# ML2R Coding Nuggets
# Solving Least Squares Gradient Flows

Christian Bauckhage
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

Pascal Welke
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

## ABSTRACT

We approach least squares optimization from the point of view of gradient flows. As a practical example, we consider a simple linear regression problem, set up the corresponding differential equation, and show how to solve it using *SciPy*.

## 1 INTRODUCTION

In this note, we revisit **linear regression** [20] and **gradient flows** [3] and demonstrate how the latter can solve the former. A didactic example of our general setting is shown in Figure 1.

The figure depicts a sample of $n$ data points $[x_j, y_j]^\intercal \in \mathbb{R}^2$ which exhibit a linear trend. To learn a representation of these data, we may therefore fit a **linear model** $y_j = w_0 + w_1\, x_j + \epsilon_j$ where the parameters $w_0$ and $w_1$ denote intercept and slope of the line to be fitted and $\epsilon_j$ represents random noise. In order to estimate optimal model parameters $w_0^*$ and $w_1^*$, we may proceed as follows:

Introducing the 2-dimensional parameter- and feature vectors

$$\boldsymbol{w} = \begin{bmatrix} w_0 & w_1 \end{bmatrix}^\intercal \tag{1}$$

$$\boldsymbol{\varphi}_j = \begin{bmatrix} 1 & x_j \end{bmatrix}^\intercal \tag{2}$$

we can write our model in terms of an inner product $y_j = \boldsymbol{\varphi}_j^\intercal \boldsymbol{w} + \epsilon_j$ and resort to **least squares (LSQ)** optimization to determine an optimal parameter vector. The error- or loss function we consider in this case is the residual sum of squares

$$E(\boldsymbol{w}) = \sum_{j=1}^{n} \bigl(\boldsymbol{\varphi}_j^\intercal \boldsymbol{w} - y_j\bigr)^2 \tag{3}$$

Recall that this loss can also be written as a squared Euclidean distance. To this end, we gather the feature vectors $\boldsymbol{\varphi}_j$ in a feature matrix $\boldsymbol{\Phi} \in \mathbb{R}^{2\times n}$ and the $y_j$ in a target vector $\boldsymbol{y} \in \mathbb{R}^n$ where

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\varphi}_1 & \cdots & \boldsymbol{\varphi}_n \end{bmatrix} \tag{4}$$

$$\boldsymbol{y} = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix}^\intercal \tag{5}$$

which then allows us to write

$$E(\boldsymbol{w}) = \bigl\| \boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y} \bigr\|^2 = \boldsymbol{w}^\intercal \boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \boldsymbol{w} - 2\,\boldsymbol{w}^\intercal \boldsymbol{\Phi}\,\boldsymbol{y} + \boldsymbol{y}^\intercal \boldsymbol{y} \tag{6}$$

Since $E(\boldsymbol{w})$ is a convex function, it has a unique global minimum and a closed form expression for the location of this minimum is easy to come by. To see this, we consider the gradient

$$\nabla E(\boldsymbol{w}) = 2\,\boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \boldsymbol{w} - 2\,\boldsymbol{\Phi}\,\boldsymbol{y} \tag{7}$$

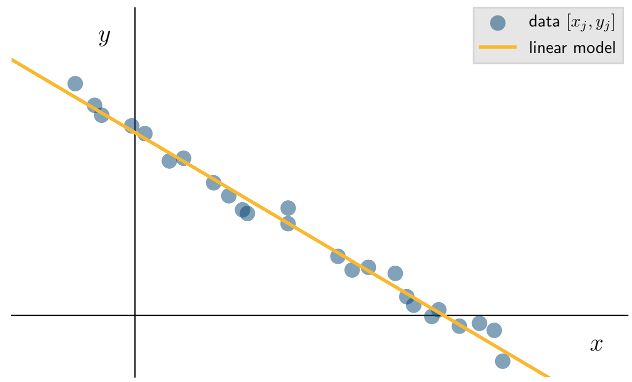


**Figure 1: A set of 2D data points and a line which was fitted via least squares regression. Intercept and slope parameters $\boldsymbol{w}^* = \begin{bmatrix} w_0^* & w_1^* \end{bmatrix}^\intercal$ of this line are $w_0^* \approx 2.97$ and $w_1^* \approx -0.59$.**

