SUMMER PROCEEDINGS 2019

CSRI Summer Program
Sandia National Laboratories

Editors:
Michael Powell and Michael L. Parks
Sandia National Laboratories

September 21, 2020

SAND2020-9969 R

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

Preface

The Computer Science Research Institute (CSRI) brings university faculty and students to Sandia for focused collaborative research on Department of Energy (DOE) computer and computational science problems. The institute provides an opportunity for university researchers to learn about problems in computer and computational science at DOE laboratories. Participants conduct leading-edge research, interact with scientists and engineers at the laboratories, and help transfer results of their research to programs at the labs. Some specific CSRI research interest areas are: scalable solvers, optimization, adaptivity and mesh refinement, graph-based, discrete, and combinatorial algorithms, uncertainty estimation, mesh generation, dynamic load-balancing, virus and other malicious-code defense, visualization, scalable cluster computers and heterogeneous computers, data-intensive computing, environments for scalable computing, parallel input/output, advanced architectures, and theoretical computer science. The CSRI Summer Program includes the organization of a weekly seminar series and the publication of this summer proceedings.

Michael Powell
Michael L. Parks

September 21, 2020

Table of Contents

Computational & Applied Mathematics

Computational & Applied Mathematics are concerned with the design, analysis, and implementation of algorithms to solve mathematical, scientific, or engineering problems. Articles in this section describe time integration methods, development of a nonlocal vector calculus, graph-based learning, neural networks applied to nonlocal models, model order reduction via neural networks, and the functional tensor train.

M. Powell
M.L. Parks

September 21, 2020

EXPONENTIAL INTEGRATORS FOR THE HOMME-NH NONHYDROSTATIC ATMOSPHERE MODEL

CASSIDY F. KRAUSE* AND ANDREW J. STEYER†

Abstract. Time-stepping in the HOMME-NH nonhydrostatic atmosphere model requires integration of a stiff initial value problem. The current implementation in HOMME-NH uses implicit-explicit Runge-Kutta (IMEX RK) methods with a horizontally explicit vertically implicit (HEVI) partitioning. We introduce a horizontally explicit vertically integrating factor exponential (HEVIE) method as an alternative to solving these stiff equations. The benefit of exponential methods is that they allow a larger step-size than explicit methods; however, their main drawback is the cost of forming the matrix exponential. Here, we show that we can mitigate this cost by taking advantage of the tridiagonal-like form of our Jacobian and parallelizing our computations, making this solver an attractive option for HOMME-NH.

1. Introduction. Stiff initial value problems are a challenge for nonhydrostatic atmosphere models like HOMME-NH, described in [10]. Several methods have been proposed to avoid the step-size restriction associated with stiff problems, including implicit-explicit Runge-Kutta (IMEX RK) methods, which are currently implemented in HOMME-NH. Here, we investigate integrating factor Runge-Kutta (IFRK) methods, a class of exponential integrators, as an alternative approach to dealing with the stiff initial value problems in HOMME-NH.

Consider an additively partitioned differential equation of the form

$$\omega_t = f(\omega) = s(\omega) + n(\omega), \tag{1.1}$$

where $\omega \in \mathbb{R}^n$, and ω_t denotes the derivative with respect to time. With this partitioning, $s(\omega)$ contains the stiff part of the equation, and $n(\omega)$ contains the nonstiff part. In HOMME-NH, we wish to solve an initial value problem of the form 1.1 with initial condition $\omega_0 = \omega(t_0)$.

IMEX methods are partitioned methods for approximating initial value problems of additively partitioned differential equations that treat the stiff terms implicitly and the nonstiff terms explicitly. The implementation and performance of various IMEX RK methods for integration of HOMME-NH with a horizontally explicit vertically implicit (HEVI) partitioning are discussed in [9].

In this paper, we consider IFRK methods [7] as an alternative integrator for HOMME-NH with a horizontally explicit vertically integrating factor exponential (HEVIE) partitioning. Given an approximate solution trajectory $\omega_m \approx \omega(t_m)$, we define a new additive partitioning of (1.1) by linearizing the stiff part, $s(\omega)$, along ω_m. Often, $L_m = f'(\omega)$, but we do not assume this is the case. Let $L_m = \frac{\partial s}{\partial \omega}(\omega_m)$, and set $N_m(\omega) = f(\omega) - L_m\omega$. With IFRK methods, we split our equation as

$$\omega_t = f(\omega) = L_m\omega + N_m(\omega).$$

With this partitioning, L_m is a linear operator containing the stiff part of the equation, and $N_m(\omega)$ contains the nonstiff terms. Using the change of variables $\nu(t) = e^{-L_m(t-t_m)}\omega(t)$, we have

$$\nu_t = e^{-L_m(t-t_m)}N_m(e^{L_m(t-t_m)}\nu). \tag{1.2}$$

We then use a fully explicit $r-$stage RK method to solve (1.2), as explained in Section 4.

*University of Kansas, ckrause@ku.edu
†Sandia National Laboratories, asteyer@sandia.gov

While exponential integrators allow for a larger time step than explicit methods, a major drawback is the computational cost of forming the matrix exponential. For this reason, matrix exponential methods were seldom used in practice, but advances in efficiently forming e^A have recently made exponential-type methods a competitive choice for integrating stiff initial value problems (e.g., [2], [3]). Furthermore, the Jacobian obtained from linearizing the stiff terms in HOMME-NH is sparse, with a tridiagonal block and a block that is a scalar multiple of the identity (3.1). This structure allows us to form these matrix exponentials quickly and in parallel over the horizontal mesh, making this approach a viable alternative to the IMEX RK methods.

The rest of the paper is structured as follows: In Section 2 we give a brief background of numerical approximations of the matrix exponential; Section 3 discusses the implementation of IFRK methods with HOMME-NH; Section 4 details the specific IFRK methods we use in this paper; and numerical results are given in Section 5.

2. Numerical Approximation of the Matrix Exponential. The matrix exponential is defined by the power series

$$e^A = \sum_{j=0}^{\infty} \frac{1}{j!} A^j.$$

A simple approach to numerically approximate e^A is through a Padé or Taylor series approximation. Other methods of approximating the matrix exponential are discussed in [8].

We choose to implement a (p,q)-Padé approximation $e^A \approx \left[Q_{pq}(A)\right]^{-1} P_{pq}(A)$, where $P_{pq}(A)$ and $Q_{pq}(A)$ are defined as follows:

$$P_{pq}(A) = \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!} A^j \tag{2.1}$$

$$Q_{pq}(A) = \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!} (-A)^j \tag{2.2}$$

After we calculate the terms P_{pq} and Q_{pq}, the product $e^A \approx \left[Q_{pq}(A)\right]^{-1} P_{pq}(A)$ is a potentially expensive computation because of the matrix inversion and matrix products. Fortunately, for our application, the matrix A we consider has a special structure that allows us to approximate e^A efficiently and accurately. This implementation is further discussed in Section 3.

The error for the (p,q)-Padé approximation is given by

$$e^A - \left[Q_{pq}(A)\right]^{-1} P_{pq}(A) = \mathcal{O}(\|A\|^{p+q+1}),$$

though in practice, this may be a particularly pessimistic estimation of the error, as shown in Table 3.2. If the matrix A is ill-conditioned, the Padé approximation can be especially inaccurate. To mitigate this problem, we implement a scaling and squaring approach. We briefly outline this procedure here; for further detail and error analysis of the scaling and squaring method, see [1] and [5].

To scale e^A, we take advantage of the properties of matrix exponentials and write it as

$$e^A = \left(e^{A/k}\right)^k,$$

where we choose k to be the smallest power of 2 so that $\|A/k\|_2 \leq 0.5$. With this factorization, the matrix A is scaled down to A/k, and the matrix exponential $e^{A/k}$ is calculated using the Padé formulas given in (2.1) and (2.2). The resulting matrix exponential is then squared repeatedly to recover the desired matrix exponential e^A. This numerical approximation to the matrix exponential e^A is then used with an explicit r-stage RK method, as described in Section 4, to solve (1.2).

3. Implementation with HOMME-NH. As mentioned in Section 2, the specific structure of the Jacobian in HOMME-NH allows us to form the matrix exponential $e^A \approx \left[Q_{pq}(A)\right]^{-1} P_{pq}(A)$ efficiently. HOMME-NH is a nonhydrostatic atmosphere model whose state variables are listed in Table 3.1. The governing equations can be decomposed into

Table 3.1: Variables in HOMME-NH

Variable Name	Description
$\mathfrak{g}$	Gravitational constant
$\vec{u} = (u, v)^T$	Horizontal velocity
w	Vertical velocity
ϕ	Geopotential
θ	Potential temperature
π	Hydrostatic pressure
η	Mass-based hybrid terrain-following vertical coordinate
p_{nh}	Nonhydrostatic pressure
μ	$\frac{\partial p_{nh}}{\partial \eta} / \frac{\partial \pi}{\partial \eta}$

non-stiff and stiff terms as follows [9]:

$$\omega_t := \begin{bmatrix} \vec{u_t} \\ w_t \\ \phi_t \\ \theta_t \\ \frac{\partial \pi}{\partial \eta} \end{bmatrix} = n(\omega) + s(\omega), \text{ with } s(\omega) = \begin{bmatrix} 0 \\ -\mathfrak{g}(1-\mu) \\ \mathfrak{g}w \\ 0 \\ 0 \end{bmatrix}.$$

Linearizing $s(\omega)$ gives the following Jacobian:

$$J = \begin{bmatrix} 0 & & & & \\ & 0 & \mathfrak{g}\frac{\partial \mu}{\partial \phi} & & \\ & \mathfrak{g}I & 0 & & \\ & & & 0 & \\ & & & & 0 \end{bmatrix}, \tag{3.1}$$

where $\dfrac{\partial \mu}{\partial \phi}$ is tridiagonal. As explained in Section 4, IFRK methods require the formation of the matrix exponential $e^{\alpha J}$. Since J has the form given in 3.1, the matrix exponential

will have the form

$$e^{\alpha J} = \begin{bmatrix} I & & & \\ & \exp\left(\begin{bmatrix} 0 & \mathfrak{g}\alpha\frac{\partial\mu}{\partial\phi} \\ \mathfrak{g}\alpha I & 0 \end{bmatrix}\right) & & \\ & & I & \\ & & & I \end{bmatrix}, \quad \alpha \neq 0.$$

So, forming $e^{\alpha J}$ for $\alpha \neq 0$ reduces to forming an exponential of the tridiagonal-like matrix $A \coloneqq \begin{bmatrix} 0 & \mathfrak{g}\alpha\frac{\partial\mu}{\partial\phi} \\ \mathfrak{g}\alpha I & 0 \end{bmatrix}$, and the matrix exponential will only act on variables w and ϕ.

The structure of A allows us to solve $e^A \approx \left[Q_{pq}(A)\right]^{-1} P_{pq}(A)$ using tridiagonal solves and back substitution. To do so, we first factor $Q_{pq}(A)$ as

$$\begin{aligned} Q_{pq}(A) &= \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!}(-A)^j \\ &= \kappa \prod_{j=1}^{q} \left[\sigma_j I - A\right], \end{aligned}$$

where $\sigma_j \in \mathbb{C}$ for $j = 1, \ldots, q$. Since we want to solve for $R \coloneqq \left[Q_{pq}(A)\right]^{-1} P_{pq}(A)$, we write

$$\prod_{j=1}^{q} \left[\sigma_j I - A\right] R = \frac{1}{\kappa} P_{pq}(A).$$

Define $R_1 \coloneqq [\sigma_2 I - A]\,[\sigma_3 I - A] \cdots \left[\sigma_q I - A\right] R$. Then our equation becomes

$$[\sigma_1 I - A]\, R_1 = \frac{1}{\kappa} P_{pq}(A). \tag{3.2}$$

Note that $[\sigma_1 I - A] = \begin{bmatrix} \sigma_1 I & -\mathfrak{g}\alpha\frac{\partial\mu}{\partial\phi} \\ -\mathfrak{g}\alpha I & \sigma_1 I \end{bmatrix}$. This form allows us to solve for R_1 using far fewer operations than if $[\sigma_1 I - A]$ were a full matrix. To do this, we similarly partition R_1 and $\frac{1}{\kappa}P_{pq}(A)$ into block matrices:

$$R_1 = \begin{bmatrix} \hat{R}_1 \\ \hat{R}_2 \end{bmatrix} \qquad P_{pq}(A) = \begin{bmatrix} \hat{P}_1 \\ \hat{P}_2 \end{bmatrix}.$$

Then (3.2) becomes

$$\begin{bmatrix} \sigma_1 I & -\mathfrak{g}\alpha\frac{\partial\mu}{\partial\phi} \\ -\mathfrak{g}\alpha I & \sigma_1 I \end{bmatrix} \begin{bmatrix} \hat{R}_1 \\ \hat{R}_2 \end{bmatrix} = \begin{bmatrix} \hat{P}_1 \\ \hat{P}_2 \end{bmatrix}.$$

If $\sigma_1 = 0$ this decomposes to a tridiagonal linear solve and a trivial solve. If $\sigma_1 \neq 0$, then we left multiply by $\begin{bmatrix} I & 0 \\ \mathfrak{g}\alpha\sigma_1^{-1} I & I \end{bmatrix}$ to obtain:

$$\begin{bmatrix} \sigma_1 I & -\mathfrak{g}\alpha\frac{\partial\mu}{\partial\phi} \\ 0 & \sigma_1 I - (\mathfrak{g}\alpha)^2 \sigma_1^{-1}\frac{\partial\mu}{\partial\phi} \end{bmatrix} \begin{bmatrix} \hat{R}_1 \\ \hat{R}_2 \end{bmatrix} = \begin{bmatrix} \hat{P}_1 \\ \mathfrak{g}\alpha\sigma_1^{-1}\hat{P}_1 + \hat{P}_2 \end{bmatrix}.$$

The matrix block $\left[\sigma_1 I - (\mathfrak{g}\alpha)^2 \sigma_1^{-1} \frac{\partial \mu}{\partial \phi}\right]$ is tridiagonal since $\frac{\partial \mu}{\partial \phi}$ is tridiagonal. Therefore we can solve for $\hat{R}_2$ with a tridiagonal solve:

$$\left[\sigma_1 I - (\mathfrak{g}\alpha)^2 \sigma_1^{-1} \frac{\partial \mu}{\partial \phi}\right] \hat{R}_2 = \left[\mathfrak{g}\alpha\sigma_1^{-1}\hat{P}_1 + \hat{P}_2\right],$$

and then form $\hat{R}_1$ as

$$\hat{R}_1 = \sigma_1^{-1}\left[P_1 + \mathfrak{g}\alpha\frac{\partial \mu}{\partial \phi}\right]\hat{R}_2.$$

In this way, we have just solved (3.2) for $R_1 = \begin{bmatrix}\hat{R}_1 \\ \hat{R}_2\end{bmatrix}$. On the other hand, we also have

$$[\sigma_2 I - A]\,[\sigma_3 I - A]\cdots\left[\sigma_q I - A\right] R = R_1.$$

We can iteratively repeat the same procedure, defining

$$R_j \coloneqq \left[\sigma_{j+1} I - A\right]\left[\sigma_{j+2} I - A\right]\cdots\left[\sigma_q I - A\right] R$$

and solving $\left[\sigma_j I - A\right] R_j = R_{j-1}$ for R_j, continuing in this way until we arrive at

$$\left[\sigma_q I - A\right] R = R_{q-1}.$$

Solving this last tridiagonal system gives us the $R = \left[Q_{pq}(A)\right]^{-1} P_{pq}(A) \approx e^A$ that we are looking for.

This algorithm for computing $R \approx e^A$ is particularly efficient if the values of p and q are small, so that we can compute $\{\sigma_j\}_{j=1}^q$ by hand ahead of time. To validate the use of a (p,q)-Padé approximation, and to determine how large p and q must be, we let the model spin up to 15 days and consider the Jacobian at that timestep. In HOMME-NH, the Jacobian will always have unique, purely imaginary eigenvalues. This allows us to calculate the matrix exponential analytically and compare it to the (p,q)-Padé approximation.

To calculate e^A analytically, suppose $\lambda_1, \lambda_2, \ldots, \lambda_k$ are the unique eigenvalues of A, and V is the matrix of corresponding eigenvectors. Then the Schur decomposition of A is

$$A = V \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_k \end{bmatrix} V^{-1},$$

and the matrix exponential can be analytically computed to machine precision as

$$\begin{aligned} e^A &= V \exp\left(\begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_k \end{bmatrix}\right) V^{-1} \\ &= V \begin{bmatrix} e^{\lambda_1} & & & \\ & e^{\lambda_2} & & \\ & & \ddots & \\ & & & e^{\lambda_k} \end{bmatrix} V^{-1}. \end{aligned}$$

Table 3.2 gives the error of several diagonal (p, q)- Padé approximations, for different values of $p = q$.

Table 3.2: Error of Padé approximation as compared to analytic computation of e^A.

Value of $p = q$	**Error**
2	1.30e-10
3	5.25e-13
4	4.92e-13
5	5.42e-13

Since a $(2, 2)$-Padé approximation yields a considerably accurate matrix exponential and also gives the benefit of easily solving for the coefficients $\{\sigma_1, \sigma_2\}$ ahead of time, this is the approximation we choose to implement in our IFRK methods.

4. Integrating Factor Runge-Kutta Methods. The focus of this paper is to implement IFRK methods in the HOMME-NH nonhydrostatic atmosphere model. IFRK methods were introduced in [6] and shown to be a type of exponential RK method by [7].

For IFRK methods, we use an explicit r-stage RK method to solve (1.2). If the Butcher tableau is given by $\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$, then the solution to (1.2) with initial condition $v(t_0) = v_0$ and step-size Δt is given by

$$\begin{cases} v_{m+1} = v_m + \Delta t \sum_{k=1}^{r} b_k e^{-L_m(t_{m,k}-t_m)} N(e^{L_m(t_{m,k}-t_m)} g_k) \\ g_j = v_m + \Delta t \sum_{k=1}^{r} A_{j,k} e^{-L_m(t_{m,k}-t_m)} N(e^{L_m(t_{m,k}-t_m)} g_k), \end{cases}$$

for $j = 1, \ldots, r$, where $t_{m,k} = t_m + c_k \Delta t$. The naming convention we use for the explicit r-stage RK methods is "$ERK-nm$," where n is the order of the method, and m is the number of stages. Here, we consider four explicit RK methods, given by the Butcher tableaux below.

$$\text{ERK-12}: \begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline & 0 & 1 & 0 \end{array} \qquad \text{ERK-35}: \begin{array}{c|cccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/5 & 1/5 & 0 & 0 & 0 & 0 & 0 \\ 1/5 & 0 & 1/5 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1/3 & 0 & 0 & 0 \\ 2/3 & 0 & 0 & 0 & 2/3 & 0 & 0 \\ 1 & 1/4 & 0 & 0 & 0 & 3/4 & 0 \\ \hline & 1/4 & 0 & 0 & 0 & 3/4 & 0 \end{array}$$

$$\text{ERK-23}: \begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 0 & 0 & 1 & 0 \end{array} \qquad \text{ERK-43}: \begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

5. Results. After implementing each of the methods described in Section 4 into HOMME-NH, we tested their convergence in the DCMIP2012 Test Case 4.1 (baroclinic instability test case) [11]. We ran the test case for 15 days with a 300 second time-step using a trusted second order implicit-explicit Runge-Kutta method to generate a nontrivial flow. The model is then restarted at day 15 and integrated over a 0.1 second time-interval for convergence testing. For the "truth", we compared our exponential integrating factor methods to a third

order, five stage Ullrich method (equation 56 in [4]) with a timestep of 10^{-4}, calculating the maximum 2-norm error over all latitudes and longitudes. To verify our methods, we calculate the error of the stiff variables, w and ϕ, which the matrix exponential acts upon. We also calculate the error of u and v, which the matrix exponential does not affect, as reference variables. The absolute errors are listed in Table 5.1, and the relative errors are listed in Table 5.2.

Table 5.1: Absolute error of state variables

(a) Absolute error of u

Method	Δt	**Error**
ERK-12	0.1	3.59e-6
ERK-12	0.05	4.11e-7
ERK-12	0.02	1.31e-7
ERK-12	0.01	1.17e-7
ERK-23	0.1	8.41e-7
ERK-23	0.05	2.12e-7
ERK-23	0.02	3.40e-8
ERK-23	0.01	8.50e-9
ERK-35	0.1	3.15e-9
ERK-35	0.05	5.20e-11
ERK-35	0.02	1.00e-11
ERK-35	0.01	2.05e-12
ERK-43	0.1	1.63e-8
ERK-43	0.05	1.03e-9
ERK-43	0.02	2.60e-11
ERK-43	0.01	1.60e-12

(b) Absolute error of v

Method	Δt	**Error**
ERK-12	0.1	1.43e-6
ERK-12	0.05	1.78e-7
ERK-12	0.02	5.33e-8
ERK-12	0.01	4.72e-8
ERK-23	0.1	3.42e-7
ERK-23	0.05	8.55e-8
ERK-23	0.02	1.37e-8
ERK-23	0.01	3.42e-9
ERK-35	0.1	6.32e-9
ERK-35	0.05	1.17e-10
ERK-35	0.02	1.92e-11
ERK-35	0.01	3.59e-12
ERK-43	0.1	7.99e-9
ERK-43	0.05	5.01e-10
ERK-43	0.02	1.27e-11
ERK-43	0.01	8.16e-13

(c) Absolute error of w

Method	Δt	**Error**
ERK-12	0.1	6.85e+0
ERK-12	0.05	4.55e+0
ERK-12	0.02	2.03e+0
ERK-12	0.01	1.05e+0
ERK-23	0.1	1.80e+0
ERK-23	0.05	4.48e-1
ERK-23	0.02	7.16e-2
ERK-23	0.01	1.79e-2
ERK-35	0.1	2.12e-2
ERK-35	0.05	2.56e-3
ERK-35	0.02	1.60e-4
ERK-35	0.01	2.00e-5
ERK-43	0.1	5.02e-2
ERK-43	0.05	3.16e-3
ERK-43	0.02	8.12e-5
ERK-43	0.01	5.09e-6

(d) Absolute error of ϕ

Method	Δt	**Error**
ERK-12	0.1	9.12e+2
ERK-12	0.05	4.45e+2
ERK-12	0.02	1.79e+1
ERK-12	0.01	8.97e+1
ERK-23	0.1	5.05e-1
ERK-23	0.05	1.28e-1
ERK-23	0.02	2.05e-2
ERK-23	0.01	5.13e-3
ERK-35	0.1	1.13e-1
ERK-35	0.05	1.41e-2
ERK-35	0.02	8.99e-4
ERK-35	0.01	1.12e-4
ERK-43	0.1	2.05e-2
ERK-43	0.05	1.29e-3
ERK-43	0.02	3.31e-5
ERK-43	0.01	2.06e-6

Table 5.2: Relative error of state variables

(a) Relative error of u

Method	Δt	**Error**
ERK-12	0.1	2.52e-7
ERK-12	0.05	2.89e-8
ERK-12	0.02	9.22e-9
ERK-12	0.01	8.18e-9
ERK-23	0.1	5.90e-8
ERK-23	0.05	1.49e-8
ERK-23	0.02	2.39e-9
ERK-23	0.01	5.97e-10
ERK-35	0.1	2.21e-10
ERK-35	0.05	3.65e-12
ERK-35	0.02	7.03e-13
ERK-35	0.01	1.44e-13
ERK-43	0.1	1.15e-9
ERK-43	0.05	7.20e-11
ERK-43	0.02	1.83e-12
ERK-43	0.01	1.13e-13

(b) Relative error of v

Method	Δt	**Error**
ERK-12	0.1	1.83e-7
ERK-12	0.05	2.27e-8
ERK-12	0.02	6.81e-9
ERK-12	0.01	6.03e-9
ERK-23	0.1	4.36e-8
ERK-23	0.05	1.09e-8
ERK-23	0.02	1.75e-9
ERK-23	0.01	4.37e-10
ERK-35	0.1	8.07e-10
ERK-35	0.05	1.49e-11
ERK-35	0.02	2.45e-12
ERK-35	0.01	4.58e-13
ERK-43	0.1	1.02e-9
ERK-43	0.05	6.40e-11
ERK-43	0.02	1.62e-12
ERK-43	0.01	1.04e-13

(c) Relative error of w

Method	Δt	**Error**
ERK-12	0.1	6.00e+3
ERK-12	0.05	3.98e+3
ERK-12	0.02	1.78e+3
ERK-12	0.01	9.22e+2
ERK-23	0.1	1.58e+3
ERK-23	0.05	3.92e+2
ERK-23	0.02	6.27e+1
ERK-23	0.01	1.57e+1
ERK-35	0.1	1.85e+1
ERK-35	0.05	2.24e+0
ERK-35	0.02	1.40e-1
ERK-35	0.01	1.75e-2
ERK-43	0.1	4.40e+1
ERK-43	0.05	2.77e+0
ERK-43	0.02	7.11e-2
ERK-43	0.01	4.45e-3

(d) Relative error of ϕ

Method	Δt	**Error**
ERK-12	0.1	1.14e-4
ERK-12	0.05	5.58e-5
ERK-12	0.02	2.24e-5
ERK-12	0.01	1.12e-5
ERK-23	0.1	6.33e-7
ERK-23	0.05	1.60e-7
ERK-23	0.02	2.57e-8
ERK-23	0.01	6.44e-9
ERK-35	0.1	1.41e-7
ERK-35	0.05	1.76e-8
ERK-35	0.02	1.13e-9
ERK-35	0.01	1.41e-10
ERK-43	0.1	2.57e-8
ERK-43	0.05	1.62e-9
ERK-43	0.02	4.15e-11
ERK-43	0.01	2.59e-12

Note that the value of w is small, so even though the absolute error of the IFRK methods is satisfactory, the relative error is rather large. Recall from Table 3.2 that the error of the $(2,2)$-Padè approximation that we are implementing is $\mathcal{O}(10^{-10})$. This does not take into account the error from the IFRK method itself, and combining this error with the loss of digits that may happen because of the small values of w, it is not surprising that the relative error of w is larger than that of the other variables.

We graph these errors in Figure 5.1. The slope of these methods flattens out as the timestep goes to zero. One possible explanation for this is that the accuracy may be limited by the accuracy of the matrix exponential. Another potential explanation is that we are limited by the accuracy of the "truth."

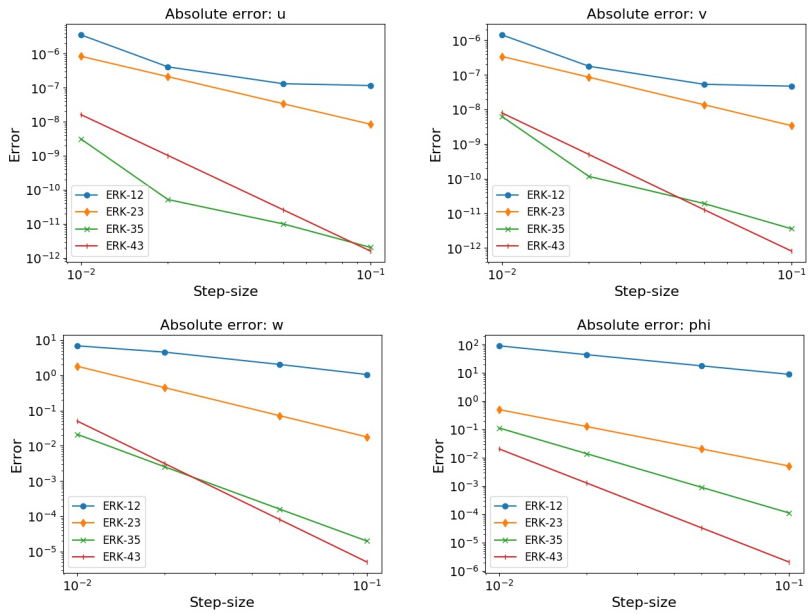


Fig. 5.1: Log-log plot of the absolute errors of IFRK methods.

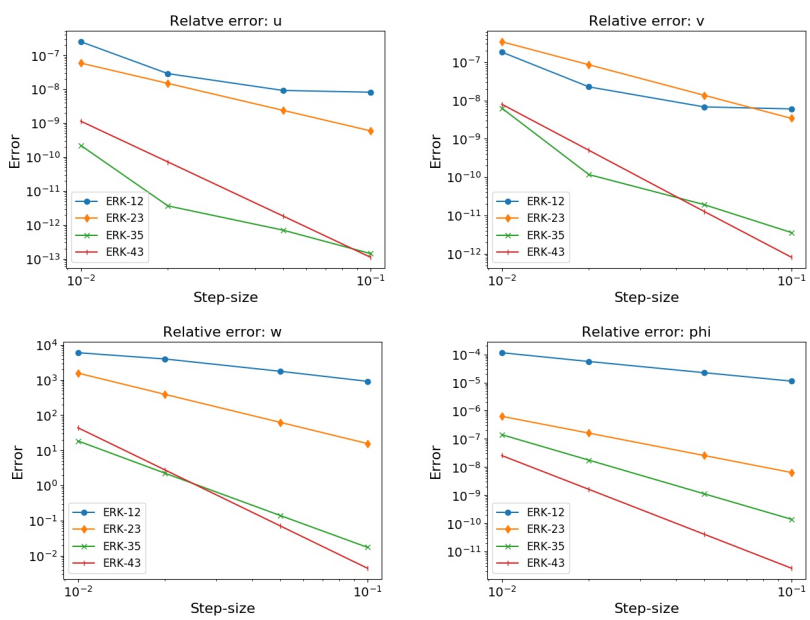


Fig. 5.2: Log-log plot of the relative errors of IFRK methods.

6. Conclusions. Another important matter to take into consideration is that the Jacobian at each point of our discretization scheme can be computed at the beginning of each time step, and we can do this computation in parallel. Furthermore, the IFRK methods rely only on the local-in-time Jacobian, so we can do the IFRK computation in parallel as well.

The IFRK methods allow us to deal with stiff initial value problems with a larger step-size than a fully explicit method, and the opportunity for parallelization with HOMME-NH boosts efficiency. Though the formation of the matrix exponential can be a costly computation, we are able to take advantage of the tridiagonal-like form of the Jacobian in HOMME-NH, to implement the IFRK methods in a way that mitigates the cost of forming the matrix exponential.

We have shown that IFRK methods are accurate and can be implemented in parallel. However, more work is needed to compare the efficiency of these methods, which will determine if IFRK methods can be competitive with the IMEX RK methods that are currently implemented in HOMME-NH. In particular, the efficiency of the IFRK methods will depend on how large of a step-size they can take, and this remains to be done.

REFERENCES

[1] A. Al-Mohy and N. Higham, *A new scaling and squaring algorithm for the matrix exponential*, SIAM Journal on Matrix Analysis and Applications, 31 (2009).

[2] C. Clancy and J. A. Pudykiewicz, *On the use of exponential time integration methods in atmospheric models*, Tellus A: Dynamic Meteorology and Oceanography, 65 (2013), p. 20898.

[3] F. Garcia, L. Bonaventura, M. Net, and J. Sánchez, *Exponential versus imex high-order time integrators for thermal convection in rotating spherical shells*, Journal of Computational Physics, 264 (2014), pp. 41–54.

[4] Guerra, J. and Ullrich, P., *A high-order staggered finite-element vertical discretization for non-hydrostatic atmospheric models*, Geosci. Model Dev., 9 (2016), pp. 2007–2029.

[5] N. Higham, *The scaling and squaring method for the matrix exponential revisited*, SIAM Journal on Matrix Analysis and Applications, 26 (2005), pp. 1179–1193.

[6] Lawson, J., *Generalized Runge-Kutta processes for stable systems with large Lipschitz constants*, SIAM J. Numer. Anal., 4 (1969), pp. 509–522.

[7] B. Minchev, *Integrating factor methods as exponential integrators*, in Large-Scale Scientific Computing, Lirkov, I., Margenov, S., and Waśniewski, J., eds., Berlin, Heidelberg, 2006, Springer Berlin Heidelberg, pp. 380–386.

[8] C. Moler and C. Van Loan, *Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later*, SIAM Review, 45 (2003), pp. 3–49.

[9] A. Steyer, C. J. Vogl, M. Taylor, and O. Guba, *Efficient imex runge-kutta methods for nonhydrostatic dynamics*, arXiv preprint arXiv:1906.07219, (2019).

[10] Taylor, M., Guba, O., Steyer, A., Ullrich, P., Hall, D., and C. Eldred, *An energy consistent discretization of the nonhydrostatic equations in primitive variables*, Preprint, (2019).

[11] Ullrich, P., Jablonowski, C., Kent, J., Lauritzen, P., Nair, R., and Taylor, M., *Dynamical core model intercomparison project (DCMIP) 2012 test case document v.1.7*, Tech. Rep., (2012).

TOWARDS A UNIFIED NONLOCAL VECTOR CALCULUS

HAYLEY A. OLSON*, MARTA D'ELIA†, AND MAMIKON GULIAN ‡

Abstract. In this work we provide the groundwork for a unified theory of nonlocal operators. Specifically, in the context of nonlocal diffusion, we prove the equivalence, for certain kernel functions, of weighted and unweighted operators. After studying general properties of the "equivalence" kernel, we show that the equivalence holds for fractional-type operators. We also make preliminary steps towards a unified well-posedness theory that holds for broad class of nonlocal operators by leveraging the well-established theory for unweighted operators, and the generalized operator definition that arises from our equivalence result.

1. Introduction. The use of nonlocal models in place of their classical differential counterparts has been steadily increasing thanks to their potential to capture effects that partial differential equations cannot describe. These effects include multiscale behavior and anomalous behavior such as super- and sub-diffusion and make nonlocal models suitable for a broad class of engineering and scientific applications ranging from fracture mechanics to image processing [1, 2, 3, 4, 5, 6, 7, 13, 14, 15, 16, 17, 18, 19, 20].

These models are characterized by integral operators, see Fig. 1.1, acting on neighborhoods B_δ (the Euclidean ball of radius δ, referred to as the *horizon*) of size much smaller than the domain or on regions much larger than the domain, including the whole space, see Fig. 1.1 (bottom). The integral form allows one to catch long-range forces and reduces the regularity requirements on the solution. As a result, in a nonlocal model, the state of a system at a point depends on a neighborhood of points. Many challenges arise from modeling and simulation of nonlocal problems, including the non-trivial prescription of boundary conditions, the unresolved treatment of nonlocal interfaces, the uncertainty and sparsity of model parameters and data and the prohibitively high computational cost as the extent of the nonlocal interactions increases; i.e. as the neighborhood becomes larger. Additionally, in the literature we have several independent definitions, formulations, and (possibly incomplete) theories of nonlocal models, see Figure 1.1 for an illustration. Similarities are evident, but they have not been rigorously proved; this is the ultimate goal of this preliminary work.

More specifically, in this paper we determine conditions on the kernel functions γ and η such that unweighted and weighted operators are equivalent. Also, we show that, for a specific choice of η, the weighted nonlocal operator is equivalent to the well-known fractional Laplacian operator. We also make preliminary steps towards a unified well-posedness theory that holds for all classes of operators by leveraging the well-established theory for unweighted operators [10], the generalized definition arising from our equivalence result, and a weighted nonlocal Green's identity [8].

Several reasons make the development of a unified theory impactful. A unified nonlocal vector calculus 1) Connects the nonlocal and fractional communities that would benefit from each other's research; 2) Includes as special cases the well-known classical differential calculus at the limit of vanishing interactions and the fractional calculus at the limit of infinite interactions; 3) Provides the groundwork for *new model discovery* thanks to the broad class of operators that it describes; 4) Describes intrinsically nonlocal phenomena that have not been analyzed or used due to the lack of theory; 5) Guides algorithm, discretization, and solver design.

This paper is organized as follows. In Section 2 we report relevant definitions and results that will be used throughout the paper. In Section 3, we present the derivation

*University of Nebraska - Lincoln, hayley.olson@huskers.unl.edu
†Sandia National Laboratories, mdelia@sandia.gov
‡Sandia National Laboratories, mgulian@sandia.gov

δ-truncated unweighted nonlocal

$$L_\delta u(x) = \int_{B_\delta(x)} (u(x) - u(y))\gamma(x,y)\,dy$$

δ-truncated ω-weighted nonlocal

$$L_\omega u(x) = \int_{B_\delta(x)} (u(x) - u(y)) \int_{\mathbb{R}^n} \eta(y,z;\omega)\,dz\,dy$$

fractional

$$L_s u(x) = \int_{\mathbb{R}^n} (u(x) - u(y)) \frac{C(s,n)}{|x-y|^{n+2s}} dy$$

Fig. 1.1: Classes of nonlocal operators.

of a nonlocal kernel which shows equivalence of the standard nonlocal Laplacian operator with the *weighted* Laplacian operator. This is followed by the analysis of properties of such kernel in Section 4, which includes the equivalence of the weighted nonlocal Laplacian and the fractional Laplacian. Finally, in Section 5, we provide some insights regarding the well-posedness of a class of nonlocal problems.

2. Background and Notation. We introduce weighted and unweighted nonlocal operators following [11]. In particular, let $\boldsymbol{\alpha} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, for $n = 1, 2, 3$, be an anti-symmetric vector two-point function. For $\boldsymbol{\nu} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, the unweighted nonlocal divergence $\mathcal{D}\boldsymbol{\nu} : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\mathcal{D}\boldsymbol{\nu}(\mathbf{x}) := \int_{\mathbb{R}^n} (\boldsymbol{\nu}(\mathbf{x},\mathbf{y}) + \boldsymbol{\nu}(\mathbf{y},\mathbf{x})) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) d\mathbf{y}, \quad \mathbf{x} \in \mathbb{R}^n. \tag{2.1}$$

Then for $u : \mathbb{R}^n \to \mathbb{R}$ the unweighted nonlocal gradient, $\mathcal{D}^* u : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, negative adjoint of (2.1), is defined as

$$\mathcal{D}^* u(\mathbf{x},\mathbf{y}) = -(u(\mathbf{y}) - u(\mathbf{x}))\boldsymbol{\alpha}(\mathbf{x},\mathbf{y}), \quad \mathbf{x},\mathbf{y} \in \mathbb{R}^n. \tag{2.2}$$

In this work, as in [11], we consider functions $\boldsymbol{\alpha}$ with bounded support; specifically, we assume that $\boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) = 0$ when $|\mathbf{x}-\mathbf{y}| > \delta$, for some $\delta > 0$. For an open bounded set $\Omega \subset \mathbb{R}^n$ we define the interaction domain Ω_I as the set of points outside of Ω which have a nonzero $\boldsymbol{\alpha}$ interaction with points inside Ω. More specifically[1],

$$\Omega_I = \{\mathbf{y} \in \mathbb{R}^n \backslash \Omega : \boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) \neq 0, \mathbf{x} \in \Omega\} = \{\mathbf{y} \in \mathbb{R}^n \backslash \Omega : |\mathbf{x}-\mathbf{y}| \leq \delta, \mathbf{x} \in \Omega\}.$$

Note that this set plays the role of *nonlocal boundary*; in fact, when solving nonlocal diffusion equations in Ω, volume constraints on the solution have to be prescribed on Ω_I to guarantee well-posedness.

For the kernel $\gamma = \boldsymbol{\alpha} \cdot \boldsymbol{\alpha}$ and $\mathbf{x} \in \Omega$ we define the unweighted δ-truncated nonlocal Laplacian as the composition of unweighted nonlocal divergence and gradient, i.e.

$$L_\delta u(\mathbf{x}) = \mathcal{D}\mathcal{G}u(\mathbf{x}) = \int_{B_\delta(\mathbf{x})} (u(\mathbf{x}) - u(\mathbf{y}))\gamma(\mathbf{x},\mathbf{y}) d\mathbf{y}. \tag{2.3}$$

[1] Note that Ω and Ω_I need not be adjacent, which differs from the boundary of a domain. Also note that if we set $\Omega = \mathbb{R}^n$, then Ω_I is empty.

In [11] the reader can find results regarding well-posedness of equations involving (2.3) and further results such as integration by parts and Green's identities.

Operators (2.1) and (2.2) are the building blocks of weighted nonlocal operators. The major shift between the two is that the weighted operators are one-point functions. Throughout, we let $\omega : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a non-negative, symmetric scalar function. For $\boldsymbol{\nu} : \mathbb{R}^n \to \mathbb{R}^n$, the weighted nonlocal divergence $\mathcal{D}_\omega \boldsymbol{\nu} : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\mathcal{D}_\omega \boldsymbol{\nu}(\mathbf{x}) := \mathcal{D}(\omega(\mathbf{x},\mathbf{y})\boldsymbol{\nu}(\mathbf{x})) = \int_{\mathbb{R}^n} (\omega(\mathbf{x},\mathbf{y})\boldsymbol{\nu}(\mathbf{x}) + \omega(\mathbf{y},\mathbf{x})\boldsymbol{\nu}(\mathbf{y})) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) d\mathbf{y}, \quad \mathbf{x} \in \mathbb{R}^n. \tag{2.4}$$

For $u : \mathbb{R}^n \to \mathbb{R}$, the weighted nonlocal gradient $\mathcal{D}^*_\omega u : \mathbb{R}^n \to \mathbb{R}^n$ is defined as

$$\mathcal{D}^*_\omega u(\mathbf{x}) := \int_{\mathbb{R}^n} \mathcal{D}^* u(\mathbf{x},\mathbf{y}) \omega(\mathbf{x},\mathbf{y}) d\mathbf{y}, \quad \mathbf{x} \in \mathbb{R}^n. \tag{2.5}$$

Paper [11] show that the latter is the negative adjoint of the former. As done in the unweighted case, we define the δ-truncated, ω-weighted nonlocal Laplacian as the composition of (2.4) and (2.5), i.e., for $\mathbf{x} \in \Omega$

$$\begin{aligned} L_\omega u(\mathbf{x}) &= \mathcal{D}_\omega \mathcal{G}_\omega u(\mathbf{x}) = \mathcal{D}_\omega \mathcal{D}^*_\omega u(\mathbf{x}) = \mathcal{D}(\omega(\mathbf{x},\mathbf{y})\mathcal{D}^*_\omega u(\mathbf{x})) \\ &= \int_{\Omega \cup \Omega_I} \Big[\omega(\mathbf{x},\mathbf{y})\mathcal{D}^*_\omega u(\mathbf{x}) + \omega(\mathbf{y},\mathbf{x})\mathcal{D}^*_\omega u(\mathbf{y}) \Big] \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) d\mathbf{y}. \end{aligned} \tag{2.6}$$

Properties of this operator and its connection with (2.3) are analyzed in the following section.

The fractional Laplacian operator. A nonlocal operator that is ubiquitous in the literature is the fractional Laplacian $(-\Delta)^s$. For $s \in (0,1)$ and $f : \mathbb{R}^n \to \mathbb{R}$, it is defined as

$$(-\Delta)^s(f)(\mathbf{x}) = c_{n,s} \int_{\mathbb{R}^n} \frac{f(\mathbf{x}) - f(\mathbf{y})}{|\mathbf{x}-\mathbf{y}|^{n+2s}} d\mathbf{y}$$

with

$$c_{n,s} = \frac{4^s \Gamma(n/2 - s)}{\pi^{n/2} |\Gamma(-s)|}.$$

Paper [9] shows that for $\delta \to \infty$, the unweighted δ-truncated nonlocal Laplacian converges to $(-\Delta)^s$; one of our goals is to show that the latter can also be expressed as a composition of ω-weighted, δ-truncated nonlocal Laplacian for a specific choice of $\boldsymbol{\alpha}$ and ω.

3. Equivalence of the weighted and unweighted Laplacian operator. Given the scalar point function u, we want to establish the equivalence of $L_\omega u(\mathbf{x})$ and $L_\delta u(\mathbf{x})$ for

some choice of kernel γ. Due to the symmetry of $\omega(\mathbf{x},\mathbf{y})$, we have

$$\begin{aligned}
\mathcal{D}_\omega\mathcal{G}_\omega u(\mathbf{x}) &= \int_{\Omega\cup\Omega_I} (\mathcal{D}^*_\omega u(\mathbf{x}) + \mathcal{D}^*_\omega u(\mathbf{y})) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y} \\
&= \int_{\Omega\cup\Omega_I} \Bigg[\int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z})d\mathbf{z} \\
&+ \int_{\Omega\cup\Omega_I} (u(\mathbf{y}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z})d\mathbf{z} \Bigg] \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y} \\
&= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \qquad (3.1) \\
&+ \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{y}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z}. \qquad (3.2)
\end{aligned}$$

Let the integral in (3.1) be A and the one in (3.2) be B. We have

$$\begin{aligned}
A &= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \\
&= \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) \int_{\Omega\cup\Omega_I} \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z}.
\end{aligned}$$

Letting $\gamma_1(\mathbf{x},\mathbf{z}) = \boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) \int_{\Omega\cup\Omega_I} \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}$, we have

$$A = \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\gamma_1(\mathbf{x},\mathbf{z})d\mathbf{z}.$$

Next,

$$\begin{aligned}
B &= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{y}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \\
&= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{y}) - u(\mathbf{x}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \\
&+ \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z}.
\end{aligned}$$

Switching $\mathbf{y}$ and $\mathbf{z}$ in the first integral, and employing the anti-symmetry of $\boldsymbol{\alpha}$ and symmetry of ω, we find

$$\begin{aligned}
B &= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{z}) - u(\mathbf{x}))\boldsymbol{\alpha}(\mathbf{z},\mathbf{y})\omega(\mathbf{z},\mathbf{y}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z})d\mathbf{z}d\mathbf{y} \\
&+ \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \\
&= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z})d\mathbf{z}d\mathbf{y} \\
&+ \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{z} \\
&= \int_{\Omega\cup\Omega_I} \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot [\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) + \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})]d\mathbf{y}d\mathbf{z} \\
&= \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z})) \int_{\Omega\cup\Omega_I} \boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot [\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) + \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})]d\mathbf{y}d\mathbf{z}.
\end{aligned}$$

Letting $\gamma_2(\mathbf{x},\mathbf{z}) = \int_{\Omega\cup\Omega_I} \boldsymbol{\alpha}(\mathbf{y},\mathbf{z})\omega(\mathbf{y},\mathbf{z}) \cdot [\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) + \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})]d\mathbf{y}$ gives us

$$B = \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\gamma_2(\mathbf{x},\mathbf{z})d\mathbf{z}.$$

By combining the above, we have

$$\begin{aligned}\mathcal{D}_\omega\mathcal{G}_\omega u(\mathbf{x}) &= A + B \\ &= \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\gamma_1(\mathbf{x},\mathbf{z})d\mathbf{z} + \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))\gamma_2(\mathbf{x},\mathbf{z})d\mathbf{z} \\ &= \int_{\Omega\cup\Omega_I} (u(\mathbf{x}) - u(\mathbf{z}))(\gamma_1(\mathbf{x},\mathbf{z}) + \gamma_2(\mathbf{x},\mathbf{z}))d\mathbf{z}.\end{aligned}$$

Thus, for

$$\begin{aligned}\gamma(\mathbf{x},\mathbf{y}) &= \gamma_1(\mathbf{x},\mathbf{y}) + \gamma_2(\mathbf{x},\mathbf{y}) \\ &= \int_{\Omega\cup\Omega_I} [\boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) \\ &\qquad + \boldsymbol{\alpha}(\mathbf{z},\mathbf{y})\omega(\mathbf{z},\mathbf{y}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y}) + \boldsymbol{\alpha}(\mathbf{z},\mathbf{y})\omega(\mathbf{z},\mathbf{y}) \cdot \boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z})]d\mathbf{z},\end{aligned} \tag{3.3}$$

the operators $L_\delta u(\mathbf{x})$ and $L_\omega u(\mathbf{x})$ are equivalent.

4. Properties of the equivalence kernel. In this section we analyze properties of the kernel in (3.3); specifically, we investigate its symmetry and show equivalence of $L_\omega u(\mathbf{x})$ with the well-known fractional Laplacian operator for a specific choice of ω and $\boldsymbol{\alpha}$.

We point out that one of our major goals is to find conditions on the equivalence kernel that guarantee well-posedness of the associated nonlocal diffusion operator. This result, that would enable characterization of a broad class of well-posed nonlocal diffusion problems, is the subject of current research.

