Fast Nonlinear Least Squares Optimization of Large-Scale Semi-Sparse Problems

EUROGRAPHICS 2020 / U. Assarsson and D. Panozzo
(Guest Editors)

Volume 39 (2020), Number 2

Fast Nonlinear Least Squares Optimization of Large-Scale Semi-Sparse Problems

M. Fratarcangeli[1,2] D. Bradley[2] A. Gruber[3] G. Zoss[2,3] T. Beeler[2]

[1]Chalmers University of Technology [2]DisneyResearch|Studios [3]ETH Zurich

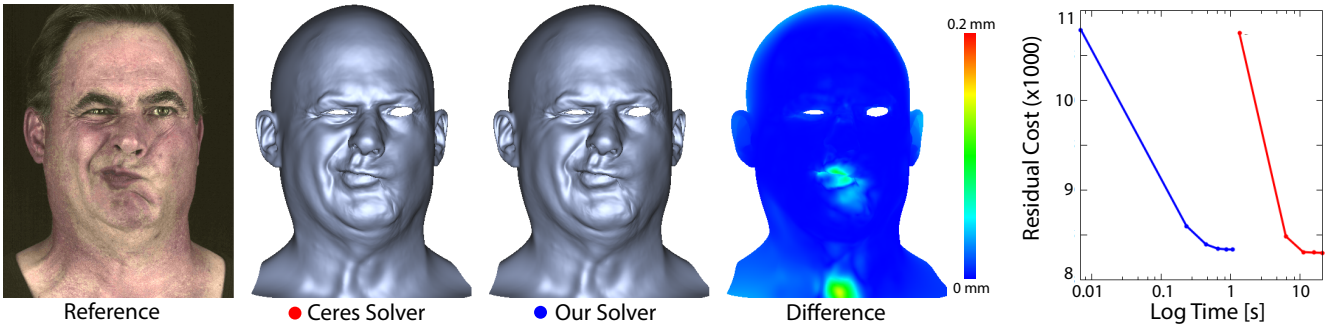


Figure 1: We propose a new method for solving nonlinear least squares problems that is highly parallel and scalable, allowing large-scale semi-sparse optimization on graphics hardware. Here we apply our solver to the problem of model-based facial capture, and compare to a standard Ceres solver implementation.

Abstract

Many problems in computer graphics and vision can be formulated as a nonlinear least squares optimization problem, for which numerous off-the-shelf solvers are readily available. Depending on the structure of the problem, however, existing solvers may be more or less suitable, and in some cases the solution comes at the cost of lengthy convergence times. One such case is semi-sparse optimization problems, emerging for example in localized facial performance reconstruction, where the nonlinear least squares problem can be composed of hundreds of thousands of cost functions, each one involving many of the optimization parameters. While such problems can be solved with existing solvers, the computation time can severely hinder the applicability of these methods. We introduce a novel iterative solver for nonlinear least squares optimization of large-scale semi-sparse problems. We use the nonlinear Levenberg-Marquardt method to locally linearize the problem in parallel, based on its first-order approximation. Then, we decompose the linear problem in small blocks, using the local Schur complement, leading to a more compact linear system without loss of information. The resulting system is dense but its size is small enough to be solved using a parallel direct method in a short amount of time. The main benefit we get by using such an approach is that the overall optimization process is entirely parallel and scalable, making it suitable to be mapped onto graphics hardware (GPU). By using our minimizer, results are obtained up to one order of magnitude faster than other existing solvers, without sacrificing the generality and the accuracy of the model. We provide a detailed analysis of our approach and validate our results with the application of performance-based facial capture using a recently-proposed anatomical local face deformation model.

CCS Concepts

• ***Computing methodologies*** → ***Massively parallel and high-performance simulations;*** *Animation;*

1. Introduction

Numerical optimization lies at the heart of many problems in computer science. Computer graphics is no exception, however most often the novelty lies in the formulation of the problem and the constraints rather than the optimization itself. This is largely because many generic solvers have already been proposed for dif-

ferent types of problems, and therefore organizing a problem to fit an existing solver nearly always represents the lion share of the novelty in our field. As an example, challenges like mesh processing, including deformation, parameterization and matching [ELaC19, SPSH*17, FBT*18, MHR*16], or for motion reconstruction and modeling [Sug11, ZNI*14, TZS*16, HLSO12, LGL*19], have been reduced to nonlinear least squares optimization problems and solved with the popular Gauss-Newton method [BV04], or its variant, Levenberg-Marquardt [Lev44, Mar63]. The issue with relying on off-the-shelf generic solvers, however, is that they may not be designed for the problem at hand.

In this work, we consider optimizing nonlinear, large-scale, semi-sparse least squares problems. By 'large-scale' we mean that the problem requires a significant number of interdependent residual functions (up to several hundreds of thousands), each one considering a large number of parameters (up to several hundreds). By 'semi-sparse' we mean that certain sub-blocks of the problem structure may appear dense, but the sub-blocks are coupled together only sparsely at the global scale. While existing optimizers, like the Ceres solver [AMO], are fully capable of finding an accurate solution to such problems, the convergence time can be quite slow. An alternative is to accept only an approximate solution in less time, however this trade-off may not be desirable. In this paper, we propose a new solver for nonlinear least squares optimization of large-scale semi-sparse problems which is both fast and accurate.

While the concepts behind our solver are generic in nature, we will limit our application and evaluation of the algorithm to one specific problem in computer graphics, that of facial performance capture using anatomical local face models as proposed by Wu et al. [WBGB16]. Facial capture has become the de-facto standard for obtaining realistic facial animation in film and game productions, and this particular model-based method has shown incredible promise in terms of accuracy, flexibility and ease-of-use. It suffers, however, from the slow computational time mentioned above. To further motivate our decision to demonstrate our new solver on this particular problem, the anatomical local model for facial capture has already been employed in the creation of digital characters for several recent blockbuster films[†]. In this problem domain, the deformation of a virtual face is modeled as a nonlinear least squares problem where the residual functions express both the anatomy of the face and the particular target face expression, including the bony configuration, the sliding of the skin and the wrinkle formation. The description of the problem requires a large number of interdependent residual functions, each one considering a large number of parameters. The residuals differ in complexity and type, making the problem non-homogeneous and difficult to decompose and parallelize. As a consequence, while several solvers are already available and able to find the solution to such a problem, the solving process is still time consuming even for short animation sequences. Just as an example, in our experiments a single solver iteration for one frame of animation required approximately 10 seconds of computation using Ceres on a 4-core CPU (8 threads) with a multi-threaded linear solver (Intel MKL). Given that high-quality facial reconstruction with this model typically requires up to 10 iterations per frame in practice, each single second of animation at 30 fps would require approximately 50 minutes of solve time (in addition to other computation not related to the solve). Thus, there is an obvious need for a fast nonlinear solver for this problem in order to make it tractable for high-throughput facial animation pipelines.

We use a standard and widely adopted technique for nonlinear least squares problems, namely the Levenberg-Marquadt algorithm [Lev44, Mar63], an iterative method known for its robustness and fast convergence speed [TMHF00, BBC*94]. Each nonlinear iteration, however, may be slow to compute depending on the number of residuals and parameters. The method, in fact, relies on the computation of the partial derivatives of each multivariate residual function with respect to each parameter, and their inner products. These are used to assemble the Jacobian matrix, which is employed to linearize the residuals and produce a linear system, the so-called *normal equations*. The normal equations are solved to improve the estimate of the unknown parameters, update the value of the residual functions and iterate the process again until the residual error is minimized.

Our minimizer leverages on the speed of modern graphics processors (GPUs) to accelerate the computation of the inner linear iterations of the Levenberg-Marquardt solver. We introduce two main novelties to this purpose. First, we describe a compact data structure to store all the partial derivatives. Its data layout is designed to optimize the parallel computation of the Jacobian on the GPU while using a small memory footprint. Second, we introduce a domain decomposition technique to quickly solve the linear normal equations by partitioning the numerical system into many small-and-local problems. These can be solved directly in parallel and then reassembled together leading to an accurate global solution.

The main benefit of such an approach is that the overall optimization process is entirely parallel and scalable, making it suitable to be mapped onto graphics hardware, without sacrificing the generality of the model. The result is a parallel GPU-based minimizer for solving large-scale nonlinear least squares problems. Applied in the context of facial capture with an anatomical local model, our method can achieve accurate facial reconstructions in a short amount of time, for example one solve can be computed in approximately one second on the GPU. So, each second of animation composed by 30 frames at 10 solver iterations per frame, is reduced to 300 seconds of solve time (i.e. 5 minutes) instead of 50 minutes using Ceres, an order of magnitude speedup. Even though we validate our results on a single (yet challenging) application, our solver is a significant step towards a unified way to solve this class of optimization problems.

