Les Cahiers du GERAD ISSN: 0711–2440

A framework around limited-memory partitioned quasi-Newton methods

J. Bigeon, D. Orban, P. Raynaud

G–2023–17

May 2023

A framework around limited-memory partitioned quasi-Newton methods

Jean Bigeon [a, b]

Dominique Orban [a, c]

Paul Raynaud [a, c, d]

[a] *GERAD, Montréal (Qc), Canada, H3T 1J4*

[b] *Nantes Université, École Centrale Nantes, CNRS, LS2N, UMR 6004, 44300 Nantes, France*

[c] *Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Qc), Canada, H3T 1J4*

[d] *GSCOP, Université Grenoble Alpes, 38400 Saint-Martin-d'Hères, France*

`jean.bigeon@ls2n.fr`
`dominique.orban@polymtl.ca`
`paul.raynaud@polymtl.ca`

May 2023
Les Cahiers du GERAD
G–2023–17

Abstract : We present a Julia framework dedicated to partially-separable problems whose element function are detected automatically. This framework takes advantage of the JuliaSmoothOptimizer ecosystem for minimizing unconstrained smooth partially-separable problems with several partitioned quasi-Newton trust-region methods. We provide three new limited-memory partitioned quasi-Newton operators: PLBFGS, PLSR1 and PLSE. These new limited-memory partitioned quasi-Newton operators rely on LBFGS and LSR1 operators to approximate element Hessians instead of dense matrices. Therefore, the memory footprint and the linear application complexity drop from $\Theta(\sum_{i=1}^{N} \frac{n_i(n_i+2)}{2})$ to $\Theta(\sum_{i=1}^{N} 2mn_i)$. Under the assumption that element function gradients are Lipchitz continuous, we establish a global convergence proof for these new methods. The numerical results compare the methods presented to the state of the art of limited-memory or partitioned quasi-Newton methods through performance profiles. To legitimate partitioned limited-memory operators, we study limits of classic partitioned quasi-Newton methods where large element functions exist.

Keywords : Partially-separable methods, limited-memory quasi-Newton, unconstrained smooth-optimization

Acknowledgements: This work has been supported by the laboratory G-SCOP/grenoble-inp and the NSERC Alliance grant 544900–19 in collaboration with Huawei-Canada.

1 Introduction

We consider the unconstrained problem

$$\min_{x \in \mathbb{R}^n} \; f(x), \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$. We assume that f is partially-separable, i.e., has the form

$$f(x) = \sum_{i=1}^{N} \widehat{f}_i(\widehat{x}_i), \quad \widehat{x}_i := U_i x, \tag{2}$$

where each $U_i \in \mathbb{R}^{n_i \times n}$ considers $n_i < n$, and each $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$ is twice continuously differentiable [22, 23]. In other words, each element function $\widehat{f}_i$ only depends linearly on a subset of variables, which is collected by U_i. Partial separability appears in a lot of problems: numerical resolution of partial derivative equations, finite element modeling, nonlinear sparse least square problems. Such structures enable parallel computations [29] and explicits how derivatives are computed or approximated:

$$\nabla f = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i, \quad \nabla^2 f \approx B = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{B}_i U_i, \quad \widehat{B}_i \approx \nabla^2 \widehat{f}_i \in \mathbb{R}^{n_i \times n_i}. \tag{3}$$

B structurally keeps the sparsity of $\nabla^2 f$ [22] and possesses a partitioned matrix vector product $Bv = \sum_{i=1}^{N} U_i \widehat{B}_i U_i^\top v, \; v \in \mathbb{R}^n$. The key to implement an efficiently partitioned linear operator is to store wisely U_i and aggregate $\widehat{B}_i U_i v$ swiftly. When n_i remains small, the memory footprint of a partitioned linear operator is cheaper than a dense matrix, which induces a faster linear application [7].

After defining the notation in Section 2, Section 3 reviews existing pieces of code dedicated to smooth partial separability. The Section 4 recalls notions related to partitioned quasi-Newton trust-region methods and conclude by outlining a quasi-Newton trust-region algorithm. Then, we present in Section 5 how our code integrates into the JuliaSmoothOptimizers (JSO) ecosystem. Divided among several solvers, it fits the abstract typed interface architecture of JSO, to provide several (limited-memory) partitioned quasi-Newton solvers exploiting partial separability at all steps of the algorithm. Section 6 presents new limited-memory partitioned quasi-Newton models. Whereas partitioned quasi-Newton models approximate $\nabla^2 \widehat{f}_i$ by a dense matrix updated with quasi-Newton formulas [22], which becomes an issue when elements become large ; the limited-memory variants rely instead on quasi-Newton linear operators to approximate $\nabla^2 \widehat{f}_i$. While the aggregation of dense element approximations has a memory footprint and linear application in $\Theta(\sum_{i=1}^{N} \frac{n_i(n_i+2)}{2})$, the variant using limited-memory Hessian approximations of memory m get complexities of $\Theta(\sum_{i=1}^{N} 2mn_i)$, $1 \leq m \ll n_i$. We demonstrate the global convergence of the resulting limited-memory partitioned quasi-Newton trust-region methods by bounding the norm of element limited-memory quasi-Newton operators $\|\widehat{B}_i\|$ and the norm of the aggregated operator $\|B\|$ in Section 6.1. Section 7 reports the numerical results about the partitioned optimization methods supported, with a focus on the limited-memory variant for large element problems. We present our future works and conclude in Section 8.

2 Notations

Lowercase letters such as s, x and y denote vectors. Uppercase letters such as B and H denote matrices. Greek letters such as α and λ denote scalars. Throughout, I_n denotes the identity matrix of size n. When there is no ambiguity, I denotes the identity matrix of appropriate size. ϵ_1 and ϵ_2 are numerical safeguards related to quasi-Newton updates. Usually, they are related to the machine's ϵ. $\omega_{\text{SR1}}, \omega_{\text{BFGS}_1}$ and ω_{BFGS_2} are numerical safeguards introduce in Section 6.1 to bound the limited-memory partitioned quasi-Newton updates.

Any notation capped as $\widehat{\cdot}_i$ refers to a structure of size n_i related to the i-th element function. For example, $\widehat{g}_i := \nabla \widehat{f}_i \in \mathbb{R}^{n_i}$ represents the gradient of the i-th element function. When the k-th iterate must be specified, the subscript k is added as $\widehat{g}_{k,i}$ is $\widehat{g}_i$. In other cases, it refers to the application of U_i onto a vector, for example $\widehat{x}_i := U_i x \in \mathbb{R}^{n_i}$, where $x \in \mathbb{R}^n$.

Here is a list of all the other capped notations:

- $\widehat{s}_i := U_i s,\ s \in \mathbb{R}^n$;
- $\widehat{y}_i := \nabla \widehat{f}_i(\widehat{x}_i + \widehat{s}_i) - \nabla \widehat{f}_i(\widehat{x}_i) \in \mathbb{R}^{n_i}$;
- $\widehat{B}_i \approx \nabla^2 \widehat{f}_i \in \mathbb{R}^{n_i \times n_i}$;
- $\widehat{z}_i := \widehat{y}_i - \widehat{B}_i \widehat{s}_i$;

The j-th quasi-Newton update retained by a limited quasi-Newton operator B_k is denoted as $B_k^{(j)}$ where $1 \leq j \leq m$. Its initializer is denoted $B_k^{(0)} = \lambda I$, where λ may be 1 or $\frac{y_k^\top y_k}{y_k^\top s_k}$.

3 Literature review

The first solver implementation optimizing continuous partially-separable problems was the PSPMIN Fortran routine [25], which implements the partitioned quasi-Newton trust-region method from [22–24] and is referenced as VE08 in the HSL library [27]. PSPMIN solves (2) subject to bound constraints using a projected gradient method [2] and a truncated conjugate gradient method [45] to solve the trust-region sub-problem. It requires subroutines to compute every $\widehat{f}_i$ and $\nabla \widehat{f}_i$. If $\nabla \widehat{f}_i$ is not provided, it relies on finite differences to approximate it. This routine was tested on the first collection of partially-separable problems [48] and led the way for other partially-separable methods.

There is a variant of VE08 named VE10 [27], dedicated to large-scale least-square problems that are partially-separable $f(x) = \frac{1}{2}\sum_{i=1}^N f_i(x)^2 = \frac{1}{2}\sum_{i=1}^N \widehat{f}_i(\widehat{x}_i)^2$ [49], considering $\widehat{f}_i : \mathbb{R}^{n_i} \to \mathbb{R}$ with $n_i < n$. VE10 maintains simultaneously a Gauss-Newton model and a partitioned quasi-Newton model made of dense element Hessian approximation $\widehat{B}_i \approx \nabla^2 \widehat{f}_i$. The choice of which model use is done adaptively to ensure a better performance of the method.

The fact that providing routines to evaluate $\widehat{f}_i$ and $\nabla \widehat{f}_i$ is error-prone and poorly extendable motivated the definition of the Standard Input Format (SIF) to ease large-scale problem definitions. The SIF is based around the concept of *group partial-separability* [7]:

$$f(x) = \sum_{j=1}^{M} h_j(l_j(x) + f_j(x)), \quad f_j(x) = \sum_{i=1}^{N_j} \widehat{f}_{j,i}(U_{j,i}x),$$

where $h_j : \mathbb{R} \to \mathbb{R}$ is the j-th group, l_j the linear term of the j-th group and $\widehat{f}_j$ its partially-separable function. Group-partial separability extends partial separability, which comes handy to handle partially-separable constraints $c(x) = 0$ by minimizing a sequence of augmented Lagrangian problems $L_{y,\rho}(x) := f(x) - y^\top c(x) + \frac{1}{2}\rho\|c(x)\|^2$. The SIF inherited the column-oriented format of MPS (Mathematical Programming System) [1], making it quite obscure and difficult to access for a new user. However, the Constrained and Unconstrained Testing Environnement (CUTE) [4] and its more recent versions CUTEr [18] and CUTEst [20] provide 1494 SIF problems that can be interfaced to modern solvers, included: IPOPT [50], KNITRO [5] and SNOPT [17] and modern high-level languages, such as Python and Julia. The SIF *decoder* [18] translates a SIF model into a set of Fortran routines evaluating $\widehat{f}_i, \nabla \widehat{f}_i, U_i, U_i^\top, h_j$ and l_j, to feed PSPMIN.

LANCELOT [7], later integrated in the library GALAHAD [19], is the most sophisticated solver developed around group-partial separability and the SIF format. LANCELOT has several special features such as:

- support internal element variables, considering U_i as a sparse matrix;
- a dedicated methods to compute partial derivatives of group-partially-separable functions;

- a procedure merging elements to reduce memory and computations during partitioned the matrix-vector products [7];
- a multi-frontal direct method dedicated to group-partially-separable problems [11, 12] as an alternative of the truncated conjugate gradient;
- structured trust-region methods [8], allowing a finer management of the step considering a specified trust-region radius $\Delta_{k,i}$ for each element i taking into account the non-linearity degree of each $\widehat{f}_i$.

Gay [16], through the commercial modeling language AMPL [14], is the first to describe how to detect partial separability from any expression. Partial separability is detected by accumulating linear sub-expressions in the directed acyclic graph issued from the user model. Then, a dedicated automatic differentiation for group partially-separable problem is applied to provide any $\nabla\widehat{f}_i$ mandatory to run LANCELOT later.

4 Optimization background

Trust-region methods

A trust-region method is an iterative method solving at each step an approximate problem restrained to a neighbourhood. At a typical k-th iteration of a quadratic trust-region method minimizing (1), we construct a quadratic model

$$m_k(s) := f(x_k) + \nabla f(x_k)^\top s + \tfrac{1}{2} s^\top B_k s, \tag{4}$$

of f about the current iterate x_k, where $B_k = B_k^\top$ is an approximation of the Hessian $\nabla^2 f(x_k)$. A step s is subsequently computed as an approximate solution of the trust-region sub-problem

$$\min_s m_k(s) \quad \text{s.t.} \|s\| \leq \Delta_k, \tag{5}$$

where $\Delta_k > 0$ is the current trust-region radius. A step only needs to produce sufficient decrease in the sense that

$$m_k(0) - m_k(s) \geq \tau \, \|\nabla f(x_k)\| \, \min\left(\frac{\|\nabla f(x_k)\|}{1 + \|B_k\|}, \Delta_k\right), \tag{6}$$

where $0 < \tau < 1$ is a constant. The decrease of the model is compared to that of f, to decide whether the step is accepted or rejected, and to finally update the trust-region radius. The scheme of a typical trust-region method is summarized in Algorithm 4.1, modelled after [9, Algorithm 6.1.1]. We enforce a maximal trust-region radius Δ_{max}, which doesn't have incidence on practical performance, to bound $\|B_k\|$ in Section 6.1. If $B_k = \nabla^2 f(x_k)$, Algorithm 4.1 becomes a Newton trust-region method.

The performance of a trust-region method falls on the efficiency of the procedure applied to find s_k satisfying (6). We chose the truncated conjugate gradient method [45], an iterative method deriving from the conjugate gradient [26] adapted to a trust-region sub-problem. It solves a linear system through successive operator-vector products $u = B_k v$, $v \in \mathbb{R}^n$. In exact arithmetic, the conjugate gradient performs at most n iterations. Moreover, the solution's norm increases at each iterate [26]. Thus, once the solution steps out of the trust-region's boundary, the method stops and finds a solution on the boundary. In practice, the solution of the linear system is generally not in the trust-region at every iteration. Thus, it performs frequently fewer iterations than n and doesn't necessarily solve the linear system.

