ParOpt: A parallel library of large-scale optimization algorithms

Graeme J. Kennedy Yicong Fu

1 Introduction

ParOpt is a parallel gradient-based optimization library implemented in C++ and is intended for solving large-scale constrained optimization problems. ParOpt can be applied to general purpose optimization problems, but is often used for large-scale topology optimization. The constraints in ParOpt fall into one of two categories: (1) constraints with full dependence on the design vector such that the constraint Jacobian is fully populated, or (2) constraints that have a specific sparse structure, described below, that enables them to be grouped independently. All operations in ParOpt use distributed design vectors and almost all computations are performed in parallel, with a small number of factorization operations performed on small dense matrices in serial. ParOpt can optionally use information from Hessian-vector products to accelerate convergence. Within the Hessian-vector product mode, inexact solutions of the KKT system are used where the tolerances are determined using the Eisenstat–Walker forcing terms.

ParOpt utilizes both a C++ and a python-level interface. The python-level interface is generated using cython. Call-backs from C++ to python are implemented using direct memory access into numpy arrays. Some care must be exercised when setting or reading values from arrays passed to python-level functions so as not to inadvertently set gradient or Jacobian values into a copied vector. Furthermore, the design variable vector passed during callbacks is used by ParOpt, so modification of design vector will produce undesirable results. ParOpt uses an abstract problem interface with an abstract vector class that can be implemented by application-specific methods. This enables the use of externally-defined vectors, as long as basic vector-vector and vector-scalar operations are defined. A default ParOptVec class is implemented to provide generic functionality.

The following document is divided into two sections: (1) a high-level description of the algorithms implemented in ParOpt, and (2) a detailed description of the implementation of ParOpt.

2 Interior point algorithm

ParOpt implements an interior-point method to solve optimization problems formulated as follows:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{such that} \quad & \mathbf{c}(\mathbf{x}) \geq 0 \\ & \mathbf{c}_w(\mathbf{x}) \geq 0 \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \end{aligned} \tag{1}$$

Here, $\mathbf{c}(\mathbf{x})$ are the dense constraints, and $\mathbf{c}_w(\mathbf{x})$ are the sparse constraints. The sparse constraint Jacobian, $\mathbf{A}_w = \nabla_x \mathbf{c}_w(\mathbf{x})$ must have a structure such that the matrix

$$\mathbf{D} = \mathbf{A}_w \mathbf{S} \mathbf{A}_w^T$$

is a block-diagonal matrix whenever $\mathbf{S}$ is a diagonal matrix. This structure arises in many topology and multimaterial optimization problems that employ weighting constraints for each element within the problem.

An interior point algorithm approximately solves a sequence of barrier problems that are designed to approach the true constrained minimizer in the limit. The barrier problem is formed by adding inequality constraints to the objective through a log barrier penalty function. This barrier function is designed to keep the iterates strictly in the interior of the feasible region. The barrier problem corresponding to (1) is the following

$$\begin{aligned} \min_{\mathbf{x},\mathbf{s},\mathbf{t},\mathbf{s}_w} \quad & \varphi(\mathbf{x},\mathbf{s},\mathbf{t},\mathbf{s}_w;\mu) = f(\mathbf{x}) + \boldsymbol{\gamma}_t^T \mathbf{t} + \boldsymbol{\gamma}_s^T \mathbf{s} - \mu \left[\log \mathbf{s} + \log \mathbf{t} + \log \mathbf{s}_w + \log(\mathbf{x}-\mathbf{l}) + \log(\mathbf{u}-\mathbf{x})\right] \\ \text{such that} \quad & \mathbf{c}(\mathbf{x}) = \mathbf{s} - \mathbf{t} \\ & \mathbf{c}_w(\mathbf{x}) = \mathbf{s}_w \end{aligned} \tag{2}$$

where $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{s}_w$ are slack variables associated with the dense and sparse constraints, respectively. The function log is the component-wise sum of the logarithms of the vector components, i.e. $\log \mathbf{s} = \sum_i \ln s_i$. As the barrier parameter, μ, decreases, the minimizer of the barrier problem (2) approaches the KKT solution.

The barrier problem (2) is related to a set of perturbed KKT conditions for the optimization problem (1). Introducing dual variables for the dense constraints $\mathbf{z}$, the sparse constraints $\mathbf{z}_w$, and the lower and upper bounds, $\mathbf{z}_l$, and $\mathbf{z}_u$, the perturbed KKT conditions can be written as follows:

$$\begin{aligned} \mathbf{r}_x &\triangleq \mathbf{g} - \mathbf{A}^T \mathbf{z} - \mathbf{A}_w^T \mathbf{z}_w - \mathbf{z}_l + \mathbf{z}_u = 0 \\ \mathbf{r}_s &\triangleq \boldsymbol{\gamma}_s + \mathbf{z} - \mathbf{z}_s = 0 \\ \mathbf{r}_t &\triangleq \boldsymbol{\gamma}_t - \mathbf{z} - \mathbf{z}_t = 0 \\ \mathbf{r}_c &\triangleq \mathbf{c} - \mathbf{s} + \mathbf{t} = 0 \\ \mathbf{r}_{c_w} &\triangleq \mathbf{c}_w - \mathbf{s}_w = 0 \\ \mathbf{r}_{z_s} &\triangleq \mathbf{S}\mathbf{z}_s - \mu \mathbf{e} = 0 \\ \mathbf{r}_{z_t} &\triangleq \mathbf{T}\mathbf{z}_t - \mu \mathbf{e} = 0 \\ \mathbf{r}_{z_w} &\triangleq \mathbf{S}_w \mathbf{z}_w - \mu \mathbf{e} = 0 \\ \mathbf{r}_{z_l} &\triangleq (\mathbf{X} - \mathbf{L})\mathbf{z}_l - \mu \mathbf{e} = 0 \\ \mathbf{r}_{z_u} &\triangleq (\mathbf{U} - \mathbf{X})\mathbf{z}_u - \mu \mathbf{e} = 0 \end{aligned} \tag{3}$$

Here, the gradient of the objective function is $\mathbf{g} = \nabla_x f(\mathbf{x})$ and the Jacobians of the constraints are $\mathbf{A} = \nabla_x \mathbf{c}(\mathbf{x})$ and $\mathbf{A}_w = \nabla_x \mathbf{c}_w(\mathbf{x})$.

At each step of the optimization algorithm, ParOpt computes an update $\mathbf{p}$ to the design variables, slacks, and multipliers, based on either an inexact or an approximate Newton step, which can be written as follows:

$$\mathbf{K}\mathbf{p} = -\mathbf{r},$$

where $\mathbf{K}$ is either the exact Jacobian or an approximate Jacobian of the perturbed KKT system (3). When a quasi-Newton method is used the Jacobian is approximate, and when Hessian-vector products are used the Jacobian is exact, but the linear system is solved inexactly.

In the quasi-Newton mode, the matrix $\mathbf{K}_B \approx \mathbf{K}$ is an approximate Jacobian due to the use of a Hessian approximation $\mathbf{B} \approx \mathbf{H} \triangleq \nabla_x^2 \left(f(\mathbf{x}) - \mathbf{z}^T \mathbf{c}(\mathbf{x}) - \mathbf{z}_w^T \mathbf{c}_w(\mathbf{x}) \right)$. The approximate KKT matrix is

$$\mathbf{K}_B \mathbf{p} = \begin{bmatrix} \mathbf{B} & 0 & 0 & 0 & -\mathbf{A}^T & 0 & 0 & -\mathbf{A}_w^T & -\mathbf{I} & \mathbf{I} \\ 0 & 0 & 0 & 0 & \mathbf{I} & -\mathbf{I} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\mathbf{I} & 0 & -\mathbf{I} & 0 & 0 & 0 \\ \mathbf{A} & -\mathbf{I} & \mathbf{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{A}_w & 0 & 0 & -\mathbf{I} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{Z}_s & 0 & 0 & 0 & \mathbf{S} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{Z}_t & 0 & 0 & 0 & \mathbf{T} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{Z}_w & 0 & 0 & 0 & \mathbf{S}_w & 0 & 0 \\ \mathbf{Z}_l & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (\mathbf{X}-\mathbf{L}) & 0 \\ -\mathbf{Z}_u & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (\mathbf{U}-\mathbf{X}) \end{bmatrix} \begin{bmatrix} \mathbf{p}_x \\ \mathbf{p}_s \\ \mathbf{p}_t \\ \mathbf{p}_{s_w} \\ \mathbf{p}_z \\ \mathbf{p}_{z_s} \\ \mathbf{p}_{z_t} \\ \mathbf{p}_{z_w} \\ \mathbf{p}_{z_l} \\ \mathbf{p}_{z_u} \end{bmatrix} = -\mathbf{r}. \quad (4)$$

ParOpt uses quasi-Newton Hessian approximations based either on compact limited-memory BFGS or compact limited-memory SR1 updates [Byrd et al., 1994]. Compact representations of limited-memory quasi-Newton approximations take the form

$$\mathbf{B} = b_0 \mathbf{I} - \mathbf{W} \mathbf{M} \mathbf{W}^T,$$

where b_0 is a scalar, $\mathbf{M}$ is a small matrix and $\mathbf{W}$ is a matrix with a small number of columns that is stored as a series of vectors. The form of these matrices depends on whether the limited-memory BFGS or SR1 technique is used. An exact solution to the update step $\mathbf{p}$ can be obtained by using the compact representation in conjunction with the Sherman-Morrison-Woodbury formula.