2. Related Work

Nonlinear optimization problems are ubiquitous in computer graphics, vision and robotics. At their core, the vast majority are least-squares problems. Generally speaking, given a nonlinear deformation energy $f(x)$, the problem is to find the displacement δ such that $(x+\delta)$ minimizes $f(x)$, while satisfying some given modeling constraints.

When each residual function depends on a small number of parameters, it is possible to reach a solution in real-time and achieve

[†] http://studios.disneyresearch.com/anyma

interesting applications, e.g., [TZS*16, MFZ*17]. The problem may also be sparse and defined on a regular domain, like an image, a grid, or a mesh, i.e., they have a local nature. In these cases, it is possible to write fast, dedicated GPU solvers, which quickly solve the particular problem at hand but may be difficult to generalize, e.g., in fluid simulation [LMAS16, WTYH18, CZY17], and shape reconstruction [DNZ*17]. If the topology of the system does not change, then the solver can even be automatically created by using a domain specific language, such as OptLang [DMZ*17].

To save memory, many GPU-based techniques use a *matrix-free* approach, e.g., [ZNI*14, WZN*14, DMZ*17], where the first derivatives of the residual functions are computed when normal equations are evaluated. Each partial derivative is recomputed every time it is needed, but such redundancy is hidden by the massive throughput of arithmetic-logic operations performed by the GPU.

In other domains, e.g., [HLSO12, WBGB16, LGL*19], the nature of the problem is global and the residual functions may affect the whole domain. Furthermore, a residual function may be rather complex and require a significant amount of data to be fetched by the video memory with the consequent delay. In these cases, matrix-free approaches become too slow. To mitigate this problem, partial derivatives can be precomputed and stored, and then used to assemble the normal equations. We apply this latter approach and present a data structure for the efficient storage and retrieval of the partial derivatives data in Sec. 4.1.

Solving the normal equations can be done by using relaxation methods, which are widely used because they are fast, simple and easy to implement on the GPU [FTP16, Wan15, WY16], but may exhibit slow convergence for poorly conditioned problems. In contrast, direct methods can achieve greater accuracy, but they scale poorly with problem size.

In our solver, we strive to achieve the same speed as relaxation methods together with the accuracy of direct methods. To this end, we use the *local Schur complement*, a technique to solve a Symmetric Positive Definite linear system of equations by reducing its size without losing information [Saa03]. It may be used for both reducing the size of large problems which do not fit into memory, as in [LMAS16], and to speed up the computation of the solver, as in [MEM*19, CZY17]. Once the size of the matrix is reduced, we simply apply a parallel Cholesky direct solver and find the global solution of the system (Sec. 4.2).

To be effective, however, the input matrix must satisfy certain conditions, in particular the majority of the coefficients must be located around the diagonal of the matrix. This can be achieved by partitioning the matrix, and assigning sequentially increasing indices to the rows in the same partition. Ideally, the partitions should have the same size for good load balancing, and be as disconnected as possible to reduce the number of coefficients outside the diagonal blocks. This is a NP-hard problem, and in most of the works in computer graphics, it is tackled by dividing the domain of the problem according to the symmetry or some other geometric feature e.g., [HLSO12, LGL*19]. In order to be generic, we abstract entirely from the geometry of the problem and perform an algebraic spectral partitioning of the matrix, as depicted in Sec. 4.3.

To our knowledge, we propose the first method for nonlinear least squares optimization of large-scale semi-sparse problems that is both parallel and scalable, making it suitable to be mapped to graphics hardware for very fast solve times.

3. Background

We begin by laying down some background information before describing our contributions in Sec. 4. We start by describing the Anatomical Local Face Model (Sec. 3.1) which forms the example to demonstrate our fast optimization, followed by a brief review of the Levenberg-Marquardt algorithm (Sec. 3.2) and dual numbers (Sec. 3.3), which are required by our method.

3.1. Anatomical Local Face Model

We will demonstrate our fast optimization method in the context of 3D facial performance capture. The task of performance-based facial animation is often posed as a model-fitting problem, where a deformable face model is used as a prior for the actor's facial movements. The animation is then reconstructed by finding the optimal model parameters that best describe the performance. Historically, these priors have been rather simple linear models such as blend-shape rigs [LAR*14], which are fast to solve but tend to lack expressivity and accuracy. Recently a new model has been proposed [WBGB16], which excels at both accuracy and flexibility but comes at the cost of optimization complexity. The model decomposes the face into many small local patches, overlapping and not necessarily aligned into a grid, each with their own set of rigid motion and non-rigid deformation parameters, coupled with global parameters that define the rigid transformation of the underlying bone structures. As the model can represent local skin sliding over bones, it is generally referred to as an *anatomical local model (ALM)*. With an ALM model as the actor prior, each frame of the performance is reconstructed by jointly optimizing all the local patch parameters together with the global rigid bone motion, given a set of constraints from input video. This is a large nonlinear optimization problem with many parameters contributing to the cost functions, and is thus an ideal candidate for our novel fast optimization procedure. In the following we provide a brief overview of the model parameters and optimization problem, and we refer to Wu et al. [WBGB16] for more details.

Model Description. The ALM model has two main components - a local patch deformation subspace and underlying bone structure (as illustrated in Fig. 2).

The local shape $X_i(t)$ of a patch i at time t is defined as

$$X_i(t) = M_i(t)\left(U_i + \sum_{k=1}^{K} \alpha_i^k(t) D_i^k\right), \tag{1}$$

where $M_i(t)$ is the rigid motion of the patch, $\{\alpha_i^1(t), \ldots, \alpha_i^K(t)\}$ are the coefficients of the K deformation components $\{D_i\}$, and U_i is the neutral patch shape. Considering all patches in the face, the resulting large set of parameters can lead to an ill-posed fitting problem, and thus the skull and an anatomical subspace is introduced. The anatomical subspace is defined as a skin thickness subspace, where the position $\tilde{\mathbf{x}}_\nu(t)$ of vertex ν can be predicted at time t as

$$\tilde{\mathbf{x}}_\nu(t) = M_s(t)\left(\tilde{\mathbf{b}}_\nu(t) - d_\nu(t)\tilde{\mathbf{n}}_\nu(t)\right), \tag{2}$$

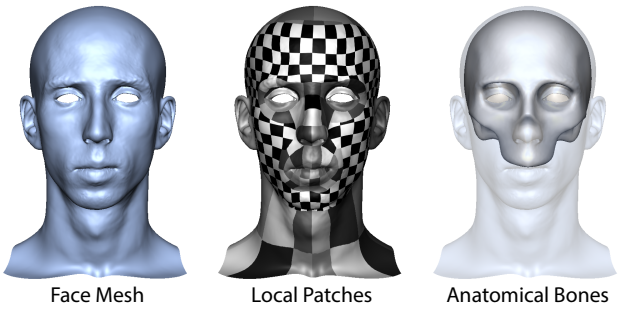


Figure 2: The anatomical local face model (ALM) divides the face into overlapping patches and optimizes for patch reconstructions using local deformation subspaces and anatomical skin thickness constraints, together with the global rigid bone motion.

where $M_s(t)$ is the rigid motion of the skull, $\tilde{\mathbf{b}}_\mathbf{v}(t)$ is the bone point subspace, $\tilde{\mathbf{n}}_\mathbf{v}(t)$ is the bone normal subspace, and $d_\mathbf{v}(t)$ is the skin thickness subspace. A thorough description of the ALM model is beyond the scope of this work, but we refer to Wu et al. [WBGB16] for detailed definitions.

Patch Optimization. Reconstructing the face is performed by estimating the model parameters that best describe the observed motion under a certain set of constraints through optimization. The unknowns to solve for include the rigid local patch motion $\{M_i\}$, the local patch deformation coefficients $\{\boldsymbol{\alpha}_\mathbf{i}\}$ and the rigid motion of the anatomical bone M_s. The parameters are optimized by minimizing the following energy

$$\underset{\{M_i\},\{\boldsymbol{\alpha}_\mathbf{i}\},M_s}{\text{minimize}} \quad E_M(t)+E_O(t)+E_A(t), \tag{3}$$

where E_M is a data term that usually describes observed 2D motion in video, E_O is a patch overlap constraint that aims to make patches conform to their neighbors in overlapping regions, and E_A is an anatomical constraint that preserves correct dynamic skin thickness as skin slides over the bones. This optimization problem will be the target of our proposed new solver. Since the patches are solved jointly, there can be hundreds of parameters to optimize, and many of them will contribute to each of the cost functions. Even though the problem structure is dense at a patch level, it is globally semi-sparse due to the sparse patch overlap constraint, making it ideally suited for our algorithm.

