Static Optimisation in the Julia Programming Language

Interim Report

Edward Stables

CID: 01220379

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 not problems for the industrial use of this software, it can be problematic in an academic 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 by implementing several optimisation software packages for the Julia programming language which are designed to provide a powerful API for implementing new algorithms, as well as making it simple to modify and experiment with existing algorithms. Julia is a uniquely designed language that allows for this goal to be approached in a very different way than would be possible in more traditional languages, allowing for a simple design while retaining high performance.

Julia has recently begun gaining traction in the numerical programming community due to its combination of powerful syntax, performance, and utility. It is commonly compared to the languages MATLAB, Python, and C, combining aspects of all three to create a very interesting new platform for programming in this space.

MATLAB is first and foremost designed as a mathematical language, limiting its utility in general purpose use-cases (as well as being limited by licensing). Python is famous for its speed of development and expressive syntax, but has slow performance and few restrictions on types, leading to hidden bugs. Finally, C remains a very high performance language, but struggles to provide an environment that allows for rapid implementation and extension of algorithms created with it. Julia takes each of these problems and provides a language with an expressive syntax focused on mathematical programming, a very powerful typing system, and performance that can near that of native C code in many circumstances. This flexibility has led to Julia being described as a solution to the 'two language problem' by its authors, allowing both algorithm development and release software to be the same code-base [1].

In this project we intend to utilise Julia to develop a set of native optimisation packages that are specifically designed to allow for extension and modification as needed. Contributions to the packages should be possible without requiring a detailed knowledge of the software architecture and the 'tricks' needed for gaining high performance. The initial project plan identified a set of algorithms that are not currently available as native Julia code. The main aim of the project is to implement these algorithms within a structured framework that allows for similar algorithms to be added to the package easily. Each of the selected algorithms is representative of a different kind of solver (for example, row action methods, or direct search methods, discussed further in Section 3), meaning that the collection of packages produced by the project will allow for simple implementation of a wide range of future algorithms.

2 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 [2], OSQP [3], and PIPS-NLP [4] algorithms respectively. In addition, an implementation of Hildreth's Algorithm [5] would be included to provide an implementation of the class of algorithms known as row-action methods. The specifics of these algorithms will be discussed further in Section 3.

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 this to be added with very minimal changes. This feature of Julia, and the design patterns it enables, is discussed further in Section 3.1. As such we decided to transition the aim of the current sub-project to design a framework in which any row-action method (the class of method to which Hildreth's algorithm belongs) 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.

At the time of writing, details of the OSQP and PIPS-NLP sub-projects have not been discussed in detail and it is not decided whether they will be designed as single solvers or frameworks for a broader class of optimisation algorithms, although it is likely that they will be more focused on the specific algorithms, with the ability to heavily customise specific aspects of the algorithms.

As previously remarked, the aim or the project is not to develop algorithms of a higher performance than existing solvers. However, well written Julia code is able to offer very high performance [6] with appropriate tuning [7]. While we will prefer simple design and readable code over minor performance improvements, the design will attempt to achieve the highest performance reasonable.

Aside from the implementation of the algorithms/frameworks themselves, it is an important requirement to ensure compatibility with the existing Julia optimisation ecosystem. The vast majority of Julia optimisation code implements a low-level API called MathOptInterface (MOI) [8] which allows for abstracted use of high level problem description packages without needing to know the APIs of individual solvers. This allows us to apply the same problem to two optimisation solvers and compare the results without needing to formulate the problem in two different ways for each of the solvers.

As this is a project that will be released on an open-source license and will continue to have contributions to it in the future, it is also a requirement to ensure good software development practices are maintained. Employing good practices will ensure that the code is maintainable by a wide group of people, and has a level of ensured reliability. The two key aspects of this are documentation, and testing. The specifics of which will be discussed in Section 4.

3 Background

A reference for the acronyms used in this report is presented in Table 1.

Acronym	Meaning	Notes
MOI	MathOptInterface	A Julia package that provides a low level optimisation problem description interface [8].
JuMP	Julia Mathematical Programming	A Julia package that provides a high level optimisation problem description interface [9].
RAM	RowActionMethods	A package developed in this project to contain solvers in the class of row action methods.
DSM	DirectSearchMethods	A package to be developed in this project to contain solvers in the class of direct search methods.
API	Application Programming Interface	A term to refer to the interface a piece of software provides for its use.

Table 1: Acronym Reference

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 [10].

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 most appropriate being called for a given set of arguments. This gives the ability to design software in a very generic way, with the ability 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 Figure 1. The function `range` has three definitions, one which takes an argument of two `Int` types, one with three `Int` arguments, and the final with two `Int`s and one `Float64`. Each method returns a range of values in a list inclusive of the two provided limits, but the exact implementation and returned data differs between each.