Quasi-Newton methods

Quasi-Newton methods seek to approximate $B_k \approx \nabla^2 f(x_k)$. The special case of secant methods update B_k to satisfy the secant equation

$$B_k s = y_k, \quad y_k := \nabla f(x_{k+1}) - \nabla f(x_k).$$

Algorithm 4.1 Trust-Region Algorithm

```
1: Choose $x_0 \in \mathbb{R}^n$, $\Delta_{max} \geq \Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$, and $0 < \omega < 1$.
2: Choose $B_0 = B_0^\top \approx \nabla^2 f(x_0)$.                              Initial approximation
3: for $k = 0, 1, \ldots$ do
4:     Compute an approximate solution $s_k$ of (5) satisfying (6).
5:     Compute the ratio
           $$\rho_k := \frac{f(x_k) - f(x_k + s_k)}{m_k(0) - m_k(s_k)}.$$
6:     if $\rho_k \geq \eta_1$ then                                                  successful step
7:         set $x_{k+1} = x_k + s_k$, $y_k := \nabla f(x_{k+1}) - \nabla f(x_k)$
8:         update $B_k$ given $s_k$ and $y_k$
9:     else                                                                         unsuccessful step
10:        set $x_{k+1} = x_k$ and $B_{k+1} = B_k$.
11:    end if
12:    Update the trust-region radius according to
           $$\Delta_{k+1} \in \begin{cases} [\gamma_3 \Delta_k,\ \gamma_4 \Delta_k & \text{if } \rho_k \geq \eta_2, \\ [\gamma_2 \Delta_k,\ \Delta_k] & \text{if } \eta_1 \leq \rho_k < \eta_2, \\ [\gamma_1 \Delta_k,\ \gamma_2 \Delta_k] & \text{if } \rho_k < \eta_1. \end{cases} \quad (7)$$
13: end for
```

Among secant quasi-Newton updates, two dominate. The BFGS (Broyden Fletcher Goldfarb Shanno) update

$$B_{k+1}^{\text{BFGS}} = B_k + \frac{y_k y_k^\top}{y_k^\top s_k} - \frac{B_k s_k s_k^\top B_k^\top}{s_k^\top B_k s_k}, \tag{8}$$

stays positive definite as long as $B_0 \succ 0$ and the curvature condition $y_k^\top s_k > 0$ holds. In practice, we perform the update only if $y_k^\top s_k > \epsilon_1 > 0$, for numerical stability. If it does not, the positive definiteness may be preserved either by skipping the update or by using damping [31]. By staying positive definite, it offers at each iterate a convex quadratic approximation well suited for convex problems.

The symmetric rank one (SR1) [10] update

$$B_{k+1}^{\text{SR1}} = B_k + \frac{z_k z_k^\top}{s_k^\top z_k}, \quad z_k := y_k - B_k s_k, \tag{9}$$

does not ensure positive definiteness. To avoid numerical instability, it is customary to only perform the SR1 update provided that

$$|s_k^\top z_k| \geq \epsilon_2 \, \|s_k\| \|z_k\|, \quad \epsilon_2 > 0.$$

Because both BFGS and SR1 rely on B_k, a dense matrix $\Theta(\frac{n(n+1)}{2})$, those methods are not applicable to intermediate or large problems. A more realistic alternative consists in using a limited-memory approximation [6, 30, 32], which allows users to set storage requirements ahead of time, and for which operator-vector products can be computed efficiently without forming B_k explicitly. The limited-memory variants of BFGS and SR1 will be referred as LBFGS and LSR1. Their memory and linear application complexities, based the m last pairs s_k, y_k, are $\Theta(mn)$. In the next sections, the j-th quasi-Newton update retained by a limited quasi-Newton operator B_k is denoted as $B_k^{(j)}$ where $1 \leq j \leq m$. Its initializer is denoted $B_k^{(0)} = \lambda I$, where λ may be 1 or $\frac{y_k^\top y_k}{y_k^\top s_k}$.

Partitioned quasi-Newton updates

As an accumulation of low rank updates, traditional quasi-Newton methods cannot replicate the sparsity structure of the Hessian. There have been attempts at sparse quasi-Newton updates, but Shanno [44] and Toint [47] referred unsatisfying results. Later on, Griewank and Toint [23] showed that any function having a sparse Hessian is partially-separable and that partitioned quasi-Newton update structurally keeps the Hessian sparsity.

Partitioned quasi-Newton updates exploit the partitioned structure of the derivatives of (2):

$$\nabla f(x) = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i(\widehat{x}_i) \quad \text{and} \quad \nabla^2 f(x) = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_i) U_i,$$

to accumulate the contribution of each element derivatives. Hence, it structurally preserves the sparsity structure of the Hessian, see the example (15). Based on the assumption that each $n_i \ll n$, partitioned quasi-Newton methods maintain a quasi-Newton approximation $\widehat{B}_{k,i} \approx \nabla^2 \widehat{f}_i(\widehat{x}_{k,i})$ for each element and aggregate them to approximate the Hessian

$$B_k := \sum_{i=1}^{N} U_i^\top \widehat{B}_{k,i} U_i. \tag{10}$$

Each $\widehat{B}_{k,i}$ update is based on $\widehat{y}_{k,i} := \nabla \widehat{f}_i(\widehat{x}_{k+1,i}) - \nabla \widehat{f}_i(\widehat{x}_{k,i})$ and $\widehat{s}_i := \widehat{s}_i$ instead of y and s. Griewank and Toint [22, 23, 24] present initially two partitioned methods: one approximating every $\widehat{B}_i$ by BFGS (PBFGS) while the second uses SR1 (PSR1). By updating potentially every element at each iterate, the update's rank may be proportional to $\min(N, n)$. In few iterations, several element function local curvatures are captured by $\widehat{B}_{k,i}$, making B_k a sharper Hessian approximation than BFGS.

Thanks to (10), the truncated conjugate gradient may compute efficiently $B_k v$ without ever assembling B_k:

$$B_k v = \sum_{i=1}^{N} U_i^\top (\widehat{B}_{k,i} U_i v).$$

Only dot products with each $\widehat{B}_{k,i}$ and $U_i v$ are required. Afterward, $U_i^\top$ scatters the results of $\widehat{B}_{k,i} U_i v$ into the appropriate components of $B_k v$.

Although PBFGS and PSR1 satisfy the secant equation as long as every element is updated

$$Bs = \sum_{i=1}^{N} U_i^\top \widehat{B}_i \widehat{s}_i = \sum_{i=1}^{N} U_i^\top \widehat{y}_i = y,$$

there is no guarantee that $\widehat{y}_i$ and $\widehat{s}_i$ always satisfy the numerical safeguards specific to quasi-Newton updates, since s is related to minimization of f and not of $\widehat{f}_i$.

Griewank and Toint [25] proposes a new partitioned quasi-Newton enhancing the chance of $\widehat{B}_i$ to be updated. It updates $\widehat{B}_i$ with BFGS as long as the element curvature condition holds and switch to SR1 the first time it fails. Toint [49] allows $\widehat{B}_i$ to be updated with BFGS even after a fail of the element curvature condition. Thus, at each iterate, the update's choice between BFGS and SR1 determined by the satisfaction element curvature condition. Even if the positive definiteness is lost, every $\widehat{B}_{k,i}$ best approximate its local landscape, with the hope of making B_k closer to the current Hessian. The combination of both quasi-Newton update will make B_k generally best satisfy the secant equation, by having more elements updated. To summarize, at the k-th iterate, each element update follows:

$$\widehat{B}^{\text{SE}}_{k+1,i} = \begin{cases} \widehat{B}^{\text{BFGS}}_{k+1,i} & \text{if } \widehat{s}_i^\top \widehat{y}_i > \epsilon_1 \\ \widehat{B}^{\text{SR1}}_{k+1,i} & \text{if } \widehat{s}_i^\top \widehat{y}_i \le \epsilon_1 \text{ and } |\widehat{s}_i^\top (\widehat{y}_i - \widehat{B}_k \widehat{s}_i)| \ge \epsilon_2 \|\widehat{s}_i\| . \|\widehat{y}_i - \widehat{B}_k \widehat{s}_i\| \\ \widehat{B}_{k,i} & \text{otherwise} \end{cases},$$

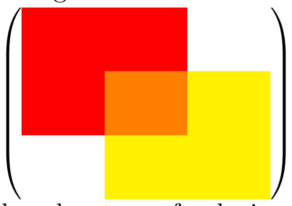
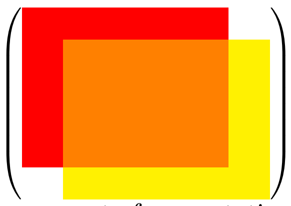
where SE stands for secant equation.

Another variant exists, similar to one of the proposition of [25], consisting on apply BFGS updates onto $\widehat{B}_i$ if $\widehat{f}_i$ is convex or SR1 updates otherwise. This variant, named PCS, will be used Section 7 to compare partitioned quasi-Newton methods, after an automatic detection of the convexity.

Partitioned quasi-Newton methods store a dense matrix for each element i, making the overall memory cost of the partitioned matrix:

$$\Theta\left(\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2}\right).$$

Note that storing a partitioned matrix may require more memory than a sparse matrix, especially when there are many overlaps between elements. In cases where some n_i are large, we fall back on the dense matrix storage issue similar to BFGS or SR1. Moreover, as each n_i becomes larger, elements are more likely to overlap. Despite the fact that the partitioned matrix-vector product may be distributed over several processors, each overlap increases the amount of computation required for $B_k v$. Conn et al. [7] presents a recursive heuristic merging two elements i, j if their shared variables, which overlaps on $\widehat{B}_i$ and $\widehat{B}_j$, induce more storage than the gain of sparsity from storing $\widehat{B}_i$ and $\widehat{B}_j$ separately. The two examples below represent the Hessian's structure related to $f^{(1)} = \widehat{f}_1(x_1, x_2, x_3, x_4) + \widehat{f}_2(x_3, x_4, x_5, x_6)$ and $f^{(2)} = \widehat{f}_1(x_1, x_2, x_3, x_4, x_5) + \widehat{f}_2(x_2, x_3, x_4, x_5, x_6)$. $f^{(1)}$ and $f^{(2)}$ have two element functions overlapping respectively on 2 and 4 variables. In the first example, the gain from sparsity (in white) overcomes the overlap (in orange), while in the second example the overlap makes maintaining two dense element Hessian approximations costlier than a matrix of size $n = 5$.

$B_k^{(1)} =$  , $B_k^{(2)} =$  .

Merging has the advantage of reducing the memory cost of (10) and the amount of computation during a partitioned matrix-vector product. However, as the number of mergers tends to N, B_k will tend to a BFGS or SR1 approximation, and accordingly, to an update of smaller rank. Ultimately, it deteriorates the performances of the algorithm [49].

5 Four Julia modules to detect and exploit partial separability

JuliaSmoothOptimizers (JSO) ecosystem architecture

Our contribution is implemented in Julia [3], a high-level language deigned for scientific computation. It naturally supports multi-precision and interface-oriented architectures. Contrary to other popular high-level programming languages (python, matlab...), pure Julia code get performances comparable to C/C++ [33].

JSO's ecosystem gathers tools developed conjointly to democratize smooth non-linear optimization methods. For example, solving:

$$\min_{x \in \mathbb{R}^n} f(x). \tag{11}$$

The current section expresses how the knowledge of f being partially-separable integrates JSO's models and the multi-precision solvers made of it. Figure 8 in Appendix A.2 summarizes graphically the dependencies graph of the different modules implemented.

One of the JSO's pillars is NLPModels.jl [37], an interface for optimization models on which solver implementations are based. Any model satisfying the NLPModel's interface implements several methods, in our context, mainly: the objective function, the gradient, the Hessian-vector product or the linear application B_k approximating the Hessian. The simplest model has its objective function defined by pure Julia's code, while its derivatives are computed by automatic differentiation [21], and will be referred as *pure Julia's model*.

The interface provided by NLPModels.jl allows the implementation of generic optimization solvers, gathered in JSOSolvers.jl [38]. Thus, the same code may run any model deriving from NLPModels while exploiting their distinct features. For this paper, we focus on TRUNK, a trust-region solver close to Algorithm 4.1, but the tools developed in this section aim to be applied on other solvers. By default, minimizing a pure Julia's model with TRUNK will use the exact Hessian and will implement a Newton trust-region solver. Another option for a pure Julia's model is to integrate a LBFGS or a LSR1 operator [36, 39] (see Section 4), which will overwrite Hessian's operations by using instead the quasi-Newton operator. As a result, it becomes a limited-memory quasi-Newton model, and TRUNK becomes a limited-memory quasi-Newton trust-region solver.

Our contribution augments a pure Julia's model to a partitioned quasi-Newton model by detecting automatically its partially-separable structure. Instead of providing a limited-memory quasi-Newton operator, the user may choose between several partitioned quasi-Newton operators: PBFGS, PSR1, PSE, PCS (see Section 4) or new limited-memory partitioned quasi-Newton operators: PLBFGS, PLSR1, PLSE (see Section 6). These partitioned models not only overwrite Hessian's operations, they also overwrite any method which may benefit partial separability. Another variant uses exact derivatives $B_k = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{f}_i(\widehat{x}_{k,i}) U_i$, computing only element functions derivatives, to reduce the computational cost of accessing $\nabla^2 f$ (see Section 4). Thus, TRUNK provides as many partitioned quasi-Newton solvers as there are partitioned quasi-Newton models.

To define efficient solvers, JSO's solvers allocate beforehand all data-structures. Thus, it requires the type of the data-structure deriving from an AbstractVector storing x_k, assuming that is the same as the one required for storing g_k, s_k and y_k as defined in Section 4 or in Algorithm 4.1. While such an implementation being somehow flexible, it stretches when it comes to consider at the same time: x_k the current point (usually a Vector), g_k the partitioned gradient and y_k the partitioned gradient difference, mandatory for partitioned quasi-Newton updates (see Section 4). The same data-structure deriving from AbstractVector must also behave properly during the truncated conjugate gradient implementation issued of the module Krylov.jl [34], which has its own solver's structure. Moreover, to follow the interface, the linear application of a partitioned quasi-Newton operator B_k is also parametrized by a PartitionedVector (and not a usual Vector). These constraints pushed us to create the type PartitionedVector deriving from AbstractVector. It behaves as a Vector for x_k and s_k and keeps element contributions for g_k and y_k. PartitionedVectors must support all the operations required by TRUNK, the truncated conjugate gradient and must fit the NLPModels's interface usually based around Vectors. Concretely, any partially-separable model must build automatically a PartitionedVectors and a partitioned quasi-Newton operator to run TRUNK and the truncated conjugate gradient based.