ParOpt approximately solves the perturbed KKT equations (3) for a sequence of barrier parameters μ_k such that $\mu_k \to 0$ for $k \to \infty$. ParOpt uses a monotone approach Fiacco and McCormick [1990] in which the barrier parameter is maintained at a fixed value and reduced only after a barrier-problem convergence criterion is satisfied. The barrier parameter criterion is that the infinity norm of the solution vector must be reduced below a factor of the barrier parameter itself

$$||\mathbf{r}||_\infty \leq 10 \mu_k. \quad (5)$$

After the barrier criterion is satisfied, the parameter is modified using the expression $\mu_{k+1} \leftarrow \min\{ \theta \mu_k, \mu_k^\beta \}$ for $\beta \in (1,2]$.

2.1 Merit function and line search

The interior-point method implemented in ParOpt uses a line search method that guarantees a sufficient decrease of a merit function at each iteration. The line search is based on the following ℓ_2 merit function:

$$\begin{aligned} \phi(\alpha) = & \varphi(\mathbf{x} + \alpha \mathbf{p}_x^s, \mathbf{s} + \alpha \mathbf{p}_s^s, \mathbf{t} + \alpha \mathbf{p}_t^s, \mathbf{s}_w^s + \alpha \mathbf{p}_{s_w}^s; \mu) + \\ & \nu ||\mathbf{c}(\mathbf{x} + \alpha \mathbf{p}_s^s) - \mathbf{s} + \mathbf{t} - \alpha(\mathbf{p}_s^s - \mathbf{p}_t^s)||_2 + \nu ||\mathbf{c}_w(\mathbf{x} + \alpha \mathbf{p}_{s_w}^s) - \mathbf{s}_w - \alpha \mathbf{p}_{s_w}^s||_2, \end{aligned} \quad (6)$$

where $\mathbf{p}^s$ is the KKT update vector $\mathbf{p}$ scaled to ensure that the primal variables remain strictly within the feasible region and so that the dual variables remain positive. The penalty parameter ν is selected to ensure a sufficiently negative descent direction, such that $\phi'(0)$ is sufficiently negative [Nocedal and Wright, 2006]. At each step ParOpt uses a line search that seeks a point that satisfies the Armijo sufficient decrease condition:

$$\phi(\alpha) < \phi(0) + c_1 \alpha \phi'(0),$$

where we typically choose $c_1 = 10^{-3}$. If a step is unsuccessful, we select the next step using a quadratic interpolation based on the initial point and slope of the merit function along the search direction, as well as the most recent merit function value. Since $\phi'(0)$ is negative, and $\phi(\alpha) \geq \phi(0) + c_1 \alpha \phi'(0)$, this sequence of step lengths is decreasing.

To ensure that the design variables remain within bounds and that the dual and slack variables remain sufficiently positive, ParOpt uses a fraction-to-the-boundary rule, such that

$$\begin{aligned}
\alpha_x &= \max\left\{\alpha \in (0,1] \mid \mathbf{x} + \alpha \mathbf{p}_x - \mathbf{l} \geq (1-\tau)(\mathbf{x} - \mathbf{l})\right\}, \\
\alpha_z &= \max\left\{\alpha \in (0,1] \mid \mathbf{z}_s + \alpha \mathbf{p}_s \geq (1-\tau)\mathbf{z}_s\right\},
\end{aligned}$$

with analogous expressions for the remaining components of the step length vector $\mathbf{p}$. Note that the sign of $\mathbf{z}$ is not directly constrained since it is treated as a multiplier for an equality constraint. The α_x and α_z parameters are then used to compute the step $\mathbf{p}^s$ such that $\mathbf{p}_x^s = \alpha_x \mathbf{p}_x$ and $\mathbf{p}_z = \alpha_z \mathbf{p}_z$. Note that α_x is the step length for the design and slack variables, and α_z is the step length for all multipliers. Following Wächter and Biegler [2006], ParOpt sets the parameter τ as follows

$$\tau = \max(0.95, 1 - \mu).$$

To avoid situations in which there is a large discrepancy between the step lengths, a check is imposed on α_x and α_z such that if $\alpha_x >> \alpha_z$, ParOpt truncates the difference between the step lengths such that

$$\alpha_x = \max(\min(\alpha_x, 100\alpha_z), \alpha_z/100),$$

otherwise if $\alpha_z > \alpha_x$, we set:

$$\alpha_z = \max(\min(\alpha_z, 100\alpha_x), \alpha_x/100).$$

Note that this modification only has an effect if the difference in step lengths exceeds 100. This modification does not interfere with the asymptotic convergence behavior of the algorithm and enables faster recovery from poor steps early in the optimization.

2.2 Solving the approximate KKT system with a compact quasi-Newton Hessian

Within ParOpt, the single most computationally expensive operation at each iteration of the optimization algorithm is the solution of the linearized KKT system obtained from the perturbed KKT conditions (3). The following section presents an overview of the methods used to solve this linear system in a computationally efficient manner in parallel.

To derive the proposed solution procedure, we first express the linearized KKT matrix as a combination of two matrices which take the form:

$$\mathbf{K}_B\mathbf{p} = \left[\mathbf{K}_0 + \mathbf{Y}\mathbf{M}\mathbf{Y}^T\right]\mathbf{p} = -\mathbf{r} \tag{7}$$

where the matrix $\mathbf{Y}$ is

$$\mathbf{Y}^T = \begin{bmatrix}\mathbf{W}^T & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix},$$

and the matrices $\mathbf{W}$ and $\mathbf{M}$ are from the compact BFGS representation. Note that the terms in $\mathbf{K}_0$ represent the diagonal term b_0 from the compact BFGS representation and all other first-order terms from the linearized KKT system.

An exact solution to the linear system (7) can be obtained using the Sherman–Morrison–Woodbury formula. This formula leads to the following expression for the update $\mathbf{p}$:

$$\mathbf{p} = \mathbf{K}_0^{-1}\mathbf{Y}\mathbf{C}^{-1}\mathbf{Y}^T\mathbf{K}_0^{-1}\mathbf{r} - \mathbf{K}_0^{-1}\mathbf{r}$$

where the matrix $\mathbf{C} \in \mathbb{R}^{2m\times 2m}$ is given as follows:

$$\mathbf{C} = \mathbf{Y}^T\mathbf{K}_0^{-1}\mathbf{Y} + \mathbf{M}^{-1}.$$

A solution of the linear system (7) can be obtained from the solution of $2m+1$ linear systems of the form $\mathbf{K}_0\mathbf{y} = \mathbf{b}$. Furthermore, if the matrix $\mathbf{Y}$ is stored as a series of column vectors, then the operations required to compute the solution consist of operations with small matrices of size $\mathcal{O}(m)$, parallel vector-vector products, and the application of $\mathbf{K}_0^{-1}$. Since vector-vector operations parallelize efficiently for distributed vectors, and the small matrix operations normally constitute a small contribution to the overall computational time, we concentrate on the parallel solution of systems of the form $\mathbf{K}_0\mathbf{p} = \mathbf{b}$. Note that this refer to the linear system $\mathbf{K}_0\mathbf{p} = \mathbf{b}$ as the diagonal KKT matrix since the Hessian term in the matrix $\mathbf{K}_0$ is replaced by a diagonal matrix, $\mathbf{B} = b_0\mathbf{I}$.