This is heavily used within the Julia source code, for example, the addition function '`+`' has 224 different implementations (at the time of writing), as different definitions of addition are applicable for different combinations of argument types. This illustrates another strength of Julia's use of multiple dispatch, these default functions can be safely overridden for certain types.

```
function range(a::Int, b::Int)::Vector{Int}
    range_result = []
    for i = a:b
        append!(range_result, i)
    end
    return range_result
end

function range(a::Int, b::Int, step::Int)::Vector{Int}
    iterator = a
    range_result = []
    while iterator <= b
        append!(range_result, iterator)
        iterator += step
    end
    return range_result
end

function range(a::Int, b::Int, step::Float64)::Vector{Float64}
    return [v for v = a:step:b]
end
```

Figure 1: Three Range Implementations for Different Type Signatures

In this project multiple dispatch is used in multiple ways, with the three main examples being:

- Framework extensibility. Each algorithm within a framework is able to define its own set of functions within a common naming scheme, simplifying the internal framework design, and allowing for a consistent design between algorithms.
- Default value calculation. In some cases where a user doesn't enter values in a function call, multiple dispatch is used to first call function to calculate default values before returning to the main call. This results in less branching operations from determining the default values in the main function.
- Stopping condition management. This is discussed in detail in Section 4.3.2. In brief, all conditions which a solver can be stopped for are given their own implementation of a common function, allowing the framework to evaluate stopping conditions that have been implemented in a custom manner.

3.1.2 Type System

Julia has an incredibly rich typing system which integrates very well with the multiple dispatch system. A simple example of types are shown in Figure 1, showing the types of arguments after their definition, and the function's return type after the function declaration. The type system is arranged in an inheritance based structure, with each type tracing its inheritance back to the top-level abstract base class `Any`.

Multiple dispatch makes use of this type hierarchy extensively. If the call `range(1, 4)` was made

to the code in Figure 1 Julia would recognise that the types of the arguments are both `Int64`. The type `Int64` is a child type of `Int`, meaning that it is valid to pass an `Int64` to any place `Int` is defined. Therefore the first function is called.

Alternatively, if the call `range(1.5, 4, 3.1)` was made, Julia recognises that the arguments have types `Float64`, `Int64`, and `Float64` respectively. None of the defined functions have a type signature that fits this, and an error is thrown.

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 [11], and JuliaSmoothOptimizers [12], 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.jl [13][9] and MathOptInterface.jl [8]. These last two packages are most relevant to this project.

3.2.1 MathOptInterface

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 the implementation of a MOI wrapper can be found in Section 4.3.3.

It is possible to define optimisation problems with the syntax offered by MOI, the objective function of the problem,

$$\begin{aligned} \max_{x,y} \; & 2x + 7y, \\ -5 \leq\; & x \leq 2, \\ & y \leq 30, \\ 2x + 8y \geq\; & 3, \end{aligned} \tag{1}$$

may be defined with MOI as,

```
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)
```

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, JuMP can be used to interface with MOI, which then translates the problem into a format used by solvers. JuMP operates at a much higher level than MOI, giving the freedom to input expressions directly.

```julia
using JuMP
using GLPK

#Define an empty model from GLPK
model = Model(with_optimizer(GLPK.Optimizer))

#Define the problem variables and their ranges
@variable(model, -5 <= x <= 2)
@variable(model, y <= 30)

#Set the objective function
@objective(model, Max, 2x + 7y)

#Set a constraint
@constraint(model, 2x + 8y >= 3.0)

#Optimize the model
JuMP.optimize!(model)

#Print the values of the objective and each variable
println(JuMP.objective_value(model)) # = 214
println(JuMP.value(x))               # =   2
println(JuMP.value(y))               # =  30
```

Figure 2: Demonstrating solving a simple optimisation problem in JuMP

The entire problem shown in ~~Equation~~ (1) can be formulated, solved, and queried in JuMP as shown in Figure 2 (using the GLPK solver for Julia [14]). What took five lines in MOI is accomplished in one with JuMP using the line `@objective(model, Max, 2x + 7y)`. This expression is directly translated into the MOI expression, but in a layer hidden from the user.

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. While each package in this project is given its own native API (very useful for testing and early development), it will also implement a MOI interface, letting the solver be used like any other optimisation package in Julia.

3.3 Hildreth's Algorithm

Hildreth's algorithm was originally put forward in this project as an introductory topic for becoming more familiar with Julia. As has been previously discussed, this aspect of the project has been expanded into developing a framework for implementing general row-action methods. However this section will focus on Hildreth's original algorithm, and the structures within it that generalise to the class of row action methods.

3.3.1 Row Action Methods

Hildreth's algorithm is an early example of what is now known as a row action method, and was first put forwards by Hildreth in 1957 [5]. Since this time other row-action methods have been

developed, with the algorithms showing good performance on huge, sparse matrices which have no detectable structure. A collection of such methods is put forwards by Censor in his 1981 paper [15]. Censor defines row-action methods as methods that:

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

This class of algorithms is not presented as a concrete set, but rather a broad category that algorithms may be adapted to meet the criteria of. Hildreth's algorithm was not originally created as a row-action method (being first proposed before the concept of these methods was defined), however the algorithm lends itself very well to this framework.

