10.2514/6.2016-1422

AIAA SciTech Forum
4-8 January 2016, San Diego, California, USA
57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference

Kona: A Parallel Optimization Library for Engineering-Design Problems

Alp Dener * Pengfei Meng † Jason Hicken ‡ Graeme J. Kennedy §
John Hwang ¶ Justin Gray ‖

Kona is a Python library targeting partial-differential-equation (PDE) governed optimization problems. To address the high computational cost of such problems, Kona permits parallel execution of linear algebra and optimization operations while remaining agnostic to the implementation details of the underlying PDE solver. To accomplish this, Kona adopts a reverse-communication-inspired interface where the optimization algorithm requests the PDE solver to perform a predetermined set of tasks on solver-generated memory. Consequently, the optimization itself is parallelized as long as the user defines parallel data structures within an abstract vector interface and performs the requested tasks in parallel. This abstraction layer also facilitates the rapid development of new optimization algorithms independently from the underlying PDE solvers. In this paper we describe Kona's software design in detail, and demonstrate its use on test cases, ranging from analytical verification problems to a PDE-constrained engineering system.

I. Introduction

Systems governed by partial-differential-equations (PDEs) are prevalent in engineering applications, and numerical optimization of these systems has become common in aerospace engineering. There are a number of factors that distinguish these problems from other optimization problems. The analysis of these engineering systems has a significant computational cost, necessitating the use of parallelized PDE solvers. Furthermore, existing legacy PDE solvers are usually specialized to their domains, making it challenging to implement efficient one-shot optimization approaches.

To the best of our knowledge, there are no optimization frameworks tailored to the unique characteristics of PDE-governed design optimization problems. The state-of-the-art approach has been to use serial libraries based on matrix-explicit sequential quadratic optimization (SQO) algorithms with quasi-Newton approximations. However, these problems can sometimes have thousands of design variables and state-based constraints, particularly for structural design, making matrix-explicit serial algorithms ill-suited. In response, practitioners often develop their own optimization libraries, but these libraries often lack the features and robustness of general-purpose libraries and are not appropriate for widespread use.

Our goal in this paper is to introduce a new optimization framework, named Kona, that parallelizes optimization tasks via a reverse-communication abstraction layer between optimization algorithms and parallelized PDE solvers. Development of Kona originally began in 2012 as a C++ library for full-space optimization[1] that was subsequently extended to provide reduced-space algorithms.[2] The present work stems from a Python re-write of Kona involving a redesign of the abstraction layer with simplified syntax focused on reduced-space optimization. We hope that our optimization framework and its reduced-space Newton-Krylov (RSNK) algorithms will help practitioners perform efficient optimization of PDE-governed systems, and allow researchers to rapidly develop and test new optimization algorithms. To that end, we have made Kona available on GitHub[a] for public use under GNU Lesser General Public License.

*PhD Student, Rensselaer Polytechnic Institute, AIAA Student Member
†PhD Student, Rensselaer Polytechnic Institute, AIAA Student Member
‡Assistant Professor, Rensselaer Polytechnic Institute, AIAA Member
§Assistant Professor, Georgia Institute of Technology, AIAA Member
¶Postdoctoral Research Fellow, University of Michigan, Ann Arbor, AIAA Student Member
‖Aerospace Engineer, NASA Glenn Research Center, MDAO Branch, AIAA Member
[a]https://github.com/OptimalDesignLab/Kona

As part of this work, Kona was also integrated into NASA Glenn Research Center's OpenMDAO framework[3] [b], a platform for development of coupled multidisciplinary models with analytic derivatives. OpenMDAO provides a unified interface for a number of optimizers, including several SQP algorithms. Kona's integration links our algorithms to a range of problems implemented as part of OpenMDAO, and allows easy comparison against other SQP methods without having to re-implement a given problem for each optimizer.

We will begin by discussing Kona's high-level software design in Section II. We will then introduce implementation examples for a solver and an optimization algorithm in Sections III and IV respectively, present test cases in Section V and finish with closing remarks in Section VI.

II. Library Design

The high-level design of Kona can be thought of in three blocks: the solver interface, the optimization algorithm interface, and an abstraction layer that bridges the two together. In this section, we will refer to these as "solver side", "optimization side" and "abstraction layer" respectively.

It is important to stress that the optimization side never directly interacts with the solver side and vice versa, instead going through the abstraction layer. Our motivation for such a separation is to allow the development of new optimization algorithms and the integration of new PDE solvers independently from one another.

Figure 1 shows a minimal UML diagram for Kona. Here, `ISolver`, `IAllocator` and `IVector` define the solver side interfaces. `IAlgorithm` and `IHessian` define the optimization side interfaces. `Optimizer` is the top level optimization controller users are intended to interact with. Finally, `KonaMemory`, `VectorFactory`, `KonaVector` and `KonaMatrix` form the reverse-communication abstraction layer.

Below we will describe the design of each block individually, explaining in detail the components listed in the UML diagram. To facilitate this discussion, we introduce a generic equality-constrained, PDE-governed optimization problem statement.

$$\begin{aligned} \min_{x} \quad & f(x,u) \\ \text{subject to} \quad & c(x,u) \geq 0 \\ \text{governed by} \quad & R(x,u) = 0 \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^n$ are the design variables and $u \in \mathbb{R}^s$ are the state variables. The objective function, $f : \mathbb{R}^n \times \mathbb{R}^s \to \mathbb{R}$, and constraints, $c : \mathbb{R}^n \times \mathbb{R}^s \to \mathbb{R}^m$, are assumed to be C^2 continuous functions. In this work, we focus on reduced-space PDE-governed optimization algorithms. Consequently, we define the state variables as implicit functions of the design variables via a discretized set of PDEs, denoted here by $R(x,u)$.

While this sample problem is formulated with only one residual, it should be noted that this residual can also represent a set of coupled PDEs, a multi-point set of uncoupled PDEs, or in general, a set of non-linear algebraic equations for u. Kona is agnostic to all these combinations, as long as the user formulates appropriate operations for this composite residual. Consequently, our references to a "PDE solver" throughout this discussion is generic and interchangeable with multiple PDE solvers or algebraic equations.

II.A. Solver-side Interface

Kona's interaction with a PDE solver is inspired by a "reverse communication" feature first proposed by Ashby and Seager[4] in the context of iterative linear system solvers. The goal of reverse communication, as stated by Ashby and Seager, is to perform algebraic operations while avoiding data-structures entirely, such that the user has the freedom to implement the data storage in whatever architecture they prefer, and adopt any parallelization scheme. Consequently, an algorithm using reverse communication never interacts with data directly. Instead, it sends requests to the user to perform certain linear algebra tasks on data stored in particular locations. This ensures the algorithm's operability with both MPI- and GPGPU-style parallelism, with no scheme-specific modifications to the optimization algorithms.

Kona adopts this reverse communication model by requiring users to create three solver-side objects that follow a predetermined interface.

- `IVector` – A vector object implementing ten basic algebra tasks on the underlying data set.

[b] http://openmdao.org/

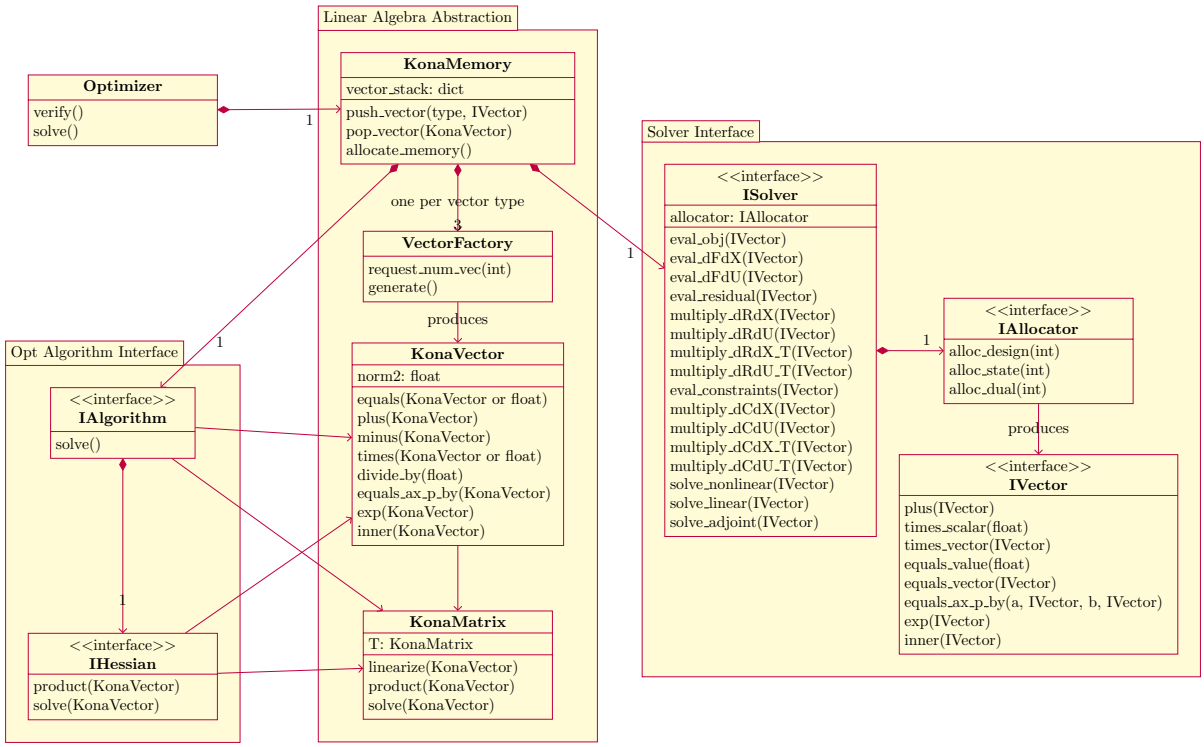


Figure 1: Minimal UML diagram for Kona, showing only high-level associations.