2.3 Parallel solution of the diagonal KKT system

The diagonal KKT system $\mathbf{K}_0\mathbf{p} = \mathbf{b}$ is solved in parallel through a series of variable eliminations. In general, this method can be susceptible to numerical cancellation, however, experience has shown that this method produces remarkably accurate steps, even for very small values of the barrier parameter. This can be attributed to the structure of the constraint Jacobians. The sequence of variable eliminations produces a linear system for the Lagrange multipliers of the dense constraints. The Schur-complement matrices that are produced during the elimination process can be precomputed and stored. The computations during the elimination process can be reduced to a sequence of vector-vector operations and can therefore be implemented efficiently in parallel. Since each operation can be performed in parallel, with a small number of dense matrix operations on matrices of size $\mathcal{O}(m)$, the entire solution procedure scales efficiently.

The solution procedure begins by obtaining the solution for the slack variables and lower and

upper Lagrange multiplier bound variables as follows:

$$\begin{aligned}
\mathbf{p}_{z_l} &= (\mathbf{X}-\mathbf{L})^{-1}(\mathbf{b}_{z_l}-\mathbf{Z}_l\mathbf{p}_x),\\
\mathbf{p}_{z_u} &= (\mathbf{U}-\mathbf{X})^{-1}(\mathbf{b}_{z_u}+\mathbf{Z}_u\mathbf{p}_x),\\
\mathbf{p}_z-\mathbf{p}_{z_s} &= \mathbf{b}_s,\\
-\mathbf{p}_z-\mathbf{p}_{z_t} &= \mathbf{b}_t,\\
\mathbf{p}_s &= \mathbf{Z}_s^{-1}(\mathbf{b}_{z_s}-\mathbf{S}\mathbf{p}_{z_s}) = \mathbf{Z}_s^{-1}(\mathbf{b}_{z_s}-\mathbf{S}(\mathbf{p}_z-\mathbf{b}_s)),\\
\mathbf{p}_t &= \mathbf{Z}_t^{-1}(\mathbf{b}_{z_t}-\mathbf{T}\mathbf{p}_{z_t}) = \mathbf{Z}_t^{-1}(\mathbf{b}_{z_t}+\mathbf{T}(\mathbf{p}_z+\mathbf{b}_t)),,\\
\mathbf{p}_{s_w} &= \mathbf{Z}_w^{-1}(\mathbf{b}_{s_w}-\mathbf{S}_w\mathbf{p}_{z_w}),
\end{aligned} \tag{8}$$

Next, using the first three equations gives

$$\begin{aligned}
b_0\mathbf{p}_x-\mathbf{A}^T\mathbf{p}_z-\mathbf{A}_w^T\mathbf{p}_{z_w}-\mathbf{p}_{z_l}+\mathbf{p}_{z_u} &= \mathbf{b}_x,\\
\mathbf{A}\mathbf{p}_x-\mathbf{p}_s+\mathbf{p}_t &= \mathbf{b}_c,\\
\mathbf{A}_w\mathbf{p}_x-\mathbf{p}_{s_w} &= \mathbf{b}_{z_w}.
\end{aligned} \tag{9}$$

Substituting the expressions for the slack and Lagrange multiplier updates (8) into the expression for the first three linearized KKT conditions (9) yields the following

$$\begin{aligned}
\mathbf{D}\mathbf{p}_x-\mathbf{A}^T\mathbf{p}_z-\mathbf{A}_w^T\mathbf{p}_{z_w} &= \mathbf{b}_x+(\mathbf{X}-\mathbf{L})^{-1}\mathbf{b}_{z_l}-(\mathbf{U}-\mathbf{X})^{-1}\mathbf{b}_{z_u},\\
\mathbf{A}\mathbf{p}_x+(\mathbf{Z}_s^{-1}\mathbf{S}+\mathbf{Z}_t^{-1}\mathbf{T})\mathbf{p}_z &= \mathbf{b}_c+\mathbf{Z}_s^{-1}(\mathbf{b}_{z_s}+\mathbf{S}\mathbf{b}_s)-\mathbf{Z}_t^{-1}(\mathbf{b}_{z_t}+\mathbf{T}\mathbf{b}_t)\\
\mathbf{A}_w\mathbf{p}_x+\mathbf{Z}_w^{-1}\mathbf{S}_w\mathbf{p}_{z_w} &= \mathbf{b}_{c_w}+\mathbf{Z}_w^{-1}\mathbf{b}_{s_w}
\end{aligned} \tag{10}$$

where the diagonal matrix $\mathbf{D}$ is defined as follows:

$$\mathbf{D} = \left[b_0\mathbf{I}+(\mathbf{X}-\mathbf{L})^{-1}\mathbf{Z}_l+(\mathbf{U}-\mathbf{X})^{-1}\mathbf{Z}_u\right].$$

This system of equations can be arranged as follows

$$\begin{bmatrix}\mathbf{D} & \mathbf{A}_w^T & \mathbf{A}^T\\ \mathbf{A}_w & -\mathbf{C} & 0\\ \mathbf{A} & 0 & -\mathbf{C}_0\end{bmatrix}\begin{bmatrix}\mathbf{p}_x\\ -\mathbf{p}_{z_w}\\ -\mathbf{p}_z\end{bmatrix} = \begin{bmatrix}\mathbf{d}_x\\ \mathbf{d}_{z_w}\\ \mathbf{d}_z\end{bmatrix} \tag{11}$$

where $\mathbf{C}_0 = (\mathbf{Z}_s^{-1}\mathbf{S}+\mathbf{Z}_t^{-1}\mathbf{T})$ and $\mathbf{C} = \mathbf{Z}_w^{-1}\mathbf{S}_w$. The right hand sides are defined as

$$\begin{aligned}
\mathbf{d}_x &= \mathbf{b}_x+(\mathbf{X}-\mathbf{L})^{-1}\mathbf{b}_{z_l}-(\mathbf{U}-\mathbf{X})^{-1}\mathbf{b}_{z_u},\\
\mathbf{d}_{z_w} &= \mathbf{b}_c+\mathbf{Z}_s^{-1}(\mathbf{b}_{z_s}+\mathbf{S}\mathbf{b}_s)-\mathbf{Z}_t^{-1}(\mathbf{b}_{z_t}+\mathbf{T}\mathbf{b}_t),\\
\mathbf{d}_z &= \mathbf{b}_{c_w}+\mathbf{Z}_w^{-1}\mathbf{b}_{s_w}
\end{aligned}$$

The reduced KKT system (**??**) can be solved via a Schur complement on the dense constraints for the unknowns $\mathbf{p}_z$. First, we introduce the quasi-definite matrix $\mathbf{D}_0$ and the rectangular matrix $\mathbf{A}_0$

$$\mathbf{D}_0 = \begin{bmatrix}\mathbf{D} & \mathbf{A}_w^T\\ \mathbf{A}_w & -\mathbf{C}\end{bmatrix}, \qquad \mathbf{A}_0 = \begin{bmatrix}\mathbf{A} & 0\end{bmatrix}.$$

Since we assume that there are only a small number of global dense constraints, $\mathbf{A}$ has only a few rows. The system of equations to determine $\mathbf{p}_z$ is

$$\mathbf{G}\mathbf{p}_z \triangleq \left(\mathbf{C}_0 + \mathbf{A}_0\mathbf{D}_0^{-1}\mathbf{A}_0^T\right)\mathbf{p}_z = \mathbf{d}_z - \mathbf{A}_0\mathbf{D}_0^{-1}\begin{bmatrix}\mathbf{d}_x \\ \mathbf{d}_{z_w}\end{bmatrix}.$$

Once the values of $\mathbf{p}_z$ are obtained, we can find the values for $\mathbf{p}_x$ and $\mathbf{p}_{z_w}$ by solving the equation

$$\begin{bmatrix}\mathbf{D} & \mathbf{A}_w^T \\ \mathbf{A}_w & -\mathbf{C}\end{bmatrix}\begin{bmatrix}\mathbf{p}_x \\ -\mathbf{p}_{z_w}\end{bmatrix} = \begin{bmatrix}\mathbf{d}_x + \mathbf{A}^T\mathbf{p}_z \\ \mathbf{d}_{z_w}\end{bmatrix}.$$

The key operation is the solution of the sparse linear system of the form of $\mathbf{D}_0$. This solution is obtained via sparse direct methods. When the sparse constraints exhibit specific structure, then significant computational savings can often be achieved.

