On Obtaining Sparse Semantic Solutions for Inverse Problems, Control, and Neural Network Training

David A. B. Hyde\textsuperscript{a,*}, Michael Bao\textsuperscript{b,**}, Ronald Fedkiw\textsuperscript{b,c,**}

\textsuperscript{a}UCLA Mathematics Department, Box 951555, Los Angeles, CA 90095-1555, United States
\textsuperscript{b}Stanford University, 353 Jane Stanford Way, Gates Computer Science Room 207, Stanford, CA 94305, United States
\textsuperscript{c}Epic Games, Inc., 620 Crossroads Boulevard, Cary, NC 27518, United States

Abstract

Modern-day techniques for designing neural network architectures are highly reliant on trial and error, heuristics, and so-called best practices, without much rigorous justification. After choosing a network architecture, an energy function (or loss) is minimized, choosing from a wide variety of optimization and regularization methods. Given the ad-hoc nature of network architecture design, it would be useful if the optimization led to a sparse solution so that one could ascertain the importance or unimportance of various parts of the network architecture. Of course, historically, sparsity has always been a useful notion for inverse problems where researchers often prefer the $L_1$ norm over $L_2$. Similarly for control, one often includes the control variables in the objective function in order to minimize their efforts. Motivated by the design and training of neural networks, we propose a novel column space search approach that emphasizes the data over the model, as well as a novel iterative Levenberg-Marquardt algorithm that smoothly converges to a regularized SVD as opposed to the abrupt truncation inherent to PCA. In the case of our iterative Levenberg-Marquardt algorithm, it suffices to consider only the linearized subproblem in order to verify our claims. However, the claims we make about our novel column space search approach require examining the impact of the solution method for the linearized subproblem on the fully nonlinear original problem; thus, we consider a complex real-world inverse problem (determining facial expressions from RGB images).

Keywords: Machine learning, Levenberg-Marquardt, principal component analysis, column space search, coordinate descent

1. Introduction

The current age of deep learning began (at least according to the Turing Award committee\textsuperscript{1}) with works addressing problems such as object classification [73, 82], reading handwritten digits and documents [80, 81, 79], and speech recognition and natural language tasks [11, 100]. Although models based on traditional scientific first principles do not exist for these sorts of problems, the underlying machine learning methods have been permeating into various scientific communities, including computational physics [69, 51, 49, 123, 48, 89, 113, 115, 132]. Perhaps the main difference between the use of machine learning for customizing advertisements [19, 55], online dating [36, 99], or self-driving cars [16, 67] and its use in computational physics is that our community has developed a fairly reasonable scientific and mathematical understanding of many of the problems of interest via a combination of theoretical, experimental, and computational approaches, especially as opposed to the ad-hoc data-driven nature of popular machine learning application areas. Unfortunately, ad-hoc approaches leave neural networks wide open to adversarial attacks [65, 3, 125], which does not bode well for predictive numerical capabilities. Therefore, one goal of our community (and perhaps contribution) would be to better understand neural network architectures in order to provide a more thorough and rigorous approach to designing them, similar to the contributions that the applied mathematics

\textsuperscript{*}dabh@math.ucla.edu, UCLA
\textsuperscript{**}mikebao@stanford.edu, fedkiw@cs.stanford.edu, Stanford University
\textsuperscript{1}https://awards.acm.org/about/2018-turing
community made to finite element simulation, e.g. reformulating spring and beam elements as basis functions [141, 142].

Techniques used in modeling and training neural networks are highly related to well-studied approaches for inverse problems and control. To understand some of the differences between inverse problems, control, and training neural networks, consider $Y = f(X; C)$, with input $X$, output $Y$, and function parameters $C$. In a typical inverse problem, one is given $Y$ and aims to find an $X$ that produces $Y$. Poor conditioning of the function $f$ or noise in the given/desired output $Y$ can lead to spurious information contained in $X$. Thus, various regularization approaches may be used to ascertain an $X$ with a high signal-to-noise ratio, see for example [26, 92, 131, 140] and the more general references [42, 39, 12]. In the control problem, $X$ and $Y$ are both given, and the goal is to ascertain some subset of the function parameters $C$ that allows one to coerce $X$ toward $Y$. Typically, most of $f$ is a well-known function, such as the Navier-Stokes equations, and thus the added controls should have a light/minimal touch; therefore, they are often included in the objective function so that their magnitude/effort is minimized. This too is regularization, and needs to be done wisely so that minimizing controls does not prevent one from hitting the target (while still considering signal-to-noise ratio, etc.), see e.g. [70, 2, 118]. When considering neural networks, the function $f$ is almost entirely ad-hoc, and one does not know which parameters might have physicality and which are more arbitrary. Thus, it becomes even more important to consider careful regularization with the hope that some of the coefficients will dominate others, providing some insight into which portions of the network architecture may have some basis in first principles as opposed to which may be considered for removal/replacement, see e.g. [137, 95, 138, 133, 54, 90, 60, 93, 102, 134, 56, 109, 85, 4, 57, 58]. Because so little is known about $f$, neural networks cannot proceed with one input $X$ and one output $Y$ as can a control problem. Instead, one requires a family of given $(X; Y)$ pairs called training data, before an attempt to identify the function coefficients $C$ can be made. Methods for formulating and optimizing neural networks are typically significantly more rudimentary and ad-hoc than those designed for inverse and control problems, relying on simple methods such as gradient descent and stochastic gradient descent (SGD) or ordinary differential equation discretizations of gradient flow, such as Adam [72], AdaGrad [37], Nesterov [103], momentum methods [114, 126], etc. [117, 17, 53].

The process of network architecture design is often motivated by heuristics that hinder the ability to subsequently train the network and find suitable coefficients. For example, the “all or none” property of biological neurons leads to discontinuous functions with identically zero derivatives almost everywhere, which is disastrous for optimization/training [98]. The idea that biological neurons fire with increased frequency for stronger signals leads to piecewise linear functions with discontinuous derivatives, also problematic for optimization. These Heaviside and rectifier/ReLU [53] models require smoothing before they can subsequently be used with numerical optimization. It seems quite dubious to design and analyze non-smooth network architectures that are later smoothed in the first significant digit when deployed in practice, especially given the nuances exposed by the numerical analysis community regarding the differences between continuous and discrete formulations (e.g. [61]) even when such occurs only at the level of machine precision (the $7^{th}$ or $15^{th}$ decimal place). This motivates our aim to better utilize various approaches to regularization and sparsity to ascertain the importance of various components of the network architecture.

In Section 2, we introduce a suitably complex model problem for demonstrating the numerical methods presented in the paper: determining facial expression from RGB images. This problem is both algorithmically challenging and grounded in physics, meaning that we can attempt to develop algorithms which find semantically meaningful solutions in terms of known physical and anatomical properties. In Section 3, we outline a general framework for optimization, showing how neural network training is recast as a nonlinear optimization problem. We highlight various approximations made in practice during this process, such as (sometimes drastic) approximations to the Hessian and Jacobian. As is typical, rank-one updates are discussed, which motivates the singular value decomposition (SVD) and principal component analysis (PCA), both of which are used in subsequent sections. Section 4 presents a novel iterative Levenberg-Marquardt [83, 97] scheme that is shown by proof and experiment to converge smoothly (and monotonically) to a regularized SVD, unlike the truncation typical of a PCA approach. Section 5 presents a novel column space search technique that focuses more on the data term than the model, again an improvement over PCA. Moreover, we explain how column space search enables the discovery of sparse and semantically meaningful solutions to fully nonlinear optimization problems. To demonstrate this experimentally, we compare to alternative strategies such as Dogleg [112, 94] and BFGS with $L_2$ or soft $L_1$ regularizers.
2. Tackling Complex Real-World Inverse Problems for Faces

We chose a fairly complex model problem which is still cutting-edge in order to illustrate the need for robust and efficient approaches. Specifically, we consider an inverse problem where a two-dimensional RGB image of a human face is processed to determine facial expressions in terms of a three-dimensional parameterized model with a semantic, anatomical basis. This inverse problem is useful throughout industries such as medicine, surveillance, intelligence gathering, entertainment, etc. Similar to many other complex processes, one can understand the problem via a pipeline with various function layers, see Figure 1.

\[
\begin{align*}
X_1 \xrightarrow{\text{input}} & f_1 (X_1, C_1) \xrightarrow{\text{output}} X_2 \\
X_2 \xrightarrow{\text{input}} & f_2 (X_2, C_2) \xrightarrow{\text{output}} X_3 \\
X_3 \xrightarrow{\text{input}} & f_3 (X_3, C_3) \xrightarrow{\text{output}} X_4
\end{align*}
\]

\[
\hat{f}(X_4) = ||X_4 - X_{\text{target}}||
\]

Figure 1: Multiple layers of functions \(f_i\) map an initial vector of inputs \(X_1\) to a final output \(X_4\), which is evaluated with an objective function \(\hat{f}\). Vectors of parameters \(C_i\) may either be prescribed or be determined via experimentation or neural network training.