- `IAllocator` – A memory allocation tool that will initialize a requested number of `IVector` objects in a given vector space.
- `ISolver` – A solver wrapper with a predetermined set of member functions performing function evaluations, partial-derivative evaluations, Jacobian-vector products, and linear and non-linear system solutions on `IVector` objects.

II.A.1. Abstract Vectors

User-created vectors are required to be an implementation of the `IVector` interface shown in the UML diagram in Figure 1. Depending on the type of problem the user aims to solve, three versions of this abstract vector may have to be implemented, one for each vector space, namely design variables, state variables and dual variables.

The `IVector` interface consists of eight specific algebra operations on the underlying data structure referenced within the abstract vector. These operations are:

- `plus(IVector)` – In-place summation with the given vector.
- `times_scalar(float)` – In-place multiplication with a scalar.
- `times_vector(IVector)` – In-place element-wise multiplication with the given vector.
- `equals_vector(IVector)` – Value assignment to the given vector.
- `equals_value(float)` – Value assignment to a scalar.
- `equals_ax_p_by(a, IVector, b, IVector)` – Value assignment to the scaled sum of two given vectors.
- `exp(IVector)` – In-place element-wise exponential operation on the given vector.
- `inner(IVector)` – In-place inner product with the given vector.

By default, the Kona library provides a basic serial implementation of these operations based on `NumPy ndarray` data storage. Source code for this standard implementation is given in Appendix C.

For high-performance applications, the choice of data structure within the abstract-vector object is left up to the user. However, it is strongly recommended to store and manipulate the data with a compiled language, in a parallelized environment, consistent with the PDE solver's data structure. This is facilitated by Python's interoperability with other languages; for example, Python can leverage compiled Fortran code using F2Py,[5] or it can be extended to C/C++ using the native Python C API, Cython,[6] or external libraries such as SWIG[7] and Boost.Python.[8]

II.A.2. Vector Allocator

Once the abstract vector object is defined, Kona then requires a way to allocate the underlying data structure in bulk. This is achieved using an `IAllocator` interface with three member functions, one for each vector space, that produce standard Python arrays containing the requested number of user vectors.

As before, Kona provides a basic serial implementation of this called `BaseAllocator` that produces `BaseVectors` sized according to the requested vector space. The source code for this implementation is provided in Appendix D.

II.A.3. Solver Wrapper

The `ISolver` wrapper interface defines most of Kona's reverse communication scheme with the PDE solver. It contains a reference to the `IAllocator` implementation, and a set of predetermined member functions that perform various linear algebra tasks on `IVector` implementations.

Table 1 outlines all `ISolver` operations that must be implemented by the user. These methods are called upon by Kona's reverse-communication abstraction layer to perform various optimization tasks. For instance, when an algorithm needs to evaluate the total design derivative of the objective function, the abstraction layer will first evaluate the partial state derivative using `eval_dFdU` and then use this as the negative right-hand-side vector in a `solve_adjoint` call in order to calculate adjoint variables. The matrix-vector product between the adjoint variables and $\frac{\partial R}{\partial x}^T$, evaluated using `multiply_dRdX_T`, will then be added to the partial design derivative of the objective function, evaluated using `eval_dFdX`. The vector summations are performed via the algebra tools, also defined by the user, under the `IVector` implementation.

Kona provides an empty base class implementation of this interface, relying on the aforementioned base implementations for vector and allocator objects, intended to assist in the rapid development of simple test problems. The example problem outlined in Section III inherits from this base class in its implementation. Other example "solvers", many of which will be used in this work as verification problems, are available under the `kona.examples` module for testing.

II.B. Reverse-Communication Abstraction

The core functionality of Kona revolves around the idea of wrapping the user-provided `ISolver` and `IVector` implementations into abstract vectors and matrices, which are then used to write optimization algorithms with a simple, easy-to-read syntax. In order to accomplish this, the reverse-communication abstraction layer implements a memory manager that contains a "stack" of user-defined vector objects and factories that wrap these user-vectors into linear algebra objects used by optimization algorithms.

A very similar abstraction layer, named Thyra, has been developed as part of the Trilinos framework.[9] Thrya implements abstract vectors in different vector spaces and provides vector factories for their construction just as Kona does. In place of abstract matrices, however, Thrya opts to implement linear operators that define matrix-related operations instead. This choice makes no significant difference in practice. Kona's abstract matrix objects do not contain any actual matrix data. They only house matrix-vector operations such as products and solutions (inverse products) which are ideally implemented matrix-free on the solver side.

II.B.1. Kona Memory Manager

`KonaMemory` is a memory manager object that contains a reference to the user-provided `ISolver` implementation, and it uses the `IAllocator` implementation within the solver to create a stack of `IVector` instances.

Table 1: Operations defined by the `ISolver` interface.

Operation	Interface Calls
Function evaluations: evaluate the objective function, discrete PDE residual, and constraints at the design-state pair (x, u)	
$f = f(x,u) \quad f \in \mathbb{R}$ $R = R(x,u) \quad R \in \mathbb{R}^s$ $c = c(x,u) \quad c \in \mathbb{R}^m$	`eval_obj`, `eval_residual`, `eval_constraints`
Objective derivatives: evaluate the objective function partial derivatives at the design-state pair (x, u)	
$y = \frac{\partial f}{\partial x} \quad y \in \mathbb{R}^n$ $v = \frac{\partial f}{\partial u} \quad v \in \mathbb{R}^s$	`eval_dFdX`, `eval_dFdU`
Matrix-vector products: evaluate Jacobian-vector products, linearized about (x, u)	
$v = \frac{\partial R}{\partial x} y, \quad y = \left(\frac{\partial R}{\partial x}\right)^T v, \quad y \in \mathbb{R}^n, v \in \mathbb{R}^s$ $v = \frac{\partial R}{\partial u} w, \quad w = \left(\frac{\partial R}{\partial u}\right)^T v, \quad v, w \in \mathbb{R}^s$ $\mu = \frac{\partial c}{\partial x} y, \quad y = \left(\frac{\partial c}{\partial x}\right)^T \mu, \quad y \in \mathbb{R}^n, \mu \in \mathbb{R}^m$ $\mu = \frac{\partial c}{\partial u} w, \quad w = \left(\frac{\partial c}{\partial u}\right)^T \mu, \quad w \in \mathbb{R}^s, \mu \in \mathbb{R}^m$	`multiply_dRdX`, `multiply_dRdX_T`, `multiply_dRdU`, `multiply_dRdU_T`, `multiply_dCdX`, `multiply_dCdX_T`, `multiply_dCdU`, `multiply_dCdU_T`
PDE solves: solve the following systems for u, w, and ϕ	
$R(x,u) = 0, \quad \left(\frac{\partial R}{\partial u}\right) w = u, \quad \left(\frac{\partial R}{\partial u}\right)^T \phi = v$ $v, w, u, \phi \in \mathbb{R}^s$	`solve_nonlinear`, `solve_linear`, `solve_adjoint`

These preallocated user-defined vectors are then wrapped into specialized Kona vectors and served to the optimization algorithms as needed. Neither the solver-side nor the optimization side are intended to interact with this memory manager. It is instantiated by the top level optimization controller, and entirely hidden from the users on either side of the abstraction layer.

`KonaMemory` implements three functions used to manage the stack:

- `push_vector(vec_type, IVector)` – Pushes the given user-defined `IVector` object onto the stack associated with the given vector space.
- `pop_vector(vec_type)` – Pops and returns a user-defined `IVector` object from the stack associated with the requested vector space.
- `allocate_memory()` – Uses the `IAllocator` implementation provided by the user to populate the vector stacks with a predetermined number of `IVector` objects.

It could be said that the memory manager simulates dynamic memory allocation for the optimization side, while still using preallocated solver memory in practice. Whenever a new `KonaVector` is created, a user-defined vector of the appropriate vector space is popped off the memory stack and embedded into the Kona vector. Conversely, whenever an existing Kona vector is destroyed, the embedded user-defined vector is pushed onto the memory stack for later use. This scheme emerges from the motivation of making the optimization algorithm syntax appear like pseudo-code, while also respecting good HPC coding practices by avoiding a large number of unnecessary allocations and deallocations.

