.Δᵖ = 2.0^(-mesh.l)
        mesh.G = Matrix(I,N,N)
        mesh.D = hcat(Matrix(I,N,N),-Matrix(I,N,N))
        return mesh
    end
end
```

Listing 27: `Mesh` type

Note that the constructor of `Mesh` implements the default values of all variables for LTMADS and OrthoMADS. It was decided that these default values are likely to be retained in future algorithms, therefore their default values would be suitable. If this does need to be changed in a one-off basis it is simple to override the constructor to provide new values. If an alternative algorithm is added that does need to change these values then a group of setter functions can easily be implemented.

Note that the `Mesh` type inherits from the `AbstractMesh` type, and the `mesh` variable in `DSProblem` is defined as `AbstractMesh`. In a future algorithm requires a different set of variables to define the mesh then a new struct that inherits from `AbstractMesh` can be defined and is already fully compatible with `DSProblem`.

6.3.2 Mesh Update

The defining feature of MADS is the update of the mesh depending on the results of the iteration. In DS this is accomplished with the `MeshUpdate!` function. The basic update rule utilised by both LTMADS and OrthoMADS is defined as a default for the `Mesh` type, Listing 28. This function will increment/decrement the mesh index and then update the poll and mesh size parameters accordingly.

This function is only called when a more specialised implementation isn't available (note that the second argument is for `AbstractPoll`). For LTMADS this function is the only update behaviour required, therefore a specialised implementation is not provided. However OrthoMADS needs to also update internal variables at the same time as the mesh variables. Therefore OrthoMADS provides its own update function to ensure its own internal state is also updated.

6.4 Search Step

The search API defines how a search algorithm, Section 2.2.1, is implemented within the package. The search step may be any algorithm that returns a set of trial points on the current mesh.

```
function MeshUpdate!(m::Mesh, ::AbstractPoll, result::IterationOutcome)
    if result == Unsuccessful
        m.l += 1
    elseif result == Dominating && m.l > 0
        m.l -= 1
    elseif result == Improving
        m.l == m.l
    end

    m.Δ^m = min(1, 4.0^(-m.l))
    m.Δ^p = 2.0^(-m.l)
end
```

Listing 28: Default Mesh Update

Listing 26 has the entry `search` of type `AbstractSearch`. This contains a type that corresponds to the implementation of a search step. This type is used for function dispatch when the search step is called.

Listing 29 shows the function Search. This performs two functions, firstly the `GenerateSearchPoints` function is called (this being the function with multiple implementations that actually defines the search), and returns a set of trial points. The trial points are evaluated with `EvaluatePoints!`, which checks the points against the cache, constraints, and the objective function to determine if any of the points are improvements to the current solution. This function is discussed later in the section.

```
function Search(p::DSProblem{T})::IterationOutcome  where T
    points = GenerateSearchPoints(p, p.search)
    return EvaluatePoint!(p, points)
end
```

Listing 29: Search Step Function

`GenerateSearchPoints` is given an implementation for each search algorithm, and is dispatched on with the type of the `search` field within the problem definition. This value is also used to configure the search step.

```
mutable struct RandomSearch <: AbstractSearch
    M::Int #Number of points to generate
end

function GenerateSearchPoints(p::DSProblem{T}, s::RandomSearch
                             )::Vector{Vector{T}} where T
    RandomPointsFromCache(p.N, p.cache, p.mesh.Δ^m, s)
end

function RandomPointsFromCache(N::Int, c::PointCache{T}, dist::T, s::RandomSearch
                              )::Vector{Vector{T}} where T
    mesh_points = CacheRandomSample(c, s.M)

    for i in 1:s.M
        dir = rand(N)
        dir *= dist/norm(dir)
        mesh_points[i] += dir
    end

    return mesh_points
end
```

Listing 30: Random Search Example

Listing 30 shows the included search step, a simple algorithm that randomly samples previous points from the cache and returns them after being offset by the current mesh size parameter in a random direction offset. This particular algorithm isn't designed to be particularly useful, but rather to serve as an example for more practical implementations.

The `RandomSearch` struct configures the search step with the number of points to generate, and then `GenerateSearchPoints` is implemented with the actual sampling algorithm.

The freedom in the search step allows for many different approaches to be taken. For instance, if an approximation of a problem is known then this could be used to generate promising trial points in the search step that are evaluated. Alternatively a generic approach could be taken (such as the cache sampling just discussed), or the search step could be skipped entirely.

Unless otherwise specified DS will set the desired search algorithm to the `NullSearch` type. The corresponding algorithm for this type is a function that returns an empty list of points, essentially skipping the search step.

6.5 Poll Step

The poll step is implemented in a very similar manner to the search step, with the `Poll` function, Listing 31, appearing to be almost identical.

```
function Poll(p::DSProblem{T})::IterationOutcome where T
    points = GeneratePollPoints(p, p.mesh)
    return EvaluatePoint!(p, points)
end
```

Listing 31: `Poll` Function

The main difference is that there is only a single implementation of the `GeneratePollPoints` function. This function is responsible for taking the directions returned by the poll algorithm and producing points. The implementation is shown in Listing 32.

Firstly the function will run the poll algorithm to generate a set of directions. These are then used to generate points from each of the incumbent points by applying an offset with the mesh size parameter in the generated direction. It is allowed within the MADS algorithm to use the same set of directions for both feasible and infeasible incumbent points. OrthoMADS is deterministic and so would resolve to the same set of directions if called twice anyway. Performing a single call just saves computations.

An interesting point is how the function has to handle both feasible and infeasible incumbent points. When the problem is instantiated both kinds of incumbent points are initially valued at `Nothing` (this is the Julia implementation of the Null value, equivilent to `None` in Python, or the null pointer in C). One of them will be set to a valid vector after the user-defined initial point is evaluated, but the other will remain with a null value. This is checked for in this function by only evaluating the poll step about the incumbent point if an actual value does exist.

```
function GeneratePollPoints(p::DSProblem{T}, ::AbstractMesh
                           )::Vector{Vector{T}} where T
    points = []
    dirs = GenerateDirections(p)

    if !isnothing(p.x)
        append!(points, [p.x+(p.mesh.Δᵐ*p.meshscale.*d) for d in eachcol(dirs)])
    end
    if !isnothing(p.i)
        append!(points, [p.i+(p.mesh.Δᵐ*p.meshscale.*d) for d in eachcol(dirs)])
    end

    return points
end
```

Listing 32: `GeneratePollPoints` Function

The meshscale parameter in each of the array comprehensions is defined in the problem description and serves as a variable scaling parameter. By default this is initialised as a vector of ones, but upper and lower bounds can be applied to each variable, which then set the scaling parameter following the expression,

$$s = 0.1(u - l), \tag{28}$$

where s is the scaling factor, and u and l are the upper and lower bounds.

This is similar to the scaling rule used in NOMAD [22]. However NOMAD sets this factor as the initial value of the mesh and poll size parameters, rather than as a scale factor on each variable. Further evaluation is required to confirm if this approach is effective, and a preferred solution would be to apply variable scaling within the objective function black box to provide DS with variables that are all of the same scale.

6.6 Poll Algorithms

In theory, the design of the software allows for a single function and type to encapsulate an entire poll step, as this is all that is called when needed. However as the poll direction generators are moderately complex algorithms, implementing them in a single function makes them far harder to optimise and test. This section will go over the structure of each of LTMADS and OrthoMADS, and show some of the optimisations made within their design (as these are one of the most performance critical parts of the algorithm.

6.6.1 LTMADS

The mathematical basis of LTMADS was briefly covered in Section 2.2.2. The initial approach taken to the design of the algorithm was to transfer each bullet point into its own function, and implement it directly into Julia code to ensure that the routine generated the correct output.

As example of this is the function `b_l_generation` which implements the $b(l)$ function in MADS. This has the initial implementation shown in Listing 33. Note that `b` and `i` are dictionaries stored in the `LTMADS` type and remain between iterations. This code first checks if the current mesh index has been used before and if so, returns the `b(l)` vector and the index `i` from within it.