4.1. Symmetry of the equivalence kernel. The symmetry of γ can be shown directly using the antisymmetry of $\boldsymbol{\alpha}$ and the symmetry of ω. Throughout this section, we let $\boldsymbol{\eta}(\mathbf{x},\mathbf{y})$ be the antisymmtric function defined as $\boldsymbol{\eta}(\mathbf{x},\mathbf{y}) = \boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})$; then, we rewrite (3.3) as

$$\gamma(\mathbf{x},\mathbf{y}) = \int_{\Omega\cup\Omega_I} [\boldsymbol{\eta}(\mathbf{x},\mathbf{y}) \cdot \boldsymbol{\eta}(\mathbf{x},\mathbf{z}) + \boldsymbol{\eta}(\mathbf{z},\mathbf{y}) \cdot \boldsymbol{\eta}(\mathbf{x},\mathbf{y}) + \boldsymbol{\eta}(\mathbf{z},\mathbf{y}) \cdot \boldsymbol{\eta}(\mathbf{x},\mathbf{z})]d\mathbf{z}$$

The antisymmetry of $\boldsymbol{\eta}$ implies that

$$\gamma(\mathbf{x},\mathbf{y}) = \int_{\Omega\cup\Omega_I} [\boldsymbol{\eta}(\mathbf{y},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{z},\mathbf{x}) + \boldsymbol{\eta}(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{x}) + \boldsymbol{\eta}(\mathbf{y},\mathbf{z}) \cdot \boldsymbol{\eta}(\mathbf{z},\mathbf{x})]d\mathbf{z}$$

Since the dot product is commutative, we switch the orders of each $\boldsymbol{\eta}$ pair:

$$\gamma(\mathbf{x},\mathbf{y}) = \int_{\Omega\cup\Omega_I} [\boldsymbol{\eta}(\mathbf{z},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{x}) + \boldsymbol{\eta}(\mathbf{y},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{z}) + \boldsymbol{\eta}(\mathbf{z},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{z})]d\mathbf{z}$$

then, switching the first two terms, we have

$$\begin{aligned}\gamma(\mathbf{x},\mathbf{y}) &= \int_{\Omega\cup\Omega_I} [\boldsymbol{\eta}(\mathbf{y},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{z}) + \boldsymbol{\eta}(\mathbf{z},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{x}) + \boldsymbol{\eta}(\mathbf{z},\mathbf{x}) \cdot \boldsymbol{\eta}(\mathbf{y},\mathbf{z})]d\mathbf{z} \\ &= \gamma(\mathbf{y},\mathbf{x}).\end{aligned}$$

4.2. Equivalence of L_ω and the fractional Laplacian. In this section we show the equivalence of the w-weighted diffusion operator and the fractional Laplacian operator for the following choice of weight and kernel functions:

$$\omega(\mathbf{x},\mathbf{y}) = |\mathbf{y}-\mathbf{x}|\phi(|\mathbf{y}-\mathbf{x}|) \quad \text{for} \quad \phi(|\mathbf{y}-\mathbf{x}|) = \frac{1}{|\mathbf{y}-\mathbf{x}|^{d+1+s}}$$
$$\boldsymbol{\alpha}(\mathbf{x},\mathbf{y}) = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|},$$

for which

$$\boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y}) = |\mathbf{y}-\mathbf{x}|\phi(|\mathbf{y}-\mathbf{x}|)\frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|} = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}.$$

Thus,

$$\begin{aligned}\gamma(\mathbf{x},\mathbf{y}) &= \int_{\Omega\cup\Omega_I} [\boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y})\cdot\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z}) + \boldsymbol{\alpha}(\mathbf{z},\mathbf{y})\omega(\mathbf{z},\mathbf{y})\cdot\boldsymbol{\alpha}(\mathbf{x},\mathbf{y})\omega(\mathbf{x},\mathbf{y}) \\ &\quad + \boldsymbol{\alpha}(\mathbf{z},\mathbf{y})\omega(\mathbf{z},\mathbf{y})\cdot\boldsymbol{\alpha}(\mathbf{x},\mathbf{z})\omega(\mathbf{x},\mathbf{z})]d\mathbf{z} \\ &= \int_{\Omega\cup\Omega_I}\left[\frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}\cdot\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}} + \frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}\right. \\ &\quad \left. + \frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}}\right]d\mathbf{z}.\end{aligned}$$

We rewrite the expression above as the sum of three terms, $K(\mathbf{x},\mathbf{y}) = I + II + III$:

$$I = \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}\cdot\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}}dz = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}\cdot\int_{\mathbb{R}^d}\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}}dz, \tag{4.1}$$
$$II = \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}dz = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|^{d+s+1}}\cdot\int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}dz, \tag{4.2}$$
$$III = \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}}dz. \tag{4.3}$$

Due to the rotational symmetry of the domain of integration, the integrals in I and II are both zero. Thus, the kernel is just the term III, that we rename K:

$$K(\mathbf{x},\mathbf{y}) = \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{z}}{|\mathbf{y}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{z}-\mathbf{x}}{|\mathbf{z}-\mathbf{x}|^{d+s+1}}d\mathbf{z}. \tag{4.4}$$

We evaluate this integral indirectly. Let $\mathbf{z}_{\text{new}} = \mathbf{z}-\mathbf{x}$. Then $\mathbf{z} = \mathbf{z}_{\text{new}} + \mathbf{x}$, $d\mathbf{z} = d\mathbf{z}_{\text{new}}$ and $\mathbf{y}-\mathbf{z} = \mathbf{y}-\mathbf{z}_{\text{new}}-\mathbf{x} = \mathbf{y}-\mathbf{x}-\mathbf{z}_{\text{new}}$. Thus,

$$K(\mathbf{x},\mathbf{y}) = \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{x}-\mathbf{z}_{\text{new}}}{|\mathbf{y}-\mathbf{x}-\mathbf{z}_{\text{new}}|^{d+s+1}}\cdot\frac{\mathbf{z}_{\text{new}}}{|\mathbf{z}_{\text{new}}|^{d+s+1}}d\mathbf{z}_{\text{new}} \tag{4.5}$$
$$= \int_{\mathbb{R}^d}\frac{\mathbf{y}-\mathbf{x}-\mathbf{z}}{|\mathbf{y}-\mathbf{x}-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}}d\mathbf{z}. \tag{4.6}$$

From this, it follows that $K(\mathbf{x},\mathbf{y})$ depends only on $\mathbf{x}-\mathbf{y}$, i.e., we can write $K(\mathbf{x},\mathbf{y}) = K(\mathbf{x}-\mathbf{y})$. Next, we show that $K(\mathbf{x}-\mathbf{y})$ is rotationally invariant. Consider a rotation $\mathcal{R}$; we have

$$K\left(\mathcal{R}(\mathbf{x}-\mathbf{y})\right) = \int_{\mathbb{R}^d}\frac{\mathcal{R}(\mathbf{y}-\mathbf{x})-\mathbf{z}}{|\mathcal{R}(\mathbf{y}-\mathbf{x})-\mathbf{z}|^{d+s+1}}\cdot\frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}}d\mathbf{z}. \tag{4.7}$$

Let $\mathbf{z} = \mathcal{R}\mathbf{z}_{\text{new}}$. Then $d\mathbf{z} = d\mathbf{z}_{\text{new}}$, and

$$K\left(\mathcal{R}(\mathbf{x}-\mathbf{y})\right) = \int_{\mathbb{R}^d} \frac{\mathcal{R}(\mathbf{y}-\mathbf{x}) - \mathcal{R}\mathbf{z}_{\text{new}}}{|\mathcal{R}(\mathbf{y}-\mathbf{x}) - \mathcal{R}\mathbf{z}_{\text{new}}|^{d+s+1}} \cdot \frac{\mathcal{R}\mathbf{z}_{\text{new}}}{|\mathcal{R}\mathbf{z}_{\text{new}}|^{d+s+1}} d\mathbf{z}_{\text{new}} \tag{4.8}$$

$$= \int_{\mathbb{R}^d} \frac{\mathcal{R}(\mathbf{y}-\mathbf{x}) - \mathcal{R}\mathbf{z}}{|\mathcal{R}(\mathbf{y}-\mathbf{x}) - \mathcal{R}\mathbf{z}|^{d+s+1}} \cdot \frac{\mathcal{R}\mathbf{z}}{|\mathcal{R}\mathbf{z}|^{d+s+1}} d\mathbf{z} \tag{4.9}$$

$$= \int_{\mathbb{R}^d} \frac{\mathcal{R}\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right)}{|\mathcal{R}\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right)|^{d+s+1}} \cdot \frac{\mathcal{R}\mathbf{z}}{|\mathcal{R}\mathbf{z}|^{d+s+1}} d\mathbf{z} \tag{4.10}$$

$$= \int_{\mathbb{R}^d} \frac{1}{|\mathcal{R}\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right)|^{d+s+1}} \frac{1}{|\mathcal{R}\mathbf{z}|^{d+s+1}} \left[\mathcal{R}\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right) \cdot \mathcal{R}\mathbf{z}\right] \, d\mathbf{z} \tag{4.11}$$

$$= \int_{\mathbb{R}^d} \frac{1}{|\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right)|^{d+s+1}} \frac{1}{|\mathbf{z}|^{d+s+1}} \left[\left((\mathbf{y}-\mathbf{x}) - \mathbf{z}\right) \cdot \mathbf{z}\right] \, d\mathbf{z} \tag{4.12}$$

$$= \int_{\mathbb{R}^d} \frac{(\mathbf{y}-\mathbf{x}) - \mathbf{z}}{|(\mathbf{y}-\mathbf{x}) - \mathbf{z}|^{d+s+1}} \cdot \frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}} \, d\mathbf{z} \tag{4.13}$$

$$= K(\mathbf{x}-\mathbf{y}). \tag{4.14}$$

Therefore, K depends only on $|\mathbf{x}-\mathbf{y}|$ and we write $K(\mathbf{x},\mathbf{y}) = K(|\mathbf{x}-\mathbf{y}|)$. Now we let $\lambda > 0$ and consider

$$K(\lambda|\mathbf{x}-\mathbf{y}|) = \int_{\mathbb{R}^d} \frac{\lambda(\mathbf{y}-\mathbf{x}) - \mathbf{z}}{|\lambda(\mathbf{y}-\mathbf{x}) - \mathbf{z}|^{d+s+1}} \cdot \frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}} d\mathbf{z}. \tag{4.15}$$

Let $\mathbf{z} = \lambda\mathbf{z}_{\text{new}}$. Then $d\mathbf{z} = \lambda^d d\mathbf{z}_{\text{new}}$, and

$$K(\lambda|\mathbf{x}-\mathbf{y}|) = \int_{\mathbb{R}^d} \frac{\lambda(\mathbf{y}-\mathbf{x}) - \lambda\mathbf{z}_{\text{new}}}{|\lambda(\mathbf{y}-\mathbf{x}) - \lambda\mathbf{z}_{\text{new}}|^{d+s+1}} \cdot \frac{\lambda\mathbf{z}_{\text{new}}}{|\lambda\mathbf{z}_{\text{new}}|^{d+s+1}} \lambda^d d\mathbf{z}_{\text{new}} \tag{4.16}$$

$$= \int_{\mathbb{R}^d} \frac{\lambda(\mathbf{y}-\mathbf{x}) - \lambda\mathbf{z}}{|\lambda(\mathbf{y}-\mathbf{x}) - \lambda\mathbf{z}|^{d+s+1}} \cdot \frac{\lambda\mathbf{z}}{|\lambda\mathbf{z}|^{d+s+1}} \lambda^d dz \tag{4.17}$$

$$= \frac{\lambda}{\lambda^{d+s+1}} \frac{\lambda}{\lambda^{d+s+1}} \lambda^d \int_{\mathbb{R}^d} \frac{(\mathbf{y}-\mathbf{x}) - \mathbf{z}}{|(\mathbf{y}-\mathbf{x}) - \mathbf{z}|^{d+s+1}} \cdot \frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}} dz \tag{4.18}$$

$$= \frac{1}{\lambda^{d+2s}} \int_{\mathbb{R}^d} \frac{(\mathbf{y}-\mathbf{x}) - \mathbf{z}}{|(\mathbf{y}-\mathbf{x}) - \mathbf{z}|^{d+s+1}} \cdot \frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}} dz \tag{4.19}$$

$$= \frac{1}{\lambda^{d+2s}} K(|\mathbf{x}-\mathbf{y}|). \tag{4.20}$$

So, we can say that

$$K(\mathbf{x}-\mathbf{y}) = \frac{1}{|\mathbf{x}-\mathbf{y}|^{d+2s}} K(\mathbf{e}), \tag{4.21}$$

where $\mathbf{e}$ is any unit vector and $K(\mathbf{e})$ is a constant, independent of the choice of $\mathbf{e}$ (since K is rotationally invariant). Thus we have

$$K(\mathbf{e}) = \int_{\mathbb{R}^d} \frac{\mathbf{e}-\mathbf{z}}{|\mathbf{e}-\mathbf{z}|^{d+s+1}} \cdot \frac{\mathbf{z}}{|\mathbf{z}|^{d+s+1}} d\mathbf{z}. \tag{4.22}$$

In one dimension, we proved that $K(\mathbf{e})$ is a positive constant; this is confirmed by numerical tests.

5. Implications on the well-posedness. Paper [12] proves that, under certain conditions on the kernel function γ, the operator L_δ is associated with a coercive variational form, or, in other words, with an energy norm. This, in turn, provides well-posedness of the following diffusion problem:

$$\begin{cases} -L_\delta(u) = f & \text{in } \Omega \\ u = g & \text{in } \Omega_I. \end{cases}$$

Utilizing the above equivalence of $\mathcal{DG}$ and $\mathcal{D}_\omega\mathcal{G}_\omega$ along with the nonlocal Green's identity for weighted operators [8], we can show that the energy norm associated with the unweighted nonlocal operators is equivalent to that of weighted operators, thus providing well-posedness of diffusion problems such as

$$\begin{cases} -L_\omega(u) = f & \text{in } \Omega \\ u = g & \text{in } \Omega_I. \end{cases}$$

More specifically, for a symmetric kernel γ, the unweighted energy norm is defined as

$$|||u|||_\delta^2 = \int_{\Omega\cup\Omega_I}\int_{\Omega\cup\Omega_I}(u(\mathbf{x})-u(\mathbf{y}))^2\gamma(\mathbf{x},\mathbf{y})d\mathbf{y}d\mathbf{x}.$$

On the other hand, the weighted energy norm is defined as

$$|||u|||_\omega^2 = \int_{\Omega\cup\Omega_I}(\mathcal{D}_\omega u(\mathbf{x}))^2 d\mathbf{x}.$$

By applying the weighted nonlocal Green's identity [8] and defining γ as in (3.3), it is easy to show that $|||u|||_\delta = |||u|||_\omega$.

The extension of this equivalence to a broad class of nonlocal operators is the subject of our current research.

Acknowledgements. This research was supported by the INTERN award for NSF-DMS 1716790 (PIs: Petronela Radu and Mikil Foss). It was also supported by Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] A. A. Buades, B. Coll, and J. Morel, *Image denoising methods. a new nonlocal principle*, SIAM Review, 52 (2010), pp. 113–147.

[2] B. Alali and R. Lipton, *Multiscale dynamics of heterogeneous media in the peridynamic formulation*, Journal of Elasticity, 106 (2012), pp. 71–103.

[3] E. Askari, *Peridynamics for multiscale materials modeling*, Journal of Physics: Conference Series, IOP Publishing, 125 (2008), pp. 649–654.

[4] D. Benson, S. Wheatcraft, and M. Meerschaert, *Application of a fractional advection-dispersion equation*, Water Resources Research, 36 (2000), pp. 1403–1412.

[5] N. Burch, M. D'Elia, and R. Lehoucq, *The exit-time problem for a markov jump process*, The European Physical Journal Special Topics, 223 (2014), pp. 3257–3271.

[6] A. Delgoshaie, D. Meyer, P. Jenny, and H. Tchelepi, *Non-local formulation for multiscale flow in porous media*, Journal of Hydrology, 531 (2015), pp. 649–654.

[7] M. D'Elia, Q. Du, M. Gunzburger, and R. Lehoucq, *Nonlocal convection-diffusion problems on bounded domains and finite-range jump processes*, Computational Methods in Applied Mathematics, 29 (2017), pp. 71–103.

[8] M. D'Elia, M. Gulian, H. Olson, and G. E. Karniadakis, *A unified calculus for fractional, nonlocal, and weighted nonlocal models*, tech. rep., Sandia National Laboratories, 2019. *In progress*.

[9] M. D'Elia and M. Gunzburger, *The fractional laplacian operator on bounded domains as a special case of the nonlocal diffusion operator*, Computers and Mathematics with applications, 66 (2013), pp. 1245–1260.

[10] Q. Du, M. Gunzburger, R. Lehoucq, and K. Zhou, *Analysis and approximation of nonlocal diffusion problems with volume constraints*, SIAM Review, 54 (2012), pp. 667–696.

[11] ———, *A nonlocal vector calculus, nonlocal volume constrained problems, and nonlocal balance laws*, Mathematical Models in Applied Science, 23 (2013), pp. 493–540.

[12] Q. Du and X. Tian, *Stability of nonlocal dirichlet integrals and implications for peridynamic correspondence material modeling*, SIAM Journal of Applied Math, 78 (2018), pp. 1536–1552.

[13] P. Fife, *Some nonclassical trends in parabolic and parabolic-like evolutions*, Springer-Verlag, New York, 2003, ch. Vehicular Ad Hoc Networks, pp. 153–191.

[14] G. Gilboa and S. Osher, *Nonlocal linear image regularization and supervised segmentation*, Multiscale Model. Simul., 6 (2007), pp. 595–630.

[15] D. Littlewood, *Simulation of dynamic fracture using peridynamics, finite element modeling, and contact*, in Proceedings of the ASME 2010 International Mechanical Engineering Congress and Exposition, Vancouver, British Columbia, Canada, 2010.

[16] M. Meerschaert and A. Sikorskii, *Stochastic models for fractional calculus*, Studies in mathematics, Gruyter, 2012.

[17] A. Schekochihin, S. Cowley, and T. Yousef, *Mhd turbulence: Nonlocal, anisotropic, nonuniversal?*, in In IUTAM Symposium on computational physics and new perspectives in turbulence, Springer, Dordrecht, 2008, pp. 347–354.

[18] R. Schumer, D. Benson, M. Meerschaert, and B. Baeumer, *Multiscaling fractional advection-dispersion equations and their solutions*, Water Resources Research, 39 (2003), pp. 1022–1032.

[19] R. Schumer, D. Benson, M. Meerschaert, and S. Wheatcraft, *Eulerian derivation of the fractional advection-dispersion equation*, Journal of Contaminant Hydrology, 48 (2001), pp. 69–88.

[20] S. Silling, *Reformulation of elasticity theory for discontinuities and long-range forces*, Journal of the Mechanics and Physics of Solids, 48 (2000), pp. 175–209.

MODEL PRETRAINING FOR GRAPH-BASED LEARNING

JAMES FOX† AND SIVASANKARAN RAJAMANICKAM‡

Abstract. Model pretraining has been used to great effect in domains such as computer vision and natural language processing. However, it is a relatively new phenomenon in the area of graph-based learning, and is not well studied. In this work, we focus on two kinds of models, a standard multi-layer neural network, as well as a more recent graph neural network. We focus on *structural identity* predictions as the downstream task for pretrained models, where the model is informed by graph structure alone. We consider and evaluate two different modes of using pretraining, full and partial fine-tuning. We find that in most cases pretraining does not hurt overall performance, and in some cases can improve convergence and final accuracy.

1. Introduction. Transfer learning has received a lot of attention in deep learning, and has been used to achieve state of the art performance in domains such as computer vision and natural language processing.

Broadly speaking, transfer learning involves using a model learned in one context, as a starting point for learning in a different context. This can enable faster model convergence, greater robustness to noise, final predictive accuracy, and other benefits [9] compared to training a model from scratch. Typically the transfer learning is achieved by first *pretraining* the model on some auxiliary task(s) and/or data, prior to training on the target data and task. Pretraining is particularly attractive when there are few labeled or ground-truth data for the target task, but a larger body of data (labeled or unlabeled) still related in some way to the target task and data.

Pretraining has long been applied and studied in the computer vision community [4][17]. Recent examples of pretraining in the NLP community include the BERT [2] model, which leverages unlabeled data and unsupervised tasks to pretrain a NLP model, before *fine-tuning* to a supervised target task. This achieves state-of-the-art performance, and outperforms directly optimizing for a specific task.

The success of pretraining in other deep learning domains motivates a study of whether pretraining can also be applied to models for learning over graph-structured data. This topic has only very recently started to receive some attention [11][10]. Learning on graph-structured data poses additional challenges compared to domains with more regular data, which is that the structure of the data itself can vary. This is challenging for both model design and prospects of transfer learning.

In this work, we study pretraining over graphs from the perspective of *structural* predictions on nodes. In other words, we assume the graph has no additional information other than its structure and training labels. In general, graphs could have additional domain information over nodes and/or edges that are informative. We separately consider two models for learning over graph-structured data: a simple multi-layer neural network, and a recent instance of the graph neural network (GNN) model, the Graph Isomorphisn Network [16]. We refer to Fig. 1.1 for a conceptual diagram of the GNN model.

We evaluate pretraining of these models on two different datasets, one based on real-world data and one synthetic.

We provide an overview of related work in Section 2. We discuss our methodology and results in Section 3. In Section 4 we summarize conclusions and look at future work.

†Georgia Institute of Technology, jfox43@gatech.edu
‡Sandia National Laboratories, srajama@sandia.gov

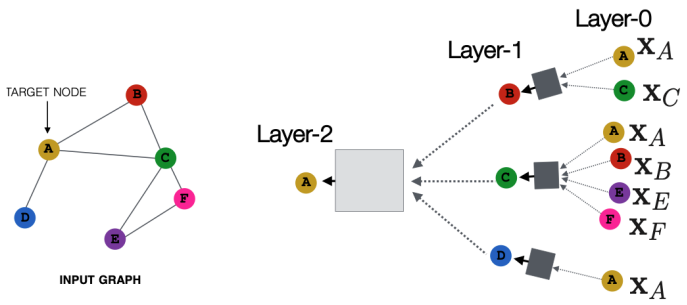


Fig. 1.1: Conceptual illustration of GNN from [7]. Features of neighbors are aggregated and mapped using nonlinear transformations, in each layer.

2. Related Work.

2.1. Node Embeddings. Node embeddings are vector representations of nodes in graphs, which can then be used in machine learning models to make predictions. Methods such as *DeepWalk* [13] and *node2vec* [6], use random walks to generate embeddings that reflect local proximity of nodes.

However, these embeddings do not adequately address *structural* similarity of nodes, where two similar nodes may not even be connected. Rolx [8], Struc2vec [14] and Graphwave [3] focus on embeddings in this context. These embeddings are then used for downstream tasks that require learning structural identity. This context is also the focus of our work.

2.2. Graph Neural Network. Kipf et. al. [12] presented the Graph Convolutional Network, which introduced the notion of convolutions to graph representation learning and outperformed existing methods. Since then, many other variations have been proposed. Xu et. al. [16] provide a theoretical framework for generalizing these variations, and also propose a theoretically optimal variation in this class. For consistency and simplicity, we refer to this class of models as *Graph Neural Network* (GNN) in the remainder of this paper. We consider the models of [12] and [16] to be instances of the GNN.

2.3. Pretraining of Models for Graph-structured Data. W. Hu et. al. [10] propose combining node-level self-supervised pretraining with graph-level supervised pretraining, to pretrain GNNs of the type in [16]. The node-level tasks are unsupervised, inspired by masking and context prediction from NLP. The graph-level pretraining uses "coarse-grained" labels from chemical/biological graphs for pretraining, before fine-tuning and predicts "fine-grained" labels on a different target dataset from the same domain. The authors show consistent performance improvement in graph prediction accuracy with their pretraining strategy.

Z. Hu et. al. [11] considers pretraining of GNNs based on graph structure alone, i.e. domain features are not available or not used. The authors propose multi-task learning using a variety of self-supervised tasks, from community detection to centrality prediction. Random power-law graphs are used for pretraining, and ground-truth for these tasks is obtained from running standard graph algorithms. The authors report significant improvement in test accuracy from pretraining, but note that even with pretraining, using structure alone is not competitive in cases where domain features are available. Our setting most closely resembles that of Z. Hu et. al., as we also considered graphs without domain features.

3. Content.

Table 3.1: Airports dataset

Name	Nodes	Edges	Diameter	Classes
Brazil	131	1038	5	4
Europe	399	5995	8	4
USA	1190	13599	5	4

3.1. Multi-Layer Perceptron Model. We investigate pretraining of a multi-layer perceptron (MLP), or vanilla neural network model, using only structural features. We pretrain and test on a set of airport graphs, originally introduced in [14]. More detail on dataset is given in later sections.

Pretraining Methodology. In our experiments, we pretrain the model on one graph, before using the weights of the pretrained model as the starting point for training on the target dataset. There are different possibilities for how to adapt the pretrained model for the downstream task–we describe two variations of the *fine-tuning* approach in this work. In *full* fine-tuning, all layers of the model can still be updated after pretraining. In the *partial* fine-tuning version, only the weights of output layers (and possibly supplementary layers such as Batchnorm) are updated, while the weights of all other layers are frozen. In the case of the MLP model, we only applied full fine-tuning.

In either case, some minimal number of training samples are needed from the target dataset, because in general the classes of the target task may not match up, or even be of same cardinality, with those of the the pretraining task. In the partial fine-tuning case, this can be interpreted as just training the classifier portion of the model, e.g. the layer(s) closest to output.

Dataset. The airports datasets are from [14], and were originally derived from air-traffic networks. The task is to predict how busy each airport is (classes are quartiles), based on network connectivity alone. See Table 3.1 for data summary.

Experiments. For pretraining the MLP model for predictions over the airports dataset, we use one graph from the dataset as pretraining target, and a different graph as the test target. We assume that all labels of the pretraining graph are available, while only a subset of the test graph's is available for training.

Experiments are split across two main hyperparameters: the number of training data available (as ratio of total labels) for the target task, and the number of epochs the model was trained for. We use 0.1 and 0.6 as training ratios, to loosely represent scarce and abundant training data cases. The number of epochs is split into two settings: 20 vs. 200 epochs. 200 epochs represents training to convergence, while 20 epochs is pre-convergence. We run each combination of the two hyperparameters, for a total of 4 experiments. Each experiment is averaged over 100 trials.

Results. We present the experimental results over 4 tables, each corresponding to a specific configuration of the training ratio and training epochs hyperparameters.

	Brazil	Europe	USA
None	0.585	0.509	0.54
Brazil	x	0.486	0.57
Europe	0.582	x	0.552
USA	0.624	0.507	x

Table 3.2: 0.1 training ratio; trained 20 epochs

	Brazil	Europe	USA
None	0.62	0.53	0.57
Brazil	x	0.54	0.588
Europe	0.63	x	0.583
USA	0.67	0.55	x

Table 3.3: 0.1 training ratio; trained 200 epochs

	Brazil	Europe	USA
None	0.648	0.536	0.557
Brazil	x	0.506	0.576
Europe	0.642	x	0.57
USA	0.726	0.539	x

Table 3.4: 0.6 training ratio; trained 20 epochs

	Brazil	Europe	USA
None	0.7	0.59	0.6
Brazil	x	0.589	0.601
Europe	0.725	x	0.599
USA	0.754	0.587	x

Table 3.5: 0.6 training ratio; trained 200 epochs

The row label indicate the graph used for *pretraining*, while the column label indicates the *target* graph used for fine-tuning and testing. The first row represents the baseline: no graph is used for pretraining, and so the model is trained from scratch on the target graph. Table entries are test accuracies from each pretraining and target graph pair. Results exceeding 2% relative different from their respective baseline are indicated by either red or green colors, corresponding to whether the difference was negative or positive. We ignore pretraining and testing on the same graph in the table, as this is a trivial case.

Table 3.3 shows the results of pretraining when the model is trained to convergence (using 10% training data) on the target dataset. The results show that the USA dataset is clearly the most impactful, both for pretraining and as the target. It is interesting that pretraining on Brazil and Europe improves final model accuracy on USA, as both are considerably smaller than USA–Brazil is roughly a magnitude smaller in terms of nodes.

We also see from Table 3.2 that in general, experiments involving the USA dataset also converge faster (at 20 epochs), compared to without pretraining. There is one instance where pretraining on Brazil actually is hurting model training at 20 epochs, but this deficit largely goes away when trained to 200 epochs.

Table 3.5 shows the at-convergence results when 60% of the target data is available for training. Compared to with 10% training data case, there are less cases where pretraining has any significant impact. Only the Brazil graph benefits from pretraining (on Europe and USA graphs). Table 3.4 mirrors trends from Table 3.2; in the 60% training data case, there could still be benefit in terms of how fast the model converges, with pretraining.

Implementation Details. The MLP model was implemented in PyTorch. The MLP has 3 fully-connected layers with ReLU activation, each with a hidden dimension of 32. Dropout is applied at each intermediate layer. We use softmax output with negative log likelihood loss.

For initial features, we used a simple set of 5 structural features, inspired from Cai et. al. [1]. These are the node degree, as well as the min, max, average, and standard deviation of its 1-hop neighborhood degrees. We normalize each feature to have zero mean and standard deviation of 1. There are many other possibilities for extracting initial features, such as those described in Sec. 2.

We reserve 20% of target data for validation, and split the rest among training and test sets (depending on training ratio). The test accuracy corresponds to the model with the best validation score within the max number of training epochs. In pretraining, we use all labels and simply train for 200 epochs (without using a validation or test set).

3.2. Graph Neural Network. We also wanted to study the effect of pretraining on a more recent type of model, the GNN. Specifically, we wanted to better understand how difference in graph structure impacts pretraining of models designed for graph structured data (such as the GNN). It often is the case that the target graph will be structurally different from ones seen during pretraining.

To study this in a controlled manner, we propose using synthetically generated graphs with distinct structural identities. As in earlier settings, there are no domain features; the

only features are what can be extracted from the graph structure itself. The node structural identities serve as the train/test labels over the graph.

To vary the structure, we create new graphs by randomly adding edges to the previously regular base graph, as a ratio of the number of original edges. We maintain the same set of labels as from the base graph. We expect that node labels are not injective with structural identity in real-world datasets, but that there is nontrivial correlation. This process of generating new versions of the graph is intended to explore that correlation, in terms of how sensitive the GNN model is to structural "noise". We use the Graph Isomorphism Network (GIN) [16] variant for all experiments.

Dataset.

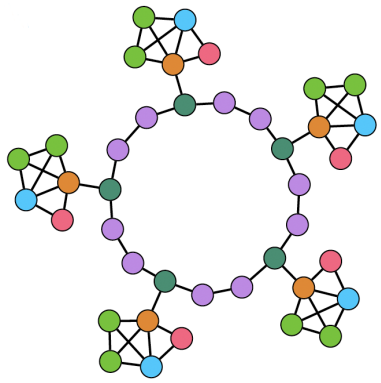


Fig. 3.1: "Ring of houses" graph structure. Concept and image from [3]. Each color corresponds to distinct node class.

We chose to generate a synthetic graph with structural identity labels as defined in Donnat et. al. [3]. Figure 3.1 shows a smaller version of the graph. In practice, we generate a larger graph, while maintaining the same ratio of class labels as in the figure, by simply extending the ring and attaching more house structures at regular intervals. This method allows for graphs with interpretable and intuitive structural labels. The base graph we generate has 2664 nodes, 3996 edges, and 6 classes of labels.

Model Details. For the GNN, we implement the GIN model from [16] using the PyTorch Geometric [5] libary. Our model uses 3 GIN layers, followed by two fully connected layers (the last of these two is output). Hidden dimension is 32, across all weights.

Each GIN layer contains two fully connected layers (applied after aggregation). Every GIN layer is accompanied by a Batchnorm layer, which proves quite beneficial to classification performance in our case.

The initial node feature fed as input to the model is just the degree of the node (single scalar). The input is normalized in same fashion as in described in Sec. 3

To establish the baseline that pretraining is compared to, we look at the accuracy of the GIN model without any pretraining.

3.2.1. GNN Baseline.

Varying noise. We vary the ratio of noisy edges added to the base graph in increments of 0.05, and examine the accuracy of the model (when trained on each graph version). A number of edges are added uniformly at random to the graph (two nodes are randomly picked), as a ratio of the original number of edges in the graph. Results are shown in Fig.

3.2. Each violin plot summarizes the distribution from 50 trials, and this is the same for remaining experiments.

The first notable trend is that the GNN learns to classify labels in the original graph, which has no noisy edge additions, almost perfectly. This result is well within the theoretical expressiveness of the GNN model, due to its connection to the Weisfeiler-Lehman isomorphism test [16].

Despite the theoretical connection, this expressiveness is not always learned in practice, and is model-dependent. We found that 3 GNN layers were needed to achieve perfect accuracy, whereas only 1 iteration of the WL test should be sufficient to recover all structural identities in the base graph.

Although we reported results for using 20 training samples per class, we found that using even 1 training sample per class is sufficient to achieve near-perfect accuracy on the base graph. The accuracy drops sharply with each additional 5% of noise added to the base graph, but the curve begins to bottom out around 30% noise. For reference, adding 25% noise to the graph reduces overall accuracy of the model by almost 50%.

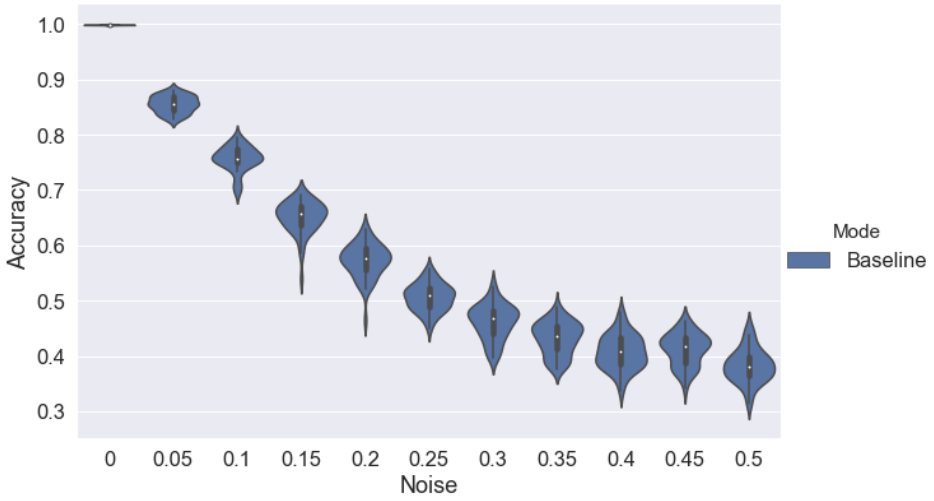


Fig. 3.2: Accuracy vs. % noisy edges added. Number of training samples per-class fixed at 20. Each violin plot summarizes 50 trials.

Varying training samples. While we fixed the number training samples at 20 per class in Fig. 3.2, the accuracy of the GNN model can also depend strongly on the number of training samples. We consider the case of the graph with 10% noise, and vary the number of samples-per-class at 1, 5, 20, 50, and 200. Results are shown in Fig. 3.3. Adding more samples results in greater performance improvement in the lower regime, with diminishing returns at higher samples-per-class. For instance, 200 samples per class corresponds to nearly 50% of the total labels. We expect that on noisy graphs, more labels are needed on average to account for variation of structure within the same class, which does not apply to the base graph.

3.2.2. GNN Pretraining. In this section, we investigate the following question: *How effective is pretraining of GNNs, as we vary the structural difference between the pretraining and target graph?* The original base graph G is the pretraining target in all instances. We also adopt a "noise" perspective with respect to the identities of G'–the labels of the target

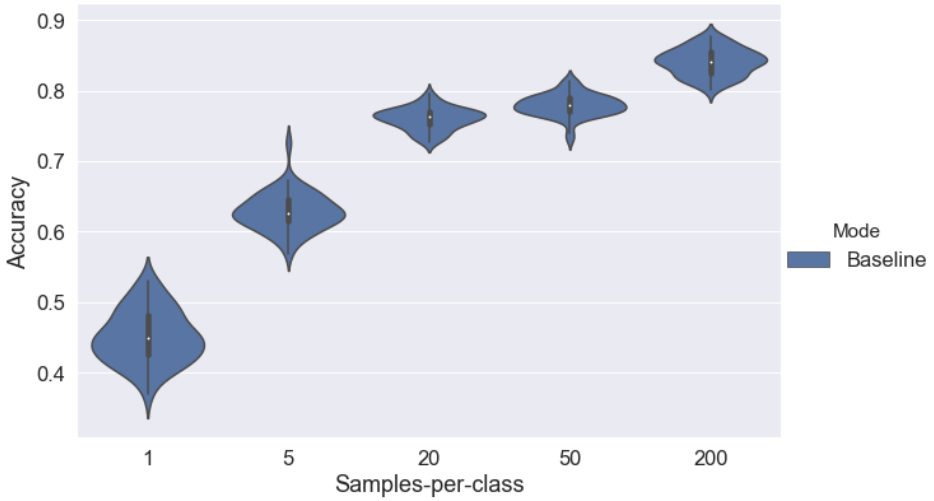


Fig. 3.3: Accuracy vs. varying number of training samples per class. Graph is fixed at 10% noise.

graph G' are unchanged from those of G, even if node structural identities have technically changed based on edge additions. Optimal accuracy could be obtained in our setting by simply copying the test predictions from the model pretrained on the base graph, as the predictions for the target graph.

An alternative perspective would be to relabel nodes in G' according to their new structural identities, e.g. using the WL test [15], and use the new labels as prediction targets.

Pretraining Setup. After pretraining the GNN on the base graph, we consider two variations for *fine-tuning* the model on a target dataset, referred to as "Full" and "Partial" in Fig. 3.4. The distinction between these two approaches were detailed in Sec. 3.1. In the partial fine-tuning version, the weights of both the output and Batchnorm layers are updated (the weights of all other layers are frozen). Similarly to the experiment in Fig. 3.2, we fixed the number of samples per class at 20, and vary the noise ratio in increments of 0.05. We refer to results in Fig. 3.4.

Results. There does not appear to be a clear relationship between the level of noisy edges added, and the effect of pretraining in our experiment. Accuracy with pretraining was helpful in some cases where noise level was higher, and not so helpful at lower noise–and vice versa. Across all target graphs, the relative performance improvement with pretraining, with respect to the average, ranges from almost none up to 10% (taking the best of the two fine-tuning modes). Fig. 3.4 provides more complete information than just the average; each violin plot summarizes the distribution from 50 trials.

In the full fine-tuning case, the final accuracy with pretraining is always better than or at least comparable to that without pretraining, in both the mean and median. This suggests that the model weights in the pretrained model are at least as good as those from random initialization.

In the partial fine-tuning case, the results are more varied across different noise levels. In several instances, the distribution with partial fine-tuning is clearly the best compared to baseline and full fine-tuning, e.g. at noise ratios 0.2, 0.3, and 0.45. However, in some other instances the results can be worse than even the baseline, with noise level 0.15 being

the notable one. While we're not entirely sure why this is, one possibility is the dependency of pretraining on the variation of the noisy graph itself. We only randomly generated one noisy graph per noise level; summarizing over multiple versions of each noise graph might have been informative.

In terms of variation, in several cases the partial fine-tuning mode significantly reduces the spread of the distribution compared to both the baseline and full fine-tuning.

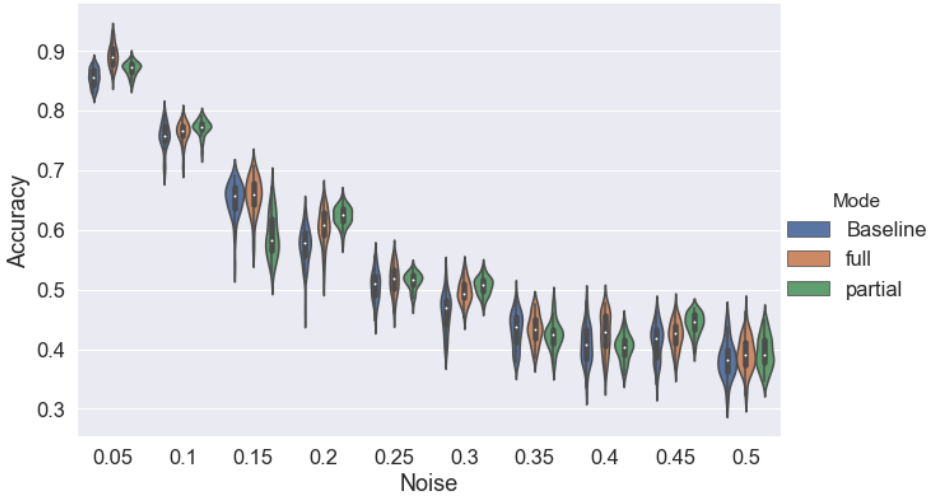


Fig. 3.4: "Full" and "Partial" refer to modes of using the pretrained model. Baseline is without any pretraining.

Implementation Details. We use the following default train-validation-test split when fine-tuning on the target graph (different from in Sec. 3): 20 samples per class for training, 200 total samples for validation, and 1000 total samples for testing (all randomly drawn). We chose to represent label scarcity in training, which we believe is more reflective of node predictions in the transductive setting.

We also use a train-validation split during pretraining. These same data splits are preserved across all experiments. We use the pretrained model corresponding to the best seen validation accuracy over 200 epochs, to evaluate final (test) accuracy.

4. Conclusions. We investigated pretraining of two different models for graph-structured data: a vanilla neural network, as well as the recent Graph Neural Network. We pretrained an MLP on airports dataset, where structural role is believed to be important. Pretraining is beneficial in some cases, but results are graph-dependent.

We also evaluated pretraining on a GNN model, but with more emphasis on controlling differences in graph structure. Our experiments on a synthetically generated graph show that the GNN can learn to perfectly distinguish structure, with minimal feature and label information. On the other hand, it can be very sensitive to addition of edge noise, falling to 50% median accuracy with 25% added edges. Finally, we evaluate pretraining of a GNN model for two different modes of fine-tuning, and show that it can help across a wide range of noise levels. However, results also depend on the fine-tuning method.

We believe there remain opportunities for characterizing robustness of pretraining in the context of families of graphs, including those of interest in graph generation.

REFERENCES

[1] C. Cai and Y. Wang, *A simple yet effective baseline for non-attributed graph classification*, 2018.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of deep bidirectional transformers for language understanding*, arXiv preprint arXiv:1810.04805, (2018).

[3] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec, *Learning structural node embeddings via diffusion wavelets*, in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, 2018, pp. 1320–1329.

[4] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, *Why does unsupervised pre-training help deep learning?*, Journal of Machine Learning Research, 11 (2010), pp. 625–660.

[5] M. Fey and J. E. Lenssen, *Fast graph representation learning with PyTorch Geometric*, 2019.

[6] A. Grover and J. Leskovec, *node2vec: Scalable feature learning for networks*, in Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2016, pp. 855–864.

[7] W. L. Hamilton, R. L. Ying, J. L. Leskovec, and R. L. Sosic, *Www18 representation learning on networks tutorial*.

[8] K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li, *RolX: structural role extraction & mining in large graphs*, in Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2012, pp. 1231–1239.

[9] D. Hendrycks, K. Lee, and M. Mazeika, *Using pre-training can improve model robustness and uncertainty*, arXiv preprint arXiv:1901.09960, (2019).

[10] W. Hu, B. Liu, J. Gomes, M. Zitnik, P. Liang, V. Pande, and J. Leskovec, *Pre-training graph neural networks*, 2019.

[11] Z. Hu, C. Fan, T. Chen, K. Chang, and Y. Sun, *Pre-training graph neural networks for generic structural feature extraction*, CoRR, abs/1905.13728 (2019).

[12] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, arXiv preprint arXiv:1609.02907, (2016).

[13] B. Perozzi, R. Al-Rfou, and S. Skiena, *DeepWalk: Online learning of social representations*, in Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2014, pp. 701–710.

[14] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo, *struc2vec: Learning node representations from structural identity*, in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 385–394.

[15] B. Weisfeiler and A. A. Lehman, *A reduction of a graph to a canonical form and an algebra arising during this reduction*, Nauchno-Technicheskaya Informatsia, 2 (1968), pp. 12–16.

[16] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, *How powerful are graph neural networks?*, arXiv preprint arXiv:1810.00826, (2018).

[17] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, *How transferable are features in deep neural networks?*, in Advances in neural information processing systems, 2014, pp. 3320–3328.

IDENTIFICATION OF NONLOCAL KERNEL PARAMETERS VIA NONLOCAL PHYSICS-INFORMED NEURAL NETWORKS

NICOLE E. BUCZKOWSKI*, MARTA D'ELIA†, AND MICHAEL L. PARKS‡

Abstract. In the context of partial differential equations and fractional partial differential equations, Physics-informed neural networks (PINNs) and fractional PINNs (fPINNs) utilize neural networks to obtain approximations of solutions or solve parameter identification problems. In this work we extend these methods to nonlocal equations and we investigate the properties of this new algorithm. The latter includes convergence with respect to number of training points and neural network parameters. Also, we provide the groundwork for the identification of nonlocal kernel parameters.

1. Introduction. Neural networks are computing systems inspired by biological brains and have been used in a wide variety of applications. In the context of solving partial differential equations (PDEs), physics-informed neural networks (PINNs) [16], which work by incorporating known PDEs into the loss function, have shown remarkable promise. Work on PINNS has been extended to include fractional PDEs, resulting in fractional physics-informed neural networks (fPINNS) [15]. In this paper we explore the extension of PINNs to nonlocal equations, known as nonlocal physics-informed neural networks (nPINNS). In Section 2 we provide a brief introduction to nonlocal models, and in Section 3 we provide a brief introduction to neural networks. In Section 4 we describe the nPINNs algorithm for both forward and inverse problems. In Section 5 we provide numerical results that illustrate the consistency and applicability of the algorithm. In particular, for the forward problem we analyze the sensitivity of nPINNs to number of training points, network parameters, optimization parameters and noise. For the inverse problem we consider both single- and multi-parameter identification for polynomial kernels. We offer conclusions in Section 6.

2. Nonlocal Models. Nonlocal models have become popular for the description of a broad class of scientific and engineering applications including fracture mechanics, image processing, stochastic processes and many others [1, 2, 3, 4, 5, 6, 7, 9, 11, 13, 14, 17, 18, 19, 20].

Their main feature is that instead of using partial differential operators, they utilize integral operators. The integral form, over a ball of radius δ or over the whole space, allows for long-range interactions and reduces the regularity requirements on the solution. Specifically, every point x in a domain, interacts with a neighborhood of points, $B_\delta(x)$, where δ can be either finite or infinite. Given a bounded domain, points on the boundary also share this property; as a consequence, such points interact with points outside of the domain in a collar, or a nonlocal boundary, of thickness δ. The latter, being the nonlocal counterpart of a local boundary, is the set where nonlocal boundary conditions must be prescribed to guarantee well-posedness of nonlocal equations [8].

In this paper we consider operators of the form

$$\mathcal{L}u = 2\int_{\mathbb{R}} (u(y) - u(x))\gamma(x,y)dy, \tag{2.1}$$

which we refer to as the *nonlocal Laplacian*. Here, γ is a symmetric kernel with bounded support, i.e.

$$\gamma(x,y) = k(x,y)\mathcal{X}_{B_\delta(x)}(y). \tag{2.2}$$

*University of Nebraska-Lincoln, nbuczkowski@huskers.unl.edu
†Sandia National Laboratories, mdelia@sandia.gov
‡Sandia National Laboratories, mlparks@sandia.gov

Without loss of generality, we work in a one-dimensional setting. With this assumption and based on (2.2), we rewrite (2.1) as

$$-\mathcal{L}u = 2\int_{x-\delta}^{x+\delta} (u(y) - u(x))k(x,y)dy. \tag{2.3}$$

Note that, when properly scaled, the nonlocal Laplacian converges to the classical Laplacian as $\delta \to 0$ [10]. We are interested in the solution of the following nonlocal diffusion problem

$$-\mathcal{L}u = f \text{ on } \Omega \tag{2.4}$$

$$u = g \text{ on } \Gamma, \tag{2.5}$$

where the nonlocal boundary Γ is defined as $\Gamma := \{y \in \mathbb{R}^n \setminus \Omega : y \in B_\delta(x) \text{ for some } x \in \Omega\}$. The constraint on Γ is a nonlocal counterpart to a Dirichlet boundary condition for classical partial differential equations.

Equations like (2.4) are not simple to solve; in particular, model parameters or the functional form of the kernel may be unknown or subject to uncertainty. The primary objective of this work is the identification of kernel parameters in the simple case of polynomial kernels. These are the first steps towards one of the most important open problems in nonlocal modeling, namely kernel calibration.