Global Patch Blending. For completeness sake, it is worth noting that solving for the model parameters will provide a reconstruction of the skull motion and individual patch deformations, but the final manifold face mesh is recovered through blending the individual patch reconstructions with soft constraints in overlapping regions, as described in the original ALM algorithm [WBGB16].

3.2. Levenberg-Marquardt Algorithm

Solving the anatomical local model (Eq. 3) is a challenging nonlinear optimization problem. In the following we briefly recall the Gauss-Newton (GN) method for nonlinear optimization, and then provide a short description of its extension, the Levenberg-Marquardt (LM) algorithm, which interpolates between Gauss-Newton and gradient descent, and forms the core of our optimization approach.

Consider

$$f(x)=\frac{1}{2}\sum_{k=1}^{m} r_k(x)^2, \tag{4}$$

where $x\in\mathbb{R}^n$, $x=(x_1,x_2,\ldots,x_n)$ and the nonlinear functions $r_k(x)\in\mathbb{R}^n\rightarrow\mathbb{R}$ are called *residuals*. We are interested in finding x^* minimizing $f(x)$. Often, $m\gg n$, i.e., the problem is *over-constrained*. To simplify the notation, $f(x)$ can be represented by the residual vector $r=(r_1,r_2,\cdots,r_m)$ and be rewritten as $f(x)=\frac{1}{2}\|r\|^2$. In the general case, when $r_k(x)$ are nonlinear, the problem is intractable so we set to find an approximate solution. One way to find x^* is to set the gradient $\nabla f(x)$ to zero. Truncating its Taylor series expansion to the first order

$$\nabla f(x)=\nabla f(x_0)+(x-x_0)^{\mathrm{T}}\nabla^2 f(x_0)+\mathcal{O}\left(\|x\|^2\right), \tag{5}$$

x^* can be found by using the following iteration scheme:

$$x_{i+1}=x_i-\left(\nabla^2 f(x_i)\right)^{-1}\cdot\nabla f(x_i). \tag{6}$$

$\nabla f(x_i)$ and $\nabla^2 f(x_i)$, i.e. the first- and the second-order gradients, are:

$$\nabla f(x)=\sum_{k=1}^{m} r_k(x)\nabla r_k(x)=J^{\mathrm{T}}(x)\,r(x) \tag{7}$$

$$\nabla^2 f(x)=J^{\mathrm{T}}(x)J(x)+\sum_{k=1}^{m} r_k(x)\nabla^2 r_k(x), \tag{8}$$

where $J\in\mathbb{R}^{m\times n}$ is the Jacobian matrix of the partial derivatives of r in x, defined as:

$$J=\frac{\partial r_k}{\partial x_j},\ 1\le k\le m,\ 1\le j\le n. \tag{9}$$

If $r_k(x)$ is small enough, or locally linear (so that $\nabla^2 r_k(x)$ will be small), Eq. 8 can be approximated as

$$\nabla^2 f(x)=J^{\mathrm{T}}(x)J(x). \tag{10}$$

By plugging Eq. 10 and Eq. 7 in Eq. 6, we obtain the so-called *normal equations*:

$$J^{\mathrm{T}}(x_i)J(x_i)(x_{i+1}-x_i)=-J^{\mathrm{T}}(x_i)\,r(x_i). \tag{11}$$

The normal equations are a linear system of size $n\times n$. Solving it for $\delta=x_{i+1}-x_i$ allows to find $x_{i+1}=x_i+\delta$, update J and r, and reiterate the process until $J^{\mathrm{T}}r$ is very small, i.e., $x_i\approx x^*$. This is the Gauss-Newton method (GN), which is known for its fast convergence even though it is sensitive to the starting point x_0 [BBC*94]. The gradient descent method behaves exactly in the opposite way: it has a relatively slow convergence but it is less sensitive to the starting point. The core idea of LM is to blend between GN and gradient descent leveraging the strength of both. In particular, the normal equations for LM are written as:

$$\left(J^{\mathrm{T}}J+\lambda I\right)\delta=-J^{\mathrm{T}}r, \tag{12}$$

Algorithm 1 Levenberg-Marquardt algorithm [Lev44, Mar63]

1: Compute $f = \frac{1}{2}\|r\|^2$
2: $\lambda = 10^{-3}$
3: **for** $k = 1,\dots,m$ **do**
4: **for** $i = 1,\dots,n$ **do**
5: Compute $j_{ki} = \frac{\partial r_k}{\partial x_i}$
6: Assemble $J^{\mathrm{T}}J$ and $J^{\mathrm{T}}r$
7: Solve $\left(J^{\mathrm{T}}J + \lambda I\right)\delta = -J^{\mathrm{T}}r$ ▷ Linear step
8: Update $x_{i+1} = x_i + \delta$
9: Compute $f_{new} = \frac{1}{2}\|r\|^2$
10: **if** $f_{new} > f$ **then** ▷ Step rejected
11: $\lambda = 10\lambda$
12: **go to** 7
13: **else** ▷ Step accepted
14: $x_i \leftarrow x_{i+1}$
15: $\lambda = 10^{-3}$
16: $f = f_{new}$
17: **go to** 3

where I is the identity matrix and λ is a tunable scalable parameter that initially has a low value (e.g., 10^{-3}) and it may change at the beginning of the iteration. If the error from one iteration to the following decreases, λ is kept constant and the convergence speed is similar to GN. If the error increases, then λ is increased as well, and the convergence slows down similarly to gradient descent until the error decreases again.

Overall, the LM algorithm can be summarized in Alg. 1. The most computationally expensive steps are the nested loop in step 2-5, computing the coefficients $J^{\mathrm{T}}J$ and $J^{\mathrm{T}}r$ in step 5, and solving the linear system in step 6. In the next section, we describe our main contributions to improve this optimization, a data structure to store and access j_{ij} (Sec. 4.1), and a parallel linear solver to quickly solve normal equations without loss of accuracy (Sec. 4.2).

3.3. Dual Numbers

We use automatic forward differentiation to find the coefficients of the Jacobian matrix (Eq. 9). In particular, we use multi-dimensional *dual numbers* [GW08].

A dual number is composed by the sum of a real component x with the infinitesimal component v, such that $v^2 = 0$. Writing a residual function $r(x)$ as a function of $x+v$ leads to a simple way to compute the exact derivative $\partial r/\partial x$. If $r(x)$ is differentiable, in fact, then its Taylor expansion in $(x+v)$ is:

$$r(x+v) = r(x) + \frac{\partial r}{\partial x}v + \frac{\partial^2 r}{\partial x^2}\frac{v^2}{2} + \frac{\partial^3 r}{\partial x^3}\frac{v^3}{6} + \dots \tag{13}$$

$$= r(x) + \frac{\partial r}{\partial x}v. \tag{14}$$

So, when r is evaluated in $(x+v)$, the coefficient of v is $\partial r/\partial x$. This simple, yet powerful, idea can be extended to a multivariate function $f(z) : \mathbb{R}^n \to \mathbb{R}^m$, where $z_i = x_i + v_i, \forall i = 1,\dots,n$. In this case, the function can be written as:

$$f(z_1, z_2, \dots, z_n) = f(x_1, x_2, \dots, x_n) + \sum \frac{\partial f}{\partial z_i}v_i. \tag{15}$$

By choosing v_i as the i-th standard basis vector in $\mathbb{R}^n$, we can extract the coefficients of the Jacobian matrix from the vector v_i.

4. Fast Parallel Linear Solver

The matrix $J^{\mathrm{T}}J$ is symmetric positive definite, and the normal equations $J^{\mathrm{T}}J\,\delta = -J^{\mathrm{T}}r$ form a large sparse linear system. In application contexts where an approximate solution is acceptable, iterative linear solvers are widely used to quickly find an approximate solution in a short amount of time, e.g., in interactive physics based animation [FTP16, Wan15, WY16]. In our case, however, a high degree of accuracy is required to obtain realistic results. This could be achieved by increasing the number of iterations but it would lead to an unacceptable loss of performance. Direct solvers, on the other hand, can find an accurate solution in one single step, but this is in general computationally demanding and slow to compute. One of the main factors affecting the performance of direct solvers is the size of the matrix.

In this section, we introduce our *divide-et-impera* technique to partition the linear system formed by the normal equations (Eq. 11), and parallelize the solving process. Our technique involves a new data structure to assemble the normal equations, and builds on top of the Local Schur Complement, a domain decomposition technique [Saa03], coupled with the spectral graph partitioning introduced in [NJW01]. The resulting parallel direct solver can handle semi-sparse linear problems efficiently and it maps surprisingly well onto modern GPUs.