The inverse problem seeks to find an \(X_1\) that outputs an \(X_4\) as close to \(X_{\text{target}}\) as possible, i.e. minimizing \(\hat{f}(X_4)\), using regularization to combat noise and overfitting when necessary. Using classical optimization, this requires differentiation that can be expressed as

\[
\frac{\partial \hat{f}}{\partial X_1} = \frac{\partial \hat{f}}{\partial X_4} \frac{\partial f_3 (X_3, C_3)}{\partial X_3} \frac{\partial f_2 (X_2, C_2)}{\partial X_2} \frac{\partial f_1 (X_1, C_1)}{\partial X_1},
\]

implying that every function layer requires differentiability with respect to its inputs. Now suppose that \(f_2 (X_2; C_2)\) represented a neural network layer that needs to be trained in order to ascertain reasonable parameters \(C_2\). In order to do this, one would consider a large number \(K\) of known training pairs of the form \((X^k_1; X^k_{\text{target}})\); however, notationally, one may stack all the training pairs into a single \(X_1\) and \(X_{\text{target}}\), at least conceptually (for the sake of exposition). Then the required differentiation is

\[
\frac{\partial \hat{f}(X_4)}{\partial C_2} = \frac{\partial \hat{f}}{\partial X_4} \frac{\partial f_3 (X_3, C_3)}{\partial X_3} \frac{\partial f_2 (X_2, C_2)}{\partial C_2},
\]

highlighting the notable differences as compared to an inverse problem. Firstly, any pre-process, such as \(f_1\) here, does not require differentiability and can utilize any known procedural methods. In fact, one might use an \(f_1\) based on first principles aiming to solve the problem outright, and then supplement the results with the composition of \(f_2\) and \(f_3\) in order to better match real-world data. This means that data-driven neural network approaches may be added on top of any existing codebase, whether it is differentiable or not. Secondly, any post-process for the neural network, such as \(f_3\), only requires as much differentiability as would be required for \(f_3\) if it were included in a typical inverse problem. Thirdly, the neural network itself, \(f_2\), does not require the usual differentiability inherent to inverse problems, but only requires differentiability with respect to its parameters \(C_2\).

Most facial pipelines take as input a set of parameters that govern the shape/geometry of a three-dimensional face, as given by triangle vertex positions. For example, a blendshape facial rig (see e.g. \([84]\)) describes how a face is deformed from a neutral rest state \(n\) in terms of a linear combination of basis facial shapes, e.g. semantic basis vectors which represent particular expressions such as “smile” or “yawn.” The basis facial shapes are often acquired using dense performance capture (see e.g. \([9, 10, 20, 50]\)) or via sculpting by an artist/modeler \([30, 75]\). A typical high-quality blendshape rig contains hundreds of basis shapes corresponding to different expressions between which one can interpolate (see e.g. \([29]\)). Once a blendshape model is obtained, stacking each blendshape into a column of a matrix \(B\) allows one to define the facial geometry as \(n + Bb(w)\), where the vector \(b\) contains a degree of freedom for each blendshape and \(w\) represents a set of meaningful controls (\(b(w)\) may be nonlinear, but should be smooth). In order to
avoid linearized rotation artifacts due to rotational jaw motion [32, 121, 143], one typically hybridizes the linear blendshape system with skinning/enveloping (see e.g. [96, 76]), which blends together the nonlinear six-degree-of-freedom rigid body transformation from the skull and jaw. Each triangle vertex is assigned weights that dictate the relative influence of the skull and jaw such that vertices far from the jaw move with the skull, vertices far from the skull move with the jaw, and vertices in between move in a blended fashion. This can be written compactly as a matrix \( T(j(w)) \), where the controls \( w \) drive the six-degree-of-freedom rigid body offset \( j \) of the jaw from the skull and \( T \) assembles all the transformations and weights so that one may write

\[
x(w) = T(j(w))(n + Bb(w)),
\]

where \( x(w) \) are the triangle vertex positions of the face surface. Importantly, as long as the dependencies in Equation 3 are chosen carefully (in a smooth enough manner), then \( x \) is differentiable with respect to \( w \).

As an alternative to blendshape approaches, one can construct an anatomically motivated finite element facial model based on soft tissue, musculature, and underlying skeletal structures (see e.g. [121, 122]). In [121], the vertex positions are differentiable as a function of the muscle activations and jaw parameters, and the authors used this differentiability to solve inverse problems. However, since anatomical facial models rely on MRI, CT scans, etc., it is difficult to make an accurate model; therefore, [121, 122] struggled to express the wide variety of shapes possible with a blendshape system. Thus, [31] augmented the results of [121] using a three-dimensional morphing process in order to derive target locations for muscles. Although the method proposed in [31] regains the expressivity of a blendshape system, their morphing process lacked differentiability. Later, [7] noted that the morphing process could be made differentiable, but that this would require a mapping from each surface vertex to all other affected vertices in the simulation mesh, which is quadratic complexity and thus impractical. Instead, [7] parameterized the morph with a standard blendshape system, so that the parameters \( b \) drive the morph, resulting in linear complexity. This was implemented (in [7]) by simulating the anatomical mesh for each blendshape (using the morphing from [31]) in order to create the muscle shapes needed in order to define a blendshape system for the muscles themselves; then, the three-dimensional target shape of each muscle can be specified by the parameters \( w \), with each muscle tetrahedron vertex \( x(w) \) defined along the lines of Equation 3. Manipulating \( w \) determines a blendshape for each muscle, which is then targeted with the anatomical finite element simulation from [31] and [121], see Figure 2. Notably, the resulting scheme is fully differentiable and hence can be used to solve inverse problems.

Figure 2: (Left) Skull and jaw (gray) with anatomical muscle shapes (red). (Right) Corresponding surface of the tetrahedral finite element mesh simulated from the targeted muscle shapes.

Given target geometry for the three-dimensional face surface, one can specify an energy that minimizes the distance between the target and the parameterized model, and then solve an inverse problem for the parameters \( w \) that drive the \( b \) and \( j \) for the muscle blendshape system, which in turn drives the quasistatic finite element simulation of [121] augmented by [31] in order to match the target (see [7] for details). In order to match a two-dimensional RGB image, one needs to render the resulting geometry with a differentiable renderer along the lines of [88, 91] and utilize an energy that considers the difference in pixel colors. Then, one can solve an inverse problem for the controls \( w \) that drive the blendshape muscles which in turn drive the
finite element simulation which results in the surface mesh that is rendered into pixel colors that minimize the energy. Unfortunately, as shown in Figure 3 Left, differentiable renderers don’t typically have the same quality as a photorealistic renderer or photograph, so aiming to match pixel colors is overly optimistic. To overcome this limitation, [6] proposed processing both the image and the differentiable render with a pre-existing/widespread face landmark detector neural network, such as 2D/3D-FAN [24] (see Figure 3). These networks were trained with vast amounts of hand-labeled data so that they could find keypoint/landmark positions from images regardless of texture, geometry, shading, lighting, shadows, etc. As such, the poor rendering quality of a differentiable renderer is also serendipitously ignored by these neural networks. In summary, [6] utilizes an energy that computes the difference between keypoints/landmarks, and solving the inverse problem requires differentiating through the keypoint detector neural network (2D/3D-FAN), the differentiable renderer, and the quasistatic finite element muscle simulation.

Figure 3: Results of a machine-learning based facial keypoint detector such as 2D/3D-FAN [24] on a synthetic render (left) as well as the corresponding photograph (right).

Figure 4: (Left) A pose of the hybridized muscle system of [7] depicted on top of the corresponding target RGB photograph. Whiter muscle shapes correspond to more activated muscles. (Middle) The corresponding blendshape weights. (Right) The corresponding muscle activations.

Figure 4 illustrates results typical of this process. Figure 4 (Middle) shows the value of $b$ along the vertical axis for each blendshape on the horizontal axis. In spite of the expression in Figure 4 (Left) being not that
complex, many blendshapes have non-zero values; worse yet, successive frames in a video produce noisy
uncorrelated blendshape values that are hard to interpret as meaningful semantic information. In contrast,
Figure 4 (Right) illustrates that the muscle activations are sparser and more indicative of the image; in
fact, successive frames tend to be highly correlated, allowing one to separate semantic information from
noise. Generally speaking, sparse semantic solutions to inverse problems are obviously preferred over dense,
optimization minimizing a cost function
3. Optimization Framework

Whether it be the search for viable inputs for an inverse problem, minimizing some measure of effort for
a control problem, or the determination of network architecture parameters that allow a neural network to
well-match training data, these problems all take the form of an optimization

\[
\hat{f}(c) \quad \text{over parameters } c.
\]

Importantly, one typically has certain conditions in mind to which \( c \) should be
subject. For example, one might want \( c \) close to a prior/initial guess, one might desire the norm of \( c \) to
be small, and/or one might want \( c \) sparse so that it carries interpretable semantic meaning. In particular,
as noted above, it would be useful if neural network training resulted in a sparse \( c \) in order to identify
unnecessary/unimportant components of the network architecture.

Either in the absence of constraints or with constraints and suitable Lagrange multipliers, the minima of
\( \hat{f} \) occur at critical points where the (column vector) Jacobian \( F(c) = J^T_f(c) = \nabla \hat{f}(c) = 0 \). Since \( F(c) = 0 \)
is (generally) a nonlinear system of equations, one typically linearizes the system by taking the first two
terms of the Taylor expansion about a point \( c^* \), \( F(c) \approx F(c^*) + F'(c)(c - c^*) \) where \( F'(c) = J_F(c) = H^T_f(c) \)
is the transpose of the Hessian of \( \hat{f} \). Newton’s method uses this relationship to write \( F(c^{q+1}) - F(c^q) = 

F'(c^q) \Delta c^q \), where \( \Delta c^q = c^{q+1} - c^q \) and \( q \) represents the current iteration. Then, one solves the linear system
\( F'(c^q) \Delta c^q = \beta F(c^q) - F(c^q) \) to update \( c^{q+1} = c^q + \Delta c^q \) where \( \beta \in [0, 1) \), and using \( \beta \neq 0 \) more slowly
shrinks \( F(c^q) \) towards 0. Alternatively, one can utilize \( \Delta c^q \) merely as a search direction and subsequently
employ a number of one-dimensional approaches, e.g. bisection search, golden section search, etc.