```
1 function b_l_generation(b::Dict{T,Vector{T}}, i::Dict{T,Int}, l::T, N::Int
2                        ) where T
3     if !haskey(b, l)
4         i[l] = rand(1:N)
5         b[l] = [rand(-2^l+1:2^l-1) for _=1:N]
6         b[l][i[l]] = rand([-1, 1]) * 2^l
7     end
8     return b[l], i[l]
9 end
```

Listing 33: LTMADS Function that Implements $b(l)$

```
x = []

for _ = 1:N
    push!(x, rand(-2^l+1:2^l-1))
end
```

Listing 34: Equivalent Form of a List Comprehension

If the mesh index has not been used before (ie the key does not exist within the dictionary), then a new index and vector are created for this mesh index and stored in the dictionaries before being returned. This works well and correctly implements the function.

Once the code is functionally correct (verified with unit tests and comparison to known-correct outcomes) it needs to be optimised. As the poll code is called on every iteration, it is important that it runs as quickly as possible. If a problem needs 10000 iterations to complete, then each poll step taking 100ms would lead to a computational overhead of 1000s. It is not possible to have the direction generation complete in zero time, but minimising it as much as possible is a priority.

In Julia allocating memory is a relatively expensive operation, if it is done every single time a loop goes around then it rapidly becomes a bottleneck in code. If new memory does not need to be allocated on each loop, then it is possible to preallocate memory and use the same variable on each loop.

A subtle case of this is on line 4 of Listing 33. The vector `b[l]` is being populated by a list comprehension. A list comprehension is a nice syntax for compactly iterating over an array of data, performing an operation on it, and creating a new array with the results. An equivalent form of a list comprehension is shown in Listing 34. This allocates an empty array, and then continuously expands it with each new entry.

The issue with this is that the act of expanding the object in memory is moderately expensive. An alternative formulation is to preallocate an array of the correct size for `b[l]` and iterate over it, inputting the correct values. This is demonstrated in Listing 35.

As is shown here, if `b[l]` needs to be calculated then initially an array of zeros is allocated in the dictionary. Following this, the zero-vector is iterated over, with each value filled in subsequently.

This kind of optimisation is not overly significant on its own, but going over each aspect of the

```julia
function b_l_generation(b::Dict{T,Vector{T}}, i::Dict{T,Int}, l::Int, N::Int
                       ) where T
    if !haskey(b, l)
        i[l] = rand(1:N)
        b[l] = zeros(T, N)

        for j in 1:N
            if j == i[l]
                b[l][j] = rand([-2^l, 2^l])
            else
                b[l][j] = rand(-2^l+1:2^l-1)
            end
        end
    end
    return b[l], i[l]
end
```

Listing 35: Improved $b(l)$ Function

code and noting where it could be improved will overall lead to much improved performance. This tweak gave a measured performance increase on this function of approximately 20%, reducing runtime from a mean of 50ns to 40ns.

The initial implementation of LTMADS required an average of 600μs to generate a basis for a 100 dimensional problem. With various optimisations (most of which are based on memory preallocation and ensuring types are used consistently) this is reduced to an average of 250μs.

6.6.2 OrthoMADS

The steps of OrthoMADS given when the paper was published [11] map well to single functions within an implementation. The steps were given in Section 2.2.3, but in brief they are:

- Halton sequence generation
- Halton sequence adjusting
- Construction of an orthogonal integer basis

Halton Sequence Generation Julia packages that generate Halton sequences are available. However, of all those found none appeared to be actively maintained. In addition, these libraries are aimed at the use of Halton sequences in a general manner, whereas OrthoMADS only requires the simplest case. Therefore a custom implementation was made, both to guaruntee that code isn't broken in a future Julia update and that any optimisations for the specific usecase can be made.

On each iteration a single Halton sequence is generated with seed value t. The other value that determines the sequence is the a prime number known as the root, with an ordered sequence of primes used for each of the N entries. The `Halton` function in Listing 36 shows these primes being generated, followed by creating the corresponding entry with the `HaltonEntry` function. This step is implementing (9).

```julia
function Halton(N::Int64, t::Int64)
    p = map(prime, 1:N)
    return map(p -> HaltonEntry(p,t), p)
```

```julia
end

function HaltonEntry(p,t)
    u = 0
    a_r = HaltonCoefficient(p,t)
    for (r,a) in enumerate(a_r)
        u += a/(p^r) #note that the equation is a/p^r+1, but julia indexes from 1
    end
    return u
end
```

Listing 36: Halton Sequence generation

What should be noted here is that (9) denotes an infinite series. This is possible to compute due to the a values being the base p expansion of t. Every number has a unique expansion in a given base (regardless of the base being prime). Two approaches were tested to implement this expansion to find the expansion.

Firstly is the simplest, and most reliable, way within Julia. This involves parsing the input number into a string in the desired base. The characters that represent each digit in the string can then be iterated over and cast back into an integer. This approach, while reliable (as it is using built-in functions of Julia) has the disadvantage of allocating a moderate amount of memory, and therefore takes a moderate amount of time to complete. Julia also requires any base entries to be in the range $2 \leq \text{base} \leq 62$, making this unsuitable for larger dimension problems.

The approach taken instead was to write a custom function to perform this operation. This function runs quickly and shows good scaling as the size of inputs increase. The previous example of using the inbuilt functions is still of use however, as it can be used in tests to confirm that the values given by the custom function are correct.

Sequence Adjustment The current adjustment routine is a close to direct replica of the adjustment step defined in (10), shown in Listing 37. An area that this could be improved is in the `argmax` function. The current implementation uses a very simple line-search routine which isn't able to find an exact solution. The non-exact solution isn't of consequence due to the rounding performed on the result, however finding some kind of analytical solution may be more performant.

```julia
function AdjustedHalton(halt, n, l)
    q = AdjustedHaltonFamily(halt)
    α = (2^(abs(l)/2)/sqrt(n)) - 0.5

    α = argmax(α, x -> norm(q(x)), 2^(abs(l)/2))

    return q(α)
end
```

Listing 37: Halton Adjustment

This stage also shows a few cases where more optimisations could be made. For instance, repeated calculations of the $2^{|l|}$ term.

Basis Construction The simplest step in the process is the construction of the orthogonal basis. This uses the Householder transform (12), and is directly implemented in the software.