2.4 Inexact Hessian-vector product mode

ParOpt can also use Hessian-vector products to accelerate convergence. This method is designed for convex optimization problems where the Hessian is positive semi-definite and the reduced Hessian is positive definite. This solution phase is entered only after the residuals of the KKT equations are satisfied to a user-specified tolerance. The exact Hessian phase employs an inexact Newton–Krylov method driven with the Eisenstat–Walker forcing parameters. The inexact solution of the lineaized KKT system is obtained using right-preconditioned GMRES. The convergence criteria within GMRES is modified to include conditions that enforce a descent direction for a line search method. The preconditioner for the system of equations is the quasi-Newton approximation $\mathbf{K}_B$. The product of the Jacobian and precondition is

$$\begin{aligned}\mathbf{K}\mathbf{K}_B^{-1} &= \left(\mathbf{K}_B + \mathbf{N}(\mathbf{H}-\mathbf{B})\mathbf{N}^T\right)\mathbf{K}_B^{-1} \\ &= \mathbf{I} + \mathbf{N}(\mathbf{H}-\mathbf{B})\mathbf{N}^T\mathbf{K}_B^{-1}\end{aligned} \tag{12}$$

Here $\mathbf{N}$ is a matrix that consists of an identity in the $\mathbf{x}$-component, and zero everywhere else such that

$$\mathbf{N}^T = \begin{bmatrix}\mathbf{I} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

The right-preconditioned operator $\mathbf{K}\mathbf{K}_B^{-1}$ only modifies the $\mathbf{x}$-components of the output vector directly. Since the Krylov subspace within GMRES is

$$\mathscr{K}_m(\mathbf{K}\mathbf{K}_B^{-1}, \mathbf{r}) = \text{span}\left\{\mathbf{r}, \mathbf{K}\mathbf{K}_B^{-1}\mathbf{r}, \left(\mathbf{K}\mathbf{K}_B^{-1}\right)^2\mathbf{r}, \ldots, \left(\mathbf{K}\mathbf{K}_B^{-1}\right)^{m-1}\mathbf{r}\right\},$$

all vectors in the GMRES algorithm consist of different $\mathbf{x}$-component values, while all remaining components are scalar multiples of $\mathbf{r}$. This property can be used to reduce the memory requirements of GMRES by storing only the $\mathbf{x}$-components of each vector and a scalar for all remaining components. Using this approach, the full vector $\hat{\mathbf{v}}_i$ is stored as a pair $(\mathbf{v}_i, \alpha_i)$ which can be extracted as

$$\hat{\mathbf{v}}_i = \mathbf{N}\mathbf{v}_i + \alpha_i(\mathbf{I} - \mathbf{N}\mathbf{N}^T)\mathbf{r}$$

Using this representation, the dot product of two vectors $\hat{\mathbf{v}}_1$ and $\hat{\mathbf{v}}_2$ stored as $(\mathbf{v}_1, \alpha_1)$ and $(\mathbf{v}_2, \alpha_2)$ can be expressed as

$$\begin{aligned}\hat{\mathbf{v}}_1^T \hat{\mathbf{v}}_2 &= \mathbf{v}_1^T \mathbf{N}^T \mathbf{N} \mathbf{v}_2 + \alpha_1 \alpha_2 \mathbf{r}^T \left(\mathbf{I} - \mathbf{N}\mathbf{N}^T\right) \mathbf{r} \\ &= \mathbf{v}_1^T \mathbf{v}_2 + \beta_r \alpha_1 \alpha_2\end{aligned}$$

where $\beta_r \triangleq \mathbf{r}^T \left(\mathbf{I} - \mathbf{N}\mathbf{N}^T\right) \mathbf{r}$.

When using a line search method, it is necessary to obtain a descent direction. This is guaranteed for the methods which employ a quasi-Newton approximation by maintaining a positive-definite Hessian and exactly satisfying the constraint condition $\mathbf{A}\mathbf{p}_x = -\mathbf{c}$ (omitting the slack variables). However, in the exact Newton method a descent direction is not guaranteed. When the objective is convex, the Hessian is positive semi-definite and a descent direction can be found if the equations are solved to sufficient precision. Within the present algorithm, the convergence criteria is modified within GMRES to require a sufficiently accurate solution that satisfies a descent criteria. A penalty parameter ν for the exact ℓ_2 merit function can be found if either

$$\mathbf{p}_x^T \nabla_x f < 0, \tag{13}$$

or

$$\mathbf{c}^T \mathbf{A}\mathbf{p}_x \leq -\gamma ||\mathbf{c}||_2^2, \tag{14}$$

for some $0 < \gamma < 1$. Therefore, if either condition is satisfied we will have a descent direction.

While it is possible to build the full solution at each GMRES iteration, and check the criteria (13) and (14), it is more efficient to modify GMRES to evaluate these criteria indirectly. GMRES works by building an orthogonal subspace $\mathbf{V}_k \in \mathbb{R}^{n \times k}$ using Arnoldi's method that satisfies the following equation

$$\mathbf{K}\mathbf{K}_B^{-1} \mathbf{V}_k = \mathbf{V}_{k+1} \bar{\mathbf{H}}_{k+1}$$

where the approximate solution is $\mathbf{p} = \mathbf{K}_B^{-1} \mathbf{V}_k \mathbf{y}_k$ where $\mathbf{y}_k$ is obtained from the solution of the least-squares problem

$$\mathbf{y}_k = \arg\min_y ||\bar{\mathbf{H}}_k \mathbf{y} - \beta \mathbf{e}_1||_2.$$

At each iteration, we compute the action of the matrix $\mathbf{w} \leftarrow \mathbf{K}\mathbf{K}_B^{-1}\mathbf{v}$ by first computing the intermediate vector $\mathbf{z} = \mathbf{K}_B^{-1}\mathbf{v}$. Next, the $\mathbf{x}$-components of the output vector $\mathbf{w}$ are obtained by computing

$$\mathbf{w}_x = \mathbf{v}_x + (\mathbf{H} - \mathbf{B})\mathbf{z}_x.$$

Before discarding the intermediate vector $\mathbf{z}$, we compute the directional derivatives of the objective and the ℓ_2 norm of the constraints, respectively as follows

$$a_k = \mathbf{z}_x^T \nabla f \qquad\qquad b_k = \mathbf{z}_x^T \mathbf{A}^T \mathbf{c}.$$

Based on these values, the directional derivative of the inexact solution at iteration k of GMRES can be evaluated as

$$\begin{aligned}\mathbf{p}_x^T \nabla f &= \mathbf{y}_k^T \mathbf{a}_k, \\ \mathbf{p}_x^T \mathbf{A}^T \mathbf{c} &= \mathbf{y}_k^T \mathbf{b}_k.\end{aligned}$$

These quantities can be computed inexpensively by evaluating $\mathbf{y}_k$ at every iteration of GMRES, rather than after the final iteration. If GMRES fails to find an inexact solution that is also a descent direction, we revert back to the quasi-Newton step which is guaranteed to produce a descent direction.

GMRES(m, γ): Inexactly solve $\mathbf{Kp} = \mathbf{b}$ while ensuring $\mathbf{p}$ is a descent direction
Given the relative stopping tolerance ε_r and the descent fraction γ
Evaluate $\beta = ||\mathbf{b}||_2$ and $\beta_r = \mathbf{b}^T \left(\mathbf{I} - \mathbf{N}\mathbf{N}^T\right) \mathbf{b}$
Set $\mathbf{v}_1 = \beta^{-1}\mathbf{b}$ and $k = 1$
while $k \leq m$ **do**
    Compute $\mathbf{z} = \mathbf{K}_B^{-1}\mathbf{v}_k$
    Compute $a_k = \mathbf{z}^T\mathbf{N}^T\nabla_x f$
    Compute $b_k = \mathbf{z}^T\mathbf{N}^T\mathbf{A}^T\mathbf{c}$
    Set $\mathbf{a}_k = (\mathbf{a}_{k-1}, a_k)$ and $\mathbf{b}_k = (\mathbf{b}_{k-1}, b_k)$
    Compute $\mathbf{w} \leftarrow \mathbf{K}\mathbf{z}$
    Compute $\mathbf{V}_{k+1} \leftarrow MGS(\mathbf{V}_k, \mathbf{w})$
    Solve $\mathbf{y}_k = \arg\min_{\mathbf{y}} ||\bar{\mathbf{H}}_k\mathbf{y} - \beta\mathbf{e}_1||_2$
    if $\mathbf{y}_k^T\mathbf{a}_k < 0$ or $\mathbf{y}_k^T\mathbf{b}_k \leq -\gamma||\mathbf{c}||_2^2$ **then**
        if $||\bar{\mathbf{H}}_k\mathbf{y} - \beta\mathbf{e}_1||_2 < \varepsilon_r\beta$ **then**
            Successfully found inexact solution satisfying descent direction criteria.
            break
        end if
    end if
    $k \leftarrow k + 1$
end while
Set $\mathbf{z} = \mathbf{V}_k\mathbf{y}_k$
Compute $\mathbf{p} = \mathbf{K}_B^{-1}\mathbf{z}$
Verify $\mathbf{p}^T\mathbf{N}\nabla_x f < 0$ or $\mathbf{p}^T\mathbf{N}^T\mathbf{A}^T\mathbf{c} \leq -\gamma||\mathbf{c}||_2^2$

