

ParOpt: A parallel interior-point optimizer

Graeme J. Kennedy

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 Algorithms

ParOpt uses 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, $\mathbf{A}_w = \nabla_{\mathbf{x}} \mathbf{c}_w(\mathbf{x})$ 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

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

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:

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

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

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 $\mathbf{B} \approx \mathbf{H} \triangleq \nabla_x^2 (f(\mathbf{x}) - \mathbf{z}^T \mathbf{c}(\mathbf{x}) - \mathbf{z}_w^T \mathbf{c}_w(\mathbf{x}))$. The approximate KKT matrix is

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

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 $\mu_k \rightarrow 0$ for $k \rightarrow \infty$. 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:

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

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 v 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

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

with analogous expressions for the remaining components of the step length vector \mathbf{p} . 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 $\mathbf{p}_z^s = \alpha_z \mathbf{p}_z$. Note that α_x is the step length for the design and slack variables, and α_z is the step length for the Lagrange 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 $\alpha_x \gg \alpha_z$, 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:

$$\mathbf{K}_B \mathbf{p} = [\mathbf{K}_0 + \mathbf{Y} \mathbf{M} \mathbf{Y}^T] \mathbf{p} = -\mathbf{r} \quad (7)$$

where the matrix \mathbf{Y} is

$$\mathbf{Y}^T = [\mathbf{W}^T \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0],$$

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:

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

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 refers 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:

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

Next, using the first three equations gives

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

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

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

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

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

Finally, \mathbf{p}_x can be eliminated for \mathbf{p}_z and \mathbf{p}_{z_w} as follows

$$\begin{aligned} (\mathbf{Z}_w^{-1} \mathbf{S}_w + \mathbf{A}_w \mathbf{D}^{-1} \mathbf{A}_w^T) \mathbf{p}_{z_w} + \mathbf{A}_w \mathbf{D}^{-1} \mathbf{A}^T \mathbf{p}_z &= \mathbf{d}_{z_w}, \\ \mathbf{A} \mathbf{D}^{-1} \mathbf{A}_w^T \mathbf{p}_{z_w} + (\mathbf{Z}^{-1} \mathbf{S} + \mathbf{Z}_t^{-1} \mathbf{T} + \mathbf{A} \mathbf{D}^{-1} \mathbf{A}^T) \mathbf{p}_z &= \mathbf{d}_z, \end{aligned} \quad (11)$$

The right hand sides, \mathbf{d}_{z_w} and \mathbf{d}_z are

$$\begin{aligned} \mathbf{d}_{z_w} &\triangleq \mathbf{b}_{z_w} + \mathbf{Z}_w^{-1} \mathbf{b}_{s_w} - \mathbf{A}_w \mathbf{D}^{-1} (\mathbf{b}_x + (\mathbf{X} - \mathbf{L})^{-1} \mathbf{b}_{z_l} - (\mathbf{U} - \mathbf{X})^{-1} \mathbf{b}_{z_u}), \\ \mathbf{d}_z &\triangleq \mathbf{b}_z + \mathbf{Z}^{-1} \mathbf{b}_s - \mathbf{Z}_t^{-1} (\mathbf{b}_{z_t} + \mathbf{T} \mathbf{b}_t) - \mathbf{A} \mathbf{D}^{-1} (\mathbf{b}_x + (\mathbf{X} - \mathbf{L})^{-1} \mathbf{b}_{z_l} - (\mathbf{U} - \mathbf{X})^{-1} \mathbf{b}_{z_u}). \end{aligned}$$

Finally, defining the matrix $\mathbf{E} \triangleq (\mathbf{Z}_w^{-1} \mathbf{S}_w + \mathbf{A}_w \mathbf{D}^{-1} \mathbf{A}_w^T)$, which is block diagonal, and introducing $\mathbf{F} \triangleq \mathbf{A} \mathbf{D}^{-1} \mathbf{A}_w^T$, gives the following equation

$$[\mathbf{Z}^{-1} \mathbf{S} + \mathbf{Z}_t^{-1} \mathbf{T} + \mathbf{A} \mathbf{D}^{-1} \mathbf{A}^T - \mathbf{F} \mathbf{E}^{-1} \mathbf{F}^T] \mathbf{p}_z = \mathbf{d}_z - \mathbf{E}^{-1} \mathbf{d}_{z_w}$$

This solution procedure is invoked each time a solution of the form $\mathbf{K}_0 \mathbf{p} = \mathbf{b}$ is required.