The rest of the section presents the details of the technical contribution. PartiallySeparableNLPModels.jl [28] defines all partially-separable models, with the participation of three other modules from JSO:

- ExpressionTreeForge.jl [41], detailed in Section 5.1, defines automatically the partially-separable structure from a standard NLPModel. It supports pure Julia's models, JuMP models [13], the most popular julia's modeling language, or the native Expr Julia type;
- PartitionedStructures.jl [42], detailed in Section 5.2, allocates the partitioned structures related to partitioned quasi-Newton approximations or partitioned derivatives;
- PartitionedVectors.jl [43], detailed in Section 5.3, implements PartitionedVector and the related methods mandatory for TRUNK and the truncated conjugate gradient.

A graph of JSO's modules connections is defined Appendix A.2 in Figure 8. PartiallySeparableNLPModels.jl manages the exploitation of partial separability to specify the computation of derivatives, and fits the NLPModel's interface (more details in Section 5.4). As a user, running TRUNK on any partitioned quasi-Newton model performs a partitioned quasi-Newton trust-region method without any supplemental needs.

5.1 ExpressionTreeForge.jl

ExpressionTreeForge.jl is a toolbox about expression trees, representing in our case the objective function. The module's design relies on a tree interface, which matches the JuMP interface. For any expression tree, a leaf is either a variable or a constant node. Intermediate nodes (including the root) are all operator nodes. ExpressionTreeForge has several features (illustrated Figure 1 and Figure 2):

- automatically detect the partially-separable structure, i.e. retrieve every $\widehat{f}_i$ and U_i. First, you walk the tree from the root to recursively remove additive operators. If f is partially-separable, then it divides the tree in subtrees, each of them being an element function $f_i : \mathbb{R}^n \to \mathbb{R}$. Next, it retrieves U_i and consequently n_i to define $\widehat{f}_i(\widehat{x}_i) = f_i(x)$. We chose to represent U_i as the subset of variables appearing in f_i. U_i as a linear operator whose rows are vectors from the Euclidean basis. Thus, U_i is represented by a vector of integers, each of which gives the index of a variable appearing in f_i. Finally, the variable's indices of the sub-expression tree f_i must be changed according to U_i to form $\widehat{f}_i$;
- infer the bounds and the convexity status for $\widehat{f}_i$ and its intermediate nodes. Leafs are initially bounded to $[-\infty, +\infty]$, considering an unconstrained optimization perspective. Then, it propagates bounds to intermediate nodes, up to the root, given the operator each node represents and its children bounds. Then, from the bounds and the convexity statuses of the leafs, initially set to *linear* or *constant*, a convexity status is propagated to any intermediate nodes by following the operator based rules defined by Fourer et al. [15]. For example, the exponential function $f(x) = e^x$ is convex increasing, making $f \circ g$ convex if g is convex [15].

For example, suppose

$$f(x) = \frac{(x_1 x_3)^4}{x_2^2 + 1} + \frac{(x_3 x_5)^4}{x_4^2 + 1} + \exp((x_1 + x_3 + x_5)^2), \tag{12}$$

then walks of the tree detailed previously for such f give Figure 1 and Figure 2.

The result of the tree walks are:

$$\begin{gathered}\widehat{f}_1(y_1, y_2, y_3) = \widehat{f}_2(y_1, y_2, y_3) = \tfrac{(y_1 y_3)^4}{y_2^2+1}, \ \widehat{f}_3(y_1, y_2, y_3) = \exp((y_1 + y_2 + y_3)^2), \\ U_1 = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}, U_2 = \begin{bmatrix} e_3 \\ e_4 \\ e_5 \end{bmatrix}, U_3 = \begin{bmatrix} e_1 \\ e_3 \\ e_5 \end{bmatrix}. \end{gathered} , \tag{13}$$

It recognizes two identic element function $\widehat{f}_1$ and $\widehat{f}_2$ differentiated only by respectively U_1 and U_2. ExpressionTreeForge provides every $\widehat{f}_i$ and U_i for the other modules to allocate the suitable partitioned data structures and then, to define on the partially-separable model.

5.2 PartitionedStructures.jl

PartitionedStructures.jl defines partitioned vectors or partitioned matrices, each of which stores element vector contributions or element matrix contributions separately. These partitioned structures are motivated by partially-separable function derivatives:

$$\nabla f = \sum_{i=1}^{N} U_i^\top \nabla \widehat{f}_i, \quad \nabla^2 f \approx B = \sum_{i=1}^{N} U_i^\top \nabla^2 \widehat{B}_i U_i, \tag{14}$$

where each element contribution is independent. In particular, any partitioned quasi-Newton method requires every the N pairs of $\widehat{y}_{k,i} := \nabla \widehat{f}_i(\widehat{x}_{k+1,i}) - \nabla \widehat{f}_i(\widehat{x}_{k,i})$ and $\widehat{s} := U_i$. Graphically, you can visualize

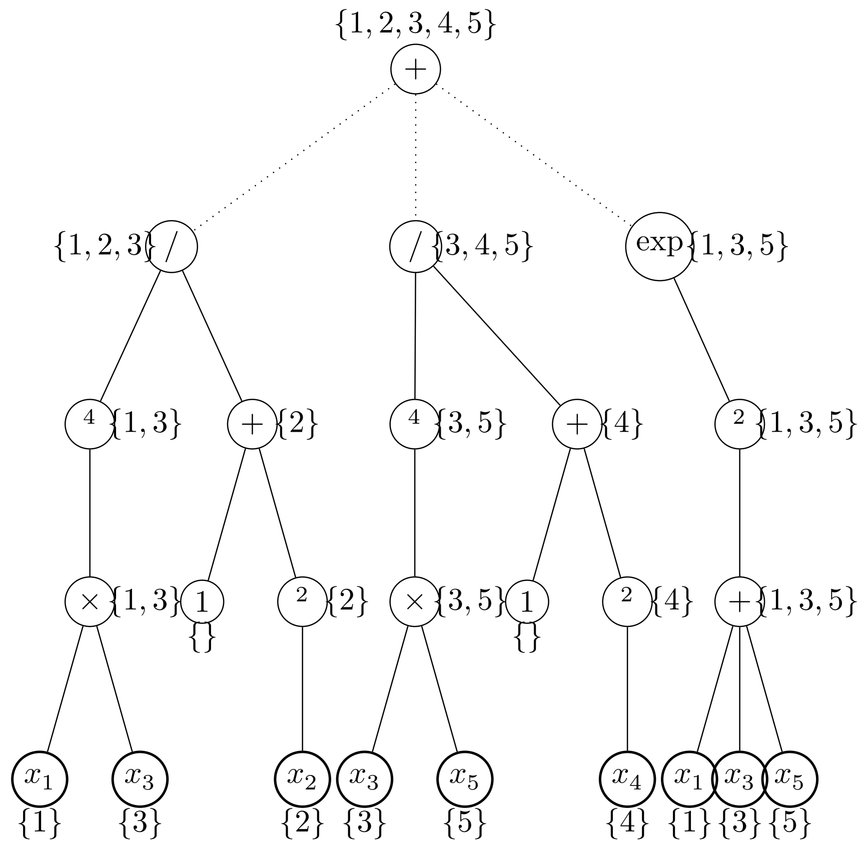


Figure 1: Automatic partial separability detection

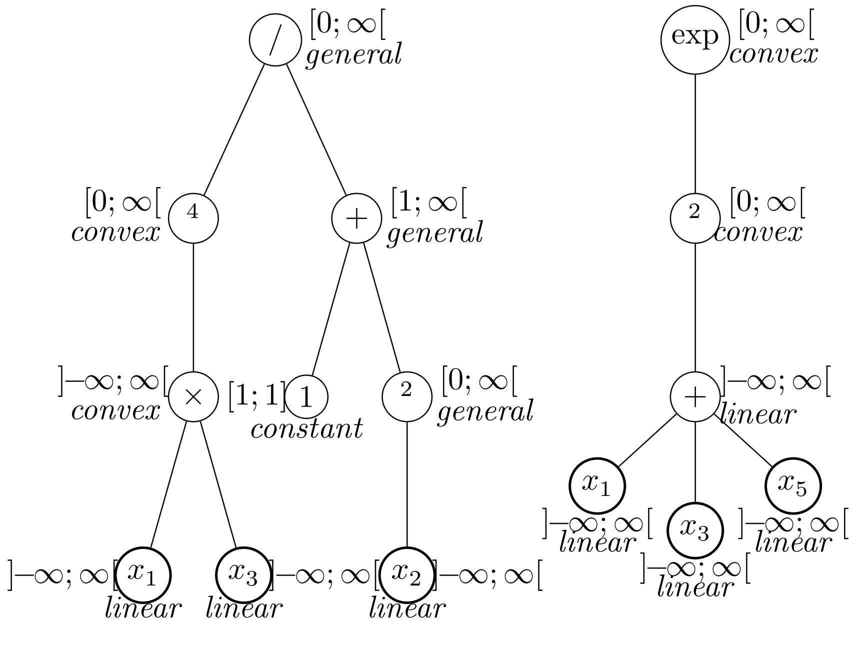
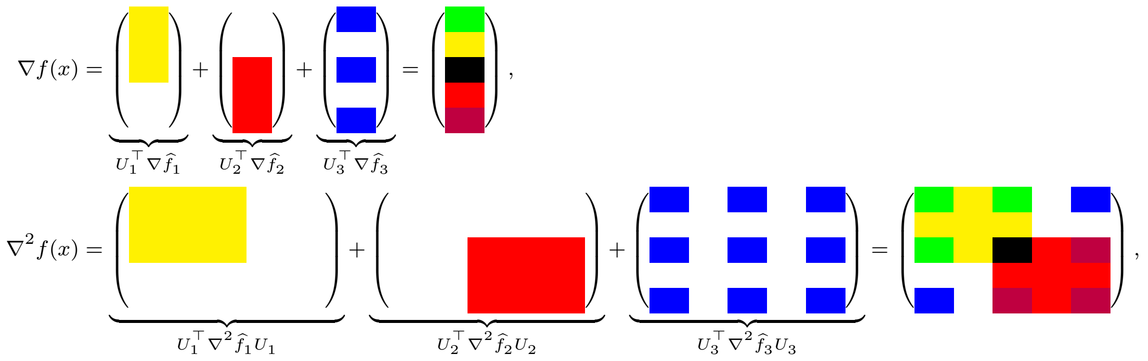


Figure 2: Bounds and convexity status of element functions

derivatives from (12) as:


$\nabla f(x) =$
$U_1^\top \nabla \widehat{f}_1$
$U_2^\top \nabla \widehat{f}_2$
$U_3^\top \nabla \widehat{f}_3$
$\nabla^2 f(x) =$
$U_1^\top \nabla^2 \widehat{f}_1 U_1$
$U_2^\top \nabla^2 \widehat{f}_2 U_2$
$U_3^\top \nabla^2 \widehat{f}_3 U_3$

(15)

where yellow, red and blue represent respectively contributions from $\widehat{f}_1$, $\widehat{f}_2$ and $\widehat{f}_3$. Other colors express the contribution of several elements for a single partial derivative. Partitioned vectors may also represent the linear application $U_i v$ for all elements $1 \leq i \leq N$, or the result of a partitioned matrix-vector product (more details in Section 5.3). Contrary to the memory footprint of a Vector or a symmetric Matrix, respectively in $\Theta(n)$ and $\Theta(\frac{n(n+1)}{2})$, the memory footprint of partitioned structures is not related to n. Storing a partitioned vector is $\Theta\left(\sum_{i=1}^N n_i\right)$ while storing a symmetric partitioned matrix is $\Theta\left(\sum_{i=1}^N \frac{n_i(n_i+1)}{2}\right)$. PartitionedStructures.jl implements subroutines to perform: basic operations for partitioned vectors, partitioned matrix-vector product and partitioned quasi-Newton updates. All partitioned quasi-Newton updates have a homogeneous interface, relying on a partitioned matrix and a pair of partitioned vectors representing every $\widehat{s}_i$ and every $\widehat{y}_i$.

A partitioned object is composed of an ordered set of element objects and an index table informing for any x_j, $1 \leq j \leq n$ which element function $\widehat{f}_i$ impacts x_j. Moreover, a partitioned vector possesses a Vector to build in-place the associated Vector while a partitioned quasi-Newton operators may aggregate element contributions to form either a SparseMatrix or a Matrix (10), mainly for test purposes. The main distinction between partitioned objects is the nature of its element objects, either a Vector, a symmetric Matrix or a quasi-Newton LinearOperator (see Section 6). Any partitioned object keeps the list of the indices related to itself, equivalent to every U_i (as described in (13)).

Among the several ways to instantiate partitioned structures, the simplest one consists in using a nested integer Vector, where each component informs the indices of variables used by each element (i.e. the U_i). The next example refers to the euclidean basis vectors isolated for $U_1 : \mathbb{R}^5 \to \mathbb{R}^3$, $U_2 : \mathbb{R}^5 \to \mathbb{R}^3$ and $U_3 : \mathbb{R}^5 \to \mathbb{R}^3$ from (13):

$$U = [[1,2,3],[3,4,5],[1,3,5]].$$

Thus given the results of Section 5.1 and Figure 1, any partitioned object may be created.

5.3 PartitionedVectors.jl

PartitionedVectors.jl implements the partitioned data structure deriving from AbstractVector. It matches as much as possible the Julia's AbstractVector interface to satisfy the requirement set by TRUNK or the truncated conjugate gradient, in particular about in-place broadcasted operations. However, to implement a partitioned quasi-Newton method, a PartitionedVector must have two distinct usages:

- usage 1: store the distinct element vector values. By doing so, the associated vector is built from element vectors aggregation, example: $\nabla f = \sum_{i=1}^N U_i^\top \nabla \widehat{f}_i$;
- usage 2: each element vector represents the linear application $\widehat{v}_i := U_i v$, $v \in \mathbb{R}^n$ for all element i. In practice, it stores simultaneously every $\widehat{x}_i$ or $\widehat{s}_i$. Considering (13) and $x = (1, 2, \sqrt{2}, \pi, 5)$, then you get the element vectors $\widehat{x}_1 = U_1 x = (1, 2, \sqrt{2})$, $\widehat{x}_2 = U_2 x = (\sqrt{2}, \pi, 5)$ and $\widehat{x}_3 = U_3 x = (1, \sqrt{2}, 5)$.