4.1. Assembling Normal Equations

Assembling the normal equations (Eq. 12 and step 7 in Alg. 1) requires to compute the coefficients of the Jacobian matrix $J \in \mathbb{R}^{m\times n}$, i.e., the partial derivatives of all the m residuals r_j w.r.t. each parameter in $x = (x_1, x_2, \dots, x_n)$. Just storing these coefficients may require a significant amount of memory. An instance of the ALM problem using a medium-resolution mesh with 95K vertices, for example, has $m \approx 1.5M$ and $n \approx 7K$, which would require more than 40GB of memory assuming to use 4 bytes for storing numbers.

Furthermore, once the partial derivatives are available, the matrices $J^{\mathrm{T}}J$ and $J^{\mathrm{T}}r$ are needed and computed according to:

$$J^{\mathrm{T}}r = \sum_1^m \frac{\partial r_k}{\partial x_i} r_k, \qquad 1 \le i \le n \tag{16}$$

$$J^{\mathrm{T}}J = \sum_1^m \frac{\partial r_k}{\partial x_i}\frac{\partial r_k}{\partial x_j}, \qquad 1 \le i,j \le n. \tag{17}$$

The computation of the i-th element of $J^{\mathrm{T}}r$ requires the sum of coefficient-wise product of all the elements of the i-th column of the Jacobian with the corresponding residual value r_k. Similarly, the computation of (i,j)-th element of $J^{\mathrm{T}}J$ requires the sum of the coefficient-wise product of the elements in the i-th and j-th columns of the Jacobian.

The data structure used to store the Jacobian has a twofold purpose: it must store the partial derivatives of the residuals in a compact way, while allowing for the fast computations of both $J^{\mathrm{T}}J$ and $J^{\mathrm{T}}r$. The main idea behind the design of our data structure is to represent J as a graph. Each node in the graph gathers a subset of consecutive parameters $x = (x_i, \ldots, x_j)$. The set of parameters for one node is disjoint from the set of any other node. Each node has the same number of parameters P, except the last one. Each node i stores all the columns from J corresponding to its parameters, forming a sub-matrix J_i. If a row of the matrix J_i is composed entirely of zeros, then it is discarded and not stored. Two nodes are connected if there is at least one residual using parameters from both nodes. The indices of the connected nodes are stored in a support data structure, a matrix M, where $M(i, j)$ stores all the indices of the rows corresponding to the shared residuals. When all the partial derivative w.r.t. a given parameter x_i are needed, the corresponding node is found by using $\lfloor i/P \rfloor$, and the index of the column inside the node is $(i \bmod P)$. The structure is depicted in Fig. 3.

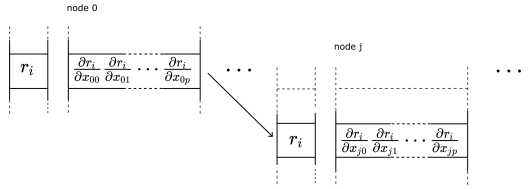


Figure 3: Graph data structure used to store the Jacobian matrix and speed up the computation of $J^{\mathrm{T}}r$ and $J^{\mathrm{T}}J$. The partial derivatives are stored in nodes, each node corresponding to a subset of consecutive columns. Each row corresponds to a residual $r_k(x)$ and two nodes are connected only if they share a residual.

When the i-th coefficient of $J^{\mathrm{T}}r$ is computed, it is sufficient to access the node where the column corresponding to x_i is stored, scan it and accumulate the result. Since just the non-zero rows of partial derivatives are stored, the number of elements to scan will be as small as possible. Furthermore, we compute all the coefficients of $J^{\mathrm{T}}r$ in a single parallel sweep because the scan of a column is independent from all the other columns. Similar considerations can be made for computing the coefficients of $J^{\mathrm{T}}J$. If the (i, j)-th elements belong to the same node, then the columns i and j are swept and their coefficient-wise product is accumulated. If they belong to different nodes, then the corresponding columns are accessed but only the elements belonging to the residuals in common between the two nodes are considered by accessing the support matrix M. Again, all the coefficients of $J^{\mathrm{T}}J$ can be computed in one single parallel step.

Choosing the size of the nodes P, and thus which parameters to store in each node, depends on the structure of the problem. In the ALM case, it is natural to map a local patch to each node, that is the parameters corresponding to the rigid parameters $M_i(t)$ and the deformation components $\{\alpha_i^1(t), \ldots, \alpha_i^K(t)\}$. The main benefit of this data structure is that it allows to quickly access the non-zero elements of the Jacobian matrix and compute $J^{\mathrm{T}}r$ and $J^{\mathrm{T}}J$.

Discussion. Recall that, when using dual numbers, *all* the partial derivatives for a given residual are computed at the same time. To compute the (i, j)-th coefficient of $J^{\mathrm{T}}J$ on the fly, all the residuals $r_k, k \in \{1, \ldots, m\}$ should be considered. So, for each $i, j \in \{1, \ldots, n\}$, and for each $r_k, k \in \{1, \ldots, m\}$:

1. compute all the partial derivatives $\frac{\partial r_k}{\partial x_i}$ using dual numbers;
2. compute the product $\frac{\partial r_k}{\partial x_i}\frac{\partial r_k}{\partial x_j}$ and discard all $\frac{\partial r_k}{\partial x_l}$ with $l \neq i, j$;
3. accumulate and consider the next residual.

Computing all the partial derivatives (step 1), would not be a big burden for modern GPUs, if the number of parameters for each residual would be small, e.g., like in [PGB03, SA07] and all the other problems solved in [DMZ*17]). If the residual depends on many parameters, however, and it needs to fetch a significant amount of data from the video memory, then a matrix-free approach becomes unsuitable for fast performance.

In semi-sparse problems, however, this is in general not true because residuals may depend on hundreds of parameters, e.g., the overlap or anatomical constraints in the ALM model is influenced by over 2000 parameters. In this case, it is more efficient to prestore the partial derivatives and access them on demand when they are needed, which, in our case, is achieved by employing the just depicted data structure.

4.2. Local Schur Complement

Solving the linear system composed by the normal equations (Eq. 12) is the main bottleneck in each nonlinear iteration of the *LM* algorithm (Alg. 1). In this section, we describe the adopted schema to solve it quickly by employing the Local Schur Complement method.

The matrix $J^{\mathrm{T}}J$ in the normal equations is a Symmetric Positive Definite (SPD) matrix by definition. We define the *adjacency graph* of $J^{\mathrm{T}}J$ as an undirected graph where each node corresponds to a row (or a column) of $J^{\mathrm{T}}J$, and two nodes are connected with each other if the corresponding entry in the matrix is not zero.

For the sake of simplicity and without loss of generality, let us consider as an example the adjacency graph of a very simple matrix, shown in Fig. 4a. This graph can be partitioned into three subgraphs by cutting some of the edges as shown in Fig. 4b. The resulting subgraphs are composed by *internal nodes* connected to only other nodes of the same subgraph, and *interface nodes* with at least one neighbour belonging to another subgraph. By relabeling the internal nodes of each subgraph in incremental order and leaving the interface nodes last, we induce a *permutation* π of the rows and columns of the input matrix (Fig. 4c), resulting in the structure shown in Fig. 5a, where most of the coefficients are now lumped in diagonal blocks.

The emerging pattern of the coefficients in the permuted matrix, shown in Fig. 5b, allows to define a heavily parallelized solving process. Each diagonal block i, corresponding to one of the subgraphs can be further decomposed in four sub-blocks:

$$\begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}, \tag{18}$$

where the matrix B_i represents the internal nodes of the subgraph

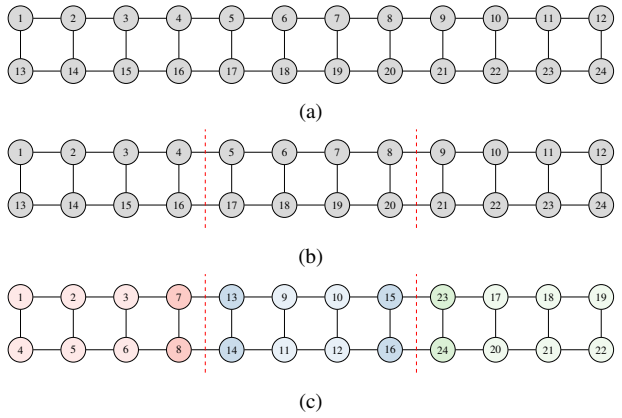


Figure 4: A simple adjacency graph (a) is partitioned into equal parts by minimizing the edge cut (b). By relabeling the nodes (c), we induce a permutation of the graph and the corresponding matrix.

