Square Root Bundle Adjustment for Large-Scale Reconstruction Supplementary Material

Nikolaus Demmel Christiane Sommer Daniel Cremers Vladyslav Usenko
Technical University of Munich
{nikolaus.demmel,c.sommer,cremers,vlad.usenko}@tum.de

In this supplementary material we provide additional background, analysis of computational complexity, and a detailed account of the convergence of all solvers for each of the 97 problems from the BAL dataset used in our evaluation. The latter are the same experiments that are aggregated in performance profiles in Figure 4 of the main paper. Section A includes a concise introduction to Givens rotations and a definition of performance profiles. Section B discusses the computational complexity of our QR-based solver compared to the explicit and implicit Schur complement solvers. Section C presents the size and density of all problems (tabulated in Section E). Finally, Section D discusses the convergence plots (shown in Section F) for all problems, grouped by *ladybug* (F.1), *trafalgar* (F.2), *dubrovnik* (F.3), *venice* (F.4), and *final* (F.5).

A. Additional details

A.1. Givens rotations

The QR decomposition of a matrix A can be computed using Givens rotations. A Givens rotation is represented by a matrix $G_{ij}(\theta)$ that is equal to identity except for rows and columns i and j, where it has the non-zero entries

$$\begin{pmatrix} g_{ii} & g_{ij} \\ g_{ji} & g_{jj} \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}. \tag{29}$$

$G_{ij}(\theta)$ describes a rotation by angle θ in the ij-plane. The multiplication of a matrix A with $G_{ij}(\theta)$ changes only two rows in A, leaving all other rows unchanged. θ can be chosen such that the element (i,j) of $G_{ij}(\theta)A$ is zero:

$$\cos\theta = \frac{a_{jj}}{\sqrt{a_{jj}^2 + a_{ij}^2}}, \quad \sin\theta = \frac{a_{ij}}{\sqrt{a_{jj}^2 + a_{ij}^2}}. \tag{30}$$

By subsequent multiplication of A with Givens matrices, all elements below the diagonal can be zeroed (see [15, p. 252] for the full algorithm). As all $G_{ij}(\theta)$ are orthogonal by construction, their product matrix Q is also orthogonal.

A.2. Performance profiles

Let $\mathcal{P}$ be a set of BA problems and $\mathcal{S}$ be the set of evaluated solvers which we run according to the termination criteria (maximum number of iterations and function tolerance). For a given problem $p \in \mathcal{P}$ and solver $s \in \mathcal{S}$, we define the minimal cost value achieved by that solver after time t as $f(p,s,t)$. The smallest achieved cost by any solver for a specific problem is denoted by $f^*(p) := \min_{s,t} f(p,s,t)$, which we use to define for a chosen accuracy tolerance $0 < \tau < 1$ the cost threshold

$$f_\tau(p) := f^*(p) + \tau(f_0(p) - f^*(p)), \tag{31}$$

where $f_0(p)$ is the initial cost of problem p. The runtime for solver s to achieve that threshold is

$$t_\tau(p,s) := \min\{t \mid f(p,s,t) \le f_\tau(p)\} \cup \{\infty\}. \tag{32}$$

With this, the performance profile of a solver s is

$$\rho_\tau(s,\alpha) := 100 \frac{|\{p \mid t_\tau(p,s) \le \alpha \min_s t_\tau(p,s)\}|}{|\mathcal{P}|}. \tag{33}$$

In other words, $\rho_\tau(s,\alpha)$ maps the relative runtime α to the percentage of problems that s has solved to accuracy τ. The curve is monotonically increasing, starting on the left with $\rho_\tau(s,1)$, the percentage of problems for which solver s is the fastest, and ending on the right with $\max_\alpha \rho_\tau(s,\alpha)$, the percentage of problems on which the solver s achieves the cost threshold $f_\tau(p)$ at all. Comparing different solvers, the curve that is more to the left indicates better runtime and the curve that is more to the top indicates higher accuracy.

B. Algorithm complexities

In Table 2, we compare the theoretical complexity of our QR-based solver to the explicit and implicit Schur complement solvers in terms of number of poses n_p, number of landmarks n_l, and mean μ_o and variance σ_o^2 of the number of observations per landmark. Note that the total number n_o of observations equals $\mu_o n_l$. While most of the entries in the table are easily determined, let us briefly walk through the not so obvious ones:

	$\sqrt{BA}$ (ours)	explicit SC	implicit SC
outer iterations			
Jacobian computation	$\mathcal{O}(\mu_o n_l)$	$\mathcal{O}(\mu_o n_l)$	$\mathcal{O}(\mu_o n_l)$
Hessian computation	0	$\mathcal{O}(\mu_o n_l)$	0
QR	$\mathcal{O}((\mu_o^2 + \sigma_o^2) n_l)$	0	0
middle iterations			
damping	$\mathcal{O}(\mu_o n_l)$	$\mathcal{O}(n_l + n_p)$	0
SC	0	$\mathcal{O}((\mu_o^2 + \sigma_o^2) n_l)$	0
preconditioner	$\mathcal{O}((\mu_o^2 + \sigma_o^2) n_l + n_p)$	$\mathcal{O}(n_p)$	$\mathcal{O}(\mu_o n_l + n_p)$
back substitution	$\mathcal{O}(\mu_o n_l)$	$\mathcal{O}(\mu_o n_l)$	$\mathcal{O}(\mu_o n_l)$
inner iterations			
PCG	$\mathcal{O}((\mu_o^2 + \sigma_o^2) n_l + n_p)$	$\mathcal{O}(n_p^2)$ (worst case)	$\mathcal{O}(\mu_o n_l + n_p)$

Table 2: Complexities of the different steps in our $\sqrt{BA}$ solver compared to the two SC solvers (explicit and implicit), expressed only in terms of n_l, n_p, μ_o, and σ_o. We split the steps into three stages: outer iterations, i.e., everything that needs to be done in order to setup the least squares problem (once per linearization); middle iterations, i.e., everything that needs to be done within one Levenberg-Marquardt iteration (once per outer iteration if no backtracking is required or multiple times if backtracking occurs); inner iterations, i.e., everything that happens within one PCG iteration.

$\sqrt{BA}$ (our solver) Assume landmark j has k_j observations. We can express the sum over k_j^2 by μ_o and σ_o^2:

$$\sigma_o^2 = \mathrm{Var}(\{k_j\}) = \frac{1}{n_l}\sum_{j=1}^{n_l} k_j^2 - \left(\frac{1}{n_l}\sum_{j=1}^{n_l} k_j\right)^2, \tag{34}$$

$$\Rightarrow \sum_{j=1}^{n_l} k_j^2 = n_l(\mu_o^2 + \sigma_o^2). \tag{35}$$

The sum over k_j^2 appears in many parts of the algorithm, as the dense landmark blocks $\begin{pmatrix}\hat{J}_p & \hat{J}_l & \hat{r}\end{pmatrix}$ after QR decomposition have size $(2k_j + 3) \times (d_p k_j + 4)$, where the number of parameters per camera d_p is 9 in our experiments. In the QR step, we need $6k_j$ Givens rotations per landmark (out of which 6 are for the damping), and we multiply the dense landmark block by $\hat{Q}^\top$, so we end up having terms $\mathcal{O}(\sum_j k_j)$ and $\mathcal{O}(\sum_j k_j^2)$, leading to the complexity stated in Table 2. For the block-diagonal preconditioner, each landmark block contributes summands to k_j diagonal blocks, each of which needs a matrix multiplication with $\mathcal{O}(k_j)$ flops, thus we have a complexity of $\mathcal{O}(\sum_j k_j^2)$. Preconditioner inversion can be done block-wise and is thus $\mathcal{O}(n_p)$. In the PCG step, we can exploit the block-sparse structure of $\hat{Q}_2^\top \hat{J}_p$ and again have the k_j^2-dependency. Because of the involved vectors being of size $2n_o + d_p n_p$ (due to pose damping), we additionally have a dependency on $2n_o + d_p n_p$. Finally, for the back substitution, we need to solve n_l upper-triangular 3×3 systems and then effectively do n_l multiplications of a $(3 \times d_p k_j)$ matrix with a vector, which in total is of order $\mathcal{O}(\sum_j k_j)$.

Explicit SC The first step is the Hessian computation. As each single residual only depends on one pose and one landmark, the Hessian computation scales with the number of observations/residuals. Damping is a simple augmentation of the diagonal and contributes terms $\mathcal{O}(n_l)$ and $\mathcal{O}(n_p)$. Matrix inversion of H_{ll} for the Schur complement scales linearly with n_l, while the number of operations to multiply H_{pl} by H_{ll}^{-1} scales with the number of non-zero sub-blocks in H_{pl}, and thus with n_o. The multiplication of this product with H_{lp} involves matrix products of sub-blocks sized $(d_p \times 3)$ and $(3 \times d_p)$ for each camera pair that shares a given landmark, i.e., $\mathcal{O}(k_j^2)$ matrix products for landmark j. The preconditioner can simply be read off from $\tilde{H}_{pp}$, and its inversion is the same as for $\sqrt{BA}$. The matrices and vectors involved in PCG all have size $d_p n_p (\times d_p n_p)$. The sparsity of $\tilde{H}_{pp}$ is not only determined by n_p, n_l, μ_o, and σ_o, but would require knowledge about which cameras share at least one landmark. In the worst case, where each pair of camera poses have at least one landmark they both observe, $\tilde{H}_{pp}$ is dense. Thus, assuming $\tilde{H}_{pp}$ as dense we get quadratic dependence on n_p. Back substitution consists of matrix inversion of n_l blocks, and a simple matrix-vector multiplication.

Implicit SC Since the Hessian matrix is not explicitly computed, we need an extra step to compute the preconditioner for implicit SC. For each pose, we have to compute a $d_p \times d_p$ block for which the number of flops scales linearly with the number of observations of that pose, thus it is $\mathcal{O}(n_o)$ in total. Preconditioner damping contributes the n_p-dependency. As no matrices except for the preconditioner are precomputed for the PCG iterations, but sparsity can again be exploited to avoid quadratic complexities, this part of the algorithm scales linearly with all three numbers (assuming the outer loop for the preconditioner computation is a *parallel for* over cameras, rather than a *parallel reduce* over landmarks). Lastly, back substitution is again only block-wise matrix inversion and matrix-vector multiplications. While avoiding a dependency on $(\mu_o^2 + \sigma_o^2)n_l$ in the asymptotic runtime seems appealing, the implicit SC method computes a sequence of five sparse matrix-vector products in each PCG iteration in addition to the preconditioner multiplication, making it harder to parallelize than the other two methods, which have only one (explicit SC) or two ($\sqrt{BA}$) sparse matrix-vector products. Thus, the benefit of implicit SC becomes apparent only for very large problems. As our evaluation shows, for medium and large problems, i.e. the majority in the BAL dataset, our $\sqrt{BA}$ solver is still superior in runtime.

C. Problem sizes

Table 3 in Section E details the size of the bundle adjustment problem for each instance in the BAL dataset (grouped into *ladybug*, the skeletal problems *trafalgar*, *dubrovnik*, and *venice*, as well as the *final* problems). Besides number of cameras n_p, number of landmarks n_l, and number of observations $n_r = \frac{N_r}{2}$, we also show indicators for *problem density*: the average number of observations per camera *#obs / cam* (which equals n_r/n_p), as well as the average number of observations per landmark *#obs / lm*, including its standard deviation and maximum over all landmarks.

In particular, a high mean and variance of *#obs / lm* indicates that our proposed $\sqrt{BA}$ solver may require a large amount of memory (see for example *final961*, *final1936*, *final13682*), since the dense storage after marginalization in a landmark block is quadratic in the number of observations of that landmark. If on the other hand the problems are sparse and the number of observations is moderate, the memory required by $\sqrt{BA}$ grows only linearly in the number of observations, similar to SC-based solvers (see Figure 6 in the main paper).