Figure 1: GMRES with descent direction convergence criteria

3 Trust region algorithm

ParOpt also implements a trust region method to solve optimization problems. For each trust region iteration, ParOpt constructs a quadratic model subproblem and solves the subproblem using the interior point solver. The general form of the optimization problem is same as Problem (1). Accordingly, the trust region quadratic subproblem with linearized constraints can be formulated as follows:

$$\begin{aligned} \min_{\mathbf{p}} \quad & f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} \\ \text{such that} \quad & \mathbf{A}\mathbf{p} + \mathbf{c} \geq 0 \\ & \mathbf{A}_w\mathbf{p} + \mathbf{c}_w \geq 0 \\ & \mathbf{l} \leq \mathbf{x}_k + \mathbf{p} \leq \mathbf{u} \\ & ||\mathbf{p}||_\infty \leq \Delta \end{aligned} \tag{15}$$

where $\mathbf{p} = \mathbf{x}_{k+1} - \mathbf{x}_k$ is the update step. Note that for the rest equations of this section, we drop the subscript k (which indicates the current trust region iteration) to simplify the notation: Similarly, $f = f(\mathbf{x}_k)$, $\mathbf{g} = \nabla_x f(\mathbf{x}_k)$ are the function value and gradient, respectively and $\mathbf{B} = \nabla_x^2(f(\mathbf{x}_k) - \mathbf{z}^T\mathbf{c}(\mathbf{x}_k) - \mathbf{z}_w^T\mathbf{c}_w(\mathbf{x}_k))$ is the approximation of the Hessian of the Lagrangian. The general and sparse constraints are denoted $\mathbf{c}$ and $\mathbf{c}_w$, respectively, and $\mathbf{A}$ and $\mathbf{A}_w$ are the Jacobians for the dense and sparse constraints. The trust-region radius is given by Δ.

The subproblem (**??**) can be reformulated so that it can be solved using the ParOpt interior point solver. First, notice that we have the identity for an arbitrary scalar variable α

$$\alpha = \max(0, \alpha) - \max(0, -\alpha)$$

and corresponding relations

$$\begin{aligned} \max(0, \alpha) &\geq 0 \\ \max(0, -\alpha) &\geq 0 \\ \max(0, \alpha) \cdot \max(0, -\alpha) &= 0 \end{aligned} \tag{16}$$

As a result, we can define slack variables

$$\mathbf{A}\mathbf{p} + \mathbf{c} = \mathbf{s} - \mathbf{t}$$

such that

$$\begin{aligned} \mathbf{s} &= \max(\mathbf{0}, \mathbf{A}\mathbf{p} + \mathbf{c}) \\ \mathbf{t} &= \max(\mathbf{0}, -(\mathbf{A}\mathbf{p} + \mathbf{c})) \end{aligned}$$

where the max function is applied to vectors in the component-wise fashion. Due to the characteristics of the max function shown in equation (**??**), we will have additional conditions for the slack variable $\mathbf{s}$ and $\mathbf{t}$:

$$\begin{aligned} \mathbf{s} &\geq \mathbf{0} \\ \mathbf{t} &\geq \mathbf{0} \\ \mathbf{S}\mathbf{t} &= \mathbf{0} \end{aligned} \tag{17}$$

where $\mathbf{S}$ is the square matrix diagonalized from vector $\mathbf{s}$, note that this convention is used multiple times in the future.

Returning to subproblem (**??**), we can now move the constraints into the objective function by using l_1 penalization and slack variables that we just defined:

$$\begin{aligned} \min_{\mathbf{p},\mathbf{s},\mathbf{t},\mathbf{s}_w} \quad & f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} + \boldsymbol{\gamma}_t^T\mathbf{t} + \boldsymbol{\gamma}_s^T\mathbf{s} \\ \text{such that} \quad & \mathbf{A}\mathbf{p} + \mathbf{c} = \mathbf{s} - \mathbf{t} \\ & \mathbf{A}_w\mathbf{p} + \mathbf{c}_w = \mathbf{s}_w \\ & \mathbf{l}_0 \leq \mathbf{p} \leq \mathbf{u}_0 \\ & \mathbf{s}_w \geq \mathbf{0} \\ & \mathbf{s} \geq \mathbf{0} \\ & \mathbf{t} \geq \mathbf{0} \\ & \mathbf{S}\mathbf{t} = \mathbf{0} \end{aligned} \tag{18}$$

where $\mathbf{l}_0 = \max(\mathbf{l} - \mathbf{x}_k, -\Delta)$, $\mathbf{u}_0 = \min(\mathbf{u} - \mathbf{x}_k, \Delta)$.

Here notice that we use both $\mathbf{s}$ and $\mathbf{t}$ as penalty terms, because this allows us to apply not only inequality constraints but also equality constraints for $\mathbf{c}$. If we want $\mathbf{c}$ to be an equality constraint, then we set $\gamma_t > 0$ and $\gamma_s > 0$, and $\mathbf{s} + \mathbf{t} = |\mathbf{A}\mathbf{p} + \mathbf{c}|$ is the constraint violation to be added as penalty. Otherwise, if we want to keep $\mathbf{c}$ as an inequality constraint, then set $\gamma_t > 0, \gamma_s = 0$, and only $\mathbf{t}$ will be added as penalty.

Moving the inequality constraints in equation (**??**) into the objective function using log barrier function, the barrier problem can be formulated:

$$\begin{aligned} \min_{\mathbf{p},\mathbf{s},\mathbf{t},\mathbf{s}_w} \quad & f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} + \boldsymbol{\gamma}_t^T\mathbf{t} + \boldsymbol{\gamma}_s^T\mathbf{s} - \mu\left[\log\mathbf{s} + \log\mathbf{t} + \log\mathbf{s}_w + \log(\mathbf{p} - \mathbf{l}_0) + \log(\mathbf{u}_0 - \mathbf{p})\right] \\ \text{such that} \quad & \mathbf{A}\mathbf{p} + \mathbf{c} = \mathbf{s} - \mathbf{t} \\ & \mathbf{A}_w\mathbf{p} + \mathbf{c}_w = \mathbf{s}_w \\ & \mathbf{S}\mathbf{t} = \mathbf{0} \end{aligned} \tag{19}$$

Finally, it can be shown that this problem can be further simplified by dropping the last constraint (see appendix **??** for proof):

$$\begin{aligned} \min_{\mathbf{p},\mathbf{s},\mathbf{t},\mathbf{s}_w} \quad & f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} + \boldsymbol{\gamma}_t^T\mathbf{t} + \boldsymbol{\gamma}_s^T\mathbf{s} - \mu\left[\log\mathbf{s} + \log\mathbf{t} + \log\mathbf{s}_w + \log(\mathbf{p} - \mathbf{l}_0) + \log(\mathbf{u}_0 - \mathbf{p})\right] \\ \text{such that} \quad & \mathbf{A}\mathbf{p} + \mathbf{c} = \mathbf{s} - \mathbf{t} \\ & \mathbf{A}_w\mathbf{p} + \mathbf{c}_w = \mathbf{s}_w \end{aligned} \tag{20}$$

This is the actual form of problem that ParOpt interior point optimizer will solve. To remain with the l_1 penalization, the l_1 merit function

$$\phi(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\gamma}_s^T \max(\mathbf{0}, \mathbf{c}(\mathbf{x})) + \boldsymbol{\gamma}_t^T \max(\mathbf{0}, -\mathbf{c}(\mathbf{x})) \tag{21}$$

and model merit function

$$q(\mathbf{p}) = f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} + \boldsymbol{\gamma}_s^T \max(\mathbf{0}, \mathbf{A}\mathbf{p} + \mathbf{c}) + \boldsymbol{\gamma}_t^T \max(\mathbf{0}, -(\mathbf{A}\mathbf{p} + \mathbf{c})) \tag{22}$$

are used to decide if the solution $\mathbf{p}$ can be accepted or rejected. The step acceptance is determined by computing the ratio:

$$\rho = \frac{\text{actual reduction}}{\text{model reduction}} = \frac{\phi(\mathbf{x}) - \phi(\mathbf{x}+\mathbf{p})}{q(\mathbf{0}) - q(\mathbf{p})} \tag{23}$$

3.1 Update penalty parameter

In order to obtain good performance, the penalty parameter γ must be chosen carefully at each iteration. In ParOpt, the steering method is used to update γ in an adaptive way. The idea of this steering method is to choose the penalty parameter γ at each iteration such that it is small enough to make sure we are decreasing the objective function, and large enough to cause the step to make sufficient progress in the linearized feasibility. It is a strategy that ensures balanced progress toward feasibility and optimality.