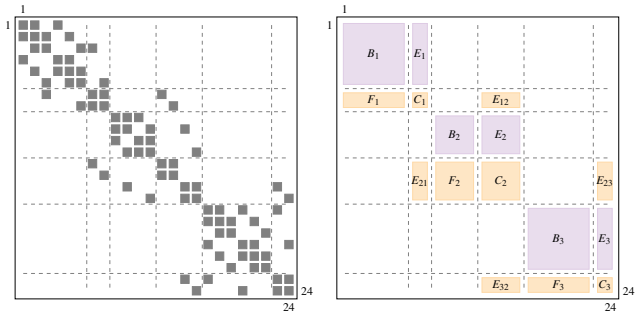


Figure 5: (a) The permuted matrix corresponding to the graph in Fig. 4c, and its block structure (b). The blocks are either corresponding to internal nodes of the graph (purple), or external nodes (orange).

i, C_i represents its interface nodes, E_i represents the coupling between internal and interface nodes, and $F_i = E_i^{\mathrm{T}}$. The off-diagonal blocks are composed by mostly zeros, and have the following structure:

$$\begin{pmatrix} 0 & 0 \\ 0 & E_{ij} \end{pmatrix}. \tag{19}$$

The matrix E_{ij} represents the coupling between the interface nodes belonging to different subgraphs. If two subdomains i and j are loosely connected, then E_{ij} has a small size; if they are entirely disconnected, then the corresponding E_{ij} is also a null matrix.

By applying the permutation π to the unknowns and the constant terms of the linear system in *Eq.* 12, the linear equation for each subgraph i can be written as

$$B_i\delta_{ai} + E_i\delta_{bi} = f_i \tag{20}$$

$$F_i\delta_{ai} + C_i\delta_{bi} + \sum_{j\in N_i} E_{ij}\delta_{bi} = g_i. \tag{21}$$

where δ_{ai} is the vector of internal nodes, δ_{bi} is the vector of interface nodes, and N_i is the set of subgraphs linked by at least one edge to subgraph i. The subvectors f_i and g_i are extracted from $-J^{\mathrm{T}}r$ after permuting it according to π. Note that Eq. 20 corresponds to the purple blocks in Fig. 5b, while Eq. 21 corresponds to the orange ones. The vector δ_{ai} can be expressed as

$$\delta_{ai} = {B_i}^{-1}(f_i - E_i\delta_{bi}) \tag{22}$$

and it can be forward substituted into Eq. 21 leading to

$$S_i\delta_{bi} + \sum_{j\in N_i} E_{ij}\delta_{bi} = g_i - F_i{B_i}^{-1}f_i = {g'}_i \tag{23}$$

where $S_i = C_i - {E_i}^{\mathrm{T}}{B_i}^{-1}E_i$ is the so-called *Local Schur Complement*.

For example, writing Eq. 23 for all the subgraphs in Fig. 5 yields the linear system of equations for the interface nodes δ_{bi}

$$\begin{pmatrix} S_1 & E_{12} & E_{13} \\ E_{21} & S_2 & E_{23} \\ E_{31} & E_{32} & S_3 \end{pmatrix} \begin{pmatrix} \delta_{b0} \\ \delta_{b1} \\ \delta_{b2} \end{pmatrix} = \begin{pmatrix} {g'}_0 \\ {g'}_1 \\ {g'}_2 \end{pmatrix}. \tag{24}$$

The off-diagonal elements of the matrix are mostly zero because the sparse blocks E_{ij} are different from the null matrix only if the subgraphs i and j are connected. The diagonal blocks S_i are in general dense and their size is equal to the number of interface nodes in subgraph i. Eq. 24 is solved using Cholesky factorization and its solution can be back-substituted into Eq. 22 allowing to recompose the global solution.

We aim to obtain small linear systems because these can be solved quickly on consumer-class GPUs using parallel direct methods. The goal of the partitioning algorithm, thus, is to keep the number of interface nodes as small as possible, as explained in the next section.

4.3. Spectral Partitioning

In this section, we explain how to decompose the adjacency graph into balanced partitions while keeping the number of interface nodes as low as possible. Having balanced partitions is useful to keep the GPU saturated, while having few interface points is important to keep the size of the matrix S in Eq. 24 small, and to solve it efficiently with a direct Cholesky factorization. This is equivalent to finding an optimal cut of the graph, which is a NP-hard partitioning problem [Ski08], but there exist many different approaches in the literature to solve its relaxed version and find semi-optimal cuts efficiently [SKK03]. We chose to use a method belonging to the *spectral partitioning* class of algorithms, which address the problem from an algebraical point of view. By doing that, the proposed method becomes entirely independent of the geometrical structure of the problem, the number of parameters for each residual function, as well as the number of residual functions, making our solver more general than many existing solvers which oftentimes rely on knowledge of the underlying domain, e.g. that they operate on the regular pixel grid of an image. In particular, we use the normalized *spectral clustering algorithm*, which has performed very well on a number of challenging clustering algorithms [NJW01].

One of the key steps of the algorithm is the computation of the

Algorithm 2 Normalized spectral partitioning [NJW01]

1: $L_n = D^{-1/2} L D^{-1/2}$
2: Compute the smaller k not null eigenvectors $v_1, \cdots, v_k$ of L_n
3: $V = [v_1, \cdots, v_k]$, $v_i \in \mathbb{R}^{n \times 1}$, $V \in \mathbb{R}^{n \times k}$
4: **for all** row vectors $(u_i)_{i=1,\cdots,n} \in V$ **do**
5: $\quad u_i \leftarrow u_i / \|u_i\|$ ▷ Normalize rows
6: Use k++-means to cluster u_i into sets $C_1, \cdots, C_k$
7: **for all** clusters $(C_i)_{i=1,\cdots,k}$ **do**
8: $\quad A_i = \{ j | u_j \in C_i \}$

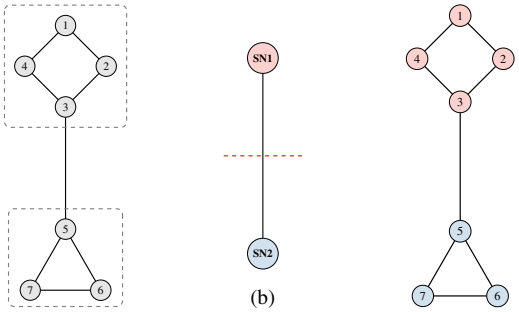
eigenvectors and the eigenvalues of the *normalized Laplacian* matrix L_n, which is defined as follows. We recall the Laplacian matrix L of an undirected graph defined as

$$L = D - W. \tag{25}$$

D is a diagonal matrix with each element d_i equal to the degree of the node i of the graph, and W is a matrix with each element $w_{ij} = 1$ only if nodes i and j are connected by an edge, 0 otherwise. The normalized Laplacian is defined as:

$$L_n = D^{-1/2} L D^{-1/2}. \tag{26}$$

In general, computing the eigenvectors of L_n is the most computationally expensive step and its cost is proportional to the number of nodes and edges in the graph. One way to speed it up is to cluster nodes into super nodes, partition the resulting clustered graph, and finally expand the obtained partitions back to the original nodes, as shown in the simple example in Fig. 6. Different criteria may be

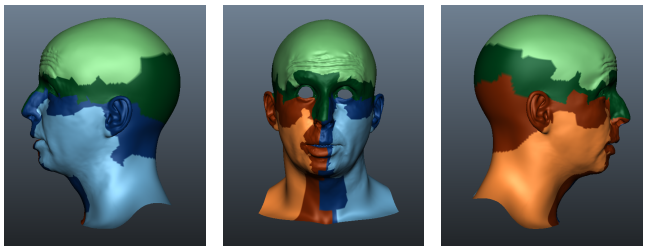


(a) (b) (c)

Figure 6: (a) Nodes in an example graph are clustered together into *super nodes*. The resulting graph is partitioned (b), and super nodes are expanded back to their original configuration leading to the final partitioning (c).

used to cluster nodes [KK98]. In the context of the ALM model, parameters influencing the same patch are always used together in all cost functions considered in this paper. It is thus natural to cluster all the nodes corresponding to the parameters affecting the same patch.

Once the partitioning is performed according to Alg. 2 [NJW01], we expand each node in a cluster of subnodes, each one representing a parameter in the normal equations (Eq. 12). A graphical example of the obtained partitioning is depicted in Fig. 7, and an example of the partitioning obtained with our approach on one of the test cases is shown in Fig. 8.