Contrary to usage 1, the associated vector doesn't aggregate element contributions. Instead, if required, it forms v from all $\widehat{v}_i$, by getting the value of $v_j \in \mathbb{R}$ from the elements where $U_i e_j \neq 0$.

The first usage is mandatory to perform partitioned quasi-Newton updates, y being computed from ∇f, as well as involved in the partitioned matrix-vector product. The second usage is used to track the current point x and the step s resulting from the truncated conjugate gradient. For both usages, it requires in-place broadcasted methods (for TRUNK) and in-place methods related to the conjugate gradient method, mainly $mul!$.

For any PartitionedVector pv, we chose that $pv[i],\ 1 \leq i \leq N$ returns the i-th element vector. Consequently, $pv[i] = v$ will set the i-th element vector to v, v being an element vector or a Vector of size n_i. All basic operations for element vectors are implemented in PartitionedStructures.jl, ex: $pv[1] + pv[1] == 2 * pv[1]$.

The broadcast mechanism gathers all broadcasted operations of a single line $x. = y. + \alpha. * z$ into a single function $f(x, y, \alpha, z)$ representing $x = y + \alpha z$ applied onto each index of the AbstractVector. Contrary to Vectors which distributes it to every component, PartitionedVectors distribute it in-place along element vectors. The requirements are that every PartitionedVector must have the same partitioned structure, e.g. the same U_i, and the same usage. Broadcast may be applied onto PartitionedVectors with different usages, but the meaning of such an operation remains unclear ; the result is not of usage 2 anymore, e.g. $\widehat{v}_i$ would generally not correspond to $U_i v$, and $\sum_{i=1}^N U_i^\top \widehat{v}_i$ aggregates element contributions corresponding to nothing. You can also run $pv\ . = 1$, it sets every element vector to $(1, 1, ..., 1) \in \mathbb{R}^{n_i}$. It makes sense for a PartitionedVector of usage 2, being the result of $U_i v$ forall i, where $v = (1, 1, ..., 1) \in \mathbb{R}^n$. For PartitionedVectors of usage 1, there is no meaning to $. = \alpha$ or $. + \alpha,\ \alpha \in \mathbb{R}^*$, except $pv. = 0$ which may initiate a PartitionedVector after $pv = similar()$. PartitionedVectors of both usages benefit also of scalar product and norm methods, mandatory for the trust-region method (see Algorithm 4.1). The scalar product may benefit the partitioned structure to perform subroutine on elements if both PartitionedVectors have different usages. Suppose w a Vector represented by a PartitionedVector (storing $U_i w,\ \forall\ 1 \leq i \leq N$) and $v = \sum_{i=1}^N U_i^\top \widehat{v}_i$:

$$v^\top w = (\sum_{i=1}^N U_i^\top \widehat{v}_i)^\top w = \sum_{i=1}^N \widehat{v}_i^\top U_i w = \sum_{i=1}^N \widehat{v}_i^\top \widehat{w}_i,$$

which accumulates element scalar products. The norm, on the other hand, may be computed directly from the Vector representing a PartitionedVector, depending on its usage. If it is of usage 1, then g is formed by accumulating $\sum_{i=1}^N U_i^\top \widehat{g}_i$ and then $\|g\|_2$ is computed directly from g. For the second usage, which represents every $U_i v, 1 \leq i \leq N$, v is retrieved and its norm computed.

The next critical step is how the conjugate gradient method exploits the partitioned matrix-vector product with PartitionedVectors:

$$Bv = \sum_{i=1}^N U_i^\top \widehat{B}_i U_i v. \tag{16}$$

Computing Bv requires first $\widehat{v}_i = U_i v$, which is known if v is a PartitionedVector of usage 2. Then, the linear application from $\widehat{B}_i$ is applied from $\widehat{v}_i$ to get $\widehat{Bv}_i = \widehat{B}_i \widehat{v}_i$. Finally, the results are aggregated into $Bv = \sum_{i=1}^N U_i^\top \widehat{Bv}_i$, making Bv a partitioned vector of usage 1. However, the solution of the trust-region sub-problem s_k is of usage 2, while being determined by Bv of usage 1. To transfer Bv to s safely, we overload the $axpy!$ method used in the Krylov.jl, equivalent to $y\ .+ = \alpha. * x, \alpha \in \mathbb{R}$, when usages are different. It builds the associated Vectors of both Bv and s to then apply $axpy!$ on them and store the result in s. At the end, every element have performed $\widehat{s}_i.+ = \alpha. * U_i Bv$. Warning, $axpy!$ and $axpby!$, $y. = x. * \alpha. + y. * \beta$, perform more computations for PartitionedVectors of usage 1 than Vectors. As $n \leq \sum_{i=1}^N n_i$, the dispatch of $axpy!$ or $axpby!$ over every element vector is costlier than on a Vector.

5.4 PartiallySeparableNLPModels.jl

After determining the partially-separable structure with ExpressionTreeForge.jl and defining dedicated partitioned structures with PartitionedStructures.jl and PartitionedVectors.jl; PartiallySeparableNLPModels.jl coordinates everything to build partially-separable optimization models. After retrieving the element functions $\widehat{f}_i$ and theirs associated U_i using ExpressionTreeForge (Section 5.1), a number of adjustments can be made in order to reduce the memory footprint of the structure. For example, large models may duplicate element functions applied on different variables, e.g. $\widehat{f}_i(\widehat{x}_i) = \widehat{f}_j(\widehat{x}_i)$ with $i \neq j$, for example $\widehat{f}_1$ and $\widehat{f}_2$ in (13). By factorizing identical element expression trees, only $M \leq N$ distinct element functions remains, as well as a matching between the U_i and these M element functions. Such identification may be realized after splitting f into $\widehat{f}_i, 1 \leq i \leq N$. It becomes precious when it comes to compute the partitioned derivatives (14) with automatic differentiation in reverse mode [21]. Instead of potentially duplicating N tapes, required for the backward pass, only $M(\leq N)$ tapes are needed to compute any $\widehat{f}_i(\widehat{x}_{k,i})$ or $\nabla^2 \widehat{f}_i(\widehat{x}_{k,i}) U_i v$. The amount of memory saved depends on the gap between N and M. The partially-separable structure may also be ignored, if an estimation of the memory required by the partitioned structure becomes unhealthy for the partitioned method performances. By doing so, it return to an unstructured problem $\widehat{f}_1 = f$ where $U_1 = I \in \mathbb{R}^{n \times n}$.

After all the steps above, the PartitionedVectors and the partitioned quasi-Newton operators, related to the U_i are allocated independently and PartiallySeparableNLPModes.jl articulates them around these M distinct element functions. Several models may be defined, each of them having a different Hessian approximation procedure:

- PBFGSNLPModel, PSR1NLPModel, PCSNLPModel, PSENLPModel using a partitioned quasi-Newton operator B_k as described in Section 4;
- PLBFGSNLPModel, PLSR1NLPModel, PLSENLPModel using new limited - memory partitioned quasi-Newton operators, described in Section 6;
- PSNLPModel, using the exact second derivatives exploiting partial - separability $B_k v = \nabla^2 f(x_k) v$. It computes every element functions Hessian vector products $\nabla^2 \widehat{f}_i U_i v$ using successively a reverse and forward automatic differentiation passes [21].

When it considers a PBFGSNLPModel, a PSR1NLPModel, a PCSNLPModel or a PSENLPModel, TRUNK is a partitioned quasi-Newton trust-region, and it becomes its limited-memory variant for respectively a PLBFGSNLPModel, a PLSR1NLPModel or a PLSENLPModel.

6 Limited-memory partitioned quasi-Newton method

All the partitioned quasi-Newton methods presented in Section 4 rely on dense matrices to store $\widehat{B}_i$. Such partitioned matrices get a memory cost and a partitioned matrix-vector product complexity in

$$\Theta\left(\sum_{i=1}^{N} \frac{n_i(n_i+1)}{2}\right), \tag{17}$$

which is much smaller than $\frac{n(n+1)}{2}$ if $n_i \ll n$ and N remains moderated. However, as the size of the elements increases, a partitioned matrix suffers from the same issue as unstructured quasi-Newton methods does. Storing each $\widehat{B}_i$ becomes expensive, and it may prevent the storage of N of them, especially if $N \approx n$. In such case, partitioned quasi-Newton methods are impracticable.

We propose partitioned quasi-Newton updates viable for large element partially-separable functions. Each $\nabla^2 \widehat{f}_i$ is approximated by $\widehat{B}_i$, a quasi-Newton linear operators (either LBFGS or LSR1). As a partitioned quasi-Newton method, it may perform an update of rank $\min(n, N)$. Replacing a dense matrix by a linear operator reduces the quadratic memory costs of each $\widehat{B}_i$ from $\Theta(\frac{n_i(n_i+1)}{2})$ to a linear memory costs $\Theta(m_i n_i)$, considering $1 \leq m_i \leq 10$ the memory of the linear operator $\widehat{B}_i$. Aggregated together, the memory cost and the partitioned matrix-vector product complexity conditioned by dense matrices (17) drop to

$$\Theta\left(\sum_{i=1}^{N} m_i n_i\right).$$

Three methods derived from this scheme:

- PLBFGS, approximating every $\widehat{B}_i$ by a LBFGS operator, well suited in case every $\widehat{f}_i$ is convex;
- PLSR1, approximating every $\widehat{B}_i$ by a LSR1 operator, which fits cases where some $\widehat{f}_i$ are non-convex;
- PLSE, using both LBFGS and LSR1 linear operators to retrieve as much local curvatures as possible from every element.

Algorithm 6.1 specifies Algorithm 4.1 when B_k is a limited-memory partitioned quasi-Newton operator: $B_k^{\text{PLBFGS}}, B_k^{\text{PLSE}}$ or B_k^{PLSE}. The resulting trust-regions only differ at step 8, when every $\widehat{B}_{k,i}$ is updated.

Algorithm 6.1 Limited-memory partitioned quasi-Newton trust-Region algorithm (PLBFGS, PLSR1, PLSE)

```
1: Choose $x_0 \in \mathbb{R}^n$, $\Delta_{max} \geq \Delta_0 > 0$, $0 < \eta_1 \leq \eta_2 < 1$, $0 < \gamma_1 \leq \gamma_2 < 1 < \gamma_3 < \gamma_4$, and $0 < \epsilon_1, \epsilon_2$.
2: Choose for every element $i$ a linear operator LBFGS or LSR1 $\widehat{B}_{0,i} = \widehat{B}_{0,i}^\top \approx \nabla^2 \widehat{f}_i(x_0)$.      *initial approximation*
3: for $k = 0, 1, \dots$ do
4:     Compute an approximate solution $s_k$ of
           $$\min_s m_k(s) \quad \text{s.t.} \|s\| \leq \Delta_k, \quad m_k(s) := f(x_k) + \nabla f(x_k)^\top s + \tfrac{1}{2} s^\top \left(\sum_{i=1}^{N} U_i^\top \widehat{B}_{k,i} U_i\right) s,$$
       bringing a sufficient decrease (6).
5:     Compute the ratio
           $$\rho_k := \frac{f(x_k) - f(x_k + s_k)}{m_k(0) - m_k(s_k)}.$$
6:     if $\rho_k \geq \eta_1$ then      *successful step*
7:         set $x_{k+1} = x_k + s_k$, $y_k := \nabla f(x_{k+1}) - \nabla f(x_k)$
8:         update every $\widehat{B}_{k,i}$ to:
               $\widehat{B}_{k+1,i}^{\text{LBFGS}}$ for $B_k^{\text{PLBFGS}}, B_k^{\text{PLSE}}$ if $\widehat{s}_{k,i}^\top \widehat{y}_{k,i} \geq \epsilon_1$,
               $\widehat{B}_{k+1,i}^{\text{LSR1}}$ for $B_k^{\text{PLSR1}}$ if $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \epsilon_2 \|\widehat{s}_{k,i}\| \|\widehat{z}_{k,i}\|$,
               $\widehat{B}_{k+1,i}^{\text{LSR1}}$ for $B_k^{\text{PLSE}}$ if $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \epsilon_2 \|\widehat{s}_{k,i}\| \|\widehat{z}_{k,i}\|$ and $\widehat{s}_{k,i}^\top \widehat{y}_{k,i} < \epsilon_1$,
           given $\widehat{s}_{k,i} := \widehat{s}_{k,i}$, $\widehat{y}_{k_i} := \nabla \widehat{f}_i(\widehat{x}_{k+1,i}) - \nabla \widehat{f}_i(\widehat{x}_{k,i})$ and $\widehat{z}_{k,i} := \widehat{y}_{k,i} - \widehat{B}_{k,i} \widehat{s}_{k,i}$.
           When the numerical safeguards fail, then $\widehat{B}_{k+1,i} = \widehat{B}_{k,i}$ for $B_k^{\text{PLBFGS}}$, $B_k^{\text{PLSR1}}$ or $B_k^{\text{PLSE}}$.
9:     else      *unsuccessful step*
10:        set $x_{k+1} = x_k$ and every $\widehat{B}_{k+1,i} = \widehat{B}_{k,i}$.
11:    end if
12:    Update the trust-region radius according to
           $$\Delta_{k+1} \in \begin{cases} [\gamma_3 \Delta_k,\ \gamma_4 \Delta_k] & \text{if } \rho_k \geq \eta_2, \\ [\gamma_2 \Delta_k,\ \Delta_k] & \text{if } \eta_1 \leq \rho_k < \eta_2, \\ [\gamma_1 \Delta_k,\ \gamma_2 \Delta_k] & \text{if } \rho_k < \eta_1. \end{cases} \qquad (18)$$
13: end for
```

6.1 Global convergence proof of limited-memory quasi-Newton methods (PLBFGS and PLSR1)

The present section exposes a convergence to first-order critical points from Algorithm 6.1. It considers assumptions made Section 1:

Assumption 1 (Assumptions on the problem)**.** The objective f is bounded below on $\mathbb{R}^n$. f is partially-separable (2) and each $\widehat{f}_i$ is twice continuously differentiable on $\mathbb{R}^{n_i}$.