Define model constraint violation by:

$$m_k(\mathbf{p}) = \gamma_s^T \max(\mathbf{0}, \mathbf{A}\mathbf{p}+\mathbf{c}) + \gamma_t^T \max(\mathbf{0}, -(\mathbf{A}\mathbf{p}+\mathbf{c})) \tag{24}$$

Then, the objective of l_1 penalty problem (**??**) can be written as

$$q_k(\mathbf{p}) = f + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{B}\mathbf{p} + m_k(\mathbf{p}) \tag{25}$$

In every trust region iteration, besides the original l_1 penalty problem (**??**), the following linear programming problem is also solved:

$$\begin{aligned} \min_{\mathbf{p}} \quad & m_k(\mathbf{p}) \\ \text{such that} \quad & \mathbf{l} \le \mathbf{x}_k + \mathbf{p} \le \mathbf{u} \\ & ||\mathbf{p}||_\infty \le \Delta_k \end{aligned} \tag{26}$$

In practice this is done by setting the penalty parameter γ in the original problem (**??**) to a very large value (e.g. $\gamma = 10^6$) and recall the optimization routine to approximately solve the linear problem (**??**). The solutions to the original penalty problem and linear programming problem are denoted as $\mathbf{p}_k$ and $\mathbf{p}_\infty$, respectively. Then $m_k(0) - m_k(\mathbf{p}_\infty)$ is the best constraint infeasibility reduction that we can achieve within the trust region radius regardless the objective function, and $m_k(0) - m_k(\mathbf{p}_k)$ is the model infeasibility reduction at $\mathbf{p}_k$.

Next, we may update γ_{k+1} based on these results. The general ideas are: if $m_k(0) \approx 0$, meaning that we've already been in the feasible region, then we might want to decrease γ_{k+1} such that we can decrease the objective function itself faster without worrying about constraint infeasibility because we are using interior point method. If $m_k(0) > 0$, and model infeasibility reduction is smaller than best reduction, meaning that the constraint penalization is not significant enough and we want to move faster towards feasible region. Thus we might want to increase γ_{k+1}.

Finally, the complete trust region Sl_1QP algorithm with adaptive γ update is presented in figure (**??**).

Choose stop criteria $\varepsilon_{infeas}, \varepsilon_{KKTerror} > 0$
Choose trust region acceptance ratio $\eta \in (0,1)$, trust region radii $\Delta_{max}, \Delta_{min}$
Choose penalty parameters $\gamma_{max}, \gamma_{min}$
Choose initial $\mathbf{x}_1, \Delta_1, \gamma_1$
for $k = 1, 2, ..., \max\_iter$ **do**
    Evaluate $f_k, \mathbf{c}_k, \mathbf{g}_k, \mathbf{B}_k, \mathbf{A}_k, \mathbf{z}_k$
    if infeasibility $< \varepsilon_{infeas}$ and KKTerror $< \varepsilon_{KKTerror}$ **then**
        break
    end if
    Solve the trust region subproblem to get update $\mathbf{p}_k$
    Compute the ratio ρ_k (equation **??**)
    if $\rho_k \geq \eta$ or $\Delta_k = \Delta_{min}$ **then**
        Accept the trust region update: set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$
    else
        Reject the trust region update: set $\mathbf{x}_{k+1} = \mathbf{x}_k$
    end if
    if $\rho_k < 0.25$ **then**
        Set $\Delta_{k+1} = \max(0.25\Delta_k, \Delta_{min})$
    else if $\rho_k > 0.75$ **then**
        Set $\Delta_{k+1} = \min(1.5\Delta_k, \Delta_{max})$
    else
        $\Delta_{k+1} = \Delta_k$
    end if
    Solve the linear programming problem (**??**) to get $\mathbf{p}_\infty$
    Compute $m_k(0) - m_k(\mathbf{p}_k)$ and $m_k(0) - m_k(\mathbf{p}_\infty)$
    for $i = 1, 2, ...$ **do**
        if $m_k(0)_i < \varepsilon_{infeas}$ and $0.5\gamma_{ki} \geq \mathbf{z}_{ki} > \varepsilon_{infeas}$ **then**
            Reduce penalty parameter: $\gamma_{(k+1)_i} = 0.5(\gamma_{ki} + \mathbf{z}_{ki}) + \gamma_{min}$
        else if $m_k(0)_i > \varepsilon_{infeas}$ and $0.995(m_k(0) - m_k(\mathbf{p}_\infty))_i > (m_k(0) - m_k(\mathbf{p}_k))_i$ **then**
            Increase penalty parameter: $\gamma_{(k+1)_i} = \min(1.5\gamma_{ki}, \gamma_{max})$
        else
            $\gamma_{(k+1)_i} = \gamma_{ki}$
        end if
    end for
end for

Figure 2: Trust region Sl_1QP algorithm with adaptive γ update

4 Implementation details

The two main classes need by users of ParOpt are `ParOptProblem` and `ParOpt`. These classes are both accessible using Python through a Cython wrapper. The `ParOptProblem` class is an abstract base class that is designed to implement the functions needed to solve an optimization problem. It is responsible for evaluating functions and constraints as well as allocating parallel design and sparse constraint vectors. The python-level implementation of this class uses a default vector implementation where the components of the vector are distributed across all processors in the communicator provided to ParOpt.

The `ParOpt` class itself is responsible for optimizing the problem. All options are set through public member-access functions that can be called from the python or C++ interfaces. These functions will be described below.

4.1 ParOptProblem class methods

The constructor for the `ParOptProblem` class takes the `MPI_Comm` communicator object:

```
ParOptProblem(MPI_Comm _comm);
```

Additionally, in the constructor of your base class, you will have to set the following information:

- `comm` is the MPI communicator for the problem. This communicator will be passed to the corresponding `ParOpt` optimizer class.
- `nvars`: The local number of design variables that are owned by this processor.
- `ncon`: The global number of dense constraints.
- `nwcon`: The local number of sparse constraints that are owned by this processor

The following functions are used to specify the structure of the design problem:

1. `int useLowerBounds()`: Should the optimizer use the lower bound constraints?
2. `int useUpperBounds()`: Should the optimizer use the upper bound constraints?

4.1.1 ParOptProblem evaluation functions

The following evaluation member functions must be defined. Their arguments are omitted here, but can be found in the file `src/ParOptProblem.h`.

1. `getVarsAndBounds`: Get the design variables at the starting point as well as the lower and upper bounds for the variables. This is called when `ParOpt` is initialized.
2. `int evalObjCon`: Given the design variable values, evaluate the objective and dense constraint functions. This returns a fail flag. When the fail flag is non-zero, the function has failed.

3. `int evalObjConGradient`: Given the design variable values, evaluate the objective and constraint gradients. The objective is a single `ParOptVec` instance. The dense constraint Jacobian is an array of `ParOptVec` instances. This function also returns a fail flag.

4. `int evalHvecProduct`: Given the design variable values, the multipliers for the dense and sparse constraints, and a direction in the design space, compute the Hessian-vector product. Note that this function combines computations using the sparse and dense constraints. This function also returns a fail flag.

4.1.2 ParOptProblem sparse constraint functions

The following functions are used exclusively for the sparse constraints

1. `evalSparseCon`: Given the design variables, evaluate the sparse constraints.

2. `addSparseJacobian` Given a scalar, the design variables, and an input direction vector the size of the number of design variables, compute the scaled Jacobian-vector product of the sparse constraints.

3. `addSparseJacobianTranspose` Given a scalar, the design variables, and an input direction vector the same size as the number of sparse constraints, compute the scaled transpose Jacobian-vector product of the sparse constraints. This function is required for computing the product $\mathbf{A}_w^T \mathbf{z}_w$.

4. `addSparseInnerProduct` Given a scalar, the design variable vector, and the diagonal of a square matrix the size of the number of sparse constraints, compute add the product, $\mathbf{D} \leftarrow \mathbf{D} + \alpha \mathbf{A}_w \mathbf{S} \mathbf{A}_w^T$.

ParOpt implements a default vector class. It is often convenient or necessary to override this class for certain design problems. ParOptProblem can override the default implementation if the following two functions are provided:

1. `ParOptVec *createDesignVec()`: Create a vector with the right type and shape to store design variables/objective gradients.