(a) (b) (c)

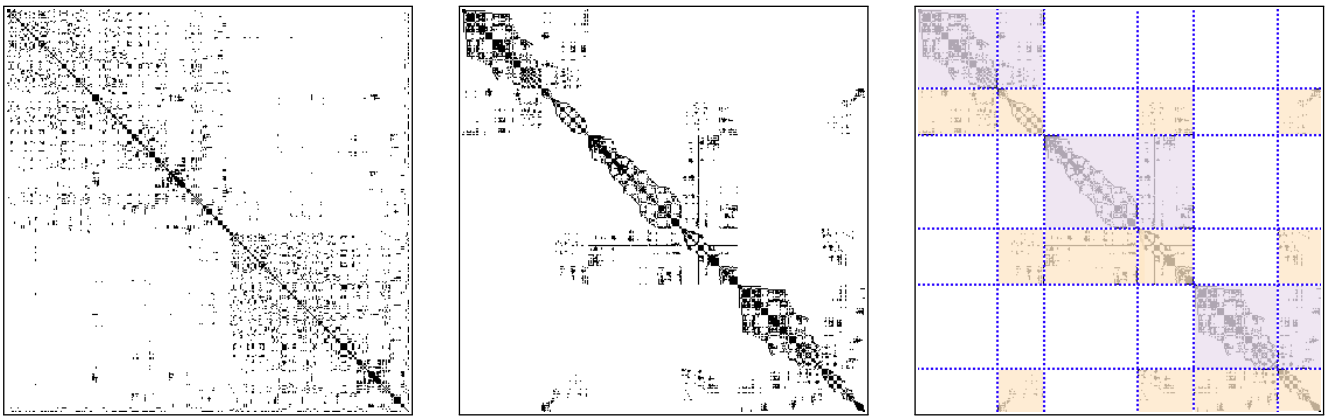
Figure 7: A face model is partitioned using Alg. 2. Lighter colors represent internal nodes in the adjacency graph, while darker colors represent external nodes.

In general, the eigendecomposition of the matrix can be onerous in terms of time. However, it is required just once per nonlinear iteration, and it took merely 40 ms in our test case when using the clustered matrix. For comparison reasons, we applied two other permutation algorithms to the same test case, namely the *Approximate Minimum Degree Ordering* [ADD96] and *METIS* [KK98]. Both methods are commonly useful to reduce the fill-in in the Cholesky factorization leading to a faster solution. The resulting reordering is visualized in Fig. 9, and the corresponding timings for solving one single linear iteration are reported in Table 1, showing that our approach is at least 1.7X faster due to how well it maps on the parallel architecture of the GPU. The performance breakdown considering the main computational steps in a single nonlinear iteration are depicted in Table 2

4.4. Implementation

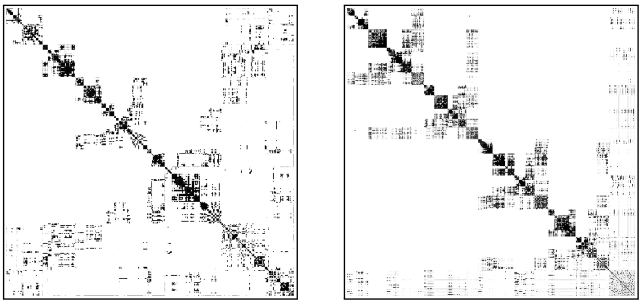
Computing the inverse matrix in ${B_i}^{-1} E_i$ and ${B_i}^{-1} f_i$ for each subdomain i is one of the main performance bottlenecks of the linear solver. We avoid it by not explicitly computing the inverse matrices. Instead, we solve the linear systems $B_i E_i' = E_i$ and $B_i f_i' = f_i$, and use E_i' and f_i' in Eq. 22 and 23. This approach is more convenient for two reasons. First, the matrices E_i are, in general, sparse and some of the columns are zeros, and are skipped during the solving process. Secondly, and more importantly, we solve the systems by performing a parallel Cholesky decomposition of each matrix B_i only once, and then solving for all the columns of E_i and f_i in a single parallel step. Additionally, the computation of each subdomain is fully decoupled from the others and can be directed to different streams. Overall, all the linear systems can be computed in one single parallel step. Besides being fast, avoiding to compute the inverse matrices is also less sensitive to numerical inaccuracies caused by the floating number representation (which we use to maintain fast performance).

The size of B_i is smaller than the size of $J^{\mathrm{T}} J$ divided by the number of partitions k. Being relatively small, the parallel Cholesky factorization can be executed quickly on modern GPUs. The number of partitions, however is limited by the structure of the graph; to maintain a low number of interface nodes while having balanced



(a) (b) (c)

Figure 8: (a) Input matrix J^TJ from one of the test scenes. (b) The permuted matrix according to Alg. 2. (c) Visualization with the same colour scheme used in Fig. 5b



(a) (b)

Figure 9: Permutations of the matrix in Fig. 8a according to *Approximate Minimum Degree Ordering* [ADD96] (left), and *METIS* [KK98] (right). The performance comparison in Table 1 shows that our method using spectral partitioning is faster.

partitions it is convenient to choose k to be the number of eigenvalues of J^TJ that are strictly positive and smaller than one [HL95].

To implement the GPU direct solver, we used the Cholesky implementation available in cuSolver (`cusolverDnSpotrf`), which is part of the Cuda toolkit 10.1.

We compared the performance of our solver with the Jacobi-Preconditioned Conjugate Gradient (PCG), as well implemented on the GPU. For each nonlinear iteration of the LM algorithm, we compared the overall time for solving the normal equations. Results are reported in Fig. 10 and show that our linear solver is up to four times faster than PCG.

5. Results

In this section, we present several numerical test cases to prove the effectiveness and scalability of our minimizer. We demonstrate that we are able to qualitatively reproduce the numerical results of

	Solve Time (ms) Single linear step	**Speed up**
Our solver	**27**	-
Symamd	46	1.7X
Metis	52	1.9X
None	57	2.1X

Table 1: Performance comparison to solve a single linear iteration with other common approaches to speed up Cholesky factorization: symamd [ADD96] and metis [KK98].

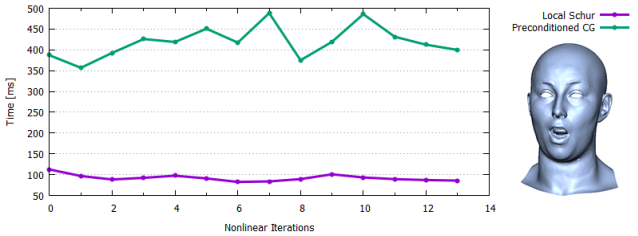


Figure 10: Comparison of the Jacobi-Preconditioned Conjugate Gradient (PCG) with our linear solver based on the local Schur complement. For each nonlinear iteration of the LM algorithm, The plot shows the time to solve the normal equations and reach convergence.

Ceres [AMO], a nonlinear minimizer running on the multi-threaded CPU widely used both in the industry and in the academia.

We implemented our solver on the GPU using Cuda, and performed the comparisons by running Ceres on an Intel i7-7700K CPU, quad-core processor with 8 hyperthreads, and our solver on the Nvidia GeForce GTX 1080 Ti with 28 Multi-Processors. In the domain of model-based facial capture using the ALM model, we reconstructed animation sequences from four different actors. In

Stage	Our Solver	Ceres - 1	Ceres - 4
Residual Eval. Only [ms]	8.373	33.48	13.17
Jacobian Evaluation [ms]	122.193	4249.98	1124.11
Linear Solve [ms]	26.723	2555.49	2277.72

Table 2: Performance comparison between Ceres running on 1 thread, on 4 threads and our solver for a single nonlinear iteration. Note that our system has 4 cores.

all the test cases, the time to reach the solution by our solver was one order of magnitude smaller than the time used by Ceres.

Fig. 11 illustrates the performance comparison for a single run of the model parameter optimization. Note that solving for 1 frame of an animation often requires up to 10 optimization solves with updated constraints each iteration, but this figure isolates just a single iteration for comparison purposes. For all four actors we illustrate both the number of nonlinear iterations as well as the total time in seconds. Our method terminates in a similar convergence pattern as Ceres (left plot), but each iteration is much faster, leading to an order of magnitude speedup (right plot, in log time). For each frame, we set a small error threshold (10^{-5}) for both solvers and let them run to convergence. Ceres' configuration was: Minimizer: `TRUST_REGION`; Trust region strategy: `LEVENBERG_MARQUARDT`, Linear solver: `SPARSE_SCHUR`. Qualitatively, we show that the accuracy of our solver is also very similar to the accuracy achieved by Ceres, by reconstructing the full sequence for each actor and visualizing the difference color coded on the meshes. Fig. 12 illustrates 3 frames from each sequence, and we refer to the accompanying video for the entire performances. Our solver produced visually indistinguishable results to Ceres in the frontal face region where most constraints are located, with only minor differences (< 0.2mm) in other parts of the head such as the back or the neck where there are fewer constraints to guide the optimization.