```
function HouseholderTransform(q)
    nq = norm(q)
    v = q./nq
    return nq^2 .* (I - 2*v*v')
end
```

Listing 38: Household Transform

6.7 Point Evaluation

Following the generation of trial points by the search or poll steps, the points must be evaluated and the algorithm updated depending on their outcome. As previously noted this is performed by the `EvaluatePoint!` function. As this is a long and complex function (dealing with many edge cases) an overview of the decisions it makes are presented in Figure 5.

Within this figure, `con` denotes the constraint function (18), y is the test point, h_x and h_i are the feasible and infeasible incumbent constraint violations respectively, f is the objective function, and c_x and c_i are the feasible and infeasible incumbent costs respectively.

This procedure is repeated for each of the trial points, with the iteration completing by choosing the optimum point(s) out of those evaluated. The problem can store both a feasible and infeasible optimal incumbent point. The feasible incumbent point may only be updated if another feasible point with a lower cost is discovered. The infeasible point will only be updated by another infeasible point with a lower constraint violation value.

If a new incumbent is a dominating point then the entire iteration is defined as dominating. Otherwise if a point is improving, the iteration is defined as improving. If no points are discovered that improve either incumbent value then the iteration is a failure. Definitions of these three outcomes are given in Section (2.2.5).

Finally this iteration result is then used to update the mesh values accordingly. Future improvements in this section would be the ability to end an iteration early as soon as a condition is met. For instance, if a dominating point is found, or any point offers a cost reduction greater than a threshold value.

6.8 Point Cache

While only briefly mentioned within any of the MADS papers, a point cache is one of the most useful and performance critical aspects of the algorithm. This is due to the potentially high cost of each function evaluation. If the algorithm has previously evaluated a trial point then the result can be referenced from within the cache. This avoids repeated objective function evaluations.

6.8.1 Data Structure

As the cache is accessed on each iteration and could potentially contain many thousands of entries (especially if a cache is preloaded before optimisation) it is necessary to make use of a data structure that does not become slower to access as it increases in size. These requirements made a dictionary the clear choice for an initial data structure. This structure pairs keys (the

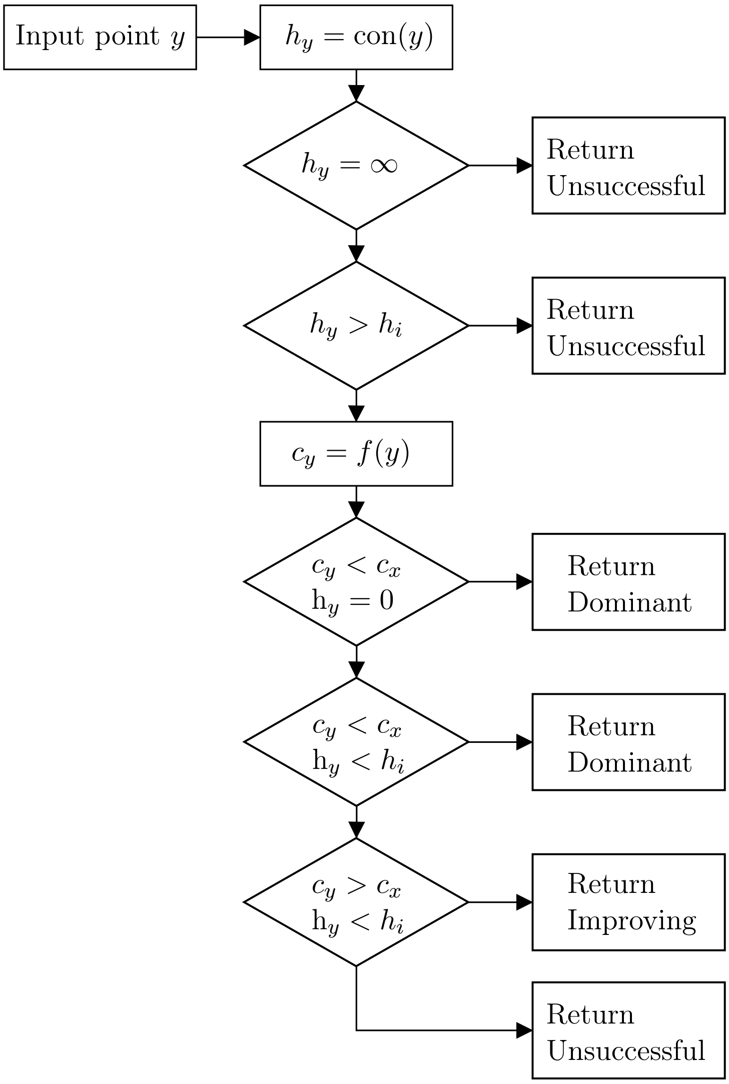

Figure 5: Logical flow of the point evaluation function

input points) with results (the cost of the points), and makes use of a hashing algorithm to access the result in constant time, regardless of the size of the dictionary.

Therefore at its most basic level, the cache should be a dictionary that maps points to a cost. This kind of data structure shows very good performance for key lookup. For a key being a 15-dimensional vector (for a 15-dimensional problem) mean lookup times are consistently in the range of 300ns-330ns, even when over a million elements are in the cache. Compared to a cache implementation based on a vector, this is clearly much more performant as a vector has no ability to hash entries, and therefore has an $O(n)$ lookup time. A dictionary uses marginally more memory than a vector, however this is a minor part of the overall memory cost, as the data itself uses the vast majority.

Unfortunately when moving to testing on problems with a high dimensionalty during testing it was shown that the dictionary lookup became quite slow. The MADS authors suggest a maximum target size of up to 500 variables [22]. A lookup operation in the cache for this key vector dimension is benchmarked as taking a mean time of 6μs. If a problem requires, for example, 100000 cache lookups then this is an additional overhead of over half a second. This

```julia
mutable struct PointCache{T} <: AbstractCache
    #Map a point to a cost value
    costs::Dict{Vector{T},T}
    #List the incumbent points in the order they are considered
    order::Vector{Vector{T}}

    function PointCache{T}() where T
        c = new()
        c.costs = Dict{Vector{T},T}()
        c.order = Vector{Vector{T}}()
        return c
    end
end
```

Listing 39: `PointCache` Data Type

is a moderate overhead, but could be improved.

This performance could be improved by using a different data structure entirely, for example the NOMAD authors achieve very good performance using a custom binary tree [22]. Or by implementing an alternative hashing algorithm that is specialised to the key data (large floating point vectors).

6.8.2 Cache API

To be useful the cache needs to have more utility than just providing a map of points to costs. For instance, it may be of use to know the order that points in the cache where considered, or how they were evaluated by the algorithm (infeasible feasible but of higher cost, etc.). This illustrates two additional downsides of a dictionary, they are unable to contain more than one of the same key, and they do not, by default, preserve the order of entries.

For this project it was decided to focus on the main cost cache, and a recording of the order of points. This does has a cost in memory, but it is not significant in the general case (i.e. when not considering problems with a very large size) given the amount of memory available in modern computers. The struct that includes this information is shown in Listing 39.

In the more restrictive cases (for example, when embedded), a different cache can be implemented that excludes the parts of the cache that are not compatible with the environment. In addition, constraints are to be added that set the maximum size of the cache, ensuring that it never uses too much memory. This could potentially also involve storing the cache to disk once it reaches a certain size.

A variety of functions are necessary for operating on the cache. As previously discussed, while the cache is exposed to the algorithms (allowing them to modify or access it directly) this harms the maintainability of the code. Therefore a large number of functions are defined to provide access to the cache, as well as writing to it.

For example writing to the cache is completed with the function `CachePush`, which performs checks to ensure the point and cost is valid, and then adds the point and its cost into the cache data structures.

Other functions include `CacheGet`, `CacheQuery`, and `CacheRandomSample` rather respectively return the cost of a point in the cache, return boolean to indicate if a point is in the cache, and return N random points from within the cache.

```
function CachePush(c::PointCache{T}, x::Vector{T}, cost::T) where T
    push!(c.costs, x=>cost)
    push!(c.order, x)
end
```

Listing 40: `CachePush` Function

```
mutable struct Constraints{T}
    collections::Vector{ConstraintCollection}
    count::Int
    function Constraints{T}() where T
end
```

Listing 41: `Constraints` Type

6.8.3 Future Cache Improvements

Core features to improve the functionality of the cache would include the ability to save and load a cache into the package. Having the ability to reference a set of precalculated results has the potential of giving much better performance, for an entire run of the algorithm. The cache would also benefit from improvements such as storing the result of constraint evaluations of points, as well as allowing an inexact match (within a small tolerance) to trigger a cache hit.

The cache would also benefit from moving to an alternative data structure that lacks some of the disadvantages of the dictionary. Fortunately the design of the package allows for an additional cache to be built alongside the current one without interfering, and then swapped in when it implements all required functionality and is fully tested.

6.9 Constraints

When considering just the extreme barrier constraints the DS constraint system was relatively easy to implement. At its core, this was originally just an array of function handles that would be called in turn. Each of these handles returns a boolean to indicate if the constraint has been violated or not. This is then used to indicate whether or not a point is feasible.

This process became more complex when dealing with progressive barrier constraints, and being able to handle combinations of extreme and progressive barriers.

6.9.1 Constraint Organisation

The chosen approach was to define all constraints within a *collection*. Each collection contains an arbitrary number of constraints all of the same type (either progressive barrier or extreme barrier for the purposes of this report).

The collections are then collated into a single data structure, `Constraints`, which is held within the problem description. By default constraints, Listing 41 contains a vector of collections, as well as a count on the number of collections that have been defined.

The collections are defined within the `ConstraintCollection` type, Listing 42. The collection contains more detail relating to its individual status and configuration. It contains the vector of

```
mutable struct ConstraintCollection{T,C<:AbstractConstraint}
    constraints::Vector{C}
    count::Int
    ignore::Bool
    h_max::T
    h_max_store::Vector{T}
    h_max_update::Function
    result_aggregate::Function
end
```

Listing 42: `ConstraintCollection` Type

constraints as well as the count. In addition it has an ignore boolean, future work would allow for individual collections to be enabled or disabled dynamically which could be useful for certain problems.

The final variables all refer to parameters used for progressive barrier constraints.

6.9.2 Progressive Barrier Constraints

Formulating the constraints within collections adds a moderate additional complexity to the constraint evaluation procedure. The original MADS-PB algorithm allows a maximum constraint violation function result of less than any previous violation result, Equation (18). And then updates this maximum violation depending on the outcome of the iteration, Equation (23).

To implement the MADS-PB algorithm a small cache is created within the `Constraint` type (this could be incorporated into the global cache but this was not implemented during the project). At the end of each iteration the cache is wiped, and built again. Algorithm 4 shows the process of iterating through all constraints and trial points. Note that the summation function used to create h_i is configurable. By default the pictured function ($\max(0, c(x)^2)$) is used, but this can be set to an alternative, e.g. an L1 norm.

Algorithm 4 MADS-PB Evaluation Algorithm