D. Convergence

In Section F, each row of plots corresponds to one of the 97 bundle adjustment problems and contains from left to right a plot of optimized cost by runtime (like Figure 5 in the main paper) and by iteration, trust-region size (inverse of the damping factor λ) by iteration, number of CG iterations by (outer) iteration, and peak memory usage by iteration. The cost plots are cut off at the top and horizontal lines indicate the cost thresholds corresponding to accuracy tolerances $\tau \in \{10^{-1}, 10^{-2}, 10^{-3}\}$ as used in the performance profiles. The plot by runtime is additionally cut off on the right at the time the fastest solver for the respective problem terminated.

We make a few observations that additionally support our claims in the main paper: all solvers usually converge to a similar cost, but for most problems our proposed $\sqrt{BA}$ solver is the fastest to reduce the cost. On the other hand, memory use can be higher, depending on problem size and density (see Section C). Missing plots indicate that the respective solver ran out of memory, which for example for $\sqrt{BA}$*-32* happens only on the largest problem *final13682*, where the landmarks have up to 1748 observations. Our single precision solver $\sqrt{BA}$*-32* runs around twice as fast as its double precision counterpart, since it is numerically stable and usually requires a comparable number of CG iterations. This is in contrast to our custom SC solver, where the twofold speedup for single precision is generally not observed. The good numeric properties are further supported by the evolution of the trust-region size approximately following that of the other solvers in most cases. Finally, for the smallest problems (e.g., *ladybug49*, *trafalgar21*, *final93*), the evolution of cost, trust-region size, and even number of CG iterations is often identical for all solvers for the initial 5 to 15 iterations, before numeric differences become noticeable. This supports the fact that the different marginalization strategies are algebraically equivalent and that our custom solver implementation uses the same Levenberg-Marquardt strategy and CG forcing sequence as Ceres.

E. Problem sizes table

	#cam (n_p)	#lm (n_l)	#obs (n_r)	#obs / cam mean	#obs / lm mean	#obs / lm std-dev	#obs / lm max
ladybug49	49	7,766	31,812	649.2	4.1	3.3	29
ladybug73	73	11,022	46,091	631.4	4.2	3.7	40
ladybug138	138	19,867	85,184	617.3	4.3	4.4	48
ladybug318	318	41,616	179,883	565.7	4.3	4.8	89
ladybug372	372	47,410	204,434	549.6	4.3	4.8	134
ladybug412	412	52,202	224,205	544.2	4.3	4.8	135
ladybug460	460	56,799	241,842	525.7	4.3	4.7	135
ladybug539	539	65,208	277,238	514.4	4.3	4.7	142
ladybug598	598	69,193	304,108	508.5	4.4	4.9	142
ladybug646	646	73,541	327,199	506.5	4.4	5.0	144
ladybug707	707	78,410	349,753	494.7	4.5	5.0	145
ladybug783	783	84,384	376,835	481.3	4.5	5.0	145
ladybug810	810	88,754	393,557	485.9	4.4	4.9	145
ladybug856	856	93,284	415,551	485.5	4.5	4.9	145
ladybug885	885	97,410	434,681	491.2	4.5	4.9	145
ladybug931	931	102,633	457,231	491.1	4.5	5.0	145
ladybug969	969	105,759	474,396	489.6	4.5	5.2	145
ladybug1064	1,064	113,589	509,982	479.3	4.5	5.1	145
ladybug1118	1,118	118,316	528,693	472.9	4.5	5.1	145
ladybug1152	1,152	122,200	545,584	473.6	4.5	5.1	145
ladybug1197	1,197	126,257	563,496	470.8	4.5	5.1	145
ladybug1235	1,235	129,562	576,045	466.4	4.4	5.1	145
ladybug1266	1,266	132,521	587,701	464.2	4.4	5.0	145
ladybug1340	1,340	137,003	612,344	457.0	4.5	5.2	145
ladybug1469	1,469	145,116	641,383	436.6	4.4	5.1	145
ladybug1514	1,514	147,235	651,217	430.1	4.4	5.1	145
ladybug1587	1,587	150,760	663,019	417.8	4.4	5.1	145
ladybug1642	1,642	153,735	670,999	408.6	4.4	5.0	145
ladybug1695	1,695	155,621	676,317	399.0	4.3	5.0	145
ladybug1723	1,723	156,410	678,421	393.7	4.3	5.0	145
	#cam (n_p)	#lm (n_l)	#obs (n_r)	#obs / cam mean	#obs / lm mean	#obs / lm std-dev	#obs / lm max
trafalgar21	21	11,315	36,455	1,736.0	3.2	1.8	15
trafalgar39	39	18,060	63,551	1,629.5	3.5	2.4	20
trafalgar50	50	20,431	73,967	1,479.3	3.6	2.7	21
trafalgar126	126	40,037	148,117	1,175.5	3.7	3.0	29
trafalgar138	138	44,033	165,688	1,200.6	3.8	3.3	32
trafalgar161	161	48,126	181,861	1,129.6	3.8	3.4	40
trafalgar170	170	49,267	185,604	1,091.8	3.8	3.5	41
trafalgar174	174	50,489	188,598	1,083.9	3.7	3.4	41
trafalgar193	193	53,101	196,315	1,017.2	3.7	3.4	42
trafalgar201	201	54,427	199,727	993.7	3.7	3.4	42
trafalgar206	206	54,562	200,504	973.3	3.7	3.4	42
trafalgar215	215	55,910	203,991	948.8	3.6	3.4	42
trafalgar225	225	57,665	208,411	926.3	3.6	3.3	42
trafalgar257	257	65,131	225,698	878.2	3.5	3.2	42

	#cam (n_p)	#lm (n_l)	#obs (n_r)	#obs / cam mean	#obs / lm mean	#obs / lm std-dev	#obs / lm max
dubrovnik16	16	22,106	83,718	5,232.4	3.8	2.2	14
dubrovnik88	88	64,298	383,937	4,362.9	6.0	6.0	65
dubrovnik135	135	90,642	552,949	4,095.9	6.1	7.1	84
dubrovnik142	142	93,602	565,223	3,980.4	6.0	7.1	84
dubrovnik150	150	95,821	567,738	3,784.9	5.9	6.9	84
dubrovnik161	161	103,832	591,343	3,672.9	5.7	6.7	84
dubrovnik173	173	111,908	633,894	3,664.1	5.7	6.7	84
dubrovnik182	182	116,770	668,030	3,670.5	5.7	6.9	85
dubrovnik202	202	132,796	750,977	3,717.7	5.7	6.7	91
dubrovnik237	237	154,414	857,656	3,618.8	5.6	6.6	99
dubrovnik253	253	163,691	898,485	3,551.3	5.5	6.6	102
dubrovnik262	262	169,354	919,020	3,507.7	5.4	6.5	106
dubrovnik273	273	176,305	942,302	3,451.7	5.3	6.5	112
dubrovnik287	287	182,023	970,624	3,382.0	5.3	6.5	120
dubrovnik308	308	195,089	1,044,529	3,391.3	5.4	6.3	121
dubrovnik356	356	226,729	1,254,598	3,524.2	5.5	6.4	122

	#cam (n_p)	#lm (n_l)	#obs (n_r)	#obs / cam mean	#obs / lm mean	#obs / lm std-dev	#obs / lm max
venice52	52	64,053	347,173	6,676.4	5.4	5.9	46
venice89	89	110,973	562,976	6,325.6	5.1	5.9	62
venice245	245	197,919	1,087,436	4,438.5	5.5	7.2	85
venice427	427	309,567	1,695,237	3,970.1	5.5	7.2	119
venice744	744	542,742	3,054,949	4,106.1	5.6	8.6	205
venice951	951	707,453	3,744,975	3,937.9	5.3	7.7	213
venice1102	1,102	779,640	4,048,424	3,673.7	5.2	7.5	221
venice1158	1,158	802,093	4,126,104	3,563.1	5.1	7.4	223
venice1184	1,184	815,761	4,174,654	3,525.9	5.1	7.3	223
venice1238	1,238	842,712	4,286,111	3,462.1	5.1	7.3	224
venice1288	1,288	865,630	4,378,614	3,399.5	5.1	7.2	225
venice1350	1,350	893,894	4,512,735	3,342.8	5.0	7.1	225
venice1408	1,408	911,407	4,630,139	3,288.5	5.1	7.1	225
venice1425	1,425	916,072	4,652,920	3,265.2	5.1	7.1	225
venice1473	1,473	929,522	4,701,478	3,191.8	5.1	7.1	226
venice1490	1,490	934,449	4,717,420	3,166.1	5.0	7.1	228
venice1521	1,521	938,727	4,734,634	3,112.8	5.0	7.1	230
venice1544	1,544	941,585	4,745,797	3,073.7	5.0	7.1	231
venice1638	1,638	975,980	4,952,422	3,023.5	5.1	7.1	231
venice1666	1,666	983,088	4,982,752	2,990.8	5.1	7.2	231
venice1672	1,672	986,140	4,995,719	2,987.9	5.1	7.2	231
venice1681	1,681	982,593	4,962,448	2,952.1	5.1	7.2	231
venice1682	1,682	982,446	4,960,627	2,949.2	5.0	7.2	231
venice1684	1,684	982,447	4,961,337	2,946.2	5.0	7.2	231
venice1695	1,695	983,867	4,966,552	2,930.1	5.0	7.2	231
venice1696	1,696	983,994	4,966,505	2,928.4	5.0	7.2	231
venice1706	1,706	984,707	4,970,241	2,913.4	5.0	7.2	232
venice1776	1,776	993,087	4,997,468	2,813.9	5.0	7.1	232
venice1778	1,778	993,101	4,997,555	2,810.8	5.0	7.1	232