Finally, we compare the time to solve the ALM model optimization as a function of the size of the ALM model, specifically the dimensionality of the individual patch subspaces, which directly correlates with the number of variables. Table 3 illustrates that the solve is supralinear in the number of parameters.

Model Size (in ALM subspace dimensions)	Solve Time (in seconds)
4	0.35
8	0.58
15	0.98
20	1.1

Table 3: Solve time is supralinear in the dimensionality of the ALM subspace.

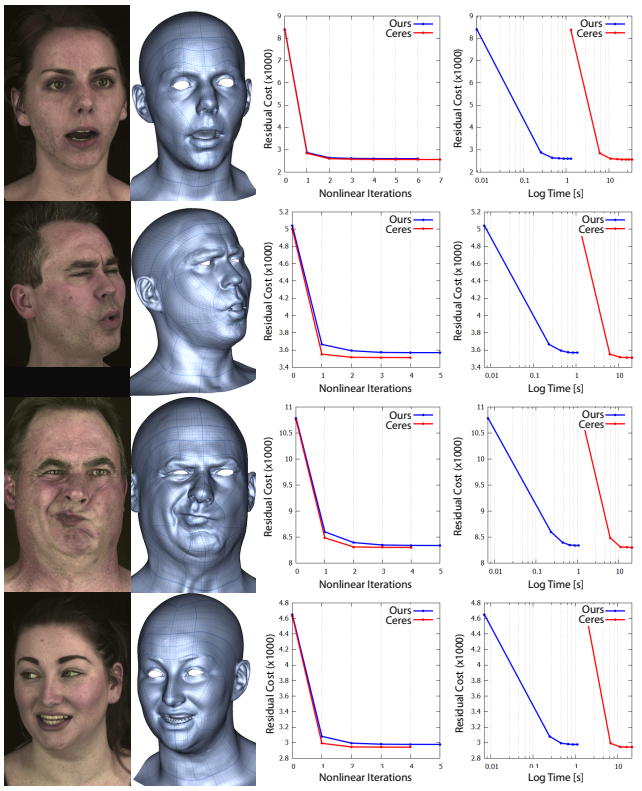


Figure 11: The performance of our method compared with the Ceres solver [AMO], when solving different animation sequences. Our method has a nearly identical convergence speed (left), but the solution is achieved one order of magnitude faster (right).

6. Conclusions

The fields of computer graphics, vision, robotics and interaction contain a very common element - at the heart of many challenges lies a nonlinear optimization problem, which often forms the bottleneck of computation. For some problems, for example those with a small number of variables or a sparse relationship between the variables and residuals, fast solutions have been proposed. However, in the case of large-scale and semi-sparse problems, the only current solution is slow iterative solvers such as Ceres. To the best of our knowledge, we propose the first GPU-based fast solver for semi-sparse problems. The novelty of our method lies in a new graph-based data structure for storing and accessing elements of the Jacobian matrix, coupled with a parallel linear solver that uses the local Schur complement, and the application of spectral space partitioning to decompose the adjacency graph into balanced partitions. The result is a highly scalable, parallel solver that can be mapped onto graphics hardware.

To demonstrate our solver, we applied it to the optimization problem of model-based facial capture using a recently-proposed anatomical local face model (ALM), which is a large nonlinear optimization problem with many parameters contributing to the cost functions. We demonstrate robustness to various instantiations of the problem, namely different actors and model sizes, and also

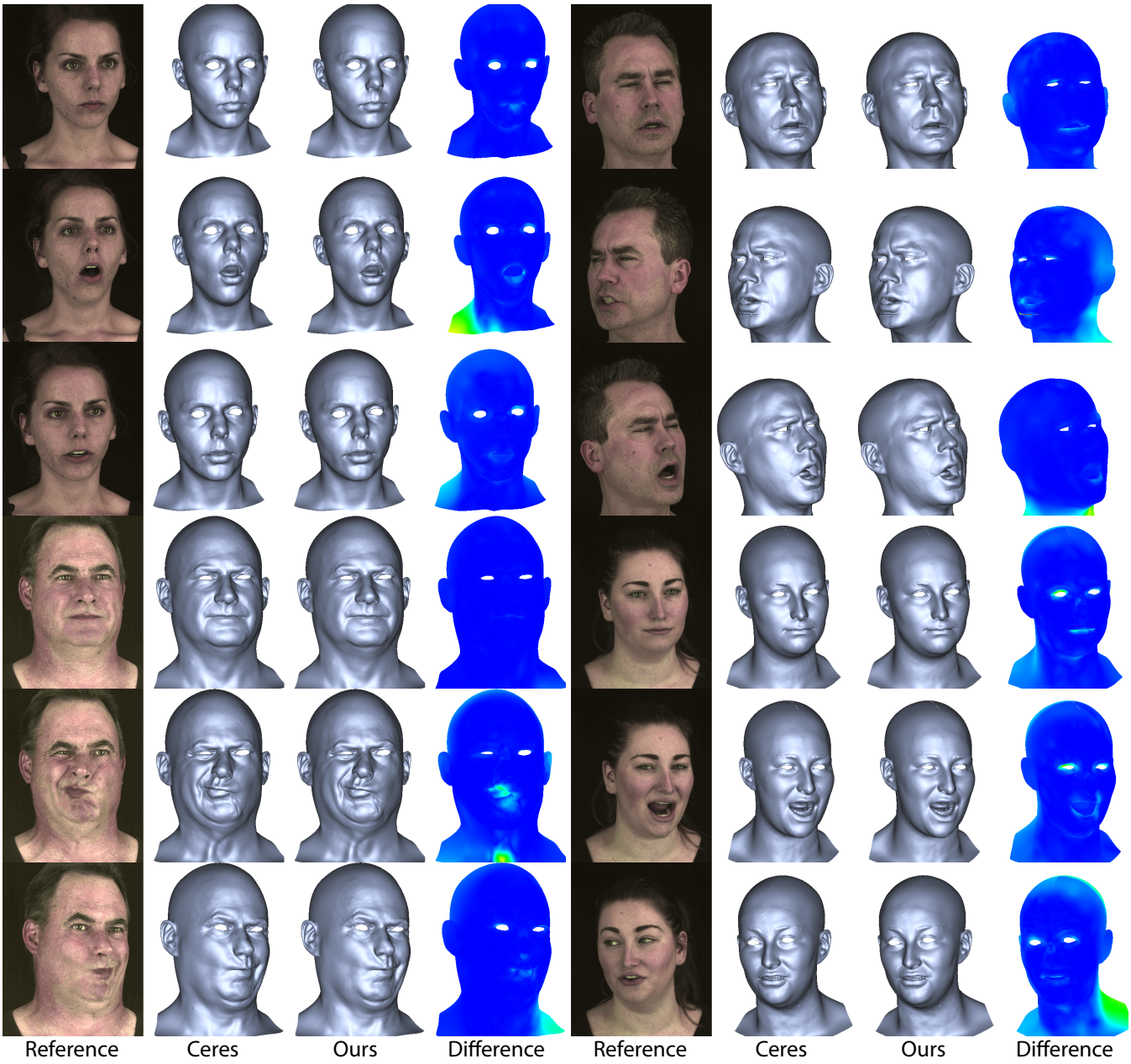


Figure 12: We demonstrate the accuracy of our fast nonlinear solver with the application of facial capture using an anatomical local model (ALM), and compare to the same result solved using the Ceres solver [AMO]. Here we show 3 frames from sequences of 4 different actors. The reconstruction results differ only very little, as illustrated in the heatmap (from blue=0mm to red=0.2mm), and the majority of the difference is in the back of the head and shoulders where there is no data to constrain the optimization. Despite nearly identical results, our method ran 10 times faster per face solve, and each frame contained 8 iteration of solving, yielding a significant speedup.

compared accuracy and run time to a CPU-based Ceres solve of the same problem. In all cases, our new solver vastly outperforms the alternative while remaining extremely accurate.