```
for all trial points $x$ do
    for all Collections do
        $h_i = 0$
        for all Constraints $c(.)$ in Collection do
            $h_i = h_i + \max(0, c(x)^2)$
            Record outcome of evaluation (Feasible, WeakInfeasible, StrongInfeasible)
        end for
        cache $h_i$
    end for
end for
return $\sum_i h_i$
```

The return type provided by Algorithm 4 has three values: `Feasible`, `WeakInfeasible`, and `StrongInfeasible`. These refer to completely feasible points, points that satisfy unrelaxable constraints but not relaxable ones, and points that violate unrelaxable constraints (or have a relaxable violation greater than h^{max}. This information, along with the cost of the point, is used by the point evaluation function to decide on the outcome of the iteration.

After the outcome of the iteration is decided the constraint variables can be updated. The MADS-PB update rule (23) is implemented by Algorithm 5. The cache is utilised to return the individual h_i value for the given constraint collection (as the combined value cannot be used to update the individual collection).

To follow the update rule for improving iterations, the cache is used again. Every recorded violation for a given collection is filtered to incude only those smaller than h_i, the greatest of which is then chosen as the new $h^{\max}$ value.

Algorithm 5 MADS-PB Update Algorithm

```
Require: Selected point $x$, Iteration outcome
  for all Collections do
    extract $h_i$ from cache for selected $x$
    if outcome is Improving then
      extract all violations for this constraint $H$
      set $h^{max} = \max\{h < h_i : h \in H\}$
    else
      set $h^{max} = h_i$
    end if
  end for
  Record total violation value
  Wipe cache
```

6.9.3 Benefit of Constraint Collections

It is clear from these algorithms that the design has been considerably complicated with the addition of the constraint collections. There are several reasons for their inclusion:

- Improved ability to organise constraints.
- Separation of unrelated constraints.
- Simplification of enabling/disabling constraints.

Organisation is not a functional necessity. However it makes it possible to store constraints in related groups with a single reference to their location, rather than it being up to the user to stored references to many different constraints themselves.

As each constraint collection maintains its own h^{max} value, similar constraints can be grouped together with a configuration that benefits them. Additionally, as h^{max} values can vary separately for each collection. It is possible that this variation could provide improved performance if unrelated constraints are group separately, and would be an interesting future investigation.

Finally, if the user desires to enable/disable constraints this is simplified by the grouping of constraints together. The additional benefit here is that by maintaining their own $h^{\max}$ value these constraints will not unduly affect the convergence towards feasibility of other constraints when introduced. This problem arises when introducing new relaxable constraints, as to even generate improving points it will likely be required to modify the stored $h^{\max}$ value, which could then negatively affect the already present constraints. For this implementation the summed $h^{\max}$ value may also need to be modified to add new constraints, but this will not affect the already present constraints as their individual $h^{\max}$ values cannot be violated, and are unaffected by the new additions.

6.10 User API

The RAM package is able to make use of the features of JuMP and MOI for interfacing with the user. DS does not have this advantage and must therefore make its own robust interfacing options available.

All information about the solver is stored in the `DSProblem` type. The initial problem is considered by instantiating the `DSProblem`. This, at a minimum, requires the problem dimension to be added. By default the problem will specify all other configuration options (for example, selecting LTMADS as the poll algorithm). These can be set manually by setting them as key word arguments to `DSProblem`. Listing 43 shows two ways a problem can be configured.

```
p = DSProblem(2)
SetObjective(p, SomeFunctionHandle)
SetInitialPoint(p, [1.0, 3.5])
SetIterationLimit(p, 1000)


p = DSProblem(2, poll=OrthoMADS(2),
              objective = SomeFunctionHandle,
              initial_point=[1.0, 3.5],
              iteration_limit=1000)
```

Listing 43: Example DirectSearch.jl API

While options can be set in the problem constructor, they can also be set with dedicated functions. This is useful for updating the problem definition after running for a time, or for giving neater code. Inputs that define the problem itself (i.e. the objective function and constraints) also have their own dedicated functions. The objective function is added with `SetObjective`, that takes a reference to any Julia function as its input. Any function passed as an objective should take an array of values as input and return a scalar cost. Unfortunately Julia does not allow this to be enforced, however the optimisation would quickly fail if a different kind of function was provided.

The optimisation itself is performed with the `Optimize!` function. At a minimum the problem must have an initial point, and an objective function that supports that dimension. If these are provided then the MADS algorithm will perform an unconstrained minimisation of the objective.

Additionally constraints can be added with the `AddProgressiveConstraint`, and `AddExtremeConstraint` functions. The details of the constraints and their requirements was previously discussed.