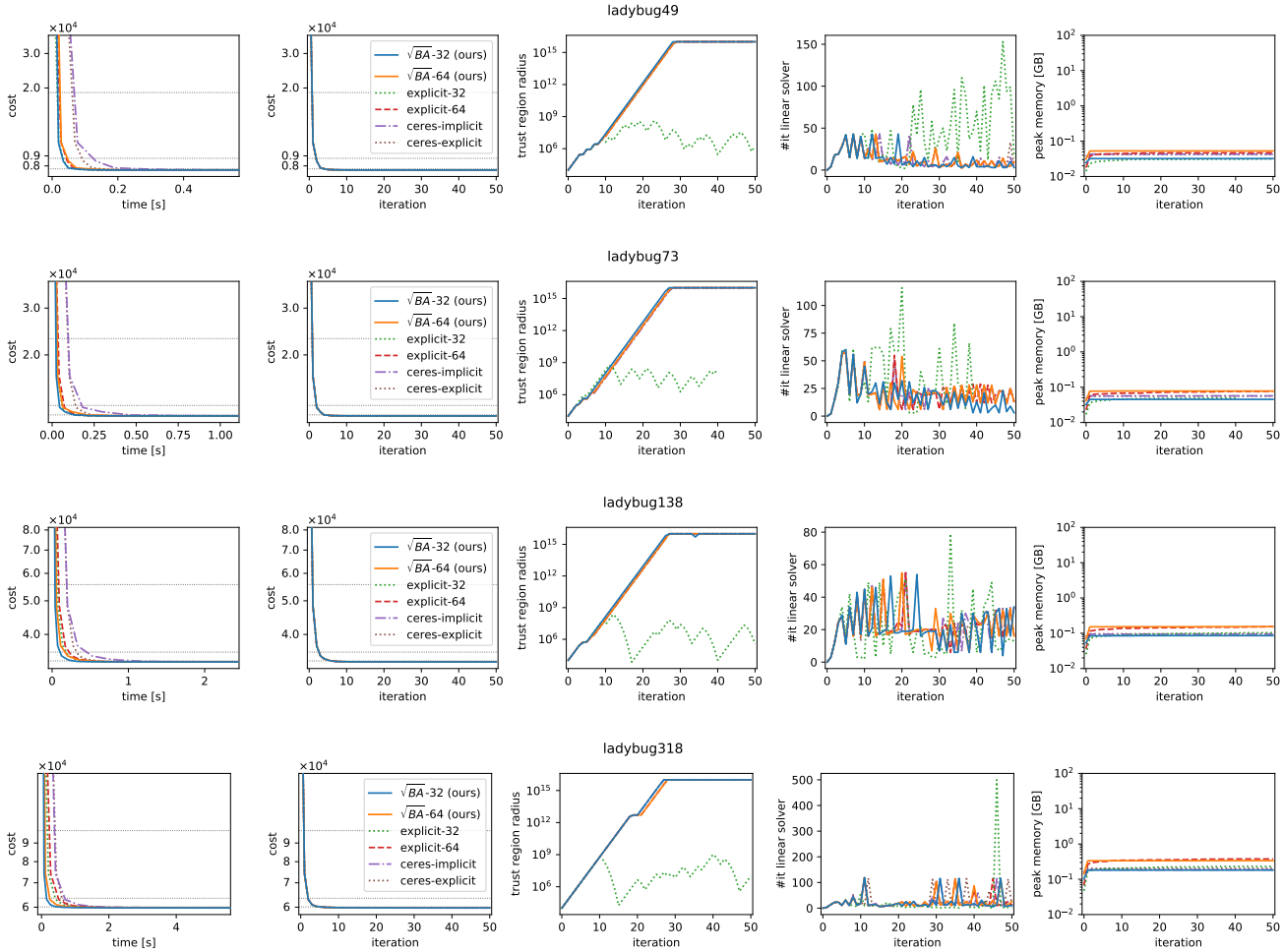
	#cam	#lm	#obs	#obs / cam	#obs / lm		

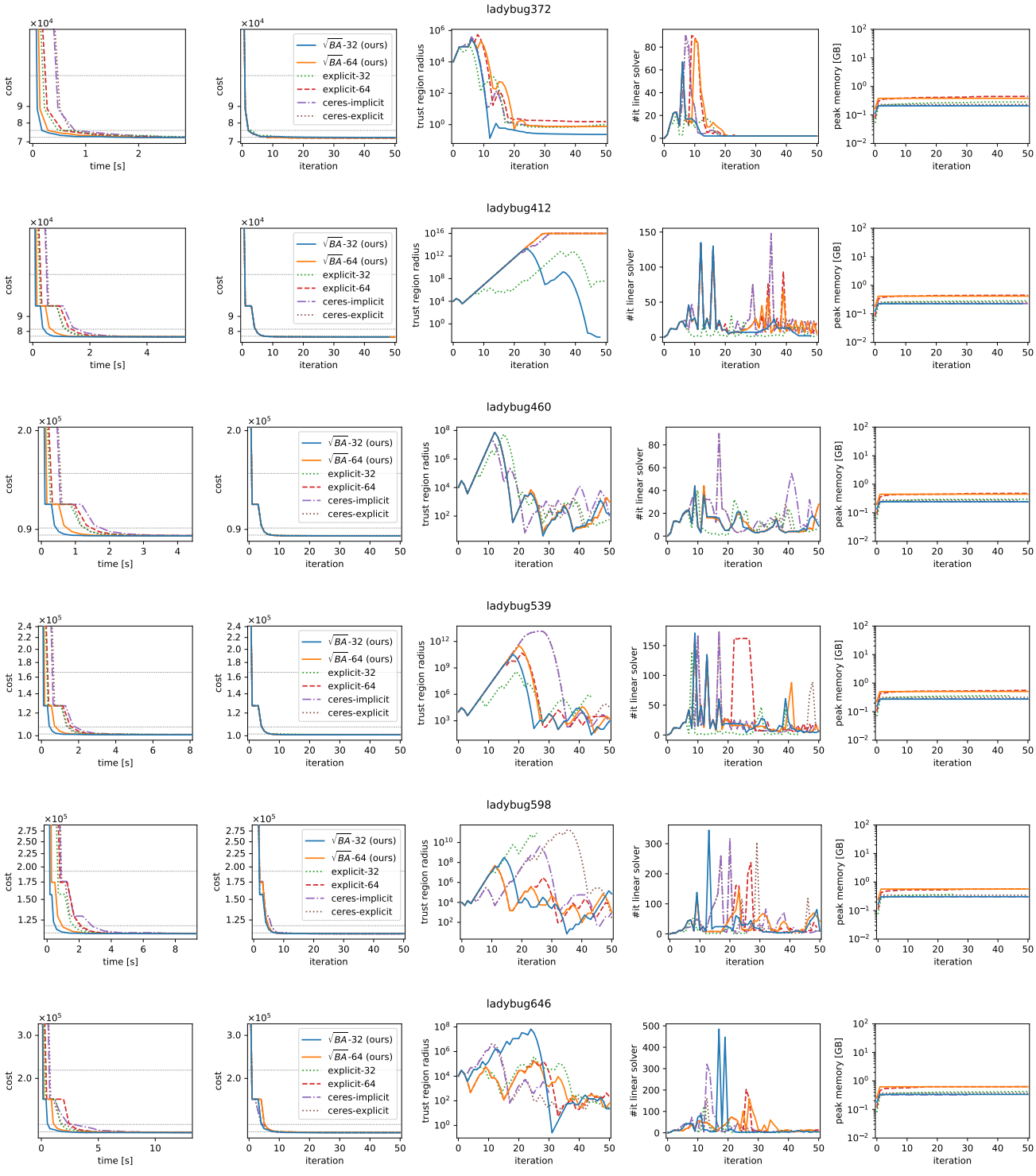
	(n_p)	(n_l)	(n_r)	mean	mean	std-dev	max
final93	93	61,203	287,451	3,090.9	4.7	5.8	80
final394	394	100,368	534,408	1,356.4	5.3	10.6	280
final871	871	527,480	2,785,016	3,197.5	5.3	9.8	245
final961	961	187,103	1,692,975	1,761.7	9.0	29.3	839
final1936	1,936	649,672	5,213,731	2,693.0	8.0	26.9	1293
final3068	3,068	310,846	1,653,045	538.8	5.3	12.6	414
final4585	4,585	1,324,548	9,124,880	1,990.2	6.9	12.6	535
final13682	13,682	4,455,575	28,973,703	2,117.7	6.5	18.9	1748

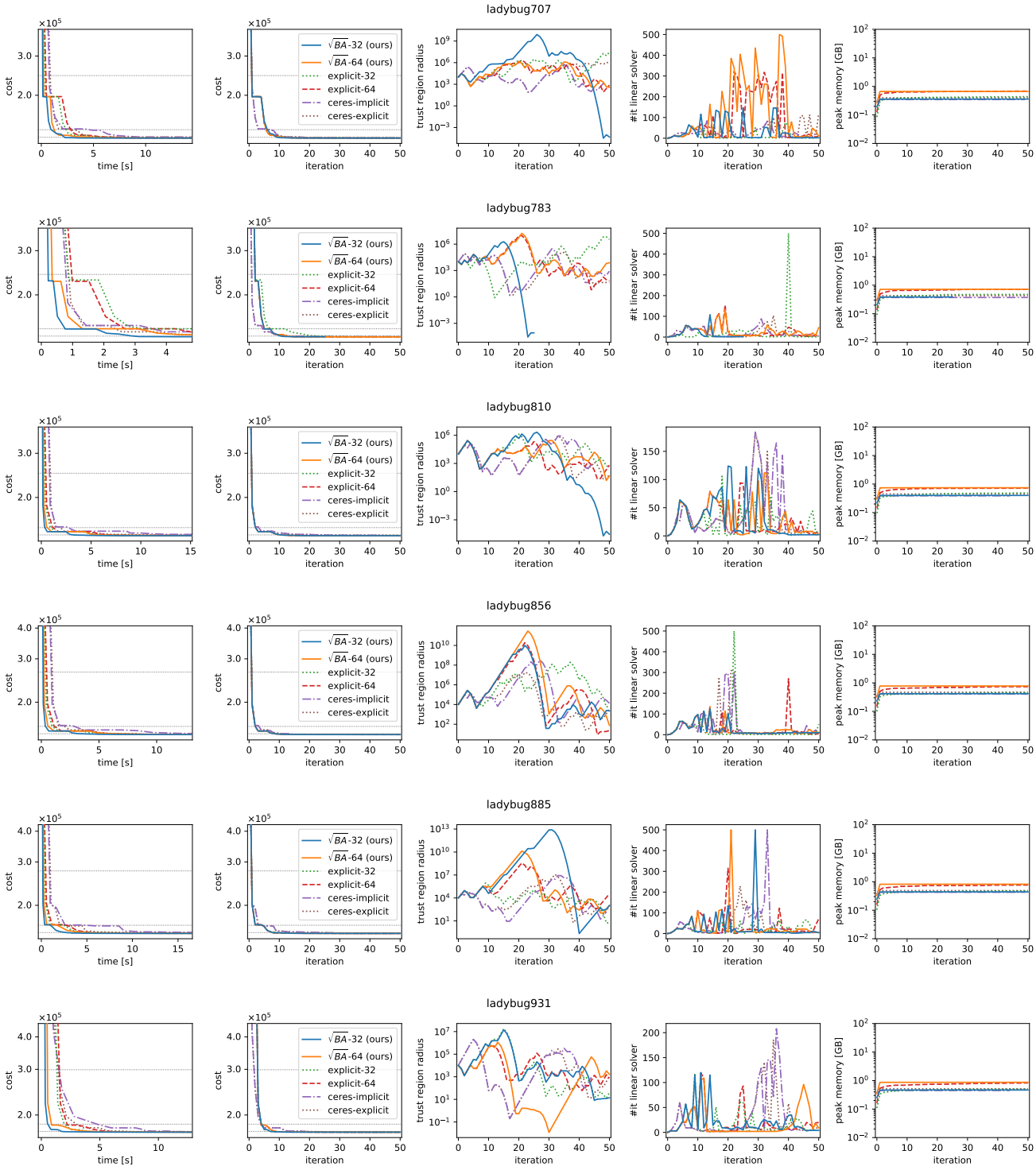
Table 3: Size of the bundle adjustment problem for each instance in the BAL dataset.