However, this design choice introduces a difficulty in counting the number of user-defined vectors that need to be preallocated ahead of the optimization. To solve this problem, we implement vector factories.

II.B.2. Vector Factories

Kona's vector factories are objects that serve two purposes: 1) tallying up vector allocation requirements for each vector space, and 2) generating specialized Kona vectors on-demand using preallocated user-defined vectors from the memory stack. Kona's memory manager creates three factories in total, one for each vector space – design (henceforth referred to as "primal"), state and dual. These factories are then passed onto the optimization algorithm, allowing the algorithm and all its components to report their total data space requirements to the memory manager during initialization. The same factories are later used by the algorithm to "dynamically" produce Kona vectors as needed. The RSNK algorithm described in Section IV offers a thorough use-case example of these factories.

II.B.3. Kona Vectors

Kona internally implements a generalized vector object, `KonaVector`, that is used as a base class for all specialized vectors. This class contains fundamental algebra methods such as vector summation and subtraction, scalar multiplication and division, and inner products, common to all user-vector spaces. Each `KonaVector` produced by a vector factory also contains a unique `IVector` object taken off the internal memory stack. The algebra operations are then linked to the algebra implementation provided by the user as part of the `IVector` interface, allowing all `KonaVector` objects to perform vector operations without direct access to the underlying data, or any awareness of the storage and parallelization scheme adopted by the user.

A `KonaVector` by itself is not sufficient to perform optimization tasks. Three other classes, `PrimalVector`, `StateVector` and `DualVector`, inherit from `KonaVector` and add specialized methods for their respective vector spaces. These specialized methods go through the memory manager and call upon solver methods, implemented by the user as part of the `ISolver` interface, to perform various optimization tasks. Consequently, these three vectors are the common vector objects used by all optimization algorithms implemented in Kona.

Within the abstract vector module, Kona also introduces the concept of a composite vector. As the name suggests, these objects are composites of the three core vector objects described above. Unlike core vectors, composite vectors do not possess their own unique data space. Instead, they contain references to component vectors that make up the composite. For instance, a `ReducedKKTVector` in Kona is defined by the composite of a `PrimalVector` and a `DualVector`. Any operations defined on this composite vector modifies the data space of the underlying core vectors.

II.B.4. Kona Matrices

In addition to abstract vectors, Kona also implements abstract matrices that allow optimization algorithms to use Jacobian-vector products and linear solves. As with vectors, Kona defines a base matrix object called `KonaMatrix` that the core set of matrices all inherit from. These core matrices are the residual Jacobians, `dRdX` and `dRdU`, and the constraint Jacobians, `dCdX` and `dCdU`.

Unlike vectors, however, the base `KonaMatrix` object does not implement any useful methods on its own. Instead, it is used to define an interface that all matrix objects in Kona must adhere to. This interface requires, at minimum, a transpose attribute `.T`, as well as `linearize()` and `product()` member functions. The state-Jacobian of the residual, `dRdU`, additionally implements a `solve()` method as well. These methods are all connected to the appropriate Jacobian-vector products and linear solves defined in the `ISolver` interface, implemented by the user.

It is important to note here that the Jacobian matrices defined in this module do not possess any data. They are containers for `ISolver` methods, which in turn are recommended to be implemented in a matrix-free fashion. The `linearize()` method does not actually perform an expensive matrix linearization. Instead, it simply updates references to the design and state vectors about which linear solves and matrix-vector products should be performed. These references are used when calling the appropriate Jacobian-vector product or linear solve method in `ISolver`. The user is responsible for performing the matrix-linearization on the solver side, should they choose to perform these operations using assembled matrices.

II.C. Optimization-side Tools and Interfaces

All data manipulation required by optimization algorithms in Kona is provided by the reverse-communication abstraction layer described above. However, we still need additional tools that build on this abstraction layer

in order to be able to perform numerical optimization. Consequently, the algorithm layer of Kona has two purposes: provide basic optimization utilities used by most optimization algorithms, and then build on top of these tools a common interface for performing the optimization.

In this paper, we will not go into the details of the optimization utilities. These tools are simply Kona-specific versions of well-known algorithms, adapted to operate on `KonaVector` and `KonaMatrix` objects. Instead, we provide a brief list of what the library has currently implemented:

- **Merit Functions** – Three types: the objective function, l_2 merit function, and the augmented Lagrangian merit function.
- **Line-Search Algorithms** – Two methods: a back-tracking line search and a line search satisfying the strong Wolfe conditions.
- **Krylov Solvers** – Four iterative system solvers: Steihaug-Toint Conjugate Gradient (`STCG`),[10] Flexible Generalized Minimum Residual (`FGMRES`),[11] Flexible Generalized Conjugate Residual with Outer Truncation (`GCROT`),[12] and a novel variation of FGMRES for equality-constrained quadratic subproblems called the Flexible Equality-Constrained Subproblem (`FLECS`) solver.[13]

The optimization interface, which we will describe in detail, is separated into two complementary components: `IHessian`, a `KonaMatrix`-like interface defining a Hessian matrix, and `IAlgorithm`, an algorithm interface that builds on the Hessian definition to perform the optimization. This separation grants one optimization algorithm the flexibility to operate with different Hessian approximations and contributes to the symbolic math syntax of the library.

II.C.1. Optimization Algorithms

Optimization algorithms in Kona implement the outer/nonlinear optimization iterations within which `KonaVector`, `KonaMatrix` and `IHessian` objects are used to perform various optimization steps. The `IAlgorithm` interface that defines these objects has very few requirements: they must be initialized using vector factories, and they must implement a `solve()` method that triggers the optimization.

One of the important functions of these algorithm objects, besides performing the outer optimization iterations, is to use the vector factories to initialize all the underlying tools required for optimization. At minimum this involves initializing the Hessian approximation, but it could also include setting up line searches, merit functions and Krylov solvers. Performing the initializations using the same vector factories allows the memory manager to correctly determine the solver-side memory requirements of the entire optimization process.

Kona currently implements several gradient-based optimization algorithms: unconstrained reduced-space quasi-Newton, unconstrained STCG-based reduced-space Newton-CG, and two constrained reduced-space Newton-Krylov (RSNK) algorithms (composite-step[14] and FLECS-based). The constrained RSNK algorithms can solve both equality and inequality constrained problems through the use of a novel exponential slack term that will be described in a future paper by the authors.

II.C.2. Hessian Approximations

Kona's `IHessian` interface is modeled after the `KonaMatrix` objects described before as part of the reverse-communication abstraction layer. This Hessian interface requires mandatory `product()` and `solve()` methods that operate on various `KonaVector` objects. Here, the `solve()` method is equivalent to a product with the approximate inverse Hessian. Optionally, some approximations might require a `linearize()` method as well.

Kona currently implements several Hessian approximations: `LimitedMemoryBFGS`, `LimitedMemorySR1`, `ReducedHessian`, `TotalConstraintJacobian`, `ReducedKKTMatrix`, `LagrangianHessian`, and `AugmentedKKTMatrix`. Our reduced-space quasi-Newton algorithm implementation can operate interchangeably with either the L-BFGS or the L-SR1 quasi-Newton approximations, while the remaining five matrices represent reduced-space systems used in our RSNK methods.

III. Solver Example: Spiral System

The Spiral system is an unconstrained non-linear optimization problem intended to illustrate how the `ISolver` interface operates in practice. The problem has one design variable and two state variables, and is defined as

$$\begin{aligned} \min_x \quad & f(x, u(x)) = \frac{1}{2}(x^2 + u_1^2 + u_2^2) \\ \text{governed by} \quad & R(x, u(x)) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - \begin{bmatrix} x^2 \cos\alpha \\ x^2 \sin\alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \\ & \theta = \frac{1}{2}(x + \pi) \quad \text{and} \quad \alpha = \frac{1}{2}(x - \pi), \end{aligned}$$

where the design variable is $x \in \mathbb{R}$, and the state variables are $u(x) = \begin{bmatrix} u_1 & u_2 \end{bmatrix}^T \in \mathbb{R}^2$.

With the problem defined, we can now easily derive all the Jacobians and partial derivatives listed in Table 1.

$$\frac{\partial f}{\partial x} = x, \qquad \frac{\partial f}{\partial u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = u,$$

$$\frac{\partial R}{\partial x} = \frac{1}{2} \begin{bmatrix} -\sin\theta u_1 + \cos\theta u_2 - 4x\cos\alpha + x^2 \sin\alpha \\ -\cos\theta u_1 - \sin\theta u_2 - 4x\sin\alpha - x^2 \cos\alpha \end{bmatrix}.$$

$$\frac{\partial R}{\partial u} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

In this simple example, explicit Jacobians are very small matrices that can be derived analytically, which is not representative of large-scale PDE governed optimization problems. When dealing with such solvers, we have had significant success relying on algorithmic differentiation to generate matrix-free Jacobian-vector products and partial derivatives.[15] We recommend that any large, parallelized solver used with Kona adopt a similar approach for implementing linearized Jacobian-vector products.