```
AddExtremeConstraint(p, x[1] -> x[1]>0)
AddProgressiveConstraint(p, x[1] -> -x[1])
```

Listing 44: DirectSearch.jl Constraints API

The two examples implement the same constraint. For the extreme barrier a simple inequality is able to determine if a limit is met or not. For progressive barrier constraint, a negation is able to cast the negative valued inputs as positive, indicating to the algorithm that they are violating the constraint.

6.11 Code Portability

As with RAM, DS has been designed using parametric types for majority of functions and structs, allowing alternate types to be used, rather than the assumed default of `Float64`. While RAM generally makes use of just a single parameteric type, certain aspects of DS have been given a second parametric type, intended for use as providing a type to replace the default `Int64` integer type. An example of this can be seen in the OrthoMADS struct, Listing 5.

It should be noted that this particular parametric behaviour has not been implemented everywhere in the package. However as the template for its inclusion has been given in several places, its addition in future should be a relatively simple matter.

As with RAM, testing the package for use with types other than the default `Float64` has not been performed. While the majority of operations should work with similar types (e.g. `Float32`) there will undoubtedly be bugs introduced when changing types. As this functionality is intended for use with custom types for specialist environments it is assumed that users in that context will ensure the package is made compatible. The parametric types are included to make this adaption process simpler.

7 Testing Methodology

In this section the requirements of each package will be revisited, and a set of tests discussed that verify that these requirements have been bet. In addition, the testing environment will be discussed along with some features of Julia that are useful for testing.

7.1 Testing Hardware

When comparing a set of optimisation software packages the main performance criteria that can be measured is the runtime of the algorithms, as well as the memory utilisation. This can be used to indicate the areas in which a solver is more or less capable. For example, if a piece of software took 50% longer to complete an optimisation operation but required only 50% of the memory compared to another, then it may be more preferable in a resource limited environment despite taking longer.

A factor that is not as import in the testing environment is the speed of the individual processors, as this applies a consistent scaling across each program under test. This then gives two main requirements of the testing environment: total memory and number of processors.

The memory must be high enough to not become saturated during testing. If the memory becomes full then the operating system will cause a swap operation to take place where memory is copied to disk. This results in a massive decrease in performance. For DS this is not problematic, the only part of the algorithm that may take a large amount of memory is the cache, and that can be limited in size without having a huge impact in performance. However for RAM this may prove to be a problem when testing on very large problems, despite the use of sparse matrices.

The number of processors sets a maximum limit on the testable parallelism of the algorithms . If an algorithm can have a high degree of parallelism then having a large number of processors available will allow this characteristic to be demonstrated. Ideally the number of processors should be high enough to allow any limitations to be shown, and to illustrate the scaling behaviour of a piece of software.

The testing machine in use by this project is a server that has a pair of Intel Xeon E5-2630v4 CPUs. These each provide 10 physical cores, and 20 logical cores (using to Intel's hyperthreading technology). Assuming that an algorithm shows perfect parallel scaling and there are no performances losses due to data transfer between the two CPUs, these processors can allow for perfect scaling for up to 20 parallel operations (the number of hardware cores). This would then be followed by good scaling up to 40 parallel operations (the number of logical cores). This is due to each pair of hyperthreaded logical cores sharing many resources and therefore not being able to perform as well as a pair of physical cores.

Unfortunately the distributing of processes is handled by the operating system, and there is no guarantee that a pair of processes will not be distributed to a single physical core even if other physical cores are free. Due to this, it is expected that the 'perfect' algorithm would show linear scaling, but less than 1:1.

The server provides 250GB of memory. This is sufficient for the test problems in use by this project.

It is also worth considering that this server is used by multiple other users. To this end it needs to be considered how the tests are impacted by resource utilisation by others, and also how the tests affect the resources available to others. For most tests a minimal number of cores will be utilised. The large number of available processors therefore gives a high probability that neither of the previous issues will occur. The small risk can then be mitigated by monitoring the server's level of resource utilisation, and performing multiple runs of each test.

Tests that require a high number of processors does require more care. Again this can be mitigated by monitoring the resource utilisation of the server, with extra care taken to time the tests when other users are not running their own work, and to ensure the test does not use 100% of the available resources.

7.2 Testing Software

While it would be possible to manually run tests, this is work intensive and increases the liklihood of introducing errors. The solution to this is creating a benchmark suite that forms tests in a consistent way and saves data in a standard format.

For RAM, the use of JuMP makes this process very simple, especially for comparisons against other solvers. A problem can be loaded from files as a JuMP model, and then the problem can be solved several times for each desired test, be that different configurations of RAM, or different solvers entirely. All required statistics can then be gathered and saved automatically.

For DS, all the required problems are available in the SIF format (from the CUTEst problem set). As there is not currently a modern SIF interpreter package in Julia, a Python package is used to implement the wrapper [32], and the Julia's python interface [33] calls the SIF files. Unfortunately this approach could not be applied to RAM, as the interface is not able to load full matrices in the manner that RAM requires.

For both RAM and DS, testing and benchmarking showed the need for an internal record of times was needed to accurately profile the code (due to different problem stages requiring different timings). Therefore each had a `Statistics` component added to their main problem, descriptions. This stores any required information about timings, and is accessed by the benchmarking suite to gather internal information.

In addition it is a necessary feature to be able to configure the number of threads that Julia is able to use. This is not a modification that can be made within a running Julia process, and is configured by an environment variable that Julia reads on startup. This requires the addition of a pair of scripts. The first is bash script that configures Julia with the requisite number of threads enabled and then runs the second script for each configuration. The second is a Julia program that calls runs the benchmarks themselves, and runs through each of the desired benchmarks.

7.3 RowActionMethods.jl Tests

A summary of the requirements of RAM is as follows:

1. Be designed to be simple to add new objectives, constraints, and algorithms.
2. Solve problems with linear constraints and quadratic objective functions.
3. Take advantage of sparse matrices.
4. Offer performance comparable to other quadratic solvers.
5. Demonstrate good scaling as the number of available processors increases.

Requirement one has been demonstrated in Section 5. Requirement two is illustrated by the choice of tests, to be covered later. Requirement three has been shown by the choice of data structures in the design, but will also be shown in a set of tests where the sparse matrices are swapped out for dense matrices.