Consequently, f is twice continuously differentiable on $\mathbb{R}^n$.

The convergence proof will rely on [9, Theorem 8.4.7]. To do so, we must show [9, assumption AM.4d]:

$$\sum_{k=1}^{\infty} \frac{1}{\varphi_k} = +\infty, \tag{19}$$

for any B_k being a limited-memory partitioned quasi-Newton operator: PLBFGS, PLSR1 or PLSE. When $B_k \approx \nabla^2 f$, whose value is the same at any point of the trust-region then

$$\varphi_k := 1 + \max_{j=1,\dots,k} \|B_j\|. \tag{20}$$

Every B_j can be divided in two parts. First, $\|B_j^{(0)}\|$, which aggregates all element initializers $B_j^{(0)} := \sum_{i=1}^{N} U_i \widehat{B}_{j,i}^{(0)} U_i^\top$. Second, $\|B_j - B_j^{(0)}\|$ which accumulates the contribution of low rank quasi-Newton update from every element $B_j - B_j^{(0)} = \sum_{i=1}^{N} U_i^\top \left(\widehat{B}_{j,i}^{(\min(m,j))} - \widehat{B}_{j,i}^{(0)} \right) U_i$. Thus, the left part of (19) may be bounded as:

$$\sum_{k=1}^{\infty} \frac{1}{\varphi_k} \geq \sum_{k=1}^{\infty} \frac{1}{1 + \|B_j^{(0)}\| + \|B_j - B_j^{(0)}\|}. \tag{21}$$

We will prove that the right's serie diverges, verifying (19) and AM4.d holds. The proof is divided in two parts. The first one proves

$$\sum_{k=1}^{\infty} \frac{1}{\|B_j^{(0)}\|} = +\infty, \tag{22}$$

under reasonable assumption, while the second bounds $\|B_k - B_k^{(0)}\|$ for any k, by adding suitable numerical safeguards onto the pairs $\widehat{s}_i, \widehat{y}_i$ collected by every quasi-Newton operator $\widehat{B}_i$. Combining both parts makes B_k verify (19) directly.

There is several choices to initialize $\widehat{B}_{k,i}^{(0)}$ at each iterate. Thus, bounding $\|B_k^{(0)}\|$ is not trivial without further assumptions. To let a degree of freedom on how each $\widehat{B}_{k,i}^{(0)}$ is chosen, and allow a growth of $\|B_k^{(0)}\|$, we assume

Assumption 2 (Maximal element initializer serie diverges)**.**

$$\sum_{k=1}^{\infty} \frac{1}{\max_{i=1,\dots,N} \|\widehat{B}_{k,i}^{(0)}\|} = +\infty. \tag{23}$$

Lemma 1. *If Assumption 2 holds, then*

$$\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} = +\infty. \tag{24}$$

Proof. As $B_k^{(0)} = \sum_{i=1}^{N} U_i^\top \widehat{B}_{k,i}^{(0)} U_i$, then

$$B_k^{(0)} \leq \sum_{i=1}^{N} \|U_i\|^2 \|\widehat{B}_{k,i}^{(0)}\| \leq N \max_{i=1,\dots,N} \|U_i\|^2 \max_{i=1,\dots,N} \|\widehat{B}_{k,i}^{(0)}\|, \tag{25}$$

where $N \max_{i=1,\dots,N} \|U_i\|^2 > 0$ is a constant. Therefore,

$$\sum_{k=1}^{\infty} \frac{1}{B_k^{(0)}} \geq \frac{1}{N \max_{i=1,\dots,N} \|U_i\|^2} \sum_{k=1}^{\infty} \frac{1}{\max_{i=1,\dots,N} \|\widehat{B}_{k,i}^{(0)}\|} = +\infty, \tag{26}$$

□

Now that the desired property $\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} = +\infty$ is verified, we focus on bounding $\|B_j - B_j^{(0)}\|$. We do so by adding numerical safeguards on the quasi-Newton updates of every $\widehat{B}_{k,i}$.

Lemma 2 (Boundness of an element quasi-Newton update). *For any $\widehat{B}_{k,i}$, simulating m quasi-Newton updates (BFGS or SR1), then*

$$\|\widehat{B}_{k,i}^{(j+1)} - \widehat{B}_{k,i}^{(j)}\| \leq \Omega^{\text{QNU}1}, \tag{27}$$

is bounded for any iterate k, any element i and for all $0 \leq j \leq m-1$.

Proof. The bounds of the two quasi-Newton updates BFGS and SR1 are distinguished, since they require different additional numerical safeguards.

Suppose that the j-th quasi-Newton update from $\widehat{B}_{k,i}^{(j)}$ to $\widehat{B}_{k,i}^{(j+1)}$ is an SR1 update. We enforce the usual safeguard $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \epsilon_2 \|\widehat{s}_{k,i}\| \|\widehat{z}_{k,i}\|$ at every iteration k with $|\widehat{s}_{k,i}^\top(\widehat{y}_{k,i} - \widehat{B}_{k,i}\widehat{s}_{k,i})| \geq \omega_{\text{SR1}} \|\widehat{y}_{k,i} - \widehat{B}_{k,i}\widehat{s}_{k,i}\|^2$, where $\omega_{\text{SR1}} > 0$. This way,

$$\|\widehat{B}_{k,i}^{(j+1)^{\text{SR1}}} - \widehat{B}_{k,i}^{(j)}\| \leq \frac{\|\widehat{s}_{k-m+j,i} - \widehat{B}_{k-m+j,i}\widehat{y}_{k-m+j,i}\|^2}{|\widehat{s}_{k-m+j,i}^\top(\widehat{y}_{k-m+j,i} - \widehat{B}_{k-m+j,i}\widehat{s}_{k-m+j,i})|} \leq \omega_{\text{SR1}}^{-1}. \tag{28}$$

In the case where the j-th quasi-Newton update from $\widehat{B}_{k,i}^{(j)}$ to $\widehat{B}_{k,i}^{(j+1)}$ is an BFGS update, $\widehat{s}_{k,i}^\top \widehat{y}_{k,i} \geq \epsilon_1$ is enforced by two additional safeguards:

$$|\widehat{y}_{k,i}^\top \widehat{s}_{k,i}| \geq \omega_{\text{BFGS}_1} \|\widehat{y}_{k,i}\|^2, \text{ and } |\widehat{s}_{k,i}^\top \widehat{B}_{k,i}\widehat{s}_{k,i}| \geq \omega_{\text{BFGS}_2} \|\widehat{B}_{k,i}\widehat{s}_{k,i}\|^2, \tag{29}$$

where $\omega_{\text{BFGS}_1} > 0$ and $\omega_{\text{BFGS}_2} > 0$. These safeguards provide the bound

$$\begin{aligned}\|\widehat{B}_{k,i}^{(j+1)^{\text{BFGS}}} - \widehat{B}_{k,i}^{(j)}\| &\leq \frac{\|\widehat{B}_{k-m+j,i}\widehat{y}_{k-m+j,i}\|^2}{|\widehat{s}_{k-m+j,i}^\top \widehat{B}_{k-m+j,i}\widehat{s}_{k-m+j,i}|} + \frac{\|\widehat{y}_{k-m+j,i}\|^2}{|\widehat{y}_{k-m+j,i}^\top \widehat{s}_{k-m+j,i}|} \\ &\leq \omega_{\text{BFGS}_1}^{-1} + \omega_{\text{BFGS}_2}^{-1}.\end{aligned}$$

Therefore, $\|\widehat{B}_{k,i}^{(j+1)} - \widehat{B}_{k,i}^{(j)}\|$ is bounded whether it represents a BFGS update or an SR1 update:

$$\|\widehat{B}_{k,i}^{(j+1)} - \widehat{B}_{k,i}^{(j)}\| \leq \max(\omega_{\text{BFGS}_1}^{-1} + \omega_{\text{BFGS}_2}^{-1}, \omega_{\text{SR1}}^{-1}) = \Omega^{\text{QNU}}, \tag{30}$$

for any iteration k and any element i. □

This proof considers that if an element Hessian approximation doesn't match the numerical conditions related to its nature, for example satisfying the curvature condition for a LBFGS operator, then the update is skipped. Another choice might be using a damped $\tilde{\widehat{y}}_{k,i}$ satisfying the numerical safeguards. In both cases, Lemma 2 remains true.

[1]QNU stands for quasi-Newton update.

Lemma 3 (Boundness of an element limited-memory quasi-Newton operator). *For any $\widehat{B}_{k,i}$, simulating m quasi-Newton updates, then*

$$\|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\| \leq \Omega^{\mathrm{LM}[2]}, \tag{31}$$

is bounded for any iterate k, any element i independently of the quasi-Newton updates simulated by $\widehat{B}_{k,i}$.

Proof. We decompose $\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}$ as

$$\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)} = \widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(m-1)} + \widehat{B}_{k,i}^{(m-1)} - \widehat{B}_{k,i}^{(m-2)} + \cdots + \widehat{B}_{k,i}^{(1)} - \widehat{B}_{k,i}^{(0)}, \tag{32}$$

to make appear the content of Lemma 2. Thus, we find an upper-bound of $\|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\|$ by applying m times the upper-bound from Lemma 2:

$$\|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\| \leq \sum_{j=1}^{m} \|\widehat{B}_{k,i}^{(j)} - \widehat{B}_{k,i}^{(j-1)}\| \leq \sum_{j=1}^{m} \Omega^{\mathrm{QNU}} \leq m\Omega^{\mathrm{QNU}} = \Omega^{\mathrm{LM}}, \tag{33}$$

for any iteration k and any element i. □

The more interesting feature of Lemma 2 and Lemma 3 is that there is no distinction between BFGS and SR1 updates. Therefore, those lemmas handle naturally the changes of quasi-Newton updates which may be performed by a single element Hessian approximation $\widehat{B}_{k,i}$, as B_k^{PLSE} may do. Moreover, Ω^{LM} also bound limited-memory quasi-Newton operators $\widehat{B}_{k,i}$ which have performed less than m quasi-Newton updates. This situation occurs for every element for the $m-1$ first iterates of an optimization method.

Lemma 4 (Boundness of limited-memory partitioned quasi-Newton update). *For any B_k a limited-memory partitioned quasi-Newton update as describe in Section 6 and Algorithm 6.1, then*

$$\|B_k - B_k^{(0)}\| \leq \Omega, \tag{34}$$

for any iteration k.

Proof. By definition,

$$B_k - B_k^{(0)} = \sum_{i=1}^{N} U_i^\top \left(\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\right) U_i. \tag{35}$$

Thus, the norm may be bounded as:

$$\|B_k - B_k^{(0)}\| \leq \sum_{i=1}^{N} \|U_i\|^2 \|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\| \leq \max_{i=1,\ldots,N} \|U_i\|^2 \sum_{i=1}^{N} \|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\|. \tag{36}$$

Lemma 3 provides an upper-bound for any $\|\widehat{B}_{k,i}^{(m)} - \widehat{B}_{k,i}^{(0)}\|$ which translates to (36) as:

$$\|B_k - B_k^{(0)}\| \leq \max_{i=1,\ldots,N} \|U_i\|^2 \sum_{i=1}^{N} \Omega^{\mathrm{LM}} \leq N \max_{i=1,\ldots,N} \|U_i\|^2 \Omega^{\mathrm{LM}} = \Omega, \tag{37}$$

for any iteration k, since $N \max_{i=1,\ldots,N} \|U_i\|^2$ is a constant again. □

Theorem 1. *If Assumption 2 holds, then any limited-memory partitioned quasi-Newton trust-region method presented in Algorithm 6.1: PLBFGS, PLSR1 and PLSE verifies:*

$$\sum_{k=1}^{\infty} \frac{1}{\varphi_k} = +\infty. \tag{38}$$

[2] LM stands for limited-memory.

Proof. We know that

$$\|B_k\| \leq \|B_k - B_k^{(0)}\| + \|B_k^{(0)}\| \leq \Omega + \|B_k^{(0)}\|, \tag{39}$$

from Lemma 4. Therefore, we get

$$\frac{1}{\|B_k\|} \geq \frac{1}{\Omega + \|B_k^{(0)}\|} \text{ and } \sum_{k=1}^{\infty} \frac{1}{\|B_k\|} \geq \sum_{k=1}^{\infty} \frac{1}{\Omega + \|B_k^{(0)}\|}. \tag{40}$$

Assumption 2 being verified, Lemma 1 provides:

$$\sum_{k=1}^{\infty} \frac{1}{\|B_k^{(0)}\|} = +\infty, \tag{41}$$

which gives straightforwardly

$$\sum_{k=1}^{\infty} \frac{1}{\Omega + \|B_k^{(0)}\|} = +\infty. \tag{42}$$

□

Theorem 1 proves [9, AM4.d] leaving only two assumptions to make before fulfilling the conditions for [9, Theorems 8.4.7].

Assumption 3. [9, AN.1] There exists a constant $\kappa_{une} \geq 1$ such that, for all k,

$$\frac{1}{\kappa_{une}} \|x\|_p \leq \|x\| \leq \kappa_{une} \|x\|_p,$$

for all $x \in \mathbb{R}^n$.

Assumption 4. [9, AM.4f]

$$\lim_{k \to \infty, k \in S} \varphi_k \left(f(x_k) - f(x_{k+1}) \right) = 0, \tag{43}$$

where S is the set of successful iterations.