3. Neural Networks. Neural networks are a powerful tool used in machine learning. When data is input into a neural network, it travels through nodes called neurons with their own activation functions employing weights and biases, returning new outputs to be fed to the next layer. This process is iterated until an output is produced. Further we can use training data to tune these weights and biases for these activation functions to return more desirable outputs. We do this through backpropagation, a process in which the optimizer works backward through the neurons, varying them slightly and observing the effect on some loss function. The optimizer works to get this loss function to its minimum. The amount that we allow these weights and biases to vary by in training is called the learning rate. We can also widen (increase the amount of neurons in a layer) or deepen (increase the amount of layers) to allow for finer tuning of the neural network.

We apply this machine learning approach to model nonlocal equations, in the spirit of PINNs and fPINNs. Neural networks are powerful tools that have also become more widely used and studied over recent years. These neural networks have opened a door for significant human advancement in upcoming years. In PINNs and fPINNs, it was shown that by including information about the PDE or fractional PDE into the loss function to be optimized, one can estimate a solution to the given PDE by manipulating the weights and biases in the neural network. This idea was expanded upon to the inverse problem to add certain parameters to be identified in the PDE itself. Since these parameters are sometimes unknown in many real-world applications, we can not incorporate them into the loss function directly, only incorporate them indirectly by giving the neural network more information about u and $\mathcal{L}u$. To incorporate this information, one can measure the forcing term of the PDE at specific sets of points.

We apply the recently introduced nPINNs (nonlocal PINNs), to the identification of the solution to the nonlocal model and some model parameters, such as kernel parameters, in generalized nonlocal operators such as (2.4). We train the optimizer to find solutions u to the nonlocal equation with the structure given earlier. We later train the optimizer to additionally find parameters in the kernel. These parameters are optimized alongside the weights and biases in the neurons in the neural network.

We discretize integrals with Gaussian quadrature. We are able to include many training points in the nonlocal boundary, to utilize in the loss function to enforce boundary conditions

on the solution. We are able to use the code similar to that used in fPINNs in the inverse problem. We use these methods to explore cases where the nonlocal Laplacian will converge to the classical Laplacian.

4. Methodology. We present the algorithm utilized in this paper. While the formulation is the same for the solution of forward and inverse problems, we have a few key differences, which are highlighted throughout the section.

First, we collect measurements of the solution and forcing term at points in different training sets. While training points for f are restricted to Ω, the solution can be collected on both Ω and Γ. More specifically, for the solution of forward problems, we only collect solution data in Γ, whereas for inverse problem we collect solution data in $\Omega \cup \Gamma$ in order to improve the conditioning of the problem. We assume that these measurements are either exact or are the result of high-fidelity simulations.

Second, we approximate the solution u with a fully connected neural network, i.e. $u \cong u_{NN}$, and we rewrite the kernel as a function of $(x, y; \mathbf{d})$, where $\mathbf{d} = \{d_1, d_2, \ldots\}$ is a set of unknown parameters that we would like to identify. Examples of $\mathbf{d}$ could be the interaction radius δ or, in case of polynomial kernels, the coefficients of the polynomial itself. Of course, when nPINNs is solely used for the solution of forward problems, the parameter set is empty and the only unknowns are the network parameters (that fully determine the approximate solution u_{NN}).

Then, biases, weights and model parameters are obtained as the result of the following optimization problem:

$$\begin{aligned} \min_{u_{NN},\mathbf{d}} loss &= loss_u + loss_{\mathcal{L}} \\ &= \sum_{i=1}^{N_u} (u_{NN}(x_i) - u(x_i))^2 + \sum_{j=1}^{N_f} (\mathcal{L}u_{NN}(x_j) - f(x_j))^2. \end{aligned} \tag{4.1}$$

where N_u and N_f are the number of measurements (or training points) of the solution and the forcing term. In this work we consider a simplified setting:
– for forward problems $\{x_j\}_{j=1}^{N_f} \subset \Omega$ and $\{x_i\}_{i=1}^{N_u} \subset \Gamma$;
– for inverse problems $\{x_j\}_{j=1}^{N_f} \subset \Omega$ and $\{x_i\}_{i=1}^{N_u} = \{x_j\}_{j=1}^{N_f} \cup \{x_k\}_{k=1}^{N_g} \subset \Omega \cup \Gamma$, where $\{x_k\}_{k=1}^{N_g} \subset \Gamma$.

Note that in (4.1), the first term is the mismatch between measured data and neural network and the second term is the residual of the nonlocal equation. As opposed to PINNs, where the operators are differential and the chain rule applies, we cannot easily apply the backpropagation algorithm on nonlocal operators, as there is no chain rule in the nonlocal calculus. Thus, before solving the optimization problem, the operator $\mathcal{L}$ must be discretized. Specifically, we use Gaussian quadrature to compute an approximation of the integral

$$\mathcal{L}u(x_j) = 2\int_{x_j-\delta}^{x_j+\delta} (u(y) - u(x_j))k(x_j, y; \mathbf{d})dy \;\; \forall\, j = 1, \ldots N_f, \tag{4.2}$$

which gives

$$\mathcal{L}u(x_j) \cong 2\delta \sum_{q=1}^{M} (u_{NN}(y_q) - u_{NN}(x_j))k(x_j, y_q, \mathbf{d})w_q, \tag{4.3}$$

where $y_q = x_j + \delta x_g$. Here, M is the number of Gauss points, and x_g and w_q are the associated points and weights.

5. Numerical tests. In this section we report the results of numerical tests for both the forward and inverse problem. To assess the accuracy of u_{NN} and $\mathcal{L}u_{NN}$ we consider the following errors:

$$e_u = \frac{\|\mathbf{u} - \mathbf{u}_{NN}\|_2}{\|\mathbf{u}\|_2}, \tag{5.1}$$

where $\mathbf{u}$ and $\mathbf{u}_{NN}$ are vectors of the values of u and u_{NN} at some validation points (different from the training points) in Ω. Here u is a manufactured solution used for testing purposes and will be specified later.

Note that in all our tests, except for the ones in Section 5.1.4, the training set is assumed to be exact. By this we mean that we use a manufactured solution to generate u and f at training points. Instead, in Section 4.1.4, the forcing term is subject to noise.

5.1. Results for Forward Problem. We consider the one-dimensional domain $\Omega = (0, 1)$ with $\delta = 0.01$, so that $\Gamma = (-\delta, 0) \cup (1, 1 + \delta)$. Unless otherwise noted, in all tests we use a width of 4, a depth of 4, a learning rate of 5e−5, 20 Gaussian quadrature points, $N_f = 50$, and $N_u = 100$ (recall that for forward problems training points for the solution belong to Γ; we place 50 points in each side of the nonlocal boundary). All training points are linearly spaced. We analyze the behavior of the loss functions and of the discretization error e_u as a function of the sampling size and the NN parameters. Later in this section, we also look at the effects of Gaussian white noise.

We use the following manufactured solution and kernel function

$$\begin{cases} u = x^2 - x^4 \\ k(x, y) = \frac{3}{2\delta^3} \\ f = -\frac{6}{5}\delta^2 - 12x^2 + 2. \end{cases} \tag{5.2}$$

As anticipated above, note that the kernel function is fully determined, i.e. does not depend on unknown parameters.

5.1.1. Quadrature Error. As we are using Gaussian quadrature and a smooth manufactured solution to estimate the integral of our predicted u_{NN}, we know that the error is $O(u^{(2m)})$ where m is the number of quadrature points [12]. In our experiments we choose m sufficiently large that to guarantees the quadrature error is negligible with respect to other sources of error.

5.1.2. Error with respect to the number of training points. We analyze the behavior of the approximation error with respect to the number of training points. We first consider varying the number of points in the domain N_f, then the points in the collar N_u, then both simultaneously. Reported results are the outcome of the optimization algorithm after 100000 iterations.

Table 5.1 displays results for a constant number of points in the nonlocal boundary, $N_u = 100$, while varying the points in the domain. Instead, in Table 5.2 we report results obtained for fixed N_f. We observe stagnation of the error after $N_f = 40$. Table 5.3 shows the results of varying number of points in the collar and in the domain.

As expected, at first the error decreases and then saturates. It appears from Table 5.3 that using more than 20 points in the domain and 20 points in each side of the nonlocal boundary does not improve the accuracy of the solution; thus, with the purpose of saving

Table 5.1: Varying Training Points in the Domain

N_f	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
5	3.63460e-04	1.22526e-04	2.40934e-04	1.55141e+00
10	1.25423e-06	4.19117e-11	1.25419e-06	3.02005e-04
20	1.68872e-06	2.09605e-12	1.68872e-06	2.00583e-05
40	2.56366e-07	8.68733e-13	2.56365e-07	1.25550e-05
100	6.18119e-07	2.36851e-11	6.18095e-07	2.03109e-05

Table 5.2: Varying Training Points in the Collar

N_u	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
5	1.60446e-03	2.23308e-05	1.58213e-03	5.60302e-02
10	7.84222e-07	2.91602e-12	7.84220e-07	1.35955e-05
20	3.28161e-06	1.13667e-08	3.27025e-06	4.01813e-04
40	2.41552e-06	3.33785e-11	2.41548e-06	2.49265e-05
100	2.09526e-06	7.44621e-12	2.09525e-06	1.84922e-05

computational time, there is no reason to use a larger training set. However, the in presence of non-smooth or highly oscillatory functions, a larger training set becomes imperative to guarantee accuracy. For example, for $f(x) = \cos(40\pi x)$ would require a much higher number of training points in the domain to avoid obtaining trivial solutions.

5.1.3. Error with respect to NN parameters and learning rate. We first analyze the behavior of the loss function and of the solution error with respect to the width and depth of the neural network. Results are reported in Tables 5.4 and 5.5.

Here, we observe that a width and depth of 4 yields a faster convergence. Thus we set such network parameters to 4 in all the tests in this section.

Table 5.4: Varying Width

Width	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
1	2.80887e+00	1.31996e+00	1.48892e+00	4.32233e+00
2	8.69865e-01	7.85086e-01	8.47786e-02	3.29125e+00
3	4.48052e-06	1.79253e-11	4.48050e-06	3.37583e-05
4	2.86862e-06	6.53523e-11	2.86856e-06	8.43196e-05
5	7.24309e-07	4.05502e-12	7.24305e-07	1.23031e-05
6	2.04371e-07	8.93422e-14	2.04371e-07	6.55593e-06
7	2.12416e-07	8.37183e-14	2.12416e-07	4.40106e-06

Table 5.3: Varying Training Points in the Domain and the Collar

N_u and N_f	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
5	1.82151e-05	1.82120e-05	3.07717e-09	9.10997e-01
10	2.96385e-07	3.27248e-09	2.93113e-07	2.59391e-03
20	2.14771e-07	5.42831e-13	2.14770e-07	6.84685e-06
40	1.18391e-06	4.31223e-11	1.18387e-06	2.92314e-05
100	1.37480e-06	5.78497e-12	1.37479e-06	1.67161e-05

Table 5.5: Varying Depth

Depth	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
1	2.51944e-05	5.41200e-10	2.51939e-05	1.87002e-04
2	1.45618e-06	1.90645e-12	1.45618e-06	1.39006e-05
3	6.16372e-07	1.07199e-11	6.16361e-07	1.49121e-05
4	1.47492e-06	4.91172e-11	1.47487e-06	3.00990e-05
5	2.08790e-07	1.10059e-11	2.08779e-07	2.95736e-05
6	2.31140e-07	2.77799e-13	2.31140e-07	3.64803e-06
7	2.32935e-07	1.92277e-12	2.32933e-07	7.05052e-06
20	6.81500e-06	6.20867e-09	6.80880e-06	2.96695e-04

We next consider the impact of the learning rate α. We first consider several constant learning rates in Table 5.6. We observe that for $\alpha <$ 5e-5 and $\alpha >$ 5e-3 the loss and errors are significantly higher. We take a closer look at $\alpha =$ 5e-3, 5e-4, 5e-5 in Tables 5.7, 5.8, and 5.9 respectively. Note that error and losses initially decrease and then their behavior becomes oscillatory; for higher values of α this is probably due to the fact that the learning rate is too high, preventing the algorithm to catch the descent direction. We use these results to guide the choice of the learning rate in our experiments and we set $\alpha =$ 5e-5 as it delivers more accurate and robust results.

Table 5.6: Varying Learning Rate

Learning Rate	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
5e−2	1.75209e+01	4.42583e-05	1.75208e+01	1.02673e+00
5e−3	2.18434e-04	3.75588e-07	2.18058e-04	1.10995e-02
5e−4	8.92233e-07	2.53704e-09	8.89696e-07	1.34916e-04
5e−5	1.86471e-07	9.84098e-14	1.86471e-07	4.33972e-06
5e−6	5.46515e-01	4.56204e-01	9.03106e-02	2.66960e+00

Table 5.7: A closer look at learning rate 5e−3

Iteration	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
0	1.73566e+01	2.20809e-03	1.73544e+01	8.68283e-01
10000	1.90421e-05	4.55015e-09	1.90376e-05	2.74259e-04
20000	1.34078e-05	1.03528e-08	1.33975e-05	3.91305e-04
30000	8.88446e-06	1.33335e-08	8.87112e-06	3.97714e-04
40000	5.23286e-03	4.26876e-05	5.19018e-03	1.72262e-02
50000	3.97902e-06	5.77030e-09	3.97325e-06	3.02267e-04
60000	2.67915e-06	2.97896e-09	2.67617e-06	2.12288e-04
70000	3.13569e-06	2.19554e-09	3.13349e-06	1.86585e-04
80000	3.90731e-04	1.10192e-05	3.79712e-04	1.11628e-02
90000	4.43934e-06	9.21067e-09	4.43013e-06	3.63260e-04
100000	2.18434e-04	3.75588e-07	2.18058e-04	1.10995e-02

Table 5.8: A closer look at learning rate 5e−4

Iteration	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
0	1.62138e+01	3.53460e-02	1.61785e+01	4.36439e-01
10000	8.01400e-05	4.04529e-08	8.00995e-05	6.40455e-04
20000	1.73644e-05	9.06907e-10	1.73635e-05	1.24621e-04
30000	1.30918e-04	1.73777e-06	1.29180e-04	5.17546e-03
40000	5.95824e-06	3.28583e-11	5.95821e-06	3.14214e-05
50000	4.17367e-06	3.47686e-11	4.17364e-06	2.87314e-05
60000	3.20441e-06	1.01291e-11	3.20440e-06	1.91423e-05
70000	2.64433e-06	1.39203e-12	2.64433e-06	1.39334e-05
80000	2.27057e-06	5.54222e-11	2.27051e-06	3.78312e-05
90000	1.45376e-05	2.35136e-07	1.43024e-05	1.90796e-03
100000	7.18145e-06	1.45630e-07	7.03582e-06	1.79139e-03

Table 5.9: A closer look at learning rate 5e−5

Iteration	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
0	1.60775e+01	7.49009e-02	1.60026e+01	7.68200e-01
10000	1.53416e-01	6.69879e-02	8.64282e-02	9.96261e-01
20000	2.75060e-05	1.03194e-08	2.74957e-05	6.18447e-04
30000	2.72827e-06	1.09926e-11	2.72826e-06	2.93784e-05
40000	1.40125e-06	1.05222e-11	1.40124e-06	2.53186e-05
50000	7.30668e-07	2.18262e-11	7.30646e-07	2.16615e-05
60000	4.15845e-07	1.74735e-12	4.15843e-07	1.09644e-05
70000	2.78988e-07	5.03405e-13	2.78987e-07	7.05431e-06
80000	2.21948e-07	1.27735e-11	2.21935e-07	1.21054e-05
90000	1.96864e-07	1.19232e-13	1.96864e-07	4.55439e-06
100000	1.86471e-07	9.84098e-14	1.86471e-07	4.33972e-06

5.1.4. Error with respect to Gaussian noise. In this section we consider the impact of noise of the data. Specifically, we add Gaussian white noise to the forcing term. We analyze both the case of training points for u placed only in Γ (Table 5.10) and in $\Omega \cup \Gamma$ (Table 5.11). In this case, reported results correspond to 100000 iterations of the optimization algorithm.

In both cases, as the noise increases, the error of u increases. However, we note that the algorithm performs better (lower errors and losses) when there are no points in the domain. We hypothesize that this is could be due to over-fitting.

Table 5.10: Noise

Noise[%]	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
0	1.15056e-06	4.45340e-10	1.15011e-06	7.65986e-05
5	3.00194e-04	1.95741e-10	3.00194e-04	2.05432e-03
10	4.51297e-03	7.02639e-10	4.51297e-03	4.63073e-03

Table 5.11: Noise with points in the domain for u

Noise[%]	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u
0	6.77939e-07	1.79414e-12	6.77937e-07	8.79139e-06
5	2.96605e-04	4.96719e-08	2.96556e-04	2.16277e-03
10	3.92610e-01	3.67651e-01	2.49585e-02	2.50848e+00

5.2. Results for Inverse Problem. We next consider results for the inverse problem. In this case, we predict both u_{NN} and the parameter vector $\mathbf{d}$.

5.2.1. Single-parameter case. We first consider the case of a polynomial u and a constant kernel. Unless otherwise noted, we use a width of 5, a depth of 5, a learning rate of $\alpha = 5\text{e-}5$, 20 Gauss quadrature points, $N_f = 50$, and $N_u = 100$ (50 points in each side of the collar). All training points are linearly spaced. For $\Omega = (0, 1)$, $\Gamma = (-\delta, 0) \cup (1, 1 + \delta)$ and $\delta = 0.01$, we analyze the behavior of the loss function and of the discretization error as a function of the sampling size and the NN parameters. We consider the following manufactured solution, forcing term, and parametrized kernel:

$$\begin{cases} u(x) = x^2 - x^4 \\ k(x, y) = \frac{d_1}{\delta^3} \\ f = -\frac{6}{5}\delta^2 - 12x^2 + 2. \end{cases} \tag{5.3}$$

Note that f corresponds to the choice $d_1 = \frac{3}{2}$; this allows us to have a reference solution that we can use in the experiments for data generation and accuracy analysis. Initialization of the optimization algorithm is very important, even in single parameter prediction. We consider several different initializations of the parameter d_1 in the following table.

Table 5.12: Different Initializations for d_1

Initial d_1	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u	Predicted d_1
-1	1.0971e-02	1.0968e-02	3.4301e-06	1.0112e+00	-5.29366
0.1	3.1309e-07	6.9428e-11	3.1302e-07	7.6542e-05	1.5001
1	4.0721e-07	2.9948e-10	4.0691e-07	1.63836e-04	1.50032
2	4.1391e-07	4.7247e-10	4.1344e-07	2.0687e-04	1.50040
10	4.5119e-03	4.5112e-03	6.8688e-07	6.4853e-01	8.47400

From Table 5.12, we note that, even when the algorithm convergences to the true value of d_1, e_u is 100 times larger than that of the forward problem for the same manufactured solution. This is an indication of the ill-conditioned nature of inverse problems.

There are two cases of bad initialization: 1) the initial value is too far from the true value; 2) the initial value has a different sign. Consequences of the former case can be seen in Table 5.12, for $d_1 = 10$ for 100000 iterations. Here the parameter reaches the optimal value 7.9 and the corresponding solution is far from the true one, see Figure 5.1.

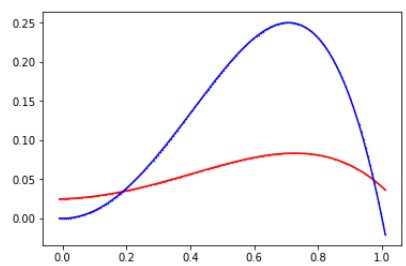


Fig. 5.1: Blue: Manufactured solution. Red: u as computed by nPINN for d_1 initialized to 10.

However, further tests indicate that, for a much higher number of iterations, the predicted parameter approaches 1.5 and the solution converges to the true one. A way to improve the convergence is to multiply $loss_u$ by a scalar $\beta \gg 1$, so that the algorithm predicts a better u. We see an example of this in Table 5.13 where we set $\beta = 10^3$ and run for 200000 iterations. The first row of the table reports results at 100000 iterations, where we see that the loss and error are lower than that of the unscaled loss function. The second row of the table reports results at 200000 iterations, where the desired value of d_1 has been reached.

Table 5.13: Weighted $loss_u$ for parameter prediction

Initial d_1	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u	Predicted d_1
10	3.1011e+00	2.7248e-03	3.7632e-01	5.0646e-01	5.5496
10	1.4344e-04	2.5644e-10	1.4318e-04	1.7074e-04	1.5002

The second issue is the initialization of d_1 to a value of opposite sign of the true value, such as -1; See Figure 5.2 where we report the predicted solution. To deal with this situation, one option is to re-initialize the parameter to a new value any time the predicted parameter

becomes negative. Another option is to restrict values to be in a certain range, as was done in [15] for fractional PDEs.

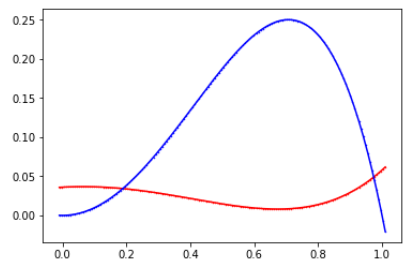


Fig. 5.2: Blue: Manufactured solution. Red: u as computed by nPINN for d_1 initialized to -1.

As a second test, we consider the case where both u and k are a polynomials:

$$\begin{cases} u(x) = x^2 - x^4 \\ k(x,y) = \frac{d_1}{\delta^5}(x-y)^2 \\ f = 2 - \frac{10\delta^2}{7} - 12x^2 \end{cases} \tag{5.4}$$

Note that f corresponds to the choice $d_1 = \frac{5}{2}$; this allows us to have a reference solution that we can use in the experiments for data generation and accuracy analysis. We consider several different initializations of the parameter d_1 and report results at iteration 100000 in Table 5.14. From these results we can infer the same considerations we discussed in case of constant kernel.

Table 5.14: Different Initializations for d_1

Initial d_1	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u	Predicted d_1
-1	1.4135e-02	1.4129e-02	6.2125e-06	1.1478e+00	-5.47035
0.1	7.1532e-08	2.0803e-10	7.1324e-08	1.3872e-04	2.49956
2	1.2176e-07	1.0032e-10	1.2166e-07	6.5511e-05	2.50015
15	4.5053e-03	4.4798e-03	2.5580e-05	6.4639e-01	13.91510

5.2.2. Multi-parameter case. In this section we consider the case with two unknown parameters in the kernel. For the same domain used in the previous section, we consider the following manufactured solution, parameterized kernel and forcing term.

$$\begin{cases} u(x) = x^2 - x^4 \\ k(x,y) = \frac{d_1}{\delta^5}(x-y)^2 + \frac{d_2}{\delta^3} \\ f = \frac{5}{2}\Big(- \frac{4}{7}\delta^2 - \frac{24}{5}x^2 + \frac{4}{5}\Big) + \frac{3}{2}\Big(- \frac{4}{5}\delta^2 - 8x^2 + \frac{4}{3}\Big), \end{cases}$$

where f corresponds to $d_1 = \frac{5}{2}$ and $d_2 = \frac{3}{2}$.

We consider several different initializations. In Table 5.15 we report the (d_1, d_2) at the 100000th iteration.

Table 5.15: Different Initializations for Kernel Parameters

Initial **d**	$loss$	$loss_u$	$loss_{\mathcal{L}}$	e_u	Predicted **d**
(-1,-1)	2.882e-02	2.881e-02	5.699e-06	1.196e+00	(-5.189,-5.189)
(-1,0)	3.039e-02	3.038e-02	9.277e-06	1.227e+00	(-5.332,-4.332)
(-1,1)	2.878e-06	7.485e-07	2.130e-06	6.53194e-03	(0.604, 2.604)
(0,-1)	3.076e-02	3.070e-02	5.885e-05	1.234e+00	(-3.977, -4.983)
(0,0)	3.183e-02	3.182e-02	1.456e-05	1.256e+00	(-4.348, -4.348)
(0,1)	1.125e-06	7.735e-07	3.513e-07	6.611e-03	(1.229, 2.229)
(1,-1)	3.064e-02	3.063e-02	8.242e-06	1.233e+00	(-3.390, -5.390)
(1,0)	8.250e-06	7.988e-07	7.451e-06	6.730e-03	(2.482, 1.480)
(1,1)	3.185e-06	7.798e-07	2.405e-06	6.604e-03	(1.856, 1.855)
(5,-1)	2.006e-06	8.918e-07	1.115e-06	7.041e-03	(5.603, -0.395)
(5,0)	1.879e-05	1.437e-06	1.735e-05	8.574e-03	(4.989, -0.010)
(5,1)	6.508e-06	1.405e-06	5.103e-06	8.347e-03	(4.364, 0.365)
(10,-1)	2.141e-05	1.692e-05	4.490e-06	2.868e-02	(8.788, -2.211)
(10,0)	2.887e-04	2.818e-04	6.815e-06	1.179e-01	(8.389, -1.610)
(10,1)	1.914e-03	1.911e-03	2.839e-06	3.075e-01	(8.474,-0.525)

These results clearly reveal the ill-posed nature of the problem. In fact, while only one initial pair converges to the true **d**, several optimal pairs yield a discretization error of the same order of the one obtained in correspondence of the true **d**. This is an example of the presence of several minima and also of the so-called mimic operators (i.e. operators whose action on a function is almost equivalent).

Remark. To explain the behavior of the algorithm in the multi-parameter case, we consider the following example:

$$\begin{cases} u(x) = x^4 \\ k(x,y) = \frac{d_1}{\delta^5}(x-y)^2 + \frac{d_2}{\delta^3}. \end{cases} \tag{5.5}$$

We compute the action of $\mathcal{L}$ on u:

$$\begin{aligned} \mathcal{L}u &= 2d_1\Big(\frac{2}{7}\delta^2 + \frac{12}{5}x^2\Big) + 2d_2\Big(\frac{2}{7}\delta^2 + \frac{12}{5}x^2\Big) \\ &= \Big(\frac{4}{7}d_1 + \frac{4}{5}d_2\Big)\delta^2 + \Big(\frac{24}{5}d_1 + 8d_2\Big)x^2 \\ &= D + Ex^2 \end{aligned} \tag{5.6}$$

for some constants D, E such that

$$\begin{cases} \delta^2\left(\frac{4}{7}d_1 + \frac{4}{5}d_2\right) = D \\ \frac{24}{5}d_1 + 8d_2 = E. \end{cases} \tag{5.7}$$

Even when the system above is well-posed, the uniqueness of the solution could be compromised by the presence of δ^2, which makes such term very small. Practically, this means that the optimizer will focus on satisfying primarily the equation for E, for which there exists an infinite number of solutions.

6. Conclusions. In this work, we applied the concepts of PINNS to to nonlocal equations. Given measurements of u in the interaction domain and of f within the domain, we demonstrated that nonlocal physics-informed neural networks (nPINNS) can accurately solve forward problems.

We also analyzed the behavior of the error with respect to the number of training points, the network parameters, the learning rate and the level of noise. The error decreasing as the number of training points increases at first, but eventually saturates. Similarly, increasing the width or depth of the network beyond a value of 4 did not result in more accurate results from the network. We also found that a learning rate between 5e-4 and 5e-5 produced the best results. We saw low, but not negligible, impact of Gaussian white noise added to the forcing term. While nPINNs was able to accurately predict kernels that depend on a single parameter, results were not as satisfactory for the multi-parameter case, and further investigation is required.

REFERENCES

[1] A. A. Buades, B. Coll, and J. Morel, *Image denoising methods. a new nonlocal principle*, SIAM Review, 52 (2010), pp. 113–147.

[2] B. Alali and R. Lipton, *Multiscale dynamics of heterogeneous media in the peridynamic formulation*, Journal of Elasticity, 106 (2012), pp. 71–103.

[3] E. Askari, *Peridynamics for multiscale materials modeling*, Journal of Physics: Conference Series, IOP Publishing, 125 (2008), pp. 649–654.

[4] D. Benson, S. Wheatcraft, and M. Meerschaert, *Application of a fractional advection-dispersion equation*, Water Resources Research, 36 (2000), pp. 1403–1412.

[5] N. Burch, M. D'Elia, and R. Lehoucq, *The exit-time problem for a markov jump process*, The European Physical Journal Special Topics, 223 (2014), pp. 3257–3271.

[6] A. Delgoshaie, D. Meyer, P. Jenny, and H. Tchelepi, *Non-local formulation for multiscale flow in porous media*, Journal of Hydrology, 531 (2015), pp. 649–654.

[7] M. D'Elia, Q. Du, M. Gunzburger, and R. Lehoucq, *Nonlocal convection-diffusion problems on bounded domains and finite-range jump processes*, Computational Methods in Applied Mathematics, 29 (2017), pp. 71–103.

[8] Q. Du, M. Gunzburger, R. Lehoucq, and K. Zhou, *Analysis and approximation of nonlocal diffusion problems with volume constraints*, SIAM Review, 54 (2012), pp. 667–696.

[9] P. Fife, *Some nonclassical trends in parabolic and parabolic-like evolutions*, Springer-Verlag, New York, 2003, ch. Vehicular Ad Hoc Networks, pp. 153–191.

[10] M. D. Foss and P. Radu, *Bridging local and nonlocal models: Convergence and regularity*, Handbook of Nonlocal Continuum Mechanics for Materials and Structures, (2018), pp. 1–21.

[11] G. Gilboa and S. Osher, *Nonlocal linear image regularization and supervised segmentation*, Multiscale Model. Simul., 6 (2007), pp. 595–630.

[12] M. T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-Hill, Boston, MA, 2nd ed., 2002.

[13] D. Littlewood, *Simulation of dynamic fracture using peridynamics, finite element modeling, and contact*, in Proceedings of the ASME 2010 International Mechanical Engineering Congress and Exposition, Vancouver, British Columbia, Canada, 2010.

[14] M. Meerschaert and A. Sikorskii, *Stochastic models for fractional calculus*, Studies in mathematics, Gruyter, 2012.

[15] G. Pang, L. Lu, and G. E. Karniadakis, *fpinns: Fractional physics-informed neural networks*, arXiv preprint arXiv:1811.08967, (2018).

[16] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*, Journal of Computational Physics, 378 (2019), pp. 686–707.

[17] A. Schekochihin, S. Cowley, and T. Yousef, *Mhd turbulence: Nonlocal, anisotropic, nonuniversal?*, in In IUTAM Symposium on computational physics and new perspectives in turbulence, Springer, Dordrecht, 2008, pp. 347–354.

[18] R. Schumer, D. Benson, M. Meerschaert, and B. Baeumer, *Multiscaling fractional advection-dispersion equations and their solutions*, Water Resources Research, 39 (2003), pp. 1022–1032.

[19] R. Schumer, D. Benson, M. Meerschaert, and S. Wheatcraft, *Eulerian derivation of the fractional advection-dispersion equation*, Journal of Contaminant Hydrology, 48 (2001), pp. 69–88.

[20] S. Silling, *Reformulation of elasticity theory for discontinuities and long-range forces*, Journal of the Mechanics and Physics of Solids, 48 (2000), pp. 175–209.

PROPER ORTHOGONAL DECOMPOSITION FOR TIME DEPENDENT PROBLEMS - NEURAL NETWORK APPROACH

PETER SENTZ*, ERIC C. CYR†, KRISTIAN BECKWITH‡, AND LUKE OLSON§

Abstract. Model order reduction is an important topic in the field of partial differential equations, particularly those which must be solved for a wide variety of physical or geometrical parameters. Many techniques rely on linearity and parametric separability of the relevant equations in order to produce a computationally efficient method; however, efficiency may greatly suffer when the problem does not satisfy these properties. Here, we extend a neural network based approach for parametrized elliptic equations to the time-dependent case. The technique is applied to a one-dimensional advection-diffusion equation, where it is compared to a classical projection based approach.

1. Introduction. In many applications in the physical sciences and engineering, phenomena are modeled by parametrized partial differential equations (PDEs). For example, solutions are sought for the Navier-Stokes equations for a range of Reynolds numbers, or to the heat equation for different values of the conductivity. The shape of the domain or boundary conditions may also be parametrized.

In the time-independent case, equations of the form

$$F\left(u(\boldsymbol{\mu}); \boldsymbol{\mu}\right) = 0, \tag{1.1}$$

are considered. Here, $\boldsymbol{\mu}$ denotes a vector of real-valued parameters, $\boldsymbol{\mu} \in D \subset \mathbb{R}^P$. The components of $\boldsymbol{\mu}$ are parameters that can be physical (e.g. Reynolds number, conductivity) or determine the geometry of the domain or boundary/forcing terms. $u(\boldsymbol{\mu})$ denotes the solution to the PDE, and is dependent on the parameters. F encodes the PDE under consideration; it too depends on the parameters explicitly.

Time-dependent problems can be similarly parametrized in the general form

$$u_t(\boldsymbol{\mu}) + F\left(u(\boldsymbol{\mu}); \boldsymbol{\mu}\right) = 0. \tag{1.2}$$

Typically, one seeks a numerical solution to such a PDE using e.g., finite element or finite difference methods. Increasing computational power and sophisticated algorithms have led to the ability to solve for very accurate numerical approximations. However, these "high fidelity" solutions can still take many hours or days to compute. This can be an unacceptable cost if solutions are required for many instances of the parameter vector $\boldsymbol{\mu}$. Two contexts in particular motivate the need for solutions with less computational complexity, the *many-query* and *real-time* contexts [19]. For an example of the many query context, one may want to solve a PDE-constrained optimization problem to find the optimal forcing function or shape of the domain. This is an optimization problem for the parameter vector $\boldsymbol{\mu}$, and a number of iterative solution methods can be employed. These methods require the solution of the PDE for a sequence of different parameter configurations. Over many iterations, the computational demands of the high-fidelity solutions may be prohibitive. Real-time contexts arise, for example, in control problems governed by PDEs [11].

A class of methods, known as *reduced basis methods*, has been developed over the past few decades to reduce the model order complexity in cases where the many-query or real-time contexts are relevant. In section 2, we will review this *projection-based* approach, and the

*University of Illinois at Urbana-Champaign, sentz2@illinois.edu
†Sandia National Laboratories, eccyr@sandia.gov
‡Sandia National Laboratories, kbeckwi@sandia.gov
§University of Illinois at Urbana-Champaign, lukeo@illinois.edu

assumptions which are required in order to develop a computationally efficient alternative to the original model. We briefly discuss what happens when these assumptions do not hold and some corresponding approaches to develop reduced-order models. In section 3, we use a parametrized linear advection-diffusion equation in one space variable to demonstrate a flexible reduced basis method using neural networks. Section 4 presents some conclusions and ideas for future work.

2. Projection-Based Model Order Reduction. There are a number of ways to reduce the computational complexity of the problem and compute lower-fidelity approximations to the true solution while controlling the amount of error. One approach that has attracted considerable attention is projection-based model order reduction [19]. Methods of this form have been applied to a wide variety of applications including electromagnetics, fluid dynamics, thermodynamics, and linear elasticity [6, 8, 14, 18]. We illustrate it here for a finite element discretization of (1.1), where F is linear in $u(\boldsymbol{\mu})$. This leads to the following variational problem: find $u_h(\boldsymbol{\mu}) \in V_h$ such that

$$a\left(u_h(\boldsymbol{\mu}), v_h; \boldsymbol{\mu}\right) = L\left(v_h; \boldsymbol{\mu}\right) \qquad \forall v_h \in W_h. \tag{2.1}$$

Here, V_h and W_h are two finite dimensional function spaces with equal dimension, $a : V_h \times W_h \to \mathbb{R}$ is a continuous and coercive bilinear form, and $L : W_h \to \mathbb{R}$ is a continuous linear mapping. a and L are obtained from (1.1) through multiplication by a test function $v_h \in W_h$ followed by integration by parts. If $V_h = W_h$, this is known as a Galerkin projection scheme. Otherwise, it is known as a Petrov-Galerkin method. For simplicity, we assume $V_h = W_h$.

Let $N_h \coloneqq \dim(V_h)$, and let $\{\phi_i\}_{i=1}^{N_h}$ be a basis for V_h. The numerical approximation to $u(\boldsymbol{\mu})$ in V_h can be expressed as the linear combination $u_h(\boldsymbol{\mu}) = \sum_{i=1}^{N_h} u_i(\boldsymbol{\mu})\phi_i$. The scalars $u_i(\boldsymbol{\mu})$ are the *degrees of freedom* for the approximation $u_h(\boldsymbol{\mu})$ with respect to the basis $\{\phi_i\}_{i=1}^{N_h}$. In the case of a standard Lagrange finite element method, the degrees of freedom correspond to nodal values of the solution, but more general degrees of freedom such as derivatives or moments are possible. The weak form (2.1) leads to a linear system of the form

$$\boldsymbol{A}_h(\boldsymbol{\mu})\boldsymbol{u}_h(\boldsymbol{\mu}) = \boldsymbol{b}_h(\boldsymbol{\mu}), \tag{2.2}$$

where $\left(\boldsymbol{A}_h(\boldsymbol{\mu})\right)_{ij} = a(\phi_j, \phi_i; \boldsymbol{\mu})$, $\left(\boldsymbol{b}(\boldsymbol{\mu})\right)_i = L(\phi_i; \boldsymbol{\mu})$, and $\boldsymbol{u}_h(\boldsymbol{\mu})$ is the vector of degrees of freedom of the finite element approximation $u_h(\boldsymbol{\mu})$. If N_h is very large, this leads to a prohibitive computational cost to solve for the full-order (high fidelity) solution for each parameter instance in the many-query or real-time context. The key to reduce the complexity is to introduce a subspace $V_M \subset V_h$ with $M = \dim(V_M) \ll \dim(V_h) = N_h$, and project the residual of (2.2) onto this subspace.

For concreteness, let $\{\xi_i\}_{i=1}^{M}$ be a basis for V_M and let $\boldsymbol{\xi}_i$ be the vector of degrees of freedom of ξ_i with respect to the basis of V_h. We also assume the vectors $\boldsymbol{\xi}_i$ are orthonormal in the Euclidean inner product on $\mathbb{R}^{N_h}$. Define the matrix $\boldsymbol{V}_M$ by

$$\boldsymbol{V}_M = \left(\boldsymbol{\xi}_1 \mid \boldsymbol{\xi}_2 \mid \ \dots \ \mid \boldsymbol{\xi}_M\right). \tag{2.3}$$

We look for a reduced order solution of the form $u_M(\boldsymbol{\mu}) = \sum_{j=1}^{M} c_j(\boldsymbol{\mu})\xi_j$, and solve for the vector of coefficients $\boldsymbol{c}(\boldsymbol{\mu})$ through the equation:

$$\boldsymbol{A}_M(\boldsymbol{\mu})\boldsymbol{c}(\boldsymbol{\mu}) = \boldsymbol{b}_M(\boldsymbol{\mu}), \tag{2.4}$$

where $\boldsymbol{A}_M(\boldsymbol{\mu}) = \boldsymbol{V}_M^T\boldsymbol{A}_h(\boldsymbol{\mu})\boldsymbol{V}_M$ and $\boldsymbol{b}_M(\boldsymbol{\mu}) = \boldsymbol{V}_M^T\boldsymbol{b}_h(\boldsymbol{\mu})$. Equation (2.4) is an $M \times M$ linear system, which is much smaller than the original $N_h \times N_h$ linear system. Following

this, the vector of degrees of freedom of $u_M(\boldsymbol{\mu})$ is computed through $\boldsymbol{u}_M(\boldsymbol{\mu}) = \boldsymbol{V}_M \boldsymbol{c}(\boldsymbol{\mu})$. This is equivalent to a Galerkin projection directly onto the reduced subspace.

2.1. Separation into Online and Offline Stages. In many applications (2.2) must be computed repeatedly for many different parameter instances. If the dimension of the problem N_h is large, this can be prohibitively expensive as the number of parameter instances grows. However, if there exists a method to compute a very cheap approximation to this full-order system, a reduced-order solution can be computed very rapidly for each new parameter instance. In particular, an effective algorithm should be independent of the problem size [9]. Of course, such an inexpensive solution procedure must be developed.

From the point of view of reduced order modeling, as long as the marginal cost or asymptotic average cost of the solution for each new parameter instance is small, a more expensive set-up phase that constructs the necessary components for the reduced-order computation is acceptable [20]. That is, if the number of parameters for which a solution is required is very large, the time taken to compute a small number of full-order solutions will not substantially effect the marginal cost. Reduced-order modeling thus divides the algorithm into two stages, an initial set-up phase (*offline* phase), and a repeated solution phase (*online* phase).

In the case of a linear elliptic PDE, we can naively separate the problem into two stages as follows. In the offline stage, we build the subspace V_M and assemble the corresponding matrix $\boldsymbol{V}_M$. This can be done by sampling the set of possible parameters $D \subset \mathbb{R}^P$ and using a proper orthogonal decomposition (POD) [1] or a greedy reduced basis (greedy-RB) [7]. Then in the online stage, for every new set of parameters $\boldsymbol{\mu}$, we assemble $\boldsymbol{A}_M(\boldsymbol{\mu})$ and $\boldsymbol{b}_M(\boldsymbol{\mu})$ and solve the resulting reduced linear system.

However, this will not result in a computationally cheap online stage, as it still depends on the dimension of the original problem. Assembling $\boldsymbol{A}_M(\boldsymbol{\mu}) = \boldsymbol{V}_M^T \boldsymbol{A}_h(\boldsymbol{\mu}) \boldsymbol{V}_M$ requires assembling $\boldsymbol{A}_h(\boldsymbol{\mu})$, and so is not independent of N_h. Thus we cannot expect dramatic decrease in computational cost in the online stage.

2.2. Improving the Online Stage - Separable Case. In the literature, an assumption is often made that the problem is *parametrically separable* (or affinely parametrizable) [6, 8, 10, 21]. In the linear case, this makes the assumption that

$$a(u, v; \boldsymbol{\mu}) = \sum_{k=1}^{Q_a} \Theta_a^k(\boldsymbol{\mu}) a^k(u, v), \qquad L(v; \boldsymbol{\mu}) = \sum_{k=1}^{Q_L} \Theta_L^k(\boldsymbol{\mu}) L^k(v), \tag{2.5}$$

where a^k and L^k are parameter-independent bilinear and linear forms, respectively, and the Θ_a^k, Θ_k^L are smooth functions of the parameter $\boldsymbol{\mu}$.

With this assumption we then have

$$\boldsymbol{A}_h(\boldsymbol{\mu}) = \sum_{k=1}^{Q_a} \Theta_a^k(\boldsymbol{\mu}) \boldsymbol{A}_h^k, \qquad \boldsymbol{b}_h(\boldsymbol{\mu}) = \sum_{k=1}^{Q_L} \Theta_L^k(\boldsymbol{\mu}) \boldsymbol{b}_h^k, \tag{2.6}$$

where $\boldsymbol{A}_h^k$ are $N_h \times N_h$ parameter-independent matrices and the $\boldsymbol{b}_h^k$ are parameter-independent vectors in $\mathbb{R}^{N_h}$. Thus, assembly of the $M \times M$ system can be done primarily offline: we simply assemble $\boldsymbol{A}_M^k := \boldsymbol{V}_M^T \boldsymbol{A}_h^k \boldsymbol{V}_M$, $\boldsymbol{b}_M^k := \boldsymbol{V}_M^T \boldsymbol{b}_h^k$. Then in the online stage, for any new parameter value $\boldsymbol{\mu}$, we can construct the linear system in (2.4) by

$$\boldsymbol{A}_M(\boldsymbol{\mu}) = \sum_{k=1}^{Q_a} \Theta_a^k(\boldsymbol{\mu}) \boldsymbol{A}_M^k, \qquad \boldsymbol{b}_M(\boldsymbol{\mu}) = \sum_{k=1}^{Q_L} \Theta_L^k(\boldsymbol{\mu}) \boldsymbol{b}_M^k \tag{2.7}$$

2.3. Improving the Online Stage - Non-Separable Case. While many linear PDEs do enjoy the desirable property of parametric separability, not all do, and it is very doubtful to hold for nonlinear PDEs. If the assumption (2.5) does not hold, one may apply techniques such as the discrete empirical interpolation method (DEIM) [5] to approximate the bilinear and linear forms by separable approximations. This results in further approximation error, is problem-dependent and *intrusive* in nature, and requires more sampling in the parameter space to construct the basis [9].

With this in mind, [9] proposes a maching learning approach through regression via neural networks for steady-state parametrized PDEs. Their approach is as follows. First, they sample the parameter domain D and compute a set of "snapshots" of full-order solutions, and perform a proper orthogonal decomposition to obtain a basis of functions $\{\xi_i\}_{i=1}^M$ with orthonormal vectors of degrees of freedom. They define an inner product on the finite element subspace V_h by

$$\langle u_h, v_h \rangle_h \coloneqq \langle \boldsymbol{u}_h, \boldsymbol{v}_h \rangle_2. \tag{2.8}$$

That is, the Euclidean inner product of each function's vector of degrees of freedom. This results in a matrix $\boldsymbol{V}_M$ of the form (2.3) with orthonormal columns.

Rather than seek the Galerkin projection $u_M(\boldsymbol{\mu})$ onto this subspace for any given parameter $\boldsymbol{\mu}$ and corresponding full-order solution $u_h(\boldsymbol{\mu})$, they seek its projection in the h-inner product. That is,

$$P_h u_h(\boldsymbol{\mu}) = \sum_{i=1}^{M} \langle u_h(\boldsymbol{\mu}), \xi_i \rangle_h \xi_i = \boldsymbol{V}_M \boldsymbol{V}_M^T \boldsymbol{u}_h(\boldsymbol{\mu}). \tag{2.9}$$

To compute the projection directly for a given $\boldsymbol{\mu}$ requires the full-order solution $u_h(\boldsymbol{\mu})$, so that this projection is unacceptable for an online stage. However, during the offline stage, they train a neural network to learn the mapping:

$$\boldsymbol{\mu} \mapsto \boldsymbol{c}(\boldsymbol{\mu}) \coloneqq \boldsymbol{V}_M^T \boldsymbol{u}_h(\boldsymbol{\mu}). \tag{2.10}$$

The trained neural network is then a mapping $G : \mathbb{R}^P \to \mathbb{R}^M$ such that

$$G(\boldsymbol{\mu}) \approx \boldsymbol{V}_M^T \boldsymbol{u}_h(\boldsymbol{\mu}). \tag{2.11}$$

In the online stage, for a new parameter vector $\boldsymbol{\mu}$, one simply evaluates the map $\boldsymbol{c}_{\mathrm{NN}}(\boldsymbol{\mu}) = G(\boldsymbol{\mu})$, and then represents the approximate solution through its vector of degrees of freedom by

$$\boldsymbol{u}_{\mathrm{NN}}(\boldsymbol{\mu}) = \boldsymbol{V}_M \boldsymbol{c}_{\mathrm{NN}}(\boldsymbol{\mu}) \tag{2.12}$$

The evaluation of G is independent of the problem size N_h, and so leads to an efficient online stage. The offline stage has considerable overhead, but this can be amortized over the online stage if the solution is needed for many values of $\boldsymbol{\mu}$.

They restrict this method to steady-state parametrized PDEs, in particular, nonlinear Poisson and steady Navier-Stokes equations. The goal of our work is to extend this method to time-dependent problems, using a simple parametrized linear advection-diffusion equation as a test problem.

We note that the method developed in this paper has similarities with other work in the field of reduced order models in *dynamics learning*. In [15], a Long Short-Term Memory (LSTM) network is used to learn low-order dynamics for computational fluid dynamics (CFD) data sets. In [4], missing CFD data is reconstructed by learning low-order dynamics that result from compression through auto-encoders. In [3], the Sparse Identification of Nonlinear Dynamics (SINDy) method, which uses symbolic regression, is used to learn the low-order dynamics of POD modes.

3. Model Problem: One-Dimensional Advection-Diffusion. We first outline the neural network approach to temporal problems through a simple 1D advection-diffusion equation with a single parameter. This problem is linear, and so can be handled by a standard Galerkin projection in each time step. However, we start with this simple problem to demonstrate the extension of the neural network approach to time-dependent problems.

3.1. Problem Definition. Fix $\mu \in [1, 30]$. We seek a function $w(x, t; \mu)$ that satisfies

$$\begin{aligned} w_t + w_x - w_{xx} &= 0 && x \in (0,1), \\ w(x,0) &= g(x) && x \in [0,1], \\ w(0,t) &= \sin(\mu t) && t > 0, \\ w_x(1,t) &= 0 && t > 0, \end{aligned} \tag{3.1}$$

with initial condition $g(x) = \sin(\pi x) + \frac{\pi}{2}x^2$. In order to introduce homogeneous boundary conditions, define the variable $u(x, t; \mu) \coloneqq w(x, t; \mu) - \sin(\mu t)$, which satisfies

$$\begin{aligned} u_t + u_x - u_{xx} &= -\mu \cos(\mu t) && x \in (0,1), \\ u(x,0) &= g(x) && x \in (0,1), \\ u(0,t) &= 0 && t > 0, \\ u_x(1,t) &= 0 && t > 0. \end{aligned} \tag{3.2}$$

Instead of seeking a reduced-order approximation to $w(x, t; \mu)$ directly, our goal is to seek an approximation $u_R(x, t; \mu) \approx u(x, t; \mu)$, and take $w_R(x, t; \mu) \coloneqq u_R(x, t; \mu) + \sin(\mu t)$.