Future research. Even though we demonstrate our results solely in the context of the ALM, most of the presented concepts are sufficiently generic to be applied to any semi-sparse problem. The partitioning, for example, abstracts entirely from the geometry of the problem by considering the algebraic structure of the matrix. In fact, the only component specific to the ALM is the clustering phase. As a future research, and in order to make the algorithm completely generic, this step could be realized by greedily detecting and clustering cliques in the adjacency graph of the Laplacian. This is the so-called *clique cover* problem, which can be reduced to a graph coloring of the Laplacian's complement graph [Kar72]. This latter is the graph on the same vertex set that has edges between non-adjacent vertices of the Laplacian. Finding independent sets in the complement graph using a fast, approximated coloring algorithm (for example, one of the parallel approaches in [FTP16]), corresponds to finding cliques in the Laplacian graph. Such cliques can be clustered together to form super nodes, reducing the size of the Laplacian and allowing a fast eigendecomposition for the spectral partitioning described in Sec. 4.3.

References

[ADD96] AMESTOY P. R., DAVIS T. A., DUFF I. S.: An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17*, 4 (Oct. 1996), 886–905. 8, 9

[AMO] AGARWAL S., MIERLE K., OTHERS: Ceres solver. http://ceres-solver.org. 2, 9, 10, 11

[BBC*94] BARRETT R., BERRY M., CHAN T. F., DEMMEL J., DONATO J., DONGARRA J., EIJKHOUT V., POZO R., ROMINE C., DER VORST H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. 2, 4

[BV04] BOYD S., VANDENBERGHE L.: *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. 2

[CZY17] CHU J., ZAFAR N. B., YANG X.: A schur complement preconditioner for scalable parallel fluid simulation. *ACM Trans. Graph. 36*, 4 (July 2017). 3

[DMZ*17] DEVITO Z., MARA M., ZOLLHÖFER M., BERNSTEIN G., RAGAN-KELLEY J., THEOBALT C., HANRAHAN P., FISHER M., NIESSNER M.: Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Trans. Graph. 36*, 5 (Oct. 2017), 171:1–171:27. 3, 6

[DNZ*17] DAI A., NIESSNER M., ZOLLÖFER M., IZADI S., THEOBALT C.: Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. *ACM Trans. Graph. (TOG)* (2017). 3

[ELaC19] EISENBERGER M., L ÄHNER Z., CREMERS D.: Divergence-free shape correspondence by deformation. *Computer Graphics Forum 38*, 5 (2019), 1–12. 2

[FBT*18] FANG X., BAO H., TONG Y., DESBRUN M., HUANG J.: Quadrangulation through morse-parameterization hybridization. *ACM Trans. Graph. 37*, 4 (July 2018), 92:1–92:15. 2

[FTP16] FRATARCANGELI M., TIBALDO V., PELLACINI F.: Vivace: A practical gauss-seidel method for stable soft body dynamics. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 214:1–214:9. 3, 5, 12

[GW08] GRIEWANK A., WALTHER A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. 5

[HL95] HENDRICKSON B., LELAND R.: An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing 16*, 2 (3 1995). 9

[HLSO12] HECHT F., LEE Y. J., SHEWCHUK J. R., O'BRIEN J. F.: Updated sparse cholesky factors for corotational elastodynamics. *ACM Trans. Graph. 31*, 5 (Sept. 2012), 123:1–123:13. 2, 3

[Kar72] KARP R.: Reducibility among combinatorial problems. In *Complexity of Computer Computations*, Miller R., Thatcher J., (Eds.). Plenum Press, 1972, pp. 85–103. 12

[KK98] KARYPIS G., KUMAR V.: A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (Dec. 1998), 359–392. 8, 9

[LAR*14] LEWIS J. P., ANJYO K., RHEE T., ZHANG M., PIGHIN F., DENG Z.: Practice and theory of blendshape facial models. In *Eurographics 2014 - State of the Art Reports* (2014), Lefebvre S., Spagnuolo M., (Eds.), The Eurographics Association, pp. 199–218. 3

[Lev44] LEVENBERG K.: A method for the solution of certain non-linear problems in least squares. *Quart. Appl. Math.*, 2 (1944), 164–168. 2, 5

[LGL*19] LI M., GAO M., LANGLOIS T., JIANG C., KAUFMAN D. M.: Decomposed optimization time integrator for large-step elastodynamics. *ACM Trans. Graph. 38*, 4 (July 2019), 70:1–70:10. 2, 3

[LMAS16] LIU H., MITCHELL N., AANJANEYA M., SIFAKIS E.: A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 201:1–201:12. 3

[Mar63] MARQUARDT D. W.: An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math. 11*, 2 (1963), 431–441. 2, 5

[MEM*19] MACKLIN M., ERLEBEN K., MÜLLER M., CHENTANEZ N., JESCHKE S., MAKOVIYCHUK V.: Non-smooth newton methods for deformable multi-body dynamics. *CoRR abs/1907.04587* (2019). 3

[MFZ*17] MEKA A., FOX G., ZOLLHÖFER M., RICHARDT C., THEOBALT C.: Live user-guided intrinsic video for static scene. *IEEE Transactions on Visualization and Computer Graphics 23*, 11 (NOVEMBER 2017). 3

[MHR*16] MUSIALSKI P., HAFNER C., RIST F., BIRSAK M., WIMMER M., KOBBELT L.: Non-linear shape optimization using local subspace projections. *ACM Trans. Graph. 35*, 4 (July 2016), 87:1–87:13. 2

[NJW01] NG A. Y., JORDAN M. I., WEISS Y.: On spectral clustering: Analysis and an algorithm. In *Neural Information Processing Systems: Natural and Synthetic* (Cambridge, MA, USA, 2001), NIPS'01, MIT Press, pp. 849–856. 5, 7, 8

[PGB03] PÉREZ P., GANGNET M., BLAKE A.: Poisson image editing. *ACM Trans. Graph. 22*, 3 (July 2003), 313–318. 6

[SA07] SORKINE O., ALEXA M.: As-rigid-as-possible surface modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2007), SGP '07, Eurographics Association, pp. 109–116. 6

[Saa03] SAAD Y.: *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. 3, 5

[Ski08] SKIENA S. S.: *The Algorithm Design Manual*, 2nd ed. Springer Publishing Company, Incorporated, 2008. 7

[SKK03] SCHLOEGEL K., KARYPIS G., KUMAR V.: Sourcebook of parallel computing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003, ch. Graph Partitioning for High-performance Scientific Simulations, pp. 491–541. 7

[SPSH*17] SHTENGEL A., PORANNE R., SORKINE-HORNUNG O., KOVALSKY S. Z., LIPMAN Y.: Geometric optimization via composite majorization. *ACM Trans. Graph. 36*, 4 (July 2017), 38:1–38:11. 2

[Sug11] SUGIHARA T.: Solvability-unconcerned inverse kinematics by the levenberg-marquardt method. *IEEE Transactions on Robotics 27*, 5 (Oct 2011), 984–991. 2

[TMHF00] TRIGGS B., MCLAUCHLAN P. F., HARTLEY R. I., FITZGIBBON A. W.: Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice* (London, UK, UK, 2000), ICCV '99, Springer-Verlag, pp. 298–372. 2

[TZS*16] THIES J., ZOLLHÖFER M., STAMMINGER M., THEOBALT C., NIESSNER M.: Face2face: Real-time face capture and reenactment of rgb videos. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016), pp. 2387–2395. 2, 3

[Wan15] WANG H.: A chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Trans. Graph. 34*, 6 (Oct. 2015), 246:1–246:9. 3, 5

[WBGB16] WU C., BRADLEY D., GROSS M., BEELER T.: An anatomically-constrained local deformation model for monocular face capture. *ACM Trans. Graph. 35*, 4 (July 2016), 115:1–115:12. 2, 3, 4

[WTYH18] WU K., TRUONG N., YUKSEL C., HOETZLEIN R.: Fast fluid simulations with sparse volumes on the gpu. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2018) 37*, 2 (2018), 157–167. 3

[WY16] WANG H., YANG Y.: Descent methods for elastic body simulation on the gpu. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 212:1–212:10. 3, 5

[WZN*14] WU C., ZOLLHÖFER M., NIESSNER M., STAMMINGER M., IZADI S., THEOBALT C.: Real-time shading-based refinement for consumer depth cameras. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 200:1–200:10. 3

[ZNI*14] ZOLLHÖFER M., NIESSNER M., IZADI S., REHMANN C., ZACH C., FISHER M., WU C., FITZGIBBON A., LOOP C., THEOBALT C., STAMMINGER M.: Real-time non-rigid reconstruction using an rgb-d camera. *ACM Trans. Graph. 33*, 4 (July 2014), 156:1–156:12. 2, 3