If we now write $\boldsymbol{w}^*$ to denote the minimizer of $E(\boldsymbol{w})$, we know that we must have $\nabla E(\boldsymbol{w}^*) = \boldsymbol{0}$. Based on this insight, we immediately find the sought after closed form solution for $\boldsymbol{w}^*$, because

$$2\,\boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal \boldsymbol{w}^* - 2\,\boldsymbol{\Phi}\,\boldsymbol{y} = \boldsymbol{0} \tag{8}$$

$$\Leftrightarrow \qquad \boldsymbol{w}^* = \bigl[\boldsymbol{\Phi}\boldsymbol{\Phi}^\intercal\bigr]^{-1} \boldsymbol{\Phi}\,\boldsymbol{y} \tag{9}$$

For experienced data scientists, all of this is well known and may even appear trivial. However, because of the "triviality" of (9), it is often overlooked that $\boldsymbol{w}^*$ indicates the point where the gradient of $E(\boldsymbol{w})$ vanishes. But this is to say that $\boldsymbol{w}^*$ can also be determined by means of iterative gradient descent

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \eta \cdot \nabla E\bigl(\boldsymbol{w}_k\bigr) \tag{10}$$

where $\eta > 0$ is an appropriate step size. This in itself is interesting, because it allows for numerically robust least squares solutions.

Yet, our main interest in this note is in an even lesser known fact, namely that the iteration in (10) has a continuous analog, the least squares flow, which allows for least squares optimization on emerging hardware platforms.

In section 2, we derive an ordinary differential equation from the above finite difference scheme and discuss general properties of the LSQ flow. In section 3, we then numerically solve the resulting initial value problem using *SciPy*'s `integrate` functionalities. Readers who want to experiment with our code should be familiar with *NumPy* and *SciPy* [15] and need to

```
import numpy as np
import numpy.linalg as la
from scipy.integrate import odeint
```

## 2 THEORY

This section explains what it means to say that the gradient descent scheme (10) has a continuous analog. That is, we show that (10) is but a finite difference approximation of a vector-valued ordinary differential equation.

To begin with, we note that $\nabla E\big(\boldsymbol{w}_k\big) = 2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w}_k - 2\,\boldsymbol{\Phi}\,\boldsymbol{y}$ is the error gradient at the current iterate of the descent procedure in (10). Hence, if we plug this expression into (10) and rearrange the resulting equation, we find

$$\frac{\boldsymbol{w}_{k+1} - \boldsymbol{w}_k}{\eta} = 2\,\boldsymbol{\Phi}\,\boldsymbol{y} - 2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w}_k \tag{11}$$

Given this expression, we next introduce a new parameter $t = k\,\eta$ so that $t + \eta = k\,\eta + \eta = (k+1)\,\eta$. If we then assume that $\boldsymbol{w}(t) = \boldsymbol{w}_k$, we have $\boldsymbol{w}(t+\eta) = \boldsymbol{w}_{k+1}$ and can rewrite equation (11) as

$$\frac{\boldsymbol{w}(t+\eta) - \boldsymbol{w}(t)}{\eta} = 2\,\boldsymbol{\Phi}\,\boldsymbol{y} - 2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w}(t) \tag{12}$$

At this point, it is obvious where we are headed, because, in the limit $\eta \longrightarrow 0$, the expression in (12) becomes

$$\tfrac{d}{dt}\,\boldsymbol{w}(t) = 2\,\boldsymbol{\Phi}\,\boldsymbol{y} - 2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w}(t) \tag{13}$$

$$= 2\,\boldsymbol{\Phi}\,\big[\,\boldsymbol{y} - \boldsymbol{\Phi}^\intercal \boldsymbol{w}(t)\,\big] \tag{14}$$

This is indeed an ordinary differential equation or a continuous time dynamical system and we can think of the gradient descent scheme in (10) as the **forward Euler method** for solving it.