While Newton’s method and similar techniques are reasonably well-justified and often converge well
in practice, they depend on access to various derivatives of the cost function \( f(c) \). To compute these
derivatives, one may utilize symbolic/analytic differentiation, finite differences, or automatic differentiation
(e.g. backpropagation). Automatic differentiation is often preferred in the context of training neural networks
both because of its ease of implementation as well as its availability via various software packages (e.g.
Tensorflow [1], Caffe [71], PyTorch [110], Theano [127], etc.); however, roundoff and other errors generally
accumulate proportional to the size of the network, which can turn out to be numerically catastrophic.
Moreover, high dimensionality makes the computation and storage of \( H^T_f \) impractical, and thus practitioners
typically resort to quasi-Newton methods that aim to avoid direct consideration of second derivatives.

Broyden’s method [21] for solving nonlinear systems, in the context of optimization, first approximates
\( (H^T_f)^0 = I \), and then iteratively uses rank-one updates aiming for successively better estimates. Each
iteration, one solves \( (H^T_f)^q \Delta c^q = -J^T_f(c^q) \) to find a search direction \( \Delta c^q \), and then uses line search to find
\( c^{q+1} \); subsequently, \( \Delta c^q \) is updated via \( \Delta c^q = c^{q+1} - c^q \). Given \( (\Delta J^T_f)^q = J^T_f(c^{q+1}) - J^T_f(c^q) \), the rank-one
update is

\[
(H^T_f)^{q+1} = (H^T_f)^q + \frac{1}{(\Delta c^q)^T \Delta c^q} (\Delta J^T_f)^q - (H^T_f)^q \Delta c^q \Delta c^q)^T
\]

so that \( (H^T_f)^{q+1} \Delta c^q = (\Delta J^T_f)^q \). When \( c \) is of large dimension, forming and inverting the dense \( O(n^2) \)
\( H^T_f \) is undesirable, especially considering that the approximation is built from rank-one updates, and thus
a matrix-free approach to the action of \( H^{-T}_f \) on a vector is preferred. That is, \( \Delta c^q = -(H^T_f)^q J^T_f(c^q) \) is
used to find the search direction for the line search used to determine \( c^{q+1} \), which is used to update \( \Delta c^q \) and
\[ \left( \Delta J_f^T \right)^q \]; then, rank-one update for \( H_f^{-T} \) is

\[
\left( H_f^{-T} \right)^{q+1} = \left( H_f^{-T} \right)^q + \frac{\left( \Delta c^q - \left( H_f^{-T} \right)^q \left( \Delta J_f^T \right)^q \right) \left( \Delta c^q \right)^T \left( H_f^{-T} \right)^q}{\left( \Delta c^q \right)^T \left( H_f^{-T} \right)^q \left( \Delta J_f^T \right)^q}.
\] (5)

so that \( \left( H_f^{-T} \right)^{q+1} \left( \Delta J_f^T \right)^q = \Delta c^q \). Other low-rank update methods such as SR1 \([33, 22]\), DFP \([33, 46]\), and BFGS \([23, 44, 52, 119]\) are similar in spirit. In particular, the limited-memory L-BFGS \([106]\) only stores the past several low-rank updates making it quite efficient, see e.g. \([78, 34]\).

Instead of performing rank-one updates to improve upon \( \left( H_f^T \right)^0 = I \) as in Broyden-style methods, gradient descent methods simply use \( H_f^T = I \) so that the search direction is obtained trivially via \( \Delta c^q = -J_f^T (c^q) = -\nabla \hat{f} (c^q) \). When problems have high dimensionality, practitioners often make further simplifications such as evaluating only a subset of the right-hand side (mini-batch gradient descent) or even just one or a few randomly-selected entries at a time (SGD), see e.g. \([117, 18]\). One can even ignore the search direction equation entirely by choosing \( \Delta c^q \) to be various basis vectors, i.e. coordinate descent \([120]\).

Furthermore, gradient descent methods can be envisioned as forward Euler approximations of gradient flow, i.e. of \( \frac{dc}{dt} = -\nabla \hat{f} (c(t)) \), which allows for the wealth of knowledge in designing and solving ordinary differential equations to be utilized. For instance, adaptive time stepping leads to such techniques as AdaGrad \([37]\), which utilizes separate learning rates (time steps) for each parameter, or AdaDelta \([139]\) and RMSprop \([129]\), both of which lessen the effects of history terms in AdaGrad in order to maintain a sufficiently positive learning rate to avoid stalling. Incorporating the effects of prior search directions and state updates can be seen as utilizing momentum, which rewrites gradient flow using Newton’s Second Law \([114]\). The Adam learning rate to avoid stalling. Incorporating the effects of prior search directions and state updates can be seen as utilizing momentum, which rewrites gradient flow using Newton’s Second Law \([114]\). The Adam method \([72]\) combines the notion of using a moving average of gradients as in momentum methods with an adaptive learning rate for each parameter. The 52,000+ citations\(^2\) of \([72]\) indicate the success practitioners have enjoyed with Adam, often finding that it converges faster than SGD.

4. Iterative Levenberg-Marquardt

When training a neural network on data \((x_i, y_i)\), one seeks to find the parameters \( c \) of a generally vector-valued function \( f(x, y, c) \) that minimize error over the training data, i.e. one desires \( \| f(x_i, y_i, c) \| \) to be close to zero for all \( i \). Choosing the \( L_2 \) norm leads to minimizing \( \hat{f}(c) = \frac{1}{2} \sum_i f(x_i, y_i, c)^T f(x_i, y_i, c) = \frac{1}{2} \hat{f}(c)^T \hat{f}(c), \) which is a nonlinear least squares problem \([14]\). Critical points have \( J_f^T (c) \hat{f}(c) = 0, \) which can be rewritten using the Taylor expansion of \( \hat{f}(c) \) about \( c^q \) as \( J_f^T (c^q) \left( \hat{f}(c^q) + J_f (c^q) \Delta c^q + \cdots \right) = 0, \) where \( \Delta c^q = c - c^q \). Dropping high-order terms and evaluating \( J_f^T \) at \( c^q \) leads to the Gauss-Newton equations

\[
J_f^T (c^q) J_f (c^q) \Delta c^q \approx -J_f^T (c^q) \hat{f}(c^q), \) which imply an estimate of \( H_f^T (c^q) = J_f^T (c^q) J_f (c^q), \) see e.g. \([107]\).

Notably, the Gauss-Newton approximation to the Hessian only requires first derivatives. Moreover, since the Gauss-Newton equations are the normal equations for \( J_f (c^q) \Delta c^q = -\hat{f}(c^q), \) one can obtain \( \Delta c^q \) via any least squares and minimum norm approach for solving this much better conditioned set of equations.

When \( J_f \) is poorly-conditioned or rank-deficient, one can regularize the Gauss-Newton equations via

\[
\left( J_f^T (c^q) J_f (c^q) + \epsilon^2 I \right) \Delta c^q = -J_f^T (c^q) \hat{f}(c^q) \] with \( \epsilon > 0 \), which is referred to as Levenberg-Marquardt or damped nonlinear least squares, see e.g. \([83, 97, 107, 14]\). This makes a tradeoff between solvability and accuracy, since the unique and least squares components of the solution will be perturbed away from their correct values. To illustrate this, consider stacking a general linear system \( Ac = b \) with the full-rank \( \epsilon I c = 0 \) to obtain

\[
\left( \begin{array}{c} A \\ \epsilon I \end{array} \right) c = \left( \begin{array}{c} b \\ 0 \end{array} \right),
\] (6)

\(^2\)as of September 2020, according to Google Scholar
which has equivalent normal equations of \((A^T A + \epsilon^2 I) c = A^T b\). Using the SVD, \(A = U \Sigma V^T\), this becomes
\[
(\Sigma^T \Sigma + \epsilon^2 I) \hat{c} = \Sigma^T \hat{b}
\]
where \(\hat{c} = V^T c\) and \(\hat{b} = U^T b\). For a general \(A\), \(\Sigma\) has the form
\[
\Sigma = \begin{pmatrix}
\hat{\Sigma} & 0 \\
0 & 0
\end{pmatrix},
\]
where \(\hat{\Sigma}\) is diagonal and full-rank. This leads to
\[
\begin{pmatrix}
\hat{\Sigma}^T & 0 \\
0 & 0
\end{pmatrix} \begin{pmatrix}
\hat{\Sigma} & 0 \\
0 & 0
\end{pmatrix} + \epsilon^2 I
\begin{pmatrix}
\hat{c}_r \\
\hat{c}_z
\end{pmatrix} = \begin{pmatrix}
\hat{\Sigma}^T \hat{b}_r \\
0
\end{pmatrix},
\]
where \(\hat{c}\) and \(\hat{b}\) have been decomposed to separate out the portions that correspond to identically-zero sub-matrices of \(\Sigma^T\). Equation 8 sets \(\hat{c}_z\) identically equal to zero as desired (i.e. minimum norm solution), but the entries in \(\hat{c}_r\) are determined via
\[
\hat{c}_k = \frac{\sigma_k}{\sigma_k^2 + \epsilon^2} \hat{b}_k = \left(\frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2}\right) \frac{\hat{b}_k}{\sigma_k},
\]
perturbing them away from their correct unique or least squares solution \(\hat{c}_k = \hat{b}_k/\sigma_k\). This perturbation is negligible for \(\sigma_k \gg \epsilon\), but smaller \(\sigma_k\) have their associated \(\hat{c}_k\) more significantly incorrectly perturbed toward zero. One typically chooses \(\epsilon\) so that it does not interfere too much with the larger (more important) singular values, while still being large enough to regularize numerical issues associated with smaller \(\sigma_k\) (as well as identically zero singular values). As a side note for weighted least squares, one adds the full-rank \(\epsilon D c\) to obtain a modified version of Equation 6, which after column scaling becomes
\[
(AD^{-1}) D c = \begin{pmatrix} b \\ 0 \end{pmatrix}.
\]

Then, a simple renaming of variables results in the original Equation 6, and the above analysis applies without modification.