2. `ParOptVec *createConstraintVec()`: Create a vector with the right type and shape to store a sparse constraint vector

The function `void writeOutput( int iter, ParOptVec *x )` can be overridden to write out design-dependent data at a specified frequency.

4.2 ParOptOptimizer interface class

`ParOptOptimizer` is a generic interface to all ParOpt optimization algorithms. This class should be used to initialize and run ParOpt optimizers. The process to initialize, optimize and retrieve results is as follows:

1. Retrieve the default options for all ParOpt optimizers by calling the static member function `static void addDefaultOptions( ParOptOptions *options );`.

2. Instantiate the `ParOptOptimizer` class with a problem instance by calling `ParOptOptimizer( ParOptProblem *problem, ParOptOptions *options );`

3. Perform the optimization by calling the member function `void optimize();`

4. Get the optimized point and multiplier values by calling the member function

```
void getOptimizedPoint( ParOptVec **x, ParOptScalar **z,
                        ParOptVec **zw, ParOptVec **zl, ParOptVec **zu );
```

4.3 Default values of ParOptOptions

The full list of ParOpt options can be obtained by using the python interface and entering the following command:

```
python -c 'from paropt import ParOpt; ParOpt.printOptionSummary()'
```

Note that options with a prefix `tr_` apply generally to the trust region method, while options with `mma_` apply generally to the method of moving asymptotes. Options without either of these prefixes apply to the interior point method. The above command gives the following output:

```
Absolute stopping criterion
abs_res_tol                             1e-06
Range of values: lower limit 0  upper limit 1e+20

Absolute stopping norm on the step size
abs_step_tol                            0
Range of values: lower limit 0  upper limit 1e+20

The type of optimization algorithm
algorithm                               tr
Range of values:                        ip
                                        tr
                                        mma

The Armijo constant for the line search
armijo_constant                         1e-05
Range of values: lower limit 0  upper limit 1

The type of barrier update strategy to use
barrier_strategy                        monotone
Range of values:                        monotone
                                        mehrotra
                                        mehrotra_predictor_corrector
                                        complementarity_fraction

The absolute precision of the design variables
design_precision                        1e-14
Range of values: lower limit 0  upper limit 1

Exponent in the Eisenstat-Walker INK forcing equation
eisenstat_walker_alpha                  1.5
```

```
Range of values: lower limit 0  upper limit 2

Multiplier in the Eisenstat-Walker INK forcing equation
eisenstat_walker_gamma                  1
Range of values: lower limit 0  upper limit 1

A small value that controls slanting envelope of the filter
filter_gamma                            1e-05
Range of values: lower limit 0  upper limit 1

Use feasibility restoration for filter method
filter_has_feas_restore_phase           True

Use sufficient reduction criteria for filter
filter_sufficient_reduction             True

The absolute precision of the function and constraints
function_precision                      1e-10
Range of values: lower limit 0  upper limit 1

The absolute GMRES tolerance (almost never relevant)
gmres_atol                              1e-30
Range of values: lower limit 0  upper limit 1

The subspace size for GMRES
gmres_subspace_size                     0
Range of values: lower limit 0  upper limit 1000

Step length used to check the gradient
gradient_check_step_length              1e-06
Range of values: lower limit 0  upper limit 1

Print to screen the output of the gradient check at this frequency during an optimization
gradient_verification_frequency         -1
Range of values: lower limit -1000000  upper limit 1000000

Do a hard reset of the Hessian at this specified major iteration frequency
hessian_reset_freq                      1000000
Range of values: lower limit 1  upper limit 1000000

The initial value of the barrier parameter
init_barrier_param                      0.1
Range of values: lower limit 0  upper limit 1e+20

Initial value of the line search penalty parameter
init_rho_penalty_search                 0
Range of values: lower limit 0  upper limit 1e+20

Checkpoint file for the interior point method
ip_checkpoint_file                      None

Maximum bound value at which bound constraints are omitted
max_bound_value                         1e+20
Range of values: lower limit 0  upper limit 1e+300
```

```
The maximum relative tolerance used for GMRES, above this the quasi-Newton approximation is used
max_gmres_rtol                          0.1
Range of values: lower limit 0  upper limit 1

Maximum number of line search iterations
max_line_iters                          10
Range of values: lower limit 1  upper limit 100

The maximum number of major iterations before quiting
max_major_iters                         5000
Range of values: lower limit 0  upper limit 1000000

Minimum fraction to the boundary rule < 1
min_fraction_to_boundary                0.95
Range of values: lower limit 0  upper limit 1

Minimum value of the line search penalty parameter
min_rho_penalty_search                  0
Range of values: lower limit 0  upper limit 1e+20

Contraction factor applied to the asymptotes
mma_asymptote_contract                  0.7
Range of values: lower limit 0  upper limit 1

Expansion factor applied to the asymptotes
mma_asymptote_relax                     1.2
Range of values: lower limit 1  upper limit 1e+20

Relaxation bound for computing the error in the KKT conditions
mma_bound_relax                         0
Range of values: lower limit 0  upper limit 1e+20

Regularization term applied in the MMA approximation
mma_delta_regularization                0.001
Range of values: lower limit 0  upper limit 1e+20

Regularization term applied in the MMA approximation
mma_eps_regularization                  1e-05
Range of values: lower limit 0  upper limit 1e+20

Infeasibility tolerance
mma_infeas_tol                          1e-05
Range of values: lower limit 0  upper limit 1e+20

Initial asymptote offset from the variable bounds
mma_init_asymptote_offset               0.5
Range of values: lower limit 0  upper limit 1

l1 tolerance for the optimality tolerance
mma_l1_tol                              1e-06
Range of values: lower limit 0  upper limit 1e+20

l-infinity tolerance for the optimality tolerance
```

```
mma_linfty_tol                          1e-06
Range of values: lower limit 0  upper limit 1e+20

Maximum asymptote offset from the variable bounds
mma_max_asymptote_offset                10
Range of values: lower limit 0  upper limit 1e+20

Maximum number of iterations
mma_max_iterations                      200
Range of values: lower limit 0  upper limit 1000000

Minimum asymptote offset from the variable bounds
mma_min_asymptote_offset                0.01
Range of values: lower limit 0  upper limit 1e+20

Move limit for design variables to prevent oscillation
mma_move_limit                          0.2
Range of values: lower limit 0  upper limit 1e+20

Ouput file name for MMA
mma_output_file                         paropt.mma

Use a linearization of the constraints in the MMA subproblem
mma_use_constraint_linearization        False

Factor applied to the barrier update < 1
monotone_barrier_fraction               0.25
Range of values: lower limit 0  upper limit 1

Exponent for barrier parameter update > 1
monotone_barrier_power                  1.1
Range of values: lower limit 1  upper limit 10

Switch to the Newton-Krylov method at this residual tolerance
nk_switch_tol                           0.001
Range of values: lower limit 0  upper limit 1e+20

The type of norm to use in all computations
norm_type                               infinity
Range of values:                        infinity
                                        l1
                                        l2


Output file name
output_file                             paropt.out

Output level indicating how verbose the output should be
output_level                            0
Range of values: lower limit 0  upper limit 1000000

Fraction of infeasibility used to enforce a descent direction
penalty_descent_fraction                0.3
Range of values: lower limit 1e-06  upper limit 1
```

```
l1 penalty parameter applied to slack variables
penalty_gamma                           1000
Range of values: lower limit 0  upper limit 1e+20

The problem name
problem_name                            None

The type of initial diagonal to use in the quasi-Newton approximation
qn_diag_type                            yty_over_yts
Range of values:                        yty_over_yts
                                        yts_over_sts
                                        inner_yty_over_yts
                                        inner_yts_over_sts

Scalar added to the diagonal of the quasi-Newton approximation > 0
qn_sigma                                0
Range of values: lower limit 0  upper limit 1e+20

The maximum dimension of the quasi-Newton approximation
qn_subspace_size                        10
Range of values: lower limit 0  upper limit 1000

The type of quasi-Newton approximation to use, note that scaled_bfgs
should be only used when there's single constraint and objective is linear
qn_type                                 bfgs
Range of values:                        bfgs
                                        scaled_bfgs
                                        sr1
                                        none

The type of BFGS update to apply when the curvature condition fails
qn_update_type                          skip_negative_curvature
Range of values:                        skip_negative_curvature
                                        damped_update

Relative factor applied to barrier parameter for bound constraints
rel_bound_barrier                       1
Range of values: lower limit 0  upper limit 1e+20

Relative function value stopping criterion
rel_func_tol                            0
Range of values: lower limit 0  upper limit 1e+20

Discard the quasi-Newton approximation (but not necessarily the exact Hessian)
sequential_linear_method                False

Minimum multiplier for the affine step initialization strategy
start_affine_multiplier_min             1
Range of values: lower limit 0  upper limit 1e+20

Initialize the Lagrange multiplier estimates and slack variables
starting_point_strategy                 affine_step
Range of values:                        least_squares_multipliers
                                        affine_step
```

```
                                        no_start_strategy

Which strategy to use to decide if a trial point can be accepted or not
tr_accept_step_strategy                 penalty_method
Range of values:                        penalty_method
                                        filter_method

The type of constraint to use for the adaptive penalty subproblem
tr_adaptive_constraint                  linear_constraint
Range of values:                        linear_constraint
                                        subproblem_constraint

Adaptive penalty parameter update
tr_adaptive_gamma_update                True

The type of objective to use for the adaptive penalty subproblem
tr_adaptive_objective                   linear_objective
Range of values:                        constant_objective
                                        linear_objective
                                        subproblem_objective

Upper and lower bound relaxing parameter
tr_bound_relax                          0.0001
Range of values: lower limit 0  upper limit 1e+20

Trust region trial step acceptance ratio
tr_eta                                  0.25
Range of values: lower limit 0  upper limit 1

Infeasibility tolerance
tr_infeas_tol                           1e-05
Range of values: lower limit 0  upper limit 1e+20

The initial trust region radius
tr_init_size                            0.1
Range of values: lower limit 0  upper limit 1e+20

l1 tolerance for the optimality tolerance
tr_l1_tol                               1e-06
Range of values: lower limit 0  upper limit 1e+20

l-infinity tolerance for the optimality tolerance
tr_linfty_tol                           1e-06
Range of values: lower limit 0  upper limit 1e+20

Maximum number of trust region iterations
tr_max_iterations                       200
Range of values: lower limit 0  upper limit 1000000

The maximum trust region radius
tr_max_size                             1
Range of values: lower limit 0  upper limit 1e+20

Maximum number of trust region iterations
```

```
tr_max_soc_iterations                    20
Range of values: lower limit 0  upper limit 1000000

The minimum trust region radius
tr_min_size                              0.001
Range of values: lower limit 0  upper limit 1e+20

Trust region output file
tr_output_file                           paropt.tr

Maximum value for the penalty parameter
tr_penalty_gamma_max                     10000
Range of values: lower limit 0  upper limit 1e+20

Minimum value for the penalty parameter
tr_penalty_gamma_min                     0
Range of values: lower limit 0  upper limit 1e+20

Update quasi-Newton approximation in second order correction steps
tr_soc_update_qn                         False

The barrier update strategy to use for the steering method subproblem
tr_steering_barrier_strategy             mehrotra_predictor_corrector
Range of values:                         monotone
                                         mehrotra
                                         mehrotra_predictor_corrector
                                         complementarity_fraction
                                         default

The barrier update strategy to use for the steering method subproblem
tr_steering_starting_point_strategy      affine_step
Range of values:                         least_squares_multipliers
                                         affine_step
                                         no_start_strategy
                                         default

Use second order correction when trial step is rejeccted
tr_use_soc                               False

Write output frequency
tr_write_output_frequency                10
Range of values: lower limit 0  upper limit 1000000

Perform a back-tracking line search
use_backtracking_alpha                   False

Use or do not use the diagonal Hessian computation
use_diag_hessian                         False

Use or do not use Hessian-vector products
use_hvec_product                         False

Perform or skip the line search
use_line_search                          True
```

```
Use or do not use the quasi-Newton method as a preconditioner
use_qn_gmres_precon                     True

Update the quasi-Newton approximation at each iteration
use_quasi_newton_update                 True

Write out the solution file and checkpoint file at this frequency
write_output_frequency                  10
Range of values: lower limit 0  upper limit 1000000
```