The dynamical system in (13), (14) has several interesting or important characteristics. Most notably, we have

Lemma 2.1. *The dynamical system in (13), (14) is a gradient flow.*

Proof. In order to unpack this claim, we recall the notion of a **gradient flow**: Given a vector space $\mathbb{V}$ and a smooth function $f : \mathbb{V} \rightarrow \mathbb{R}$, a gradient flow is a smooth curve $\boldsymbol{x} : \mathbb{R} \rightarrow \mathbb{V}$, $t \mapsto \boldsymbol{x}(t)$ such that $\tfrac{d}{dt}\,\boldsymbol{x}(t) = -\nabla f\big(\boldsymbol{x}(t)\big)$.

To prove our claim, we must therefore find a function $f\big(\boldsymbol{w}(t)\big)$ such that

$$\tfrac{d}{dt}\,\boldsymbol{w}(t) = -\nabla f\big(\boldsymbol{w}(t)\big) \tag{15}$$

But we already know that such a function exists. It simply is the least squares error

$$E\big(\boldsymbol{w}(t)\big) = \big\|\,\boldsymbol{\Phi}^\intercal \boldsymbol{w}(t) - \boldsymbol{y}\,\big\|^2 \tag{16}$$

we considered in the previous section. □

Now that we know that the system in (13), (14) is a gradient flow, we next consider its convergence behavior. Here, we note

Lemma 2.2. *The LSQ flow has a unique equilibrium point.*

Proof. Recall that, in order for a point $\boldsymbol{w}^*$ to be an equilibrium point of the differential equation

$$\tfrac{d}{dt}\,\boldsymbol{w}(t) = -\nabla E\big(\boldsymbol{w}(t)\big) \tag{17}$$

we must have $-\nabla E\big(\boldsymbol{w}^*(t)\big) = \boldsymbol{0}$ for all $t$. Again, we already know that such a point exists, namely

$$\boldsymbol{w}^* = \big[\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal\big]^{-1} \boldsymbol{\Phi}\,\boldsymbol{y} \tag{18}$$

We also already know that this point is unique, because $E\big(\boldsymbol{w}(t)\big)$ is convex. □

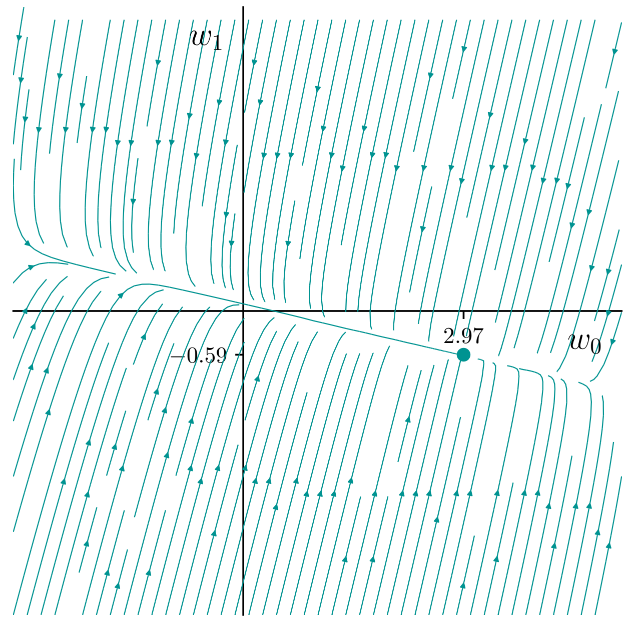


**Figure 2: Visualization of the least squares gradient flow for the linear regression problem posed by the data in Fig. 1.**

We furthermore have the following two lemmas whose proofs can be found in the appendix.

Lemma 2.3. *The equilibrium point of the LSQ flow is asymptotically stable.*

Lemma 2.4. *For any initial value $\boldsymbol{w}(0)$, the LSQ flow converges exponentially fast to its equilibrium point.*

In conclusion, all this means that, irrespective of where it starts, the flow in (13), (14) is guaranteed to quickly settle to the solution of the least squares optimization problem

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \big\|\,\boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y}\,\big\|^2 \tag{19}$$

For our introductory example where $\boldsymbol{w} \in \mathbb{R}^2$, we can actually visualize this behavior. Figure 2 shows the **flow field** of the least squares flow with $\boldsymbol{\Phi}$ and $\boldsymbol{y}$ as in (4) and (5). The flow lines indicate how points $\boldsymbol{w}$ in this two-dimensional parameter space move under the dynamics in (13), (14). The green dot marks the point $[2.97, -0.59]^\intercal$ which (rounded to two decimal places) corresponds to the solution of the least squares regression problem in Fig. 1. All the flow lines in Fig. 2 do indeed converge to this point. Once they reach it, they never leave it which is to say that the point is an asymptotically stable equilibrium of the flow.