Motivated by the Broyden-style iterative methods (discussed in the previous section) which began with a simple guess for the Hessian and then corrected it after each iteration, we propose a similar strategy for Levenberg-Marquardt. That is, we start with \(\epsilon I c = 0\) but subsequently update the right-hand side as the iteration proceeds, progressively removing the erroneous perturbation of the least squares solution shown in Equation 9. Our approach converges to the exact solution for larger singular values, as for example would also be achieved using PCA; however, unlike the all-or-nothing approach of PCA, our approach smoothly tapers between the exact solution for larger \(\sigma_k\) and robust regularization for smaller \(\sigma_k\).

We start with a guess \(c^*\) for \(c\) and stack \(A c = b\) with \(\epsilon I c = \epsilon c^*\) leading to the normal equations
\[
(A^T A + \epsilon^2 I) c = A^T b + \epsilon^2 c^*.
\]

Substituting the SVD of \(A\) leads to
\[
(\Sigma^T \Sigma + \epsilon^2 I) \hat{c} = \Sigma^T \hat{b} + \epsilon^2 V^T c^* = \Sigma^T \hat{b} + \epsilon^2 \hat{c}^*.
\]
where \(\hat{c}^* = V^T c^*\). This modified version of Equation 8 sets \(\hat{c}_z\) equal to \(\hat{c}^*_z\), while the entries in \(\hat{c}_r\) are determined via
\[
\hat{c}_k = \left(\frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2}\right) \frac{\hat{b}_k}{\sigma_k} + \left(\frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\right) \hat{c}^*_k
\]
illustrating that \(\hat{c}_k\) is a convex combination of the exact solution \(\hat{b}_k/\sigma_k\) and the initial guess \(\hat{c}^*_k\). When \(\sigma_k \gg \epsilon\), the associated \(\hat{c}_k\) tend toward the correct solution as usual. When \(\sigma_k \ll \epsilon\), the associated \(\hat{c}_k\) tend toward \(\hat{c}^*_k\).

---

\[\text{This method/proof was derived for a CS205L lecture at Stanford in Winter quarter 2019 [41].}\]
Starting with a guess of \(c^* = 0\), one obtains \(\hat{c}^* = 0\) and thus \(\hat{c}_z = 0\) as desired. In addition, \(\hat{c}^*_c = 0\) and Equation 13 is identical to Equation 9. Multiplying by \(V\) transforms \(\hat{c}\) back to the \(c\) that would result from solving Equation 11. Setting \(c^* = \) equal to this newly obtained value of \(c\) and repeating the above analysis maintains \(\hat{c}_z = 0\) (as desired), while

\[
\hat{c}_k = \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \frac{\hat{b}_k}{\sigma_k} \tag{14}
\]

so that Equation 13 becomes

\[
\hat{c}_k = \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \hat{b}_k + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} = \left(1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} \tag{15}
\]

Repeating the entire process again results in

\[
\hat{c}_k = \left(1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} \tag{16}
\]

and further iterations give

\[
\hat{c}_k = \left(1 + \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right) \left( \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \right) \frac{\hat{b}_k}{\sigma_k} \tag{17}
\]

where the term in parentheses is a geometric series with \(r = \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2}\).

Since the geometric series in Equation 17 converges to \(\frac{1}{1-r} = \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2}\). Equation 17 converges to the exact solution \(\hat{c}_k = \frac{\hat{b}_k}{\sigma_k}\). Any practical numerical method will only take \(q\) steps, leading to the partial sum

\[
1 - r^q = \frac{\sigma_k^2}{\sigma_k^2 + \epsilon^2} \left(1 - \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right)^q, \tag{18}
\]

which yields

\[
\hat{c}_k = \left(1 - \left( \frac{\epsilon^2}{\sigma_k^2 + \epsilon^2} \right) \right)^q \frac{\hat{b}_k}{\sigma_k}. \tag{19}
\]

The scalar term premultiplying \(\frac{\hat{b}_k}{\sigma_k}\) monotonically approaches 1 as the iteration proceeds, and thus each \(\hat{c}_k\) converges monotonically to the exact solution and converges more quickly for larger \(\sigma_k\) as desired.

### 4.1. Examples

Typical inverse, control, and learning problems involve numerically challenging data, where linear sub-problems may have coefficient matrices with both small and identically zero singular values and the right-hand side may not be in the range of the coefficient matrix. Accordingly, we evaluate our approach against these types of problems. Here, we compare our iterative Levenberg-Marquardt (iLM) approach to PCA because PCA is a widely applied and well-understood algorithm. Since our goal is merely to demonstrate the feasibility of iLM, we utilize straightforward Matlab implementations of both methods. For iLM, we solve Equation 11 using Matlab’s \texttt{pcg} routine with no preconditioner, i.e. conjugate gradients. For PCA, we compute the largest singular values and corresponding singular vectors using Matlab’s \texttt{svds} function, which finds these quantities via either Lanczos bidiagonalization [5, 77] or a computation of the full SVD depending on the number of singular vectors desired. All experiments were run on a workstation equipped with Matlab R2020a, 128GB RAM, and a 24-core Intel CPU running at 3.00GHz.

First we consider rather large dense matrices and compare PCA and iLM for a varying number of singular values, noting that an increased number of iLM iterations is required for increased accuracy. We generate
The aforementioned tests are unfair to iLM because they stringently require iLM to do as well as PCA on the values PCA estimates nearly exactly while ignoring the fact that PCA obtains totally inaccurate (identically zero) solutions for all the other $\hat{c}_k$. To illustrate the added benefit of smooth convergence obtained via iLM, we construct a small (to make the graphs easier to read) $100 \times 100$ random matrix with ten of its singular values set to zero (see Figure 9). A random right-hand side $b$ outside the range of $A$ is then formed. Figures 10 and 11 show the results of iLM and PCA, illustrating how well the obtained $\sigma_k \hat{c}_k$ reconstruct the projected right-hand side $\hat{b}_k$. iLM leverages rich information about the structure of $A$ even when $\epsilon$ is larger than the largest singular value of $A$ (substantial regularization). In these experiments we used a CG tolerance of $1e-6$ and a maximum of 1000 CG iterations.

---

4 We observed that, by default, Matlab seems to wait too long to switch from Lanczos bidiagonalization to computing the full SVD.
Figure 6: Number of iterations required for iLM for the medians of the trials in Figure 5 (Right) with $\epsilon = 5.0$. (When $\epsilon = 0.1$, iLM mostly converges to the desired tolerance in one iteration.)

Figure 7: Using iLM, as more $\hat{c}_k$ are sought or as more accuracy is desired, a tighter CG tolerance needs to be used to prevent convergence from stalling. Plotted are the experimentally-determined maximum CG tolerances which yielded convergent results, ranging from 100 to 4500 $\hat{c}_k$ sought. (Left) $\epsilon = 0.1$. (Right) $\epsilon = 5.0$. 
Figure 8: Performance of iLM and PCA for estimating a fixed number of $\tilde{c}_k$ (500 of them) given a dense $n \times n$ square matrix with $n^2$ randomly-generated entries (post-processed to have 100 zero singular values) and a randomly-generated right-hand side not in its range. (Left) $\epsilon = 0.1$. Only one iLM iteration is required for these trials, and hence the growing cost is a combination of the increased cost per CG iteration and the number of CG iterations required. (Right) $\epsilon = 5.0$. For large $n$, the added regularization appears to aid in the convergence of CG.