3.2. Spatial Discretization. To discretize this equation in space, subdivide the interval $[0, 1]$ into 100 equally spaced cells, and let V_h be the corresponding finite element space of continuous piecewise linear functions that vanish at $x = 0$. Integrate (3.2) against a test function $v_h(x) \in V_h$ and perform integration by parts to obtain the weak form

$$\int_0^1 \left(u_t v_h + u_x v_h + u_x (v_h)_x\right) \mathrm{d}x = -\mu \cos(\mu t) \int_0^1 v_h \, \mathrm{d}x, \qquad \forall v_h \in V_h. \tag{3.3}$$

Define the finite element approximation to $u(x, t; \mu)$ by

$$u_h(x, t; \mu) = \sum_{j=1}^{N_h} u_j(t; \mu) \phi_j(x), \tag{3.4}$$

where $\{\phi_j\}_{j=1}^{N_h}$ is a basis for V_h, $N_h \coloneqq \dim(V_h)$, and $u_j(t; \mu) \in \mathbb{R}$. Defining the matrices

$$\boldsymbol{M}_{ij} = \int_0^1 \phi_j \phi_i \, \mathrm{d}x, \qquad \boldsymbol{C}_{ij} = \int_0^1 \phi_j' \phi_i \, \mathrm{d}x, \qquad \boldsymbol{D}_{ij} = \int_0^1 \phi_j' \phi_i' \, \mathrm{d}x, \tag{3.5}$$

and vectors

$$\boldsymbol{u}_i(t; \mu) = u_i(t; \mu), \qquad \boldsymbol{b}_i = \int_0^1 \phi_i \, \mathrm{d}x, \tag{3.6}$$

(3.3) is expressed as the system of ordinary differential equations (ODEs) given by

$$\boldsymbol{M}\boldsymbol{u}'(t; \mu) + \boldsymbol{C}\boldsymbol{u}(t; \mu) + \boldsymbol{D}\boldsymbol{u}(t; \mu) = -\mu \cos(\mu t) \boldsymbol{b}. \tag{3.7}$$

Finally, define the matrix $\boldsymbol{X} \coloneqq -\boldsymbol{M}^{-1} (\boldsymbol{C} + \boldsymbol{D})$ and the vector $\boldsymbol{h} \coloneqq -\boldsymbol{M}^{-1} \boldsymbol{b}$, to obtain

$$\boldsymbol{u}'(t; \mu) = \boldsymbol{X}\boldsymbol{u}(t; \mu) + \mu \cos(\mu t) \boldsymbol{h}. \tag{3.8}$$

3.3. Time Discretization. We use the second-order trapezoidal rule to discretize the ODE (3.8). Introduce a sequence $0 = t_0 < t_1 < t_2 < \cdots < t_N = T$ and fixed step-size $t_n - t_{n-1} = \Delta t$. The discrete approximation to $u(t_n;\mu) \approx u^{(n)}(\mu)$ is obtained through the time-stepping scheme

$$\begin{aligned} &\boldsymbol{u}^{(0)}(\mu) = \boldsymbol{g}, \\ &\left(I - \frac{\Delta t}{2}\boldsymbol{X}\right)\boldsymbol{u}^{(n)}(\mu) = \left(I + \frac{\Delta t}{2}\boldsymbol{X}\right)\boldsymbol{u}^{(n-1)} + \frac{\mu\Delta t}{2}\left(\cos(\mu t_n) + \cos(\mu t_{n-1})\right)\boldsymbol{h}. \end{aligned} \tag{3.9}$$

Here $\boldsymbol{g}$ is the vector of degrees of freedom from the finite element interpolant of the initial condition $g(x)$, which is independent of μ. Define $f^{(n)}(\mu) := \mu\Delta t\cos(\mu t_n)$. Then the time-stepping relationship can be written compactly as

$$\begin{aligned} &\boldsymbol{u}^{(1)}(\mu) = F_1\left(f^{(1)}(\mu), f^{(0)}(\mu)\right), \\ &\boldsymbol{u}^{(n)}(\mu) = F_2\left(\boldsymbol{u}^{(n-1)}, f^{(n)}(\mu), f^{(n-1)}(\mu)\right), \qquad n \geq 2. \end{aligned} \tag{3.10}$$

Since the initial condition $\boldsymbol{u}^{(0)}$ is independent of the parameter μ, and the time-steps are uniform, the coefficients at the first time-step only depend on the initial forcing and the forcing at the first time-step. Beyond the first-time step however, the coefficients also depend on the coefficients at the previous time-step.

3.4. Proper Orthogonal Decomposition in the L^2 Inner Product. We develop a reduced order model to $u_h(x,t;\mu)$ by constructing a subspace $V_M \subset V_h$ with $M := \dim(V_M) \ll N_h = \dim(V_h)$, and writing

$$u_M^{(n)}(\mu) = \sum_{j=1}^{M} c_j^{(n)}(\mu)\xi_j, \tag{3.11}$$

where $\{\xi_j\}$ is a basis for V_M. We construct V_M through a proper orthogonal decomposition of a collection of full-order solutions or snapshots. As above, let $\boldsymbol{u}^{(n)}(\mu)$ be the vector of degrees of freedom of the full-order solution given by (3.9). Let $\Xi_{\text{POD}} := \{\mu_1, \mu_2, \ldots, \mu_Q\}$ be a discrete subset of the set of possible parameters $D = [1, 30]$, choose $\{n_1, n_2, \ldots, n_r\} \subset \{0, \ldots, N\}$, and form the "snapshot" matrix

$$\boldsymbol{S} := \left[\boldsymbol{u}^{(n_1)}(\mu_1), \boldsymbol{u}^{(n_2)}(\mu_1), \ldots, \boldsymbol{u}^{(n_r)}(\mu_1), \boldsymbol{u}^{(n_1)}(\mu_2), \ldots, \boldsymbol{u}^{(n_r)}(\mu_Q)\right]. \tag{3.12}$$

We then perform the following algorithm:

- Assemble $\boldsymbol{M}_{ij} := \langle \phi_j, \phi_i \rangle_{L^2(0,1)}$,
- Form $\boldsymbol{K} = \boldsymbol{M}^{1/2}\boldsymbol{S}\boldsymbol{S}^T\boldsymbol{M}^{1/2}$,
- Solve $\boldsymbol{K}\boldsymbol{v}_i = \lambda_i\boldsymbol{v}_i, \quad i = 1, 2, \ldots M$,
- Set $\boldsymbol{\xi}_i = \boldsymbol{M}^{-1/2}\boldsymbol{v}_i, \quad i = 1, 2, \ldots M$.

Here, $\boldsymbol{M}^{1/2}$ refers to the matrix square root; since $\boldsymbol{X}$ is symmetric positive definite, it admits an eigendecomposition $\boldsymbol{M} = \boldsymbol{Q}\boldsymbol{D}\boldsymbol{Q}^T$, so $\boldsymbol{M}^{1/2} = \boldsymbol{Q}\boldsymbol{D}^{1/2}\boldsymbol{Q}^T$. Letting $\boldsymbol{V}_M := [\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_M]$, we obtain a matrix where the columns are the degrees of freedom of a collection of L^2-orthonormal finite element functions, i.e. $\boldsymbol{V}_M^T\boldsymbol{M}\boldsymbol{V}_M = \boldsymbol{I}$. This is in contrast the proper orthogonal decomposition performed in [9], which resulted in functions that are orthonormal is the discrete $\langle\cdot,\cdot\rangle_h$ inner product. If we choose basis functions that are orthonormal in the $\langle\cdot,\cdot\rangle_{L^2}$ inner product, we can easily compute the L^2 norm of the difference between two

functions $v, w \in V_M$, through their expansion in the basis $\{\xi_j\}$. That is, if $v = \sum_{j=1}^{M} v_j \xi_j$ and $w = \sum_{j=1}^{M} w_j \xi_j$, then

$$\|v - w\|_{L^2}^2 = \sum_{j=1}^{M} (v_j - w_j)^2 = \|\boldsymbol{v} - \boldsymbol{w}\|_2^2, \tag{3.13}$$

where $\boldsymbol{v}$ and $\boldsymbol{w}$ are the vectors of the expansion coefficients v_j and w_j, and $\|\cdot\|_2$ is the standard Euclidean norm on $\mathbb{R}^M$.

In the training phase, when a neural network is trained to compute the expansion coefficients, using the mean squared error of the POD coefficients as the loss function is thus equivalent to using the squared L^2-norm of the underlying function space as the loss. Using the proper orthogonal decomposition from [9], a mean squared loss function instead corresponds to the discrete $\|\cdot\|_h$ norm.

To select M, the dimension of the basis, one can terminate whenever the relative energy

$$\frac{\sum_{i=1}^{M} \lambda_i}{\sum_{i=1}^{R} \lambda_i}, \tag{3.14}$$

is sufficiently close to 1, where $R = \operatorname{rank}(\boldsymbol{S})$.

3.5. Approximating the Solution in the POD Basis. The coefficients in (3.11) can be computed through the L^2 projection of the full-order approximation $u_h^{(n)}(\mu)$, i.e., $c_j^{(n)}(\mu) = \langle u_h^{(n)}(\mu), \phi_j \rangle_{L^2}$. This can also be expressed as

$$\boldsymbol{c}^{(n)}(\mu) = \boldsymbol{V}_M^T \boldsymbol{M} \boldsymbol{u}^{(n)}(\mu). \tag{3.15}$$

Of course, this is not practical for the online stage of the reduced order model, as the full-order solution must be computed up to time t_n and then projected onto the basis. A classical approach is to project the initial conditions onto the basis, and then evolve the coefficients forward in time by projecting the POD onto the subspace V_M. This will be referred to as POD-ROM. Similarly to the full-order scheme, we define matrices

$$\widetilde{\boldsymbol{M}}_{ij} = \int_0^1 \xi_i \xi_j \,\mathrm{d}x = \delta_{ij}, \quad \widetilde{\boldsymbol{C}}_{ij} = \int_0^1 \xi_j' \xi_i \,\mathrm{d}x, \quad \widetilde{\boldsymbol{D}}_{ij} = \int_0^1 \xi_j' \xi_i' \,\mathrm{d}x, \tag{3.16}$$

along with the vector

$$\widetilde{\boldsymbol{b}}_i = \int_0^1 \xi_i \,\mathrm{d}x. \tag{3.17}$$

Projecting the full-order time-stepping scheme (3.9), a vector of approximate reduced coeffients $\boldsymbol{c}_R^{(n)}(\mu)$ is computed through the time-stepping scheme:

$$\begin{aligned} \left(I + \frac{\Delta t}{2}\left(\widetilde{\boldsymbol{C}} + \widetilde{\boldsymbol{D}}\right)\right) \boldsymbol{c}_R^{(n)}(\mu) = \left(I - \frac{\Delta t}{2}\left(\widetilde{\boldsymbol{C}} + \widetilde{\boldsymbol{D}}\right)\right) \boldsymbol{c}_R^{(n-1)} \\ - \frac{\mu \Delta t}{2}\left(\cos(\mu t_n) + \cos(\mu t_{n-1})\right) \widetilde{\boldsymbol{b}}. \end{aligned} \tag{3.18}$$

It must be emphasized that for $n \geq 1$,

$$\boldsymbol{c}_R^{(n)}(\mu) \neq \boldsymbol{V}_M^T \boldsymbol{M} \boldsymbol{u}^{(n)}(\mu) = \boldsymbol{c}^{(n)}(\mu). \tag{3.19}$$

That is, only the reduced coefficients at time 0 are guaranteed to be the projection coefficients of the full order solution. We can represent these two approximations of $u^{(n)}(\mu)$ in the basis of V_M schematically. The POD-ROM approach can be written as in figure (3.1(a)).

If we require the true projection coefficients $\boldsymbol{c}^{(n)}(\mu) = \boldsymbol{V}_M^T \boldsymbol{M} \boldsymbol{u}^{(n)}(\mu)$ for every n, then we can evolve the full-order solution forward in time and project onto V_M every time step. This is represented in figure (3.1(b)).

As mentioned before, this is not a practical method for constructing a reduced order model, as it is just as expensive as the full-order model.

Instead, it is possible to evolve the solutions for many different parameter values μ and store the true coefficients. We propose training a neural network to learn the map between $\boldsymbol{c}^{(n-1)}(\mu)$ and $\boldsymbol{c}^{(n)}(\mu)$, and bypass the computation of the full-order solution at the next time step. This is shown in figure (3.1(c))

Thus, we train two neural networks N_1 and N_2, that compute approximate projection coefficients in a manner analogous to the full order solution in (3.10).

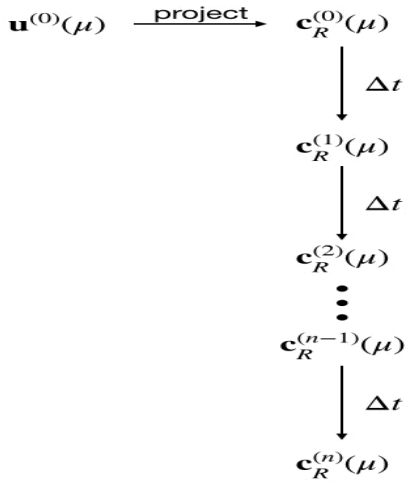
$$\begin{aligned} \boldsymbol{c}_{\mathrm{NN}}^{(1)}(\mu) &= N_1\left(f^{(1)}(\mu), f^{(2)}(\mu)\right) \\ \boldsymbol{c}_{\mathrm{NN}}^{(n)}(\mu) &= N_2\left(\boldsymbol{c}^{(n-1)}(\mu), f^{(n)}(\mu), f^{(n-1)}(\mu)\right), \qquad n \geq 2. \end{aligned} \tag{3.20}$$

3.6. Numerical Results. As mentioned previously, we divide the interval $[0, 1]$ into 100 cells, and use a finite element space of linear polynomials corresponding to this partition for the space discretization. To construct the POD basis, we fix $\Delta t = \frac{\pi}{150}$, and compute the full-order solutions for parameter values $\mu \in \{1, 2, \ldots, 29, 30\}$ up to time $T = 2\pi$, or 300 timesteps. The parameter-independent initial condition is retained as the first POD-snapshot, then for each sampled parameter μ we save the corresponding solution at each time step as an additional POD-snapshot. This results in the matrix $\boldsymbol{S}$ from (3.12) of size 100×6001. Only 3 basis functions are needed to capture 99.99% of the energy in (3.14).

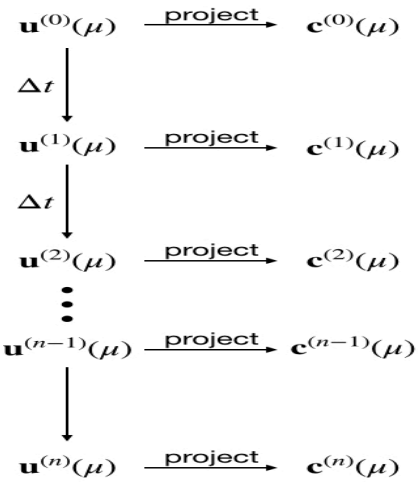
We next generate 30 samples $\mu \in [1, 30]$ to serve as our training data, and an additional 15 randomly generated samples as the validation set to avoid overfitting.

We approximate the map $\boldsymbol{c}_{\mathrm{NN}}^{(1)} = N_1\left(f^{(1)}, f^{(0)}\right)$, which is a mapping $\mathbb{R}^2 \to \mathbb{R}^M$. We train M neural networks to learn each individual generalized coordinate in $\mathbb{R}^M$. The training process minimizes the mean squared loss in (3.13) over the training set using Pytorch's Limited Memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) algorithm [16, 17]. The optimization is terminated after 50 iterations or if the average loss on the validation set does not decrease for 5 consecutive iterations. Since convergence is sensitive with respect to initial network weights, the optimization is repeated for 7 different initializations, and the one with smallest loss on the validation set is retained. We also use the average loss on the validation set to select the network architecture and activation function. We use a feedforward neural network with two hidden layers, and choose a layer width from between 2 and 5. We also use both the standard Rectified Linear Unit (ReLU) activation function [12], and the Continuously Differentiable Exponential Linear Unit (CELU) [2]. A hyperparameter scan is performed to determine the layer width and activation function. For the first network, the optimal architecture is to use 3 nodes in each hidden layer for the first and third POD coefficients, and 5 nodes for the second coefficient. Using the CELU activation function results in a smaller average loss on the validation set. It was found that regularization via weight decay was not necessary for the network to generalize well to the validation and testing data.

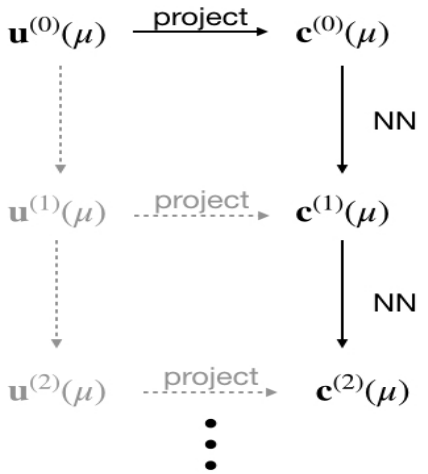
The first network only computes the POD coefficients for the first time step. To compute POD coefficients beyond this first time step, we must train a second network. There



(a) Schematic representation of the POD-ROM method. The reduced POD coefficients only agree with the true projection at the initial condition.



(b) Schematic representation of the evolution of the true POD coefficients. This is too expensive for an efficient reduced-order model, as it requires the computation of the full-order solution at each time step.



(c) Schematic representation of the proposed neural network evaluation of the projection coefficients. The bolded elements are computed in the online stage, while the gray elements show how the training data is constructed in the offline stage.

Fig. 3.1

are a number of choices we must make about the training set. First, for each of the 45 parameters used in the training and validation set, how many snapshots should be computed when constructing the training data? Second, how many of these snapshots should actually be retained in the training data itself? If a network can be trained to predict POD coeffi-

cients for long simulations using the projection coefficients for a small subset of snapshots, this would result in considerable reduction of computational cost, even if this step in the algorithm is part of the offline stage.

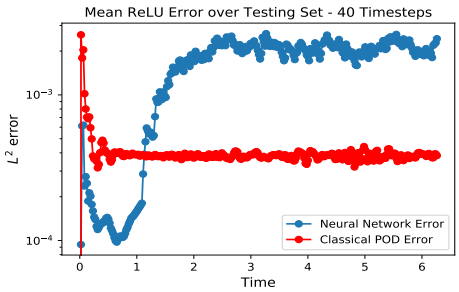
Our goal is to accurately compute POD coefficients out to time $T = 2\pi$, which was the final time used in constructing the snapshot matrix $\boldsymbol{S}$. We keep Δt fixed at $\frac{\pi}{150}$. We solve the full-order problem for varying number of timesteps $N_{\text{train}} \in \{10, 20, 30, 40, 50, 75\}$ and retain 9 randomly selected snapshots. The training data are the corresponding projection coefficients, and the desired outputs are the projection coefficients at the next timestep. While we have access to much more data than 9 snapshots, obtaining an accurate neural network using only a small subset of snapshots is a desirable outcome, especially for problems that are more computationally intensive. Thus, we choose a small number in order to capture the effect of the size of the training data on network accuracy. Again, we select the network architecture and through the validation set. We compare results using both ReLU and CELU activation functions. See figures (3.2) and (3.3).

To test the accuracy of the results we randomly generate 60 more parameters in the interval $[1, 30]$, and take the mean L^2 error over the test parameters for each timestep out to $T = 2\pi$. In figures (3.2) and (3.3), the red curves show the error between the true L^2 projection of the full-order solution and the POD-ROM approximation, and the blue curve shows the error between the projection and the neural network approximation.

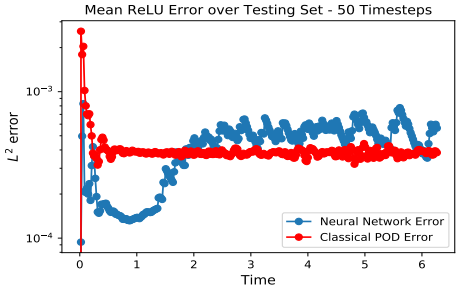
With the ReLU activation function, figure (3.2), we clearly see the benefit of adding a larger number of potential timesteps in the training data. Using only 40 or 50 timesteps in the training data leads to an approximation that quickly becomes worse than the POD-ROM approximation. Using 75 timesteps however, gives a method that is able to outperform POD-ROM.

The CELU activation, figure (3.3), is not as clear cut. With 40 timesteps, we already obtain an approach that is better than POD-ROM. However, using 50 timesteps, which should improve the results, results in a method that is less accurate on average than the standard POD approach. However, once we use 75 timesteps, we obtain an approach that is more accurate than both POD-ROM and the ReLU network over the entire time interval $[0, 2\pi]$.

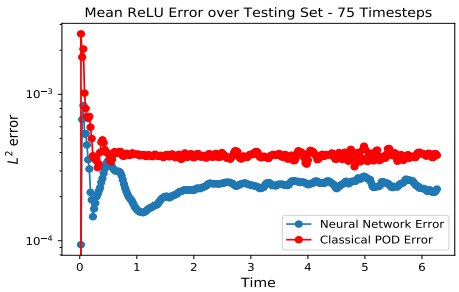
With only a relatively small subset of the available training data, we see a clear gain in accuracy over the POD-ROM method.



(a) 40 Timesteps

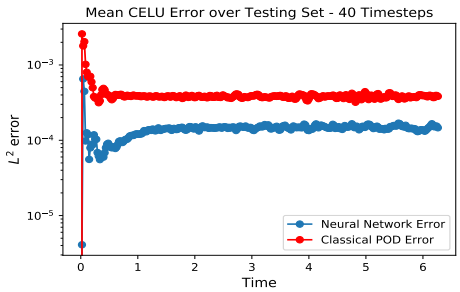


(b) 50 Timesteps

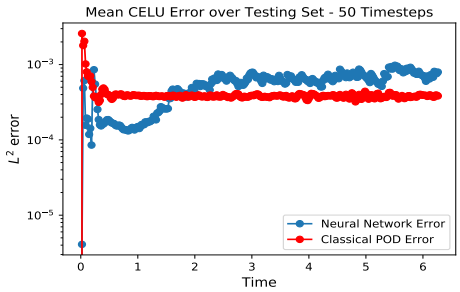


(c) 75 Timesteps

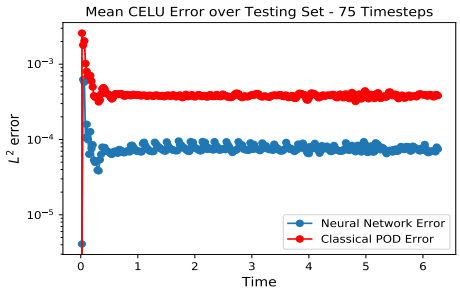
Fig. 3.2: Mean L^2 over Testing Set using ReLU approximation. The red curve is the error between the L^2 projection and the POD-ROM approximation. The blue curve is the error between the projection and the Neural Network approximation. Different numbers of timesteps are used when generating testing data.



(a) 40 Timesteps



(b) 50 Timesteps



(c) 75 Timesteps

Fig. 3.3: Mean L^2 over Testing Set using CELU approximation.

4. Conclusions and Future Work. We have demonstrated the success of the neural network approach to time-dependent parametrized PDEs on a simple test problem in one space dimension. Since the neural network is trained to predict the L^2 projection coefficients, with enough training data, it outperforms the POD-ROM approach in the corresponding norm. This method thus has promise as an improvement over standard model order reduction techniques.

At this point, all results are experimental in nature. A theoretical analysis using approximation results from neural networks would help explain the impact of the size of the training data, as well as the influence of the activation function and architecture of the network. This would also allow for the investigation of how error is propagated through time due to incorrect inputs from previous neural network evaluations.

Extending the technique to more complicated problems is also an obvious next step. Higher-dimensional problems will demonstrate the technique's potential to tackle more physically relevant problems. Applying this method to nonlinear problems is also important, because it is this situation where a very efficient online stage is crucially important. Classical projection techniques cannot result in online stages that are independent of problem size, which is an essential ingredient for computationally efficient reduced order modeling.

Finally, the architecture of the neural network itself should be explored. Residual neural networks are an alternative architecture choice, and recurrent neural networks should be used to try and capture the time-dependent nature of the problem. Nonlinear alternatives to POD compression include autoencoder compression [13], which may be necessary to properly resolve hyperbolic and advection-dominated equations.

REFERENCES

[1] J. A. Atwell and B. B. King, *Proper orthogonal decomposition for reduced basis feedback controllers for parabolic equations*, Mathematical and computer modelling, 33 (2001), pp. 1–19.

[2] J. T. Barron, *Continuously differentiable exponential linear units*, arXiv preprint arXiv:1704.07483, (2017).

[3] S. L. Brunton, J. L. Proctor, and J. N. Kutz, *Discovering governing equations from data by sparse identification of nonlinear dynamical systems*, Proceedings of the National Academy of Sciences, 113 (2016), pp. 3932–3937.

[4] K. T. Carlberg, A. Jameson, M. J. Kochenderfer, J. Morton, L. Peng, and F. D. Witherden, *Recovering missing cfd data for high-order discretizations using deep neural networks and dynamics learning*, Journal of Computational Physics, (2019).

[5] S. Chaturantabut and D. C. Sorensen, *Nonlinear model reduction via discrete empirical interpolation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 2737–2764.

[6] S. Deparis and G. Rozza, *Reduced basis method for multi-parameter-dependent steady navier–stokes equations: applications to natural convection in a cavity*, Journal of Computational Physics, 228 (2009), pp. 4359–4378.

[7] M. A. Grepl, *Reduced-basis approximation a posteriori error estimation for parabolic partial differential equations*, PhD thesis, Massachusetts Institute of Technology, 2005.

[8] M. W. Hess and P. Benner, *Fast evaluation of time–harmonic maxwell's equations using the reduced basis method*, IEEE Transactions on Microwave Theory and Techniques, 61 (2013), pp. 2265–2274.

[9] J. S. Hesthaven and S. Ubbiali, *Non-intrusive reduced order modeling of nonlinear problems using neural networks*, Journal of Computational Physics, 363 (2018), pp. 55–78.

[10] D. Huynh and A. Patera, *Reduced basis approximation and a posteriori error estimation for stress intensity factors*, International Journal for Numerical Methods in Engineering, 72 (2007), pp. 1219–1259.

[11] K. Ito and S. Ravindran, *A reduced basis method for control problems governed by pdes*, Proceedings of control theory for distributed parameters systems, Birkhauser Verlag, Basel, (1997), pp. 173–191.

[12] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, nature, 521 (2015), p. 436.

[13] K. Lee and K. Carlberg, *Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders*, arXiv preprint arXiv:1812.08373, (2018).

[14] R. Milani, A. Quarteroni, and G. Rozza, *Reduced basis method for linear elasticity problems with many parameters*, Computer Methods in Applied Mechanics and Engineering, 197 (2008), pp. 4812–4829.

[15] A. T. Mohan and D. V. Gaitonde, *A deep learning based approach to reduced order modeling for turbulent flow control using lstm neural networks*, arXiv preprint arXiv:1804.09269, (2018).

[16] J. Nocedal and S. Wright, *Numerical optimization*, Springer Science & Business Media, 2006.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, *Automatic differentiation in pytorch*, (2017).

[18] A. T. Patera and E. M. Rønquist, *Reduced basis approximation and a posteriori error estimation for a boltzmann model*, Computer Methods in Applied Mechanics and Engineering, 196 (2007), pp. 2925–2942.

[19] A. Quarteroni, A. Manzoni, and F. Negri, *Reduced basis methods for partial differential equations: an introduction*, vol. 92, Springer, 2015.

[20] G. Rozza, D. B. P. Huynh, and A. T. Patera, *Reduced basis approximation and a posteriori error estimation for affinely parametrized elliptic coercive partial differential equations*, Archives of Computational Methods in Engineering, 15 (2007), p. 1.

[21] S. Sen, *Reduced-basis approximation and a posteriori error estimation for many-parameter heat conduction problems*, Numerical Heat Transfer, Part B: Fundamentals, 54 (2008), pp. 369–389.

IMPLEMENTING PHYSICAL DEPENDENCE IN THE FUNCTIONAL TENSOR TRAIN

TIM REID*, COSMIN SAFTA†, ALEX GORODETSKY‡, JOHN JAKEMAN§, AND KHACHIK SARGSYAN¶

Abstract. The functional tensor train is a functional analogue to the tensor train decomposition that can be used as a surrogate model. We adapt this formulation to model data that depends on both stochastic and physical inputs, while treating stochastic and physical inputs separately. We also develop an optimization algorithms to fit the model to data and test the algorithms on canonical systems of differential equations. The goal of this project is to eventually use a functional tensor train as a surrogate model for the E3SM land model.

1. Introduction. We want to create a surrogate model to functions of the form

$$f(x,p) : \mathcal{X} \times \mathcal{P} \to \mathbb{R}^V, \tag{1.1}$$

where $\mathcal{X}$ is the set of stochastic inputs to the function and $\mathcal{P}$ is the set of physical inputs to the function. The surrogate model that we use is the Functional Tensor Train (FTT) model [12, 6, 5]. The formulation is a continuous analogue to the Tensor Train decomposition [12, 6, 5]. Our goal is to apply the surrogate model to the Energy Exascale Earth System Model (E3SM) Land Model [1]. The E3SM Land Model has both stochastic and physical inputs. Currently FTT models deal with all inputs in the same way and approximate functions with one output [6]. Our goal is to treat the stochastic and physical inputs separately, approximate functions with multiple outputs, and develop optimization methods to fit the modified FTT models to data that depends on stochastic and physical inputs.

1.1. Notation and Tensor Multiplication. A tensor is a multidimensional array, i.e. a vector is a one dimensional tensor, and a matrix is a two dimensional tensor. Lowercase bold letters denote vectors: $\mathbf{a}$ and uppercase bold letters denote matrices: $\mathbf{A}$. Calligraphic uppercase bold letters denote order 3 or higher tensors: $\boldsymbol{\mathcal{A}}$. A lowercase letter with subscripts denote elements of the corresponding tensor; for example, $a_{i,l,j}$ is the entry in position (i,l,j) of $\boldsymbol{\mathcal{A}}$. Variables with superscripts in parentheses denote a specific instance of that variable; for example, $\boldsymbol{\mathcal{A}}^{(k)}$ is the kth tensor in the sequence of tensors $\{\boldsymbol{\mathcal{A}}^{(1)}, \boldsymbol{\mathcal{A}}^{(2)}, \ldots, \boldsymbol{\mathcal{A}}^{(d)}\}$.

Let $\boldsymbol{\mathcal{A}}$ be a $N_1 \times N_2 \times \cdots \times N_d$ tensor and $\mathbf{c}$ be a length N_k vector. The tensor-vector multiplication of $\boldsymbol{\mathcal{A}}$ with $\mathbf{c}$ is an order $(d-1)$, $N_1 \times N_2 \times \cdots \times N_{k-1} \times N_{k+1} \cdots \times N_d$ size tensor with entries

$$(\boldsymbol{\mathcal{A}} \times_k \mathbf{c})_{n_1,n_2,\ldots,n_{k-1},n_{k+1},\ldots,n_d} = \sum_{n_k=1}^{N_k} a_{n_1,n_2,\ldots,n_d} c_{n_k}.$$

The subscript in $\times_k$ refers to the dimension of the tensor that is being contracted with the vector [9].

1.2. Overview of previous work. A FTT approximation for a function with one output that does not depend on a physical input is given by

$$f(x) \approx \widehat{f}(x,\theta) = \mathbf{F}^{(1)}(x_1,\theta_1)\mathbf{F}^{(2)}(x_2,\theta_2)\cdots\mathbf{F}^{(d)}(x_d,\theta_d), \tag{1.2}$$

*North Carolina State University, twreid@ncsu.edu
†Sandia National Laboratories, csafta@sandia.gov
‡University of Michigan, goroda@umich.edu
§Sandia National Laboratories, jdjakem@sandia.gov
¶Sandia National Laboratories, ksargsy@sandia.gov

where d is the dimension of $\mathcal{X}$ [12, 6]. The term $\mathbf{F}^{(k)}(x_k, \theta_k)$, called a *tensor train core*, is a $r_k \times r_{k+1}$ matrix where the entries, $f_{i,j}^{(k)}(x_k, \theta_{k,i,j})$, are univariate functions in x_k that depend on a set of parameters $\theta_{k,i,j}$. The first and last ranks $r_1 = r_{d+1} = 1$ so that the multiplication through the tensor train results in a scalar output. Linearly parameterized univariate functions are written as

$$f_{i,j}^{(k)}(x_k) = \sum_{l=0}^{p_k} \theta_{k,i,j,l} \phi_l^{(k)}(x_k),$$

where $\phi_l^{(k)}(x_k)$ is a basis function with corresponding coefficient $\theta_{k,i,j,l}$. One example of linearly parameterized univariate functions are Polynomial Chaos Expansions (PCEs) that represent the stochastic input x_k [11], where $\phi_l^{(k)}(x_k)$ is a degree l orthonormal basis polynomial [11]. Another choice of a linearly parameterized function is a Fourier series expansion [4]. Non-linearly parameterized functions can, for example, be represented as Gaussian kernels,

$$f_{i,j}^{(k)}(x_k) = \sum_{l=1}^{p_k/2} \theta_{k,i,j,l} \exp\left(\frac{-1}{\sigma^2}(x - \theta_{k,i,j,l+p_k/2})^2\right),$$

where σ is a standard deviation chosen by the user [6]. The tensor train cores do not need to have the same parameterization as the entire set of cores.

In order to simplify the notation, for the rest of this paper we assume that the univariate functions are linearly parameterized. Non-linearly parameterized functions are implemented similarly to linearly parameterized functions.

When the univariate functions are linearly parameterized, we express the tensor train cores as [6],

$$\mathbf{F}^{(k)}(x_k, \theta) = \boldsymbol{\mathcal{A}}^{(k)} \times_2 \boldsymbol{\phi}^{(k)}(x_k),$$

where $\boldsymbol{\mathcal{A}}$ is a $r_k \times p_k \times r_{k+1}$ tensor with

$$a_{i,l,j}^{(k)} = \theta_{k,i,j,l},$$

and $\boldsymbol{\phi}^{(k)}$ is a length p_k vector with basis polynomial ϕ_l as entry l of the vector.

An illustration of a FTT for a function with $d = 5$ inputs is shown in Figure 1.1. The first rank of the first core and the last rank of the last core are both one because we are using the FTT to approximate a function with one output. In the illustrations in this paper blue cores only depend on stochastic inputs x, red cores only depend on physical inputs p, and purple cores depend on both stochastic and physical inputs x and p.

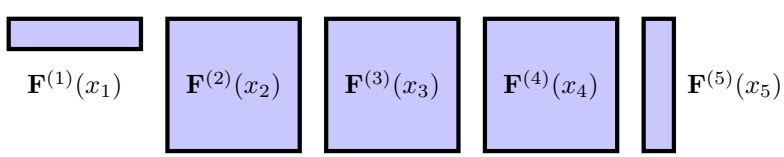


Fig. 1.1: Illustration of FT with $d = 5$ inputs. The cores do not depend on physical inputs.

2. Variations of the Functional Tensor Train Approximation. We present three FTT formulations that will be used to approximate functions with both stochastic inputs and physical inputs. These variants differ in how the physical input is handled by in the ensemble of FTT cores. These options are called the Multi-Output Tensor Train (MTT), the Augmented Tensor Train (ATT), and the Parametric Tensor Train (PTT).

2.1. Multi-Output Tensor Train (MTT). The MTT incorporates dependence on physical inputs by evaluating the FTT at N points at the same time. These N points represent values for N different physical inputs.

The FTT is evaluated at multiple points by fitting multiple FTTs to the data points with one FTT per data point. Some cores can be shared between the FTTs. This happens when the input corresponding to the shared core does not vary for the different outputs. Sharing FTT cores across several models reduces the total number of parameters.

Figure 2.1 illustrates an FTT with $d = 5$ inputs and three outputs. The first, third, and fifth cores are shared between the three outputs, while the second and fourth cores are different instances.

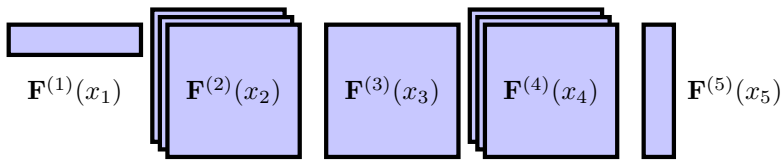


Fig. 2.1: Illustration of FT with $d = 5$ inputs and 3 outputs. The first, third, and fifth cores are shared for all three outputs, while the second and fourth core have a different instance for each output.

This method of creating a FTT with multiple outputs is used by the ATT and PTT versions of the FTT to approximate a function with multiple outputs.

2.2. Augmented Tensor Train (ATT). The ATT incorporates dependence on physical inputs by inserting an additional tensor train core in position k that depends on the physical inputs. This amounts to treating the physical inputs similarly to the stochastic inputs. The resulting tensor train,

$$\widehat{f}(x,p,\theta) = \mathbf{F}^{(1)}(x,\theta_1)\mathbf{F}^{(2)}(x,\theta_2),\cdots,$$
$$\mathbf{F}^{(k-1)}(x,\theta_{k-1})\mathbf{F}^{\mathcal{P}}(p,\theta_{\mathcal{P}})\mathbf{F}^{(k)}(x_k,\theta_k)\cdots\mathbf{F}^{(d)}(x_d,\theta_d),$$

has $d+1$ cores: d cores for the stochastic inputs and 1 core for the physical input. The tensor train core dependent on the physical inputs, p, is

$$\mathbf{F}^{\mathcal{P}}(p,\theta_{\mathcal{P}}) = \boldsymbol{\mathcal{A}}^{\mathcal{P}} \times_2 \boldsymbol{\psi}(p),$$

where $\boldsymbol{\mathcal{A}}^{\mathcal{P}}$ is a tensor of coefficients and $\boldsymbol{\psi}(p)$ is a vector of basis functions evaluated at p.

The basis functions, ψ, do not need to be univariate. If the physical input is multidimensional, then ψ can be a multivariate function, and it can be constructed as a tensor product of the univariate basis functions [11].

An example of the ATT is given in Figure 2.2. There are $d = 5$ stochastic inputs and the fourth core corresponds to physical inputs. The cores dependent on stochastic inputs are blue and the core dependent on physical inputs is red.

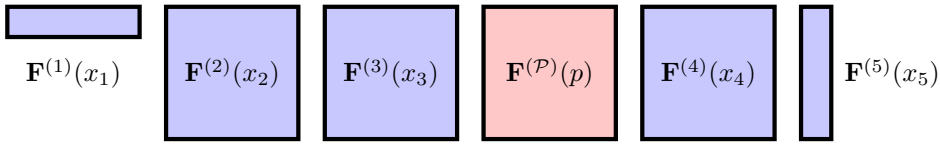


Fig. 2.2: Augmented tensor train with 5 stochastic inputs and with the tensor core for the physical inputs as the fourth core.

2.3. Parametric Tensor Train (PTT). The PTT incorporates the physical inputs, p, by making the parameters, $\theta_{k,i,j,l}$, dependent on the physical inputs. We do this by redefining the parameter, $\theta_{k,i,j,l}$, as a parameterized function:

$$\theta_{k,i,j,l} = \sum_{s=1}^{S} \theta_{k,i,l,j,s} \psi_s(p),$$

where $\psi_s(p)$ is a basis function with corresponding coefficient $\theta_{k,i,j,l,s}$. The basis functions for the physical inputs are handled in the same way as for the ATT approach. The dependence on two sets of basis functions redefines the tensor train cores as:

$$\mathbf{F}^{(k)}(x, p, \theta_k) = \left(\boldsymbol{\mathcal{A}}^{(k)} \times_4 \boldsymbol{\psi}(p) \right) \times_2 \boldsymbol{\phi}(x_k). \tag{2.1}$$

The coefficient tensor, $\boldsymbol{\mathcal{A}}^{(k)}$, is now a $r_k \times p_k \times r_{k+1} \times S$ tensor. The last dimension of $\boldsymbol{\mathcal{A}}^{(k)}$ are the coefficients to the basis functions, ψ. The result of the tensor contraction $\boldsymbol{\mathcal{A}}^{(k)} \times_4 \boldsymbol{\psi}(p)$ is a $r_k \times p_k \times r_{k+1}$ tensor that is used in the same way as the coefficient tensor for the MTT and PTT.

An example of the PTT is given in Figure 2.3. There are $d = 5$ stochastic inputs and all cores are also dependent on the physical inputs.

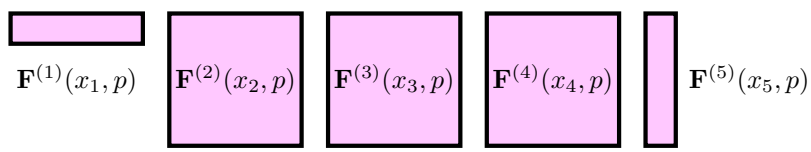


Fig. 2.3: Illustration of PTT for problem with 5 stochastic inputs and all cores also dependent on physical inputs.

It is possible to introduce dependence on the physical inputs to only a subset of cores, leaving the rest dependent only on the stochastic inputs. This can be done if some of the stochastic inputs do not depend on the physical inputs. This is somewhat similar to when only some cores in the MTT have a different instance for each physical input. The advantage of not introducing a dependence on all the cores is that it reduces the amount of parameters that need to be computed.

The ATT can be viewed as a special case of the PTT. This is because the PTT can be made into the ATT using the following steps:

1. Add a dimension to stochastic input x in position k
2. Make the univariate functions corresponding to the new dimension of x degree zero polynomials
3. Only introduce dependence on p to the parameters corresponding to the new dimension of x

This works because parameterizing the new tensor train core with degree zero polynomials removes the dependence on x from that tensor train core.

3. Computing the coefficients for the PTT. We start with a data set $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{M \times N \times V}$ that was obtained from some function f in the form of (1.1). The first dimension of $\boldsymbol{\mathcal{Y}}$ corresponds to the output of the function for M instances of the stochastic inputs x, the second dimension corresponds to N instances of the physical inputs p, and the third dimension corresponds to the V outputs of the function. To fit a PTT, $\widehat{f}(x, p)$, to the data,

we find the set of parameters, θ, that minimize the objective function

$$J(\theta) = \frac{1}{2}\left[\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{u=1}^{V}\left(y_{m,n,u} - \widehat{f}_u(x^{(m)}, p^{(n)}, \theta)\right)^2 + \lambda\|\theta\|_2^2\right]. \tag{3.1}$$

This the least squares objective function with L^2 regularization [2]. The regularization parameter, λ, is chosen by the user. When the univariate functions in the tensor train cores are orthogonal and linearly parameterized, then the L^2 regularization is equivalent to the regularization presented in [6]. To minimize this function we can use alternating least squares [9, 13] or gradient based methods such as L-BFGS [3] and Stochastic Gradient Descent [2] or its variants [8].

3.1. Gradient of the Objective Function. To use gradient based methods, we need to know the partial derivative of (3.1) with respect to all the coefficients θ.

PROPOSITION 3.1. *[6, Extension of Proposition 1] The partial derivative of the objective function $J(\theta)$ from (3.1) with respect to the coefficient $\theta_{k,i,j,v,l,s}$ is*

$$\frac{\partial J}{\partial \theta_{k,i,j,v,l,s}} = \sum_{m=1}^{M}\sum_{n=1}^{N}\left(y_{m,n,v} - \widehat{f}_v(x^{(m)}, p^{(n)})\right)\frac{\partial \widehat{f}_v(x^{(m)}, p^{(n)}, \theta)}{\partial \theta_{k,i,j,v,l,s}} + 2\lambda\theta_{k,i,j,v,l,s}$$

The partial derivative of the PTT with respect to the same coefficient is

$$\begin{aligned}\frac{\partial \widehat{f}_v(x^{(m)}, p^{(n)}, \theta)}{\partial \theta_{k,i,j,v,l,s}} &= \mathbf{F}_v^{(1)}(x_1^{(m)}, p^{(n)}, \theta_1)\mathbf{F}_v^{(2)}(x_2^{(m)} p^{(n)}, \theta_2)\cdots \\ &\quad \mathbf{F}_v^{(k-1)}(x_{k-1}^{(m)}, p^{(n)}, \theta_{k-1})\mathbf{G}\mathbf{F}_v^{(k+1)}(x_{k+1}^{(m)}, p^{(n)}, \theta_{k+1})\cdots\mathbf{F}_v^{(d)}(x_d^{(m)}, p^{(n)}, \theta_d),\end{aligned} \tag{3.2}$$

where $\mathbf{F}_v^{(k)}$, is the tensor train core corresponding to output v of $f^{(k)}$. $\mathbf{G}$ is a $r_k \times r_{k+1}$ matrix defined as

$$g_{w,z} = \begin{cases} \frac{\partial f_{w,z,v}(x_k^{(m)}, p^{(n)}, \theta)}{\partial \theta_{k,i,j,v,l,s}} & (w,z) = (i,j) \\ 0 & (w,z) \neq (i,j), \end{cases} \tag{3.3}$$

where $f_{w,z,v}$ is entry (w,z) of $\mathbf{F}_v^{(k)}$.

Proof. Because taking a partial derivative is a linear operator, it is sufficient to show the derivative for one term in the sum of the objective function, J, from (3.1).

First apply the chain rule to one term in the sum to get

$$\begin{aligned}&\frac{\partial}{\partial \theta_{k,i,j,v,l,s}}\frac{1}{2}\left(y_{m,n,u} - \widehat{f}_u(x^{(m)}, p^{(n)}, \theta)\right)^2 \\ &\qquad = \left(y_{m,n,u} - \widehat{f}_u(x^{(m)}, p^{(n)}, \theta)\right)\frac{\partial \widehat{f}_u(x^{(m)}, p^{(n)}, \theta)}{\partial \theta_{k,i,j,v,l,s}}\end{aligned} \tag{3.4}$$

with $1 \leq n \leq N$, $1 \leq m \leq M$, and $1 \leq u \leq V$.

Now we compute the partial derivative of the PTT approximation, given by

$$\widehat{f}_u(x^{(m)}, p^{(n)}) = \mathbf{F}_u^{(1)}(x_1^{(m)}, p^{(n)})\mathbf{F}_u^{(2)}(x_2^{(m)}, p^{(n)})\cdots\mathbf{F}_u^{(d)}(x_d^{(m)}, p^{(n)})$$

Apply the product rule to this equation and get

$$\frac{\partial \widehat{f}_u(x^{(m)}, p^{(n)})}{\partial \theta_{k,i,j,v,l,s}} = \sum_{q=1}^{d} \left(\prod_{b=1}^{q-1} \mathbf{F}_u^{(b)}(x_b^{(m)}, p^{(n)}) \right) \frac{\partial \mathbf{F}_u^{(q)}(x_q^{(m)}, p^{(n)})}{\partial \theta_{k,i,j,v,l,s}} \left(\prod_{b=q+1}^{d} \mathbf{F}_u^{(b)}(x_b^{(m)}, p^{(n)}) \right). \quad (3.5)$$

Finally, we compute the partial derivative of core q. The parameter $\theta_{k,i,j,v,l,s}$ only appears in core k for output v, so

$$\frac{\partial \mathbf{F}_u^{(q)}(x_q^{(m)}, p^{(n)})_u}{\partial \theta_{k,i,j,v,l,s}} = 0, \qquad q \neq k \text{ or } u \neq v \quad (3.6)$$

Additionally, parameter $\theta_{k,i,j,v,l,s}$ only appears in entry (i,j) of $\mathbf{F}_v^{(k)}(x_k^{(m)}, p^{(n)})$, so the partial derivative of this core with respect to $\theta_{k,i,j,v,l,s}$ is the partial derivative of the univariate function in position (i,j):

$$\mathbf{G} := \frac{\partial \mathbf{F}_v^{(k)}(x_k^{(m)}, p^{(n)})}{\partial \theta_{k,i,j,v,l,s}} = \begin{cases} \frac{\partial f_{w,z,v}(x_k^{(m)}, p^{(n)}, \theta)}{\partial \theta_{k,i,j,v,l,s}} & (w,z) = (i,j) \\ 0 & (w,z) \neq (i,j). \end{cases}$$

Substituting this equation along with (3.6) into (3.5) results in (3.2). Substituting (3.2) into 3.4) proves the proposition. □

3.2. Alternating Linear Least Squares. If the univariate functions in the tensor train cores are linearly parameterized, then we can use alternating *linear* least squares to minimize (3.1). Alternating least squares works by fixing the parameters of all cores except one then performing linear least squares to solve for the parameters in the remaining core. We then cycle through the same process for the rest of the cores, and possibly repeat the process.