The majority of tests will relate to requirements four and five. The first group of tests will compare RAM against other QP solvers, illustrating its performance and its ability to find optimal points in a competitive time for a variety of constrained QP problems. This will also show how the performance scales with the number of iterations, and illustrate the key downside of using the dual formulation required for Hildreth's method.

Finally, to illustrate the scaling property of requirement five, a subset of previous tests will be run with a range of threading configurations. This will be shown for a range of test sizes, and compared to the single threaded implementation.

7.3.1 Testset

The common set of tests for evaluating optimisation algorithms is the CUTEst testset. Unfortunately these tests are available only in the SIF format, for which Julia (and more specifically, JuMP) does not have a parser for. Fortunately a package exists for parsing MPS files, a subset of SIF that describes QP problems. A group of QP problems in the MPS format is available from the CUTEst project, the Maros and Meszaros Convex Quadratic Programming Test Problem Set [34].

Of this set, six tests have been chosen for inclusion in this report to demonstrate the performance and behaviour of RAM. The details of these tests are shown in Table 2. The Maros and Meszaros test repository contains over 100 quadratic programming problems with which all have positive semi-definite problem matrices. As RAM is only compatible with positive-definite matrices, the tests were filtered to only include those with this format.

The final set of tests were chosen to show performance with several different contexts. *HS21* and *HS118* represent performance on small problems. The two *LISWET* problems and *AUG2DC* show performance on larger problems. And finally *CONT-100* was selected to show a case where RAM performs very poorly.

Test	Variables	Constraints
HS21	2	1
HS118	15	17
LISWET1	10002	10000
LISWET2	10002	10000
AUG2DC	20200	10000
CONT-100	10197	9801

Table 1: RowActionMethods.jl Testset

7.4 DirectSearch.jl Tests

The requirements of DS are as follows:

1. Implement the LTMADS, OrthoMADS, and progressive barrier algorithms
2. Required number of function evaluations should be similar to NOMAD

3. Final result should be found in a similar time to NOMAD
4. It should be possible to swap in and out different sub-algorithms.
5. It should be possible to define new kinds of constraints easily

Requirements one, five, and six have been shown in the design section. This section will discuss the tests that have been performed to prove requirements two and three.

All the tests used are part of the CUTEst problem set. A Python interface, PyCUTEst [32], has been used via Julia's PyCall package [33]. This adds additional overheads (due to type conversion and Python's slow execution speed), but this limitation is applied to all software being tested, therefore does not affect the results.

NOMAD is being used via its Julia wrapper *NOMAD.jl* [35]. This allows the interface provided to NOMAD to be identical to that provided to DS. This also makes the tested user environment as similar as possible, interacting with the solvers through a high-level interface. The limitation of using *NOMAD.jl* is that several configuration options cannot be defined. For example, selecting between LTMADS and OrthoMADS, or the number of directions generated by each. Therefore NOMAD is configured to use as similar as possible a configuration as DS (e.g. not using a search step), but it otherwise used 'as is'.

7.4.1 Testset

The paper that introduces OrthoMADS [11] includes group of tests that compare OrthoMADS to LTMADS, as well as GPS (the precursor algorithm to MADS). As the results of these tests are available for cross-referencing, a subset of these will be used.

Ideally the entire test-set would have been used, however it was not possible to find several of the tests in a format that could easily be adapted for benchmarking. Therefore the subset of these tests that are present in CUTEst have been used.

Test	Variables	Optimal Cost
ARWHEAD	500	0.0
BDQRTIC	100	378.769
BIGGS6	6	0.0
BROWNAL10	10	0.0
BROWNAL100	100	0.0

Table 2: RowActionMethods.jl Testset

As LTMADS is non-deterministic multiple runs are required to characterise its performance, minimum, mean, and maximum number of function evaluations. OrthoMADS requires only a single run as it is deterministic. As *NOMAD.jl* enforces the use of OrthoMADS, the result of this single run is given.

8 Results

8.1 RowActionMethods.jl

In this Section, RAM is compared against the solvers OSQP [23] and IPOPT [30]. These two solvers have each been chosen for a different reason. OSQP is a C program for solving quadratic problems, and provides a comparison in the 'traditional' manner, that of a high performance C program that is accessed via an interface in a high level language. IPOPT is a general non-linear problem solver and was included to demonstrate how the specialisation of RAM is able to provide good performacne compared to a more general solver.

Note that both comparison solvers, as well as RAM itself, are being accessed through the JuMP interface. The three comparisons are being operated 'as is', with no extra configuration steps being taken in JuMP. The benchmarking suite records the time taken for JuMP to build the problem separately from the solving time

8.1.1 Solution Correctness

While both the single-threaded and multi-threaded implementations of Hildreth's algorithm do not differ in their mathematical formulation they do show considerably different results in the numbers that they produce. As was discussed in the design of the multi-threaded implementation this is essentially due to the use of a form of almost-cyclic control.

The observed effect of this change for most of the test problems is that the multi-threaded algorithm will very quickly approach the optimal value, but then take a long time to converge to a precise solution. It is also likely that the solution will be approached from an infeasible side of the constraints. This is not interpreted as a failure of the algorithm, as in all tested cases it has been very close to the solution, and does approach in an asymptotic manner.

As was covered in the background discussion of Hildreth's algorithm, it makes use of repeated projections onto the constraints of the problem, and as Figure 1 illustrates, having constraints with a small angle between them can result in poor convergence. The problem CONT-100 has been included as an example of a problem with poor performance, likely due to such constraint properties.

CONT-100 is a very good example of how the algorithm modification required for multi-threading can improve performance. When testing on the original algorithm 1000 iterations gives an objective cost of 349985 (for a known optimum value of -4.699), but just 100 iterations with the modified algorithm gives a cost of -4.76. This is not a feasible solution, and further iterations show it slowly converging to the known optimum (e.g. it reaches a cost of -4.72 after 1000 iterations). However for many problems an infeasible solution very close to a feasible optimum is much preferable to a feasible solution that is far from the optimum.

8.1.2 Algorithm Performance

Table 3 illustrates the performance differences observed between RAM, OSQP, and IPOPT. As RAM has two distinct stages that each take time they have each been presented. The reason for this is that the build time is not neccessarily a cost that must be undertaken every time the algorithm is run. For instance, if the limits of a constraint are changed the full build doesn't need to be run again. However the optimisation time is needed for each run.

The results show that RAM runs quickly when considering small problems, *HS21* and *HS118*, compared to OSQP and IPOPT. However this performance advantage falls away when considering larger problems.

Test	RAM Build	RAM Optimisation	OSQP	IPOPT
HS21	9.799×10^{-5}	2.503×10^{-5}	1.619×10^{-3}	1.071×10^{-2}
HS118	2.533×10^{-4}	3.749×10^{-4}	1.122×10^{-3}	1.338×10^{-2}
LISWET1	19.513	38.616	1.245×10^{-1}	3.991
LISWET2	19.663	39.616	1.239×10^{-1}	3.433
AUG2DC	69.257	2.891	1.424×10^{-1}	1.271
CONT-100	21.227	799.542	4.980×10^{-1}	1.672

Table 3: Performance of Single Threaded RAM