**Note:** Before we conclude our theoretical discussion, we should point out that none of the arguments we brought forth depended on the dimensionality of the particular problem we considered. In other words, while we based our discussion on the particular, two-dimensional parameter estimation problem that was set up in the introduction, everything we said applies to (much) higher-dimensional least squares problems as well.

## 3 PRACTICAL COMPUTATION

Having discussed theoretical properties of LSQ flows, the obvious question is, if we could actually use them to solve least squares optimization problems? Yes, we can! And we next show how to accomplish this with *SciPy*.

Without loss of generality, we consider the uni-variate linear regression problem from the introduction where we are given given data points $[x_j, y_j]^\intercal$ and want to regress the $x_j$ onto the $y_j$.

Hence, we first of all assume the given $x_j$ and $y_j$ have been gathered in two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ which we represent in in terms of one-dimensional *NumPy* arrays

```
vecX = np.array([ ... ])
vecY = np.array([ ... ])
```

(For those who want to work with a specific example, we provide the data from Fig. 1 in Listing 1.) Given vector $\boldsymbol{x}$, we can compute the feature matrix $\boldsymbol{\Phi}$ in (4). To this end, we may use *NumPy* functions such as these

```
matF = np.vstack((np.ones_like(vecX), vecX))
```

As a ground truth result, we first compute the least squares solution in (9). Recalling our discussion in [2], we may use

```
vecWopt = la.lstsq(matF.T, vecY, rcond=None)[0]
print ('w0 =',vecWopt[0])
print ('w1 =',vecWopt[1])
```

which yields

```
>>> w0 =  2.9731819627593143
>>> w1 = -0.5929611518969758
```

These are indeed the parameters of the yellow line plotted in Fig. 1

In what follows, we will work with the expression in (14) which, once again reads

$$\tfrac{d}{dt}\,\boldsymbol{w}(t) = 2\,\boldsymbol{\Phi}\left[\,\boldsymbol{y} - \boldsymbol{\Phi}^\intercal\,\boldsymbol{w}(t)\,\right] \tag{20}$$

Scrutinizing this expression, we realize that solving this LSQ flow means to compute

$$\boldsymbol{w}(t) = \int_0^t 2\,\boldsymbol{\Phi}\left[\,\boldsymbol{y} - \boldsymbol{\Phi}^\intercal\,\boldsymbol{w}(\tau)\,\right] d\tau \tag{21}$$

which begs the question of how to solve the integral on the right?

The strategy we adhere to in the following is to use numerical integration. To this end, we will resort to function `odeint` which is available in *SciPy*'s `integrate` module.[1]

Listing 2 shows a function `integrateLSQFlow` which illustrates the use of `odeint` for our purpose. It is called with five parameters `vecW0`, `matF`, `vecY`, `tmax`, and `nsteps`.

Parameter `vecW0` is a 1D array representing a vector $\boldsymbol{w}(0)$ that indicates (an arbitrary choice of) the initial value of the LSQ flow at time $t = 0$. Parameter `matF` again represents the feature matrix $\boldsymbol{\Phi}$ and `vecY` the corresponding target vector $\boldsymbol{y}$; the roles of the other two parameters will become clear shortly.

[1]**NOTE:** `odeint` is now considered a legacy function and users of the latest versions of *SciPy* are encouraged to work with `solve_ivp` instead. This function, too, is found in the `integrate` module. However, its API has undergone some changes over the past couple of *SciPy* releases so that discussing its use would entail the risk that readers working with slightly older *SciPy* versions could not run our code. Hence, we stick with "good old" `odeint`.