Here we show alternating least squares algorithm to compute the parameter tensor $\mathcal{A}^{(\mathbf{k},\mathbf{v})}$ in the PTT by showing how to express the PTT as a matrix-vector product with a reshaped version of $\mathcal{A}^{(\mathbf{k},\mathbf{v})}$ as the vector. Express the PTT as

$$\widehat{f}(x^{(m)}, p^{(n)}, \theta)_v = \mathbf{T}^{(m,n,v)} \times \left(\left(\boldsymbol{\mathcal{A}}^{(k,v)} \times_4 \Psi(p^{(n)}) \right) \times_2 \Phi^{(k)}(x_k^{(m)}) \right) \times \mathbf{U}^{(m,n,v)}$$

where $\mathbf{T}^{(m,n,v)}$ and $\mathbf{U}^{(m,n,v)}$ are $r_1 \times r_k$ and $r_k \times r_{d+1}$ matrices, respectively, representing the multiplication of all cores before and after core k for output v and instance m and n of inputs x and p respectively. Define the $r_1 M r_{d+1} NV \times r_k p_k r_{k+1} S$ matrix $\mathbf{H}$ as

$$h_{\alpha,\beta} = \psi_s(p^{(n)}) \phi_l(x^{(m)}) t_{w,r}^{(m,n,v)} u_{q,z}^{(m,n,v)}$$

and the $r_k p_k r_{k+1} S$ length vector $\widehat{\boldsymbol{a}^{(k,v)}}$ as

$$\widehat{a^{(k,v)}}_\beta = a_{r,p,q,s}^{(k,v)},$$

where the indices α and β are

$$\alpha = wMr_{d+1}NV + mr_{d+1}NV + zNV + nV + v$$
$$\beta = rp_k r_{k+1} S + pr_{k+1} S + qS + s.$$

Then

$$\mathbf{H}\widehat{\boldsymbol{a}^{(k,v)}} = \widehat{\boldsymbol{y}},$$

where $\widehat{\boldsymbol{y}}$ is a reshaped version of $\boldsymbol{\mathcal{Y}}$ with

$$\widehat{y}_\alpha = y_{m,n,v}.$$

Linear least squares can be used to solve

$$\begin{pmatrix} \mathbf{H} \\ \lambda I \end{pmatrix} \widehat{\boldsymbol{a}^{(k,v)}} = \begin{pmatrix} \widehat{\boldsymbol{y}} \\ 0 \end{pmatrix}$$

for $\widehat{\boldsymbol{a}^{(k)}}$, which is equivalent to solving for the optimal coefficient tensor $\boldsymbol{\mathcal{A}}^{(k)}$ with L^2 regularization parameter λ.

3.3. Choosing the Regularization Parameter. To choose the regularization parameter λ, we divide our data for fitting an FTT model into a training set and a validation set. We then use the training set to fit the FTT and use the validation set to measure the error of the fitted FTT model for several values of λ. The value of λ that produces the lowest error with the validation set is selected for the optimization process.

4. Testing the Algorithms. We test the algorithms that compute the FTT approximation by validating the gradient computation and then using the FTT to approximate synthetic data.

4.1. Testing the Gradient. The gradient in the FTT algorithms is computed analytically using the formula in Proposition 3.1. To verify that the gradient was derived and implemented correctly, we compare the analytic gradient to an approximation of the gradient computed using finite differences. The error was measured using the relative infinity norm error between the analytical expression and finite differences gradients. The gradient was compared using 10 random PTTs that have 10 cores each. The median and maximum errors from the 10 tests are in given Table 4.1. The PTT code is used to generate both the MTT and ATT variants of the FTT so it is sufficient to only test the gradient function with the PTT code.

Median infinity-norm error	Maximum infinity-norm error
3.84×10^{-7}	6.90×10^{-7}

Table 4.1: Relative errors for the gradient functions

4.2. Approximating a random tensor with a functional tensor train model. Next we approximate a randomly generated FTT with a similar model. For this test we assumed a 10-dimensional input x, the rank of each tensor train core was 4, and the univariate functions for x were degree 2 Legendre polynomials. For the ATT and PTT tests, the functions for p were degree 2 Chebyshev polynomials. We trained and tested the FTT on 100 instances of x and for the ATT and PTT, 100 instances of p. The PTT had physical dependence imposed on the last core. The FTT approximations were trained using L-BFGS. The errors on the training and test data are in Table 4.2

Model	Training Error	Testing Error	Time To Compute
MTT	2.43×10^{-2}	4.92×10^{-2}	1.03
ATT	1.50×10^{-2}	3.90×10^{-2}	4.55
PTT	3.60×10^{-3}	1.17×10^{-2}	17.33

Table 4.2: Training and testing error for FTT approximations with all three models of random FTT data as well as the time to compute the approximations.

The ATT and PTT both had lower error than the MTT while the PTT took longer to compute than the others approaches. This is because imposing spatial dependence on an existing core multiplies the number of parameters that need to be solved, having the same effect on the number of parameters as adding S tensor cores without dependence. This is contrasted with the ATT that only adds one tensor core to the problem. The quality of the code does not have an effect on the speed difference of the PTT and ATT because the ATT was computed using the PTT code. The accuracy and time behaviors are seen in the other test problems as well.

4.3. Approximating systems of ODEs with an FT. We also generated test data using systems of ODEs, specifically the Lorenz modes [10],

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x)\\ \frac{dy}{dt} &= x(\rho - z) - y\\ \frac{dz}{dt} &= xy - \beta z,\end{aligned}$$

and the SIR model [7],

$$\begin{aligned}\frac{dS}{dt} &= -\beta SI\\ \frac{dI}{dt} &= \beta SI - \gamma I\\ \frac{dR}{dt} &= \gamma I.\end{aligned}$$

We treated the parameters in the models above as stochastic inputs and time as the physical input. For both ODE systems, none of the FTT cores were shared between the different outputs, and all cores in the PTT were dependent on time.

For the Lorenz system, we sampled 160 times points between $t = 2$ and $t = 3$ and 160 normally distributed parameters with means of 28, 10, and 8/3 for σ, ρ, and β respectively and with standard deviations of 5% of those nominal values. The univariate functions for x were degree 2 Hermite polynomials and the functions for p were 50 term Fourier series. The ranks of all the tensor train cores was 4 and the FTT models were computed using Alternating Least Squares. The training and test errors are given in Table 4.3, and plots of ODE solutions with one sample of training and testing data each are in Figure 4.1.

Model	Training Error	Testing Error	Time To Compute
ATT	1.07×10^{-1}	1.53×10^{-1}	0.46
PTT	5.08×10^{-2}	1.35×10^{-1}	45.29

Table 4.3: Training and testing error for FTT approximations ATT and PTT models of Lorenz model data as well as the time to compute the approximations.

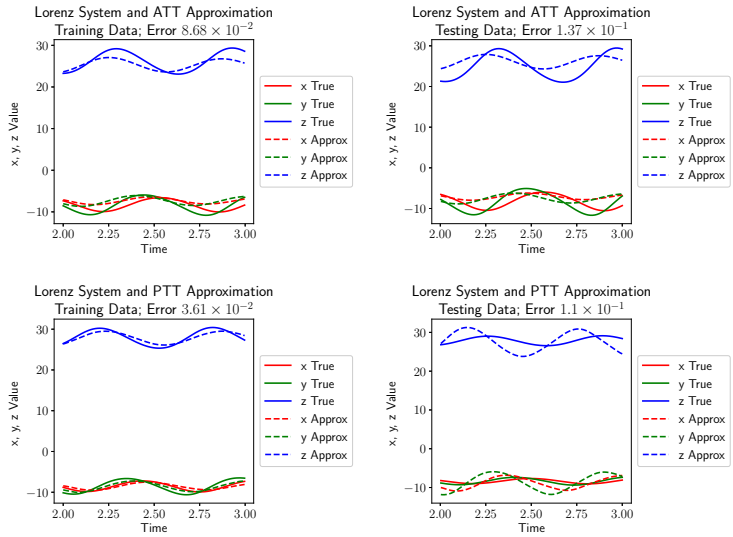


Fig. 4.1: Lorenz system true value and FTT model approximation. Approximations with the median error for the training and testing sets were chosen. The error in the plot title is the relative error for this particular sample.

The SIR model was approximated in the same way as the Lorenz system except for a few changes. We selected 100 points along the time axis between 0 and 3. A number of 50 stochastic parameter samples were picked from independent normal distributions with means 0.2 and 0.8 for β and γ respectively with a standard deviation of 25% of those nominal values. The FTT models were computed using L-BFGS. The results for the SIR model FTT approximations are in Table 4.4, and plots of ODE with one sample of training and testing data each are in Figure 4.2.

Model	Training Error	Testing Error	Time To Compute
ATT	6.21×10^{-2}	9.31×10^{-2}	11.82
PTT	1.20×10^{-2}	9.29×10^{-2}	481.54

Table 4.4: Training and testing error for FTT approximations using ATT and PTT for the SIR model data as well as the time to compute the approximations.

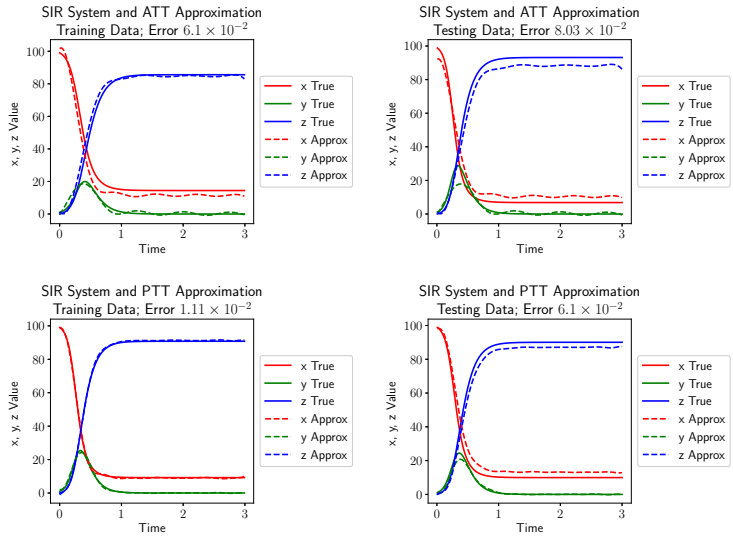


Fig. 4.2: SIR model true value and FTT model approximation. Approximations with the median error for the training and testing sets were chosen. The error in the plot title is the relative error for this particular sample.

The results for the Lorenz and SIR ode systems mirror the results to approximating the random FTT. The ATT and PTT have similar levels of accuracy and the PTT is slower to compute than the ATT. The plots of the ODE and the FTT approximation show the approximation from the training or testing set that has the median error. The approximations for the SIR model are more accurate than the approximations for the Lorenz system. This is most likely because the chaotic nature of Lorenz model.

5. Summary and Future Work. The functional tensor train can be used to approximate data that depends on some inputs. We implemented three variations of the functional tensor train that treat stochastic and physical inputs to the data separately. The variations incorporate physical inputs into the FTT by adding a new core and treating the physical inputs as one more stochastic input (ATT), or the physical inputs were incorporated by making the coefficients in the cores dependent on the physical inputs (PTT). In our numerical experiments both methods result in similar levels of accuracy but the ATT model is faster to compute than the PTT model since the number of parameters in the former is less compared to the later approach.

REFERENCES

[1] *Energy Exascale Earth System Model.* e3sm.org. Accessed: July 17, 2019.
[2] L. Bottou, F. E. Curtis, and J. Nocedal, *Optimization methods for large-scale machine learning*, Siam Review, 60 (2018), pp. 223–311.
[3] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, *A limited memory algorithm for bound constrained optimization*, SIAM Journal on Scientific Computing, 16 (1995), pp. 1190–1208.
[4] W. Gautschi, *Numerical Analysis*, Springer Science & Business Media, 1997.
[5] A. Gorodetsky, S. Karaman, and Y. Marzouk, *A continuous analogue of the tensor-train decomposition*, Computer Methods in Applied Mechanics and Engineering, 347 (2019), pp. 59–84.
[6] A. A. Gorodetsky and J. D. Jakeman, *Gradient-based optimization for regression in the functional tensor-train format*, Journal of Computational Physics, 374 (2018), pp. 1219–1238.

[7] W. O. Kermack and A. G. McKendrick, *A contribution to the mathematical theory of epidemics*, Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 115 (1927), pp. 700–721.

[8] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[9] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.

[10] E. N. Lorenz, *Deterministic nonperiodic flow*, Journal of the Atmospheric Sciences, 20 (1963), pp. 130–141.

[11] H. N. Najm, *Uncertainty quantification and polynomial chaos techniques in computational fluid dynamics*, Annual Review of Fluid Mechanics, 41 (2009), pp. 35–52.

[12] I. Oseledets, *Constructive representation of functions in low-rank tensor formats*, Constructive Approximation, 37 (2013), pp. 1–18.

[13] A. Uschmajew, *Local convergence of the alternating least squares algorithm for canonical tensor approximation*, SIAM Journal on Matrix Analysis and Applications, 33 (2012), pp. 639–652.

Software & High Performance Computing

Articles in this section discuss the implementation of high performance computing (HPC) and productivity software. In many cases, performance improvements and portability are demonstrated for many-core architectures, such as conventional multicore CPUs and graphical processing units (GPUs). Other articles discusss virtual machines and web applications.

M. Powell
M.L.Parks

September 21, 2020

LINEAR ALGEBRA-BASED TRIANGLE COUNTING VIA FINE-GRAINED TASKING ON HETEROGENEOUS ENVIRONMENTS

ABDURRAHMAN YAŞAR* AND SIVASANKARAN RAJAMANICKAM†

Abstract. Triangle counting is a representative graph problem that shows the challenges of improving graph algorithm performance using algorithmic techniques and adopting graph algorithms to new architectures. In this paper, we describe an update to the linear-algebraic formulation of the triangle counting problem. Our new approach relies on fine-grained tasking based on a tile layout. We adopt this task based algorithm to heterogeneous architectures (CPUs and GPUs) for up to 10.8x speed up over past year's graph challenge submission. This implementation also results in the fastest kernel time known at time of publication for real-world graphs like twitter (3.7 second) and friendster (1.8 seconds) on GPU accelerators when the graph is GPU resident. This is a 1.7 and 1.2 time improvement over previous state-of-the-art triangle counting on GPUs. We also improved end-to-end execution time by overlapping computation and communication of the graph to the GPUs. In terms of end-to-end execution time, our implementation also achieves the fastest end-to-end times due to very low overhead costs.

1. Introduction. With increased use of accelerators for achieving better performance, proposing algorithms that require ideally no architecture specific changes on code base is crucial for portability on heterogeneous environments. This paper addresses these two primary problems, using general purpose accelerators for the graph challenge and doing that in a portable manner.

In this paper, we focus on a triangle counting algorithm that utilizes both CPUs and GPUs on a compute node and uses a portable tiled layout that requires almost no (algorithmic or layout) change between different architectures. This paper improves our previous work [12, 13] by using the linear algebra based triangle counting algorithm on a tiled layout and exploiting multiple levels of shared-memory parallelism on CPUs and GPUs.

This algorithmic changes allow us to achieve the fastest times for real world graphs like twitter and friendster compared to past champions utilizing the GPUs. We achieve 3.7 seconds on twitter and 1.8 seconds on friendster graphs as opposed to 6.5 and 2.1 seconds by past champions [5] when the graph is GPU resident. However, we believe assuming the graph is on the GPU is not realistic for several use cases, therefore in this paper we also focus on an end-to-end time metric when the graph is not GPU resident. In this case, we are able to count triangles in twitter and friendster graphs in 4.6 and 3.1 seconds where data copy is overlapped with computation. In the following sections we only report end to end results. In this paper, we propose a new **linear algebraic formulation** for triangle counting problem that uses tiles and a **fine-grained parallel algorithm** that exploits multiple level of parallelism on different architectures. We implement a highly efficient **multi-core**, **multi-GPU** hybrid framework that outperforms state-of-the-art. Experimental results demonstrate that our codes achieve the fastest kernel times on real-world graphs when the graph resides on the GPU and the fastest end-to-end times when the graph is not on the GPU. The performance improvement is up to 10× over our previous state-of-the-art implementation.

2. Background.

2.1. 2017 Static Graph Challenge. We used a linear algebra-based triangle counting implementation, KKTri (previously designated TCKK) [12] in the the 2017 Static Graph Challenge [10]. That work, focused on efficient shared memory parallelism on top of a portable SpGEMM (called KK-MEM) [2] in the Kokkos Kernels library [6]. The primary focus of that work was on two linear-algebra based formulations of triangle counting:

*Georgia Institute of Technology, ayasar@gatech.edu
†Sandia National Laboratories, srajama@sandia.gov

1. $D = (L \times U). * L$: This formulation represented triangle counting in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication where L and U are the lower and upper triangle parts of the adjacency matrix for the graph.
2. $D = (L \times L). * L$: This formulation was used primarily for the 2017 Graph Challenge. This formulation follows the same logic as the previous method.

Three optimizations were used to achieve good performance: (1) in-place masked SpGEMM which reduced the memory needed for triangle counting; (2) data compression on the right hand side matrix that allowed using efficient bitwise operations (3) ordering of the vertices a common heuristic to reduce number of operations.

2.2. 2018 Static Graph Challenge. For the 2018 Static Graph Challenge [9], we designed a linear algebra-based triangle counting implementation KKTri-Cilk that inherited from the KK-SpGEMM algorithm [3] and improved load-balancing and efficient hyper-thread usage issues using Cilk based programming model and optimizations.

The parallelization strategy and the runtime system is the main difference between KKTri-Cilk and KKTri. KKTri used a very simple scheme, partitioning the matrix evenly into partitions of a fixed number of rows. To balance the work among the tasks, KKTri-Cilk uses an heuristic to find the partitions, creating partitions such that the number of non-zeros within each partition are approximately equal.

Compression of the right hand side matrix can decrease the problem size significantly, and allow using efficient bitwise operations. However, compression is not always successful because of the natural order of the matrices.

3. Approach. A lightweight 2D partitioning algorithm is implemented to create tiles. This algorithm partitions the graph in two dimensional space where diagonal tiles are required to be squares. This partitioning is known as symmetric generalized block distribution [4].

In Equation 3.1, we propose a new linear algebra based triangle counting formulation that uses tiles. Similar to LL and LU this formulation represents triangle counting in terms of sparse matrix-matrix multiplication followed by an element-wise matrix multiplication but with tiles. If number of tiles is one then this formulation is identical with LL Algorithm [12]. Figure 3.1 illustrates this formulation. In the context of this paper, a "task", t, counts the triangles in a given triple of tiles, $t = \{T_{i,j}, T_{j,k}, T_{i,k}\}$. However, counting triangles in all possible triple of tiles, ends up counting each triangle three times. Considering tasks, $t = \{T_{i,j}, T_{j,k}, T_{i,k}\}$ where $i \leq j \leq k$ avoids unnecessary counting.

$$D_{i,j} = (T_{i,j} \times T_{j,k}). * T_{i,k} \tag{3.1}$$

Latapy *et al.* [7] proposes to use list intersection based counting for small degree vertices and a hashmap based intersection for the other. In this work, a similar approach is used; i.e. any task with sparse tiles will use list based intersection and denser tiles will use a dense hashmap accumulator.

We use both CPUs and GPUs to process tasks together. There is no architecture specific algorithmic change in the code-base. This will allow execution on any accelerator and CPU combination. However, because of architectural differences, the parallelization approach differs between CPU and GPU itself. While, CPU threads execute different tasks in parallel, GPU threads execute cooperatively to complete the same task in parallel. Assigning heavier tasks to GPUs is one of the primary optimizations that is being used in hybrid approaches [11] due to massive parallelism capabilities of these devices. Hence, we try to estimate and execute heavier tasks on GPUs.

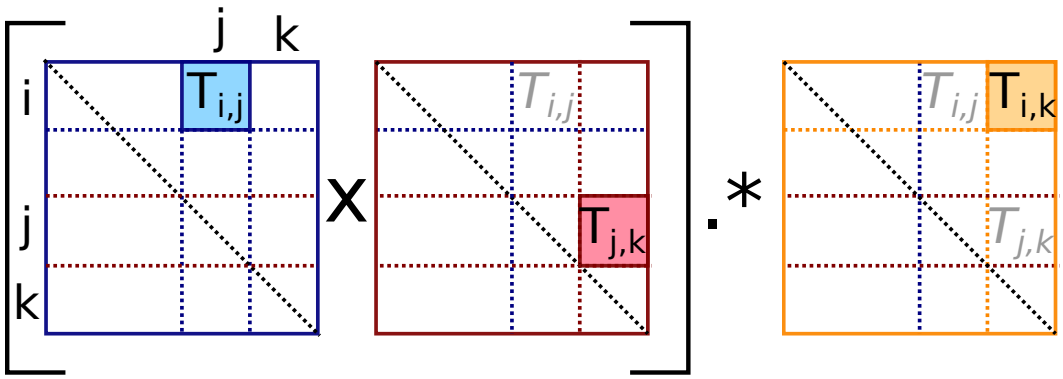


Fig. 3.1: Tiled Triangle Counting

We use Cuda streams to simultaneously execute several tasks on the GPUs. Four Cuda streams are created for each GPU in the node. Then, a CPU thread is created and made responsible for the operations on the stream. GPUs and CPUs compete for tasks and get a new one from a queue when they are ready. Tasks are ordered based on their size in a task queue. GPUs start to process tasks starting from the heaviest task and CPUs start to process tasks from the lightest tasks. This continues until all tasks have been executed.

4. Experimental Evaluation. We present several experiments to identify the performance trade-offs of the proposed work. These experiments were carried out on three architectures with multicore processors and GPUs that are shown in Table 4.1. GNU compiler (g++) version 7.2, Cuda runtime version 10.0 and OpenMP version 4.0 are used to compile and run the code on all the architectures.

4.1. Dataset and Copy Time Included Peak Rates. Table 4.2 lists 23 graphs that we used in our experiments along with the number of vertices ($|V|$), number of edges ($|E|$), number of triangles ($|T|$), size of the graph in memory (Raw Size), number of tiles and number of tasks in the graph. In addition to 20 Challenge graphs for which triangle counting is particularly costly, 3 additional large real-world graphs [8, 1] are included in our experiments. We used the Graph Challenge procedure of symmetrizing the matrices (using undirected graphs). For all experiments, we report the best of the median time of five runs with different number of GPUs. Table 4.2 reports copy included best execution time and rates on different architectures for each graph. This work outperforms last year's submission by 6× on friendster graph and achieves 5.6×10^8 rate.

4.2. Relative Speedup comparisons to Last Year's Submission. Figure 4.1 presents relative speedup between this work and our last submission (KKTri-Cilk) [13]. This work outperforms KKTri-Cilk in 20 of 23 cases. In three small instances (flickrEdges, amazon0505 and cit-Patents) KKTri-Cilk performs better. This years work can achieve up to 11× speedup on large graphs in which GPUs become more useful.

4.3. Strong Scaling on GPUs. Figure 4.2 presents strong scaling of the three largest graphs with respect to different number of GPUs using Stream and CopyFirst algorithms. These results are gathered on DGX architecture and all tasks are executed on GPUs. From Figure 4.2(b) we observe that triangle counting algorithm has an almost linear strong scaling with respect to number of GPUs. However, for the CopyFirst algorithm (see Figure 4.2(c)), when number of GPUs on the node is increased copy time becomes the bottleneck and

Table 4.1: Overview of the Architectures. Pinned: transfers using pinned memory. Page-able: transfers using page-able memory. H2D: host to device. D2H: device to host.

		DGX	**Newell**	**Minsky**
CPU		Intel, E5-2698	POWER9	POWER8-NVL
Cores		2×40	2×16	2×8
Host Memory		512 GB	320 GB	512 GB
L2-Cache		256 KB	512 KB	512 KB
L3-Cache		50 MB	10 MB	8 MB
GPU		V100-SXM2	V100-SXM2	P100-SXM2
GPU Memory		32 GB	32 GB	16 GB
Number of GPUs		8	2	4
Pageable	H2D	9.2 GB/S	12.2 GB/S	15.1 GB/S
	D2H	8.0 GB/S	14.2 GB/S	8.9 GB/S
Pinned	H2D	10.7 GB/S	60.0 GB/S	31.4 GB/S
	D2H	12.1 GB/S	60.0 GB/S	32.6 GB/S

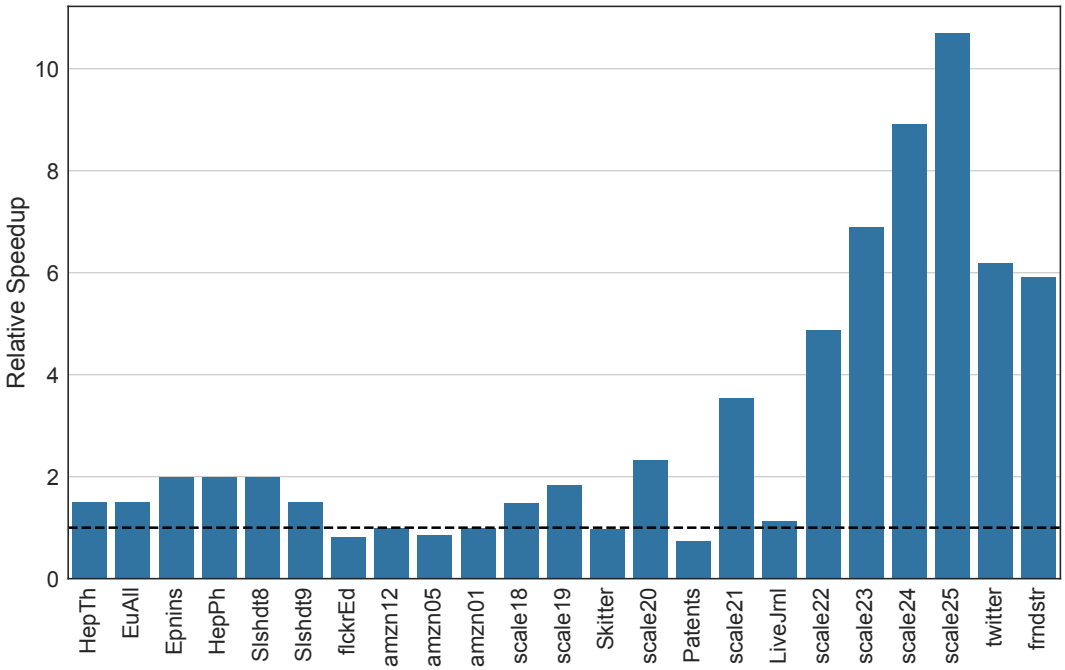
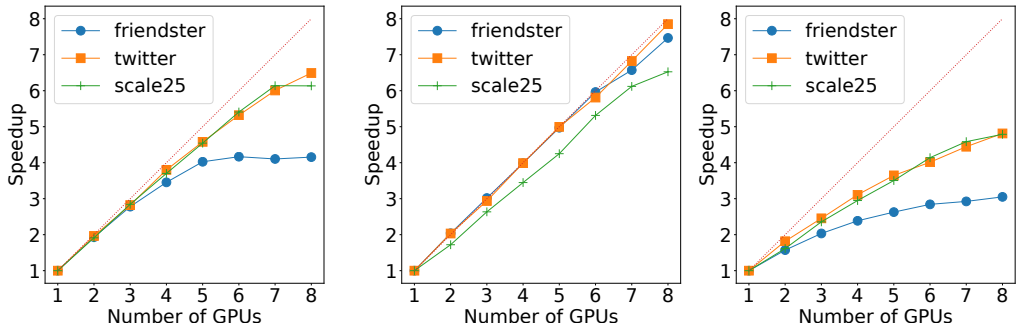


Fig. 4.1: Relative speedup between this work (on DGX architecture) and last year's submission. Black dashed line represents the baseline. Graphs are sorted on x-axis based on their number of edges.

strong scaling is affected badly. From Figure 4.2(a) we observe that, Stream algorithm scales very good up to 5 GPUs. However after 5 GPUs, scaling starts to get worse due to DGX machine's poor bandwidth (see Table 4.1).

Note that, friendster graph's strong scaling gets affected faster than twitter and scale25 graphs. Because, computation time of friendster graph's tasks are lighter than twitter and scale25. Hence, GPUs finish processing tasks faster and becomes idle until required tiles for the next task are copied.

4.4. Effect of bandwidth on speedup. Figure 4.3 presents strong scaling of the friendster graph up to 4 GPUs on two machines; with NVLink and without NVLink. We observe from this figure that with NVLink enabled architecture we get better scaling. Hence,



(a) Stream (S) (b) CopyFirst-Alg (C^+) (c) CopyFirst-Total (C)

Fig. 4.2: Strong scaling of the largest three graphs on GPUs.

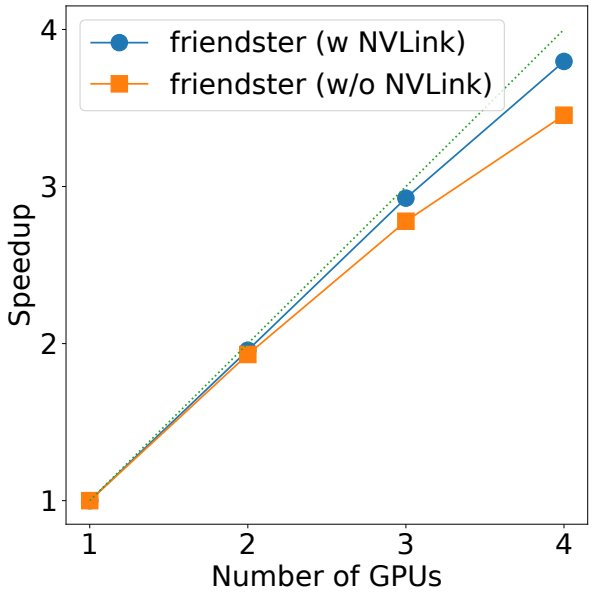


Fig. 4.3: Effect of bandwidth

we believe that if we would have a server with 8 Volta GPUs with NVLink, we could get even better execution times than the reported ones.

Table 4.2: Properties of the dataset. Best of the medians of the copy time included execution times in seconds and corresponding rates are reported. Green - instances that are more than 5× faster than last year submission. Blue - the fastest rate for a graph.

Data Set	$\|V\|$	$\|E\|$	$\|T\|$	**Raw Size**	**Tiles**	**Tasks**	**Best Time (s)**	**Rates ($\times 10^8$)**		
								DGX	**Newell**	**Minsky**
cit-HepTh	27,770	352,285	1,478,735	2.2 MB	64	120	0.002	1.7	0.9	0.9
email-EuAll	265,214	364,481	267,313	9.5 MB	64	120	0.002	1.9	1.1	0.9
soc-Epinions1	75,879	405,740	1,624,481	3.9 MB	64	120	0.002	2.1	0.9	1.0
cit-HepPh	34,546	420,877	1,276,868	2.7 MB	64	120	0.002	2.0	1.6	1.1
soc-Slashdot0811	77,360	469,180	551,724	5.4 MB	144	364	0.002	2.9	0.9	1.3
soc-Slashdot0902	82,168	504,230	602,592	5.7 MB	144	364	0.002	3.1	1.3	1.2
flickrEdges	105,938	2,316,948	107,987,357	14 MB	144	364	0.016	1.5	1.3	1.1
amazon0312	400,727	2,349,869	3,686,467	28 MB	144	364	0.006	3.9	3.4	3.1
amazon0505	410,236	2,439,437	3,951,063	29 MB	144	364	0.007	3.7	3.4	2.8
amazon0601	403,394	2,443,408	3,986,507	28 MB	144	364	0.006	4.4	4.7	3.3
scale18	174,147	3,800,348	82,287,285	26 MB	256	816	0.021	1.8	1.4	1.4
scale19	335,318	7,729,675	186,288,972	56 MB	400	1540	0.041	1.9	1.5	1.4
as-Skitter	1,696,415	11,095,298	28,769,868	146 MB	400	1540	0.027	4.1	3.4	3.2
scale20	645,820	15,680,861	419,349,784	110 MB	400	1540	0.079	2.0	1.7	1.5
cit-Patents	3,774,768	16,518,947	7,515,023	352 MB	400	1540	0.038	4.4	3.6	3.4
scale21	1,243,072	31,731,650	935,100,883	216 MB	400	1540	0.144	2.2	1.8	1.7
soc-LiveJournal1	4,847,571	42,851,237	285,730,264	534 MB	400	1540	0.121	3.6	3.3	2.7
scale22	2,393,285	64,097,004	2,067,392,370	464 MB	576	2600	0.325	2.0	1.7	1.5
scale23	4,606,314	129,250,705	4,549,133,002	915 MB	576	2600	0.549	2.4	1.5	1.1
scale24	8,860,450	260,261,843	9,936,161,560	1.9 GB	784	4060	1.154	2.3	1.2	0.8
scale25	17,043,780	523,467,448	21,575,375,802	4.0 GB	1024	5984	2.400	2.2	0.9	0.6
twitter	61,578,414	1,202,513,046	34,824,916,864	13 GB	1296	8436	4.582	2.6	1.1	1.0
friendster	65,608,366	1,806,067,135	4,173,724,142	16 GB	1296	8436	3.133	5.8	2.6	2.2
							Geomean:	**2.6**	**1.7**	**1.5**

5. Conclusions. We developed a fine-grained tasking based multi-core, multi-GPU, triangle counting method. This linear algebra implementation is up to 10.8× faster than our previous submission. This implementation results in the fastest end-to-end time known at time of publication due to very low overhead costs.

Acknowledgments: We thank Simon Hammond, Cynthia Phillips, and Stephen Olivier for helpful discussions.

REFERENCES

[1] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), (2011), p. 1.

[2] M. Deveci, C. Trott, and S. Rajamanickam, *Performance-portable sparse matrix-matrix multiplication for many-core architectures*, in Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2017, pp. 693–702.

[3] M. Deveci, C. Trott, and S. Rajamanickam, *Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures*, Parallel Computing, (2018), pp. 33–46.

[4] M. Grigni and F. Manne, *On the complexity of the generalized block distribution*, in International Workshop on Parallel Algorithms for Irregularly Structured Problems, Springer, 1996, pp. 319–326.

[5] Y. Hu, H. Liu, and H. H. Huang, *High-performance triangle counting on gpus*, in IEEE High Performance extreme Computing Conference (HPEC), IEEE, 2018, pp. 1–5.

[6] KokkosKernels.

[7] M. Latapy, *Main-memory triangle computations for very large (sparse (power-law)) graphs*, Theoretical Computer Science, (2008), pp. 458–473.

[8] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, 2017.

[9] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, et al., *GraphChallenge. org: Raising the Bar on Graph Analytic Performance*, arXiv preprint, (2018).

[10] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, *Static graph challenge: Subgraph isomorphism*, in IEEE High Performance extreme Computing Conference (HPEC), IEEE, 2017.

[11] G. Teodoro, T. D. R. Hartley, Ü. V. Çatalyürek, and R. Ferreira, *Run-time optimizations for replicated dataflows on heterogeneous environments*, in IEEE International Symposium on High Performance Distributed Computing (HPDC), 2010, pp. 13–24.

[12] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, *Fast linear algebra-based triangle counting with kokkoskernels*, in IEEE High Performance extreme Computing Conference (HPEC), IEEE, 2017, pp. 1–7.

[13] A. Yaşar, S. Rajamanickam, M. M. Wolf, J. W. Berry, and Ü. V. Çatalyürek, *Fast triangle counting using cilk*, in IEEE High Performance extreme Computing Conference (HPEC), IEEE, 2018, pp. 1–7.

VAGRANT AND ANSIBLE INTEGRATION

AMY N. WOODS* AND ELIJAH D. AGBAYANI†

Abstract. We explore the use of Vagrant and Ansible to set up the standardized virtual machine environment used by staff on a large software engineering project for development and testing. The current setup process of building a virtual machine from scratch is manual, tedious, and time consuming. This research explores the advantages and challenges of using Vagrant and Ansible individually and as a team. The Vagrant/Ansible setup is tested by different users, and is timed to evaluate the efficiency of the new setup procedure relative to the current procedure. The results demonstrate the Vagrant/Ansible setup process is faster and more user friendly.

1. Introduction. For our team a virtual machine is used as a standard environment to run the software project on a Red Hat Enterprise, Linux environment. To properly function as a development environment, the basic Red Hat installation requires additional setup through shell scripts and manual actions. Time spent on this process is overhead; therefore, it is important that it is quick and straightforward. Unfortunately, while people have improved the process over time, both in terms of decreased setup times and fewer interactions, the current setup still requires a substantial amount of time and leaves much room for user error. This sparked the need to explore different tools, such as Vagrant and Ansible, to further improve the setup method. Ansible was explored as a replacement to shell scripts and manual commands; Vagrant was explored as a tool that could complement Ansible by automating the creation of the virtual machine itself.

2. Background.

2.1. Past Issues with VM Environment Setup. Initially, the environment setup process was lengthy and highly manual, with setup instructions maintained on a set of wiki pages. While a person familiar with the process could complete it in one to two hours (assuming no problems), in practice newcomers to the project often found it took days to complete. The instructions were often out-of-date as important changes – for example, a change in the remote repository for software packages – would be made to existing environments but not to the wiki pages. Despite their best intentions, developers would often not revise and re-test the instructions in light of the change, as they would have already lost valuable time making and troubleshooting it on their existing environments. The instructions were also contradictory in some areas, as developers operating in heterogeneous host configurations (such as different hardware, software, and networking) would often encounter different problems in creating their VM environment.
The steps to create the virtual machine environment included:

- Creating the virtual machine itself
- Installing the Red Hat Enterprise Linux (RHEL) operating system
- Specifying remote repositories for system updates, software packages, etc.
- Configuring network proxies over HTTP and SSH
- Distributing credentials required for various services
- Installing required software applications and libraries
- Configuring environment variables
- Initializing background services
- Fetching code repositories

*University of California Davis Computer Science, anwoods@ucdavis.edu
†Sandia National Laboratories, eagbaya@sandia.gov

- Verifying the result by building and running the code

Many of the above steps, including the entire sequence of creating the virtual machine and installing RHEL, were done manually. Some parts of the process were automated with bash scripts. Unfortunately, these were prone to break, especially when the environment of the person running the scripts differed from that of the scripts' authors. The team also experimented with using golden images, which are discussed in the next section. While this approach has brought improvements to the process, there were still lingering problems.

2.2. Golden Image. A golden image is a template for a virtual machine such that can be cloned and used by other users. Using the golden image has advantages including limiting the number of opportunities for error that arise from manually building a virtual machine through command line entries. The golden image allows developers to avoid having to install the many of the required tools and applications. The main disadvantage inherent to the golden image is its static behavior. This means if a developer wishes to update their current golden image, it would require creating a completely separate golden image and reuploading it. The reuploading process would then, depending on the size of the image and networking conditions, take up to a full day to upload. For the case of changing or adding one line of code in the image, this is an inefficient use of time and discourages developers from regularly updating the golden image. In addition, the golden image could not cover the entire setup process, as certain aspects of the environment (e.g. credentials) must be customized for each developer; as a result, it did not eliminate the problems associated with manual setup. While the Vagrant and Ansible approach described later is not fully divorced from golden images – in particular, Vagrant boxes are close to golden images – it does mitigate these shortcomings.

2.3. Ansible. Ansible is an "IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration and many other IT needs"[1]. While shell scripts also focus on automation, Ansible differs from them in its use of idempotent modules. The concept of idempotency is that an operation can be repeated multiple times, with the same result as if the operation was only done once. Ansible's idempotent modules check the state of the machine to determine if any operations need to be performed to achieve a desired state; if the machine is already in that state, Ansible avoids changing it. Creating idempotent shell scripts is possible but it requires more code and concentration compared to Ansible, which has pre-made idempotent modules. Using these, one can create playbooks consisting of different tasks that automate the configuration management process. A playbook in Ansible is an idempotent script; unlike a shell script, which may have adverse and unexpected effects if executed multiple times, Ansible ensures that it is safe to execute playbooks more than once. Additionally, playbooks are easier to read and follow than shell scripts as the playbooks use the Jinja2 language and YAML syntax that is considered to be more human-readable. Ansible can help our goal of automating the setup of a virtual machine environment by converting configuration management shell scripts to quicker, and easier to read Ansible scripts.

2.4. Vagrant. Vagrant "is a tool for building and managing virtual machine environments" in a single setup process as much as possible [2]. It accomplishes this through configuration files and boxes. A box is a compressed file that contains a virtual machine environment that has been formatted for Vagrant and can be copied onto multiple machines. Vagrant instantiates an environment from the box using a Vagrantfile, a configuration file

which describes machine requirements when provisioning the virtual environment. As a result, Vagrant lowers the setup time of a virtual machine as well as the room for human error. When combined with Ansible playbooks, Vagrant can help make the setup process easier for users by distributing the box to their machines and enabling the configuration of the box in a timely, automated manner.

3. Ansible and Vagrant Integration.

3.1. Ansible Script Integration. The current process of the golden image still required setup tasks consisting of interactive shell scripts that prompted the user for input throughout the process. First, we converted those shell scripts into tasks that were embedded inside an Ansible playbook. Each task in Ansible is defined as a module with the required arguments and a name. They are individual actions that Ansible completes if the computer is not in the desired state. The conversion to the Jinja2 language and YAML syntax made the scripts easier to read and therefore gave a better understanding of what the tasks are doing. To further minimize the user interaction, an additional task was added to the beginning of the playbook to require the user to define most of the variables before the playbook executes. The variables can be assigned via command line entry; however, the playbook will inform the user if any variables were not defined in the command to prevent any errors. After the variable check, the playbook begins to complete its required tasks. The script was purposely designed to have most of the required user interaction take place at the beginning of the script. This provided clarity for user to know when their attention is required for the script to continue.

3.2. Vagrant Integration. After the Ansible scripts were integrated into the current golden image, we used VMware management tools commands to defragment and shrink the vmdk file from approximately 50 GB to 15 GB. All the necessary files are tarred into the .box format that Vagrant reads. The .box file is then imported to Vagrant through a command so that the box can be referenced in the configuration management file called Vagrantfile. The main jobs of the Vagrantfile are to define the box to create the virtual machine from, the type of provider, how to provision the machine, and any VM-specific settings such as networking and hardware configurations. Once the Vagrantfile and box are created, then the user only has to type “vagrant up” to instruct vagrant to begin creating and provisioning a new VM. For our research we used VMware as the provider and tool to run the virtual machine. During the ”vagrant up” command, Vagrant begins to clone the box requested by the Vagrantfile and runs the virtual machine on the provider, VMware. When the virtual machine is powered on, the user can run the Ansible script to provision the machine. The combination of using Vagrant and Ansible can be accomplished differently. Vagrant has provisioning capabilities allowing a user to provision the guest virtual machine by specifying an Ansible playbook within the Vagrantfile. Therefore, the “vagrant up” command would cause Vagrant to read the Vagrantfile and automatically execute the specified Ansible playbook on the guest machine. This alternative approach of Vagrant and Ansible was attempted but we ran into a ”ssh” error that is believed to be caused by proxy issues. Consequently, the Vagrant and Ansible setup process that will be used is the one where the user has to manually run the Ansible script on the guest machine.

4. Testing Vagrant/Ansible Setup Process.

4.1. Procedure. A Vagrant box and Vagrantfile were first created and distributed to three different operating systems via a network shared drive. The operating systems were

RHEL 7 Linux, macOS Mojave, and Windows 10. On each operating system, two different users, totaling of six different users, followed instructions from a wiki page to create a virtual machine starting with the Vagrant box. The timer started with the user adding the "golden" Vagrant box to Vagrant and then running the command "vagrant up" to create the virtual machine. After the machine was created, the user followed an interactive Ansible script previously baked into the "golden" Vagrant box to provision the virtual machine. Provisioning the VM included: creating an ssh key, linking maven to artifactory to enable a Java build, configuring git, and cloning repositories and directories. The timer officially stopped once the Ansible script was completed.

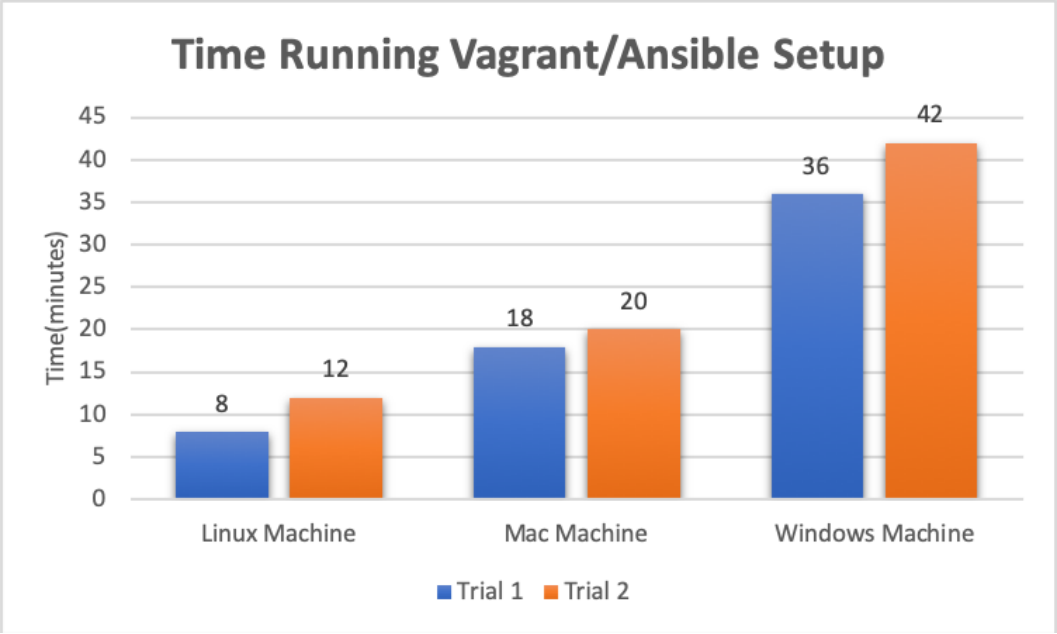


Fig. 4.1: A bar graph depicting how long it took to run the Vagrant/Ansible setup process on a specific operating system. Each bar depicts a different user, total six users.

4.2. Results. The results initially were surprising as there was a drastic range of time it took to complete the setup process on different machines. This required further investigation to understand the variability of the results. Observations during the trials indicated that most of the variation occurred during steps which were disk-intensive, such as "vagrant up". Therefore, a disk performance test was run on each of the machines using the command:

time sh -c "dd if=/dev/zero of=/tmp/testfile bs=10000k count=1k && sync"

This test used the dd (disk dump) command which can be used to simulate large disk write operations. The arguments after dd provide a test which writes 10GB into /tmp/testfile; in conjunction with the time command, this produces a figure for disk speed. The results demonstrated that the Linux machine's disk performed at 1700 MB/s while the Windows machine's disk performed at 155 MB/s - an order of magnitude discrepancy. The "vagrant up" command clones the Vagrant box, meaning the command requires reading and writing of about 15 GB of data. If the disk speed is slow then it will take the Vagrant command longer to complete reading and writing from the Vagrant box. The slow disk speed of the Windows machine provided an explanation of why it took longer to complete the setup

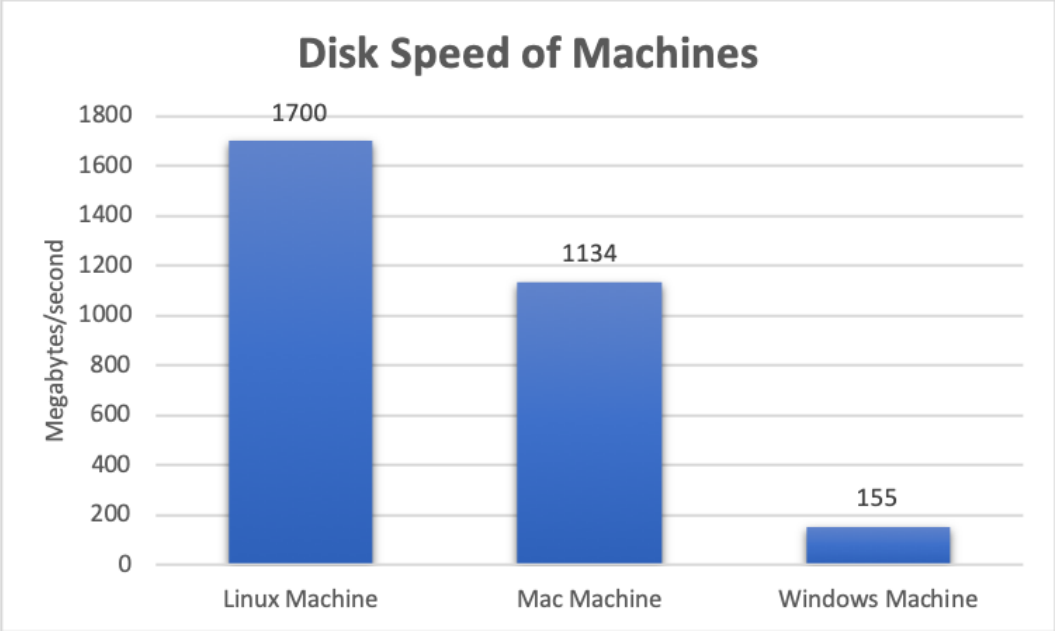


Fig. 4.2: A bar graph depicting the disk speed of the machine with the specific operating system.

process compared to the Mac and Linux machines [1]. These results demonstrate the relationship: the faster the disk speed of the machine, the faster the setup process will be completed.