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

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

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

$$\mathbf{N}^T = [\mathbf{I} \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$$

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

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

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, α_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, α_1) and (\mathbf{v}_2, α_2) can be expressed as

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

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

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

$$\mathbf{p}_x^T \nabla_x f < 0, \quad (13)$$

or

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

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

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

GMRES(m, γ): Inexactly solve $\mathbf{K}\mathbf{p} = \mathbf{b}$ 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 $\beta_r = \mathbf{b}^T (\mathbf{I} - \mathbf{N}\mathbf{N}^T) \mathbf{b}$
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 $\mathbf{a}_k = (a_{k-1}, a_k)$ and $\mathbf{b}_k = (b_{k-1}, b_k)$
 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
 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

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

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

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.

3 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.

3.1 ParOptProblem class methods

The constructor for the ParOptProblem class takes the following form:

```
ParOptProblem( MPI_Comm _comm,  
               int _nvars, int _ncon,  
               int _nwcon, int _nwblock );
```

The arguments to this constructor are:

- `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
- `nwblock`: The size of the block in block-diagonal matrix formed from the weighting constraints $\mathbf{A}_w^T \mathbf{A}_w$. This size must be the same on all processors.

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

1. `int isDenseInequality()`: Are the dense constraints inequality or equality constraints?
2. `int isSparseInequality()`: Are the sparse constraints inequality or equality constraints?
3. `int useLowerBounds()`: Should the optimizer use the lower bound constraints?
4. `int useUpperBounds()`: Should the optimizer use the upper bound constraints?

3.1.1 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.

3.1.2 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.

3.2 ParOpt class methods

The prototype for the constructor for the ParOpt class is:

```
ParOptInteriorPoint( ParOptProblem *_prob,  
                    int _max_qn_subspace,  
                    QuasiNewtonType qn_type=BFGS,  
                    double _max_bound_val=1e20 );
```

The ParOptProblem defines the problem size, structure, and serves as a factory for generating problem-specific vectors. The max_qn_subspace denotes the maximum size of the quasi-Newton subspace while qn_type denotes the type of quasi-Newton method. Note that these parameters must be specified at initialization.

The following are the primary member functions used in ParOpt:

1. `int optimize(const char *checkpoint=NULL)`: Begin the optimization algorithm and use the given checkpoint file. This binary file can be used to restart the optimization from the last iterate with the same design variables, barrier parameter and multipliers. This does not retain the same set of optimization parameters.
2. `void checkGradients(double dh)`: Check the gradients of the objective and constraints as well as the Hessian-vector products if they are defined. This call should be made when debugging the optimization problem. This check uses finite-difference or complex-step method if the code is compiled in complex mode.
3. `void setMaxMajorIterations(int iters)`: Set the maximum number of iterations to use during the optimization
4. `void setAbsOptimalityTol(double tol)`: Set the absolute optimality tolerance used within ParOpt. This is an absolute tolerance on the ℓ_∞ norm of the KKT conditions.
5. `void setRelFunctionTol(double tol)`: Set the relative function tolerance stopping criterion. ParOpt will terminate a barrier problem when the decrease in the objective function value falls below the specified tolerance. This defaults to zero, in which case the criteria is never satisfied.
6. `void setInitBarrierParameter(double mu)`: Set the initial barrier parameter
7. `void setMaxLineSearchIters(int iters)`: Set the maximum number of iterations before the line search fails
8. `void setBacktrackingLineSearch(int truth)`: Use a backtracking line search rather than
9. `void setArmijoParam(double c1)`: Set the Armijo parameter in the line search acceptance criteria (first Wolfe condition).
10. `void setOutputFrequency(int freq)`: Set the output iteration frequency. This is the frequency at which the `writeOutput` function is called in ParOptProblem.