However, running the same tests when using the multi-threaded implementation of Hildreth's algorithm shows a marked improvement in performance. This scaling itself is shown in the following section, the results here show the best performance that RAM provided.

Note that the recorded time for *CONT-100* is the time taken to reach 10000 iterations, at which point the algorithm was ended. This is due to exceptionally slow performance due to problem being ill-suited to the algorithm, as was previously discussed during the coverage of result accuracy. All other times are taken after the result converges to the reference result given by OSQP.

As was just covered when discussing the accuracy of the solution, the multithreaded implementation shows very fast convergence to an approximately correct solution, and then slow convergence towards a feasible optimum. Table 4 shows the time requires to converge to a result within 0.1% and 0.01% of that given by by OSQP.

It is clear that for small tests (such as *HS21*) the addition of threading significantly impacts the optimisation speed. This is due to each iteration only requiring a handful of row calculations, therefore the overhead of distributing operations to threads is more than any savings made. However larger tests (such as *LISWET1*) have a huge speedup. This is due to a combination of less iterations being required to arrive at a solution, as well as each iteration being faster.

Test	RAM Optimisation		OSQP	IPOPT
	0.1%	0.01%		
HS21	8.798×10^{-5}	8.798×10^{-2}	1.619×10^{-3}	1.071×10^{-2}
LISWET1	0.1024	0.1256	0.1245	3.991
LISWET2	0.1151	0.1265	0.1239	3.433
AUG2DC	1.963×10^{-2}	1.963×10^{-2}	0.1424	1.271

Table 4: Performance of RAM with 35 Threads (measured in seconds)

As before, *CONT-100* is a very difficult problem for RAM to solve and does not come within the boundaries given for the results of the other problems. However, the result provided by the threaded algorithm is much closer to the known optimal value, with an average result within 1%.

Finally, to confirm that the runtime scales linearly with the number of iterations, Figure 6 shows the predicted linear relationship.

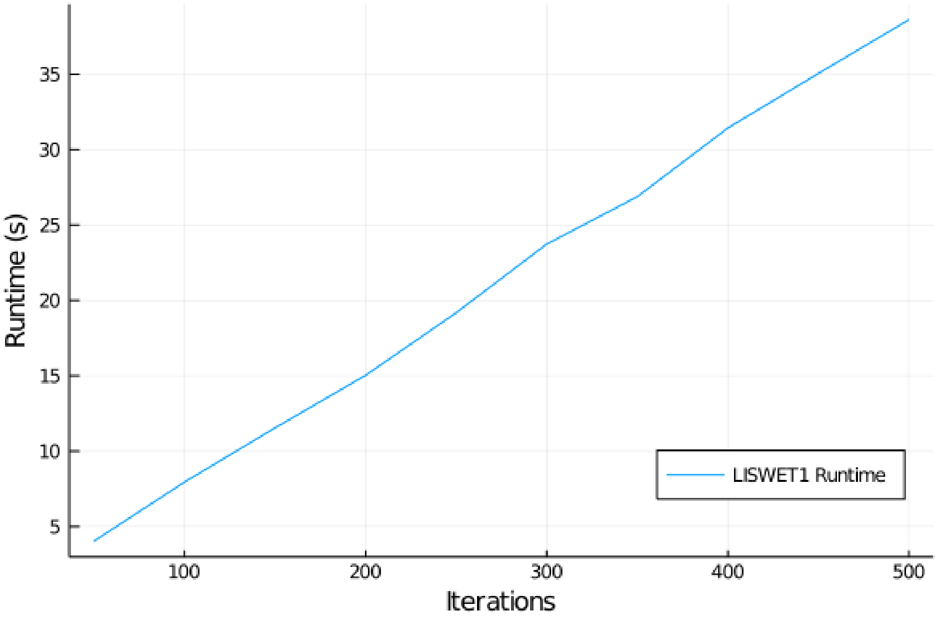

Figure 6: *LISWET1* iteration scaling performance

8.1.3 Algorithm Scaling

One of the requirements of RAM was to illustrate its scaling with number of processors, ideally with a linear relationship. It was expected that any advantage would only be shown in larger tests, and this is shown by the result in Figure 7, where *HS21* shows a broadly linear decrease in performance as more threads are made available to it. As *HS21* has only a single constraint it is not possible to distribute operations over all threads, meaning that additional threads just increase the overhead cost without giving any performance benefit.

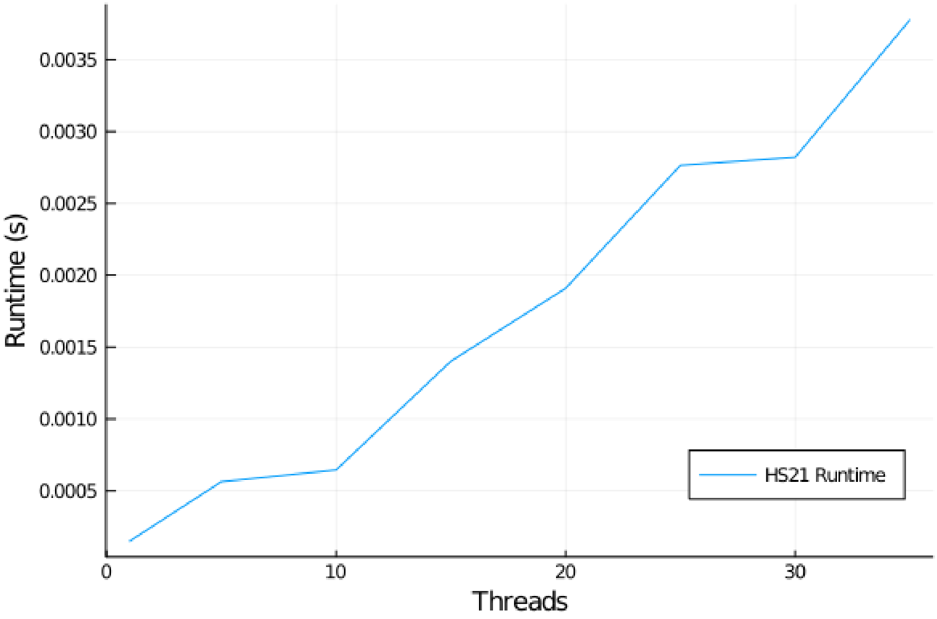

Figure 7: *HS21* thread scaling performance

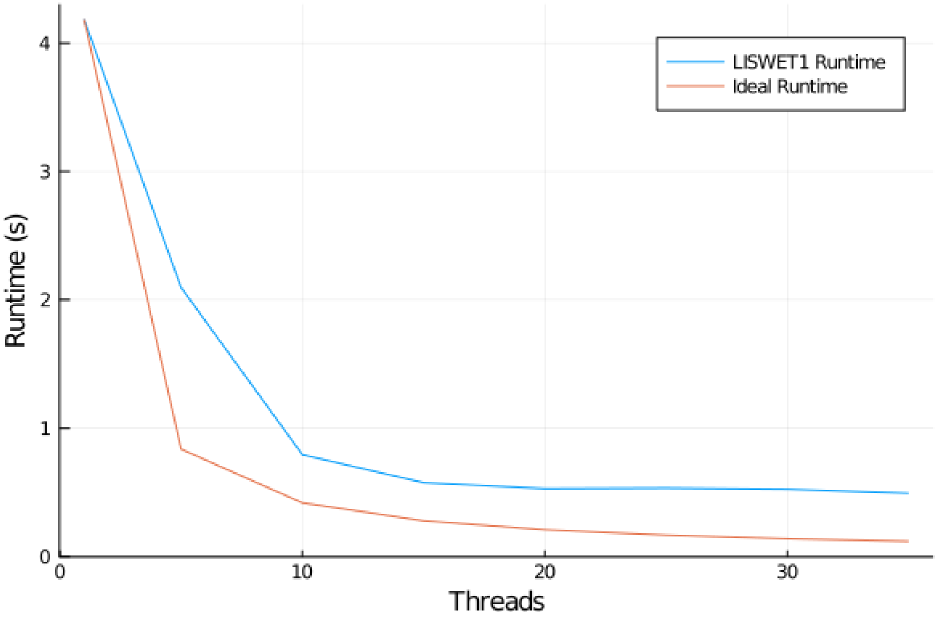

Figure 8: *LISWET1* thread scaling performance

However, when testing larger problems scaling close to the expected result is observed. Figure 8 shows the performance variation as the number of threads is increased for *LISWET1*. Also pictured is the ideal scaling. Putting this on a log-log plot, Figure 9, demonstrates that while additional processors create diminishing returns, good scaling is still present.

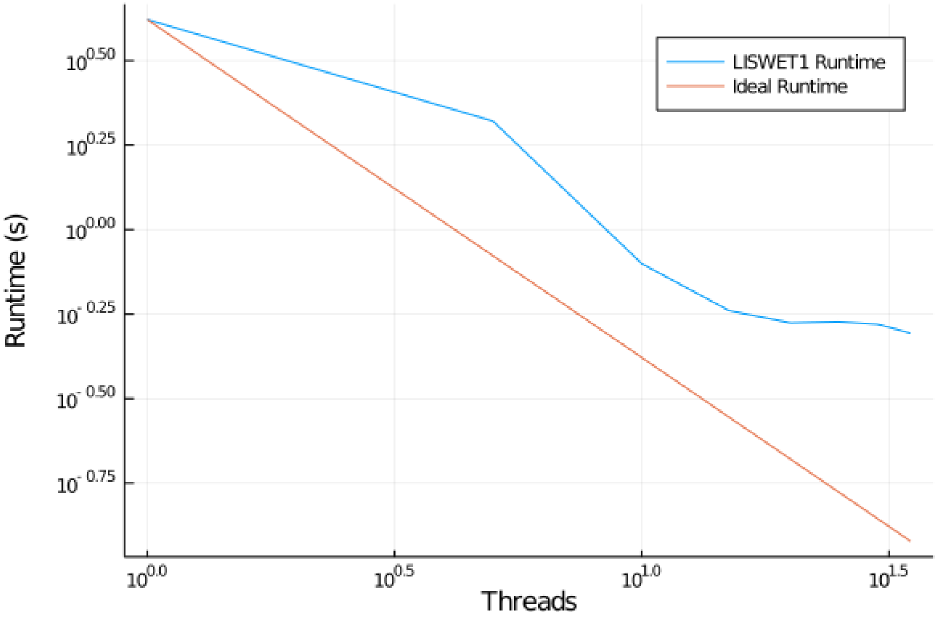

Figure 9: *LISWET1* thread scaling performance with log scaling

The exact cause of the diminishing returns requires further investigation, but it is likely that it is related to overheads with distributing jobs to the threads.

8.2 DirectSearch.jl

LTMADS, OrthoMADS, and NOMAD have all been run on the tests described in the previous section. Table 5 gives the number of objective function evaluations, as well as the objective cost. Table 6 illustrates how multiple runs of LTMADS can offer different results. Table 7 compares the total runtime of each of the three algorithms.

The results for LTMADS were achieved after 30 runs of the algorithm. Note that the mean value in Table 5 excludes any LTMADS runs that take an extremely high or low number of function evaluations, or that provide an extremely good or bad cost. Table 6 gives the true mean values, along with the best and worst results.

The results of *ARWHEAD* and *BROWNAL100* have been omitted for NOMAD due to the algorithm taking too long to converge.

Each DS trial has been configured to stop running after 5 iterations have passed without an improved point being found. This particular strategy results in a large number of objective function calls that could be avoided with a less lenient stopping policy. For example, *BIGGS6* is shown with a mean of 36795 function evaluation for LTMADS to get a mean cost of 2.923×10^{-4}. If a less accurate result is needed, costs in the range 2.8×10^{-3} can be obtained with less than 10000 evaluations. This pattern is relatively common throughout each of the tests.

Test	DS LTMADS (mean)		DS OrthoMADS		NOMAD	
	Evals	Value	Evals	Value	Eval	Value
ARWHEAD	492	0.0	500	0.0	-	-
BDQRTIC	177277	378.769	129859	378.769	22547	378.769
BIGGS6	36795	2.923×10^{-4}	80280	2.2388×10^{-4}	1782	5.655×10^{-3}
BROWNAL10	4140	9.739×10^{-5}	19709	9.977×10^{-5}	2568	0.0
BROWNAL100	1190481	2.015×10^{-3}	1026864	56.111	-	-

Table 5: DirectSearch test comparison

A good result is able to be found for all tests, with the exception of OrthoMADS on *BROWNAL* with 100 decision varaibles. If this test is rerun with an allowance for a larger amount of failed iterations, the found optimum is in the range 0.001. This shows that it still converges, but just takes a longer period of time.

Test	Function Evaluations			Function Cost		
	Worst	Mean	Best	Worst	Mean	Best
ARWHEAD	976	492	25	0.0	0.0	0.0
BDQRTIC	238019	177277	160080	378.769	378.769	378.769
BIGGS6	87662	38490	5125	0.243	0.00840	9.507×10^{-5}
BROWNAL10	19709	4641	1431	9.999×10^{-5}	9.739×10^{-5}	7.558×10^{-5}
BROWNAL100	1350053	1190481	179129	8.406×10^{-3}	2.016×10^{-3}	9.854×10^{-5}

Table 6: LTMADS test variation

The timing results shown for NOMAD imply that there may be a problem with its configuration. Testing with multiple different installations of NOMAD on different computers all showed similar timing results, so further investigation is needed to determine the cause of the slow performance.

Test	Runtime (s)		
	DS LTMADS (mean)	DS OrthoMADS	NOMAD
ARWHEAD	0.8112	0.7148	-
BDQRTIC	7.1182	4.9134	10217
BIGGS6	2.2779	2.3845	23.921
BROWNAL10	0.1455	0.3991	36.949
BROWNAL100	60.723	59.298	-

Table 7: Test Runtime Comparison

It may be possible that NOMAD has difficulties with very large problems; this would show an advantage of DS, which (as far as has been tested) is limited only by the amount of available memory.

Assuming that the timing results are characteristic of NOMAD, the comparison it provides to DS is very interesting. NOMAD takes much longer to complete an optimisation process, but needs far fewer function evaluations. As both packages had the same overhead for sampling the objective functions, the most obvious conclusion is that NOMAD itself introduces additional heuristic computations on top of the basic MADS algorithms. These computations appear to allow it to require fewer evaluations, at the cost of runtime.

This balance would be in the favour of NOMAD for a very expensive objective function, but for problems of the complexity illustrated here, the complexity is much more favoured towards DS.

An additional possibility for the large difference in the number of function evaluations is that, as was previously discussed, the cache implementation in DS requires an exact match in order to register a hit. As the package uses floating point numbers, there is a high probably that two numbers that are equally valued will not appear to the be the same to the cache. As NOMAD's cache is much more robust to this, this could be a significant contributor to the number of evaluations.

9 Conclusions and Further Work

9.1 Summary

This project has seen the design, implementation, and testing of a pair of packages. These packages do not just implement a single algorithm, but are designed from the ground up to be highly extensible, and very easy to understand. This will facilitate their future use in prototyping, and experimenting with algorithms in a framework that provides confidence to the design process.

The performance overheads added by these packages is minimal (and less than originally envisaged), meaning that they could very easily be adapted for use in a production environment, despite this not being an original goal of the project.