3.3.2 Algorithm

The core algorithm presented by Hildreth solves the quadratic problem,

$$\min_x \frac{1}{2}x^T Cx + d^T x \text{ s.t. } Gx \geq h, \tag{2}$$

where C is a positive definite $n \times n$ matrix, G is an $m \times n$ matrix, and $x, d \in \mathbf{R}^n$. This is reformulated into the dual problem,

$$\min_z z^T Az + b^T z \text{ s.t. } z \geq 0, \tag{3}$$

where,

$$A = -\frac{1}{4}GC^{-1}G^T, \tag{4}$$

$$b^T = \frac{1}{2}d^T C^{-1}G^T + h^T. \tag{5}$$

Each iteration of the algorithm performs the following operations, where p is the iteration counter, a_{pq} denotes the (p,q)th element of A, i iterates through the range $(1, m)$ (m being the number of constraints),

$$z_i^{(p+1)} = \max(0, w_i^{(p+1)}), \tag{6}$$

$$w_i^{(p+1)} = -\frac{1}{a_{ii}}\left(\sum_{j=1}^{i-1} a_{ij}z_j^{(p+1)} + \sum_{j=i+1}^{m} a_{ij}z_j^{(p)} + \frac{b_i}{2}\right). \tag{7}$$

Equation 7 shows that each iteration updates the dual variable by considering every row in turn, with the decided values based solely off the values in the current row. The maximum of w_i and 0 is selected to ensure that the constraint of $z \geq 0$ is met. In general the initial value of z can be any value in the feasible range.

The core operation this algorithm performs is the orthogonal projection of the current point onto the constraint considered by the current row. Geometric interpretations of this operation are provided in both [16] and [15].

3.4 MADS Algorithm

The Mesh Adaptive Direct Search (MADS) algorithm [2] was first presented as an improvement on the Generalised Pattern Search (GPS) class of algorithms [17]. GPS is now defined as a special case of MADS. This class of algorithm is derivative free, making it particularly suitable in cases where the objective function and constraints are ill-defined, or expensive to compute. MADS is also suited to non-smooth functions.

MADS describes a general class of algorithm, and several variations and implementations have been made, for example the OrthoMADS algorithm [18], or the NOMAD package [19]. NOMAD is the main standalone implementation of MADS and is implemented in C++, with Python and MATLAB interfaces also available. A simple example of MADS given in the original paper, named LTMADS, will be the initial algorithm developed within the MADS package for this project.

As is implied by its name, MADS defines a mesh structure of points over which it searches for a minimisation the target function. This mesh is not a set of points that is actually defined and stored, rather the algorithm stores the current minimum point (the incumbent), and step distance parameters. From these parameters, the next set of admissible points on the mesh are found and evaluated. If a new minimum is found then the incumbent point is updated and the step distance is increased (with the intention being that a coarser movement will give faster convergence). If no new minimum is present then the step distance is decreased, giving a finer movement.

MADS defines a set of requirements for the selection of directions to poll, and the update rules of the step distances, but algorithm designers have a large degree of freedom in the exact implementation of each part of the algorithm, and will still fall within the convergence results of MADS.

3.5 PIPS-NLP & OSQP Algorithms

Work on the implementation of these algorithms has been scheduled for a later stage in the project and background research on the algorithms has not been completed to the same degree as Hildreth's Algorithm or MADS. Therefore this section contains just a very brief summary of the algorithms.

PIPS-NLP is a filter line-search interior-point algorithm that provides a NLP (Non Linear Programming) extension to PIPS (Parallel Interior Point Solver) [4]. Interior point methods operate by describing the problem in terms of penalty functions (named barrier functions) that have their parameters varied on each iteration until an optimum is achieved. Solvers based on interior point methods are very common in commercial software, with the method showing strong performance on a range of problems.

OSQP (Operator Splitting Quadratic Program) [3] is a more novel approach to the optimisation of quadratic problems, with the intention of providing few restrictions on the kind of constraints and the conditioning of problems it is given. The algorithm is also designed to allow for use in an embedded environment, for example, the algorithm may be configured to allow for no division operations to be done after the initial setup of variables.

4 Implementation

4.1 Initial Project Plan

After deciding the solvers that this project would focus on (Hildreth, MADS, PIPS-NLP, and OSQP), and doing research on the additional features that would be needed for this to be successfully implemented (specifically, implementing solver interfaces and benchmarking utilities), the Gantt chart shown in Figure 3 was created. This shows that a period of two months was set aside for each solver, with the reduced time for Hildreth's algorithm due to work already done at that point. This plan also takes into account the time needed for exam preparation, and the lack of progress that would take place during that period.