Based on the Fig 4.1, the average time it takes to complete the setup process from all three operating systems is 22.67 minutes. A part of the setup process skipped during the trials included installing Vagrant, VMware, and the VMware plugin for Vagrant. This installation process will vary per machine as it also relies on the disk speed. The installation steps were also skipped because it only has to occur once, while the development environment might be provisioned multiple times. When installing the required software before the trials began on the Mac machine, the estimated completion time was about 1 hour. This means on average the total setup process would take about one hour and 23 minutes. According to our team lead the original setup process took about 4 hours at best and could even take multiple days at worst. When comparing to this time, the Vagrant/Ansible process is approximately three times as fast as the original setup.

5. Discussion.

5.1. Golden Image versus Vagrant Box. Both the golden image and the Vagrant box act as a template for a virtual machine, having essential applications prebaked into the machine. In both setup processes the virtual machine still requires provisioning which is done by the shell scripts or the Ansible script. The main difference that exists between these two processes is the behavior of the golden image versus the Vagrant box. The golden image has a static behavior, i.e. it is difficult to change its contents without creating another golden image. Initially, the Vagrant box has the same static behavior as the golden image; however

[1] Fig 4.2 does not represent the disk speed of all Linux, Mac, or Windows machines

it is Vagrant's provisioning capabilities that adds a dynamic behavior to the Vagrant box. During the "vagrant up" command, , Vagrant will create the machine according to the configuration in the Vagrantfile, which can be easily edited as a text file. Furthermore, if the Vagrantfile includes a provisioning Ansible script then Vagrant will automatically provision the virtual machine according to that script. For our project specifically, Vagrant could automatically clone the most recent repositories and directories onto the VM. This automatic provisioning allows the Vagrant box to be dynamic as the Ansible script can keep changing and updating with the most recent repositories and directories - without having to recreate the box each time.

5.2. Editing a Vagrant Box. Another key disadvantage that came with using a golden image is the amount of time it takes to edit and reupload the golden image. Consequently, the editing and repackaging of the Vagrant box were tested. Vagrant provides a command that takes the currently running virtual machine and packages the state of the machine into a different box file. This means if a user wanted to add a folder to the "golden box", the user would create the folder in his current machine and then run the vagrant command:

vagrant package –output golden-box_vX.Y.box

to create a new box named "golden-box_vX.Y". Changing and uploading the golden image was a tedious process that could take 19 hours to package. Based on 10 trials of running the command with one new directory in the virtual machine, the average time to repackage the box was 40 minutes and 29 seconds. These trials were run on the Mac machine for the purpose of using a medium disk speed. This vagrant command makes editing and reuploading the Vagrant box 28 times faster than the original method. Overall, the Vagrant box has better tools than a golden image to allow easier editing and provisioning capabilities.

6. Future Steps. Further steps to improve the Vagrant/Ansible setup include editing the Ansible script to further limit required user interaction. One part of the Ansible script includes copying and pasting from a website to a terminal and vice versa. This part, during the trials, was observed to be the part that slowed users down the most. It required them to read the directions which tend to be skipped. A way to improve the Ansible script is to minimize both the user interaction and the directions the user has to read. Another way to increase the speed of the setup would be to have Vagrant automatically provision the Ansible script during the "vagrant up" command. This was attempted but was blocked by a complex Vagrant ssh error. Despite venturing through numerous command logs it is difficult to determine what is causing this connection error. If this error gets fixed then the automatic provisioning would eliminate the user having to copy and paste the command to run the Ansible script manually. Additionally, the provisioning uses a standard entity account baked into the Vagrant box; the name of this account is not the same as a user's domain ID, which requires additional configuration to specify this ID during certain operations. The Ansible scripts could be edited to provision a local account for a user that is aligned with their domain ID; doing so would simplify the existing setup and avoid potential issues in the future.

7. Conclusions. Based on the results, using Vagrant commands to create a virtual machine from a "golden" box file with Ansible scripts baked in is approximately three times as fast as the original setup process. Running Vagrant and the Ansible script is reliant on the disk speed of the machine. Consequently, if the user wants the fastest setup time then

they should acquire a machine with fast disk speed. Despite the varying times from the different operating systems, the results prove that the new setup process can successfully be implemented on several operating systems. The Ansible script was coded so that most of the user interaction takes place at the beginning of the script. Having the user define the variables in the command line eliminates unnecessary pausing in the script to ask for user input. Similarly, the use of Vagrant limits user interaction to only three commands: vagrant box add, vagrant up, and ansible playbook. After using Vagrant, the Ansible script requires limited user interaction; therefore, minimizing the number of times user error may occur. Overall, the combination of Vagrant and Ansible resulted in a more expedient setup process that sufficiently limited user interaction and user error.

REFERENCES

[1] R. Ansible, *How Ansible Works*. https://www.ansible.com/overview/how-ansible-works, 2019 (accessed August 30, 2019).

[2] HashiCorp, *Introduction to Vagrant*. https://www.vagrantup.com/intro/index.html, 2019 (accessed August 30, 2019).

AN ANALYSIS OF SCALABLE DATABASE DESIGN TECHNIQUES FOR MODULE BASED WEB APPLICATIONS

ARTHUR K. ZHANG† AND MATTHEW WONG‡

Abstract. Designing scalable and maintainable databases is a common software challenge with web applications. Traditionally, only backend web developers needed to worry about designing efficient databases for managing user data. As such, using memory efficient storage techniques like relational databases with multiple joins became the standard in database design. However, the increased processing power of personal computers rapidly led to the development of complicated web-based user interfaces with complex workflows. This increases the difficulty of managing user interface states and challenges trusted database designs to meet the demands of modern browsers. This paper presents our ideas on how to implement efficient frontend state management and improvements upon backend database design principles to better suit modern web design. We demonstrate that these new implementations enable create, read, and update operations in constant time without sacrificing the ability to perform atomic operations. This improves upon previous limitations with database design that force users to decide between having bidirectional references or the ability to perform atomic updates when working with denormalized data. Furthermore, we demonstrate our design's robustness and scalability by analyzing its efficacy on DMAMC CharCat (Data Mining Analysis and Modeling Cell, Characterization Catalog), a web-based instrument response catalog for CWMD (Countering Weapons of Mass Destruction) designed from the ground up with these principles in mind.

1. Introduction. This paper considers the implementation of database design from the point of view of CharCat, an emerging web application where a preexisting structure has not been defined yet. CharCat is a web-based nuclear detector characterization catalog designed for CWMD, and by extension DHS (Department of Homeland Security). Its primary purpose is to store and display instrument response data collected by scientists. This reduces the amount of duplicated response tests conducted using instruments that are already in the catalog, which saves scientists time for more important work. Researchers are able to quickly find these instruments on the catalog because each instrument is assigned a unique serial number reference. Early web applications, such as CharCat, rapidly evolve and often take on unforeseen features, both from the programmers' and users' perspective. This presents several notable challenges to the database architect. When dealing with complex web applications, a key aspect of both frontend and backend database design is composability, a system design principle that deals with the inter-relationships of components. Modern web development practices often revolve around the concept of component based application design. Component based design is the practice of splitting the user interface (UI) into smaller, more manageable parts with clear names [1]. There are three groups of user interfaces that must be managed by the database: pages, the actual screens of the application; compositions, the containers within the pages that contain components; and components, blocks on the screen that contain elements such as graphs, plots, and buttons. We show an example of these UI modules on CharCat in Figs. 1.1, 1.2, and 1.3. Instrument response data is blurred for privacy purposes.

As UI workflows become more complex, we find that it is necessary to properly design a DBMS (Database Management System) for storing persistent application data as well as databases that track the current state of the user's session as well. Below, we will discuss at a high level the different challenges frontend and backend database architects may encounter while working on a component based application.

†University of Michigan, Computer Engineering Undergraduate, arthurzh@umich.edu
‡Sandia National Laboratories, mhwong@sandia.gov

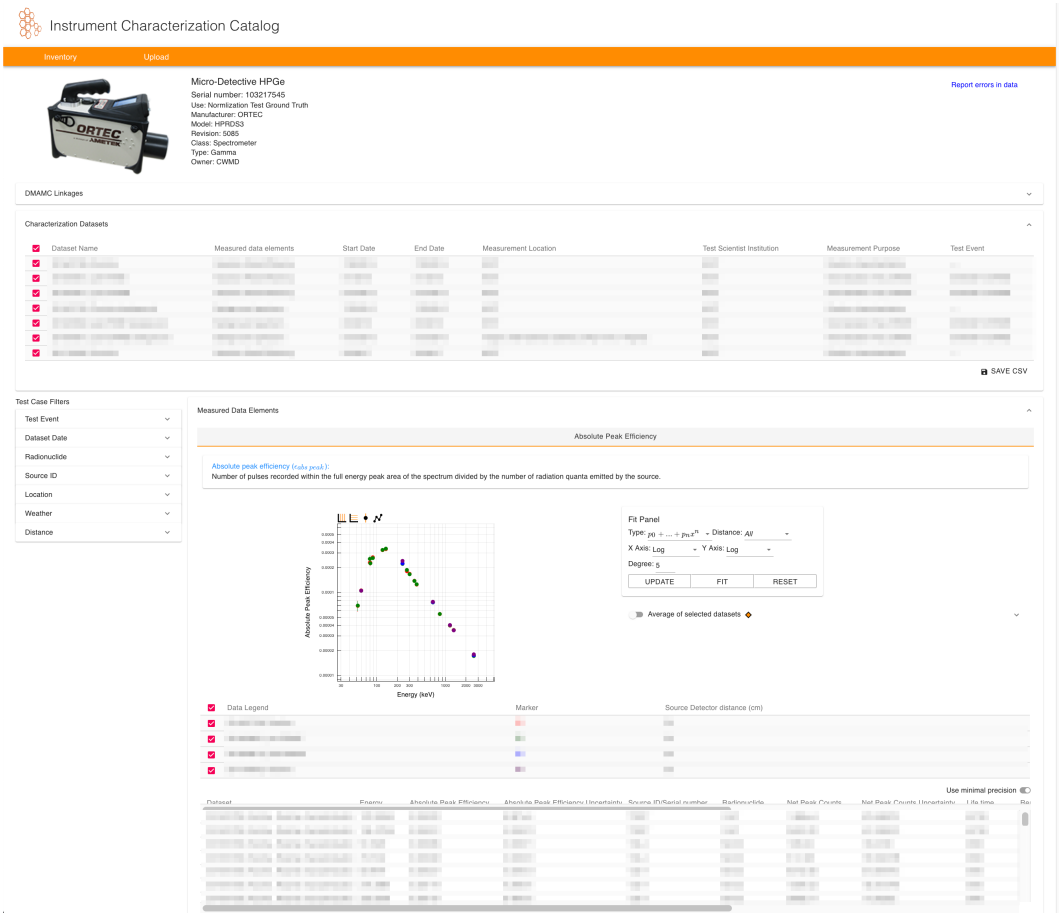


Fig. 1.1: Page module

Fig. 1.2: Container module

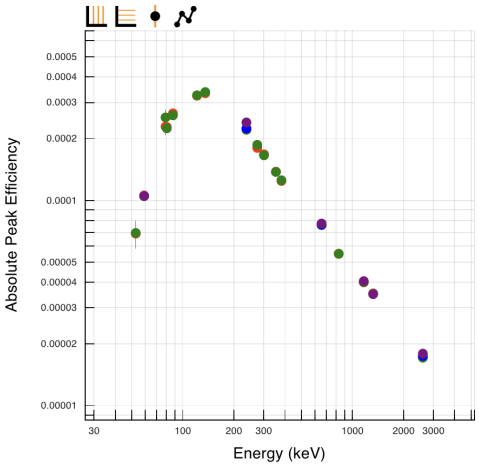


Fig. 1.3: Component module

2. Background.

2.1. Frontend DBMS Requirements. Component based design for scientific purposes presents an interesting problem to frontend database architects by making it difficult to define a set schema for the application. New components may need to be added at any time while existing components may need to be adjusted to account for changes in the application. In addition, existing data for these components will not only be constantly queried, but also constantly modified as well. Along with tracking individual states for each of the components, our front end database must somehow manage a global state for each page the user interacts with. We would also like our new database to address several shortcomings of existing database management systems (DBMS). More specifically, our database should be highly extensible, able to adapt to eventual schemas of scientific investigations; highly efficient, able to process exponentially increasing datasets in sub-exponential time; and intuitive to non-backend engineers, such that frontend programmer could easily modify the existing data store. In past years, computer scientists have come up with architectures such as Flux[1], an architecture pattern for managing client-side application data to address these issues. However, we have found that Redux, a library for managing client-side data stores, is more suitable for database design because it gives architects the ability to strictly enforce standard principles such as immutability of data objects and atomic updates. This is important for creating a database that is safe from race conditions, unintended behavior that is dependent on the sequence or timing of uncontrollable events. A summary of the key distinctions between Flux and Redux can be found in figure 2.1.

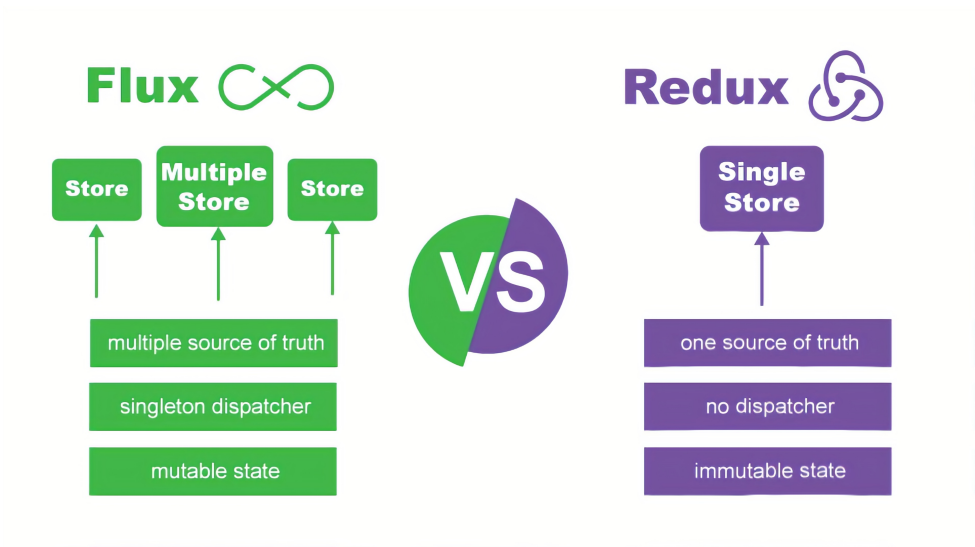


Fig. 2.1: High level feature comparison between Flux and Redux [4]

2.2. Backend DBMS Requirements. Traditionally, there are two distinct database styles used in industry: relational and non-relational databases. Relational databases store data in tables and rows and allow programmers to link information from different tables

[1] https://facebook.github.io/flux/

through foreign keys, which is useful for processing complicated queries and database transactions. Relational databases follow ACID (Atomicity, Consistency, Isolation, Durability), which are a set of properties for database transactions intended to guarantee validity of operations in the event of unforeseen events: power failures, errors, etc [7]. Non-relational databases simply store data without explicit mechanisms to link data from different tables to one another [2]. Non-relational databases do not strictly follow the ACID protocol by design. Despite the consistent behavior that relational databases provide, they are not well suited for storing records in the same collection with different fields, de-normalized database schemas, and a dynamic schema as a result of the data model. This is applicable to early stage scientific web applications because scientists and programmers are unable to predict exactly what is needed due to natural volatility of data in research. In addition, experimental data from scientific research, when modeled, is often deeply nested, resulting in unnecessarily bulky join operations and is more naturally suited to a de-normalized data structure. Given these reasons, we believe that non-relational databases are preferable to use for early stage application development. For our implementation, we have chosen to use MongoDB, a non-relational database, in conjunction with Mongoose to assist in delivering complex queries. To guarantee the validity of transaction operations, a common issue with non-relational databases, we have developed several database rules that enforce the CAP theorem (Consistency, Availability, Partition tolerance), a similar set of principles for non-relational databases [3]. A summary of key distinctions between relational and non-relational databases can be found in figure 2.1.

Table 2.1: High level feature comparison between relational and non-relational databases

	Relational	**Non-Relational**
Data	Structured data in tables	Unstructured data in JSON
Schema	Static	Dynamic
Scalability	Vertical	Horizontal
Queries	joins for complex queries	Library prepares complex queries
Flexible	rigid schema	Non-rigid and flexible schema
Transaction	ACID	CAP theorem

3. Methods.

3.1. Redux DBMS Implementation. Our Redux store is denormalized and avoids deep nesting, allowing us to update specific UI components as slices in the database. Each UI component is initially unknown to Redux, and is only created once the user has modified the component. This minimizes the memory required to initially load and operate CharCat. When each component is used, the client calls a unique "Action", which is a function that refers to a unique set of instructions on how to appropriately modify the component in the Redux store. To modify the appropriate UI component in the store, the client generates a "Context Driven Component ID" for the modified component. This generated ID is a period separated string that uses the location of the component as tags. For instance, in our application, we have created entities known as instruments, each of which have been assigned unique serial ID. Each of these instruments display different data types, and each of these data types contain different UI components. If two of the same UI components are found in the same data type, they are further differentiated by a designator number. With all of these identifiers working together, we are able to piece together a unique ID for each component that can also be used to determine where the component is modified in the context of the application. The ability to determine the parent container for any given

object in the database using its ID is especially useful for updating individual components efficiently. Instead of iterating through a deeply nested structure, we can simply perform an efficient $O(1)$ lookup for the component in a slice of the database and update the slice instead, another $O(1)$ operation. This also makes adding new components to the Redux store systematic. To track new components, the user simply needs to define the default schema to initialize the component with on creation and write the component's create/update function. The task of generating context driven IDs and efficiently updating the component is automatically done by our implementation of Redux. In addition, this makes the process of querying between components intuitive for even non-computer scientists. If component A needed to access the contents of component B, component A could simply use context information like which instrument and data type component B belongs in to do so. An overview of the Redux store structure is shown below in figure 3.1.

Despite storing denormalized data, we retain the ability to atomically update the database by enforcing numerous design rules. One such rule is that users cannot reassign parent and child ID links between one to many and many to one relationships between components. This prevents the database from having to update both the child and parent references to each other, which is a non-atomic transaction. The key benefit in enforcing these rules is that we maintain the advantages of denormalized data: efficient lookups and infrequent joins; while avoiding its flaws: non-atomic updates and creations. However, these advantages come at the cost of storing duplicate data and additional documentation. In addition to optimizing CRUD (Create, Read, Update, Delete) operations with Redux, we strictly use immutable objects in the Redux store and rely on pure functions for mutating the Redux state. By using immutable objects, we allow sophisticated change detection techniques to be implemented easily, which ensures that computationally expensive operations like updating the DOM are only performed when absolutely necessary [6]. The rationale behind mutating the state with pure functions stems from the fact that multiple React components may attempt to edit the same datum asynchronously. With a shared state and multiple parallel processes running, this quickly becomes a race between the processes that results in nondeterministic behavior. By forcing programmers to update the redux state using pure functions, we guarantee that given the same input, our functions always return the same output without any side effects. Removing side effects is essential because in Javascript, all non-primitive objects are passed into functions as references. If our function directly mutates a property on an object, the object changes outside the function as well. Therefore, the only way to know the full effect a function has on an object is by knowing the full history of the object that is passed in. This produces non-deterministic behavior, which is difficult to debug, especially with applications with complex logic and state management. However, by following these design rules, we reduce the possibility for race conditions in our application and ensure fully deterministic behavior.

3.2. MongoDB DBMS Implementation. The MongoDB is quite different from the Redux store in that it has no notion of a user interface, rather it stores data based on how it is organized by test scientists. Experimental data is first organized by which instrument it belongs to, the dataset number, the data element type, and then the entry number. This structure is shown in figure 3.2 for clarity.

Each object type uses a different schema that represents its contents. In addition, if an object is an immediate parent to a child, it contains a link to the child by using the child's ID. However, the child is unaware of the parent and does not link back. This lack

Screen types referred to as entities

UI components grouped together

```
{
    entities: {
        instruments: {
            byId: {
                "serialNumber1": {
                    id: "serialNumber1",
                    dataElements: {
                        dataElementName: ["uiElementId1", "uiElementId2"],
                        // data elements will be added as new unique components are rendered in ui
                    },
                    selectedDataElement: dataElementId2,
                    selectedDatasetIds: [datasetId1, datasetId2 ... ]
                }
            },
            allIds: ["serialNumber1", "serialNumber2"]
        }
    },
    ui: {
        datasetLegends: {
            byId: {
                "datasetLegendId1": {
                    id: "datasetLegendId1",
                    colorMap: {
                        datasetId1: "Red",
                        datasetId2: "Blue,
                        ...
                    },
                    selectedDatasets: ["datasetId1", "datasetId2"],
                    shapeMap: {
                        "radionuclide1": "Circle",
                        "radionuclide2": "Square",
                        ...
                    },
                    rads: ["radionuclide1", "radionuclide2"],
                    hasBeenSet: true
                }
            },
            ...
            allIds: ["datasetLegendId1", "datasetLegendId2"]
        },
        fits: {
            byId: {
                "fitId1": {
                    dataElementId: "dataElementId1",
                    fitting: {
                        id: "dataElementId",
                        instrumentId: "instrumentId1",
                        type: "polynomial",
                        degrees: "2",
                        coeffs: ["2", "4", "1"],
                        xScale: "linear",
                        yScale: "linear",
                        distance: "50",
                        points: [{"x1", "y1"}, {"x2", "y2"}, ...],
                        stdErrs: [{"x1Err", "y1Err"}, {"x2Err", "y2Err"}],
                        changed: true
                    }
                },
            }
            ...
            allIds: ["fitId1", "fitId2", ...]
        },
        testFilters: {
            byId: {
                "testFilterId1": {
                    activeFilters: {
                        "Distance": ["50", "100", ...],
                        "Radionuclide": ["radionuclide1", "radionuclide2", ...],
                        "Source ID": ["sourceID1", "sourceID2", "souceID3" ...],
                        "Test Event": ["TestEvent1", ...],
                        ...
                    }
                },
                ...
                allIds: ["testFilterId1", "testFilterId2", ...]
            }
        }
    }
  }
}
```

Fig. 3.1: Redux store structure

of bidirectional referencing between parent and children is especially important to maintaining atomic consistency in denormalized data stores [5]. If a child component needs to switch parents, this operation can be conducted in a single update because only the parent collection needs to be modified. However, if the child referenced the parent ID as well, then we would not be able to guarantee this transaction's validity because ownership transfers

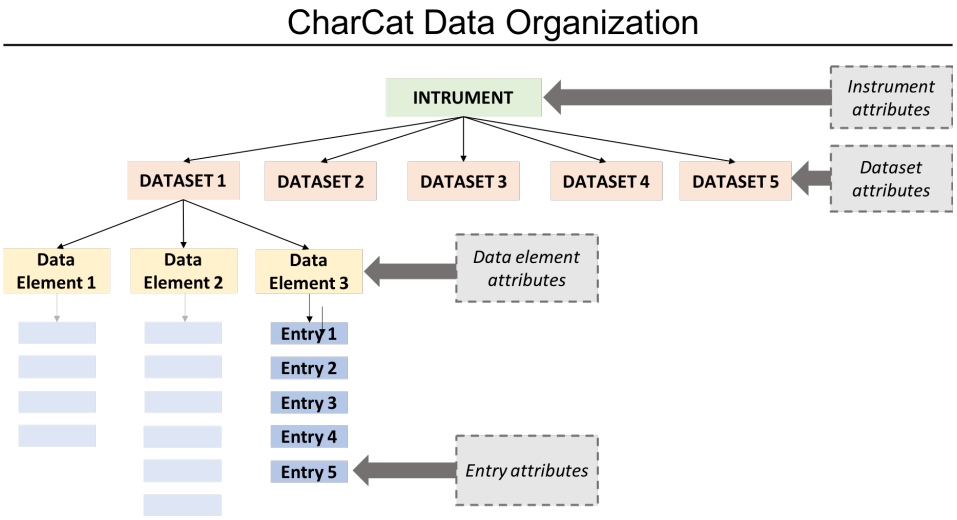


Fig. 3.2: MongoDB organization structure

require two operations. An example of this parent child relation is shown in figure 3.3 for the Dataset object. Each Dataset contains an array of data elements, and each of these data elements own data entries. The data elements reference the data entries by ID unidirectionally. We chose to use Mongoose, a schema-based solution to model application data. This library comes with built in type casting, validation, and advanced query building features. These features help guarantee our database's adherence to the CAP theorem. Mongoose validates all database transactions to ensure consistency, simplifies response handling for high availability, abstracts database sharding to maintain partition tolerance, and offers an efficient solution to process complex queries to MongoDB, an essential functionality of any complete DBMS solution. In addition, Mongoose also supports several atomic update transactions for MongoDB collections that are not deeply nested. This works in tandem with our denormalized database to ensure the validity of database updates.

```
/**
 * Dataset schema
 */
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const datasetSchema = new Schema({
  serialNumber: { type: 'String' },
  "Dataset Number": {type: 'Number'},   Unique Identifiers
  name: { type: 'String' },
  testStartDate: { type: 'Date' },
  testEndDate: { type: 'Date' },
  measurementStartDate: { type: 'Date' },
  measurementEndDate: { type: 'Date' },

  testScientistInstitution: { type: 'String' },
  measurementPurpose: { type: 'String' },
  testCampaign: { type: 'String' },

  rndrURL: { type: 'String' },
  raasURL: { type: 'String' },

  coolerRunHours: { type: 'Number' },  // HPGe only

  measurementLocation: { type: 'String' },
  measuredDataElements: { type: 'String' },
  elements: [{
    name: { type: 'String' },
    description: { type: 'String' },
    elementName: { type: 'String' },
    entries: [{ type: Schema.Types.ObjectId, ref: 'Entry' }],   ID Links to children components

    fullTable: [{ type: 'String' }],
    displayTable: [{ type: 'String' }],
    x: { type: 'String' },
    xError: { type: 'String' },
    y: { type: 'String'},
    yError: { type: 'String' },
    y2: { type: 'String'},
    y2Error: { type: 'String' },
    searchFields: [{ type: 'String' }],
    "Element Name": { type: 'String' },
  }],


});

export default mongoose.model('Dataset', datasetSchema);
```

Fig. 3.3: MongoDB Dataset Collection Code Snippet

4. Conclusion. We developed two database implementations that handle commonly encountered issues in modern web development. Both database implementations, Redux and MongoDB, have required little to no maintenance once set up and efficiently scaled to meet the needs of CharCat as more datasets and instruments were added. Redux's usage of context driven user interface IDs is well suited for UI state tracking for components because it only tracks components as needed, which guarantees the same initial application performance speed regardless of the number of datasets loaded. In addition, any update transactions to Redux only update a small slice of the database, ensuring that operations remain efficient. In fact, a typical Redux transaction requires 70 milliseconds for completion on average with slice updates, while full updates require 300 milliseconds. The overall

Redux structure required zero significant refactors during the development cycle of CharCat once defined, demonstrating its durability towards user interface refactors. The usage of non-relational, denormalized data stores for both Redux and MongoDB resulted in less bulky joins and improved the efficiency of query lookups and updates to $O(1)$ speed while maintaining ACID principles. This reduces the likelihood of race conditions at the cost of additional memory used for storing duplicate data. In fact, zero race conditions were encountered during database transactions with MongoDB during the lifetime of the app. In the future, we plan to explore novel methods for further improving the efficiency of high volume database transactions to MongoDB and reducing the amount of duplicate information stored for both Redux and MongoDB.

REFERENCES

[1] M. Bakaev and V. Khvorostov, *Component-based engineering of web user interface designs for evolutionary optimization*, IEEE, 1 (2018).

[2] V. N. Gudivada, D. Rao, and V. V. Raghavan, *Nosql systems for big data management*, 2014 IEEE World congress on services, 1 (2014), pp. 190–197.

[3] C. Győrödi and R. Gyorodi, *A comparative study of relational and non-relational database models in a web-based application*, International Journal of Advanced Computer Science and Applications, 1 (2015).

[4] D. Lillie, *Flux vs. redux: A comparison*, 2017. [Online; accessed July 24, 2019].

[5] I. Mapanga and P. Kadebu, *Database management systems: A nosql analysis*, International journal of Modern Communication Technologies and Research, 1 (2013).

[6] L. Mezzalira, *Front-End Reactive Architectures*, vol. 1, Springer, 2018.

[7] Y. Zhou, Z. Gennert, N. I. Hachem, and M. O. Ward, *Requirements of a database management system for global change studies*, Proc. ASPRS /ACSM '92—Mapping and Monitoring Global Change, (1992), pp. 186–194.

PARALLEL GRAPH COLORING USING MPI AND KOKKOS

IAN BOGLE[†], ERIK G. BOMAN[‡], KAREN DEVINE[§], SIVASANKARAN RAJAMANICKAM[¶], AND GEORGE M. SLOTA[‖]

Abstract. Graph coloring can be used as a preprocessing step used to parallelize scientific computations, many of which happen in a distributed, multi-GPU environment. Many algorithms exist for graph coloring that run on single GPUs, or in distributed memory, but hybrid MPI+GPU algorithms are less common. We present an implementation of Gebremedhin et. al's [3] distributed algorithm that uses an implementation of Deveci et. al's [7] parallel coloring for on-node parallelism. The on-node parallel coloring is implemented in Kokkos Kernels [8], meaning that we have an MPI+X coloring algorithm. Additionally we propose a new algorithm that aims to reduce the total amount of communication involved in other distributed distance-1 coloring algorithms. We examine the effects of different prepartitioning methods on our algorithms' runtimes, as well as the number of colors used in the graph coloring.

1. Introduction. Graph coloring is a graph problem where colors are assigned to vertices in such a way that no neighboring vertices have the same color. There are many useful applications of graph coloring, but it is typically used to find concurrency in computations [7] [1], and printed circuit testing [9]. Additionally, graph coloring is used as a preprocessing step to speed up the computation of Jacobian and Hessian matrices [11]. The problem of minimizing the number of colors in a graph coloring is NP-hard, but the applications that use coloring as a preprocessing step benefit from colorings that use fewer colors. Deveci et. al. [7] show that a smaller number of colors used by a coloring-based preconditioner improves the runtime of a conjugate gradient solver by 33%.

We present and examine two hybrid MPI+X implementations of two distance-1 coloring algorithms. Distance-1 colorings are an instance of graph coloring where neighbors are defined as vertices that are connected with an edge. We focus on hybrid implementations because most scientific computing applications run on distributed-memory systems, and it may not be feasible to assemble the associated graphs on a single node and run a sequential coloring algorithm [3]. In order to apply to the widest variety of architectures, we use the Kokkos parallelism framework [8] for on-node parallelism. The combination of Kokkos and MPI allows us to use either OpenMP or Cuda over multiple compute nodes in a system.

Our contribution is two MPI+X algorithms that are able to run in both CPU and GPU-equipped systems. For Multi-GPU runs, we achieve a 3.6x speedup, while the number of colors used increases by about 24% in the worst case.

2. Related Work. There are many variations of the graph coloring problem. Distance-1 graph coloring is the typical instance. Distance-2 coloring is a variant where each vertex v must have a different color than any vertex at most two edges away from v. Partial Distance-2 coloring is a special case of Distance-2 coloring on a bipartite graph in which only one set of the vertices gets colored.

For graph coloring in general, minimizing the number of colors is NP-hard, but serial algorithms based on greedy heuristics are effective for a number of applications [10]. The serial greedy algorithm colors vertices one at a time, using a heuristic to control the order in which the vertices get colored. Generally, colors are represented as numbers, and the smallest usable number is used as a vertex's color. Conflicts in a coloring are edges where

[†]Rensselaer Polytechnic Institute, boglei@rpi.edu
[‡]Sandia National Laboratories, egboman@sandia.gov
[§]Sandia National Laboratories,kddevin@sandia.gov
[¶]Sandia National Laboratories, srajama@sandia.gov
[‖]Rensselaer Polytechnic Institute, slotag@rpi.edu

both endpoints share the same color, and colorings that contain conflicts are not valid colorings, or "pseudo-colorings". A Coloring's "quality" refers to the number of colors used; colorings with higher quality use fewer colors, while lower quality colorings use more colors.

In a distributed-memory setting, graphs get split into subgraphs and assigned to separate processes. In order for processes to communicate values associated with their local vertices, there must be a consistent global identifier (GID) for each vertex. The local identifiers (LIDs) that each process uses for vertices is likely different from these GIDs, so each process must have a way of mapping their LIDs to GIDs.

There are two popular approaches to parallel graph coloring. The first is to concurrently find independent sets of vertices, then concurrently color all of the vertices in each set found. The second, referred to as "speculate and iterate", is to color as many vertices as possible in parallel, and then fix the conflicts in the resultant pseudo-coloring iteratively until no conflicts remain. Jones and Plassmann [15] propose a parallel coloring algorithm based on the independent set approach. Çatalyürek et. al. [5] and Rokos et. al. [17] present shared-memory implementations based on the "speculate and iterate" approach. Additionally, distributed-memory and hybrid algorithms such as those proposed in [3, 12, 18] largely use the "speculate and iterate" approach, and Bozdağ et. al. showed that in distributed memory this approach is more scalable.

Our approach builds on the framework presented by Bozdağ et. al. Their framework groups vertices on each process into a group of vertices that do not neighbor any vertices on another process, called "interior vertices", and vertices that do neighbor vertices on another process, called "boundary vertices". Each process's set of interior vertices can then be colored independently, without creating any conflicts, and without requiring any communication. Boundary vertices are colored in rounds in order to reduce the chance of conflicts occurring, which in turn reduces the amount of communication necessary to color the boundary vertices.

In this framework, we are free to color the interior vertices in any way, and we use an algorithm proposed by Deveci et. al. [7] for a number of reasons. Their algorithms are implemented in KokkosKernels [8], which allows them to run efficiently using either OpenMP or Cuda without requiring any changes to the implementation. Additionally, Deveci et. al. propose and implement several optimizations to traditional vertex-based colorings, and their most optimized version is called VB-BIT. They also note that approaches that iterate over vertices and check neighbors' colors have inherent load-balance issues due to the fact that in real graphs vertex degrees vary widely. To counter this load balance, they propose an edge-based algorithm that they show is more efficient than VB-BIT on the GPU.

3. Methods. We propose two MPI+X implementations that build off of the distributed framework presented by Bozdağ et. al. and the parallel coloring algorithms presented by Deveci et. al. There are several coloring algorithms implemented in KokkosKernels, but we only use the algorithms proposed by Deveci et. al, namely the vertex-based (VB-BIT) and the edge-based (EB) approaches. VB-BIT is an optimized vertex-based coloring that can be run on GPUs. VB-BIT's implementation in KokkosKernels can finish incomplete colorings, as it assumes that a vertex with a color of zero is uncolored. EB is an edge-based algorithm that is more efficient than VB-BIT on GPUs. Since the algorithms proposed by Deveci et. al. are implemented in KokkosKernels, they are able to run with OpenMP and on GPUs without modification. Algorithm 1 shows our general approach.

Because our algorithms run in distributed memory, each process has a subgraph of the original input graph. Typically, we refer to this subgraph as a process's "local graph", and the vertices as "local vertices". A process is said to "own" its local vertices.

Our approach initially colors the entire local graph on each process independently using

Algorithm 1 Overview of our approach

```
procedure HYBRID-COLOR(Graph G=(V,E), rand)
    Color all local vertices with KokkosKernels
    Communicate colors of owned vertices to ghost copies
    Detect global conflicts
    while Global Conflicts Exist do
        global conflicts = Resolve-Conflicts(G, colors, rand)
```

the VB-BIT for CPU platforms and EB for GPU platforms. Then, as Bozdağ et. al. point out, only boundary vertices can be in conflict with one another. Similarly to Bozdağ's approach, we initially tried to reorder the order of the vertices in the graph so that we could easily access the boundary vertices without having to also loop through interior vertices. This reordering is done by changing the underlying structure of the graph representation to group all of the boundary vertices together. This optimization did not pay off, as this reordering procedure was an order of magnitude slower than the KokkosKernels coloring of the entire local graph. Because of this, we decided not to reorder the graph, and simply use KokkosKernels' coloring to fix any distributed conflicts.

After the local vertices are colored on each process, we communicate this new coloring information to vertex copies on other processes using the Trilinos library [13]. We used the FEMultiVector class of the Tpetra package [14] to communicate the colors of locally-owned vertices to their copies on other processes, also known as "ghosts". After each process gets this coloring information, it detects conflicts by checking every owned vertex's color against the color of each of its neighbors. This conflict-finding process is trivially parallelizable, and we parallelize it using Kokkos.

When a conflict is found, only one vertex involved in that conflict must be recolored. Additionally, since the conflicts will likely happen on two processes, it is critical that both processes recolor the same vertex, otherwise conflicts may never get resolved. We adopt the same random conflict resolution scheme presented in [3]. Specifically, we use a random number generator seeded on the Global Identifier (GID) of each vertex, as this produces a consistent set of random numbers across processors and does not require any communication. In a conflict, the vertex with the larger random number gets recolored. The *rand* argument in Algorithm 1 represents this set of consistent random numbers.

We use KokkosKernels coloring to recolor the vertices in need of recoloring. First, we needed to alter the KokkosKernels coloring function to allow it to accept a partial coloring. It was also necessary to test the reaction of KokkosKernels to various partial colorings. For instance, we found that VB-BIT would fix the conflicts in an input coloring. Since we control the conflicts using a distributed scheme, this sometimes meant that our coloring could not resolve all the conflicts, as each process may be correcting the coloring. If no conflicts were present in the coloring provided to VB-BIT, then the colors were not changed. This was unfortunately not the case for the implementation of EB, as each attempted recoloring, with or without conflicts, resulted in a coloring with around the same number of conflicts as the initial coloring.

Thus, our final recoloring solution used the VB-BIT implementation in Kokkos Kernels, and our conflict resolution approach is illustrated in Algorithm 2. We set all vertices that needed recolored to have a color of zero, which VB-BIT interprets as those vertices being uncolored. Each process may need to recolor some ghosted vertices, even though that process does not have enough coloring information to correctly recolor that vertex. This is done to prevent VB-BIT from resolving conflicts on its own, and these colors are overwritten in the

Algorithm 2 Our coloring conflict resolution approach

```
procedure RESOLVE-CONFLICTS(Graph G = (V, E), colors, rand)
    for all v ∈ V do
        for all neighbors n of v do
            if colors[v] == colors[n] then
                if rand[v] > rand[n] then
                    colors[v] = 0
                else
                    colors[n] = 0
    colors = VB-BIT(G, colors)
    Communicate colors of owned vertices to ghost copies
    global-conflicts = 0
    for all v ∈ V do
        for all neighbors n of v do
            if colors[v] == colors[n] then
                global-conflicts + = 1
    return global-conflicts
```

subsequent communication.

Our first implementation, called "1 Ghost Layer" (1GL), follows the methods described so far. Our second implementation, called "2 Ghost Layer" (2GL), follows these methods, but adds to the subgraphs stored on each process. In a typical graph distribution, the subgraphs for each process do not include any of the neighbors of ghost vertices, unless those neighbors are owned by the process. In our 2GL approach, we include all the neighbors of the ghosted vertices in each process's subgraph, giving us two ghost layers.

To the best of our knowledge, this 2GL approach has not been explored before. The main reason we use this approach is in an attempt to reduce the total amount of communication involved in the 1GL approach. The 2GL approach has the potential to reduce the total number of collective communication operations because in well-partitioned and regular input graphs, the second ghost layer is likely to be made up of interior vertices that are owned by other processes. This means that after their initial coloring, these vertices are likely to retain their initial color. Thus, each process should be able to resolve more conflicts independently and in a way that is consistent across processes. However, in the 2GL scheme, each communication is more expensive than in the 1GL, so in order to see a speedup the number of rounds must decrease enough to make up for the increased cost of each communication.

Algorithm 3 shows our approach to constructing the connectivity of the copied vertices. The *gids* argument is an array containing the GID of each local vertex, and the *owners* argument is an array containing the owning process ID of each local vertex.

After this connectivity information is added, we follow the same general approach as before. However, we are able to optimize our conflict detection in a similar way to the boundary vertex reordering as presented in [3]. We are able to find all the distributed conflicts by only looking through the ghost vertices' adjacencies, since they neighbor all local boundary vertices. Thus, if we keep the new connectivity information distinct from the original graph, we can safely detect all conflicts by only looking at the ghost vertices' colors and their neighbors' colors.

Algorithm 3 How we build the adjacency information for copied vertices

```
procedure BUILD-COPY-ADJS(Graph G = (V, E), gids, owners)
    gids-to-send ← ∅
    for all (u,w) ∈ E do
        if u ∉ V then
            gids-to-send ← gids[u]
        if w ∉ V then
            gids-to-send ← gids[w]
    send each entry in gids-to-send to its owner
    for all gids received do
        send vertex's adjacency list back to sender
    Construct graph from received adjacency information.
```

4. Experimental Setup. The scaling results we present were obtained from the DCS supercomputer housed at Rensselaer Polytechnic Institute. The system has 16 nodes each equipped with 2 IBM Power 9 processors clocked at 3.15 GHz, 4 NVIDIA Tesla V100 GPUs with 16 GB of memory each, 512 GB of RAM, 1.6 TB Samsung NVMe Flash memory, connected with a Mellanox Infiniband interconnect.

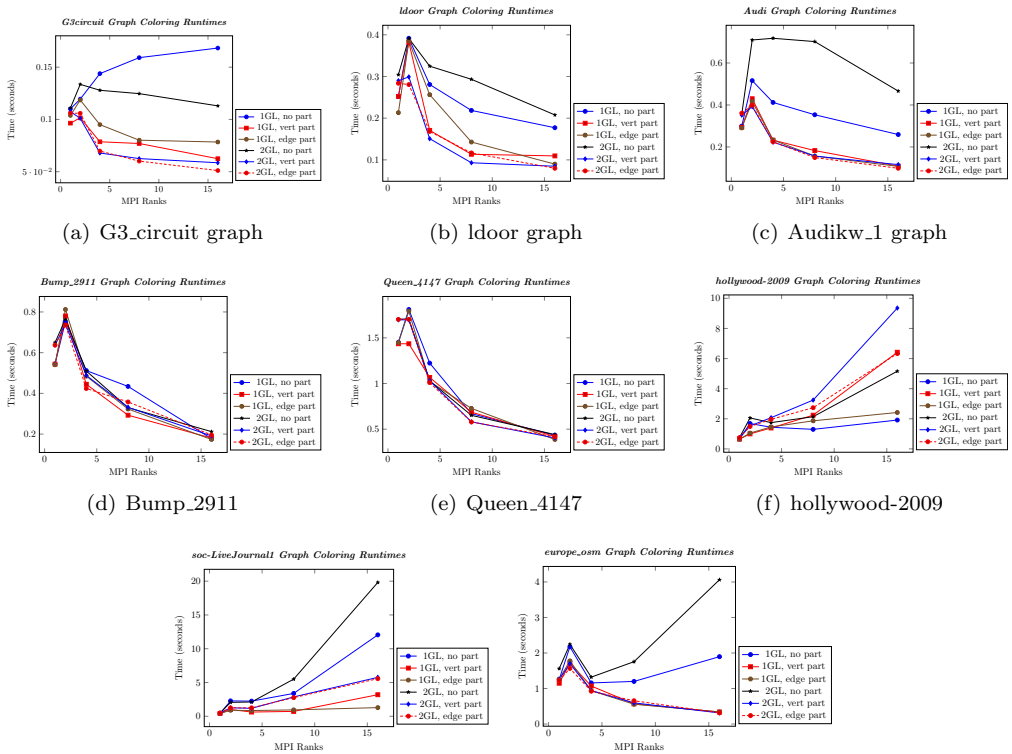
Table 4.1: Summary of input graphs

Graph	Class	#Vertices	#Edges	δ_{avg}	δ_{max}
G3_circuit	Circuit	1.6 M	7.7 M	4.83	5
ldoor	PDE Problem	0.9 M	42.5 M	44.6	77
Audikw_1	PDE Problem	0.9 M	76.7 M	81.28	345
Bump_2911	PDE Problem	2.9 M	124.8 M	42.87	194
Queen_4147	PDE Problem	4.1 M	325.3 M	78.45	89
hollywood-2009	Social Network	1.1 M	112.8 M	98.91	11467
soc-LiveJournal1	Social Network	4.8 M	85.7 M	17.68	20333
europe_osm	Road Network	50.9 M	108.1 M	2.12	13

A summary of the inputs we used is listed in Table 4.1. We used graphs from the SuiteSparse Matrix Collection (formerly UFL Sparse Matrix Collection [6]). The max degree, δ_{max}, is used as an upper bound for the number of colors used, as any incomplete, connected undirected graph can be colored using at most δ_{max} colors [4]. We selected many of these inputs because they were used by Deveci et. al. and we would be able to compare our performance against theirs. We include inputs which are primarily from Partial Differential Equation (PDE) problems because they are close to our target application. We include social networks to show how our approach fairs against small-world inputs. Our preprocessing removed self-edges and multi-edges, but did not reorder the vertices as in [3]. We used ParMETIS [16] for pre-partitioning, and ran experiments for pre-partitioning based on vertices and edges. Our performance results include runtime, number of colors used, and how many rounds of communication were necessary.

5. Results. Figure 5.1 shows the runtimes of both of our approaches with different pre-partitioning strategies. The timings do not include the time it takes to construct the graph in either case, to focus analysis solely on the performance difference between these two approaches after the graphs are constructed.

Fig. 5.1: MultiGPU runtimes for the 1 Ghost Layer (1GL) and 2 Ghost Layer (2GL) approaches. Pre-partitioning schemes include no pre-partitioning (no part), vertex pre-partitioning (vert part), and partitioning where vertices are weighted by degree (edge part).



(a) G3_circuit graph (b) ldoor graph (c) Audikw_1 graph

(d) Bump_2911 (e) Queen_4147 (f) hollywood-2009

(g) LiveJournal (h) europe_osm

We use ParMETIS [16] to pre-partition our inputs. We use two different partitioning objectives on our inputs. The first creates partitions so that each process has a similar number of vertices. The second weights vertices by their degree and partitions them so each process has a similar total weight. We will refer to this strategy as "edge-balanced" partitioning. The default, which we call "no partitioning" partitions the vertices by their global vertex ID and gives each process a different set of contiguous vertex IDs.

The single-node runs only include the time it takes for KokkosKernels to color the entire input graph on a single GPU. In these runs, each MPI rank gets its own GPU, so the number of MPI Ranks corresponds directly to the number of GPUs. The initial spike in the runtimes is due to the introduction of communication overhead. For the pre-partitioned inputs, this overhead should decrease as the number of ranks increases. This is because partitioning aims to reduce the number of edges between vertices owned by different processes, thus reducing the amount of communication necessary between processes. This also speeds up our conflict resolution, as we need to communicate the updated colors after conflicts are recolored. Our results show that pre-partitioning the inputs is essential to obtaining speedup from the single-node runs.