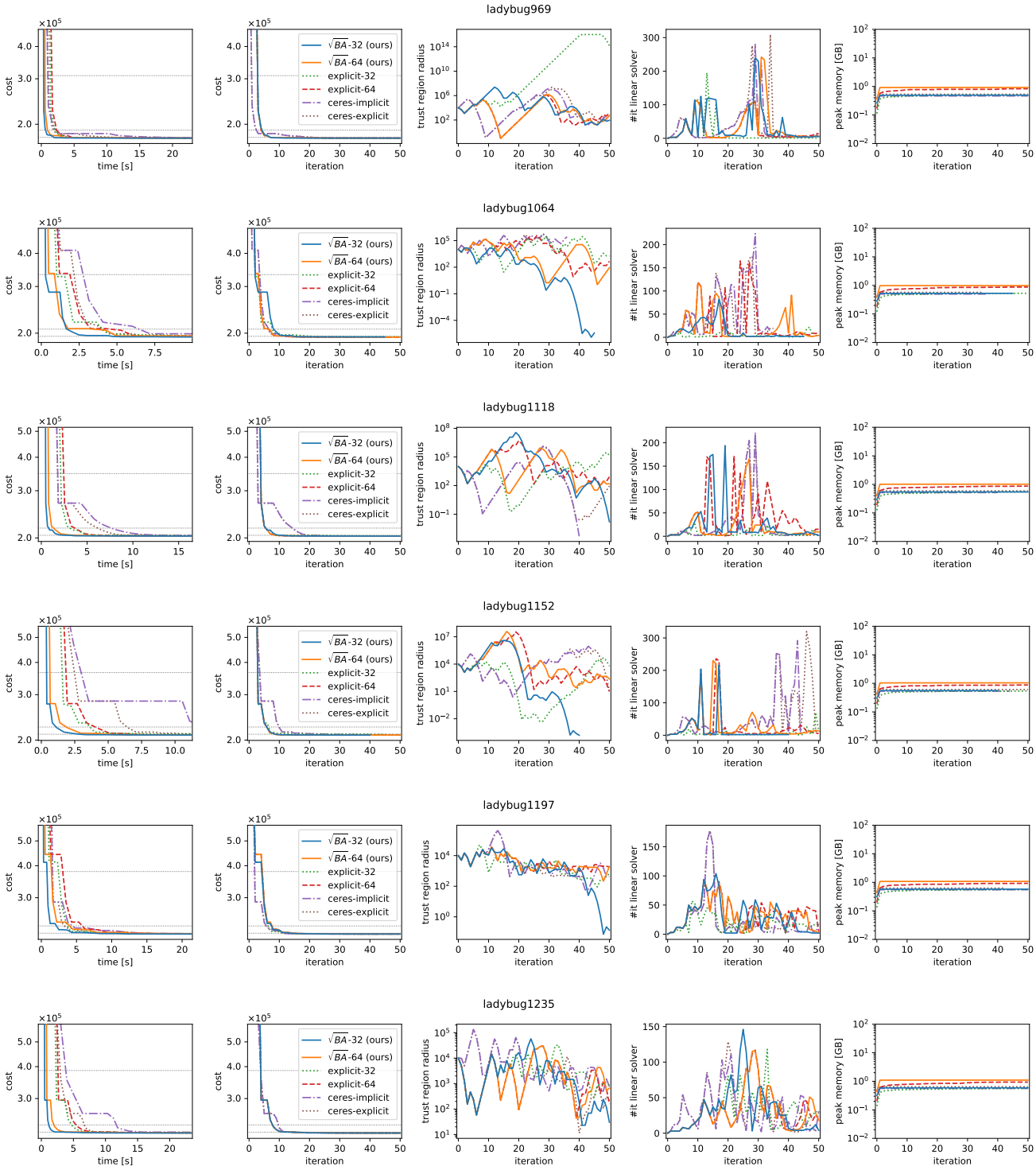
F. Convergence plots

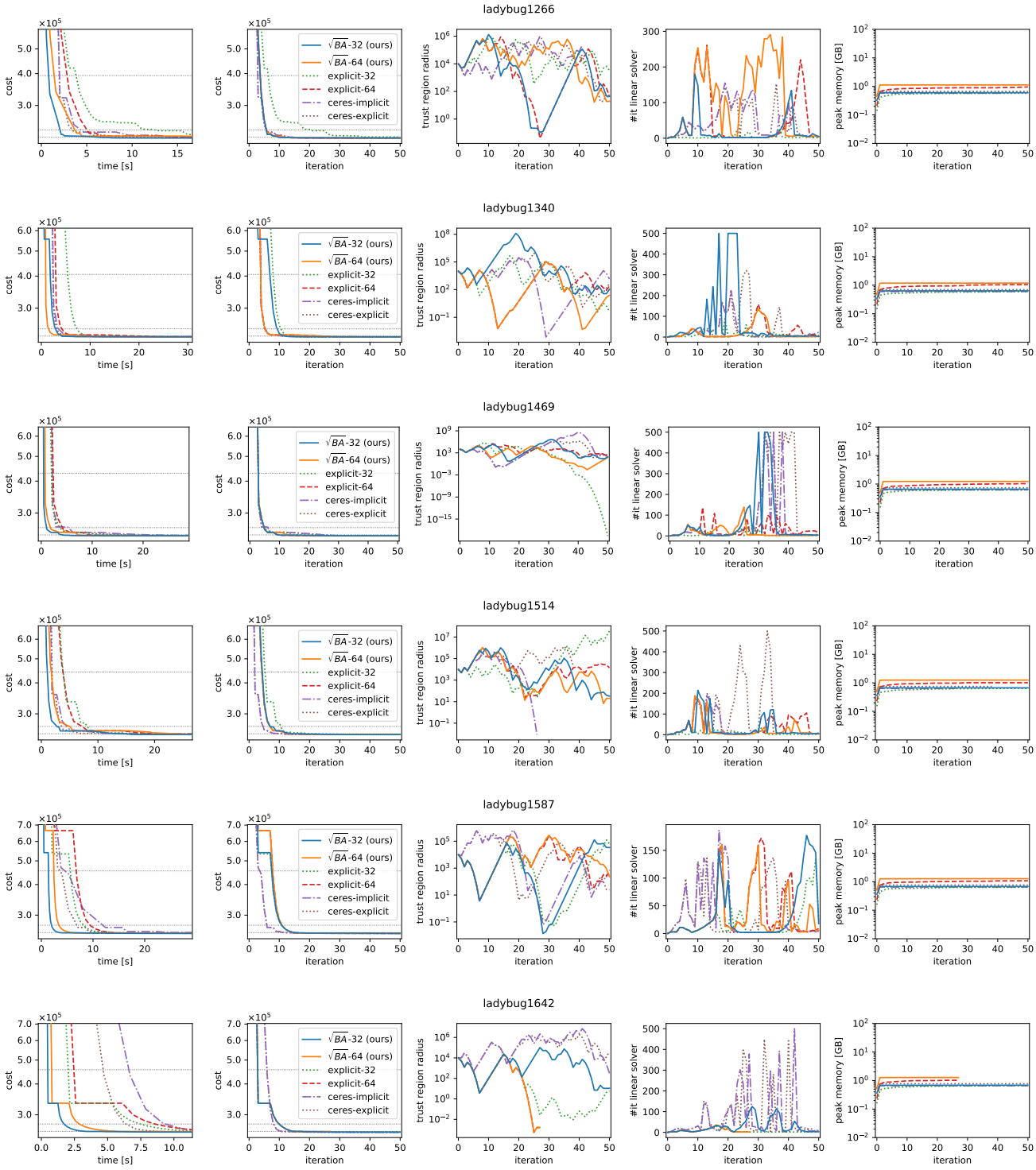
F.1. Ladybug

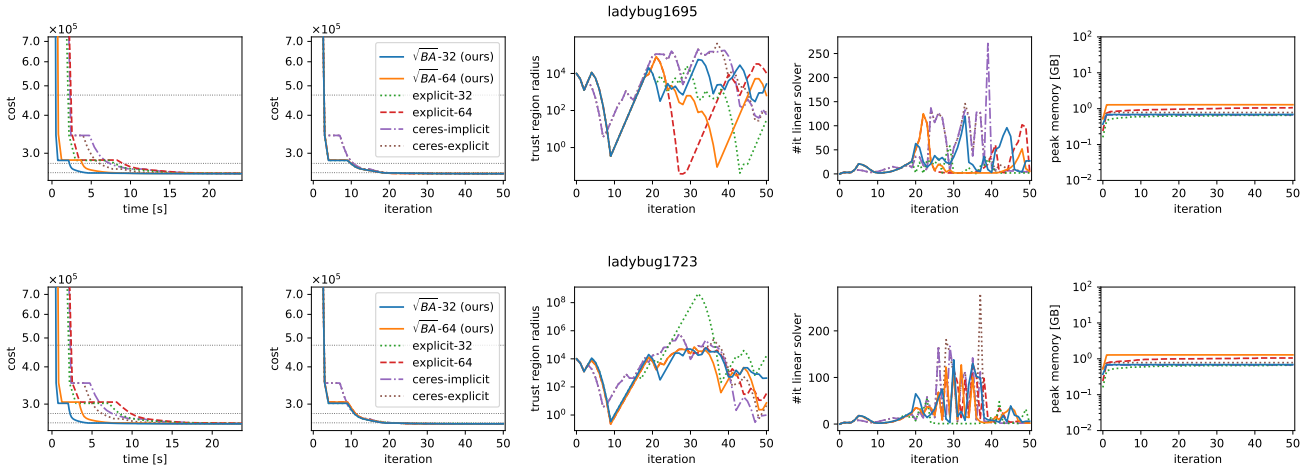



ladybug372
ladybug412
ladybug460
ladybug539
ladybug598
ladybug646
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

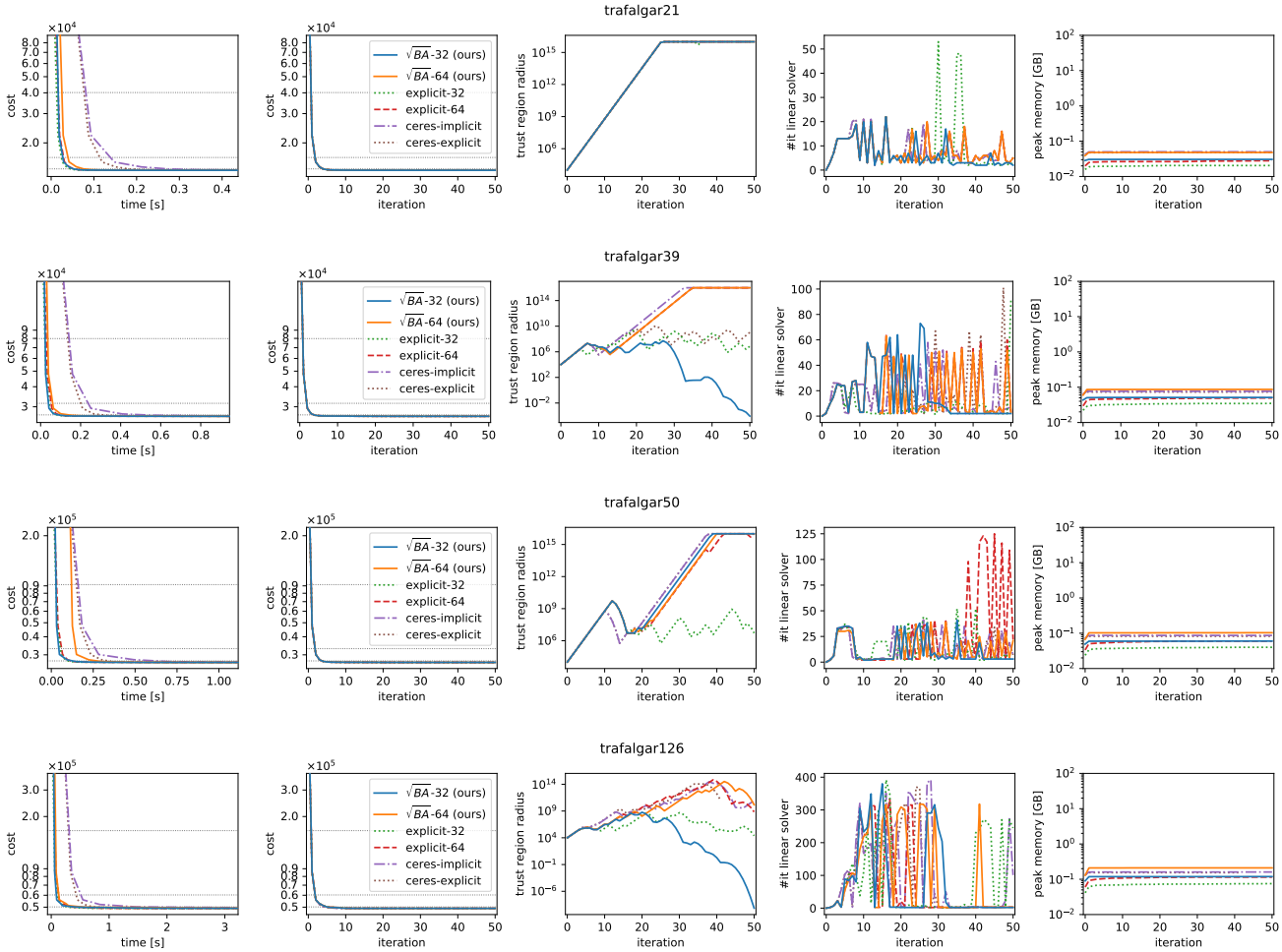

ladybug707
ladybug783
ladybug810
ladybug856
ladybug885
ladybug931
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