Theorem 2 (Boundness of limited-memory partitioned quasi-Newton opeartors (PLBFGS, PLSR1 and PLSE)). *Suppose that Assumption 1, Assumption 3 and Assumption 4 holds. The sequence of points x_k produces by each of the limited-memory partitioned quasi-Newton trust-region methods: PLBFGS, PLSR1, PLSE formalized by the Algorithm 6.1 converges.*

$$\lim_{k \to \infty} \|\nabla f(x_k)\| = 0.$$

Proof. Since Assumption 1 holds and B_k is a partitioned linear operator, then m_k is C^2 and $f(x_k) = m_k(0)$. Thus, [9, AF.1-2, AM.1, AM.2, AM.3] are covered. The choice of the truncated conjugate-gradient [46] to solve (5) covers [9, AA.1]. γ_3 and γ_4 fulfill [9, AA.4]. Theorem 1 covers [9, AM.4d]. By adding Assumption 4, all the conditions mandatory for [9, theorem 8.4.7]: AF.1-3, AM.1, AM.2, AM.3, AA.1, AA.4, AM.4d are met. Thus, we can state $\lim_{k \to \infty} \|\nabla f(x_k)\| = 0$. □

7 Numerical results

The results are split in two parts. The first part compares all the quasi-Newton trust-region methods presented in this paper on a set of 65 partially-separable problems from Orban et al. [40] whose partial separability details are informed in Appendix A.1. They are based from Algorithm 4.1 on which a backtracking line search along s_k [9, 10.3.2] is performed when s_k is an unsuccessful step. The following methods are:

- PS and TRUNK where $B_k v = \nabla^2 f(x_k)v$ in the trust-region sub-problem (5). TRUNK computes $\nabla^2 f(x_k)v$ using successively a reverse and a forward automatic differentiation on f [21], while PS aggregates the contribution of every $\nabla^2 \widehat{f}_i(\widehat{x}_{k,i})U_i v$, from the same automatic differentiation procedure applied on $\widehat{f}_i$;
- LBFGS_TR and LSR1_TR (TR for trust-region), which considers respectively B_k as a LBFGS operator or a LSR1 operator;
- the partitioned quasi-Newton methods considering $B_k v = \sum_{i=1}^N U_i \widehat{B}_{k,i} U_i v$. When every $\widehat{B}_{k,i}$ is a dense matrix, as defined in Section 4, the methods are:
  - PBFGS where each $\widehat{B}_{k,i}$ is updated with the BFGS formula if $\widehat{y}_{k,i}^\top \widehat{s}_{k,i} > \epsilon_1 > 0$ or skip otherwise;
  - PSR1 where each $\widehat{B}_{k,i}$ is updated with SR1 formula if $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \omega \, \|\widehat{s}_{k,i}\| \|\widehat{z}_{k,i}\|$, where $\widehat{z}_i := \widehat{y}_{k,i} - \widehat{B}_{k,i}\widehat{s}_{k,i}$ and $\omega > 0$ or skip otherwise;
  - PCS updating the convex elements $\widehat{B}_{k,i}$ with BFGS and the non-convex elements with SR1, according to the same numerical safeguards;
  - PSE updating $\widehat{B}_{k,i}$ by BFGS if $\widehat{y}_{k,i}^\top \widehat{s}_{k,i} > \epsilon_1$ or else by SR1.

  When every $\widehat{B}_{k,i}$ is a linear operator, as defined Section 6, the methods are:
  - PLBFGS where each $\widehat{B}_{k,i}$ is a LBFGS operator, updated only when $\widehat{y}_{k,i}^\top \widehat{s}_{k,i} > \epsilon_1$;
  - PLSR1 where each $\widehat{B}_{k,i}$ is a LSR1 operator, updated only when $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \epsilon_2 \|\widehat{s}_{k,i}\| \|\widehat{z}_{k,i}\|$, $|\widehat{s}_{k,i}^\top \widehat{z}_{k,i}| \geq \omega_{\text{SR1}} \|\widehat{z}_{k,i}\|$, and $|\widehat{s}_{k,i}^\top \widehat{y}_{k,i}| \geq \epsilon_1$;
  - PLSE where some $\widehat{B}_{k,i}$ are LBFGS operators and the rest of them are LSR1 operators. Equally to PSE, if $\widehat{y}_{k,i}^\top \widehat{s}_{k,i} > \epsilon_1$, then the pair $\widehat{s}_{k,i}, \widehat{y}_{k,i}$ will make $\widehat{B}_{k+1,i}$ perform a (L)BFGS update or else perform a (L)SR1 update, according to the same numerical safeguards .

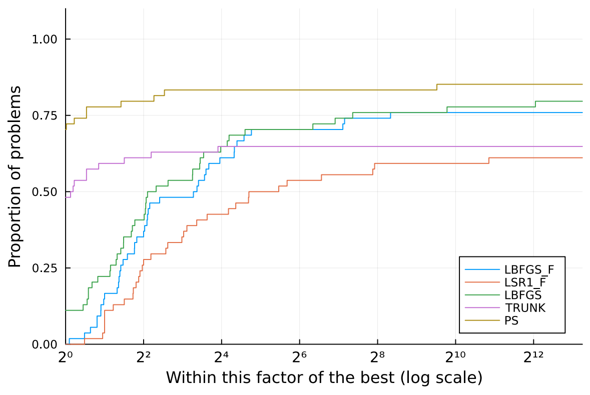
  If an element i can't satisfy all the numerical safeguards, then $\widehat{B}_{k+1,i} = \widehat{B}_{k,i}$.

For a comparative purpose, we add LBFGS, a line search method using a linear operator $H_k = B_k^{-1}$, similar to Liu and Nocedal [30]. The results are summarized as performances profiles [35], from criteria such as time and the number of iterations before it obtains a first order convergence criteria. The second part compares partitioned quasi-Newton methods depending on the nature of element Hessian approximation $\widehat{B}_{k,i}$ either a limited-memory operator or a dense matrix. This study details an artificial partially-separable problem with an element's size growing as the problem's size increases, pushing classical partitioned quasi-Newton methods to their limits.

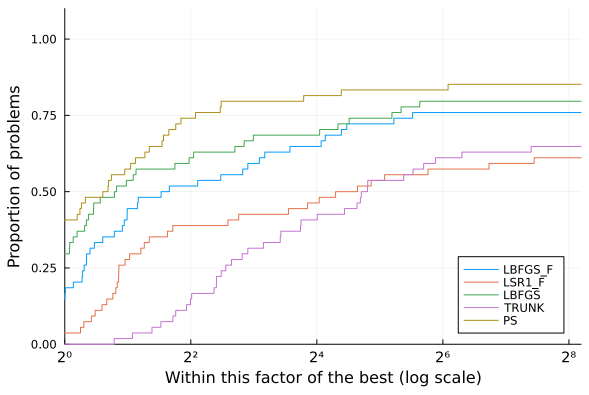
7.1 Comparing quasi-Newton methods

For the following profiles, the partially-separable problems considered have generally elements of (very) small size compared to the problem's size (e.g. $n_i \ll n = 5000$). More specific information about the partial separability each problem of the set can be found in Appendix A.1. The profiles consider a maximal budget of time and iterations. When a solver runs for more than one hour or reach 50 000 evaluation of the objective, it stops. Such executions don't attain the absolute or the relative first-order convergence criteria, respectively $\nabla f(x_k) \leq 10^{-6}$ or $\nabla f(x_k) \leq 10^{-6} \nabla f(x_0)$, where x_0 the initial point.

Figure 3 shows the performance profiles for Newton and quasi-Newton methods. PS utterly dominates TRUNK in time due to faster evaluations of $\nabla^2 f(x_k)v$, solving the trust-region sub-problem faster. Among quasi-Newton methods, LBFGS and LBFGS_TR display similar performances making LSR1_TR the poorer method on this set of problems. While for quasi-Newton methods $B_k \approx \nabla^2 f$, Newton methods perform fewer iterations, as $B_k = \nabla^2 f$. However, the gap of iterations between Newton and quasi-Newton is not translated to time consumption. Even if the partial separability is exploited, making a real difference between PS and TRUNK, the time gap is not closed with quasi-Newton methods.



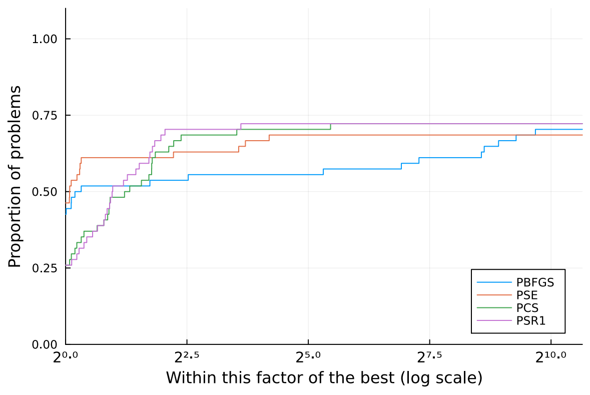
(a) Profile of iterations



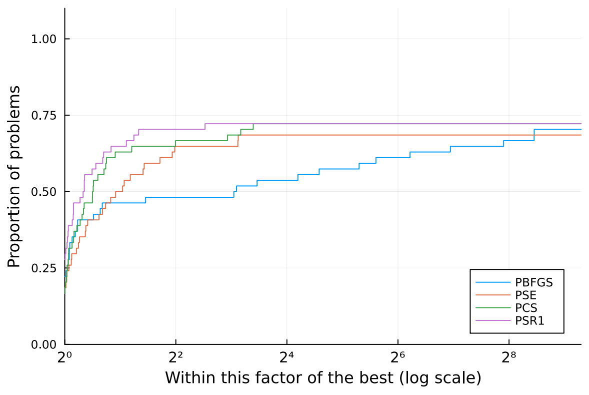
(b) Profile of time

Figure 3: Iteration and time performance profiles for Newton and quasi-Newton methods

Figure 4 compares PBFGS, PSR1, PCS and PSE. In terms of iteration, PSE and PSR1 come on top, while PCS follows closely their trajectories and PBFGS makes the most iterations to generally solve a problem. Figure 4b PSR1 performing better in time than PSE making it the most efficient partitioned quasi-Newton method.



(a) Profile of iterations

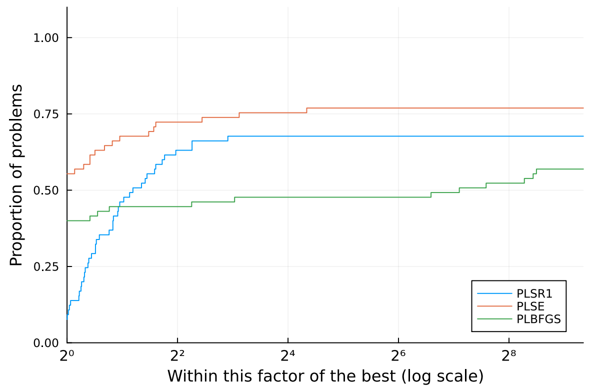


(b) Profile of time

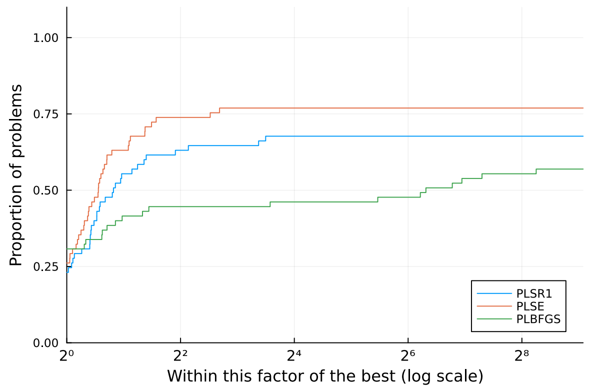
Figure 4: Iteration and time performance profiles for partitioned quasi-Newton methods

Figure 5 presents the limited-memory variants of Figure 4. Among PLBFGS, PLSR1 and PLSE, PLSE solves most problems with fewer iterations and less time than PLBFGS and PLSR1. Strangely, PLSR1 does not reproduce the performances of PSR1 among the usual partitioned quasi-Newton methods. However, you can note that the gap in iterations (Figure 5a) is not automatically transferred to the time (Figure 5b). This is due to the change of the quasi-Newton linear operator between $\widehat{B}_{k,i}$ and $\widehat{B}_{k+1,i}$, which induces allocations and inevitably downgrade the run time efficiency. Moreover, from Figure 4 and Figure 5, both PBFGS and PLBFGS seem to struggle, since the element functions from the partially-separable problems are not necessarily convex.

The last profiles, Figure 6 gathered most significant methods from previous profiles to analyze it together. As expected, the Newton method PS performs less iterates than other methods but takes a lot more time. The LBFGS line search, as the only method not exploiting the partial separability, requires more iterations and time but manage to solve overall more problems than partitioned methods. Among partitioned methods, the amount of iterations required differs slightly but remains clustered. The visible differences for those methods are more about the time required to solve the problems. Figure 6b seems to shrink the difference between PSE and PSR1 presented in Figure 4b, showing how close those methods are compared to the other methods. Although these problems don't favour limited-memory partitioned quasi-Newton methods, PLSE remains pertinent and more performant than Newton methods or unstructured quasi-Newton method.