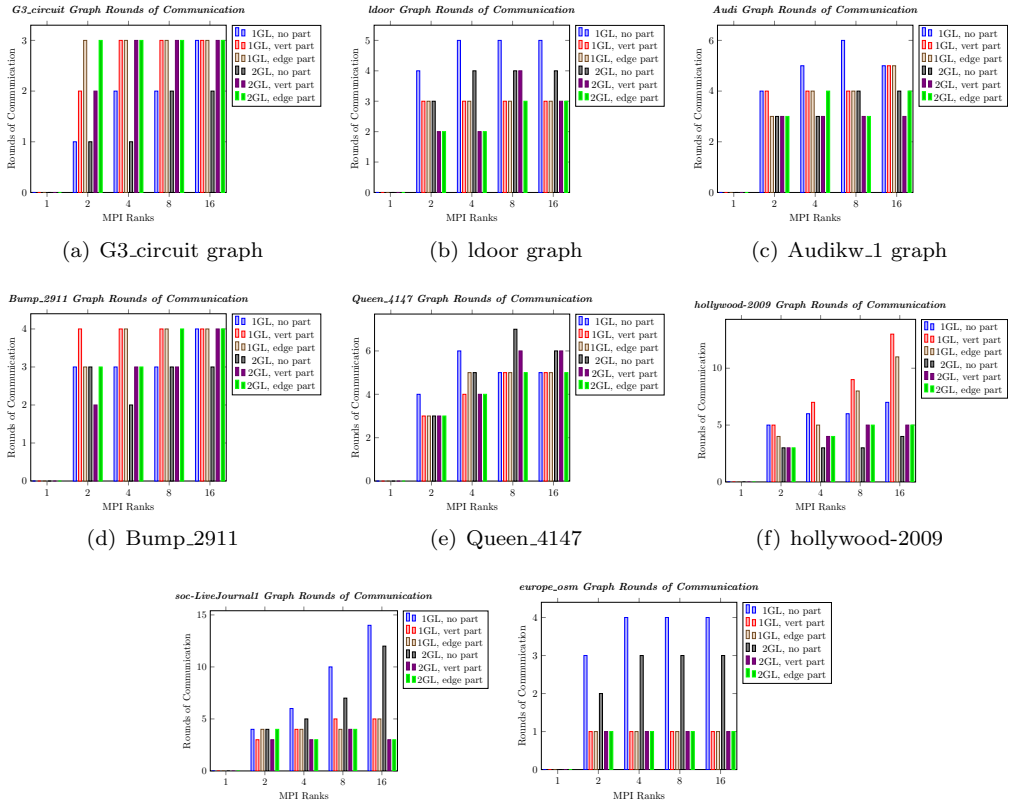
As Figure 5.1 shows, both of our approaches scale for a majority of our inputs. For the hollywood-2009 (Figure 5.2(f)) and soc-LiveJournal1 (Figure 5.2(g)) inputs, we do not see scaling because these graphs are from social networks which means they are almost certainly

"small-world" graphs. Small-world graphs characteristically have a few vertices with very high degree, and many vertices with relatively low degrees. This makes partitioning difficult, and it means distributed algorithms will need a large amount of communication to color this class of graphs. Our results show this, but they also show that partitioning is able to mitigate this communication overhead somewhat.

The rest of our inputs are from problems that we would expect our algorithms to handle well, and our results show this. The best speedup from the single node runs is seen in the europe_osm graph, in Figure 5.2(h). The 2GL approach sees almost a 5x speedup from its single node run. Both algorithms see on average around a 3x speedup, with the smallest input, G3_circuit (Figure 5.2(a)) having the smallest speedup of 2.18x.

Another interesting note is that there is no clear-cut winner between both of our methods. On the Audikw_1 graph, the 1GL approach beats the 2GL by 1.17x. However, for G3_circuit, the 2GL is faster by 1.13x. This likely means that for certain graphs, the communication cost of maintaining the second ghost layer does not result in a corresponding decrease in the rounds of communication. However, in other graphs, the number of rounds decrease enough to offset the increased cost of communication.

Fig. 5.2: Number of collective communication rounds used for the 1 Ghost Layer (1GL) and 2 Ghost Layer (2GL) approaches. Pre-partitioning schemes include no partitioning (no part), vertex partitioning (vert part), and edge-balanced partitioning (edge part).



(a) G3_circuit graph (b) ldoor graph (c) Audikw_1 graph

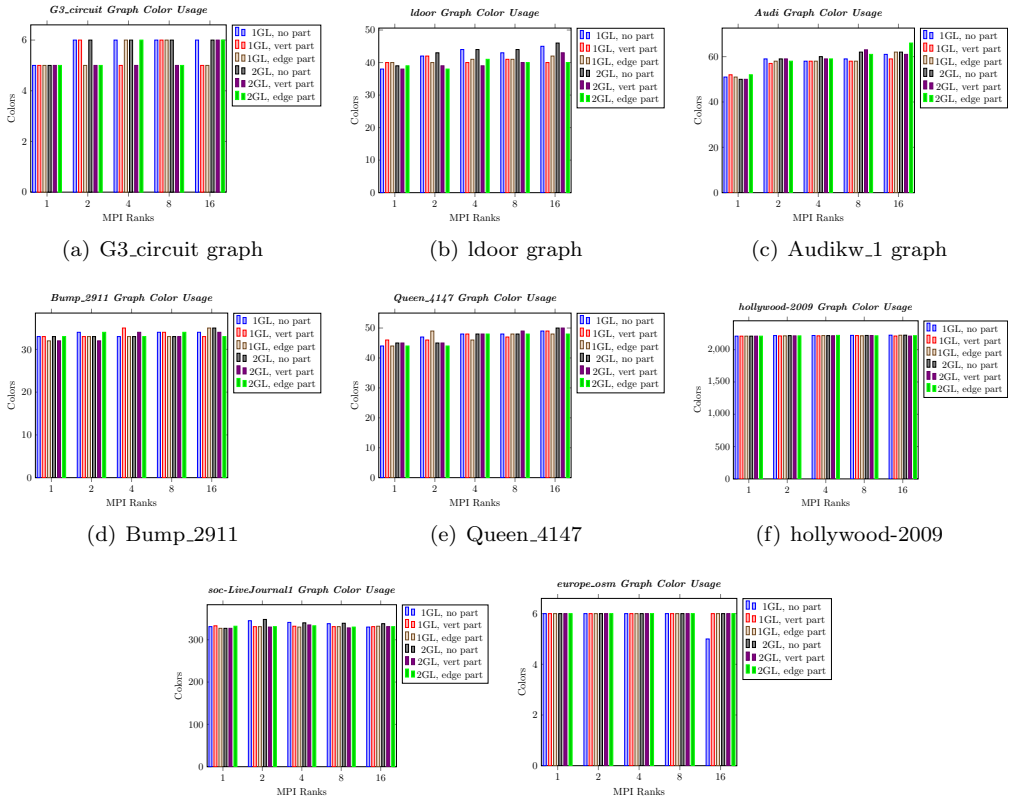
(d) Bump_2911 (e) Queen_4147 (f) hollywood-2009

(g) LiveJournal (h) europe_osm

Figure 5.2 shows how many rounds of collective communication are used by each ap-

proach and pre-partitioning strategy. Our results show that for our input graphs, the 2GL approach generally uses fewer rounds than the 1GL approach using the same pre-partitioning strategy. There are a few exceptions to this rule, such as the 8 node run of the Queen_4147 graph, (Figure 5.3(e)). Also worthy of note is that even though the 2GL approach sees a decrease in the rounds of communication for LiveJournal (Figure 5.3(g)) and hollywood-2009 (Figure 5.3(f)), the corresponding runtime plots show that the partitioned 1GL generally outperform the 2GL, because the decrease in rounds does not make up for the additional communication cost.

Fig. 5.3: Number of colors used for the 1 Ghost Layer (1GL) and 2 Ghost Layer (2GL) approaches. Pre-partitioning schemes include no partitioning (no part), vertex partitioning (vert part), and edge-balanced partitioning (edge part).



(a) G3_circuit graph (b) ldoor graph (c) Audikw_1 graph

(d) Bump_2911 (e) Queen_4147 (f) hollywood-2009

(g) LiveJournal (h) europe_osm

Figure 5.3 shows how many colors our approaches use as the number of MPI ranks increase. Generally, we expect to see that increasing the number of MPI ranks increases the number of colors used. This is because we are using independent, parallel greedy coloring algorithms on each process, and then resolving those greedy colorings together. However, our plots show that there does not seem to be a strong trend to how the number of colors change as the number of ranks increases. The most pronounced increase can be seen in the Audikw_1 graph, in Figure 5.4(c), but in the worst case, that input sees about a 24% increase in the number of colors used. We also stay below the max degree for most of these inputs, with the exception being G3_circuit (Fig. 5.4(a)), where we use one more color.

6. Future work. We have many directions for future work based on this project. We have an idea that may eliminate iteration in a distance-1 coloring that involves the concept of vertex separators. Vertex separators are sets of vertices that divide a graph into two disconnected components. If, instead of coloring the boundary vertices, we are able to find vertex separators between each process's subgraphs and color those separators, each process should be able to color the rest of its subgraph independently without any iteration.

Additionally, we plan to extend this hybrid implementation to distance-2 coloring, and partial distance-2 coloring. Our final aim will be to have a hybrid MPI+X implementation for all of these coloring problems. This work's final application is the optimization of the computation of sparse Jacobian and Hessian matrices, both of which are used in automatic differentiation and other computational problems [2].

REFERENCES

[1] J. Allwright, R. Bordawekar, P. Coddington, K. Dincer, and C. Martin, *A comparison of parallel graph coloring algorithms*, SCCS-666, (1995), pp. 1–19.

[2] D. Bozdağ, Ü. V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, *Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 2418–2446.

[3] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.

[4] R. L. Brooks, *On colouring the nodes of a network*, in Mathematical Proceedings of the Cambridge Philosophical Society, vol. 37, Cambridge University Press, 1941, pp. 194–197.

[5] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, *Graph coloring algorithms for multi-core and massively multithreaded architectures*, Parallel Computing, 38 (2012), pp. 576–594.

[6] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

[7] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, *Parallel graph coloring for manycore architectures*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 892–901.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3202–3216.

[9] M. Garey, D. Johnson, and H. So, *An application of graph coloring to printed circuit testing*, IEEE Transactions on circuits and systems, 23 (1976), pp. 591–599.

[10] A. H. Gebremedhin and F. Manne, *Scalable parallel graph coloring algorithms*, Concurrency: Practice and Experience, 12 (2000), pp. 1131–1146.

[11] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, *Colpack: Software for graph coloring and related problems in scientific computing*, ACM Transactions on Mathematical Software (TOMS), 40 (2013), p. 1.

[12] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, *Evaluating graph coloring on gpus*, ACM SIGPLAN Notices, 46 (2011), pp. 297–298.

[13] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al., *An overview of the trilinos project*, ACM Transactions on Mathematical Software (TOMS), 31 (2005), pp. 397–423.

[14] M. F. Hoemmen, *Tpetra project overview.*, tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[15] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[16] G. Karypis, K. Schloegel, and V. Kumar, *Parmetis*, Parallel graph partitioning and sparse matrix ordering library. Version, 2 (2003).

[17] G. Rokos, G. Gorman, and P. H. Kelly, *A fast and scalable graph coloring algorithm for multi-core and many-core architectures*, in European Conference on Parallel Processing, Springer, 2015, pp. 414–425.

[18] A. E. Sariyüce, E. Saule, and Ü. V. Çatalyürek, *Scalable hybrid implementation of graph coloring using mpi and openmp*, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 1744–1753.

PROVIDING SOFTWARE ENGINEERING SUPPORT FOR EMPIRE

JOSHUA R. BRAUN* AND JASON M. GATES†

Abstract. The EMPIRE project is a plasma physics simulation code developed by Sandia National Laboratories. In an effort to improve overall software quality and software engineering practices, certain tasks were accomplished. These include creating an onboarding checklist in EMPIRE's documentation to better orient newcomers to the team, renovating an incomplete testing suite to improve the amount of code covered by the tests and the documentation of the tests, and rewriting a collection of scripts to be more organized, better documented, and fully tested. Continued work includes restructuring the project's wiki.

1. Introduction. Sandia National Laboratories' Electromagnetic Plasmas in Realistic Environments (EMPIRE) team at focuses on bleeding edge research in the area of plasma physics simulation software. Over the past two years, Jason M. Gates has been working to improve the professionalism of EMPIRE's software infrastructure. This includes Software Engineering practices such as:

- Using GitLab for lightweight project management.
- Establishing a clear workflow and guidelines for contributing to the code.
- Encouraging documentation and code style guidelines for the software.
- Protecting the main development branches and requiring code review before merging changes.
- Setting up and maintaining nightly code testing and integration processes utilizing the Jenkins continuous integration (CI) software.

As a 2019 summer intern working on this project, I have helped further develop the software infrastructure for the EMPIRE team. The main projects involved in doing this were

- Creating an onboarding checklist,
- Renovating an incomplete test suite, and
- Rewriting a collection of scripts to be better organized, better documented, and fully unit tested.

2. Creating an Onboarding Checklist. This first task was aimed at providing better documentation for those who are new to the EMPIRE team. At the start of the summer, the state of EMPIRE's documentation was very much geared towards those already familiar with the project. An important software engineering practice is to develop great documentation such that anyone can take a look at it and quickly come up to speed on the project, how to use it, and how to contribute to it in a way that meshes well with the existing team [7]. The first step towards that goal was to create a dedicated *Onboarding Checklist* page.

This page is placed on EMPIRE's GitLab wiki, and it contains checklist items for new team members to work their way through as a first step in getting up to speed on the project. The team member's mentor copies and pastes that into a GitLab *issue* and assigns the issue to the new team member. Any questions the new member has can be posted in the comments section of the issue, where anyone on the team can see it and reply.

3. Renovating `test_empire_testing_utils`. The next task in developing better software infrastructure for EMPIRE was to renovate a test suite for a section of EMPIRE code. The amount of code covered by these tests was below what it needed to be, and the tests needed some restructuring for better documentation and cleaner code. This is another

*Oral Roberts University, School of Engineering, bbraunj@gmail.com
†Sandia National Laboratories, Software Engineering and Research Department, jmgate@sandia.gov

key software engineering practice[2]: write well-documented, clean, and readable code. The idea behind this is that code is more often read than it is written. Even if it is only yourself who is going to be looking at the code, it is still good practice to do this. If code is hard to read and undocumented, it can be very easy to come back a couple weeks, months, or years later and not have any idea why the code was written the way it was, or even what it does.

The test suite in this task was written using *pytest*, a Python testing library that aims to "make it easy to write small tests, yet scales to support complex functional testing for applications and libraries"[4]. Many of this testing library's features, such as parametrization, and built-in fixtures, like *monkeypatch* and *tmpdir*, were used to create elegant, effective tests.

The code coverage at the beginning of renovating this test suite was 57%, and by the time work was finished, the coverage was raised to 82%, as shown in Figure 3.1. Generally, code coverage above 80% is considered good[3], so this is a huge improvement over the previous code coverage level. Several files did not have any tests written to test them, so tests had to be created from scratch.

Module ↓	*statements*	*missing*	*excluded*	*coverage*
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/StREEQ.py	39	10	0	74%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/__init__.py	100	7	0	93%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/__main__.py	12	12	0	0%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/automatic_diagnostics.py	58	52	0	10%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/convergence_grid.py	133	11	0	92%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/cubit_runner.py	36	0	0	100%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/empire_execution_control.py	64	43	0	33%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/empire_tpls.py	34	4	0	88%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/exodus_reader.py	151	7	0	95%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/h5part_reader.py	25	0	0	100%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/history_file_reader.py	123	6	0	95%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/parse_constants.py	40	8	0	80%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/read_install_info.py	9	1	0	89%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/trilinos_bin.py	55	3	0	95%
/home/josbrau/em_builds/empire_2019-09-30/build-hybrid-spin/vvtest_config/empire_testing_utils/xdmfdiff.py	183	29	0	84%
Total	**1062**	**193**	**0**	**82%**

Fig. 3.1: Code coverage for `empire_testing_utils`.

4. One `build_empire.py` Script to Rule Them All. At the start of the summer, the scripts used to build EMPIRE from its source were scattered and disorganized. There were different scripts written in bash for each different machine, and consequently, they were not very well-maintained across the various machine scripts. Additionally, unit testing options for bash are limited; some options exist, but none are as useful as pytest, which would be used in the new build script. These build scripts were collected into a single GitLab repository, but because these were specific to Sandia testbeds and high-performance computers (HPCs), team members also had their own build scripts lying around.

Furthermore, the main EMPIRE wiki had several different pages for instructions on building EMPIRE, most of which were out of date and would no longer work. Documentation of the scripts was hit or miss, and certainly no HTML documentation existed.

The goal for this task was to create a single `build_empire.py` script to rule them all. It would be written fully in Python3 (although some calls to bash are necessary), fully documented, and fully unit tested (meaning a coverage score above 80%). Its capabilities would include:

- Clone the EMPIRE repositories from GitLab.
- Configure EMPIRE for building.

- Choose an existing Trilinos installation to build against or build and install Trilinos from scratch (using another script file).
- Build EMPIRE in different ways, depending on command-line options.
- Run various components of the EMPIRE test suite.

This script would be deployed across all nightly testing of EMPIRE that currently happens. Additionally, all EMPIRE developers would be guided to this script for their building needs via the wiki.

4.1. Documentation. A major goal in writing this script is to showcase what well-documented code in EMPIRE should look like. This is done through the following:

- Descriptive, Google-style[6] docstrings.
- Sphinx-generated HTML documentation from docstrings.
- Helpful README's within the BuildScripts GitLab repository.

4.1.1. HTML Documentation. Figure 4.1 shows the main page of the Sphinx-generated HTML documentation. This takes after the style of the popular documentation website Read the Docs.

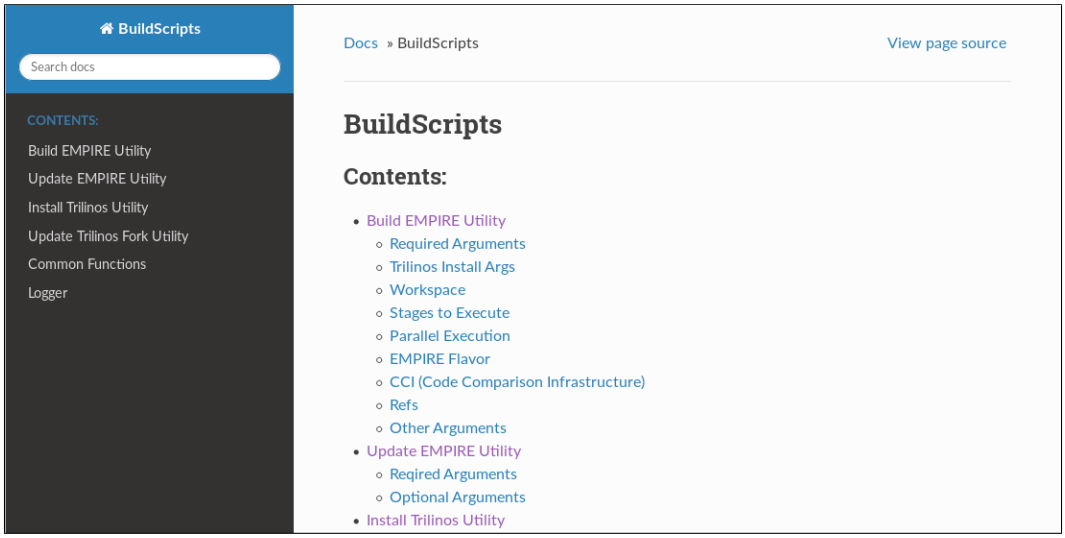


Fig. 4.1: Homepage of Sphinx-generated HTML documentation.

The way that documentation is generated from docstrings within the code is shown through the code snippet in Listing 1 and the corresponding Sphinx-generated HTML in figure 4.2. The class docstring is a combination of Google-style and reStructuredText methods of Python documentation. This allows for a combination of readability within the code provided by the Google-style (i.e. the Attributes: section) and useful syntax highlighting of reStructuredText (i.e. the double backticks around 'em', 'pic', 'fluid', 'hybrid', and 'spin' in the first bullet point). These docstrings and the syntax within, as shown in figure 4.2, are intelligently interpreted by Sphinx and converted into easy-to-read HTML documentation.

```
class BuildEmpire():
  """
  This class takes care of any *EMPIRE* building needs. For
  usage information, run ``python build\_empire.py --help``
  on the command line. The basic capabilities of this class
```

```
  are the following:
    - Clone *EMPIRE* (choosing ``em``, ``pic``,
      ``fluid``, ``hybrid``, and/or ``spin``).
    - Build *EMPIRE*.
    - Choose existing *Trilinos* install to build
      against.
    - Build & Install *Trilinos* (using another script
      file).
    - If all goes well, update list of blessed SHAs on
      the wiki.

  Attributes:
    args (Namespace):  Contains data for all the flags/
      arguments passed to this script.
    build_dir (str):  Path to *EMPIRE*/*SPIN*'s build
      directory.
  ...
  """
```

Listing 1: Class docstring for figure 4.2

In addition, Sphinx is able to recognize code snippets within docstring and provide formatted, syntax-highlighted representations of them in the HTML documentation. This is shown in figure 4.3.

Another notable feature of Sphinx is its ability to use plugins to extend its features. This capability was used in the BuildScripts documentation to document the command-line options of the `build_empire.py` script. The plugin *sphinx-argparse* converted the Python *argparse* object used in the script into Sphinx-compatible documentation. An example of how this looks in the HTML is shown in figure 4.4.

To make it easy for users to compile the code documentation into HTML, a simple script was created and placed in the `doc` folder of the BuildScripts repository. All users have to do is run the following command:

```
$ ./make_html_docs.sh
```

4.1.2. README Documentation. An important part of developing good documentation is having helpful READMEs throughout the repository. This gives users information that will help them get started with the software quickly and with clear direction. Because modern code hosting websites will typically display the formatted README's contents below the file listing, it makes it even easier for users to get information quickly. An example of how this is used in the updated BuildScripts repository is shown in figure 4.5.

4.2. Unit Testing. As mentioned in section 3, unit testing is an important software engineering practice. Consequently, any further development work that is done (such as this `build_empire.py` script) should be fully unit tested, meaning the test suite covers at least 80% of the code base [3].

Since it was unclear at first where the script was going and what exact behavior was to be expected, it was difficult to achieve an ideal test-driven development (TDD) workflow; that is, where one writes a test, sees it fail, writes code to make the test pass, refactors the code, and repeats this process to develop the software[1]. Rather than this approach, a method called spiking was used. Through this method, one simply focuses on exploring the libraries that will be used and writing code that works. It is meant to be "quick and dirty"[5], but I modified the approach somewhat by writing good docstrings as I went.

class **build_empire.BuildEmpire**

Bases: `object`

This class takes care of any *EMPIRE* building needs. For usage information, run `python build_empire.py --help` on the command line. The basic capabilities of this class are the following:

- Clone *EMPIRE* (choosing `em`, `pic`, `fluid`, `hybrid`, and/or `spin`).
- Build *EMPIRE*.
- Choose existing *Trilinos* install to build against.
- Build & Install *Trilinos* (using another script file).
- If all goes well, update list of blessed SHAs on the wiki.

args

Contains data for all the flags/arguments passed to this script.

Type

Namespace

build_dir

Path to *EMPIRE/SPIN*'s build directory.

Type

str

Fig. 4.2: Sphinx-generated class documentation from class docstring in Listing 1.

This made it easier to write the docstrings so it would not be one large task at the end that had been put off. Once a clear understanding of what the code should look like has been established, the method calls to throw away the code and start from scratch using a methodical TDD approach. Rather than doing this, I focused on writing tests that enforced functionality as desired and refactoring code to be better organized, more concise, and more readable.

Code coverage is not the only measure of how well-tested the code is, but it is the most concrete. All it refers to is how many statements of the code are run by the testing suite compared to the total number of statements in the code. Much can be run by the test suite while a certain functionality is not explicitly tested. The goal in unit testing the `build_empire.py` script was to achieve both high code coverage and all functionality tested.

The testing library used to test the `build_empire.py` script is the same as that used in section 3: pytest. The code coverage of the `build_empire.py` script, helper files like `common_functions.py` and `logger.py` (discussed further in section 8), and the `build_trilinos.py` script (discussed further in section 5) are shown in figure 4.6.

class **common_functions.Timer**(*timer_name, dict_to_save_to*)

Bases: `object`

This class can be used in a `with` statement to automatically time whatever is in the `with` statement and save that to the dictionary that is passed in.

For example, this code:

```
db = {}
with Timer('spam', db):
   print("Going to sleep")
   time.sleep(1.234)
print(db)
```

Produces this output:

```
**** Start Timer 'spam'
Going to sleep
**** End Timer 'spam' (1.23 seconds)
{'Timer: spam': 1.2343378067016602}
```

Fig. 4.3: Code snippets in Sphinx-generated HTML documentation

Build EMPIRE Utility

```
usage: Build EMPIRE Utility [-h] [--trilinos-install-dir TRILINOS_DIR]
                            [--install-trilinos]
                            [--compiler {gnu,intel,cuda,clang}]
                            [--build-type {opt,debug}]
                            [--node-type {serial,Pascal60,Kepler37,openmp}]
                            [--lib-type {shared,static}]
                            [--trilinos-ref TRILINOS_REF]
                            [--workspace WORKSPACE]
                            [--stage {clone,configure,build,ctest,pytest,vvtest,extendedtesting} |
                            [-j J] [-jbuild JBUILD] [-jtest JTEST]
                            [-jctest JCTEST] [-jvvtest JVVTEST]
                            [-jpytest JPYTEST]
                            [--flavor {em,pic,fluid,hybrid}] [--spin] [--cci]
                            [--blessing-timestamp BLESSING_TIMESTAMP]
                            [--develop] [--prefix PREFIX] [--send-email]
                            [--SNLCLUSTER SNLCLUSTER] [--ref REF]
                            [--em-ref EM_REF] [--pic-ref PIC_REF]
                            [--fluid-ref FLUID_REF] [--hybrid-ref HYBRID_REF]
                            [--spin-ref SPIN_REF] [--ex-test-ref EX_TEST_REF]
                            [--build-dir BUILD_DIR] [--dry-run] [--nightly]
                            [--replay] [--build-system {ninja,make}]
                            [--warning-emails] [--werror-off]
```

Required Arguments

`--trilinos-install-dir`

Install directory for Trilinos to build against. Note that this is NOT REQURIRED for CCI.

Trilinos Install Args

`--install-trilinos`

Install Trilinos from scratch into the directory specified by '–trilinos-install-dir' (unless `--cci` is specified).

Default: False

Fig. 4.4: Sphinx-generated argparse documentation.

BuildScripts

This repo contains four main scripts:

1. **build_empire.py**
   - Configure, build, and test *EMPIRE*.
2. **install_trilinos.py**
   - Configure, build, test, and install *Trilinos*.
3. **update_empire.py**
   - Utility that helps check for any *EMPIRE* updates, push those updates to a separate branch for testing, and merge those tested branches into `master`.
4. **update_trilinos_fork.py**
   - Utility that helps bring updates from `trilinos/Trilinos:develop` into the *EM-Plasma* fork of *Trilinos* while verifying that changes do not break compatibility with *EMPIRE*.

Getting Started

Building EMPIRE

The basic required arguments for the **build_empire** script are the following:

- `--trilinos-install-dir`: The location of the *Trilinos* install to build *EMPIRE* against.
- `--flavor FLAVOR` or `--spin`: One of these flags must be specified. The `FLAVOR` can be either `em`, `pic`, `fluid`, or `hybrid`.

For further information about command line options, run:

```
$ ./build_empire.py --help
```

Fig. 4.5: Portion of the README in the root directory of the BuildScripts repository

Module ↓	*statements*	*missing*	*excluded*	*coverage*
build_empire.py	591	110	0	81%
common_functions.py	296	37	0	88%
install_trilinos.py	226	15	0	93%
logger.py	175	2	0	99%
Total	**1288**	**164**	**0**	**87%**

Fig. 4.6: Code coverage for BuildScripts

5. One `build_trilinos.py` Script to Rule Them All. The situation for the scripts used to to build, test, and install Trilinos at the beginning of the summer was not as bad as the situation for those used to build EMPIRE, but redoing them was still in order. They were still written in bash, not unit tested, and not very well documented.

The goal for this task was much as it was for the `build_empire.py` script: create a single `build_trilinos.py` script to rule them all. It would be written fully in Python3 (some calls to bash necessary), fully documented, and fully unit tested. Its capabilities would include:

- Clone the Trilinos repository from GitLab.
- Configure Trilinos for building with a variety of command-line options.
- Build Trilinos.
- Test Trilinos.
- Install Trilinos.

The approach for this script was much as it was for the `build_empire.py` script, but development went much quicker having the other script as a base. Essentially, the `build_empire.py` script was copied and pasted into a new `build_trilinos.py` file, modifications were made to support Trilinos rather than EMPIRE, and unit tests were made. The task was much simpler because, unlike the `build_empire.py` script which had to support multiple configurations (the various combinatorics of five source repositories) and multiple test types (ctest, pytest, multiple vvtest[1] suites), the Trilinos script only had to support one configuration (Trilinos) and one test type (ctest). As seen in figure 4.6, the test coverage for this script is 93%.

6. `update_empire.py` Script. Another utility that needed to be converted to Python was a script called `updateEMPIRE`. This script has the following capabilities:

- Check the EMPIRE repositories' `develop` branches to see if they are ahead of their respective `master` branches. If so, copy the `develop` branches to new branches where tests will be run on them to see if they are suitable to be merged into `master`.
- Merge updates from the testing branches into `master`.

The expectations of the creation of this script were similar to those of the previous two scripts: fully documented and fully unit tested. Documentation is up to the same standards as the previous two scripts, but because of the nature of this script, it was not feasible to run unit tests on most of it. Most all of the logic and actions of this script involve Python *subprocess* calls to an external program: git. Because the interface of doing this is through a single `run_cmd()` function, which runs the bash commands, the ability to test the script is limited.

Normally, when testing a script or module that interacts with an external service, it is desired to "break" this dependency on the external service. The best way to do this is with mocks, which allow a test to create a Python object that poses as the external service. That way, tests are able to verify certain behaviors (such as "does the script behave correctly when git does xyz?") consistently, without the unpredictable responses of the external service.

Mocks work by posing as other Python objects or functions. However, because so many different git commands are run via the single `run_cmd()` function, there is no opportunity to mock any one git command. This means that tests are not able to be reproducible; they are at the mercy of the response of the external git commands. Some days there may be EMPIRE updates and other days there may not be. These limitations have led to the testing suite for this script being very limited in scope, with a code coverage of 44%.

[1] A Sandia-developed testing tool.

7. update_trilinos_fork.py Script. A similar utility to updateEMPIRE that also needed to be converted to Python was a script called updateTrilinosFork. To understand the capabilities of this script, one must first understand the structure of EMPIRE's use of Trilinos. Because EMPIRE and Trilinos are both under active development and EMPIRE wishes to use the latest code developed by the Trilinos team, a system must be set up to ensure that updates to the develop branch of Trilinos are safe to use with EMPIRE before using them full-time. To do this, EMPIRE maintains a separate fork of Trilinos with four main branches:

- trilinos-develop point to the latest commit on the real Trilinos develop branch (trilinos/Trilinos:develop) that EMPIRE has validated through its nightly testing pipeline.
- develop contains everything in trilinos-develop, plus any changes that EMPIRE developers have made that have not yet made it into Trilinos' develop branch. These additional changes are rare, but EMPIRE requires this flexibility as they continue to push the bleeding edge.
- potential-trilinos-develop points to the tip of the real Trilinos develop branch, and it will become the new trilinos-develop whenever EMPIRE's nightly testing pipeline passes.
- potential-develop points to the merge commit created when you merge potential-trilinos-develop into develop, and is the branch that will be tested in the midst of the nightly pipeline.

Because the process for getting updates into Trilinos' develop branch can take days with its automated testing system for pull requests, the develop branch of the EMPIRE fork serves as a quicker way to merge in new changes that EMPIRE developers may need to make. Changes that are merged into this branch still need to be reviewed and approved by Trilinos developers, but they don't have to go through the long automated testing process.

With this system in mind, the update_trilinos_fork.py script has the following capabilities:

- Check to see if there are any updates to Trilinos' develop branch such that we can update the trilinos-develop and develop branches of EMPIRE's fork.
- Push updates from potential-develop to develop.

This script, like the update_empire.py script, has excellent documentation but has limitations when it comes to unit testing. In fact, this script is so reliant on Python subprocess calls that writing tests at all is unfeasable. Consequently, this script lacks a test suite.

8. Logger. In the process of creating the build_empire.py script, a Logger utility was created. This utility was subsequently integrated into the other three scripts because of its usefulness. Every shell command and print statement in these scripts is executed via a Logger object. As a way to document the commands, a message describing its purpose is required. The output of these commands and print statements is put into an HTML file, where each command is expandable to show more information about it. This includes the timestamp, the command itself, the current working directory, return code, stdout, and stderr. For an example, see figure 8.1.

The way this is structured in the scripts is that there is one parent Logger object for the whole script and each stage within the script has its own Logger object that is a child of the parent. This makes it easier to categorize commands and print statements. In the HTML output, each child Logger is initially collapsed (figure 8.2) but can be expanded to see more information (figure 8.3).

In addition to the HTML output, the Logger class outputs all the log data except

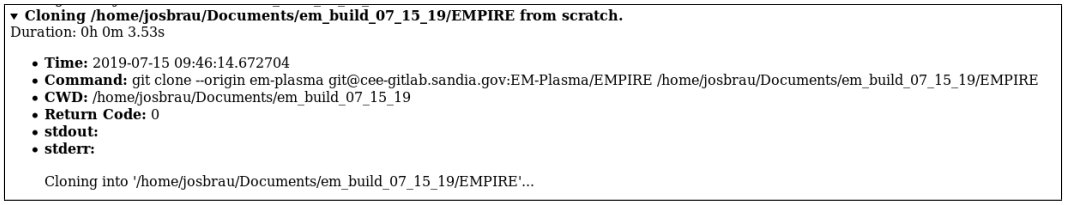


Fig. 8.1: HTML expanded command information

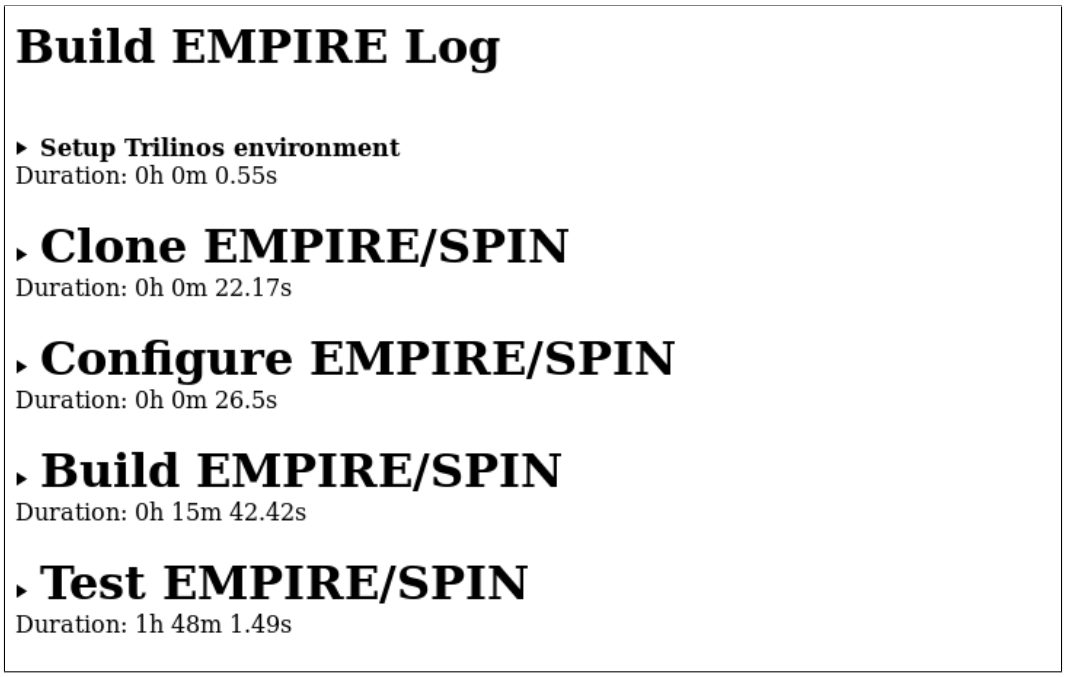


Fig. 8.2: HTML collapsed child Loggers

for `stdout` and `stderr` into a JSON file. This allows for quick machine parsing of that information so that it can be analyzed over time. For example, one could see how the duration of the build stage changed over time by parsing this information from multiple JSON log files.

Another consideration in the creation of the Logger class was memory consumption with large `stdout` and `stderr` streams. The particular use case that brought about this consideration was building EMPIRE with CUDA, which can generate `stdout/stderr` logs several gigabytes in size. Because of this, the `stdout` and `stderr` streams cannot simply be stored inside of the Logger object as an attribute. This would use up far too much memory.

To solve this issue, the `stdout` and `stderr` streams are saved to files uniquely associated with that command as the streams are generating data. When the HTML file is built, those files are read and written to the HTML file one line at a time to avoid loading all of that information into memory. It was deemed unnecessary to put `stdout` and `stderr` in the JSON file since viewing information is not its purpose. However, since it has all

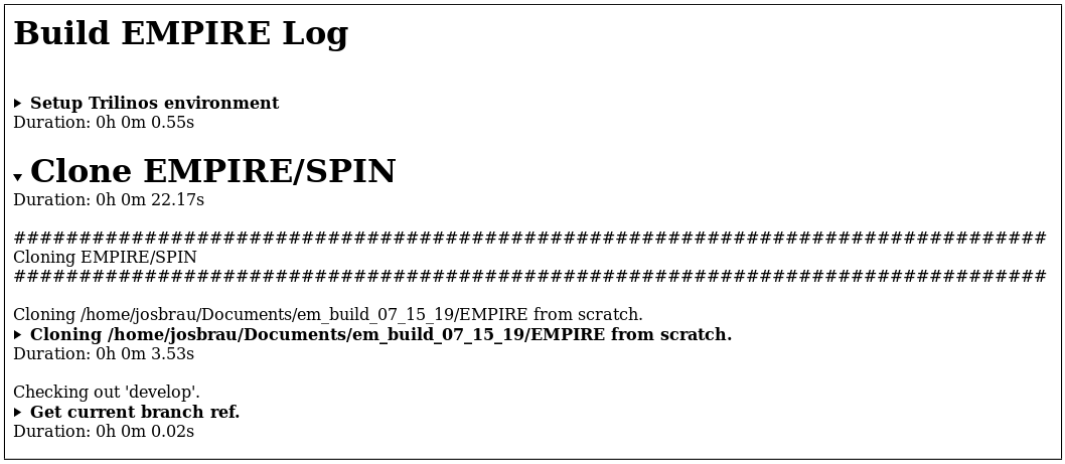


Fig. 8.3: HTML expanded child Logger

the other information on the commands, including the unique command ID associated with the `stdout/stderr` files, the HTML file is able to be generated using just the JSON and `stdout/stderr` files.

9. Refactor Project Wiki. The last task for the summer was to refactor the project wiki. The state of it at the beginning of the summer was one that made it difficult for newcomers to the project to get up to speed. The Onboarding Checklist was a first step to make this process better, but the whole wiki really needed to be restructured. The home page was essentially just a list of helpful links and resources, rather than a README-style page with intro, install, usage, and contributing sections. The overall goal in refactoring the wiki was to provide easy-to-access information about EMPIRE that guides one through the basics of the project and provides resources for digging deeper. This process is currently in development, but is getting close to being finished. Overall, this process will be very helpful for the use of EMPIRE as a software package.

10. Conclusions. The changes to the EMPIRE wiki and existing test suite, the renovated scripts, and the new Logger utility are worthwhile improvements to the quality of EMPIRE as a software package. Not only do they lessen the barrier to new team members/users, but they also unify parts of the project that had become disjointed over time.

Continued work will involve restructuring the EMPIRE wiki to be even better suited for people unfamiliar with the project. Currently, the homepage is essentially just a list of helpful links for those who already know what is going on. The goal would be to provide documentation that is easy for newcomers to follow and get up to speed on all they need to know about the project.

REFERENCES

[1] U. Bob, *The Three Rules Of TDD*, http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd.

[2] T. Peters, *PEP 20—The Zen of Python*, Aug 2004, https://www.python.org/dev/peps/pep-0020/.

[3] S. Pittet, *Introduction to Code Coverage*, https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[4] Pytest-Dev, *pytest-dev/pytest*, Jul 2019, https://github.com/pytest-dev/pytest.

[5] J. Reid, *How to TDD the Unknown with a Spike Solution*, Jan 2015, https://qualitycoding.org/spike-solution/.

[6] R. Ruana, *Example Google Style Python Docstrings*, 2015, https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html.

[7] A. Trica, *The Importance of Documentation in Software Development*, Sep 2018, https://filtered.com/blog/post/project-management/the-importance-of-documentation-in-software-development.

THE FUTURE OF COMPUTING: INTEGRATING SCIENTIFIC COMPUTATION ON NEUROMORPHIC SYSTEMS

LEAH REEDER[†], JAMES B. AIMONE[‡], AND WILLIAM M. SEVERA[§]

Abstract. Neuromorphic computing is known for its integration of algorithms and hardware elements that are inspired by the brain. Conventionally, this nontraditional method of computing is used for many neural or learning inspired applications. Unfortunately, this has resulted in the field of neuromorphic computing being relatively narrow in scope. In this paper we discuss two research areas actively trying to widen the impact of neuromorphic systems. The first is Fugu, a high-level programming interface designed to bridge the gap between general computer scientists and those who specialize in neuromorphic areas. The second aims to map classical scientific computing problems onto these frameworks through the example of random walks. This elucidates a class of scientific applications that are conducive to neuromorphic algorithms.

1. Introduction. Computer scientists have been fascinated by the brain and its connections to scientific computation for years. The father of theoretical computer science Alan Turing himself was tormented with the question "can machines think?" in his 1950 paper *Computing Machinery and Intelligence*. Now with modern technology, we are finally able to begin developing an answer to that question. The brain has inspired several projects that are paramount to our technological advancements today.

One such project is the EU's Human Brain Project, that has tasked itself with partially simulating a human brain. Using traditional computing methods, the brain is a computational nightmare due to its enormous number of neurons with their high connectivity [3]. However, as technology advances, we are able to simulate these neurons and their connections more efficiently using computing methods that are inspired by how the actual brain computes: neural networks.

Another common research project that has taken the world by storm and was first inspired by the brain is Artificial Intelligence (AI). The consequence of AI is that we are able to replicate human behavior on computers without having to hardcode that behavior explicitly. Machine Learning (ML) is a highly popular subset of AI that allows computers to take in data to solve an unknown algorithm. An exciting and popular result of ML is that computers, when exposed to more data, can distinguish items on their own without an explicit algorithm [2]. It has become hugely popular to use neural networks to assist this training of computers. ML techniques that are reliant on artificial neural networks are similar to human brains, as they get trained and essentially "learn".

In order to make these types of projects more scalable, numerous researchers in these fields have turned to neuromorphic computing, which is also known for its inspiration from the brain and use of neural networks. Although neuromorphic computing is widespread in many learning and neural applications, it is not found in many other engineering or computing applications. It is important to integrate these more general applications onto neuromorphic systems to see the benefits of neuromorphic spread into other areas.

1.1. Neuromorphic Computing. Neuromorpic computing blends two aspects of computing, both influenced by the brain. This nontraditional method of computing combines software paradigms that are brain-inspired with physical computer hardware that uses artificial neurons as their computational elements. In this paper we are primarily referring to spikng neural algorithms due to their ability to relay a significant amount of information

[†]Colorado School of Mines, lreeder@mines.edu
[‡]Sandia National Laboratories, jbaimon@sandia.gov
[§]Sandia National Laboratories, wmsever@sandia.gov

in small interactions [7, 10]. The spiking neural networks can be programmed onto hardware that is specifically designed to run these networks in an efficient manner with impressively low energy.

We can pair neural inspired algorithms that result in spiking neural networks with new computer architectures to achieve significant energy and volume efficiency [12]. This is especially desirable in the event that traditional computers could require more power than is reasonable due to the projected future of semiconductors and the end of Moore's Law [4, 16]. While spiking neural networks can run on traditional computing devices, when these networks are coupled with neuromorphic hardware, the computational advantages are significant [6]. Unfortunately, implementing these spiking neural algorithms and programming them on this specialized hardware is non-trivial, which has prevented neuromorphic computing from being wide spread in the general scientific community.

1.2. The Future of Computing. For decades, the scientific computing community has been focused on numerically solving challenging mathematical problems. The world's best supercomputers have been built in order to comply with the high demands scientists, engineers, and mathematicians have to compute these difficult problems. When this technology curve winds down due to the end of Moore's law, nontraditional methods of computing, such as neuromorphic, can rise up in the ranks [9].

Unfortunately, neuromorphic computing is not as ubiquitous as its traditional counterparts because a large knowledge base of neural computation is required. In this paper we will discuss several efforts at Sandia National Laboratories to try to bridge this gap. In Section 2 we will discuss Fugu, a high-level interface currently being developed to make neuromorphic computing more accessible to the general scientific computing community. In Section 3 we will then present a well known scientific computing technique, random walks, that has been implemented with a neural algorithm and on neuromorphic hardware. These two sections embody current efforts to show that not only can we implement non-neural and non-learning applications on these frameworks, but we can also achieve significant performance benefits on these computations as well.

2. Fugu. Fugu, a high-level tool for scientific applications, allows users to utilize and build spiking neural algorithms for arbitrary computations [1]. This is especially desirable because it allows users to be able to adopt neural algorithms for general computations, showing that neuromorphic paradigms can be used in strictly non-neural application spaces. In addition, Fugu significantly expands the population space of those who can benefit from neuromorphic computing. There are many potential contributors to Fugu, which we have grouped into the following three classes,

- A: Those without intimate knowledge of neural computing.
- B: Those who have experience with writing neural algorithms.
- C: Those with significant knowledge of neuromorphic hardware.

Our goal is to ensure that each type of user will be able to use Fugu with relative ease and contribute to it in any of the three manners. Fugu contains a library of different neural algorithms that are able to solve a variety of computations.

For example, several graph algorithms have been instantiated, such as shortest path and breadth first search. People in group B will be able to contribute to Fugu and implement other useful algorithms identified by the people in group A's needs. In addition to algorithms, Fugu also contains several different neuromorphic backends that will be able to run these algorithms efficiently, as they have been designed and optimized for neural algorithms. People in group C will be able to develop new backends as more neuromorphic hardware becomes available. The end result is a plethora of spiking neural algorithms for a multitude of application spaces with plenty of backend options.

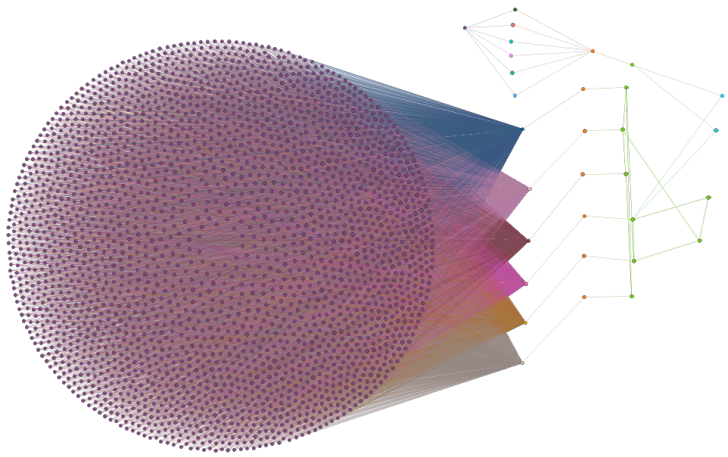


Fig. 2.1: An example of the underlying neural structure of a large graph computation with Fugu. Each different colored node in the graph corresponds to a neuron of a particular brick. In this instance, the purple neurons represent the input map, the orange represents the classification, green the graph search, and blue the constraint satisfaction.

2.1. Structure. Fugu itself is composed of two main parts: bricks and scaffolds. The idea is that bricks are simple computations that users may want to use as a part of a larger computation. In this sense, users can combine bricks to form a scaffold.