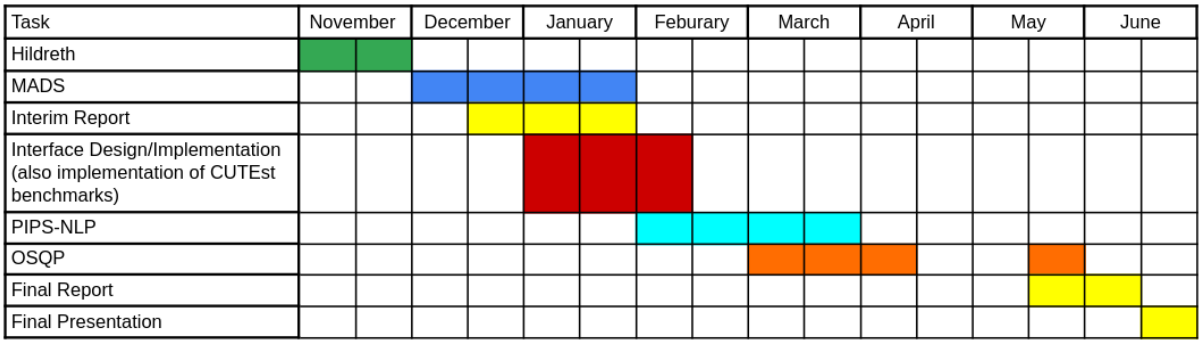

Figure 3: Original Project Plan

Due to several factors, it was not possible to follow this plan exactly during the Autumn term. This was the result of the transition from implementing just Hildreth's algorithm to the implementation of a row-action framework, requiring the refactoring of a large body of code, and extra time needed for design and testing.

In addition, it became clear that to properly evaluate the frameworks's performance and verify solver stability on a variety of problems, it was necessary to run benchmarks on a wide range of problems. As will be discussed further in 5.1 the CUTEst benchmarking software [20] is utilised for this. A Julia package for interfacing with the CUTEst environment is already implemented as part of the JuliaSmoothOptimizers package [21], importing the problems in a format compatible with JuMP. Therefore, loading these problems into the framework for benchmarking requires a MOI wrapper to be implemented. This earlier need for the MOI wrapper shifted this task earlier in the plan, further delaying the start on subsequent tasks.

4.2 Current Solver Design Progress

All of the implementation progress so far has been based on the row action method framework, with the package being named RowActionMethods.jl (RAM). At the time of writing, initial work on the MADS algorithm and its associated package DirectSearchMethods.jl (DSM) is commencing, but has not yet had code implemented, therefore will not be discussed here.

4.2.1 Initial Work

The first version of this package was a standalone implementation of Hildreth's original method, with the code initially adapted from the MATLAB code presented in [22]. This was modified into a form that made more sense for Julia's design, and split into smaller functions that allowed for easier testing.

This process of splitting the code made it clear that the package could be generalised with relatively few design changes. This package design process is discussed further in Section 4.3.

4.2.2 Hildreth's Algorithm Implementation

With the framework of the problem abstracted away from the Hildreth specific implementation it became necessary to redesign the solver to better fit with the requirements of the framework. This takes the form of a data-structure and iteration function pair, shown in Figure 4.

```
mutable struct HildrethModel <: ModelFormulation
    C::Array{Float64}
    d::Vector{Float64}
    G::Array{Float64}
    h::Vector{Float64}
    A::Array{Float64}
    b::Vector{Float64}
    workingvars::Dict{String, Any}
end

function iterate!(model::HildrethModel)
    z =  model.workingvars["z"]
    model.workingvars["z_old"] = copy(z)
    A = model.A
    b = model.b

    for (i,l) in enumerate(z)
        w = (A[i:i,:] * z)[1] - A[i,i] * l
        w += b[i]
        w /= -A[i,i]
        z[i] = max(0, w)
    end

    model.workingvars["iterations"] += 1
end
```

Figure 4: Data structure and iteration function for Hildreth's algorithm (minimal example adapted from the main codebase)

The `struct` HildrethModel describes every variable that is needed for defining and solving the problem. The first four variables (`C`, `d`, `G`, and `h`) describe the QP problem described in Equation 2. The remaining variables act as intermediary variables for the solver to operate on. The generation of the variables `A` and `b` is defined by Equations 4 and 5 respectively. The final entry is a dictionary that allows for variables needed in the solver to be dynamically designed and removed as needed. In most cases it contains the keys `"iterations"`, `"z"`, and `"z_old"`. Respectively, these map to the number of iterations that have been run, the current decision variable, and the decision variable of the previous iteration. It is not seen in Figure 4, but the `"iterations"`, and `"z_old"` entries in `workingvars` are utilised when checking for whether the algorithm has met the conditions for stopping.

The function `iterate!` is the function that calculates one iteration of the algorithm and ap-

plies it to the current problem state. The function is passed a variable named `model` of type `HildrethModel`, the type of the struct previously discussed. Firstly, temporary variables are allocated from those within the model, just to increase readability of the following calculations. Following this, the algorithm iterates over every row in the dual formulation, implementing an equivalent formulation of Hildreth's algorithm [16, 2.7] to that previously discussed in Section 3.3.2.