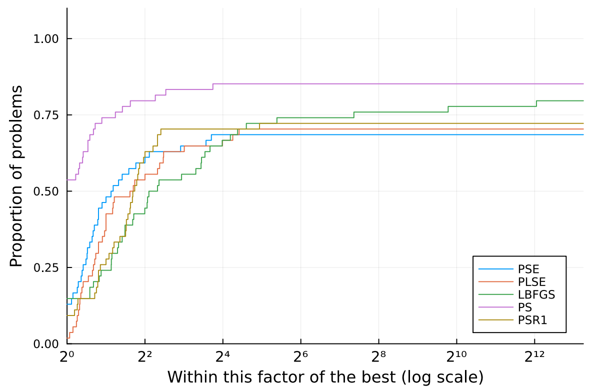


(a) Profile of iterations

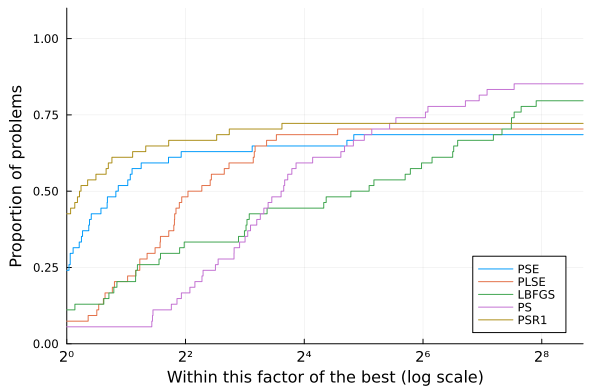


(b) Profile of time

Figure 5: Iteration and time performance profiles for limited-memory partitioned quasi-Newton methods



(a) Profile of iterations



(b) Profile of time

Figure 6: Summary of performance profiles for (limited-memory) partitioned quasi-Newton methods

7.2 Experiments about limited-memory partitioned quasi-Newton methods

The following artificial partially-separable function

$$\sum_{j=1}^{\frac{n}{\sqrt{n}}-3} \frac{\left(\sum_{i=(j-1)\sqrt{n}}^{(j+2)\sqrt{n}} i * x_i\right)^2}{1+x_j^2} + \sum_{j=1}^{\frac{n}{\sqrt{n}}-5} \frac{\left(\sum_{i=(j-1)\sqrt{n}+5}^{(j+4)\sqrt{n}+5} i * x_i\right)^2}{1+x_j^2}, \tag{44}$$

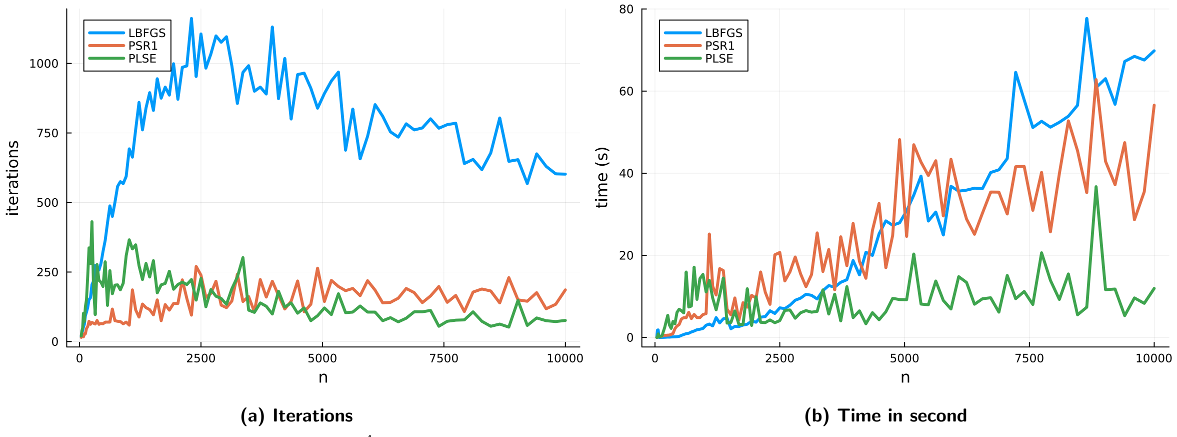
is made to push gradually the standard partitioned quasi-Newton methods on their limits as the size n grows. It has $N \approx 2\sqrt{n} - 8$ element functions: $\widehat{f}_j = \frac{\left(\sum_{i=(j-1)\sqrt{n}}^{(j+2)\sqrt{n}} i*x_i\right)^2}{1+x_j^2}$ for $1 \leq j \leq \sqrt{n}-3$ and $\widehat{f}_j = \frac{\left(\sum_{i=(j-1)\sqrt{n}+5}^{(j+4)\sqrt{n}+5} i*x_i\right)^2}{1+x_j^2}$ for $\sqrt{n}-3 \leq j \leq 2\sqrt{n}-8$. Therefore, as n increases, the number of element increases, as well as their sizes. By doing so, it pushes standard partitioned quasi-Newton methods to store large dense matrices $\widehat{B}_i$.

$name$	N	$min.ED$	$meanED$	$max.ED$	$min.EC$	$meanEC$	$max.EC$
$limit36$	4	18	21.75	31	0	2.41667	4
$limit625$	42	75	100.238	127	0	6.736	9
$limit2500$	92	150	200.38	252	0	7.374	9
$limit10000$	192	300	400.443	502	0	7.6885	9

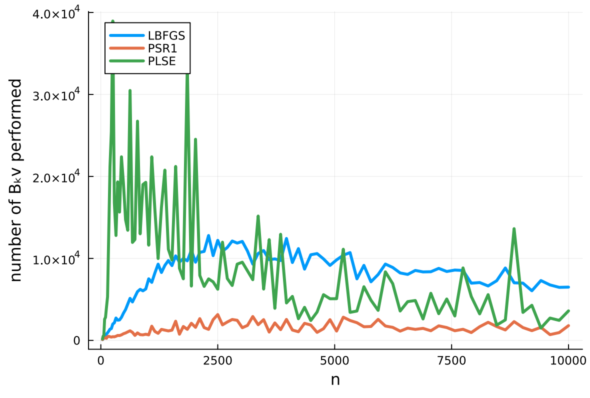
,

where ED stands for element dimension and EC stands for element contribution, counting how many elements contribute to a single variable.

The figures in Figure 7 respectively express the iterations, the time and number of product $B_k v$ required by a LBFGS, a PSR1 or a PLSE trust-region method before it reaches a first-order criteria, and seeks to show the interest of the PLSE method.



(a) Iterations (b) Time in second



(c) Number of $B_k v$ performed

Figure 7: Trust-region methods applied to solve (44)

Figure 7a shows that partitioned quasi-Newton methods require fewer iterations than LBFGS to converge, and seems to be independent to n. Both PSR1 and PLSE iterations remain particularly low, even for large instances. This behavior, inherent to partitioned methods, is explained by a superior quality of B, which approximates more accurately $\nabla^2 f$. By not considering the partially-separable structure of (44), a LBFGS Hessian approximation B_k lacks accuracy compared to partitioned methods.

Figure 7b informs about the time required to converge, and it differs from the iterations. While the amount of iterations for LBFGS seems to stagnate, the time needed increases as n grows. On the contrary, PSR1 and PLSE follow the same tendencies as their iteration curves. However, spikes of iterations in Figure 7a are not always echoed by a spike in Figure 7b. The reason is that one of the main cost of our trust-region methods is the truncated conjugate gradient is the product $B_k v$ provided by Figure 7c. In some cases, it can overshadow the cost related to evaluation of f or ∇f. The amount of product $B_k v$ performed is not directly related to the number of iterations, see for example the spike of PSE in Figure 7c round $n = 9\,200$, which does not having a corresponding spike in Figure 7a, but the spike is echoed in Figure 7b with a time's spike as well. The limit of these partitioned methods for such a partially-separable function is the cost of B_k and $B_k v$. While Figure 7c shows stable number of $B_k v$ for PSR1, its time increase in Figure 7b. Storing individually $\widehat{B}_{k,i}$ as a dense matrix becomes quadratically heavier as n_i grows, making any operation relying on B_k more costly. In this context, PLSE, by using linear operators (see Section 6) for every $\widehat{B}_{k,i}$, stays applicable. Figure 7b shows that the growth of the time for PLSE remains moderated compared to PSR1, even if Figure 7c shows PLSE performing more $B_k v$ product than PSR1. This study proves that limited-memory partitioned quasi-Newton methods are more suited for large element partially-separable functions, by keeping an accurate partitioned approximation of the Hessian, a reasonable memory requirement and an efficient $B_k v$ product.

8 Future works and conclusion

At the moment of the submission, our framework does not support all extensions available in LANCELOT, mainly:

- internal element variables: U_i rows are linear combination of variables appearing in $\widehat{f}_i$ [7];
- partitioned trust-region method: each element function $\widehat{f}_i$ has its own radius $\Delta_{k,i}$ [8];
- a sophisticated merging element procedure, to enhance the memory footprint of a partitioned quasi-Newton operator and the performance of its linear application [7], developed in Section 4.

While being aware of these functionalities, we chose to stick first with our research, about limited-memory partitioned quasi-Newton operators, and we set cornerstones to add gradually these important features later.

In the same time, we plan to enhance the variety element Hessian approximations, for example, supporting diagonal quasi-Newton operators to approximate all 1 sized nonlinear element functions merged together. In addition, we want to address the drawback of PLSE by introducing new quasi-Newton linear operators which could capitalize on both LSR1 and LBFGS updates without untimely allocations. That way, each $\widehat{B}_{k,i}$ from PLSE would get more depth, by considering generally more update from the last iterates. We will also push to define a partitioned quasi-Newton operator that could aggregate element Hessian contribution of different nature, combining dense element matrices, element linear operators and element diagonal quasi-Newton linear operators. Such features will broaden the portfolio of partitioned quasi-Newton operators supported. The last step would be to integrate partially-separable constraints, to exploit the group partial separability of the resulting augmenting Lagrangian model.

The extension of the JuliaSmoothOptimizer library presented recognizes and exploits automatically the partial separability of f, to minimize it through partitioned quasi-Newton methods. We proposed three new limited-memory partitioned quasi-Newton updates: PLBFGS, PSLR1, PLSE accompany by their global convergence proofs, to extend the scope of partially-separable problems which could be practically solved. Integrated in JuliaSmoothOptimizer solvers, the same solver's code furnish as many partitioned methods as there are partitioned quasi-Newton operators. The performance profiles in Section 7.1 justify the interest on partitioned quasi-Newton methods, while Section 7.2 legitimates their limited memory variants for partially-separable functions having large elements.

A Appendix

A.1 Partially-separable problem structures

This appendix details in Table 1 the information of the partially-separable problems considered producing the performance profiles Section 7. In order to restrain column's heading sizes, consider:

- **mid** for minimal element dimension;
- **med** for mean element dimension;
- **mad** for maximal element dimension;
- **mic** for minimal elemental contribution (for 1 variables);
- **mec** for mean elemental contribution (for 1 variables);
- **mac** for maximal elemental contribution (for 1 variables).

Table 1: Partial separability details of the partially-separable problems set

name	n	N	M	constant	linear	quadratic	cubic	general	convex	concave	general	mid	med	mad	mic	mec	mac
arwhead5000	5000	9999	3	1	4999	0	0	4999	9999	5000	0	0	1.49985	2	2	2.9994	4999
bdqrtic5000	5000	9992	2	0	0	4996	0	4996	9992	0	0	1	3.0	5	1	5.9952	4996
brybnd5000	5000	5000	7	0	0	0	0	5000	0	0	5000	2	6.9968	7	2	6.9968	7
chnrosnb5000	5000	9998	5000	0	0	4999	0	4999	4999	0	4999	1	1.5	2	1	2.9994	3
clplatea5000	4900	19045	5	0	1	9522	0	9522	19045	1	0	1	1.9927	2	0	7.7451	8
clplateb4900	4900	19114	5	0	70	9522	0	9522	19114	70	0	1	1.98912	2	0	7.75918	8
clplatec4900	4900	19046	7	0	2	9522	0	9522	19046	2	0	1	1.99265	2	0	7.74531	8
cragglvy4900	4900	12245	5	0	0	2449	0	9796	7347	0	4898	1	1.6	2	2	3.99837	4
dixmaane4900	4899	9799	6534	1	0	6532	0	3266	4900	1	4899	0	1.49985	2	3	3.0	3
dixmaanf4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaang4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaanh4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaani4899	4899	9799	6534	1	0	6532	0	3266	4900	1	4899	0	1.49985	2	3	3.0	3
dixmaanj4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaank4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaanl4899	4899	14697	6535	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaanm4899	4899	9799	9799	1	0	6532	0	3266	4900	1	4899	0	1.49985	2	3	3.0	3
dixmaann4899	4899	14697	14697	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaano4899	4899	14697	14697	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixmaanp4899	4899	14697	14697	1	0	6532	0	8164	4900	1	9797	0	1.66653	2	4	4.99959	5
dixon3dq4899	4899	4899	2	0	0	4899	0	0	4899	0	0	1	1.99959	2	1	1.99959	2
dqdrtic4899	4899	4899	5	0	0	4899	0	0	4899	0	0	1	1.0	1	1	1.0	1
dqrtic4899	4899	4899	4899	0	0	0	0	4899	4899	0	0	1	1.0	1	1	1.0	1
edensch4899	4899	14695	4	1	0	4898	0	9796	9797	1	4898	0	1.33324	2	2	3.99918	4
engval14899	4899	9797	3	1	4898	0	0	4898	9797	4899	0	0	1.49985	2	1	2.99939	3
errinros4899	4899	9796	4899	0	0	4898	0	4898	4898	0	4898	1	1.5	2	1	2.99939	3
extrosnb4899	4899	4899	2	0	0	1	0	4898	1	0	4898	1	1.9998	2	1	1.9998	2
fletcbv24899	4899	14698	5	0	4899	4900	0	4899	9799	4899	4899	1	1.33324	2	4	4.0	4
fletcbv34899	4899	14698	4	0	0	4900	0	9798	4900	0	9798	1	1.33324	2	4	4.0	4
freuroth4899	4899	9796	4	0	0	0	0	9796	0	0	9796	2	2.0	2	2	3.99918	4
genhumps4899	4899	9797	3	0	0	4899	0	4898	4899	0	4898	1	1.49995	2	2	2.99959	3
genrose4899	4899	9797	3	1	0	4898	0	4898	4899	1	4898	0	1.49985	2	1	2.99939	3
genrose-nash4899	4899	9797	3	1	0	4898	0	4898	4899	1	4898	0	1.49985	2	1	2.99939	3
morebv4899	4899	4899	4899	0	0	0	0	4899	0	0	4899	2	2.99959	3	2	2.99959	3
ncb204899	4899	14667	4893	1	4888	10	10	9758	9788	4889	4879	0	7.30872	20	1	21.8814	23
noncvxu24899	4899	9798	6	0	0	4899	0	4899	4899	0	4899	2	2.99959	3	4	5.99918	10
noncvxun4899	4899	9796	6	0	0	4898	0	4898	4898	0	4898	1	2.99959	3	2	5.99796	10
nondia4899	4899	4899	2	0	0	1	0	4898	1	0	4898	1	1.9998	2	1	1.9998	4899
nondquar4899	4899	4899	2	0	0	2	0	4897	4899	0	0	2	2.99959	3	2	2.99959	4898
penalty34899	4899	2454	6	1	0	2449	0	4	2450	1	4	0	8.98289	4899	3	4.49969	5
powellsg4899	4896	4896	4	0	0	2448	0	2448	4896	0	0	2	2.0	2	2	2.0	2
quartc4896	4896	4896	4896	0	0	0	0	4896	4896	0	0	1	1.0	1	1	1.0	1
sbrybnd4896	4896	4896	4896	0	0	0	0	4896	0	0	4896	2	6.99673	7	2	6.99673	7
schmvett4896	4896	14682	3	0	0	0	0	14682	0	0	14682	2	2.33333	3	2	6.99714	7
sinquad4896	4896	4896	3	0	0	0	0	4896	1	0	4895	1	2.99939	3	1	2.99939	4896
sparsine4896	4896	4896	4896	0	0	0	0	4896	0	0	4896	1	5.99285	6	4	5.99285	9
sparsqur4896	4896	4896	4896	0	0	0	0	4896	4896	0	0	1	5.99285	6	4	5.99285	9
spmsrtls4896	4897	9791	9791	0	0	0	0	9791	0	0	9791	1	2.16658	3	3	4.33184	5
srosenbr4897	4896	4896	2	0	0	2448	0	2448	2448	0	2448	1	1.5	2	1	1.5	2
tointgss4896	4896	4894	2	0	0	0	0	4894	0	0	4894	3	3.0	3	1	2.99877	3
tquartic4896	4896	4895	2	0	0	1	0	4894	1	0	4894	1	1.9998	2	0	1.99939	4895
tridia4896	4896	4896	4896	0	0	4896	0	0	4896	0	0	1	1.9998	2	1	1.9998	2
vardim4896	4896	4898	3	0	0	4897	0	1	4898	0	0	1	2.99878	4896	3	3.0	3
woods4896	4896	7344	5	0	0	4896	0	2448	4896	0	2448	1	1.66667	2	2	2.5	3