References

R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(1-3):129–156, 1994. ISSN 0025-5610. doi:10.1007/BF01582063.

A. V. Fiacco and G. P. McCormick. *Nonlinear Programming*. Society for Industrial and Applied Mathematics, 1990. doi:10.1137/1.9781611971316.

J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2nd edition, 2006.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, May 2006. ISSN 0025-5610. doi:10.1007/s10107-004-0559-y.

A Equivalence of the two barrier problems

We would like to show that for the following two optimization problems, we can replace the first one by second one.

$$\begin{aligned} \min_{\mathbf{x},\mathbf{s},\mathbf{t}} \quad & f(\mathbf{x}) + \gamma \mathbf{e}^T \mathbf{t} - \mu[\log \mathbf{s} + \log \mathbf{t}] \\ \text{such that} \quad & \mathbf{c}(\mathbf{x}) = \mathbf{s} - \mathbf{t} \\ & \mathbf{S}\mathbf{t} = 0 \end{aligned} \tag{27}$$

$$\begin{aligned} \min_{\mathbf{x},\mathbf{s},\mathbf{t}} \quad & f(\mathbf{x}) + \gamma \mathbf{e}^T \mathbf{t} - \mu[\log \mathbf{s} + \log \mathbf{t}] \\ \text{such that} \quad & \mathbf{c}(\mathbf{x}) = \mathbf{s} - \mathbf{t} \end{aligned} \tag{28}$$

In fact, problem (**??**) and problem (**??**) have the equivalent KKT systems. For problem (**??**), define the Lagrangian:

$$\mathscr{L}_1(\mathbf{x},\mathbf{s},\mathbf{t};\mathbf{y},\mathbf{z}) = f(\mathbf{x}) + \gamma \mathbf{e}^T \mathbf{t} - \mu[\log \mathbf{s} + \log \mathbf{t}] - \mathbf{y}^T[\mathbf{c}(\mathbf{x}) - \mathbf{s} + \mathbf{t}] - \mathbf{z}^T \mathbf{S}\mathbf{t} \tag{29}$$

Then the KKT system is:

$$\begin{aligned}
\nabla_{\mathbf{x}}\mathscr{L}_1 &= \nabla f(\mathbf{x}) - \mathbf{A}^T(\mathbf{x})\mathbf{y} &&= 0\\
\nabla_{\mathbf{s}}\mathscr{L}_1 &= -\mu\mathbf{S}^{-1}\mathbf{e} + \mathbf{y} - \mathbf{Z}\mathbf{t} &&= 0\\
\nabla_{\mathbf{t}}\mathscr{L}_1 &= \gamma\mathbf{e} - \mu\mathbf{T}^{-1}\mathbf{e} - \mathbf{y} - \mathbf{Z}\mathbf{s} &&= 0\\
\nabla_{\mathbf{y}}\mathscr{L}_1 &= \mathbf{c}(\mathbf{x}) - \mathbf{s} + \mathbf{t} &&= 0\\
\nabla_{\mathbf{z}}^T\mathscr{L}_1 &= \mathbf{S}\mathbf{t} &&= 0
\end{aligned} \tag{30}$$

Note that here diagonal matrices $\mathbf{T}$ and $\mathbf{Z}$ are defined in the same way as in (**??**). For the second and third equation, multiply through by $\mathbf{S}$ and $\mathbf{T}$, respectively, also notice that since $\mathbf{S}, \mathbf{T}$ and $\mathbf{Z}$ are diagonal matrices, we have $\mathbf{SZt} = \mathbf{ZSt}, \mathbf{TZs} = \mathbf{ZSt}$. Then we can eliminate the last terms in these two equations, because we have $\mathbf{St} = \mathbf{0}$. Finally, multiply through by $\mathbf{S}^{-1}$ and $\mathbf{T}^{-1}$, respectively, then the KKT system (**??**) becomes:

$$\begin{aligned}
\nabla f(\mathbf{x}) - \mathbf{A}^T(\mathbf{x})\mathbf{y} &= 0\\
-\mu\mathbf{S}^{-1}\mathbf{e} + \mathbf{y} &= 0\\
\gamma\mathbf{e} - \mu\mathbf{T}^{-1}\mathbf{e} - \mathbf{y} &= 0\\
\mathbf{c}(\mathbf{x}) - \mathbf{s} + \mathbf{t} &= 0
\end{aligned} \tag{31}$$

And this turns out to be the KKT system for problem (**??**), which means that we can get the solution to the problem (**??**) by solving the problem (**??**), thus these two optimization problems are equivalent in this sense. Also notice that from (**??**) to (**??**) we exclude the last equation $\mathbf{St} = \mathbf{0}$, this is because we have removed the multiplier $\mathbf{z}$ from the problem, then the last equation of (**??**) is redundant for (**??**).