4.2.3 Extended Hildreth's Implementation

An additional implementation of Hildreth's algorithm is also in the process of being implemented in the framework. This takes into account multiple refinements presented in [16]. The main improvements to the algorithm are the addition of almost-cyclic control and relaxation parameters.

The implementation is very similar to that discussed in Section 4.2.2, with some interesting additions. For example, the almost-cyclic control requires that the row iterations can be controlled by the user, not automated as in Figure 4. A possible solution to this is Julia's feature of treating functions as first class objects. In practice this means that a function is just another variable that can be passed around to other functions. In this case, the user can design a function that is able to generate the list of indexes that an iteration should access, with this function stored in the problem description struct just like any other variable.

As with Hildreth's original method, this algorithm was implemented prior to the addition of the MOI wrapper to the RAM package. However, Hildreth's algorithm was updated alongside the progress of the wrapper for testing purposes, and the extended algorithm was not. This has left it in a state currently inconsistent with the design styles discussed elsewhere in the report, hence its lack of detailed discussion.

4.3 Current Framework Design Progress

In addition to the implementation of the algorithms themselves, a significant amount of effort has been given to the design of the framework. Three distinct parts of the design process are presented here.

4.3.1 API Design

This aspect of the design of the framework is crucial for allowing implemented algorithms to run in a high-performance manner, and yet still be flexible enough to adapt to the majority of requirements an algorithm may present.

Table 2 describes the current API that each solver must implement to be compatible with RAM. The API is not currently complete, as the package is still in development, but the main sections that describe the requirements of RAM are present. The intention is to provide future developers as much freedom as possible when implementing their algorithms. To this end, each function and datatype are quite broad in their definition, with the intention that the developer is able to concentrate on the internal function of their algorithms. Note that the function naming follows the Julia convention of using minimal underscores, unless the name would be otherwise difficult to read, and for using an exclamation point at the end of the name to indicate that the function modifies one of the variables it is passed.

Each algorithm also implements a `struct` that describes their problem in its entirety (for example, the `struct` in Figure 4 is used in the implementation of Hildreth's algorithm). These

Function Name	Description
`iterate!`	Runs one iteration of the solver (an implementation is seen in Figure 4)
`setobjective!`	Sets the passed variables as the objective function of the solver
`setconstraint!`	Adds a constraint to the solver
`buildmodel!`	Generates internal variables from the provided objective and constraints
`is_empty`	Returns a boolean to show if any variables have been set in the model
`resolver!`	Generates the primal decision result value from the current problem variables
`objective_answer`	Returns the minimal objective value
`decision_answer`	Returns the decision variables corresponding to the minimum objective value

Table 2: Current RAM Framework API

types are used to enable multiple dispatch on each of the functions in the API. Each algorithm implements their own version, that takes only the data types they have defined as an argument, and Julia ensures that when the main framework calls the function, only the appropriate solver responds.

```
function iterate_model!(model::ModelFormulation,
                        conditions::StoppingCondition
                       )
    #Returns if unconstrained optimum is valid
    if valid_unconstrained(model)
        set_unconstrained!(model)
        return
    end

    #Run iterations until stop conditions are met
    while !stopcondition(model, conditions)
        iterate!(model)
    end

    #Calculate solution
    resolver!(model)
end
```

Figure 5: Main iteration function used for the framework.

Figure 5 shows the main iteration function within the framework. This is a unique function, unlike those in Table 2, which calls the `iterate!` function of whichever algorithm is appropriate for the `model` variable, with Julia's multiple dispatch neatly assigning the correct function without any manual parsing required.

As can be seen from these examples, Julia allows for an API to be defined, and new functionality to be added without editing any code that is currently in place. Adding another solver simply requires implementing the necessary functions, and including the new files in the same direct-

ory as the rest of the package, with no need to make multiple changes to parsers or decision blocks.

4.3.2 Stopping Conditions

An important aspect of the framework design is the implementation of termination conditions of the solver. As with the rest of the design of the framework, having a system that is high performance, but also highly flexible and extensible is necessary. This is further complicated by the variation that each solver can have, due to the flexibility of the API.

In an effort to implement a simple but robust system of stopping conditions for the framework, influence was taken from a talk given at Juliacon 2019 [23], showcasing a design pattern that makes use of multiple dispatch to implement a high expandable system of condition checking. This approach defines a single function `stopcondition` that is implemented separately for each desired stopping condition. Each stopping condition is given its own type, and then multiple dispatch is able to call the correct checks. This solution is very suitable for this framework, as it places absolutely no responsibility on the framework itself for knowing the internals of each solver, or the stopping conditions.

```
struct SC_Iterations <: StoppingCondition
    value ::Int64
end

function stopcondition(model::ModelFormulation,
                       iterations_limit::SC_Iterations
                      )::Bool
    return get_iterations(model) >= iterations_limit.value
end
```

Figure 6: An example of a stopping condition implemented in the RAM framework

Figure 6 shows the implementation of a possible stopping condition. If a model is optimised with the stopping condition child type `SC_Iterations`, then before each iteration the corresponding `stopcondition` function is run. In this case the function implements a check between the number of iterations of the current model, and the limit of iterations that has been set by the `SC_Iterations` stopping condition type, returning a boolean to indicate if the condition for stopping has been met.