Figure 9: The singular values of the matrix used for the experiments shown in Figures 10 and 11. Ten of the singular values are identically zero.
4.1.1. Comparisons for Nonlinear Optimization Problems

The above examples demonstrate the utility of iLM when solving linear problems such as those that arise on each iteration of a standard nonlinear optimization algorithm. We now explicitly consider solving nonlinear optimization problems using iLM and related approaches.
We first consider an objective $f(x_1, x_2) = x_1^2 + 5x_2^2 - 4$ with an initial guess of $x^0 = (-3, -4)^T$. Figure 12 shows the results of using Newton’s method, gradient descent, Levenberg-Marquardt, Fan’s modified Levenberg-Marquardt [40], and iLM to solve this problem. Each method is allowed to run until either the objective at the current iterate is within $10^{-6}$ of the analytic minimum value or until the $L_2$ norm of the iterate changes less than $10^{-6}$ between iterations. All CG solves use a tolerance of $10^{-6}$ and a maximum of 1,000 iterations. Since the objective is quadratic, Newton’s method converges to the unique, global minimum in one iteration. Gradient descent, which lacks second-derivative information, oscillates around

Figure 11: Same as Figure 10 except with $\epsilon = 5.0$. The increased regularization slows down iLM convergence, as expected.
the (non-uniformly-scaled) energy landscape before eventually reaching the minimum. A learning rate of 0.15 was used. Levenberg-Marquardt can be seen as blending between the Newton and gradient descent iterates. With little regularization ($\epsilon = 0.1$), Levenberg-Marquardt looks similar to Newton’s method, while with a large regularization parameter ($\epsilon = 100.0$), the number of iterations required for convergence significantly increases. The modified Levenberg-Marquardt of [40] can offer cubic convergence rates under suitable conditions by performing essentially two Levenberg-Marquardt steps on each iteration (a standard step and a forward-looking step based on the standard step). We implemented this method using the same parameters as in Section 4 of [40], except we used an initial $\mu_0$ of 10 in order to be similar to our regularization of the other Levenberg-Marquardt variants. Finally, we consider iLM using 1, 10, or 100 iterations, all with $\epsilon = 10.0$ and using an initial guess of $c^* = 0$. Note that iLM uses an $\epsilon^2$ rather than an $\epsilon$ scaling of the identity term, so $\epsilon = 10.0$ is equivalent to $\epsilon = 100.0$ with Levenberg-Marquardt. iLM converges to the Newton step (when the Newton step is defined) as the number of iterations increases, so iLM has a quadratic order of convergence in the best case; though of course, like Levenberg-Marquardt, gradient descent, etc., it is possible to design parameters and scenarios which make iLM converge poorly or not at all. Moreover, we stress that various strategies for adaptive learning rates and adaptive regularization terms may improve the performance of these methods. In particular, the adaptive parameter values used for our implementation of [40] are quite useful for aiding the convergence of the method, and in practice one would want to utilize adaptive regularization schemes for Levenberg-Marquardt and iLM as well (which would likely remove many of the small steps those algorithms take as they approach the solution).

As a potentially greater challenge, we consider adding a third coordinate to our objective. We alter our initial guess to have a value of 1 along this direction. Since the objective function does not depend on this third component, it is possible for the solution iterate to drift along this additional axis, e.g. when regularization is perturbing the solution away from the true minimum. Newton’s method is not applicable in this case since the Hessian becomes singular, although iLM appears to converge to what Newton’s method would compute using the Hessian’s pseudoinverse. Interestingly, all methods appear to converge to the solution $(0 \ 0 \ 1)^T$, rather than e.g. the minimum norm solution at the origin. We also consider rotating the objective and initial guess by 30 degrees about the $x_1$ and $x_3$ axes in order to make the null space of the Hessian less obvious. However, the optimization methods we tested still reach the minimum in the same number of iterations as reported in Figure 12, except for Newton’s method, which remains undefined.

Further differences in the behavior and convergence of these optimization algorithms can be elucidated by considering the slightly modified objective $f(x) = \min \left[ x_1^2 + 5x_2^2 - 4, (x_1 + 1)^2 + 5(x_2 - .1)^2 - 4 \right]$, which has minima at $(0 \ 0 \ 1)^T$ and $(-1 \ 1 \ .1)^T$. With the same initial guess of $x^0 = (-3 \ -4 \ -4)^T$, the nearest minimum in the $L_2$ norm is the minimum-norm solution $(0 \ 0)^T$. However, if an algorithm does not proceed directly towards this solution, it may instead converge towards the other minimum with greater norm and less sparsity. This is demonstrated in Figure 13. Newton’s method converges in one step to $(0 \ 0)^T$. With enough iterations, iLM approximates the Newton step and also selects the minimum-norm solution. With fewer iterations, though, iLM behaves more like gradient descent and Fan’s modified Levenberg-Marquardt, which select the non-zero minimum. Standard Levenberg-Marquardt can be driven to select different minima by tuning the regularization parameter. In general, one must consider the types of solutions one seeks to an optimization problem (e.g., minimum-norm or sparse solutions) when selecting an algorithm and its parameters. Practical considerations like this and real-world performance tradeoffs can often overshadow theoretical convergence guarantees, as seen for example with the continued ubiquity of (stochastic) gradient descent.

5. Column Space Search

For the sake of motivation, consider the $2 \times 2$ linear subproblem $Ac = b$ where $A = \begin{bmatrix} 1 & -1 \\ .1 & 1 \times 10^{-6} \end{bmatrix}$ and $b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}^T$. Although the right-hand side is in the range of $A$, it is not “easily” in the range of $A$; in other words, the columns of $A$ are mostly orthogonal to $b$ leading to a solution that utilizes large multipliers $c_1 = 1/(1 \times 10^{-6} + .1)$ and $c_2 = 1/(1 \times 10^{-6} + .1)$ on the columns of $A$. See Figure 14. Even though this is the exact solution to the linear subproblem, it misleadingly heavily weights columns of $A$ that do not correlate well with the desired $b$. Large values of $c_1$ and $c_2$ seemingly indicate that those columns of $A$ are
Figure 12: Different algorithms applied to minimizing $f(x_1, x_2) = x_1^2 + 5x_2^2 - 4$ with an initial guess of $x^0 = (-3 - 4)^T$. Contours of the function are drawn and shaded by contour value. Arrows indicate steps taken on each iteration of the optimization as the algorithm is allowed to converge to $(0 0)^T$ (the black x). The number of iterations required for each method to converge to a tolerance of $10^{-6}$ is reported. We emphasize that these methods have different computational requirements; for instance, a Levenberg-Marquardt (LM) step requires one linear solve, while an iteration of [40] requires two.
(a) Newton’s method: 1 iteration  
(b) Gradient descent: 23 iterations  
(c) Levenberg-Marquardt ($\epsilon = 0.1$): 3 iterations  
(d) Levenberg-Marquardt ($\epsilon = 100$): 204 iterations  
(e) Modified LM [40]: 5 iterations  
(f) iLM (1 iteration): 204 iterations  
(g) iLM (10 iterations): 21 iterations  
(h) iLM (100 iterations): 3 iterations

Figure 13: Repeating the experiment of Figure 12 with an objective of $f(x_1, x_2) = \min (x_1^2 + 5x_2^2 - 4, (x_1 + .1)^2 + 5(x_2 - .1)^2 - 4)$. Algorithms converge to either $(0 \ 0)^T$ (the black x) or to $(-.1 \ .1)^T$ (the red star).
important, even though they mostly cancel each other out being nearly orthogonal to the right-hand side. The regularized least squares problem \( \min_c \| b - Ac \|_2^2 + \lambda \|c\|_2^2 \) reduces the values of \( c_1 \) and \( c_2 \), although it does not alleviate the fact that these columns mostly work to cancel each other out, making minimal progress towards \( b \). At best, heavy regularization could drive \( c_1 \) and \( c_2 \) even further towards zero.

As previously discussed, the columns of the linear subproblem are often quite erroneous approximations to the Hessian, which itself is only a linearization of the nonlinear problem; yet the linear subproblem is often solved and used to increment the solution vector (i.e. via \( c^{t+1} = c^t + \Delta c^t \)). The original nonlinear problem may include significant noise and heavy regularization, and thus it seems more important to focus on controls that make direct progress towards energy/loss minimization than those that make only incidental progress while competing with and largely cancelling each other. Thus, we advocate dropping parameters from consideration when the gains made toward the solution by some combination of those parameters are incidental compared to the parameters’ main actions. As discussed previously, in regard to neural networks, this allows one to identify and differentiate which building blocks of the neural network are more or less important than others. In order to identify the more important parameters, we make note of two common misconceptions/flaws in the pursuit of solving linear subproblems. First, solving the linear subproblem exactly is not necessarily desirable since the columns of \( A \) may be terrible approximations to those of the Hessian, which itself is a linearization. Second, the largest singular values of \( A \) do not necessarily represent the most important features of the problem (as is assumed by typical PCA approaches); oftentimes, the more important notion is which columns of \( A \) are well-correlated with the right-hand side \( b \), allowing one to make clean, non-competitive progress toward the solution.

Next, consider the right-hand side \( b = [5 \ 1]^T \), which is better correlated with at least one of the columns of \( A \). See Figure 15. In this case, the exact solution in Figure 15b is an improvement over Figure 14 (Right), but still contains problematic cancellation. The regularized solution shown in Figure 15c makes more progress towards the solution as compared to Figure 14 (Right), except it uses a lot more of \( a_2 \) and a lot less of \( a_1 \) than one might expect given how much better correlated \( a_1 \) is with \( b \). Regularization damps the use of \( a_1 \) hindering its progress towards the solution; as such, \( a_2 \) ends up being utilized significantly. One could obtain a better solution for this example by changing the regularization in the least squares problem to have the form \( \min_c \| b - Ac \|_2^2 + \lambda_1 c_1^2 + \lambda_2 c_2^2 \) with \( \lambda_1 = 0 \). Figure 15d shows the result for \( \lambda_2 = 1 \) which is highly improved. One could do even better using only \( a_1 \) as shown in Figure 15e, obtained using \( \min_c \| b - a_1 c_1 \|_2^2 \).

For more discussion on various regularization strategies, especially pertaining to the facial expression inverse problem described in Section 2, see [15, 25, 86, 128, 13, 64, 136, 63, 87, 20, 68, 105].