**Listing 1: data $(x_j, y_j)$ used in Fig. 1**

```
print (vecX)
[ 3.52663187  5.26826326  0.16147591  2.48587331 -0.96982989
  5.83416256  0.78903847  0.55833354  1.52348867  1.7466569
  2.48806918 -0.65600672 -0.05321209  4.93386817  4.41435732
  5.97179385  1.27788725 -0.54061263  3.79093283  4.22726313
  3.29603093  4.82117652  4.52661869  1.82588778  5.5941991 ]

print (vecY)
[ 0.73982087 -0.17451263  2.95659681  1.49251154  3.76861084
 -0.24208398  2.55615381  2.51249452  1.94519087  1.71712361
  1.74627798  3.4169818   3.08288685  0.09202541  0.30491949
 -0.74382731  2.15664118  3.25568333  0.78319727  0.68466866
  0.96123544 -0.01571908  0.16697727  1.65817162 -0.12739275]
```

**Listing 2: numerically integrating, i.e. solving, the LSQ flow**

```
1 def integrateLSQFlow(vecW0, matF, vecY, tmax=1.0, nsteps=501):
2
3     def derivative(vecW, t, matF, vecY):
4         return 2 * matF @ ( vecY - matF.T @ vecW)
5
6     steps = np.linspace(0, tmax, nsteps)
7     matW  = odeint(derivative, vecW0, steps, (matF, vecY))
8
9     return matW
```

At the beginning of `integrateLSQFlow` (in lines 3 and 4), we define a function `derivative` which implements the differential equation in (14).

When using `odeint` to numerically integrate such a differential equation, we must specify (a sequence of) time points at which to solve the equation. Line 6 initializes a corresponding array `steps`. Here, the initial time point is `0`, the last one is `tmax`, and the number of steps in between is given by `nsteps`. These are the two additional parameters passed to `integrateLSQFlow` whose default values are 1.0 and 501. However, in general, users may have to choose the values of these parameters with respect to the problem at hand.

Line 7 then invokes `odeint` to solve the LSQ flow. Of the many parameters of `odeint`, the following ones are most important for our current setting:

- the 1st required parameter is a callable object, i.e. a function that computes the differential equation we wish to solve; here we set it to `derivative`; note that parameter `t` of function `derivative` does not occur in the function's body but `odeint` requires it to be present; also note that the order in which parameters `vecW` and `t` occur in the definition of `derivative` is another requirement of `odeint`
- the 2nd required parameter represents the initial condition of the system to be solved; hence, we pass array `vecW0`
- the 3rd mandatory parameter represents the time points at which to solve the differential equation under consideration; here, we therefore pass the array `steps`
- `args` is an optional parameter that is only required if the function passed in the first argument has additional parameters (other than `vecW` and `t`); in our case it has, namely `matF` and `vecY` and so we pass them in a tuple

Used in this fashion, `odeint` produces a *NumPy* array of `nsteps` rows which we store in `matW` and return from `integrateLSQFlow`.

Hence, given `matF` and `vecY` as defined above and assuming that `m, n = matF.shape`, we may use

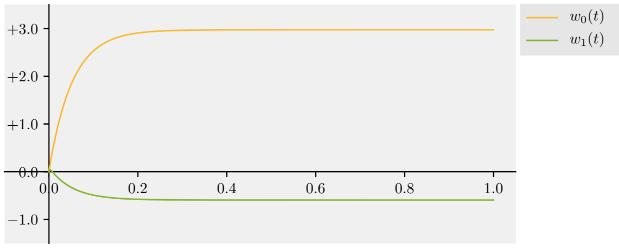


**Figure 3: Visualization of a least squares gradient flow for the linear regression problem in Fig. 1. Starting at $\boldsymbol{w}(0) = \boldsymbol{0}$, the figure shows how the components $w_0(t)$ and $w_1(t)$ of $\boldsymbol{w}(t)$ evolve over time. Confirming theoretical expectations, the flow quickly reaches a stable point.**

```
vecW0 = np.zeros(m)
matW  = integrateLSQFlow(vecW0, matF, vecY)
```

to obtain an array whose rows represent the states of the LSQ flow at the `nsteps` time points between `0` and `tmax`. If `tmax` is large enough, the last row `matW[-1]` of array `matW` represents a vector $\boldsymbol{w}(t_{\max})$ that corresponds to a stable equilibrium of the flow. For instance, for our practical problem / data in Fig. 1, we find

```
print ('w0 =',matW[-1,0])
print ('w1 =',matW[-1,1])
>>> w0 =  2.9731819300499662
>>> w1 = -0.5929611442824372
```

This almost perfectly agrees with our ground truth result obtained from using `la.lstsq`. The minute differences between the two results (they only differ after the seventh decimal place) can be attributed to numerical imprecision. All in all, our example therefore confirms the theoretical expectation that the LSQ flow converges to the solution of the least squares regression problem.