4.3.3 MOI Wrapper

The MOI wrapper is a crucial aspect for the usability of the software. Implementing this wrapper makes the package compatible with JuMP, allowing for very simple definition of problems, as well as automatic reformulation of problems.

'Wrapper' in this context refers to the implementation of an API defined within MOI that allows external software to control the package without knowing anything of its internal design. The design of MOI makes extensive use of multiple dispatch to implement its API, which has the advantage of making it highly flexible for the end user.

An example of this flexibility is when setting objective functions. MOI defines a large range of equation types for objective functions, but most solvers will only be able to use an handful of

these (e.g. Hildreth's algorithm is not able to take a linear function as its objective function). The package designer need only implement the interface that they actively support (the interface for setting a quadratic objective function). MOI is able to see the range of values that a package supports, and either inform the end-user that the solver does not support what they have asked for, or reformulate their request into an equivalent format that the solver does support.

Full compatibility with the MOI interface is quite difficult to ensure, as it requires a very fine degree of control over the internal variables of the solver. For example, the full interface should be able to make modifications to existing constraints and variables by accessing them with an identifier that is returned when they are instantiated. This functionality, while useful, brings a large amount of requirements for the internal structure of the solver that goes against the design of this framework. The same example of modifying existing constraints would require the addition of several new functions to the API in Table 2, reducing the freedom that algorithm designers have.

It is possible that a half-way implementation is performed, where these methods are made accessible to the designers, who made choose to implement them or not. The decision has not been made on this as of yet, as the MOI wrapper itself has only just become functional in a basic manner. It should also be noted that designing a single MOI wrapper around multiple solvers is not within the original design intentions of MOI, and requires careful planning to ensure that functionality is not lost.

4.4 Updated Project Plan

Moving into the Spring term, it is obvious that the initial plan needs to be revised. Having become more experienced with packages like MOI, the amount of work needed for full compatibility is much clearer. MOI allows for a large amount of functionality with only a subset of its API implemented, but as the package is used more often, a larger degree of of MOI's API will be needed. Based on the current level of implementation, it seems likely that a usable implementation of MOI (and therefore also JuMP) will be possible shortly after the submission of this report. This will allow for benchmarking, and a greater degree of testing to be performed.

It can safely be assumed that having a greater number of large testcases will result in issues being found within the algorithm implementations, requiring more time commitment to the RAM package before it reaches a stable state. However, during this debugging time, work can start on implementation of the next package. This next package will be similar to RAM in that it will implement a framework that contains similar algorithms, this time it will be based on the category of Direct Search algorithms, with the main algorithm of interest being MADS, discussed in Section 3.4. It is believed that the experience gained from developing RAM, as well as the larger amount of available time for development, should result in the Direct Line Search package being much faster to implement.

Following this, the PIPS-NLP and OSQP algorithms will be implemented, whether these will be implemented as single algorithms or frameworks of categories of algorithms has not yet been decided. It is most likely that they will not be designed in as general as a way as RAM and DSM are, rather they will be focused on the single underlying algorithm, but designed to allow certain aspects of the solver to be swapped out, to allow for experimenting with different methods for single parts.

An updated Gantt chart of the predicted progress is shown in Figure 7. This plan shows work starting on the Direct Line Search package within two weeks of the submission of this report, and debugging/optimisation work on RAM continuing for another few weeks. Note that time

is dedicated to MOI integration and benchmarking at the end of the implementation period of each solver.

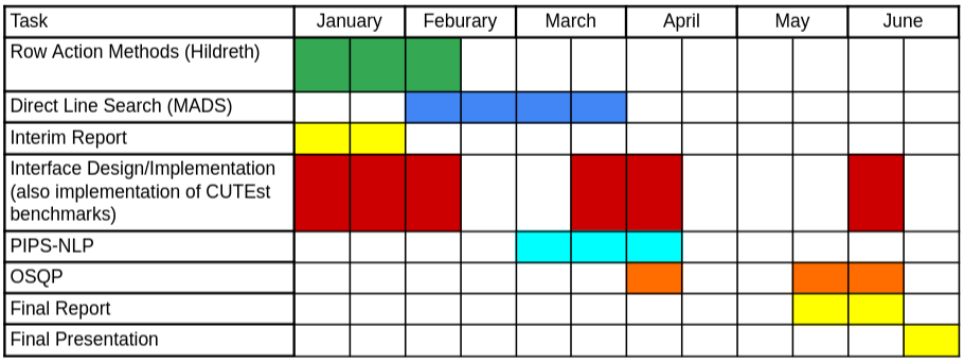

Figure 7: Updated Project Plan

5 Evaluation Plan

As previously stated, the purpose of this project is not to implement algorithms that are able to provide higher performance than existing software. Rather, it is intended that the packages will provide a set of frameworks that can easily be understood, modified, and expanded, while providing performance not significantly lower than existing optimised packages, and faster than implementations in MATLAB or Python.