9.2 Evaluation

The requirements of each of the packages can broadly be divided into two types. The first of these kinds of requirements are those related to performance. Section 7 illustrated that the performance requirements of the packages have generally been met, with a few caveats.

For example, the build-times shown when testing RAM are incredibly detrimental to the overall utility of the package. However this was expected due to the use of the dual formulation. Additionally, DS requires significantly more objective function evaluations than its comparison package, although it retains good performance.

However, excluding the build time of Hildreth's algorithm and the large number of evaluations, the performance of each algorithm compares very favourably against the competition. This will only be improved in future as the code becomes more optimised, and more efficient algorithms are implemented.

Additionally, the goals of designing a simple set of APIs for the implementation of future algorithms have been met. This will hopefully be supported in future as more algorithms as integrated into the package, but the discussions in Sections 5 and 6 should have demonstrated how this design requirement has been met.

9.3 Future Work

Throughout the discussion on the design and testing of RAM and DS several points where designs could be extended or improved have been noted. This section will close the report by giving an overview of the next steps each of the packages can take.

A notable point where RAM does not meet the project requirements is a lack of up-to-date unit tests and documentation. It was intended to implement these after a major refactoring partway through the project, however it was continuously pushed back and never finished. As the project has now concluded, time is available will be taken to ensure that RAM is brought up to the same standard as DS. This opportunity will also be used to ensure that the code is brought to a high standard across both packages and is in an appropriate state for future maintainers to use.

Some future improvements the packages would benefit from, apart from the updates just covered, are as follows.

For RAM, the first major update that should be made is the addition of constraint modification. This feature was actually present in prototype throughout much of the development of the package, however a late redesign of the storage of the constraints made this incompatible. Including this again, in a manner that allows for problems to not be fully rebuilt would be a strong improvement to the package.

Additionally for RAM, finding a solution to the problems discussed in 5.3.1 related to the storage of a sparse factorisation of the problem variable would lead to reduced memory utilisation and improved build times.

For DS the improvements to the cache discussed in 6.8 would be a very useful addition, possibly integrating the constraint cache into it. The cache would also benefit from memory management tools, limiting the maximum size it can grow to.

Finally, DS includes a small section for reporting statistics and information about the solve that it is performing (this was not discussed due to it being in a very basic/prototype form). Formalising the design of this subsection of the program, along with the ability to store and load configurations (including preloading cache contents) would greatly benefit the utility of the package.

10 Appendix

10.1 Acronym Reference

RAM	*RowActionMethods.jl*
DS	*DirectSearch.jl*
MADS	Mesh Adaptive Direct Search
GPS	General Pattern Search
LTMADS	Lower Triangular MADS
OrthoMADS	Orthogonal MADS
MADS-EB	MADS Extreme Barrier
MADS-PB	MADS Progressive Barrier
MOI	*MathOptInterface.jl*
JuMP	*Julia for Mathematical Programming*
API	Application Programming Interface

10.2 Code Reference

At the time of submission the code for *RowActionMethods.jl* is available at `https://github.com/ImperialCollegeLondon/RowActionMethods.jl/` and the code for *DirectSearch.jl* is available at `https://github.com/EdwardStables/DirectSearch.jl`.

Each package either has or will shortly have documentation and examples of their use.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[2] “Julia Microbenchmarks.” [Online]. Available: https://julialang.org/benchmarks/

[3] “ROW-ACTION METHODS FOR HUGE AND SPARSE SYSTEMS AND THEIR APPLICATIONS* YAIR CENSORS,” Tech. Rep. [Online]. Available: http://www.siam.org/journals/ojsa.php

[4] J. Liu, S. J. Wright, and S. Sridhar, “An Asynchronous Parallel Randomized Kaczmarz Algorithm,” 1 2014. [Online]. Available: http://arxiv.org/abs/1401.4780

[5] “Directional direct-search methods,” Tech. Rep., 2008. [Online]. Available: http://www.siam.org/journals/ojsa.php

[6] C. Audet and J. E. Dennis, “Mesh adaptive direct search algorithms for constrained optimization,” *SIAM Journal on Optimization*, vol. 17, no. 1, pp. 188–217, 2007.

[7] S. Kaczmarz, “Angenaherte Auflösung von Systemen linearer Gleichungen,” *Bulletin International de l’Académie Polonaise des Sciences et des Lettres. Classe des Sciences Mathématiques et Naturelles. Série A, Sciences Mathématiques,*, pp. 355–357, 1937.

[8] C. Hildreth, “A QUADRATIC PROGRAMMING PROCEDURE,” Tech. Rep.

[9] A. Lents, “EXTENSIONS OF HILDRETH’S ROW-ACTION METHOD FOR QUADRATIC PROGRAMMING*,” Tech. Rep. 4, 1980. [Online]. Available: http://www.siam.org/journals/ojsa.php

[10] V. Torczon and S. J. Optim, “ON THE CONVERGENCE OF PATTERN SEARCH ALGORITHMS *,” Tech. Rep. 1, 1997. [Online]. Available: http://www.siam.org/journals/ojsa.php

[11] M. A. Abramson, C. Audet, J. E. Dennis, and S. Le Digabel, “Orthomads: A deterministic MADS instance with orthogonal direct ions,” *SIAM Journal on Optimization*, vol. 20, no. 2, pp. 948–966, 2009.

[12] C. Audet and J. E. Dennis, “A progressive barrier for derivative-free nonlinear programming,” *SIAM Journal on Optimization*, vol. 20, no. 1, pp. 445–472, 2009.

[13] J. H. Halton, “On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals,” Tech. Rep.

[14] A. S. Householder, “Unitary Triangularization of a Nonsymmetric Matrix*,” Tech. Rep.

[15] “JuliaOpt: Optimization packages for the Julia language.” [Online]. Available: https://www.juliaopt.org/

[16] “JuliaSmoothOptimizers.” [Online]. Available: https://juliasmoothoptimizers.github.io/

[17] “Introduction · JuMP.” [Online]. Available: https://www.juliaopt.org/JuMP.jl/v0.19.0/

[18] I. Dunning, J. Huchette, and M. Lubin, “JuMP: A modeling language for mathematical optimization,” *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.

[19] “Manual · MathOptInterface.” [Online]. Available: https://www.juliaopt.org/MathOptInterface.jl/dev/apimanual/#Manual-1

[20] "JuliaOpt/GLPK.jl: GLPK wrapper module for Julia." [Online]. Available: https://github.com/JuliaOpt/GLPK.jl

[21] P. C. Hansen and J. S. Jørgensen, "AIR Tools II: algebraic iterative reconstruction methods, improved implementation," *Numerical Algorithms*, vol. 79, no. 1, pp. 107–137, 9 2018.

[22] S. Le Digabel, "Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm," *ACM Transactions on Mathematical Software*, vol. 37, no. 4, 2 2011.

[23] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An Operator Splitting Solver for Quadratic Programs," 11 2017. [Online]. Available: http://arxiv.org/abs/1711.08013

[24] N. Chiang, C. G. Petra, and V. M. Zavala, "Structured Nonconvex Optimization of Large-Scale Energy Systems Using PIPS-NLP," Tech. Rep.

[25] "Julia Style Guide." [Online]. Available: https://docs.julialang.org/en/v1/manual/style-guide/index.html

[26] "Documenter.jl Package." [Online]. Available: https://github.com/JuliaDocs/Documenter.jl

[27] "Travis." [Online]. Available: https://travis-ci.com/

[28] "GitHub." [Online]. Available: https://github.com/

[29] "GitHub Pages." [Online]. Available: https://pages.github.com/

[30] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 5 2006.

[31] "MPI Forum." [Online]. Available: https://www.mpi-forum.org/

[32] "PyCUTEst." [Online]. Available: https://jfowkes.github.io/pycutest/_build/html/index.html

[33] "Julia PyCall." [Online]. Available: https://github.com/JuliaPy/PyCall.jl

[34] "A Repository of Convex Quadratic Programming Problems * IstvánIstv´István Maros † and Csaba M ´ eszárosesz´eszáros ‡," Tech. Rep., 1997.

[35] "NOMAD.jl Repository." [Online]. Available: https://github.com/bbopt/NOMAD.jl