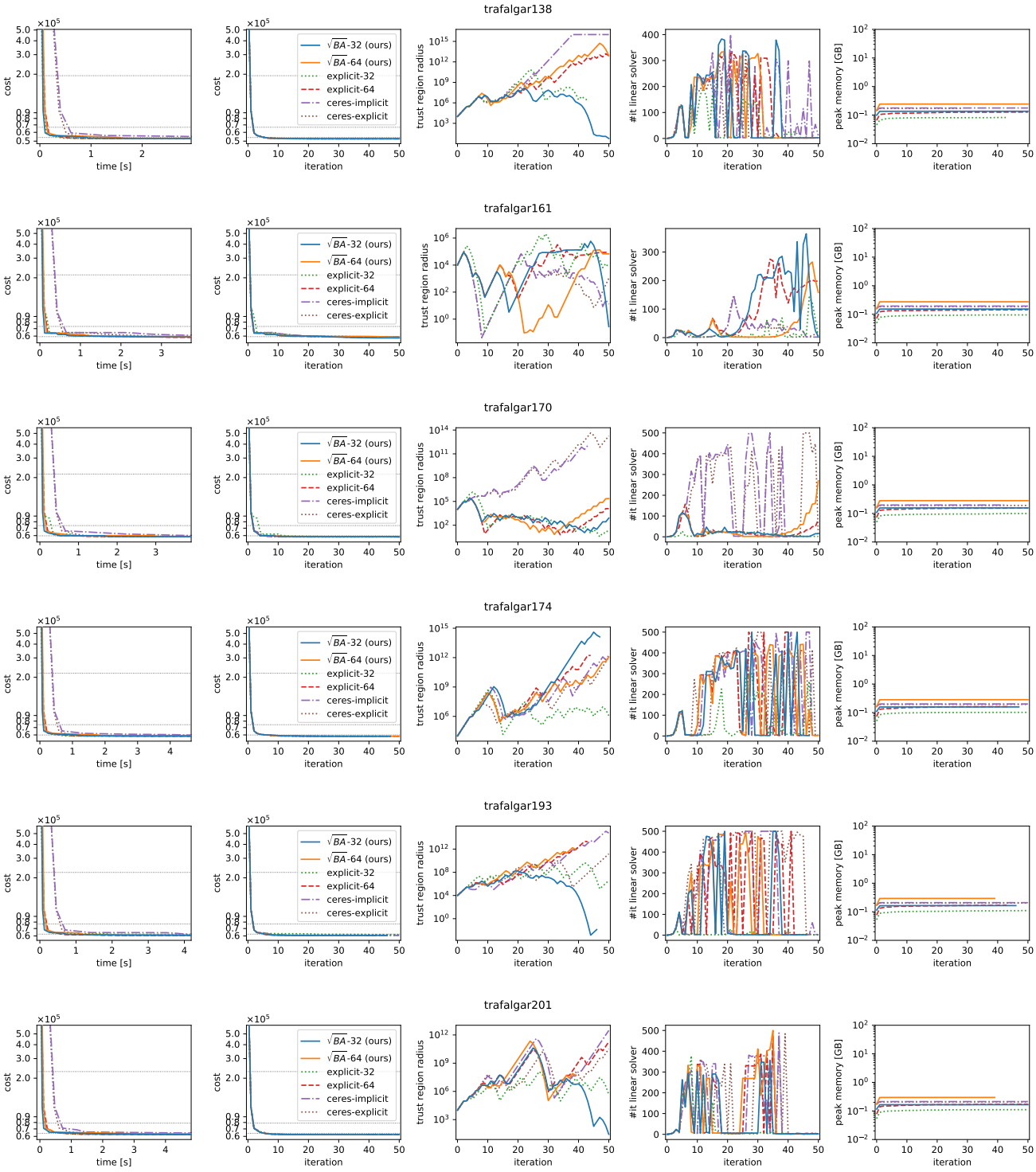

ladybug969
ladybug1064
ladybug1118
ladybug1152
ladybug1197
ladybug1235
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

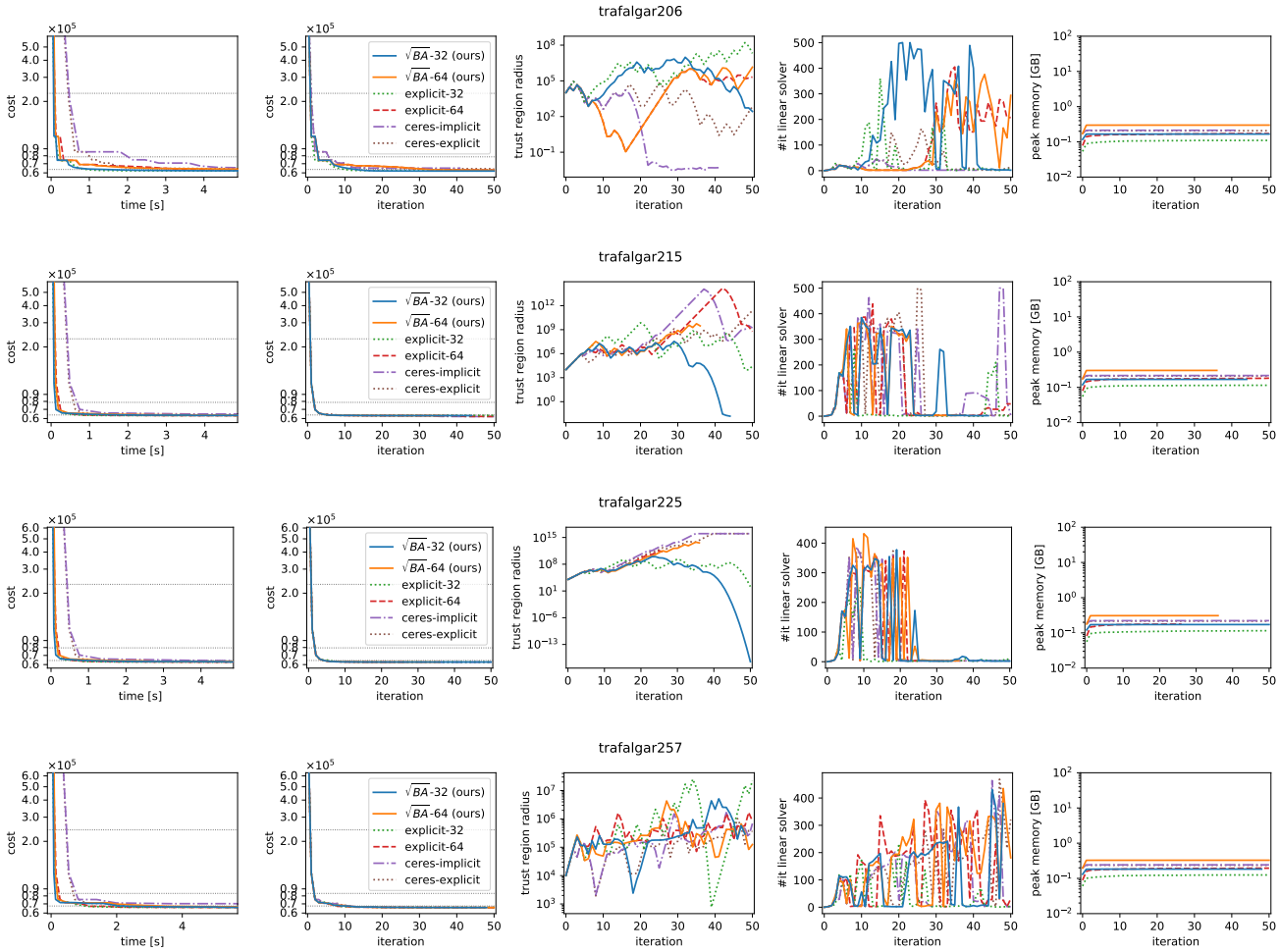

ladybug1266
ladybug1340
ladybug1469
ladybug1514
ladybug1587
ladybug1642
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit



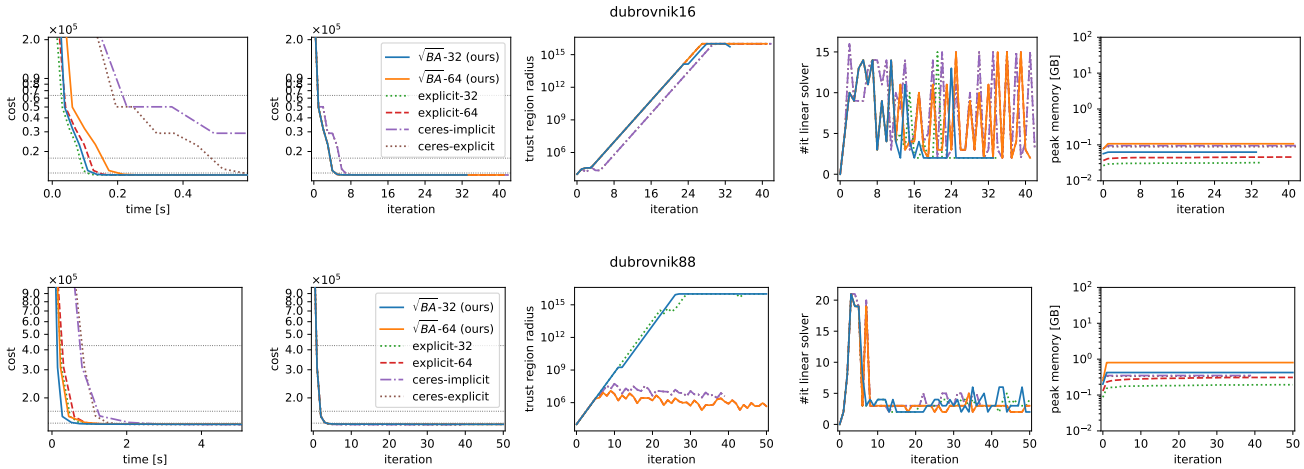
F.2. Trafalgar

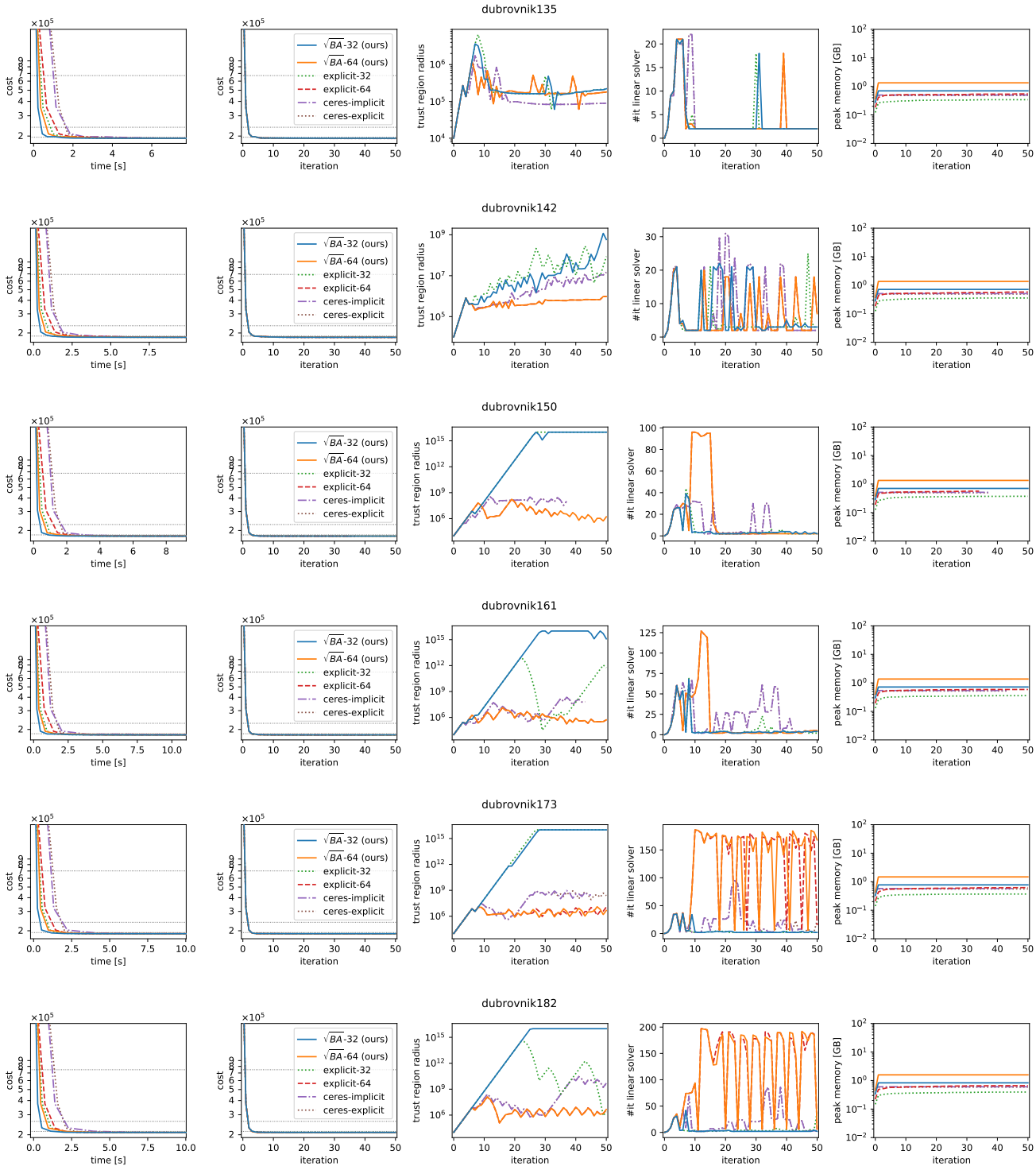


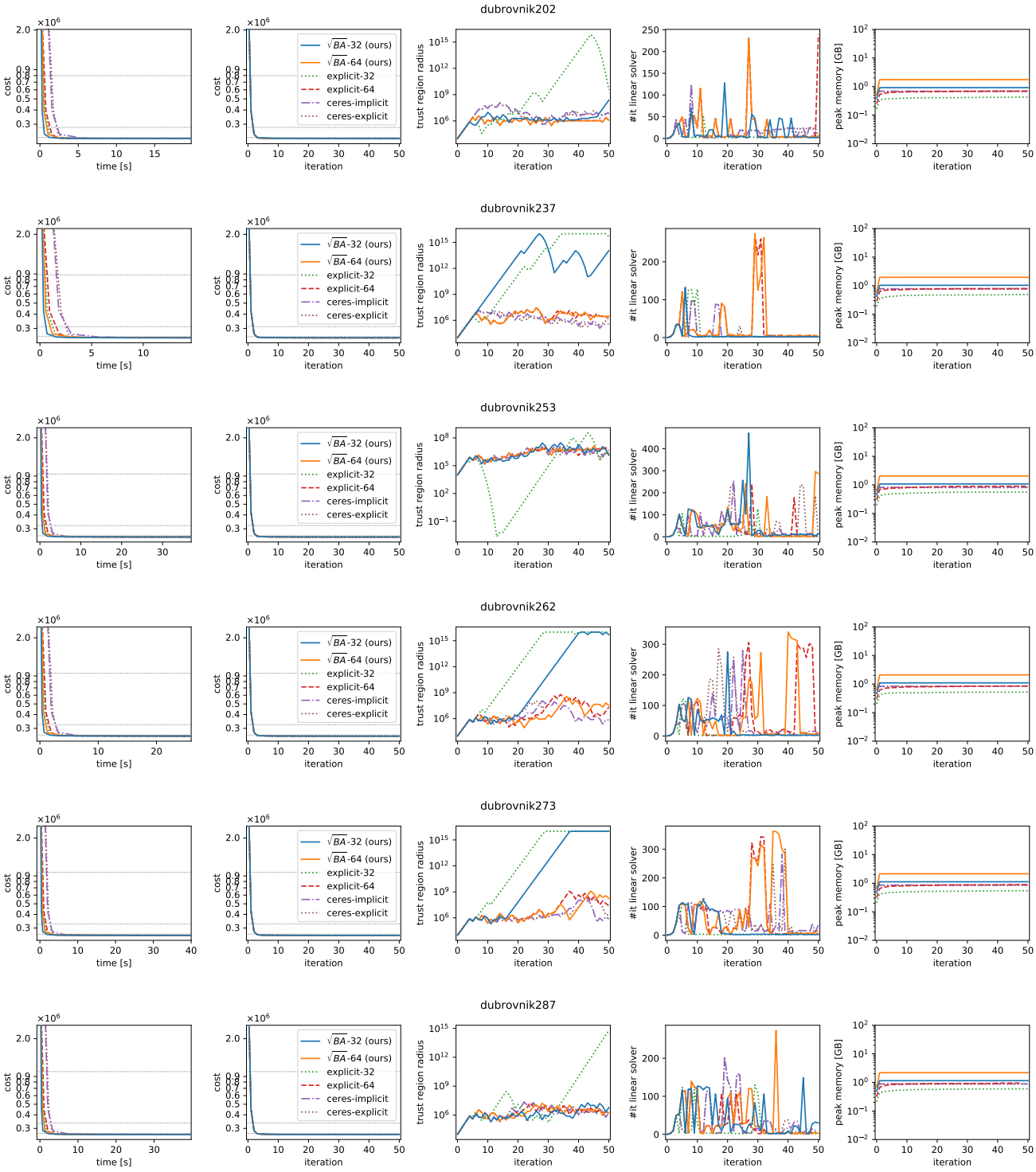

trafalgar138
trafalgar161
trafalgar170
trafalgar174
trafalgar193
trafalgar201
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