The aforementioned discussion motivates the notion of choosing only the columns of \( A \) which are most correlated with \( b \). Such an approach can be implemented one column at a time using a basic coordinate descent algorithm [111]. Importantly, this allows one to circumvent null spaces without adding regularization, making coordinate descent an attractive option for use on ill-posed, poorly-conditioned problems. At each iteration, the column can be chosen stochastically [104] or deterministically. Popular deterministic methods for choosing the next search direction include cyclic coordinate descent [74], the Gauss-Southwell (GS) and Gauss-Southwell-Lipschitz rule [108], and the maximum block improvement (MBI) rule [28]. Instead of looking at a single column at a time, block coordinate descent can be used to update multiple columns simultaneously [130]; however, regularization may still be needed when the block of columns is poorly-conditioned or does not have full rank. See [120, 135] for more discussion. Typical coordinate descent algorithms may choose a large number of poorly correlated coordinates in place of a smaller number of more strongly correlated coordinates. Using correlation to choose the next coordinate to add to the model can alleviate this problem and is the central idea behind MBI [28], forward and backward stepwise regression [35], and LARS [38]. The latter statistical regression methods are often used to gain better prediction accuracy and interpretability of the model [59]. However, LARS converges to the least squares solution of the linear subproblem [38] because it eventually uses uncorrelated coordinates.

In order to facilitate our goal of obtaining sparse, semantic solutions to optimization problems, particularly without adding unnecessary heuristic regularization which can lead to overfitting and error, we propose\(^5\) solving linear subproblems by first pruning away any coordinates that are geometrically uncorrelated with the right-hand side as motivated by least angle regression (LARS) [38]; then, we estimate the remaining

---

\(^5\)This approach was first proposed in the following preprint: \[8\].
coordinates via coordinate descent, eliminating the need to regularize for solvability.

5.1. Pruning Geometrically Uncorrelated Directions

We illustrate our approach, hereafter referred to as Column Space Search (CSS), by again considering solving a generic nonlinear least squares optimization problem of the form \( \min ||f(x, y, c)||^2 \). Using a Gauss-Newton based method, every iteration of the optimization requires solving the linear subproblem \( J_f^T (\epsilon^t) J_f (\epsilon^t) \Delta \epsilon^t = -J_f^T (\epsilon^t) \hat{f} (\epsilon^t) \) to find the \( \Delta \epsilon^t \) subsequently used to make progress towards the solution. Again, one may equivalently consider \( J_f (\epsilon^t) \Delta \epsilon^t = -\hat{f} (\epsilon^t) \).

We first compute the geometric correlation between each column \( j_i \) of \( J_f (\epsilon^t) \) and the right-hand side \(-\hat{f} (\epsilon^t)\). Similar to LARS [38] and MBI [28], we use \(|\hat{j}_i \cdot \hat{f} (\epsilon^t)|\), where \( \hat{j}_i = j_i / \| j_i \|_2 \). Poorly geometrically correlated columns can only make significant progress towards the solution either when partially cancelled by other poorly geometrically correlated columns (as in Figure 14 (Right)) or as corrections to better geometrically correlated columns (as in Figure 15b). However, this so-called progress, while valid for the linear subproblem, may pollute the sparsity and semantics of the solution to the original nonlinear problem. See Figure 16. Thus, we prune poorly geometrically correlated columns from \( J_f (\epsilon^t) \) resulting in a lower-rank \( J_S \). Motivated by the Gauss-Southwell rule, one might instead prune using gain correlation \(|\hat{j}_i \cdot \hat{f} (\epsilon^t)|\), which considers large residual decreases with smaller variable values; however, we instead prefer removing poorly geometrically correlated columns even when they may have large gains as it seems to lead to better semantic interpretation. Additionally, one could drop the absolute value and consider \( \hat{j}_i \cdot \hat{f} (\epsilon^t) \) in order to prune columns that are only semantically sensible in one direction.

Pruning columns of \( J_f (\epsilon^t) \) to get a reduced \( J_S \) has the additional benefit of potentially eliminating portions of the null space of \( J_f (\epsilon^t) \), as the pruned out columns or a combination of them with the non-pruned columns may have linear dependencies; this pruning may also improve the condition number. This is especially prudent when working with a large number of dimensions, in which case the dimension of the

Figure 14: (Left) A visualization of the columns of \( A \) as well as \( b \) for the linear subproblem \( Ac = b \) from Section 5 when \( b = [0 \ 1]^T \). Note how the columns of \( A \) are mostly orthogonal to \( b \). (Right) The exact solution utilizes quite large values of \( c_1 \) and \( c_2 \), over-scaling largely competing columns of \( A \) in order to make progress towards \( b \). Since the columns of \( A \) are often poor approximations to the Hessian, and the Hessian itself is only a linearization of the nonlinear problem, it seems imprudent to over-utilize controls \( c_1 \) and \( c_2 \) in order to make progress towards \( b \). A regularized solution (with \( \lambda = 2 \)) is also shown in the figure. It does reduce the magnitudes of \( c_1 \) and \( c_2 \) but still demonstrates the same non-desirable competitive behavior between the columns.

Figure 15: (a) A visualization of \( A \)'s columns and \( b \) for the linear problem \( Ac = b \) from Section 5 when \( b = [5 \ 1]^T \). (b) The exact solution depicted by \( a_1c_1 \) and \( a_2c_2 \). (c) The regularized solution with \( \lambda = 1 \). (d) The regularized solution with \( \lambda_1 = 0 \) and \( \lambda_2 = 1 \). (e) The solution when solving for \( c_1 \) only. (f) The solution obtained after a few iterations of coordinate descent using the MBI selection rule.
null space of $\tilde{J}(c^g_0)$ and the condition number of $\tilde{J}(c^g_0)$ may be quite large. Moreover, these difficulties are exacerbated when regularization is not used.

### 5.2 Solving the Pruned System

We avoid regularization entirely in order to avoid changing the solution to the problem; thus, we pursue a coordinate descent strategy to solve $J_S \Delta c^g_j = -\tilde{f}(c^g)$ where $\Delta c^g_j$ is a subset of $\Delta c^g$. At each iteration, a single column $j_i$ of $J_S$ is used to make progress towards $-\tilde{f}(c^g)$. We generally only execute a few iterations to mimic the regularization effects of early stopping [53] and truncated-Newton methods [101], as it helps to prevent overfitting to the linearized subproblem or reaching the undesirable least squares solution as in LARS [38]. Furthermore, we also terminate early if the decrease in $L_2$ error is low.

Choosing the most geometrically correlated column $j_i$ (as in MBI [28]) allows one to best minimize the remaining residual; however, small-magnitude columns may require large, undesirable step sizes $\alpha(j_i)$ to make progress. Instead, motivated by the Gauss-Southwell rule [108], we choose the column $j_i$ that maximizes a discretized ratio of residual reduction to step size, i.e.

$$\frac{\Delta(r^T r)}{\Delta \alpha} = \frac{||r(\Delta c^g_j)||^2 - ||r(\Delta c^g_j) - \alpha(j_i)j_i||^2}{|\alpha(j_i)|}, \tag{20}$$

where $r(\Delta c^g_j)$ is the current residual as a function of the current estimate for $\Delta c^g_j$. In addition, $\alpha(j_i)$ is the step size obtained when choosing column $j_i$: Flipping all $j_i$ so that $r(\Delta c^g_j) j_i > 0$ leads to $\alpha(j_i) > 0$, which allows one to equivalently maximize

$$\begin{align*}
M &= \frac{r(\Delta c^g_j)^T r(\Delta c^g_j) - \left( r(\Delta c^g_j)^T r(\Delta c^g_j) - 2\alpha(j_i) r(\Delta c^g_j)^T j_i + (\alpha(j_i))^2 j_i^T j_i \right)}{\alpha(j_i)} \tag{21a} \\
&= 2r(\Delta c^g_j)^T j_i - \alpha(j_i) ||j_i||^2 \tag{21b} \\
&= r(\Delta c^g_j)^T j_i + (r(\Delta c^g_j) - \alpha(j_i) j_i)^T j_i. \tag{21c}
\end{align*}$$

The greedy choice of $\alpha(j_i)$ removes as much of the residual as possible, setting $\alpha(j_i) j_i = (r(\Delta c^g_j) \cdot j_i) \hat{j}_i$ or

$$\alpha(j_i) = (r(\Delta c^g_j) \cdot j_i) / ||j_i||_2^2, \tag{22}$$

which zeros out the second term in Equation 21c leaving only $r(\Delta c^g_j)^T j_i$, i.e. gain correlation.