A.2 UML of a subpart of the julia smooth optimizers ecosystem

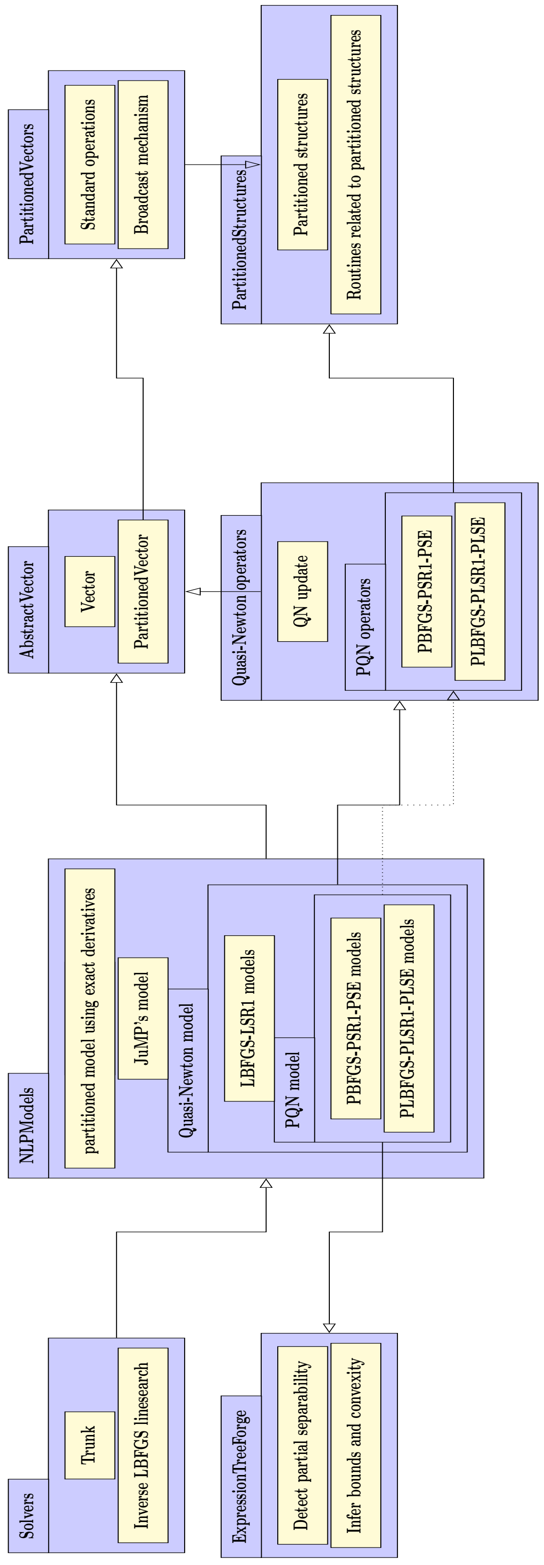


Figure 8: Abstract type and interface dependencies of JSO related to partial separability

References

[1] International business machine corporation: Mathematical programming system/360 version 2, linear and separable programming-user's manual. mps standard. Technical Report H20–0476-2, IBM Corporation, 1969.

[2] Dimitri P. Bertsekas. Projected newton methods for optimization problems with simple constraints. SIAM Journal on Control and Optimization, 20(2):221–246, 1982. DOI: 10.1137/0320018. URL `https://doi.org/10.1137/0320018`.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. 59 (1):65–98, 2017. DOI: 10.1137/141000671.

[4] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. ACM Transactions on Mathematical Software, 21(1):123–160, 1995.

[5] R. H. Byrd, J. Nocedal, and R. A. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, Large-Scale Nonlinear Optimization, pages 35–59. New York, 2006. DOI: 10.1007/0-387-30065-1˙4.

[6] Richard H. Byrd, Jorge Nocedal, and Robert B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. 63(1):129–156, Jan 1994. DOI: 10.1007/BF01582063.

[7] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. Computing Methods in Applied Sciences and Engineering, pages 42–54, 1990. DOI: 10.1007/BF02592099.

[8] A R Conn, N I M Gould, A Sartenaer, and Ph. L. Toint. Convergence properties of minimization algorithms for convex constraints using a structured trust region. 6(4):1059–1086, 1996. DOI: 10.1137/S1052623492236481.

[9] Andrew R. Conn, Nicholas I. M. Gould, and Ph. L. Toint. Trust Region Methods. 2000. DOI: 10.1137/1.9780898719857. MPS-SIAM Series on Optimization.

[10] William C. Davidon. Optimally conditioned optimization algorithms without line searches. 9(1):1–30, Dec 1975. DOI: 10.1007/BF01681328.

[11] I. S. Duff and J. K. Reid. MA27–a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE R10533, 1982.

[12] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. ACM Transactions on Mathematical Software, 9(3):302–325, September 1983. DOI: 10.1145/356044.356047.

[13] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. SIAM Review, 59(2):295–320, 2017. DOI: 10.1137/15M1020575.

[14] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. 36: 519–554, 1990. DOI: 10.1287/mnsc.36.5.519.

[15] R Fourer, C Maheshwari, A Neumaier, D Orban, and H Schichl. Convexity and concavity detection in computational graphs: Tree walks for convexity assessment. 22(1):26–43, 2010. DOI: 10.1287/ijoc.1090.0321.

[16] David M Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, Computational differentiation: techniques, applications, and tools, Proceedings of the second international workshop on computational differentiation, pages 173–184, February 1996. URL `https://ampl.com/REFS/ad96.pdf`.

[17] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. 47(1):99–131, 2005. DOI: 10.1137/S0036144504446096.

[18] N. I. M. Gould, D. Orban, and Ph. L. Toint. Cuter and sifdec: A constrained and unconstrained testing environment, revisited. ACM Trans. Math. Softw., 29(4):373–394, dec 2003. DOI: 10.1145/962437.962439.

[19] N. I. M. Gould, D. Orban, and Ph. L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. ACM Transactions on Mathematical Software, 29(4):353–372, 2003. DOI: 10.1145/962437.962438.

[20] N. I. M. Gould, D Orban, and Ph. L. Toint. CUTEst: a constrained and unconstrained testing environment with safe threads for mathematical optimization. 60(3):545–557, 2015. DOI: 10.1007/s10589-014-9687-3.

[21] A Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, Mathematical Programming: recent developments and applications, Mathematics and its applications, pages 83–108. 1989.

[22] A Griewank and Ph. L. Toint. Partitioned variable metric updates for large structured optimization problems. 39:119–137, 1982. DOI: 10.1007/BF01399316.

[23] A Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M. J. D Powell, editor, Nonlinear Optimization 1981, pages 301–312. 1982. Publication editors : M.J.D. Powell.

[24] A Griewank and Ph. L. Toint. Local convergence analysis for partitioned quasi-Newton updates. 39(3): 429–448, 1982. DOI: 10.1007/BF01407874.

[25] A Griewank and Ph. L. Toint. Numerical experiments with partially separable optimization problems. In David F. Griffiths, editor, Numerical Analysis, pages 203–220, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg. DOI: 10.1007/BFb0099526.

[26] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. 49(6):409–436, 1952. DOI: 10.6028/jres.049.044.

[27] HSL Mathematical Software Library. A catalogue of subroutines (HSL 2000), 2000. URL http://www.hsl.rl.ac.uk.

[28] P. Raynaud J.Bigeon, D. Orban. PartiallySeparableNLPModels.jl: Automatic detection and exploitation of partial separability toward partitioned quasi-newton methods. https://github.com/JuliaSmoothOptimizers/PartiallySeparableNLPModels.jl, July 2022.

[29] M Lescrenier. Partially separable optimization and parallel computing. 14(1):213–224, 1988. DOI: 10.1007/BF02186481.

[30] D Liu and J Nocedal. On the limited memory BFGS method for large scale optimization. 45(1-3):503–528, 1989. DOI: 10.1007/BF01589116.

[31] Sanae Lotfi, Tiphaine Bonniot de Ruisselet, Dominique Orban, and Andrea Lodi. Stochastic damped L-BFGS with controlled norm of the hessian approximation. CoRR, abs/2012.05783, 2020. URL https://arxiv.org/abs/2012.05783.

[32] Xuehua Lu. A computational study of the limited memory SR1 method for unconstrained optimization. University of Colorado, Boulder, 1996.

[33] Miles Lubin and Iain Dunning. Computing in operations research using julia. INFORMS Journal on Computing, 27(2):238–248, 2015. DOI: 10.1287/ijoc.2014.0623.

[34] A. Montoison, D. Orban, and contributors. Krylov.jl: A Julia basket of hand-picked Krylov methods. https://github.com/JuliaSmoothOptimizers/Krylov.jl, June 2020.

[35] Jorge J. Moré and Stefan M. Wild. Benchmarking derivative-free optimization algorithms. SIAM Journal on Optimization, 20(1):172–191, 2009. DOI: 10.1137/080724083.

[36] D. Orban, A. S. Siqueira, and contributors. LinearOperators.jl. https://github.com/JuliaSmoothOptimizers/LinearOperators.jl, September 2020.

[37] D. Orban, A. S. Siqueira, and contributors. NLPModels.jl: Data structures for optimization models. https://github.com/JuliaSmoothOptimizers/NLPModels.jl, July 2020.

[38] D. Orban, A. S. Siqueira, and contributors. JSOSolvers.jl: JuliaSmoothOptimizers optimization solvers. https://github.com/JuliaSmoothOptimizers/JSOSolvers.jl, March 2021.

[39] D. Orban, A. S. Siqueira, and contributors. NLPModelsModifiers.jl: Model modifiers for nlpmodels. https://github.com/JuliaSmoothOptimizers/NLPModelsModifiers.jl, March 2021.

[40] D. Orban, A. S. Siqueira, and contributors. OptimizationProblems.jl: A collection of optimization problems in julia. https://github.com/JuliaSmoothOptimizers/OptimizationProblems.jl, July 2022.

[41] J. Bigeon P. Raynaud, D. Orban. ExpressionTreeForge.jl: A manipulator of expression trees. https://github.com/JuliaSmoothOptimizers/ExpressionTreeForge.jl, July 2022.

[42] J. Bigeon P. Raynaud, D. Orban. PartitionedStructures.jl: Partitioned derivatives storage and partitioned quasi-Newton updates. https://github.com/JuliaSmoothOptimizers/PartitionedStructures.jl, Month 2022.

[43] J. Bigeon P. Raynaud, D. Orban. PartitionedVectors.jl:. https://github.com/JuliaSmoothOptimizers/PartitionedVectors.jl, Month 2022.

[44] D. F. Shanno. On variable-metric methods for sparse Hessians. 34(150):499–514, 190. DOI: 10.1090/S0025-5718-1980-0559198-2.

[45] T Steihaug. The conjugate gradient method and trust regions in large scale optimization. 20(3):626–637, 1983. DOI: 10.1137/0720042.

[46] Trond Steihaug, Shahadat Hossain, and Laurent Hascoët. Structure in optimization: Factorable programming and functions. In Erol Gelenbe and Ricardo Lent, editors, Computer and Information Sciences III, pages 449–458, London, 2013. Springer London. DOI: 10.1007/978-1-4471-4594-3-46.

[47] Ph L. Toint. A sparse quasi-Newton update derived variationally with a nondiagonally weighted Frobenius norm. 37:425–433, 1981. DOI: 10.1090/s0025-5718-1981-0628706-6.

[48] Ph. L. Toint. Test problems for partially separable optimization and results for the routine PSPMIN. Technical Report 83/4, February 1983.

[49] Ph. L. Toint. On large scale nonlinear least squares calculations. 8(3):416–435, 1987. DOI: 10.1137/0908042.

[50] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. 106(1):25–57, 2006. DOI: 10.1007/s10107-004-0559-y.