This aim shows two key goals; high performing packages, and code that can easily be maintained. The first of these is quantifiable with use of benchmarking on a large range of optimisation problems. This will be a crucial part of testing the packages for stability and and correctness, as well as performance evaluation. A widely used set of problems for this purpose is the CUTEst package of problems. The second aim can be accomplished through application of software development techniques such as stringent documentation and unit-testing. These result in code that can be understood easily, and trusted to work as intended.

5.1 Benchmarking

CUTEst provides a very large range of problems of a variety of types, with packages making the problems available in most environments, meaning that the same problems can very easily be used to compare this Julia code against its contemporaries. An important factor in the benchmarking of software is ensuring that the environment code is run in is fair between tests. This requires that tests are performed on the same hardware, with steps taken to ensure that the computer's resources are fully available to the tests.

Ideally tests will also be performed on a variety of different computing platforms to ensure that different hardware doesn't introduce otherwise unknown performance penalties. This also has the potential to identify stability issues on different platforms, for example, some types may not be natively available on some platforms (such as `Int64` variables on a 32-bit computer). In theory Julia's compilation should be able to adjust for this, but this cannot be proven without testing. To this end, testing on a variety of hardware platforms is crucial.

As well as comparing the packages against implementations in other languages, benchmarking will be very useful when optimising the performance of the packages. As the problems presented in CUTEst are representative of large real problems, they will bring up areas of the code that are causing bottlenecks, and make it easy to compare different approaches, choosing the ones that result in the best overall performance.

5.2 Documentation

As the packages developed in this project are intended to continue development in the future, it is crucial that all software is designed in an expandable manner. An important aspect of this is documenting the APIs, features, and requirements of the frameworks to make it clear what the framework supports already, and where it needs to be expanded.

Julia features a system for embedding code documentation within comment blocks, which can be scanned to automatically create formatted documentation. This system removes obstacles that prevent keeping the current documentation up-to-date by placing the documentation right next to the code that is being updated. It should be made a matter of routine that when code is changed in a function, the associated documentation is also edited. This ensures that the code always has an easily available and accurate explanation for its function.

Embedded documentation is very good at providing information on specific parts of the codebase but is poor at giving an overview of package functionality and the intended usage patterns. To this end, a certain amount of documentation discussing the higher level design of the software is needed. This documentation does not need to be as detailed or up-to-date as the embedded documentation, as the concepts they discuss are unlikely to change a great deal except at major releases. These documents will be crucial for ensuring that the package can be properly maintained in future.

The project currently features embedded documentation on over 90% of its functions, as well as several guides on the use of the package, and developing new algorithms. As the project moves towards a more usable state, these guides will need to be expanded to better explain the internal design in a more cohesive manner.

5.3 Versioning

This project has adopted the ubiquitous 'semantic-versioning' scheme, which is the standard system used for versioning in Julia [24]. This details a three-number system of versioning specifying major versions, minor versions, and patches respectively. The expectation is that for a major version number of zero, the software is in an alpha state and any minor version update may introduce breaking changes, but patch changes should not do so. Major versions of one and greater are expected to be official releases, with an expectation of stability and correctness, and that further minor versions are unlikely to introduce breaking changes.

As an example of the timescales spent on each stage of this versioning system (although it of course varies between projects and their scales), JuMP version 0.1.0 was released publicly in October 2013, with steady releases of minor versions roughly every two months initially, with versions being less frequent and larger in scope as the project matured. In early 2020 it has reached version 0.20.1, and yet to announce its first major release. This is a common state in open-source software, and it is unlikely that any of the packages in this project will reach a state of maturity that allows them to take a version of 1.0.0 within the project timescales. However the advantage of having this as an open-source project is that work can continue after the initial project concludes.

As shown by JuMP, it is very possible to have pre-release software that is still high performance and stable. It is intended that this is the state that this project will reach.

5.4 Testing

Rigorous testing of code is crucial for any project that has an expectation of quality and correctness. There are two general forms of testing in use by this project; unit testing and continuous integration.

5.4.1 Unit Testing

A unit test is a series of small tests that are designed to ensure that a function gives the expected results for its main use case, as well as any possible edge cases. As unit tests should be quick to run, they are not able to check every edge case a package might encounter, with the intention being that they provide a 'good enough' assurance of correctness. Ideally every function in software will have some unit-tests implemented, however it is often the case that many functions are only indirectly tested via the unit-tests of other functions.

Julia contains a fairly rich unit-testing framework that can easily be run during development.

Unit tests provide a certain expectation that the majority of the code is correct in most circumstances, assuming that the developer has taken the time to match their changes with updated tests. The code developed in this project currently has a minimal set of unit-tests, with the intention to increase their scope as the design becomes more stable.