There are two subtleties to consider regarding Equations 20–22. First, we do not necessarily use columns with the largest gains because, as discussed in Section 5.1, we prune away poorly geometrically correlated columns before considering Equations 20–22. Second, one typically limits the size of $\alpha(j_i)$ when training neural networks and/or solving optimization/control problems, see e.g. trust region methods [124, 45, 107], adaptive step sizes for temporal numerical integration [43, 47], and adaptive learning rate techniques such as Adam [72], ADADELTA [139], etc. Thus, shorter $j_i$ will not necessarily yield the greedy $\alpha(j_i)$ shown in Equation 22, leaving the second term in the last line of Equation 21c non-zero. See Figure 17.
Consider bounding the step size $\alpha(j)$ from above with some $\alpha_{\text{max}}$. One can choose a reference frame such that $r(\Delta c^q_S)$ is a unit vector along the $y$-axis and $j_i$ is in the first quadrant of the $xy$-plane, as shown in Figure 18. Referring to the greedy $\alpha(j)$ in Equation 22, we plot curves representing vectors $j_i$ where the greedy $\alpha(j)$ is equal to 1/1.5, 1, and 1/7.5 in the figure. When bounding $\alpha(j)$ from above by some $\alpha_{\text{max}}$, the $\alpha_{\text{max}}$ curve represents the boundary between the tips of longer vectors that can use the greedy $\alpha(j)$ and the tips of shorter vectors where $\alpha(j)$ would be clamped. In particular, for the $\alpha_{\text{max}} = 1$ case, the yellow region in the figure represents the tips of longer vectors and the green region represents the tips of shorter vectors. In the green region, the second term in Equation 21c is added to the usual $r(\Delta c^q_S)^T j_i$ gain correlation, increasing preference for search directions that remain well-correlated after using them. Figure 18 (Right) shows the magnitude of the second term in Equation 21c. Additionally, one could multiply the second term in Equation 21c by an arbitrary scaling constant and increase its influence.

We also consider the case of clamping based on total progress. Figure 19 uses the same reference frame as Figure 18 but draws the boundary where $||j_i||_2 = ||r(\Delta c^q_S)||_2/2$. Figure 19 (Left) draws three search directions taking the greedy step. Regardless of the length of $j_i$, the greedy $\alpha(j)$ rescales such that $\alpha(j)j_i$ ends at the boundary between the green and yellow regions where $r(\Delta c^q_S) - \alpha(j_i)j_i$ is orthogonal to $j_i$. Figure 19 (Right) shows how clamping the progress limits the ability of a search direction to take the greedy step, resulting in the second term in Equation 21c being non-zero. Another way of choosing $\alpha(j)$ is based on the observation that $r(\Delta c^q_S) - \alpha(j_i)j_i$ is always less geometrically correlated with $j_i$ than $r(\Delta c^q_S)$ is, since it points to the left instead of upwards. Hence, one could choose $\alpha$ in order to bound how much worse the gain/geometric correlation of $r(\Delta c^q_S) - \alpha(j_i)j_i$ is allowed to have compared to that of $r(\Delta c^q_S)$. In general, there are many potential strategies, but in all such cases, our methodology is to first prune so that large gain correlation does not introduce poorly geometrically correlated vectors, and then to consider correlation of the new residual $r(\Delta c^q_S) - \alpha(j_i)j_i$ in addition to correlation of the current residual in order to favor scenarios like Figure 16 (Right) over Figure 16 (Left).

5.3. Examples

We consider the problem of determining parameters $w$ that best match a three-dimensional synthetic face model to a real image, as discussed in Section 2. Although we used CSS to generate Figure 4 and for related efforts, here we consider a slightly modified situation in order to better isolate and demonstrate the behavior of CSS. Let $w$ represent the controls for the face blendshapes, jaw angles, and jaw translation, and $x(w)$ represent the synthetic three-dimensional face surface obtained from $w$. We replace the inference-based neural network keypoint detector with a more deterministic artist-drawn rotoscoping of curves for the eyes and mouth, as shown in Figure 20 (Left). In order to generate comparable keypoints on the synthetic face
We choose a reference frame where \( r(\Delta c q_S) \) is a unit vector along the \( y \)-axis and \( j_i \) is in the first quadrant of the \( xy \)-plane. (Left) Poorly geometrically correlated vectors (those with tips in the red region) are pruned as in Section 5.1. Referring to the greedy \( \alpha(j_i) \) in Equation 22, we plot curves representing the tips of vectors \( j_i \) where the greedy \( \alpha(j_i) \) is equal to \( 1/1.5 \), \( 1 \), and \( 1/1.75 \). When bounding \( \alpha(j_i) \) from above by \( \alpha_{\text{max}} \), the \( \alpha_{G} = \alpha_{\text{max}} \) curve represents the boundary between the tips of longer vectors that can use the greedy \( \alpha(j_i) \) and the tips of shorter vectors where \( \alpha(j_i) \) would be clamped. In particular, for the \( \alpha_{\text{max}} = 1 \) case, the yellow region represents the tips of longer vectors and the green region represents the tips of shorter vectors. (Right) The magnitude of the second term in Equation 21c for the \( \alpha_{\text{max}} = 1 \) case. Note that it is non-zero only for shorter vectors with tips inside the \( \alpha_{G} = 1 \) curve.

Then, \( x(w) \) determines the three-dimensional location of these barycentrically embedded curves, which subsequently are projected into the image plane using calibrated camera intrinsic and extrinsic parameters [62]; this simple projection replaces the differentiable renderer. See Figure 20 (Right). In order to obtain comparable keypoints, we label easily-identifiable locations on both sets of curves (i.e. those drawn on the synthetic model and those drawn on the image), e.g. corners of the mouth and eyes, middles of the lips, etc. To increase the number of comparable keypoints, we uniformly sample between the projected (into the image plane) locations of the easily-identifiable keypoints. Letting \( C^* \) be the two-dimensional keypoints on the real image and \( C(x(w)) \) be the corresponding projected keypoints determined by the parameters \( w \) of the synthetic model, we then solve

\[
\min_w \|C^* - C(x(w))\|_2^2
\]

in order to recover the parameters \( w \) that best match the two sets of keypoints together.

For comparison against CSS, we consider solving Equation 23 using Dogleg [112, 94] with no prior, Dogleg with a prior weight of \( \lambda = 3600 \), and BFGS [107] with a soft-\( L_1 \) prior with a weight of 3600 (i.e. with an extra term \( 3600 \sum_i 2(\sqrt{1 + w_i^2} - 1) \) [27]). When solving with CSS, we first prune all columns whose angle to the residual has an absolute cosine less than 0.3. Then, \( \alpha(j_i) \) is set to a fixed size of 0.01 and coordinate descent is run until the linear \( L_2 \) error no longer sufficiently decreases or when over 10 coordinates are used. We limit all four methods to at most 10 Gauss-Newton linearization iterations. Figure 21 shows the results. CSS and methods using regularization give the most reasonable geometric results. A major benefit of CSS is the resulting sparsity of the weights: while Dogleg with and without regularization sets nearly all the parameters to a non-zero value, CSS generally uses only a small number of non-zero weights. The soft \( L_1 \) regularized solution is sparser than the \( L_2 \) regularized solution; however, due to approximations in the chosen optimization approach (BFGS, Soft \( L_1 \)), it produces many small (i.e. \( < 1 \times 10^{-3} \)) weights instead of identically zero values. While one could clamp small values to zero, care must be taken to not accidentally clamp weights that contribute significantly to the overall solution.

To further elucidate the performance of these approaches, we construct a known exact solution and subsequently add an increasing amount of noise. First, an exact set of keypoints is determined by subsampling the contours on the three-dimensional face geometry. Then, a smile expression is created by setting two specific components of \( w \) to 1 while the other components remain set to zero. This known \( w^* \) determines

\[
\min_w \|C^* - C(x(w))\|_2^2
\]
Figure 19: We use the same reference frame as in Figure 18 but draw the boundary where \( \| j_i \|_2 = \| \Delta c_j \|_2 / 2 \). The yellow and green regions are shaded as in the \( \alpha_{\text{max}} = 1 \) case from Figure 18. (Left) Three search directions are drawn taking the greedy step. Regardless of the length of \( j_i \), the greedy \( \alpha(j_i) \) rescales such that \( \alpha(j_i) j_i \) ends at the boundary between the green and yellow regions where \( r(\Delta c_j - \alpha(j_i) j_i) \) is orthogonal to \( j_i \). (Right) Clamping progress limits the ability of a search direction to take the greedy step, resulting in the second term in Equation 21c being non-zero.

Figure 20: (Left) Hand-drawn rotoscope curves on a real image. (Right) Barycentrically embedded curves on the three-dimensional geometric facial model are projected into the image plane (red) and compared to curves drawn/rotoscoped on the real image (blue). The inset shows the facial parameters \( w \) for this pose.

532 face geometry \( x(w^*) \) along with barycentrically embedded keypoints that can be projected into the image plane to determine \( C^* \). The results obtained are shown in Figures 22a and 22d and are similar to those obtained using the real image data in Figure 21. Next, we add an increasing amount of uniformly distributed noise to \( C^* \) in the image plane. As expected, Dogleg with no regularization produces reasonable results when the noise is low but begins to overfit to the erroneous data as the amount of noise increases, see Figure 22 (first row). Both of the regularized approaches as well as CSS are able to target the noisy curves without overfitting, producing more reasonable geometry. The right half of Figure 22 shows that CSS yields sparser, more semantic solutions even with the added noise. Tables 1 and 2 demonstrate quantitative results for these examples. As seen in Table 1, CSS performs the best in all cases as measured by various metrics. Table 2 uses two sparsity measures: the \( l^0 \) metric counts how many facial parameters are strictly 0, and the Gini metric is \( 1 - 2 \sum_{i=1}^N \hat{w}_i / \| \hat{w} \|_1 (1 - \frac{N - i + 0.5}{N}) \), where \( \hat{w} \) are the sorted parameters with \( \hat{w}_i \) the \( i \)th largest [66]. Note how regularization improves the Gini metric, but does not necessarily improve the \( l^0 \) metric.