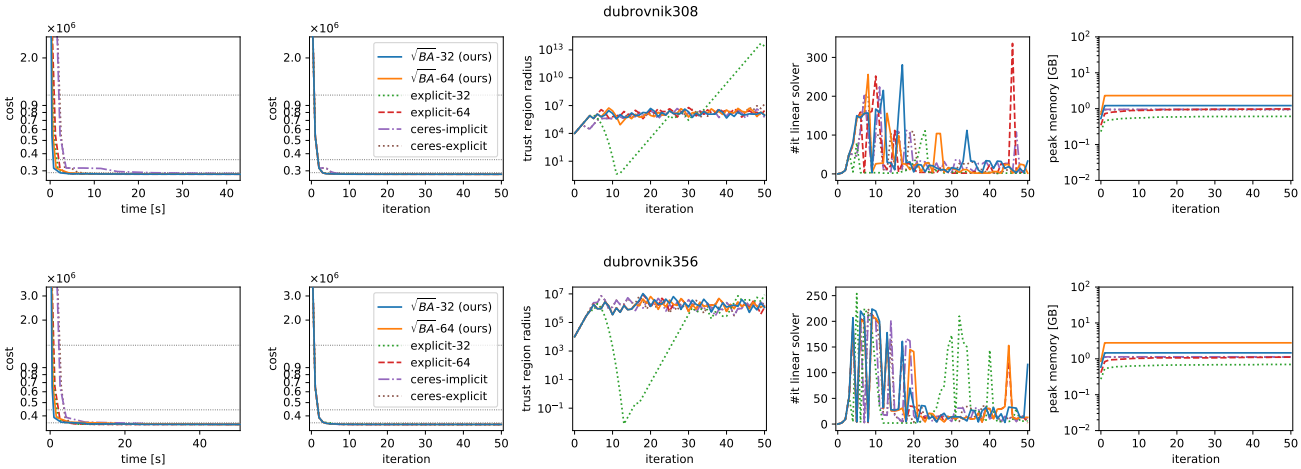


F.3. Dubrovnik

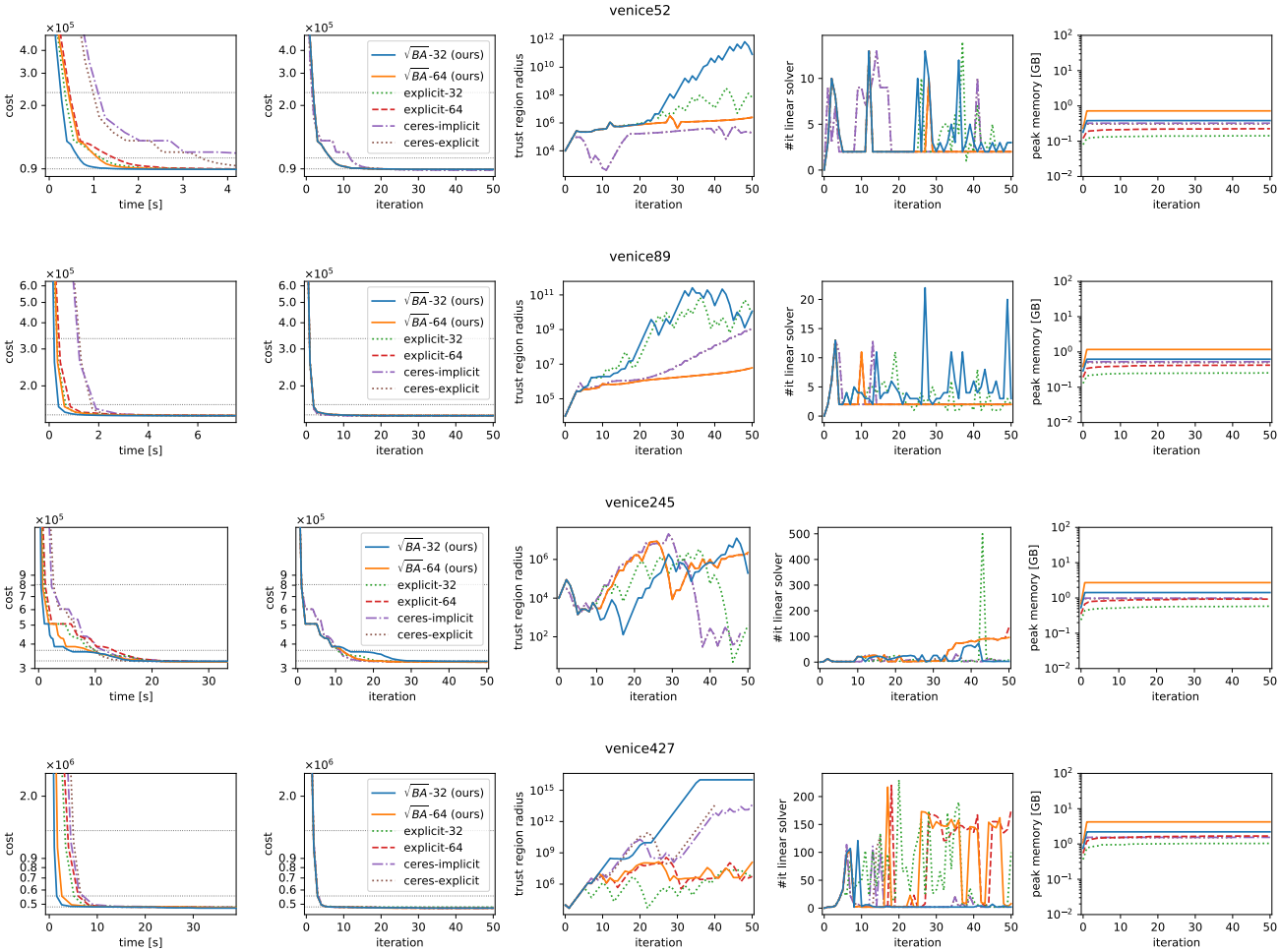


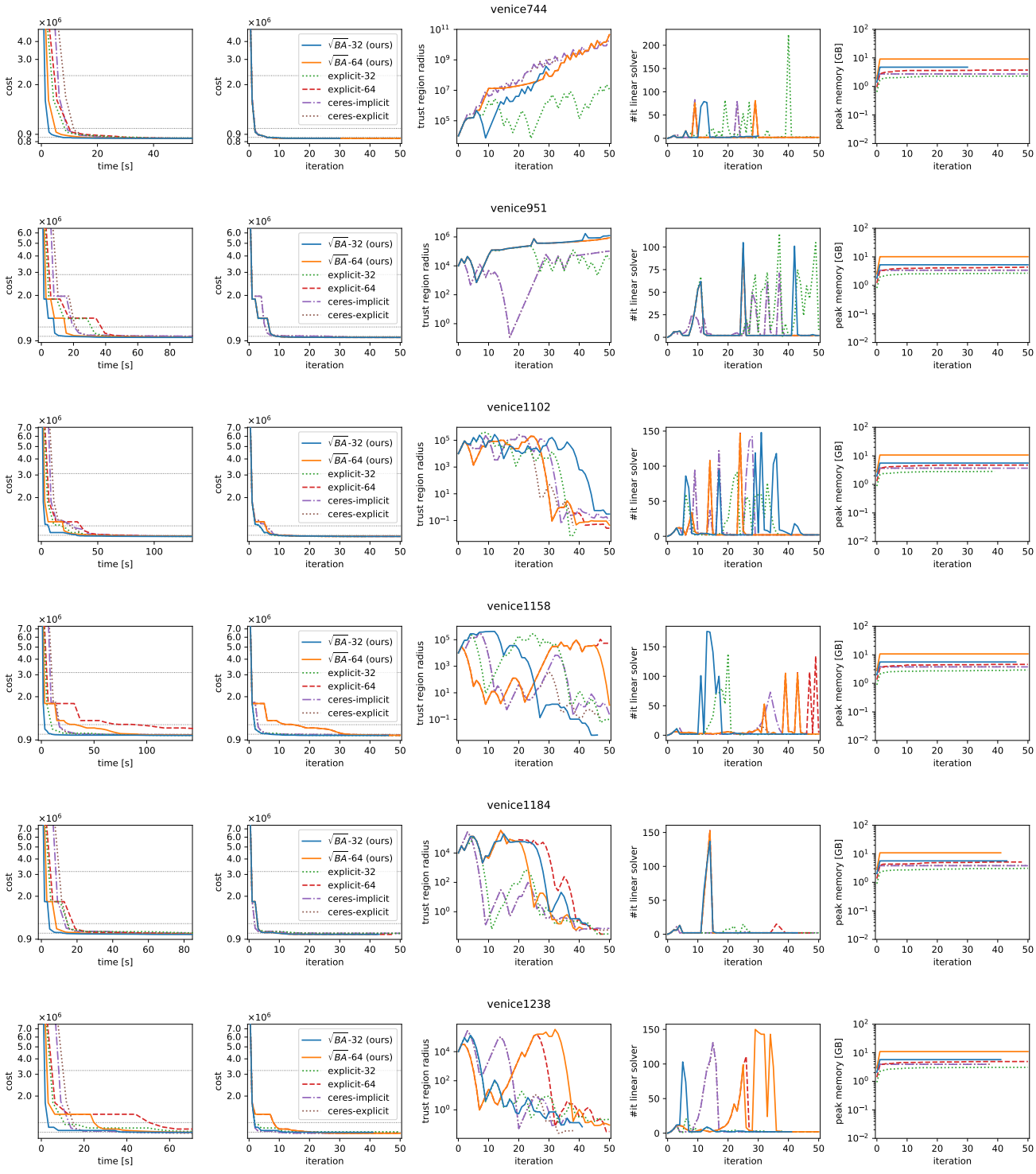

dubrovnik135
dubrovnik142
dubrovnik150
dubrovnik161
dubrovnik173
dubrovnik182
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

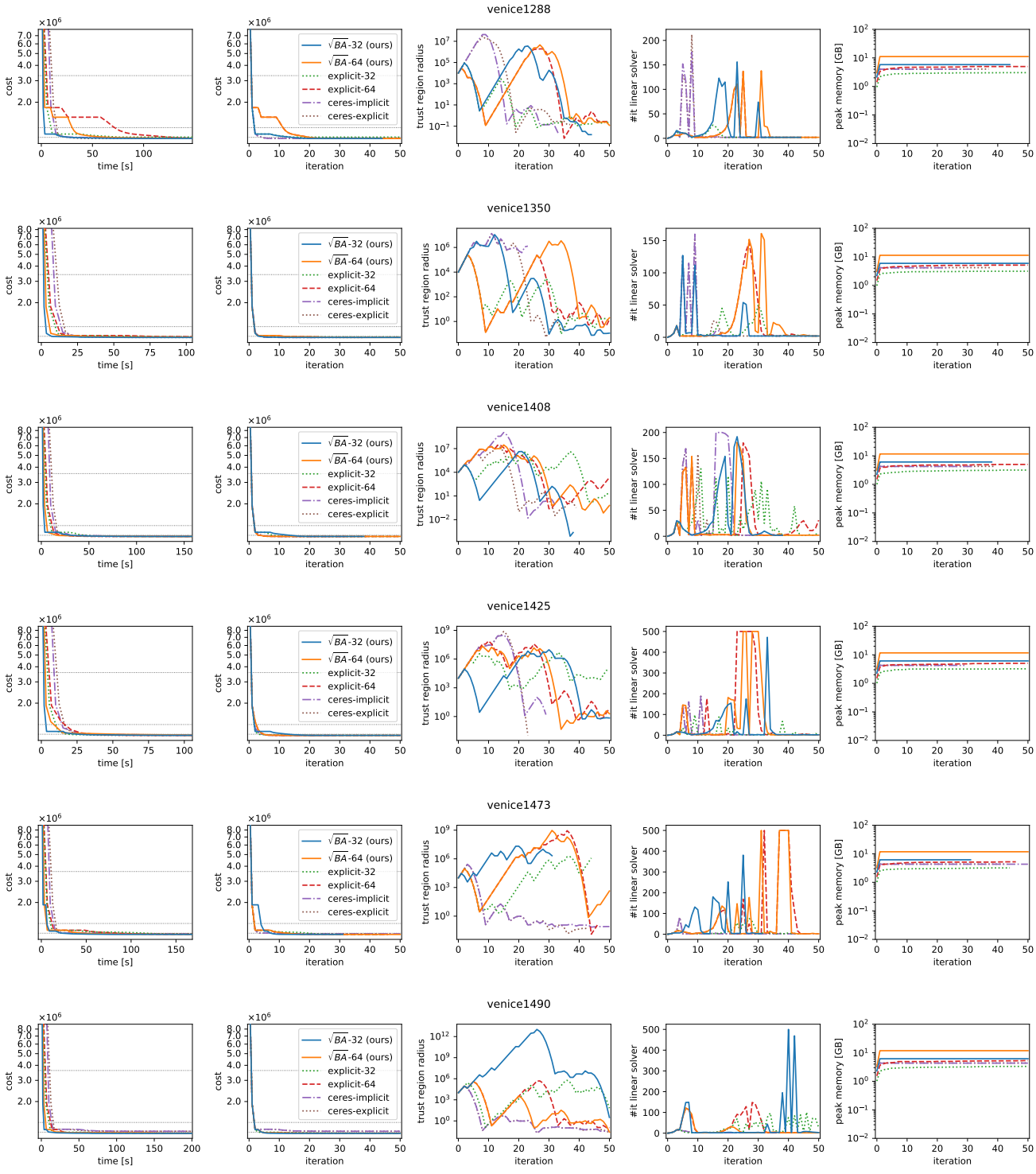

dubrovnik202
dubrovnik237
dubrovnik253
dubrovnik262
dubrovnik273
dubrovnik287
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