Appendix E includes the Python source code for a class implementing this problem as a standalone "PDE" solver. This solver is then wrapped into an `ISolver` implementation in Appendix F for operability with Kona. In this case, the `ISolver` implementation inherits from an empty base class linked to Kona's default `BaseVector` and `BaseAllocator` implementations based on `NumPy ndarray` data storage.

Since this is a matrix-based solver, it contains a linearization method that updates the non-linear Jacobians and gradients with the design and state points Kona passes onto the `ISolver` interface as arguments. As noted before, this approach is typically inadvisable in high-performance applications, as it will yield a large number of computationally expensive Jacobian assemblies and re-factorizations.

The optimization results of this problem can be found in Section V.A, among other analytical verification problems.

IV. Algorithm Example: Reduced-Space Newton-CG

Reduced-space Newton-Krylov (RSNK) algorithms are a central part of our approach in addressing the challenges of PDE-governed design optimization. Consequently, the unconstrained RSNK algorithm offers a good implementation example that will not only provide a practical application of the `IAlgorithm` and `IHessian` interfaces, but also help justify Kona's software design choices.

Our focus in this section is the implementation details of the second-order adjoint formulation for the Hessian-vector product, and the use of this Hessian in the RSNK algorithm. The full derivation of this formulation, for both unconstrained and equality-constrained problems, is available in previously published papers by the authors.[2,15] However, this description will focus solely on the unconstrained formulation.

We begin by introducing the first-order optimality equations for an unconstrained optimization problem

$$g(x, u(x)) = \nabla_x f(x, u(x)) = 0.$$

where $g(x, u(x)) \in \mathbb{R}^n$ is the total derivative of the objective function, $\nabla_x f$. Next we linearize about x_k to obtain the system

$$\mathsf{H}_k p_k = -g_k,$$

where $\mathsf{H}_k \in \mathbb{R}^{n \times n}$ is the Hessian of the objective function, $p_k \in \mathbb{R}^n$ is the search direction, and g_k as defined above, all evaluated at the design $x_k \in \mathbb{R}^n$.

The RSNK approach inexactly solves this system at each Newton iteration using a Krylov solver, leveraging the rapid convergence of Newton's method. In addition, employing a Krylov solver allows us to avoid an explicit Hessian formulation, and instead iteratively solve the system using Hessian-vector products of the form $\mathsf{H}w$, where $w \in \mathbb{R}^n$ is an arbitrary vector in the design space. However, this introduces two challenges: addressing nonconvexity and efficiently computing the Hessian-vector product itself.

When detailing the optimization side interface for Kona, we introduced some Krylov solvers already implemented in the library based on Kona's abstract vectors. Out of these implementations, the unconstrained RSNK algorithm uses the Steihaug-Toint Conjugate Gradient (STCG) solver, while the constrained version uses the novel Flexible Equality-Constrained Subproblem (FLECS) solver. Both Krylov solvers adopt trust-region methods for globalization, thereby addressing the challenge of convexity.

In order to define the Hessian-vector product, RSNK uses a second-order adjoint approach. To facilitate the discussion of this formulation, we will introduce some new notation.

In general, the optimization problem is governed by a non-linear PDE or state system defined by the residual $R(x, u) = 0 \in \mathbb{R}^s$. The implicit dependence of the state variables, $u \in \mathbb{R}^s$, on the design variables is satisfied through this residual. Consequently, the total design derivative of an objective function that depends on the states can be formulated using the adjoint method:

$$g(x, u, \psi) \equiv \nabla_x f = \frac{\partial f}{\partial x} + \frac{\partial R}{\partial x}^T \psi,$$

where $\psi \in \mathbb{R}^s$ are the first-order adjoint variables. The adjoint system is defined as

$$S(x, u, \psi) = \frac{\partial R}{\partial u}^T \psi + \frac{\partial f}{\partial u}.$$

For the second-order adjoint formulation, we begin by constructing the Lagrangian,

$$\mathcal{L}(x, u, \psi, w, z, \lambda) = g(x, u, \psi)^T w + [R(x, u)]^T \lambda + [S(x, u, \psi)]^T z,$$

where $w \in \mathbb{R}^n$ is the arbitrary vector in the Hessian-vector product, and $\lambda, z \in \mathbb{R}^s$ are the second-order adjoint variables. These second-order adjoints can be calculated by solving the following two linear systems, which are obtained by differentiating the Lagrangian with respect to ψ and u, respectively.

$$\frac{\partial R}{\partial u} z = -\frac{\partial R}{\partial x} w, \qquad \frac{\partial R}{\partial u}^T \lambda = -\frac{\partial g}{\partial x}^T w - \frac{\partial S}{\partial u}^T z,$$

The Hessian-vector product is then defined as the derivative of the Lagrangian with respect to the design variables:

$$\mathsf{H}w = \nabla_x \mathcal{L} = \frac{\partial g}{\partial x}^T w + \frac{\partial R}{\partial x} \lambda + \frac{\partial S}{\partial x}^T z.$$

Table 2: Analytical test problems categorized by their features.

Problems	Governing Equation	Equality Constraints	Inequality Constraints	State-based Constraints
Rosenbrock[16]	×	×	×	×
Spiral	✓	×	×	×
Sphere	×	✓	×	×
Exponential	×	×	✓	×
Sellar[17]	✓	×	✓	✓

It is important to note that this formulation of the Hesian-vector product contains second-derivatives included in $\frac{\partial g}{\partial x}^T$ and $\frac{\partial S}{\partial x}^T$, but these derivatives always appear in matrix-vector products. Consequently, our RSNK implementation approximates the product terms using inexpensive directional finite-differences.

The code for the second-order adjoint formulation is an `IHessian` implementation, presented as an abstract Kona matrix object named `ReducedHessian`. It includes the optional `linearize()` method discussed earlier, internally assembling the total objective gradient g_k and the first-order adjoint residual S_k at the design-state-adjoint linearization point (x_k, u_k, ψ_k)[c]. Following the linearization, the `product()` method then performs the Hessian-vector product as described above for any given w vector.

Using this abstract Hessian object, the RSNK algorithm itself reduces to a simple Newton loop that uses STCG to solve the Newton system defined by the Hessian object at each iteration. Appendix G shows the source code for the core Newton loop in the `STCG_RSNK` object. Kona's Hessian/algorithm separation and vector abstraction layer together yield an easily readable algorithm, hiding the PDE solver implementation details completely from the optimization side.

For constrained problems, a similar `ReducedKKTMatrix` object implements the primal-dual matrix-vector product used with the FLECS Krylov solver. Individual matrix objects for the Hessian and constraint Jacobian blocks of the primal-dual matrix are also available, and currently used in the composite-step algorithm.

In Section V.A, we use the STCG-based RSNK algorithm on the Spiral optimization problem as part of Kona's analytical verification tests.

V. Test Cases

To demonstrate Kona's various capabilities, we present a selection of test cases: a diverse set of analytical verification problems, and a stress-constrained mass minimization problem.

V.A. Analytical Verification Problems

In order to test, debug and verify Kona, we have implemented five analytical test problems with different features intended to target various different algorithms in our optimization framework. Table 2 categorizes these problems based on the existence of state variables and the types of constraints. The Rosenbrock[16] function and the Sellar[17] problem are documented in literature, and the Spiral problem was described above in Section E. Appendix A provides mathematical statements of the verification problems developed by the authors.

The verification problems are tested with the following algorithms in Kona:

- **L-BFGS RSQN** – Reduced-space quasi-Newton with limited-memory BFGS updates (unconstrained).
- **STCG-RSNK** – Reduced-space Newton-CG (unconstrained).
- **CS-RSNK** – Composite-step reduced-space Newton-Krylov (constrained).
- **FLECS-RSNK** – FLECS-based reduced-space Newton-Krylov (constrained).

[c]This assembly does not involve any adjoint recalculations.

As part of this work, we have also integrated Kona into OpenMDAO 1.x Alpha as an optimization driver. All the test problems in this section are implemented and solved via OpenMDAO using various optimization algorithms in Kona. SNOPT,[18] a quasi-Newton SQP library available in pyOptSparse,[19] is used as a benchmark in the verification. Figure 2 shows the objective function evaluations across nonlinear iterations. The measured quantity is the difference between the numerical objective value and the analytical optimum, normalized by the initial error.

These results demonstrate that the optimization algorithms in Kona recover the expected optima.

V.B. Stress-constrained Mass Minimization

High-performance structures are often designed for minimum weight with a constraint that the stress within the structure cannot exceed some specified value. When the behavior of the structure is modeled using a PDE, these types of design problems constitute challenging PDE-governed optimization problems with local state-based constraints. In this paper, we will consider stress-constrained mass minimization of 2D plane stress structures. We begin by describing the structural design problem that will be solved.

In a 2D plane-stress problem, the structure consists of a planar domain where the loads are restricted to lie in-plane. The design parameters in this problem are structural thicknesses, and the state variables are the displacements along the in-plane coordinate directions. The stress at any point in the structure can be evaluated based on the in-plane displacements. The stress-constrained problem has several interesting properties within the context of PDE-governed optimization. Firstly, the governing equations are self-adjoint and therefore both the direct and transposed matrix-vector products are implemented using the same code. Furthermore, the preconditioner for both forward and transpose products is also shared. In this case, we use a preconditioner based on V-cycle multigrid where the operators are formed from a Galerkin projection between mesh levels.