## 4 CONCLUSION

In this note, we called attention to the (well known) fact that least squares problems can be solved via gradient descent. More crucially, we then saw that such gradient descent procedures can be understood as discretized versions of continuous gradient flows.

While this is an interesting result that allows for deeper insights into the behavior of (robust) least squares methods [1], one may wonder if it provides immediate practical benefits? At this point in time, the answer is a resounding NO, BUT …

While we saw that numerical integration can solve LSQ flows, the required numerical methods are rather demanding and usually cannot compete with high performance linear algebra routines for least squares computation. In this sense, dynamical systems for least squares optimization are of limited use when working with conventional computers.

However, we are currently witnessing the (re)emergence of next generation computing devices which transcend certain limitations of digital computers. Indeed, analog computers (as well as special purpose VLSI circuits) can solve differential equations [8, 16]. Since there have been interesting developments in this area [6, 9, 12, 18], the LSQ flow may become of practical vlaue. There has also been progress in quantum computing and quantum computers, too, can solve differential equations [4, 11, 13]. As they promise up to exponential speed up over classical computers, this, too, may lead to practical use cases for the LSQ flow.

In short, since least squares optimization plays a fundamental role in data analysis, pattern recofnition, and machine learning (here are but a few works by ML2R researchers which illustrate this [5, 7, 10, 14, 17, 19]), and since the LSQ flow allows for computing solutions in a manner that suites next generation computing devices, it may soon play a bigger role than hitherto.

## A APPENDIX

In the following, we provide the proofs of Lemma 2.3 and 2.4. For convenience, we will switch from the Leibniz notation to Newton's notation for temporal derivatives and write the LSQ flow in (13) as

$$\dot{\boldsymbol{w}}(t) = 2\,\boldsymbol{\Phi}\,\boldsymbol{y} - 2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal\,\boldsymbol{w}(t) \tag{22}$$

Having recalled common notational conventions, we can proceed and provide the

Proof of Lemma 2.3. To show that the equilibrium point $\boldsymbol{w}^*$ of the LSQ flow is asymptotically stable, we first rewrite the differential equation in (22) in a more "standard form". To this end, we define

$$\boldsymbol{A} \equiv -2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \tag{23}$$

$$\boldsymbol{b} \equiv \phantom{-}2\,\boldsymbol{\Phi}\,\boldsymbol{y} \tag{24}$$

and write

$$\dot{\boldsymbol{w}}(t) = \boldsymbol{A}\,\boldsymbol{w}(t) + \boldsymbol{b} \tag{25}$$

Next, we recall that an equilibrium point of this equation is stable, if an only if all the eigenvalues of matrix $\boldsymbol{A}$ have negative real parts. To see that this is indeed the case, we observe that

$$\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal = \sum_{j=1}^{n} \boldsymbol{\varphi}_j\,\boldsymbol{\varphi}_j^\intercal \tag{26}$$

is an auto-correlation matrix. It is thus symmetric $\left[\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal\right]^\intercal = \boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal$ and positive definite, because, for any $\boldsymbol{x} \neq \boldsymbol{0}$, we have

$$\boldsymbol{x}^\intercal\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal\boldsymbol{x} = \sum_{j=1}^{n} \boldsymbol{x}^\intercal\boldsymbol{\varphi}_j\,\boldsymbol{\varphi}_j^\intercal\boldsymbol{x} = \sum_{j=1}^{n}\left(\boldsymbol{x}^\intercal\boldsymbol{\varphi}_j\right)^2 > 0 \tag{27}$$

Since $\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal$ is symmetric and positive definite, all its eigenvalues are real and positive. Since $\boldsymbol{A} = -2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal$, all its eigenvalues are real and negative. □

Having shown the equilibrium point of the LSQ flow to be stable, we next prove that the flow converges exponentially fast to this equilibrium regardless of the initial value of the system.