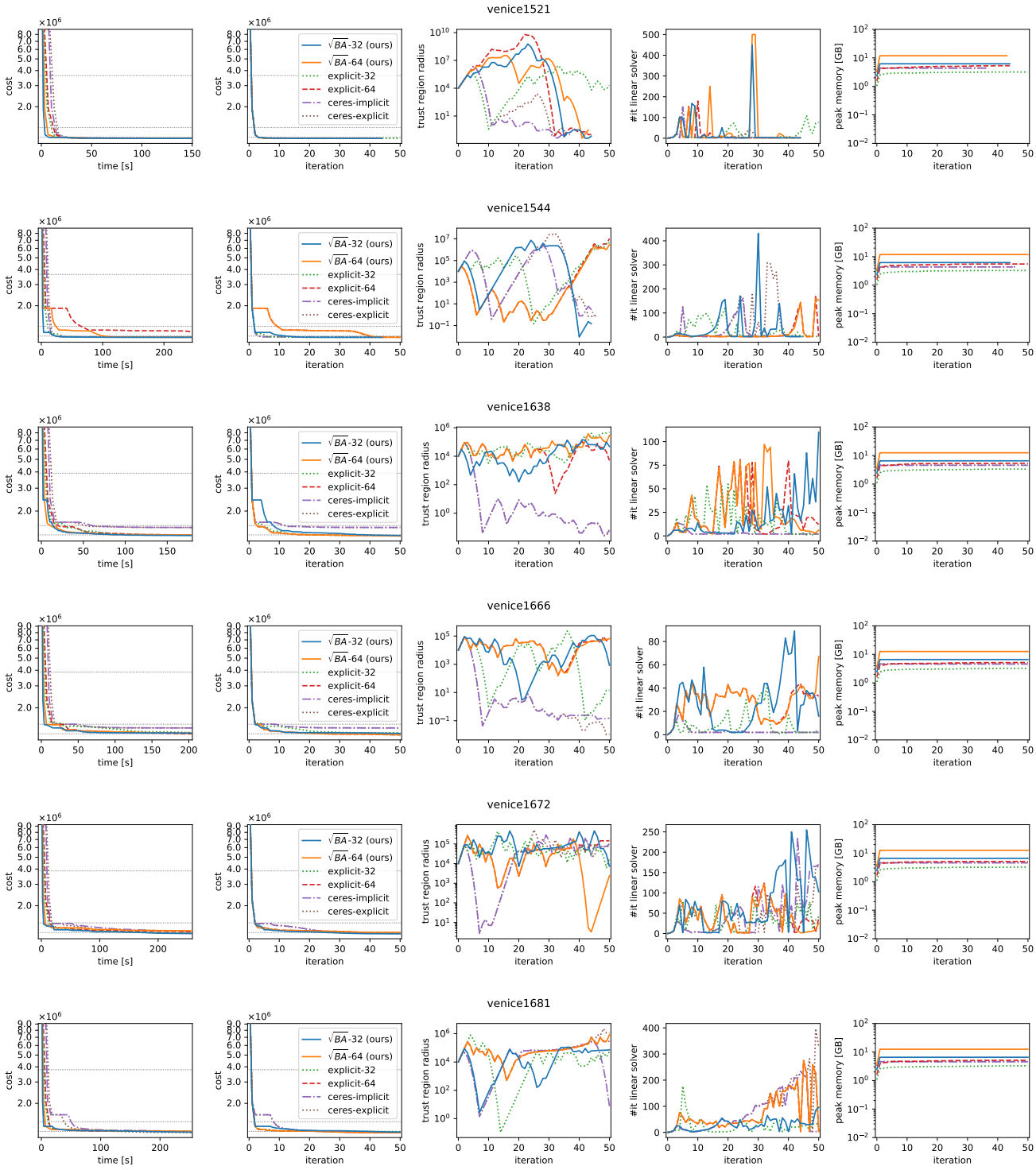

dubrovnik308
dubrovnik356
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

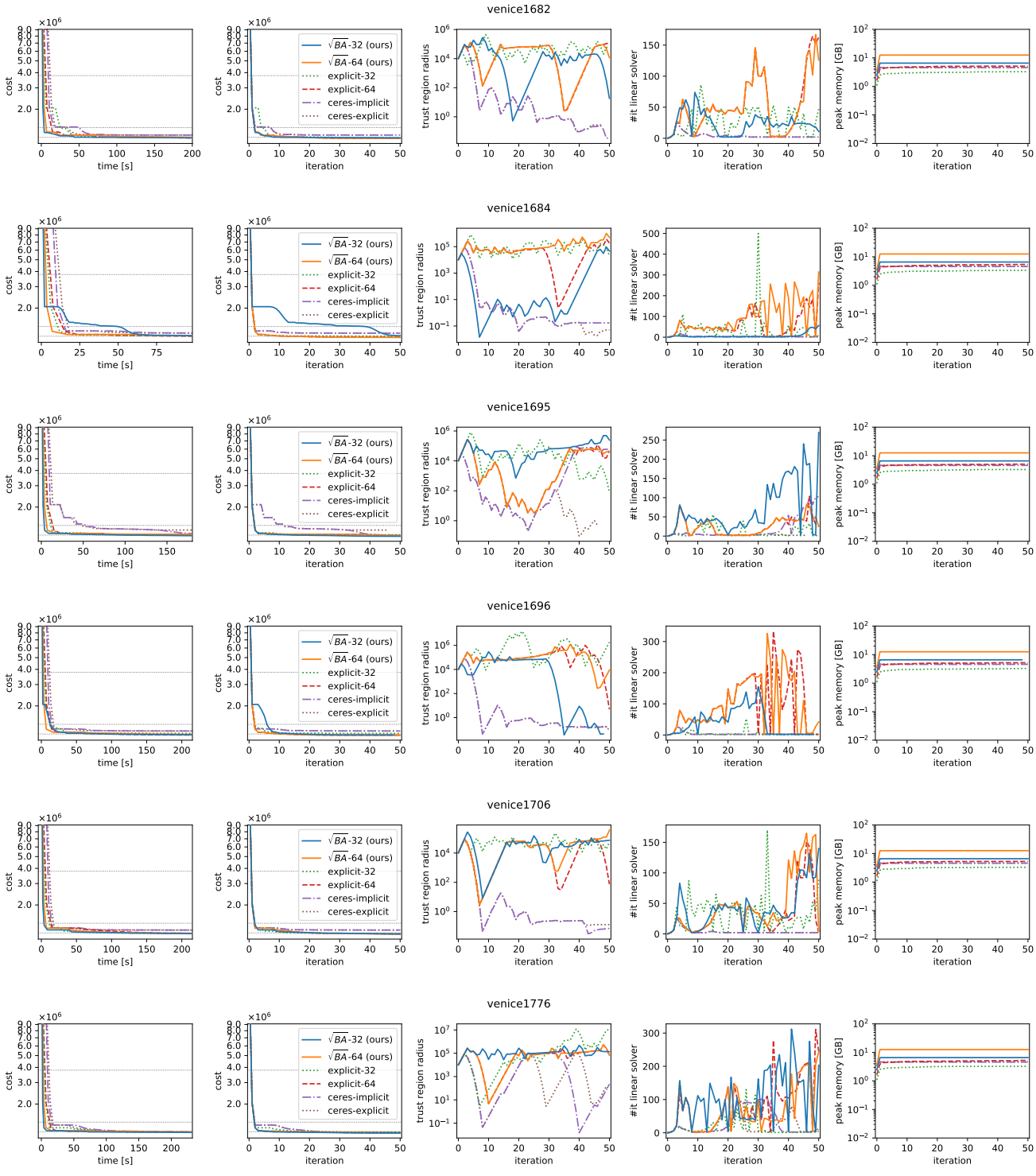
F.4. Venice

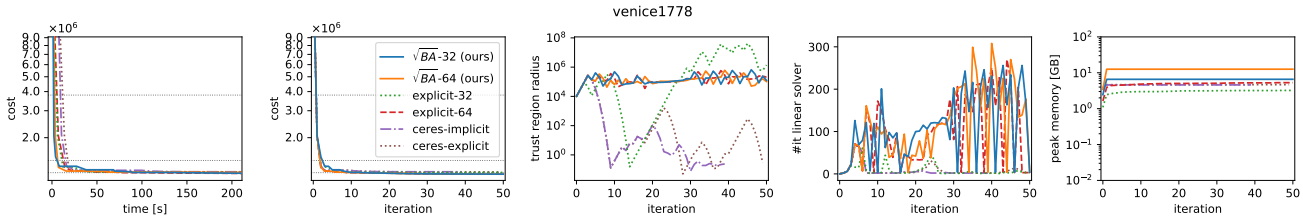

venice52
venice89
venice245
venice427
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit


venice744
venice951
venice1102
venice1158
venice1184
venice1238
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

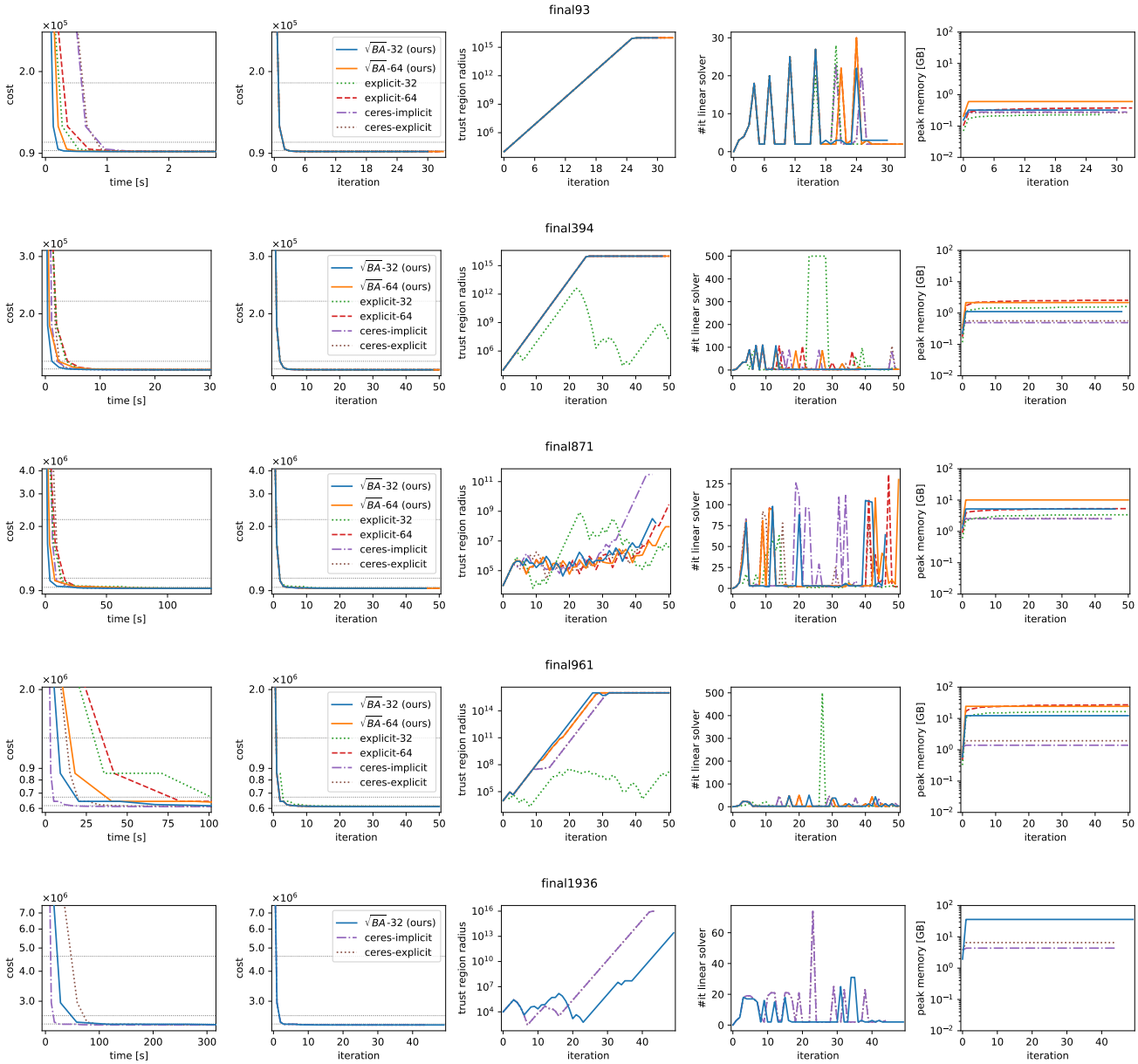

venice1288
venice1350
venice1408
venice1425
venice1473
venice1490
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

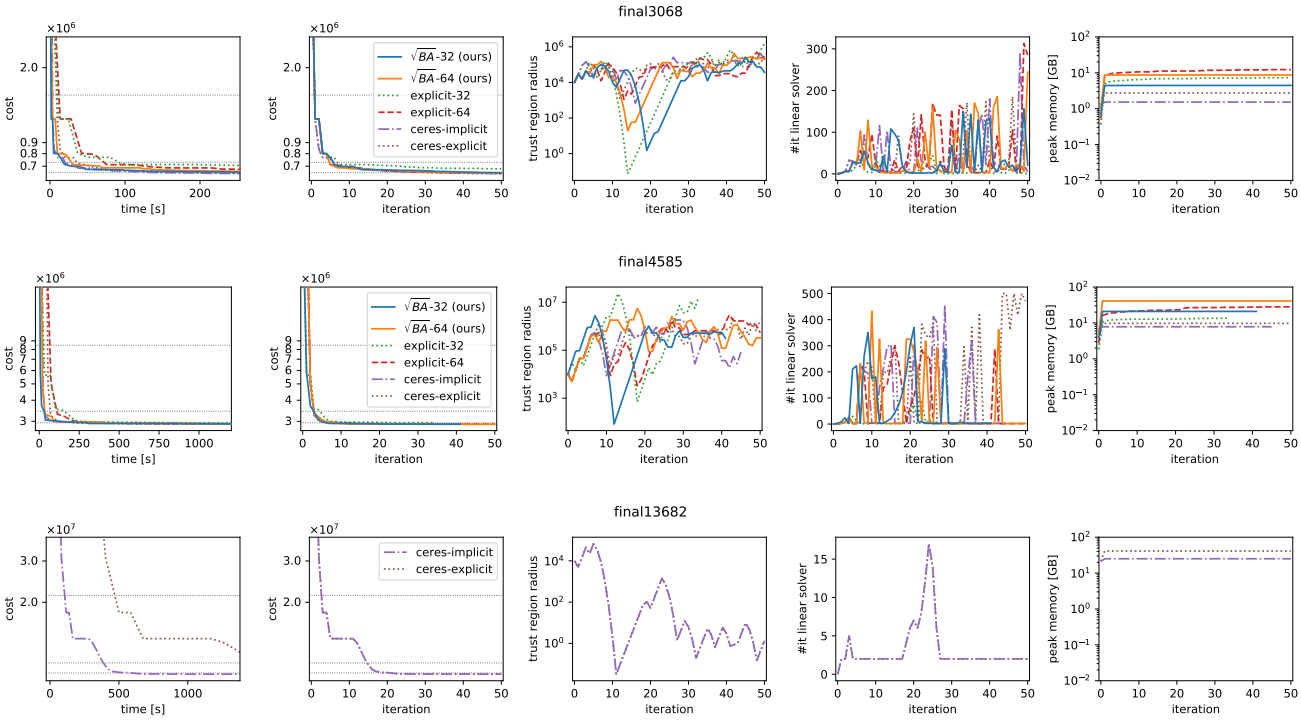

venice1521
venice1544
venice1638
venice1666
venice1672
venice1681
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit


venice1682
venice1684
venice1695
venice1696
venice1706
venice1776
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit


venice1778
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit

F.5. Final


final93
final394
final871
final961
final1936
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit


final3068
final4585
final13682
cost
time [s]
iteration
trust region radius
#it linear solver
peak memory [GB]
√BA-32 (ours)
√BA-64 (ours)
explicit-32
explicit-64
ceres-implicit
ceres-explicit