In this particular implementation, we have adopted the use of inverse thickness design variables in order to facilitate more favorable trust-region globalization behavior near the lower thickness bound. In addition, we have also implemented a conic spatial filter, such that the thickness at each element is the weighted combination of design variables in adjacent elements within the filter. With this parameterization, the thicknesses can be expressed a function of the design variables:

$$t_i(x) = \sum_{k \in \mathcal{F}_i} \frac{w_{ik}}{x_k} m \qquad i = 1, \ldots, n,$$

where n is the number of structural components in the model.

The Jacobian of the governing equations, i.e. the stiffness matrix, can be written as a linear combination of the thicknesses as follows:

$$\mathsf{K}(x) = \sum_{i=1}^{n} t_i(x) \mathsf{K}_i,$$

where K_i are positive semi-definite matrices associated with each structural component in the model. The governing equations are then written as

$$R(x, u) = \mathsf{K}(x)u - f = 0,$$

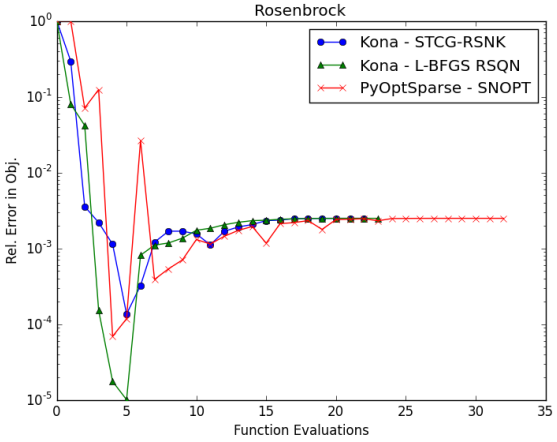
where u are the state variables and f is the force vector. The local stress constraints are based on a von Mises stress criteria and can be written as quadratic functions of the state variables as follows:

$$c_i(u) = 1 - u^T B_i^T G B_i u \geq 0 \qquad i = 1, \ldots, n,$$

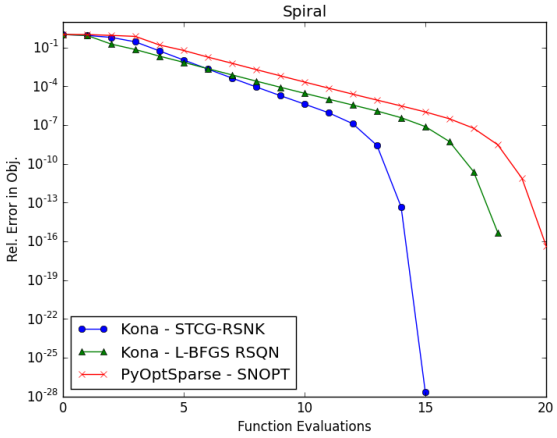
where $B_i \in \mathbb{R}^{3 \times s}$ is a linear operator that returns the strain at a point within the structure as a function of the displacement vector, and $G \in \mathbb{R}^{3 \times 3}$ is a positive-definite matrix with entries that depend on the material stiffness properties and the maximum allowable stress.

Finally, the mass can be written as

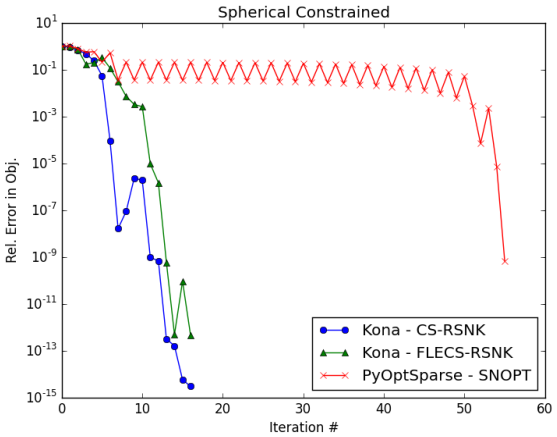
$$m(x) = \sum_{i=1}^{n} m_i \frac{1}{x_i},$$



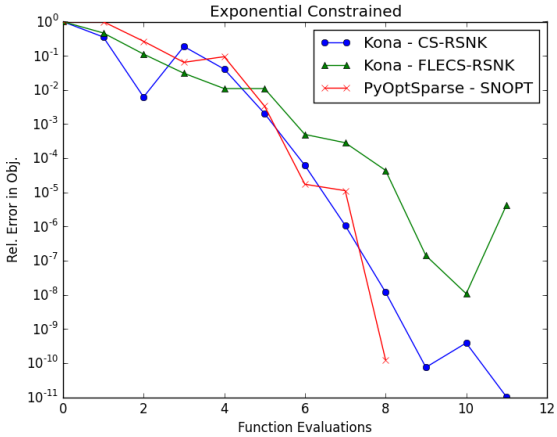
(a) Rosenbrock function optimization.



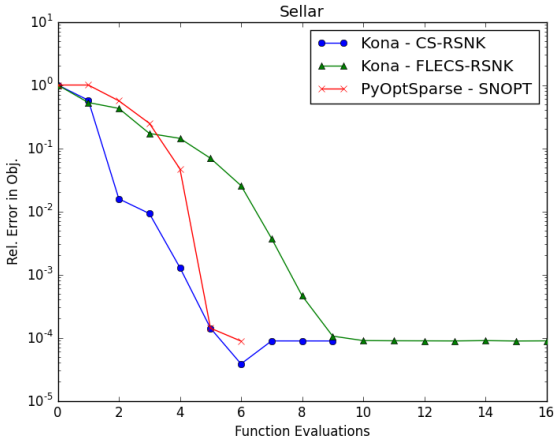
(b) Spiral problem optimization.



(c) Sphere problem optimization.



(d) Exponential problem optimization.



(e) Sellar problem optimization.

Figure 2: Relative error in the objective value across function evaluations for all verification problems

where m_i is the mass per unit thickness of the i-th element of the discrete structure. The full design problem is

$$\begin{aligned} \min_{x} \quad & m(x) \\ \text{subject to} \quad & c(u) \geq 0 \\ & x_{\max} \geq x \geq x_{\min} \\ \text{governed by} \quad & \mathsf{K}(x)u - f = 0 \end{aligned}$$

where $x_{\min}$ and $x_{\max}$ are the lower and upper bounds on the design variables, recovered from the upper and lower bounds on plate thicknesses.

In this paper we apply the above problem to a cantilever beam configuration with an applied point load, as shown in Figure 3. A similar formulation on the same PDE solver, with forcing applied as traction along the entire edge instead of a point load in the corner, has previously been solved using a full-space Newton-Krylov approach.[20] Here we will be using the FLECS-based reduced-space Newton-Krylov algorithm. Kona's abstraction layer is suitable for full-space optimization problems, but we have not yet implemented a full-space (i.e. one-shot, or SAND) algorithm as part of our optimization framework. We will be exploring this option in future work.

Figure 3: Cantilever beam configuration.

One of the features of this problem is that the number of design and state variables are easily scalable by the discretization resolution of the cantilever plate. In this work, we present only the results obtained from a 64-by-32 cartesian grid. This produces 2048 design variables and 6144 constraints; 2048 each for upper and lower bounds on the design variables, and 2048 total stress constraints across all elements.

Figure 4 shows the convergence history for this problem, where mass, optimality and feasibility metrics are all normalized by their initial values. Figure 5 shows solutions at several iterations along the optimization run, where the thickness values increase from red to yellow to blue. While the FLECS-based RSNK algorithm reduces the optimality and feasibility metrics by roughly three orders of magnitude from their initial values, we believe that the solution can be converged further. The stress constraints in the reduced-space make the primal-dual systems exceedingly difficult for the Krylov-based solver to solve to a sufficiently tight tolerance, which then causes asymptotic nonlinear convergence to suffer.

This mirrors our previous experience applying the RSNK approach to aerodynamic shape optimization,[15] and once again highlights the need to develop effective preconditioners for the reduced-space primal-dual system. This is ongoing work in our research.

VI. Closing Remarks

In this paper, we have described the software design of a novel Python optimization library, called Kona, that aims to perform parallel optimization of large-scale PDE-governed engineering systems. This is achieved via a reverse-communication abstraction layer that allows optimization algorithms implemented in Kona to perform all necessary optimization tasks without access to direct data. This separation makes the optimization algorithms agnostic to distributed data structures and parallelization schemes implemented by the underlying "PDE solver".