5.4. Parameter Study

Column Choice. Returning to the solution of Equation 23 for the real image data, we compare our approach for choosing the next coordinate descent column (Section 5.2) to using Gauss-Southwell and MBI. For each approach, we linearize and solve with no thresholding for the relative decrease in \( L_2 \) error, an upper limit of 10 unique coordinates used, and a fixed step size of 0.01; in these examples, we remove the eye rotoscope
Since the problem has been normalized so that the step size and convergence. Gauss-Southwell and CSS are overused/overweighted; our examples demonstrate that MBI chooses those coordinates more quickly than coordinate descent. It is generally a matter of when, not if, the algorithm chooses a coordinate that will be overused parameter. The parameters within a reasonable range while maintaining the sparsity of the solution. We note that with weighted shapes have magnitudes of 85.78 and 63.12. On the other hand, Gauss-Southwell and CSS keep the parameters within a reasonable range while maintaining the sparsity of the solution. We note that with coordinate descent it is generally a matter of when, not if, the algorithm chooses a coordinate that will be overused/overweighted; our examples demonstrate that MBI chooses those coordinates more quickly than Gauss-Southwell and CSS.

**Step Size & Convergence.** Since the problem has been normalized so that the $\alpha(j_i)$ generally make most sense between 0 and 1, here we compare fixed step sizes of $\alpha(j_i) = 0.01, 0.02, 0.1, 0.5$ and 1.0 to the full, greedy step in Equation 22. Without pruning, we run 10 Gauss-Newton iterations with no thresholding for the relative decrease in $L_2$ error and an upper limit of 10 unique coordinates used. We find that smaller step sizes achieve better overall facial shapes and less overused parameters (see Figure 24). In particular, the greedy step tends to be greater than 1 while step sizes of 0.02 and 0.01 only set 4. Removing the eye rotoscope curves causes the overused parameters to disappear; however, as seen in Figure 25, the greedy step causes the mouth to move unnaturally. This would seem to indicate that always taking the greedy step will result in some overfitting.

We also compare the effect of using fixed step sizes in Equation 20 versus the full, greedy step size equivalent to Gauss-Southwell without pruning. To isolate this variable, we run 10 Gauss-Newton iterations with no thresholding for the relative decrease in $L_2$ error and an upper limit of 10 unique coordinates used. We vary $\alpha(j_i)$ in Equation 20 but set the actual step size taken to be fixed at 0.01. As shown in Figure 26, while the resulting geometry and weights are all similar, our approach of allowing the step size to influence the chosen coordinate allows the optimization to more quickly reduce the error in earlier Gauss-Newton iterations than when using Gauss-Southwell (see Figure 27). Therefore, it may be beneficial to use CSS with a fixed size step when only a few Gauss-Newton iterations are desired.

**Pruning.** We rescale $r$ to $r = r/\|r\|$ and then compare different threshold values for pruning: 0.0 (no pruning), 0.2, 0.3, and 0.5. We run 10 Gauss-Newton iterations with a step size of 0.01 with no thresholding for the relative decrease in $L_2$ error. To emphasize the effect of pruning, we allow up to 50 unique coordinates per linearization, and focus only on the rotoscope curves around the mouth. With little to no pruning the model overfits and the geometry around the mouth deforms unreasonably. As the pruning threshold increases,

---

**Table 1:** Comparing the accuracy of estimating the facial parameters in the synthetic tests under various metrics. CSS produces the best results regardless of noise and metric.

<table>
<thead>
<tr>
<th>Method</th>
<th>$L_2$ Error</th>
<th>$L_1$ Error</th>
<th>EMD [116] Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Noise 0.005 0.01</td>
<td>No Noise 0.005 0.01</td>
<td>No Noise 0.005 0.01</td>
</tr>
<tr>
<td>Dogleg</td>
<td>2.578 8.15 20.325</td>
<td>19.227 70.852 157.22</td>
<td>0.128 0.485 1.07</td>
</tr>
<tr>
<td>Dogleg+$L_2$</td>
<td>0.972 0.952 1.209</td>
<td>4.954 5.324 5.704</td>
<td>0.034 0.036 0.039</td>
</tr>
<tr>
<td>BFGS+Soft $L_1$</td>
<td>0.923 0.91 1.023</td>
<td>3.139 3.057 3.359</td>
<td>0.0215 0.021 0.023</td>
</tr>
<tr>
<td>CSS</td>
<td>0.741 0.392 0.509</td>
<td>2.208 0.99 1.08</td>
<td>0.015 0.007 0.007</td>
</tr>
</tbody>
</table>

**Table 2:** The sparsity of the results of the synthetic tests using common sparsity metrics (a larger number is better).

<table>
<thead>
<tr>
<th>Method</th>
<th>$\ell^0$ Metric</th>
<th>Gini Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Noise 0.005 0.01</td>
<td>No Noise 0.005 0.01</td>
</tr>
<tr>
<td>Dogleg</td>
<td>21 21 21</td>
<td>0.628 0.580 0.607</td>
</tr>
<tr>
<td>Dogleg+$L_2$</td>
<td>21 21 21</td>
<td>0.807 0.745 0.739</td>
</tr>
<tr>
<td>BFGS+Soft $L_1$</td>
<td>21 21 21</td>
<td>0.913 0.905 0.916</td>
</tr>
<tr>
<td>CSS</td>
<td>128 137 140</td>
<td>0.949 0.974 0.978</td>
</tr>
</tbody>
</table>
the geometry becomes more regularized and the facial parameters are sparser, as the optimization is forced
to use only the most correlated directions. See Figure 28. However, we caution that too much pruning causes
MBI style column choices.

6. Conclusions

In difficult nonlinear problems such as the one described in Section 2, one often solves linear subproblems
to make progress. Although PCA is quite popular for solving such problems, especially when there are issues
with null spaces and the right-hand side not being in the range of the linearized system, we showed that our
iLM method not only efficiently monotonically converges to the exact solution of the linearized subproblem,
but does so more smoothly. We subsequently pointed out that the larger singular values of the coefficient
matrix can be less important than considering which controls are optimal for obtaining the right-hand side.
These considerations motivated our column space search (CSS) approach. We chose a complex real-world
problem, estimating three-dimensional facial expressions from a mere eight contours drawn on a single two-
dimensional RGB image, that allows even non-experts to simply glance at an image and comprehend the
effects of noise, overfitting, and regularization. We were able to robustly estimate clean sparse parameter
values with good semantic meaning in a highly underconstrained situation where one would typically need
significant regularization. In fact, the standard approach without regularization was wildly inaccurate, and
although regularization helped to moderate the overall face shape, it excited almost every parameter in the
model, clouding semantic interpretation.

Acknowledgements

Research supported in part by ONR N00014-13-1-0346, ONR N00014-17-1-2174, ARL AHPCRC W911NF-
07-002, and generous gifts from Amazon and Toyota. In addition, we would like to thank both Reza and
Behzad at ONR for supporting our efforts into computer vision and machine learning, as well as Cary Phillips
and Industrial Light & Magic for supporting our efforts into facial performance capture. M.B. was supported
in part by The VMWare Fellowship in Honor of Ole Agesen. We would also like to thank Paul Huston for
his acting.
Figure 21: Dogleg without regularization clearly overfits to the curves, producing highly unrealistic face shapes. Dogleg with regularization performs better but sometimes overfits as well. This could be tuned by increasing the regularization weight at the cost of potentially damping out the performance. Our approach produces facial expressions that are reasonably representative of the captured image. The inset bar plots demonstrate the sparsity of the weights for each of the methods. Our method generally produces the sparsest set of weights; e.g., in frame 1142, our method has 12 non-zero parameter values while $L_2$ regularization produces fully dense results and soft $L_1$ regularization has 49 significant parameter values (i.e., $>1 \times 10^{-3}$).
Figure 22: A synthetic test where a known $w^*$ is used to create blue target curves. (Left) As we increase the amount of noise added to the points on the blue target curve, the Dogleg method without regularization overfits causing the mesh to “explode” in spite of having the smallest error as measured by Equation 23 (typical of overfitting). On the other hand, both standard regularization and our approach prevent the model from overfitting to the noisy curves. (Right) The corresponding facial parameters. The target solution was generated by setting the two orange columns to one and the blue columns to zero. The figure heights are clipped at 1.0 and many parameter values exceed that. Though the regularized solves have smaller, spurious weights than the non-regularized version (second and third row vs. first row), our approach (last row) produces a much sparser solution with more semantic meaning even in the presence of noise.
Figure 23. A comparison of the coordinates chosen by the MBI rule, the Gauss-Southwell (GS) rule, and CSS when solving without the eye rotoscope curves. The top row are the results after a single Gauss-Newton iteration, and the bottom row are the results after 10 Gauss-Newton iterations.

Figure 24: We compare the behavior of the geometry and the facial parameters when using different step sizes.

Figure 25: A comparison of the behavior of the geometry and the facial parameters when using different step sizes without the eye rotoscope curves.
Figure 26: A comparison of the geometry and parameter results from varying the step size used for choosing the next coordinate in CSS.
Figure 27: A comparison of the average $L_2$ errors plotted before every Gauss-Newton iteration when varying the step size used to choose the next coordinate direction in CSS. The brown lines plot the average $L_2$ errors when using the Gauss-Southwell approach; notice how CSS allows for a faster reduction in error.

Figure 28: As we increase the threshold for pruning, the resulting solution becomes sparser and more regularized.
References


[16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016, 1604.07316.


[57] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding, 2015, 1510.00149.


[78] Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning.