Proof of Lemma 2.4. Again, we consider equation in (22) in a more "standard form", i.e. we define

$$\boldsymbol{A} \equiv -2\,\boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \tag{28}$$

$$\boldsymbol{b} \equiv \phantom{-}2\,\boldsymbol{\Phi}\,\boldsymbol{y} \tag{29}$$

and write

$$\dot{\boldsymbol{w}}(t) = \boldsymbol{A}\,\boldsymbol{w}(t) + \boldsymbol{b} \tag{30}$$

We already know that this system has an equilibrium point $\boldsymbol{w}^*$ for which $\dot{\boldsymbol{w}}^*(t) = \boldsymbol{A}\,\boldsymbol{w}^*(t) + \boldsymbol{b} = \boldsymbol{0}$. But this is to say that

$$\boldsymbol{w}^* = -\boldsymbol{A}^{-1}\boldsymbol{b} = \left[2\,\Phi\,\Phi^\intercal\right]^{-1} 2\,\Phi\,\boldsymbol{y} \tag{31}$$

$$= \tfrac{1}{2}\left[\Phi\,\Phi^\intercal\right]^{-1} 2\,\Phi\,\boldsymbol{y} = \left[\Phi\,\Phi^\intercal\right]^{-1} \Phi\,\boldsymbol{y} \tag{32}$$

which simply rephrases our result in the introduction. What is interesting about the equation $\boldsymbol{w}^* = -\boldsymbol{A}^{-1}\boldsymbol{b}$ is that it allows for rewriting the differential equation in (30) in homogeneous form w.r.t. a deviation from the equilibrium, namely

$$\dot{\boldsymbol{w}}(t) = \boldsymbol{A}\,\boldsymbol{w}(t) + \boldsymbol{b} \tag{33}$$

$$= \boldsymbol{A}\left[\boldsymbol{w}(t) + \boldsymbol{A}^{-1}\boldsymbol{b}\right] \tag{34}$$

$$= \boldsymbol{A}\left[\boldsymbol{w}(t) - \boldsymbol{w}^*\right] \tag{35}$$

If we consider an initial value of $\boldsymbol{w}(0)$, then the solution to this differential equation is given by

$$\boldsymbol{w}(t) = \boldsymbol{w}^* + e^{\boldsymbol{A}\,t}\left[\boldsymbol{w}(0) - \boldsymbol{w}^*\right] \tag{36}$$

$$= \left[\boldsymbol{I} - e^{\boldsymbol{A}\,t}\right]\boldsymbol{w}^* + e^{\boldsymbol{A}\,t}\boldsymbol{w}(0) \tag{37}$$

where $\boldsymbol{I}$ denotes the identity matrix and

$$e^{\boldsymbol{A}\,t} = \sum_{k=0}^{\infty} \frac{1}{k!}\left[\boldsymbol{A}\,t\right]^k \tag{38}$$

is a matrix exponential. The validity of this solution is easily verified by differentiating

$$\frac{d}{dt}\left[\boldsymbol{w}^* + e^{\boldsymbol{A}\,t}\left[\boldsymbol{w}(0) - \boldsymbol{w}^*\right]\right] = \boldsymbol{A}\,e^{\boldsymbol{A}\,t}\left[\boldsymbol{w}(0) - \boldsymbol{w}^*\right] \tag{39}$$

$$= \boldsymbol{A}\left[\boldsymbol{w}(t) - \boldsymbol{w}^*\right] \tag{40}$$

where the second step follows from the fact that (36) can be written as $e^{\boldsymbol{A}\,t}[\boldsymbol{w}(0) - \boldsymbol{w}^*] = \boldsymbol{w}(t) - \boldsymbol{w}^*$.