5.4.2 Continuous Integration

Continuous integration is the act of ensuring that changes to code are frequently pushed back into the development repository, at which point automation software is able to run a larger set of tests on the codebase. Larger tests are able to take longer, running more edge-cases, benchmarks and providing more trust that the code is correct. These services are also able to ensure that the code is stable for running on multiple platforms, e.g. Linux, Windows, and macOS.

While an over-generalisation, it is useful to think of unit-tests as tests for syntax and function-level correctness, and continuous integration tests as testing the correctness of the overall software design. Continuous integration is also able to provide a detailed analysis of the code. For example, it can be set to run benchmarks showing that the performance of the code has not been decreased by new additions. It is also able to provide reports measuring code coverage, a metric showing that the tests are exercising a total percentage of the codebase. Ideally 100% code coverage can be reached, showing that the entire codebase is being covered when tests are run. Unfortunately writing tests to find every edge case is very difficult, with coverage typically being in the range of 80% to 95%.

5.5 Version Control

With the use of continuous integration, new code is potentially pushed to the main repository multiple times a day. As it would be irresponsible to update the available codebase with potentially broken additions, good use of version control software is necessary. The most common tool used by open-source software (and increasingly so in commercial projects) is Git [25], generally through services like GitHub, GitLab, or BitBucket.

Git allows for a single codebase to maintain multiple development paths running concurrently via branches. A branch can easily be created from any other, allowing a developer to work on a fix or feature without impacting any other code. Once complete, this code can be integrated back into its parent branch, ensuring that any new code is at least somewhat stable. The master branch is the branch that traditionally holds the current release version of the code. Merges of code from development branches back into the master branch is something that needs to be verified with other maintainers of the repository. Github (the git provider of choice for this project) provides a very powerful mechanism for inspecting changes, and how changes would affect the new release in terms of code coverage and any breaking changes. These can be discussed and decided, with controls put in place to ensure that only working code is released.

6 Ethical, Safety, and Legal Planning

6.1 Ethical

As this project is based on software design and the implementation of algorithms, there is no area which is potentially unethical.

But people could use the alg. for evil.

6.2 Safety

As this project is based on software design there is no risk of danger apart from ensuring normal standards for ergonomics are met to avoid injuries such as repetitive strain injury from long periods of information technology use.

6.3 Legal

This project does plan to re-implement algorithms published elsewhere, however this is done without using the source code of these algorithms directly, meaning that only the algorithms themselves are being used. In cases where design inspiration or influence is taken (an example of which is covered in Section 4.3.2), no code is used verbatim, and credit is given to the original designer.

What about licenses! MIT, etc?

7 Conclusions

This project intends to create a software package that has a rather unique focus, and a rather large scope. This report should have given a good overview of the groundwork that has been done for the design and implementation of the codebase for the project, and made a good argument for why this software is necessary and why Julia is the correct language for this to be done in.

Julia stands to be one of the most important languages in numerical computing moving into the coming decade, and for that to come to fruition it needs to have access to as wide a range of software as is available to its contemporaries. Having high-performance, native, and reliable libaries is a crucial step in this process, and it is hoped that this project will play a part in that transition.

References

[1] The Bottom Line, "Julia: A Solution to the Two-Language Programming Problem," 10 2018. [Online]. Available: https://thebottomline.as.ucsb.edu/2018/10/julia-a-solution-to-the-two-language-programming-problem

[2] 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.

[3] 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

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

[5] C. Hildreth, "A QUADRATIC PROGRAMMING PROCEDURE," Tech. Rep.

[6] "Julia Micro-Benchmarks." [Online]. Available: https://julialang.org/benchmarks/

[7] "Performance Tips · The Julia Language." [Online]. Available: https://docs.julialang.org/en/v1/manual/performance-tips/

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

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

[10] 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.

[11] "JuliaOpt: Optimization packages for the Julia language." [Online]. Available: https://www.juliaopt.org/

[12] "JuliaSmoothOptimizers." [Online]. Available: https://juliasmoothoptimizers.github.io/

[13] "Introduction · JuMP." [Online]. Available: https://www.juliaopt.org/JuMP.jl/v0.19.0/

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

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

[16] 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

[17] 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

[18] 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.

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

[20] N. Gould, D. Orban, and P. Toint, "CUTEst: a constrained and unconstrained testing environment with safe threads," Tech. Rep., 2013. [Online]. Available: http://www.fundp.ac.be/

[21] "JuliaSmoothOptimizers/CUTEst.jl: Julia's CUTEst Interface." [Online]. Available: https://github.com/JuliaSmoothOptimizers/CUTEst.jl

[22] Liuping Wang, *Model Predictive Control system Design and Implementation Using MATLAB*, 2009.

[23] "JuliaCon 2019 | Writing Maintainable Julia Code | Scott Haney - YouTube." [Online]. Available: https://www.youtube.com/watch?v=tldyDnjLozo&feature=youtu.be

[24] "Julia's Release Process." [Online]. Available: https://julialang.org/blog/2019/08/release-process/

[25] "Git." [Online]. Available: https://git-scm.com/