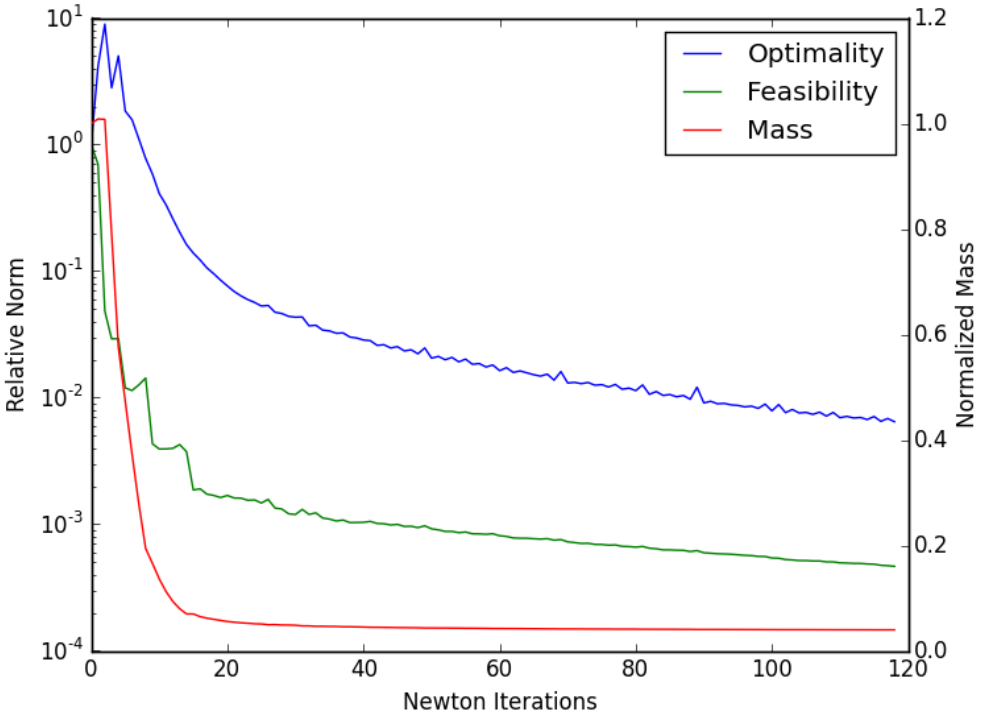


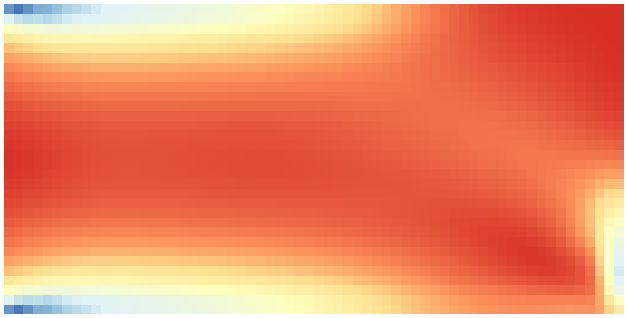
Figure 4: Convergence history for the stress-constrained mass minimization problem.

Through our test problems we have demonstrated integration with OpenMDAO and basic verification of various optimization algorithms implemented in our library. In the future, we aim to leverage this OpenMDAO integration to test our algorithms on large, challenging problems such as a CubeSat satellite optimization[21] and aircraft mission-allocation.[22]

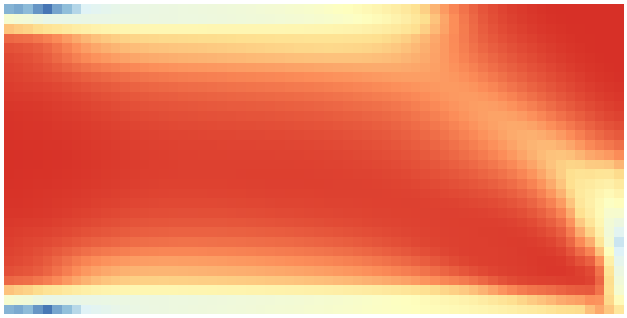
We exercised the RSNK algorithm on a stress-constrained mass minimization, a challenging problem with a large number of state-based constraints. While convergence performance can be improved, we are nonetheless able to make significant strides towards the optimum. We are pursuing ongoing research in ways to improve the convergence of the reduced-space primal-dual system for such problems. In the process, the linear algebra abstraction layer in Kona facilitates rapid development of new algorithms independently from the underlying solver, helping us achieve short turnaround times while exploring new primal-dual system preconditioners.

While the results shared here do not demonstrate parallelization, the Python-rewrite of Kona has been used to perform parallelized aerodynamic shape optimization. This work builds on our previous efforts using the C++ version of Kona on this problem[15] and will be explored further in future research by the authors.

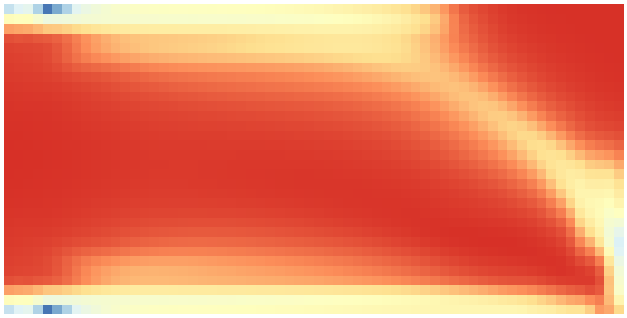
(a) Initial cantilever beam.



(b) Solution at the 20^{th} Newton iteration.



(c) Solution at the 70^{th} Newton iteration.



(d) Final solution.

Figure 5: Thickness distribution of the plate at representative points in the optimization. Thickness values increase from red, to yellow, to blue.

References

[1]Hicken, J. E. and Alonso, J. J., "Comparison of Reduced- and Full-space Algorithms for PDE-constrained Optimization," *51st AIAA Aerospace Sciences Meeting*, Jan. 2013, AIAA 2013–1043.

[2]Hicken, J. E., "Inexact Hessian-vector products in reduced-space differential-equation constrained optimization," *Opti-*

mization and Engineering, Vol. 15, No. 3, 2014, pp. 575–608.

[3]Gray, J. S., Hearn, T. A., Moore, K. T., Hwang, J., Martins, J., and Ning, A., "Automatic Evaluation of Multidisciplinary Derivatives Using a Graph-Based Problem Formulation in OpenMDAO," *15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, American Institute of Aeronautics and Astronautics, 2014/07/08 2014.

[4]Ashby, S. F. and Seager, M. K., "A proposed standard for iterative linear solvers," Tech. rep., UCRL-102860, Lawrence Livermore national Laboratory, 1990.

[5]Peterson, P., "F2PY: a tool for connecting Fortran and Python programs," *International Journal of Computational Science and Engineering*, Vol. 4, No. 4, 2009, pp. 296–305.

[6]Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., and Smith, K., "Cython: The Best of Both Worlds," *Computing in Science Engineering*, Vol. 13, No. 2, 2011, pp. 31 –39.

[7]Beazley, D. M. and Lomdahl, P. S., "Feeding a Large-scale Physics Application to Python," *In 6th International Python Conference*, 1997, pp. 21–28.

[8]Abrahams, D. and Grosse-Kunstleve, R. W., "Building hybrid systems with Boost. Python," *CC Plus Plus Users Journal*, Vol. 21, No. 7, 2003, pp. 29–36.

[9]Bartlett, R. A., "Thyra Linear Operators and Vectors," Tech. rep., SAND2007-5984, Sandia National Laboratories, Aug. 2010.

[10]Steihaug, T., "The conjugate gradient method and trust regions in large scale optimization," *SIAM Journal on Numerical Analysis*, Vol. 20, No. 3, 1983, pp. 626–637.

[11]Saad, Y., "A flexible inner-outer preconditioned GMRES algorithm," *SIAM Journal on Scientific Computing*, Vol. 14, No. 2, 1993, pp. 461–469.

[12]Hicken, J. E. and Zingg, D. W., "A simplified and flexible variant of GCROT for solving nonsymmetric linear systems," *SIAM Journal on Scientific Computing*, Vol. 32, No. 3, 2010, pp. 1672–1694.

[13]Hicken, J. E. and Dener, A., "A Flexible Iterative Solver for Nonconvex, Equality-Constrained Quadratic Subproblems," *SIAM Journal on Scientific Computing*, Vol. 37, No. 4, 2015, pp. A1801–A1824.

[14]Heinkenschloss, M. and Ridzal, D., "A matrix-free trust-region SQP method for equality constrained optimization," *SIAM Journal on Optimization*, Vol. 24, No. 3, 2014, pp. 1507–1541.

[15]Dener, A., Kenway, G. K., Lyu, Z., Hicken, J. E., and Martins, J., "Comparison of Inexact- and Quasi-Newton Algorithms for Aerodynamic Shape Optimization," *53rd AIAA Aerospace Sciences Meeting*, American Institute of Aeronautics and Astronautics, Reston, Virginia, Jan. 2015.

[16]Rosenbrock, H., "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, Vol. 3, No. 3, 1960, pp. 175–184.

[17]Sellar, R., Batill, S., and Renaud, J., "Response surface based, concurrent subspace optimization for multidisciplinary system design," *AIAA paper*, Vol. 714, 1996, pp. 1996.

[18]Gill, P. E., Murray, W., and Saunders, M. A., "SNOPT: An SQP algorithm for large-scale constrained optimization," *SIAM journal on optimization*, Vol. 12, No. 4, 2002, pp. 979–1006.