Finally, since $\boldsymbol{A}$ is a negative multiple of a symmetric, positive definite matrix whose eigenvalues have strictly positive real parts, namely $\boldsymbol{A} = -2\,\Phi\,\Phi^\intercal$, we have

$$\lim_{t\to\infty} e^{\boldsymbol{A}\,t} = \boldsymbol{0} \tag{41}$$

where the decay is exponential. This, in turn, establishes that

$$\lim_{t\to\infty} \boldsymbol{w}(t) = \lim_{t\to\infty}\left[\left[\boldsymbol{I} - e^{\boldsymbol{A}\,t}\right]\boldsymbol{w}^* + e^{\boldsymbol{A}\,t}\boldsymbol{w}(0)\right] = \boldsymbol{w}^* \tag{42}$$

which is to say that $\boldsymbol{w}(t)$ converges to $\boldsymbol{w}^*$ regardless of whatever initial value $\boldsymbol{w}(0)$ we consider. □

## ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01IS18038C). The authors gratefully acknowledge this support.

## REFERENCES

[1] A. Ali, J.Z. Kolter, and R.J. Tibshirani. 2019. A Continuous-Time View of Early Stopping for Least Squares. In *Proc. AISTATS*.
[2] C. Bauckhage. 2015. NumPy / SciPy Recipes for Data Science: Ordinary Least Squares Optimization. researchgate.net. https://dx.doi.org/10.13140/2.1.3370.3209/1.
[3] C. Bauckhage, S. Müller, and F. Beaumont. 2021. *ML2R Coding Nuggets: Solving the Single Unit Oja Flow.* Technical Report. MLAI, University of Bonn.
[4] D.W. Berry. 2014. High-order Quantum Algorithm for Solving Linear Differential Equations. *J. of Physics A: Mathematical and Theoretical* 47, 10 (2014).
[5] D. Biesner, E. Brito, L.P. Hillebrand, and R. Sifa. 2020. Hybrid ensemble predictor as quality metric for German text summarization: Fraunhofer IAIS at GermEval 2020 task 3. In *Proc. SWISSTEXT / KONVENS*.
[6] O. Bournez and A. Pouly. 2021. A Survey on Analog Models of Computation. In *Handbook of Computability and Complexity in Analysis*, V. Brattka and P. Hertling (Eds.). Springer. to appear.
[7] U. Brefeld, T. Gärtner, T. Scheffer, and S. Wrobel. 2006. Efficient Co-regularised Least Squares Regression. In *Proc. ICML*.
[8] R.W. Brockett. 1992. Analog and Digital Computing. In *Future Tendencies in Computer Science, Control and Applied Mathematics*, A. Bensoussan and J.P. Verjus (Eds.). LNCS, Vol. 653. Springer.
[9] A. Celik, M. Stanacevic, and G. Cauwenberghs. 2005. Gradient Flow Independent Component Analysis in Micropower VLSI. In *Proc. NIPS*.
[10] G.D. Evangelidis and C. Bauckhage. 2013. Efficient Subframe Video Alignment Using Short Descriptors. *IEEE Trans. Pattern Analysis and Machine Intelligence* 35, 10 (2013).
[11] L. Franken, B. Georgiev, S. Muecke, M. Wolter, N. Piatkowski, and C. Bauckhage. 2020. Gradient-free Quantum Optimization on NISQ Devices. *arXiv:2012.13453 [quant-ph]* (2020).
[12] D. Fu, S. Shah, T. Song, and J. Reif. 2018. DNA-Based Analog Computing. In *Synthetic Biology*, J. Braman (Ed.). Humana Press.
[13] B.T. Kiani, G. De Palma, D. Englund, W. Kaminsky, M. Marvian, and S. Lloyd. 2020. Quantum Advantage for Differential Equation Analysis. *arXiv:2010.15776 [quant-ph]* (2020).
[14] J. Kunegis, D. Fay, and C. Bauckhage. 2010. Network Growth and the Spectral Evolution Model. In *Proc. CIKM*. ACM.
[15] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[16] H.T. Siegelmann and S. Fishman. 1998. Analog Computation with Dynamical Systems. *Physica D* 120, 1–2 (1998).
[17] R. Sifa. 2020. DESICOM as a Metaheuristic. In *Proc. LION*.
[18] D. Solli and B. Jalali. 2015. Analog Optical Computing. *Nature Photonics* 9 (2015).
[19] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).
[20] P. Welke and C. Bauckhage. 2020. *ML2R Coding Nuggets: Linear Programming for Robust Regression.* Technical Report. MLAI, University of Bonn.