11. `void setOutputFile(const char *filename)`: Set the output file name for the optimization summary file. This file is flushed at the same frequency as the `writeOutput` call.
12. `int writeSolutionFile(const char *filename)`: Write a binary solution file containing the multipliers/design variables
13. `int readSolutionFile(const char *filename)`: Read a binary solution file and set the variables internally. This requires that the problem sizes are exactly the same as the original problem. A different number of processors may be used if the design variables and multipliers are numbered consistently.

3.2.1 Advanced ParOpt functions

The following are more advanced options/member functions needed for specific applications

1. `getInitMultipliers`: Get the multiplier vectors stored internally in ParOpt. This is needed if you have specific knowledge of what the best multiplier estimates should be before starting the optimization algorithm.
2. `getOptimizedPoint`: Get the optimized point including the design variables and multipliers from ParOpt. This can be called at any time to return the current approximate solution from ParOpt.
3. `setMaxAbsVariableBound(double max_bound)`: Set the maximum absolute value of the bound variables in the problem. Components of the **l** and **u** vector that are specified beyond this limit will be ignored by ParOpt.
4. `void setInitStartingPoint(int init)`: Tells ParOpt whether it should attempt to guess the initial multipliers or use the values that are set in ParOpt, possibly with a call to `getInitMultipliers`.
5. `void setBarrierFraction(double frac)`: Set the fraction applied during the Fiacco–McCormick barrier parameter update scheme
6. `void setBarrierPower(double power)`: Set the barrier power applied during the Fiacco–McCormick barrier parameter update scheme
7. `double getBarrierParameter()`: Get the barrier parameter
8. `ParOptScalar getComplementarity()`: Compute the average complementarity product at the current design point
9. `void setUseLineSearch(int truth)`: Set a flag to indicate whether a line search will be performed. If false, then the new point is automatically accepted once the fraction-to-the-boundary rule is applied to the step (to ensure the multipliers/slacks remain positive).
10. `void setHessianResetFreq(int freq)`: Set the frequency at which the Hessian will be reset to a diagonal matrix

11. `void setBFGSUpdateType(LBFGS::BFGSUpdateType bfgs_update)`: Set the type of BFGS update to use when a negative curvature condition is encountered. This will either skip or update the either damped or skip
12. `void setUseHvecProduct(int truth)`: Set the flag to either the Hessian-vector product code. This relies on the implementation of the Hessian-vector product code in the optimization class itself.
13. `void setUseQNGMRESPreCon(int truth)`: Use the BFGS or SR1 update as a preconditioner in the GMRES solver.
14. `void setNKSwitchTolerance(double tol)`: Set the optimality tolerance at which to switch to the full Newton–Krylov method.
15. `void setEisenstatWalkerParameters(double gamma, double alpha)`: Set the Eisenstat–Walker parameters for the forcing term
16. `void setGMRESSubspaceSize(int gmres_subspace_size)`: Set the size of the GMRES subspace to use during the optimization algorithm
17. `void setPenaltyDescentFraction(double frac)`: Set the penalty parameter
18. `void resetQuasiNewtonHessian()`: Force a reset of the quasi-Newton Hessian approximation
19. `void resetDesignAndBounds()`: Force a reset the design variables and bounds within the problem

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](https://doi.org/10.1007/BF01582063).
- A. V. Fiacco and G. P. McCormick. *Nonlinear Programming*. Society for Industrial and Applied Mathematics, 1990. doi:[10.1137/1.9781611971316](https://doi.org/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](https://doi.org/10.1007/s10107-004-0559-y).