[19]Hwang, J. T., Kenway, G. K. W., and Martins, J. R. R. A., "Geometry and Structural Modeling for High-Fidelity Aircraft Conceptual Design Optimization," *Proceedings of the 15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Atlanta, GA, June 2014.

[20]Kennedy, G. J., "A Full-Space Method with Matrix Aggregates for Stress-Constrained Structural Optimization," *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Dallas, TX, 2015.

[21]Hwang, J. T., Lee, D. Y., Cutler, J. W., and Martins, J. R. R. A., "Large-Scale Multidisciplinary Optimization of a Small Satellite's Design and Operation," *Journal of Spacecraft and Rockets*, Vol. 51, No. 5, September 2014, pp. 1648–1663.

[22]Hwang, J. T., Roy, S., Kao, J. Y., Martins, J. R. R. A., and Crossley, W. A., "Simultaneous aircraft allocation and mission optimization using a modular adjoint approach," *56th AIAA SDM Conference*, Kissimmee, FL, 2015.

A. Verification Problem Formulations

A.A. Sphere Constrained Problem

$$\begin{aligned} \min_{x,y,z} \quad & f(x,y,z) = x + y + z \\ \text{subject to} \quad & 3 - (x^2 + y^2 + z^2) \geq 0 \end{aligned}$$

A.B. Exponential Constrained Problem

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = x + y^2 \\ \text{subject to} \quad & e^x - 1 \geq 0 \end{aligned}$$

A.C. Sellar Problem

With $x = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ and $u = \begin{pmatrix} u_1 & u_2 \end{pmatrix}^T$:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(x,u) = x_3^2 + x_2 + u_1 + e^{-u_2} \\ \text{governed by} \quad & R(x,u) = \begin{bmatrix} u_1 - x_1^2 - x_3 - x_2 + 0.2u_2 \\ u_2 - \sqrt{u_1} - x_1 - x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \text{subject to} \quad & \frac{u_1}{3.16} - 1 \geq 0 \\ & 1 - \frac{u_2}{24} \geq 0 \\ & -10 \leq x_1 \leq 10 \\ & 0 \leq x_2 \leq 10 \\ & 0 \leq x_3 \leq 10 \end{aligned}$$

B. Full UML Diagram

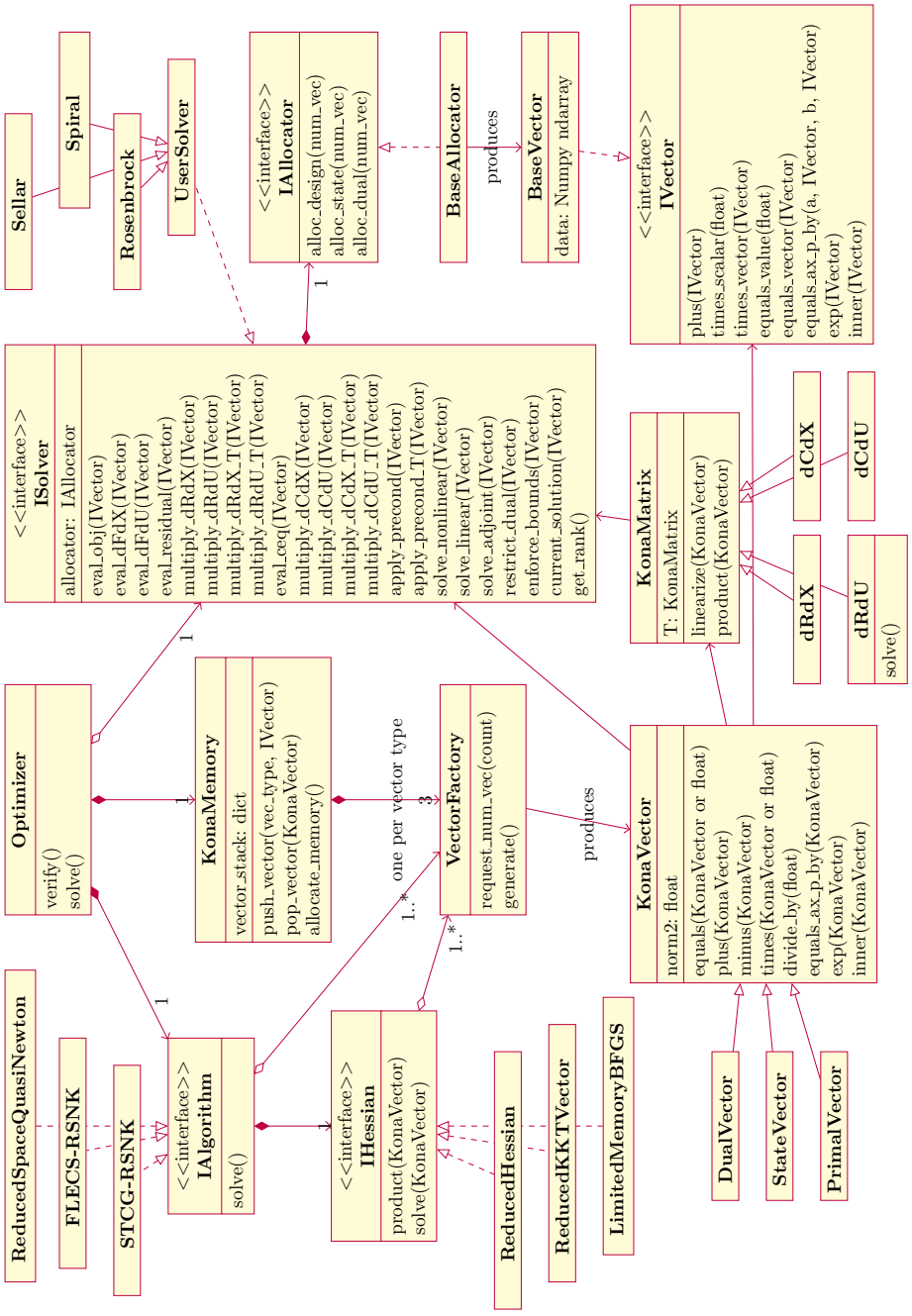


Figure 6: Complete UML diagram for Kona optimization library.

C. BaseVector – IVector Implemention with NumPy

```
import numpy as np

class BaseVector(object):
    """
    Kona's default data container, implemented on top of NumPy arrays.

    Any user defined data container must implement all the methods below.

    These vectors are initialized by the user-created `BaseAllocator` object.
    Therefore, the initialization implementation does not need to exactly
    follow the example below. The user is free to initialize these vector
    objects any which way they like, as long as it is in sync with the
    `BaseAllocator` implementation.

    Parameters
    ----------
    size: int
        Size of the 1-D numpy vector contained in this object.
    val : float, optional
        Data value for vector initialization.

    Attributes
    ----------
    data : numpy.array
        Numpy vector containing numerical data.
    """
    def __init__(self, size, val=0):
        if np.isscalar(val):
            if val == 0:
                self.data = np.zeros(size, dtype=float)
            elif isinstance(val, (np.float, np.int)):
                self.data = np.ones(size, dtype=float)*val
        elif isinstance(val, (np.ndarray, list, tuple)):
            if size != len(val):
                raise ValueError(
                    'size given as %d, but length of value %d'%(size, len(val)))
            self.data = np.array(val)
        else:
            raise ValueError(
                'val must be a scalar or array like, ' +
                'but was given as type %s'%(type(val)))

    def plus(self, vector):
        """
        Add the given vector to this vector.

        Parameters
        ----------
        vector : BaseVector
            Incoming vector for in-place operation.
        """
        self.data = self.data + vector.data

    def times_scalar(self, value):
        """
        Multiply all elements of this vector with the given scalar.

        Parameters
        ----------
        value: float
        """
        self.data = value*self.data

    def times_vector(self, vector):
        """
        Perform element-wise multiplication between vectors.
```

```
        Parameters
        ----------
        vector : BaseVector
            Incoming vector for in-place operation.
        """
        self.data = self.data*vector.data

    def equals_value(self, value):
        """
        Set all elements of this vector to given scalar value.

        Parameters
        ----------
        value : float
        """
        self.data[:] = value

    def equals_vector(self, vector):
        """
        Set this vector equal to the given vector.

        Parameters
        ----------
        vector : BaseVector
            Incoming vector for in-place operation.
        """
        self.data[:] = vector.data[:]

    def equals_ax_p_by(self, a, x, b, y):
        """
        Perform the elementwise scaled addition defined below:

        .. math:: a\\mathbf{x} + b\\mathbf{y}

        The result is saved into this vector.

        Parameters
        ----------
        a : double
            Scalar coefficient of ``x``.
        x : BaseVector
            Vector to be operated on.
        b : double
            Scalar coefficient of ``y``.
        y : BaseVector
            Vector to be operated on.
        """
        self.data = a*x.data + b*y.data

    def inner(self, vector):
        """
        Perform an inner product between the given vector and this one.

        Parameters
        ----------
        vector : BaseVector
            Incoming vector for in-place operation.

        Returns
        -------
        float
            Result of the operation.
        """
        if len(self.data) == 0:
            return 0.
        else:
            return np.inner(self.data, vector.data)

    def exp(self, vector):
        """
```

```
        Calculate element-wise exponential operation on the vector.

        Parameters
        ----------
        vector : BaseVector
            Incoming vector for in-place operation.
        """
        self.data = np.exp(vector.data)
```

D. BaseAllocator – IAllocator Implementation for BaseVector

```
from kona.user import BaseVector

class BaseAllocator(object):
    """
    Allocator object that handles the generation of vectors within the
    problem's various vector spaces.

    Parameters
    ----------
    num_primal : int
        Primal space size.
    num_state : int
        State space size.
    num_dual : int
        Dual space size.

    Attributes
    ----------
    num_primal : int
        Primal space size.
    num_state : int
        State space size.
    num_dual : int
        Dual space size.
    """
    def __init__(self, num_primal, num_state, num_dual):
        self.num_primal = num_primal
        self.num_state = num_state
        self.num_dual = num_dual

    def alloc_primal(self, count):
        """
        Initialize as many instances of the "primal" vector space object as
        Kona requested.

        Parameters
        ----------
        count : int
            Number of vectors requested in the primal-space.

        Returns
        -------
        list
            A standard Python array containing the reuqested number of
            instances of the user's primal-space `BaseVector` implementation.
        """
        out = []
        for i in xrange(count):
            out.append(BaseVector(self.num_primal))
        return out

    def alloc_state(self, count):
        """
        Initialize as many instances of the "state" vector space object as
        Kona requested.

        Parameters
        ----------
        count : int
            Number of vectors requested in the state-space.

        Returns
        -------
        list
            A standard Python array containing the reuqested number of
            instances of the user's state-space `BaseVector` implementation.
        """
```

```
        out = []
        for i in xrange(count):
            out.append(BaseVector(self.num_state))
        return out

    def alloc_dual(self, count):
        """
        Initialize as many instances of the "dual" vector space object as
        Kona requested.

        Parameters
        ----------
        count : int
            Number of vectors requested in the dual-space.

        Returns
        -------
        list
            A standard Python array containing the reuqested number of
            instances of the user's dual-space `BaseVector` implementation.
        """
        out = []
        for i in xrange(count):
            out.append(BaseVector(self.num_dual))
        return out
```

E. Spiral Problem – "PDE" Solver

```
import numpy as np

class SpiralSolver(object):

    def linearize(self, at_design, at_state=None):
        self.X = at_design.data[0]
        if at_state is not None:
            self.U = at_state.data
        else:
            self.U = None

    @property
    def theta(self):
        return 0.5*(self.X + np.pi)

    @property
    def alpha(self):
        return 0.5*(self.X - np.pi)

    @property
    def dRdX(self):
        dRdX1 = -np.\sin{self.theta)*self.U[0] + np.\cos{self.theta)*self.U[1] \
                - self.X*np.\cos{self.alpha) + (self.X**2)*np.\sin{self.alpha)
        dRdX2 = -np.\cos{self.theta)*self.U[0] - np.\sin{self.theta)*self.U[1] \
                - self.X*np.\sin{self.alpha) - (self.X**2)*np.\cos{self.alpha)
        return 0.5*np.array([[dRdX1],[dRdX2]])

    @property
    def dRdU(self):
        return np.array([[np.\cos{self.theta), np.\sin{self.theta)],
                         [-np.\sin{self.theta), np.\cos{self.theta)]])

    @property
    def rhs(self):
        return np.array([(self.X**2)*np.\cos{self.alpha),
                         (self.X**2)*np.\sin{self.alpha)])

    @property
    def R(self):
        return self.dRdU.dot(self.U) - self.rhs

    @property
    def F(self):
        return 0.5*(self.X**2 + self.U[0]**2 + self.U[1]**2)

    @property
    def dFdX(self):
        return np.array([self.X])

    @property
    def dFdU(self):
        return np.array([self.U[0], self.U[1]])
```

F. Spiral Problem – `ISolver` Implementation

```
from kona.user import UserSolver

class Spiral(UserSolver):

    def __init__(self):
        super(Spiral, self).__init__(1,2,0)
        self.PDE = SpiralSolver()

    def eval_obj(self, at_design, at_state):
        self.PDE.linearize(at_design, at_state)
        return self.PDE.F

    def eval_residual(self, at_design, at_state, store_here):
        self.PDE.linearize(at_design, at_state)
        store_here.data = self.PDE.R

    def multiply_dRdX(self, at_design, at_state, in_vec, out_vec):
        self.PDE.linearize(at_design, at_state)
        out_vec.data = self.PDE.dRdX.dot(in_vec.data)

    def multiply_dRdU(self, at_design, at_state, in_vec, out_vec):
        self.PDE.linearize(at_design, at_state)
        out_vec.data = self.PDE.dRdU.dot(in_vec.data)

    def multiply_dRdX_T(self, at_design, at_state, in_vec, out_vec):
        self.PDE.linearize(at_design, at_state)
        out_vec.data = self.PDE.dRdX.T.dot(in_vec.data)

    def multiply_dRdU_T(self, at_design, at_state, in_vec, out_vec):
        self.PDE.linearize(at_design, at_state)
        out_vec.data = self.PDE.dRdU.T.dot(in_vec.data)

    def eval_dFdX(self, at_design, at_state, store_here):
        self.PDE.linearize(at_design, at_state)
        store_here.data = self.PDE.dFdX

    def eval_dFdU(self, at_design, at_state, store_here):
        self.PDE.linearize(at_design, at_state)
        store_here.data = self.PDE.dFdU

    def init_design(self, store_here):
        store_here.data = np.array([1.0])

    def solve_nonlinear(self, at_design, result):
        # linearize the PDE
        self.PDE.linearize(at_design)
        # perform the solution
        self.PDE.U = np.linalg.solve(self.PDE.dRdU, self.PDE.rhs
        # write result and return cost
        result.data = self.PDE.U
        return 0

    def solve_linear(self, at_design, at_state, rhs_vec, rel_tol, result):
        self.PDE.linearize(at_design, at_state)
        result.data = np.linalg.solve(self.PDE.dRdU, rhs_vec.data)
        return 0

    def solve_adjoint(self, at_design, at_state, rhs_vec, rel_tol, result):
        self.PDE.linearize(at_design, at_state)
        result.data = np.linalg.solve(self.PDE.dRdU.T, rhs_vec.data)
        return 0
```

G. STCG-based RSNK Algorithm – Newton loop

```
# use vector factories to generate abstract Kona vectors
x = self.primal_factory.generate()
p = self.primal_factory.generate()
dJdX = self.primal_factory.generate()
primal_work = self.primal_factory.generate()
state = self.state_factory.generate()
adjoint = self.state_factory.generate()
state_work = self.state_factory.generate()

# set initial design and solve for state
x.equals_init_design()
state.equals_primal_solution(x)

# solve for adjoint
adjoint.equals_adjoint_solution(x, state, state_work)

# calculate the total gradient
dJdX.equals_total_gradient(x, state, adjoint, primal_work)

# get objective value
obj = objective_value(x, state)

# calculate convergence criterion
grad_norm0 = dJdX.norm2
grad_tol = self.primal_tol*grad_norm0

# start Newton loop
for i in xrange(self.max_iter):
    # check convergence
    grad_norm = dJdX.norm2
    if grad_norm < grad_tol:
        break

    # define adaptive Krylov tolerance for superlinear convergence
    krylov_tol = self.krylov.rel_tol*min(1.0, sqrt(grad_norm/grad_norm0))
    krylov_tol = max(krylov_tol, grad_tol/grad_norm)
    self.krylov.rel_tol = krylov_tol

    # use -dJdX as the Newton step RHS vector
    dJdX.times(-1.0)
    # update Krylov solver trust-radius
    self.krylov.radius = self.radius
    # set the evaluation point for the Hessian-vector product
    self.hessian.linearize(x, state, adjoint)
    # solve the system (H*p = -dJdX)
    pred, active = self.krylov.solve(self.hessian.product, dJdX, p)
    # revert the sign on dJdX
    dJdX.times(-1.0)

    # update the design point with the calculated step
    x.plus(p)

    # compute the objective function reduction and trust parameter rho
    obj_old = obj
    state_work.equals(state)
    if state.equals_primal_solution(x):
        obj = objective_value(x, state)
        rho = (obj_old - obj)/pred
    else:
        rho = np.nan

    # update trust radius if necessary
    if rho < 0.25 or np.isnan(rho):
        self.radius *= 0.25
    else:
        # if we hit the trust boundary, increase radius
        if active and (rho > 0.75):
```

```
            self.radius = min(2*self.radius, self.max_radius)

    # revert the solution if necessary
    if rho < 0.1:
        x.minus(p)
        state.equals(state_work)
    else:
        # if the solution is good, solve for the adjoint
        adjoint.equals_adjoint_solution(x, state, state_work)
        # calculate the total gradient
        dJdX.equals_total_gradient(x, state, adjoint, primal_work)
```