For example, say a user wanted to determine if a destination on a map is within a certain vehicle's range. To compute this with Fugu, the following bricks would be needed: an input of the map, classification on that image, a resulting graph search, and then a constraint satisfaction. Each of these bricks would then be combined into a scaffold and computed in the specified order. The resulting network of neurons created from this example simulation can be seen in Figure 2.1.

2.2. Features. Fugu has several features that make it especially distinct compared to other platforms that run neural algorithms efficiently. The main distinction is that Fugu can run non-neural applications in addition to many learning and neural applications. In Section 3 we present one non-neural application, a Markov process random walk. This exemplifies a class of computations that fit nicely in spiking neural algorithms.

Another Fugu feature, shown in Figure 2.2, is the debugging tool created for Fugu. The idea behind this tool is that when users are creating a brick, it is important for them to determine if their brick works. This debugging tool can be used to ensure that the spikes are happening when expected and the bricks are connected in a reasonable way. A difficult part of not only neuromorphic, but computing in general is the fact that debugging is difficult. As neuromorphic computing is not as widespread as traditional computing, there are not any best practices for debugging in place. We hope that this debugging visualization tool can be used effectively for any computation used on Fugu and can help set good practices for others in the neural network computing realm as well.

2.3. Goals. Eventually, we hope to have Fugu open to the general public. For now, it is being developed at Sandia National Laboratories with collaborators at Lawrence Livermore National Laboratory and Los Alamos National Laboratory. Our goal is to have Fugu be

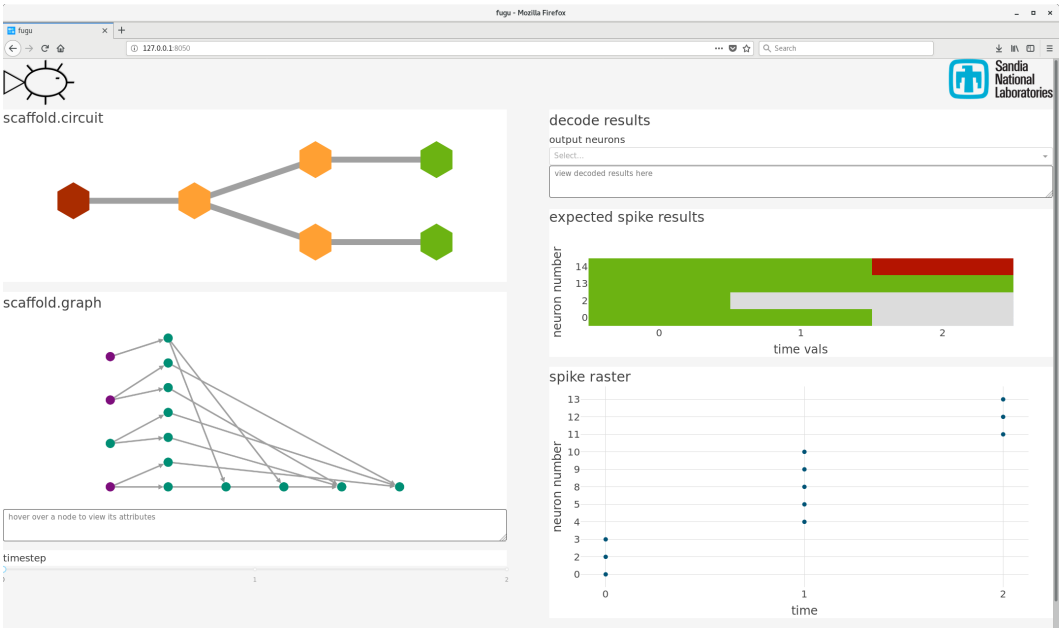


Fig. 2.2: An example of a Fugu scaffold computation in the built in debugging tool. In this example, the input is a vector. Then, that vector is copied and two different dot products are applied to one copy of the vector. Finally, the resulting value of the dot products are checked against two thresholds to determine if the resulting value is above or below each threshold.

open source so a variety of people can use it and contribute to it, which would enhance the neuromorphic community as a whole.

3. Random Walks and Diffusion. Here we present a scientific application, random walks, that has been shown to map well onto a neuromorphic system. A random walk models particles moving in a diffusion scheme, or randomly in a free space. In a simplified scheme, particles start at a location on a one dimensional line and have defined probabilities of moving one step to the right or left. If we let the particle take one step at each time step, over time the motion of the particle would result in a random walk over its domain. A depiction of a 1-D random walk can be seen in Figure 3.1. This can extend easily to two or more dimensions. For an in-depth introduction on random walks, see [15].

The motion of particles in a random walk inherently solves the well known diffusion equation partial differential equation (PDE) [14]. The diffusion equation, also referred to as the heat equation, models the phenomenon of particles spreading throughout a domain. This is a well known PDE that arises in many scientific computing and engineering applications.

3.1. Spiking Neural Algorithm. The spiking neural algorithm that we use for random walks was first presented in [13]. This algorithm tracks positions of random walkers in time by tracking the nodes on a graph and counting the number of walkers at each node at any given point in time.

To do this, there is a spiking neural circuit embedded at each node in the graph that randomly determines which direction the walkers will go. A random walker's direction is determined by a specific neuron spiking in a probability gate. At a given time step, if a walker is at a node, neurons will propagate and spike throughout the circuit at that node, ending at this probability gate. The output neuron that spikes in the probability gate specifies

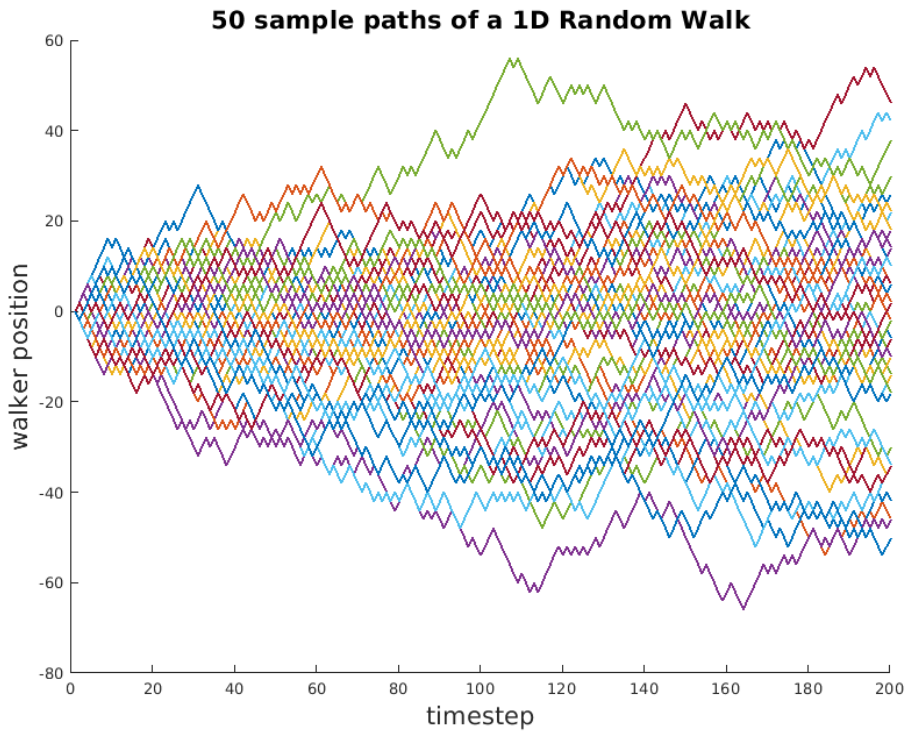


Fig. 3.1: One dimensional random walk simulation depicting 50 particles.

which neighboring node the current node will send a walker to. This neighboring node is where the walker will be at the next time step. These probabilities are all user-determined, so this algorithm can be ubiquitous over a variety of applications.

3.2. Applications. The most common applications of random walks are a variety of graph algorithms and image processing tools such as path finding and image segmentation. More details on these types of applications and how they were implemented as spiking neural algorithms can be found in [11]. Here we focus on several more general engineering applications—radiation transport and electrical capacitance. Each of these applications can be modeled with different modifications of a simple random walk.

3.2.1. Radiation Transport. Radiation transport is an important and relevant application as many scientists try to model particles being emitted from a radioactive slab. Solving this radiation transport problem with random walks is not new; it is common to use Monte Carlo approaces to solve these types of particle transport problems [5]. Here we focus specifically on a one dimensional model problem. For a one dimensional radiation transport scheme, we consider random movement throughout a slab, where spatial locations are only defined along one direction. Particles are either absorbed or reflected at random spatial intervals along the slab. This results in three random numbers to consider at each particle location: distance particle traveled, direction of particle movement, and rate at which the particle is absorbed.

We are working on mapping this to our spiking neural algorithm. The first inherent

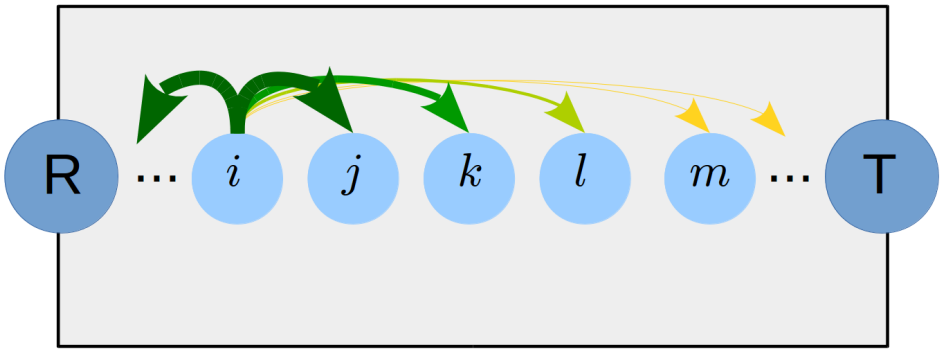


Fig. 3.2: A potential 1D radiation transport scheme. The slab of material contains nodes along the x direction. A neural circuit is embedded at each node. Walkers have a probability to move from one node to another with a certain probability determined by a double tailed exponential distribution. Arrows represent transitions from node i. Transitions with higher probabilities are denoted in green with a thicker arrow. Transitions with low probabilities are denoted in yellow with a thin arrow. Nodes R and T represent reflection and transmittance nodes respectively.

difference from a simple random walk lies in the fact that we have more random events occurring. As it is, our algorithm can deal with particles moving in an arbitrary number of directions, but we will need to instantiate other random factors on top of that, such as a random absorption event. We also have particles that are randomly diffracted and change directions rapidly. As we are considering a one-dimensional case, we can represent this by creating highly connected neural circuits, so more potential directions are available at a single instance.

This does not require changes to the algorithm itself, but changes to how it is instantiated by the user. To model the possible directions and transition probabilities needed for this application, the length of the slab needs to be discretized. We set the transition probabilities following a double tailed exponential distribution, where walkers at a location are much more likely to move to closer neighbors rather than positions located on the other side of the slab. Here it is important to ensure that probability is conserved over all of the possible directions. Figure 3.2 shows how the model can be set up, with thicker arrows corresponding to connections with higher probabilities.

In the algorithm, we are primarily concerned with particles being either reflected, transmitted, or absorbed. We can keep track of this throughout the random walk simulation because if a particle reaches beyond the left-most point of the domain, we can qualify this as the particle being reflected back from where it first entered the slab. If a particle reaches beyond the right-most point of the domain, then we can qualify this as the particle being fully transmitted throughout the slab. To quantify these positions in the actual simulation, we append two more possible locations to either end of the slab, one to simulate reflection and one to simulate transmittance.

Particles that are being absorbed fit more naturally to the original random walk model. To integrate absorption into the model, we create another random draw before spikes are sent to neighboring neurons. If we check the result of this draw to the specified probability of absorption, we can move spikes along if it does not meet the threshold and hold spikes if it does, signifying that the walker has been absorbed.

3.2.2. Electrical Capacitance. Using introductory electrostatic concepts, calculating the capacitance can be thought of as calculating the surface integral of the charge density of a capacitor. Let $S \in \mathbb{R}^3$ represent a conductor and ∂S be its surface in which we are calculating the charge density on. Additionally, let C represent the capacitance of S, and μ represent the charge density distribution. Then, capacitance can be found with the following equation,

$$C = \int_{\partial S} \mu(y) d\sigma(y). \tag{3.1}$$

To calculate the charge density $\mu(y)$ of a given object, a random walk on the surface of that object can be conducted. After n timesteps, the charge density can be calculated by determining the amount of particles at each position on the surface of the object [8].

In this random walk scheme, a particle starts at a particular location on the surface of a cube. The particle can then randomly move to another face of the cube with a random angle θ along the current face of the cube and random angle ϕ above the current face of the cube. This can be seen in Figure 3.3.

This general random walk model can be implemented quite easily with the neural algorithm that we already have; however, there will be a large bottleneck if we do not modify how the particles are connected. In the original scheme, each particle has the potential to move to any location on any remaining face of the surface. In our neural algorithm, this translates to building a network of neurons before the simulation begins and connecting a neural circuit at each node to circuits at all of the remaining nodes that can be reached. Although this can be done, our random walk algorithm will be very inefficient if we implement it at face value due to the significant number of highly connected neurons.

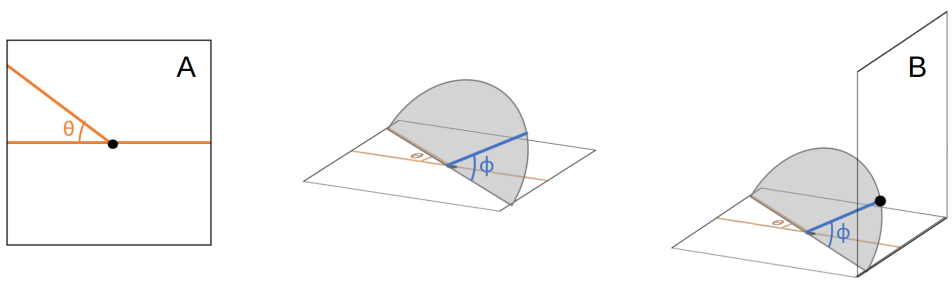


Fig. 3.3: A depiction of how the particles move in a random walk scheme on a surface of a cube. A particle begins at face A then moves to face B as a result of two random draws: one for the angle $\theta \in [0, 2\pi)$ and one for the angle $\phi \in [0, \pi]$.

Instead, we are exploring hierarchical connections of the nodes so we do not have to initialize a significant amount of neurons and connections that have a very low probability of being used. As we build our network of neurons before the simulations start, we need to prune the number of outcomes of walkers that arrive at each node so that there are a more reasonable number of connections at each node. This can be done in a variety of ways, but it is not clear just yet what the best implementation is. It is important when implementing these applications on neural platforms that we are able to adjust the algorithm so that it is able to work well throughout the application spaces and continually show performance benefits with this nontraditional method of computing.

4. Conclusion. We have shown that neuromorphic computing is a promising field with a significant number of unexplored research applications. Previous concerns regarding neuromorphic systems have accused it of being niche as it does not extend well to non-neural applications. However, we have discussed a variety of non-neural applications that show how neuromorphic can be used in addition to or instead of traditional architectures and paradigms. We hope to further this area of research and continue to develop useful algorithms that utilize the unique features of neuromorphic architectures to their fullest extent.

REFERENCES

[1] J. B. Aimone, W. Severa, and C. M. Vineyard, *Composing neural algorithms with fugu*, arXiv preprint arXiv:1905.12130, (2019).

[2] E. Alpaydin, *Introduction to Machine Learning*, The MIT press, 2nd ed., 2010.

[3] A. Calimera, E. Macii, and M. Poncino, *The human brain project and neuromorphic computing*, Functional neurology, 28 (2013), p. 191.

[4] J.-A. Carballo, W.-T. J. Chan, P. A. Gargini, A. B. Kahng, and S. Nath, *Itrs 2.0: Toward a re-framing of the semiconductor technology roadmap*, in 2014 IEEE 32nd International Conference on Computer Design (ICCD), IEEE, 2014, pp. 139–146.

[5] S. A. Dupree and S. K. Fraley, *A Monte Carlo primer: A Practical approach to radiation transport*, vol. 1, Springer Science & Business Media, 2002.

[6] A. J. Hill, J. W. Donaldson, F. H. Rothganger, C. M. Vineyard, D. R. Follett, P. L. Follett, M. R. Smith, S. J. Verzi, W. Severa, F. Wang, et al., *A spike-timing neuromorphic architecture*, in 2017 IEEE International Conference on Rebooting Computing (ICRC), IEEE, 2017, pp. 1–8.

[7] W. Maass, *Networks of spiking neurons: the third generation of neural network models*, Neural networks, 10 (1997), pp. 1659–1671.

[8] M. Mascagni and N. A. Simonov, *The random walk on the boundary method for calculating capacitance*, Journal of Computational Physics, 195 (2004), pp. 465–473.

[9] D. Monroe, *Neuromorphic computing gets ready for the (really) big time*, Communications of the ACM, 57 (2014), pp. 13–15.

[10] W. Olin-Ammentorp, K. Beckmann, C. D. Schuman, J. S. Plank, and N. C. Cady, *Stochasticity and robustness in spiking neural networks*, arXiv preprint arXiv:1906.02796, (2019).

[11] L. E. Reeder, A. J. Hill, J. B. Aimone, and W. M. Severa, *Exploring applications of random walks on spiking neural algorithms*, Center for Computing Research Summer Proceedings - Technical Report SAND2019-5093R, (2018), pp. 145–156.

[12] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, *A survey of neuromorphic computing and neural networks in hardware*, arXiv preprint arXiv:1705.06963, (2017).

[13] W. Severa, R. Lehoucq, O. Parekh, and J. B. Aimone, *Spiking neural algorithms for markov process random walk*, arXiv preprint arXiv:1805.00509, (2018).

[14] R. W. Shonkwiler and F. Mendivil, *Explorations in Monte Carlo Methods*, Springer Science & Business Media, 2009.

[15] L. Sjogren, *Chapter 2 : Random walks*, http://physics.gu.se/~frtbm/joomla/media/mydocs/LennartSjogren/kap2.pdf.

[16] M. M. Waldrop, *The chips are down for Moore's law*, Nature News, 530 (2016), p. 144.

PERFORMANCE MODELING OF VECTORIZED SNAP INTER-ATOMIC POTENTIALS ON CPU ARCHITECTURES

MARK P. BLANCO* AND KYUNGJOO KIM†

Abstract. SNAP potentials are inter-atomic potentials for molecular dynamics that enable simulations at accuracy levels comparable to density functional theory (DFT) at a fraction of the cost. As such, SNAP scales to simulation sizes on the order of $10^4 - 10^6$ atoms. In this work, we explore CPU optimization of potentials computation using SIMD. We note that efficient use of SIMD is non-obvious as the application features an irregular iteration space for various potential terms, necessitating use of SIMD across atoms in a cross matrix, batched fashion. We present a preliminary analytical model to determine the correct batch size for several CPU architectures across several vendors (Intel, IBM, and ARM), and show end-to-end speedups between 1.66x and 3.22x compared to the original.

1. Introduction. Spectral Neighbor Atom Potentials (SNAP) is a molecular dynamics approach that achieves similar accuracy to density functional theory (DFT) codes, while scaling to many more atoms at a fraction of the time [9]. SNAP achieves this by projecting inter-atomic interactions between a source atom and each of its neighbors into a basis space called the 'bispectrum.' The collection of inter-atomic bispectrum components are used as a feature space for a statistical model previously trained on high-accuracy reference data for a training set of atomic configurations. In this way SNAP scales to simulations with orders of magnitude more atoms (on the order of $10^5 - 10^6$ atoms).

Performance modeling of scientific applications plays a pivotal role in achieving efficient and portable execution of applications on diverse parallel computing platforms. As the variety of parallel computing architectures increases, establishing an accurate model becomes also more complicated, since a model must account for a variety of factors on performance. Furthermore, performance modeling of scientific applications is even more challenging as it requires domain-specific knowledge of both applications and computer architectures to accurately model their dynamics. For this reason, performance modeling tends to focus on characterizing the performance of a kernel in an algorithm on a given hardware system i.e., roofline models. Although roofline models can expose inherent hardware limits of kernels, they do not expose performance bottlenecks or design issues of the algorithm implementation itself. Hence, exploring and modeling the performance opportunities in an application, across platforms, is necessary to identify performance-portable expressions of workloads that can be analytically tuned for a range of platforms.

In this work, we focus on developing a single-core performance model to guide developers (or their compilers) in how to best implement SNAP and take advantage of SIMD hardware through data parallelism. First, we review the mathematics behind SNAP and analyze a baseline code written directly from the mathematical formulation. We identify critical operations in SNAP subroutines where most computations occur, and offer a batched execution model that enables better exploitation of data parallelism. Next, we design an initial performance model based on our data-parallel formulation that narrows the space of batch sizes that need be considered in deploying SNAP to other architectures. Finally, we evaluate our model and demonstrate performance improvements on Intel, IBM, and ARM processors, showing end-to-end speed-ups between 1.66x and 3.22x across all test cases.

2. SNAP, Initial Algorithm, and Profiling. In this section we briefly introduce relevant mathematical details of the SNAP formulation. We then review the current algorithmic formulation, our analysis of the iteration spaces present, and run-time profiling

*Carnegie Mellon University and Sandia National Labs, markb1@cmu.edu
†Sandia National Laboratories, kyukim@sandia.gov

results on the test implementation. These analyses and profiling results guide our optimization efforts by highlighting important trends in computational cost and opportunities for parallelism.

2.1. Overview of Mathematics for SNAP. For additional details on the SNAP formulation, we direct the reader to reference [3, 9]. A key part of the approach taken by Thompson et al. in the design of Spectral Neighbor Analysis Potentials is in mapping spatial relationships between pairs of atoms to a basis space called the bispectrum. As a result of mapping to this basis space, a large fraction of the computations in SNAP no longer scale in complexity with the number of neighbors to each atom, but instead to the size of the basis space used.

Wigner Matrices. To arrive at the new basis space, SNAP represents 3D spatial relationships between atom pairs in terms of two angular components and a radius. The three coordinates are mapped onto a point on the surface of the unit 3-sphere in 4D space. A set of hyper-spherical harmonic basis functions known as Wigner matrices are evaluated at the point. The Wigner matrices are complex-valued functions of three angles that we write here as $U^{j_{m,m'}}(\theta_0, \theta, \phi)$. Further details on this coordinate transformation can be found in [3, 9, 10]. For a given atom i and neighbor i', their relative polar coordinates map to the following partial term:

$$u^j_{m,m';\, partial(i,i')} = f_c(r_{ii'}) w_i U^j_{m,m'}(\theta_0, \theta, \phi) \tag{2.1}$$

where $f_c(r_{ii'})$ is a function designed to decrease the effects of neighboring atoms i' that are farther away, going smoothly to zero at some cutoff distance; w_i is a weighting for atom i; and the third polar angle is given by $\theta_0 = \theta_0^{max} \frac{r}{R_{cut}}$ for cut-off radius R_{cut} and cut-off angle θ_0. These components are accumulated for all atom neighbors i':

$$u^j_{m,m'} = U^j_{m,m'}(0,0,0) + \sum_{r_{ii'}} u^j_{m,m';\, partial(i,i')} \tag{2.2}$$

Repeated for each atom, we obtain the complex-valued expansion coefficients for each atom in the simulation. Throughout this text, this part of the computation is referred to as computation of u-terms, and in algorithm listings is part of Phase I of SNAP.

Bispectrum Components. Each atom i's component representation in the bispectrum is given by a combination of u-terms over a set of iteration labels j_1, j_2, j_3, and their corresponding m_n, m'_n sub-labels:

$$B_{j_1,j_2,j_3} = \sum_{m_1,m'_1=-j_1}^{j_1} \sum_{m_2,m'_2=-j_2}^{j_2} \sum_{m_3,m'_3=-j_3}^{j_3} (u^{j_3}_{m_3,m'_3})^* H^{j_3 m_3 m'_3}_{j_1 m_1 m'_1, j_2 m_2 m'_2} u^{j_1}_{m_1,m'_1} u^{j_2}_{m_2,m'_2} \tag{2.3}$$

The term $H^{j_3 m_3 m'_3}_{j_1 m_1 m'_1, j_2 m_2 m'_2}$ is a product of Clebsch-Gordan coefficients, defined by Thompson et al. in [9]. The iteration labels $j_1 = j_1, j_2, j_3$ are defined for $j_n = [0, 0.5, 1, 1.5, \ldots J_{max}]$ and the labels m_n, m'_n are defined over $[-j_n \ldots j_n]$ for a given value of j_n.

Note that the j_n iteration labels are are defined on a 'half-integral' space; for computational practicality these are scaled by two in order to be integral, and the m_n, m'_n labels are correspondingly scaled and translated to fall in the range $[0 \ldots 2 \times j_n]$. Finally, we note that $2J_{max} = 2 \times J_{max}$ determines how many bispectrum omponents SNAP will use in computations, which therefore scales accuracy and computational complexity. The j_n terms are constrained in the following manner:

$$\begin{aligned} 0 \le j_1 \le 2J_{max}, \\ 0 \le j_2 \le j_1, \\ j_1 - j_2 \le j_3 \le j_1 + j_2 s.t. j_1 + j_2 + j_3 \in 2\mathbb{Z} \end{aligned} \tag{2.4}$$

We direct the reader to [9, 10] for additional details regarding these constraints.

Gradients, Energies, and Forces. To compute inter-atomic energies, the bispectrum components for a given atom are linearly combined using pre-trained beta coefficients (β_{j_1,j_2,j_3}). For atomic forces, the gradients of the bispectrum components with respect to each neighbor, by way of $r_{ii'}$, are linearly combined with the same coefficients.

Computing Partial Bispectrum Components. As shown by Thompson et al., there are shared terms between the bispectrum components (B) and the gradients of the bispectrum components. Therefore, they define a set of Z- and Y-terms for each atom i, where the Z-terms are defined as a portion of the bispectrum components:

$$Z^{m_3,m_3'}_{j_1,j_2,j_3} = \sum_{m_1,m_1'=-j_1}^{j_1} \sum_{m_2,m_2'=-j_2}^{j_2} H^{j_3 m_3 m_3'}_{j_1 m_1 m_1', j_2 m_2 m_2'} u^{j_1}_{m_1,m_1'} u^{j_2}_{m_2,m_2'}. \tag{2.5}$$

The Y terms are computed as the product sum over j_1 and j_2 of $Z^{m_3,m_3'}_{j_1,j_2,j_3}$ terms and their β_{j_1,j_2,j_3} coefficients:

$$Y^{m_3,m_3'}_{j_3} = \sum_{j_1} \sum_{j_2} Z^{m_3,m_3'}_{j_1,j_2,j_3} \beta_{j_1,j_2,j_3} \tag{2.6}$$

By computing and storing Y terms, both the inter-atomic energies and their update forces can share repeated computation effort and reduce computational load. In Algorithm 4, computation of partial bispectrum components (Y-terms) is performed in Phase II, and computation of gradients and force terms occurs in Phase III.

Next, we analyze the performance of the baseline implementation of the mathematics above, highlighting the three distinct phases of SNAP computation: computing expansion coefficients (u-terms), computing partial Y-terms in the bispectrum, and finally computing gradients for use in force update calculations.

2.2. Initial Performance Analysis. The baseline algorithmic formulation of SNAP is shown in Algorithm 4. Our initial analysis of SNAP is concerned with identifying where the most time is spent in SNAP, and how data-parallelism is currently extracted in the baseline. The algorithm listing highlights via comments the three main phases of computation corresponding to mathematical operations described in the previous section. These phases are distinct from each other because of the different iteration spaces used in computing terms for each phase.

To identify the primary phases of computation, we profiled the baseline code using `gprof` and `Intel vTune`. Reference data sets for the cases $2J_{max} = 8$ and $2J_{max} = 14$ were used to verify that the calculations were correct, modulo differences due to order-of-operation effects. These two cases give a realistic number of bispectrum components for SNAP simulations and provide good accuracy in simulations [9]. For both tests, the number of atoms was 2000 and the numbers of bispectrum components to be computed were 55 and 204, respectively.

Multi-platform Profiling with GNU gprof. Single-threaded `gprof` profiling runs were done on the Mayer, Blake, and White testbeds, which have ARM Cavium ThunderX2, Intel Skylake-X, and IBM Power8 CPUs, respectively. Codes were compiled with g++ 7.2 and level two optimizations, along with architecture-specific flags. SNAP was run for 10 iterations in each of the two test cases described above. The results of these profiling runs are shown in Figure 2.1. Depending on the number of bispectrum components used as determined by $2J_{max}$, computation of Y-terms (Phase II) and computation of partial

Algorithm 4 High-level of SNAP Algorithm Implementation in [9]

Inputs: Initial atom forces and positions, set of atoms, neighbors, inter-atom distances, beta coefficients, and atom weights
Output: Updated forces and positions of atoms
initialize CG coefficients
set $iterations = 0$
while $iterations < LIMIT$ **do**
  for atom $a_i \in$ atoms **do**
    populate a_i neighbor information from reference data
    ▷ // Phase 1: u-terms
    for neighbor $a_{i'} \in neighborhood(a_i)$ **do**
      compute $u_{i,i'}$ expansion coefficients
      reduce $u_{i,i'}$ terms to total u_array
    ▷ // Phase 2: Y-terms
    for $j_3 \in$ iteration space defined by $2j_{max}$ **do**
      compute $Y^i_{j_3}$
    ▷ // Phase 3: gradients and forces
    for neighbor $a_{i'} \in neighborhood(a_i)$ **do**
      compute $\frac{\delta u_i}{\delta r_{i'}}$
      compute $\frac{\delta E_i}{\delta r_{i'}}$
      Update force components on atoms i and i'
  $iterations += 1$

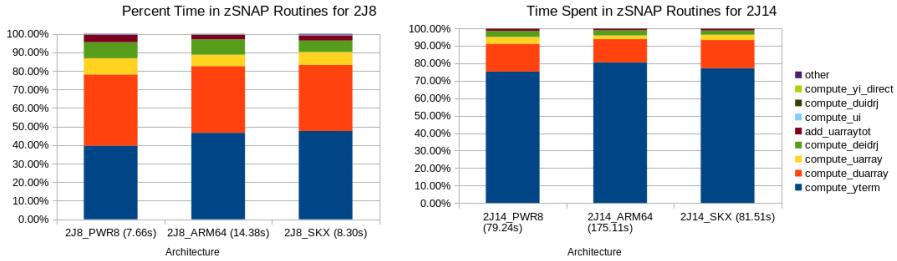


Fig. 2.1: Proportion of time spent in SNAP routines across three architectures. The total time to execute the 10-iteration run for each architecture is shown in the x-axis label, in seconds.

derivatives (Phase III) trade places as the most computationally expensive. In the figure legend, `compute_yi_direct` and `compute_yterm` correspond to computation in Phase II, while `compute_deidrj`, `compute_duidrj`, and `compute_duarray` correspond to computations in Phase III. All other subroutines listed correspond to Phase I, or 'other', which represents one-time initialization of data structures and constant coefficients.

For the smaller case of $2J_{max} = 8$, these two phases are very close in runtime. When $2J_{max} = 14$, the most expensive function across all tests and architectures is computation of Y-terms, which must also compute Z-terms even if they are not stored. As expected, when $2J_{max}$ is increased, the proportion of time spent in this routine increases since the number of Z-terms increases much faster than any other compute-space characteristic of the workload. For visual reference, Fig. 2.2 shows how the number of u-terms (ucount),

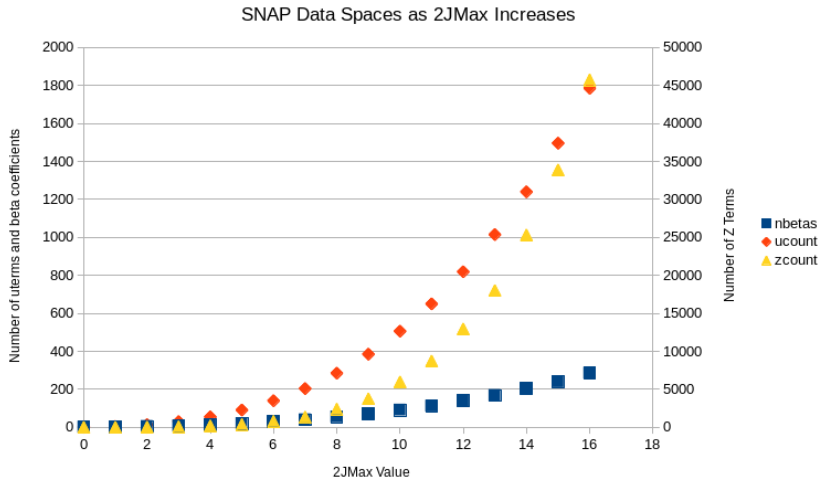


Fig. 2.2: The number of bispectrum components (B_{j_1,j_2,j_3}) has the lowest scaling rate w.r.t $2J_{max}$. The number of intermediate $u^j_{mm'}$ and $Z^j_{mm'}$ terms scale much more rapidly. Note that the number of Z terms is on the second y axis, as otherwise the other two scaling trends would be unreadable.

Z-terms (zcount), and bispectrum components (nbetas) scales as $2J_{max}$ is increased.

Examining Autovectorization with Intel VTune. The opportunity for performance through data-parallel hardware is especially present on the Intel platform, where 512-bit SIMD vectors are supported. To explore how well this resource was being utilized by ICPC, we examined code on the Intel Skylake-X (SKX) platform using VTune. Examination in one of the inner-most loop for computation of Y-terms revealed limited vectorization. Listing 2, lifted from the compiled code, shows this use of SIMD as generated by the compiler. In this instance, it can be seen that the vectorization is being done across values that must eventually be reduced to one complex number. This means that the real (imaginary) values that are side-by-side in the %ymm6 (%ymm1) SIMD register must be reduced horizontally. This is expensive because there are no instructions for adding all values in one SIMD register together. Instead, the top upper halves of the original SIMD registers must be moved to temporaries (lines 1 and 2), which can then be added to the unmoved lower halves (lines 3 and 4). This process continues until all items in the original two real and imaginary registers have been reduced to two scalar components (lines 5-10). Such use of SIMD has high overhead because the majority of the operations are either vector packing and unpacking operations, or vector additions of increasingly narrow width compared to the full width offered on the platform.

```
1  vextractf128 $0x1, %ymm6, %xmm10
2  vextractf128 $0x1, %ymm1, %xmm14
3  vaddpd %xmm10, %xmm6, %xmm11
4  vaddpd %xmm14, %xmm1, %xmm15
5  vunpckhpd %xmm11, %xmm11, %xmm12
6  vunpckhpd %xmm15, %xmm15, %xmm16
7  vaddsd %xmm12, %xmm11, %xmm13
8  vaddsd %xmm16, %xmm15, %xmm17
9  vaddsd %xmm7, %xmm13, %xmm7
10 vaddsd %xmm9, %xmm17, %xmm9
```

Listing 2: Horizontal add along registers compiled by ICPC

Additional analysis by VTune for both the 2J8 and 2J14 cases indicate that accesses to main memory are not a bottleneck, and even that accesses to cache are a performance limitation in only small percent of the runtime. This is consistent with estimates of working set size based on Fig. 2.2; since Z-terms are only computed in-flight, most of the memory required is for u-terms, Y-terms, and B-terms.

Iteration Spaces. Between the three phases identified in SNAP, there are several iteration spaces that offer the potential for data parallelism. These include the outer loop over all atoms, loops over neighbors, and loops over iteration spaces for computing Y-terms and gradient terms.

In Phase I, u-terms are computed across iterations for each atom, for each neighbor, and for each labeling (j, m, m'). Within the innermost loop over (j, m, m') labels, there exists a dependency between values. Therefore we do not consider this space further for data parallelism. On the other hand, vectorizing across neighbors or across multiple atoms could offer viable iteration spaces for data-parallel computation. This is similar to the approach taken by Kim et al. in computing small matrix multiplications in batches [6]. Assuming a fixed or minimum number of neighbors per atom, cleanup code would only be necessary for a number of neighbors not evenly divisible by the batch size. Similarly, batching across atoms would require cleanup code only when the number atoms is not evenly divisible.

The iteration space over (j_1, j_2, j_3) for computation of partial Y-terms is complex due to the conditions an outer loop's iterator label places on the inner labels (See Eqn.2.4). Vectorizing over the (j_1, j_2, j_3) iteration space would be difficult in practice as it requires complex control code. On wide-vector architectures such as Intel, it would be difficult to fill the vector registers, leaving significant performance potential unrealized. However, in contrast to the case of computing u-terms where atoms may have a different number of neighbors, computing Y-terms has exactly the same iteration space across all atoms. Thus, by combining Y-term computation across multiple atoms in a data-parallel fashion, the cost of looping machinery to iterate over the complex space can be amortized and regular SIMD operation now becomes possible.

Finally, the computation of gradient terms in Phase III follows a similar iteration space to that of computing u-terms themselves. Hence, data-parallelism across atoms or neighbors are both possible for this phase.

In our profile analysis of SNAP, we identified that computation of Y-terms in Phase II is one of the most expensive portions across both test scenarios. Therefore, we focus on data parallel operation across multiple atoms for all three phases. We do this to avoid re-packing data between phases, since data-parallel operation across neighbors or across atoms require different changes to storage of computed values. In doing this, we can pursue improvement and analytical modeling for Phase II of SNAP while at the same time using the same data-parallel strategy to extract SIMD performance from the rest of the application.

3. Batched Approach and Analytical Model for SNAP. This section describes the end-to-end data-parallel algorithm for SNAP, arranged for batched execution. This work organization is intended to leverage SIMD hardware across multiple cores on CPUs.

3.1. Performance Principles. Vector based computation, and in general single program multiple data (SIMD) hardware, require several conditions to be met to achieve their peak possible performance. These conditions follow naturally from principles developed around scalar programs, but can nonetheless be unintuitive or result in code that looks very different from scalar scientific computing code.

Independent Operations. The first performance principle is that of filling functional unit pipelines with *independent* operations. To make use of this principle, we define concepts

of *latency* and *throughput* in a processor. In this work, we view throughput as the number of cycles that must pass before another independent operation of a particular type can be issued to a functional unit following issue of a prior independent operation. This definition fits with Fog's definition in his x86 Instruction Tables for 'reciprocal throughput' [4]. In contrast, latency is *not* the reciprocal of throughput. For our purposes, we take latency to be the number of CPU cycles between issue of one instruction, and issue of the next instruction of that type with a read-after-write dependency from the first to the second.

Based on principles of throughput and latency, more efficient use of SIMD hardware on the test platform (SKX) should be possible over the horizontal add shown in the assembly block. In Listing 2, two AVX-256 bit (ymm) registers containing four double values have resulted from vectorization on the actual computations to be performed in computing Y-terms. However, because computation is being performed for a *complex* Y-term, there are two independent chains of operations: the horizontal addition of partial imaginary and real Y-term components. The upper half of each ymm register is moved to a the lower half of a different vector register, denoted as xmm. The upper and lower halves are then summed, and the result placed into a 128-bit xmm register. The horizontal addition is repeated so that half of the xmm registers are moved and added, giving the sum of four partial terms from each of the two initial (real and imaginary) ymm registers.

Note that the above process of horizontally adding values in vector registers represents overhead incurred in order to use SIMD hardware. Again, note that there are only two independent summations being performed at a time in `compute_yterms`. Furthermore, if the code is not vectorized, that only the real and imaginary portions of a single Y term are being computed limits used pipeline stages of scalar floating-point operations to two as well. On many systems, the latency of a floating-point addition or multiplication is more than two cycles, leaving bubbles in the pipelines [4]. Therefore, while the code is originally written with scalar execution in mind and would perform adequately in that mode, it is very difficult for the compiler to fully leverage the throughput offered by SIMD, or even scalar hardware, as-is.

Vectorization and Data Access. The second performance principle, relevant to both scalar and SIMD computation, is that of data organization. Modern computing systems are designed with multiple levels of memory. Starting from the processor and moving farther away, the speed of the memory decreases, both in latency and bandwidth, while the overall capacity increases. This encourages that both scalar and SIMD programs be designed to place and retain data in closer cache levels. This also means that it is best to access data in a contiguous fashion, such that the next piece of data to be accessed will be one next to or very close to the last piece accessed.

For scalar programs, this improves spatial locality, temporal-reuse of cache lines, and allows pre-fetchers to be effective in bringing data to nearer cache levels by the time it is needed. For SIMD code, contiguous accesses are vital to ensure efficient use of the memory subsystem and load/store functional units. While gather-scatter instructions for long-vector architectures such as Intel Skylake-X do exist, these dramatically increase the cost of loading data into a vector register and storing to memory. Therefore, data for each SIMD vector of data to be processed must be arrayed contiguously in memory to ensure packed loads and stores.

Batching for Increased Throughput. The performance principles outlined above must be satisfied by a SNAP re-written to effectively use SIMD hardware. To achieve higher SIMD performance compared to the current results produced by the compiler, more independent operations need to be introduced so that the functional units can fill their execution pipelines and operate closer to peak throughput rather than being latency-bound. In ad-

dition, vectorization requires contiguous data access for each vector register to avoid being load- or store-bound. In the previous section, we noted that the iteration space in SNAP most amenable to SIMD execution is over independent source atoms. This strategy fits with the first performance principle: each additional atom introduces another independent operation across all compute phases in SNAP. In this way, atoms can be computed across the entirely of SNAP in *batches*. Batching atoms also accommodates the second performance principle: input and output data for batched atoms can be placed contiguously, thereby enabling efficient SIMD loads and stores. The end-to-end algorithm with batching is shown in Algorithm 5.

Algorithm 5 End-to-End Batched SNAP Algorithm

```
initialize cg_list(), root_pq(), u_indices(), z_indices()
for atom_batch ∈ num_atoms do
    u_BATCH(:,:,:), u_BATCH_TOT(:,:,:), y_BATCH(:,:) ← 0
    let n(i) be the neighbors of atom i
    ▷ Phase One:
    for j ∈ [0,num_neighbors) do
        for jju ∈ u_indices do
        ▷ Note: jju has loop-carried dependence
        ▷ vectorized on CPU:
            for i ∈ atom_batch do
                compute u_BATCH(j, jju, i)
                u_BATCH_TOT(jju, i) += u(j, jju, i)
    ▷ Phase Two:
    for jjz ∈ z_indices do
    ▷ vectorized on CPU:
        for i ∈ atom_batch do
            compute y_BATCH(jjz, i)
    ▷ Phase Three:
    for neighbor j ∈ num_neighbors do
    ▷ vectorized on CPU:
        for i ∈ atom_batch do
            compute_duidrj(j, i)
            compute_deidrj(j, i)
            update forces for atoms i and j
            update energies for atoms i and j
```

3.2. Analytical Model. An analytical model is a system of rules for implementation of an HPC workload that select program parameters, such as cache blocking or batch sizes, across a range of targeted architectures. This approach differs from auto-tuning and statistical inference models in several important ways. First, while all three approaches are in theory capable of selecting program parameters for a given architecture, each new architecture will require a new search of the implementation space on the part of an auto-tuner. Second, a machine learning model, like an analytical model, should generalize to new architectures. However, a machine learning model may not provide an explanation or clarity for why certain parameters are selected. In contrast, an analytical model is defined on an high-level machine model, enabling it to generalize to similar future architectures and provide explanations of the performance achieved.

In this section, we apply the performance principles in Section 3.1 to batched SNAP to develop an initial analytical model that delineates a performant range of atom batch sizes to be processed across the entirety of the application. Note that the number of atoms is being selected, but in our current end-to-end batched implementation we focus on auto-vectorizable code. We make this decision in order to make the model and code easier to read and maintain, while recognizing that hand-written kernels based on SIMD intrinsics may perform better.

The performance model is defined over several inequalities based on the target architecture and relevant problem parameters for SNAP. The architectural parameters are the following:

- V_{LEN} - target vector length
- V_{LAT} - latency of FMA or ADD and MUL vector operations
- V_{THP} - throughput of FMA or ADD and MUL vector operations
- C_{LIM} - size capacity of the limiting cache level
- E_{SZ} - the size of a double-precision floating point element

In this work, we assume fused-multiply-accumulate (FMA) to be the bottleneck compute operation. Finally, for the capacity limit C_{LIM}, we select on each system either the size of a core's slice of the shared level cache, or the largest private cache it has available. Selecting the largest cache that reasonably will not incur capacity contention from other cores is especially relevant for SNAP operating in thread-parallel mode, as threads will all contend for shared cache resources. Additionally, limiting atom batch size to avoid heavy use of further caches stands to reduce the latency penalties incurred by the processor.

The relevant SNAP problem parameters are $2J_{max}$ and n_{max}. $2J_{max}$ affects data sizes including the number of Clebsch-Gordan coefficients, u-terms, and Y-terms. $nmax$ represents the number of neighbors per atom, which we assume is constant and is set to 26 in our tests.

We define our performance model by a lower limit based on pipeline filling, and upper-bounded by the working set size of Y-term computation. The lower limit is based on the principles for latency and throughput and the work of Low et al. [7] as follows:

$$S_{batch} \geq \frac{V_{LEN} \times V_{LAT} \times V_{THP}}{2} \tag{3.1}$$

The terms on the R.H.S. indicate the architectural parameters necessary to fill processor pipelines with useful work at every cycle. The factor of two is included because the most important phases of computation in SNAP (computation of Y-terms and gradient terms) involve complex numbers, which thereby already provide two independent operation chains per atom being computed. Inequality, rather than strict equality, is used because a larger batch size may be beneficial. A larger batch may prolong the CPU's high-throughput steady-state between filling and draining pipelines.

Table 3.1: Input and output data arrays for each phase of SNAP computation. Arrays of real and imaginary values are sub-scripted with r and i, respectively.

Phase	Inputs	Outputs
u-terms (I)	$r_{ij}, w_j, r_{ij,cut}, root_{pq}$	$ulist_r, ulist_i,$ $ulist_{r\_tot}, ulist_{i\_tot}$
Y-terms (II)	$ulist_{r\_tot}, ulist_{i\_tot},$ $betalist, cglist$	$ylist_r, ylist_i$
$\frac{\delta u_i}{\delta r_j}$-terms (III)	$root_{pq}, ulist_r, ulist_i$	$dulist_r, dulist_i$

Table 3.2: Data array sizes for batched SNAP.

Array Name	Array Elements
r_{ij}	$3 \times S_{batch} \times n_{max}$
n_{ij}	$S_{batch} \times n_{max}$
w_j	$S_{batch} \times n_{max}$
$r_{ij;cut}$	$S_{batch} \times n_{max}$
$root_{pq}$	$(2j_{max}+2)^2$
$cglist$	$(2j_{max}+1)^3$
$beta$	$num_{Z-terms} \times 6$ *
$ulist_{[i\|r]}$	$S_{batch} \times n_{max} \times num_{u-terms}$
$ulist_{[i\|r]\_tot}$	$S_{batch} \times num_{u-terms}$
$dulist_{[i\|r]}$	$S_{batch} \times num_{u-terms}$
$ylist_{[i\|r]}$	$S_{batch} \times num_{u-terms}$

The second inequality is from the working set size of the second phase in SNAP: computation of Y-terms. While other phases of SNAP, especially computation of gradient terms, do impact overall runtime, we see the most gains in the Y-term computation phase and therefore focus on this phase's working set size. Especially for $2J_{max} = 14$, computation of the Y-terms dominates that of the gradient terms and all other less costly phases. The working set size for Y-terms is based on the input and output data arrays used, shown in Table 3.1. Table 3.2 gives the dimensions of each data array for a given batch in SNAP.

The working set size for Phase II is defined as:

$$\begin{aligned} WSS_{Phase\ II} = E_{SZ} \times [(2J_{max}+1)^3 + num_{u-terms} \times \\ S_{batch} \times 4 + num_{Z-terms} \times 6] \end{aligned} \tag{3.2}$$

From this, we have an upper bound:

$$C_{LIM} \geq E_{SZ} \times [(2J_{max}+1)^3 + num_{u-terms} \times S_{batch} \times 4 + num_{Z-terms} \times 6] \tag{3.3}$$

Rearranging terms, we obtain: The working set size for Phase II is defined as:

$$S_{batch} \leq \frac{C_{LIM}/E_{SZ} - (2J_{max}+1)^3 - num_{Z-terms} \times 6}{num_{u-terms} \times 4} \tag{3.4}$$

Additionally, it is desirable to set S_{batch} such that it is an integer multiple of V_{LEN} so that masked SIMD operations and partial SIMD use be avoided.

Given Eqns. 3.1 and 3.4, upper and lower limits on the batch sizes can be determined for $2J_{max} = 8, 14$. Note that the upper and lower limits are based on the computationally expensive Y-term computations. Therefore, refining understanding of how atom batch size affects other SNAP phases is a direction for future work.

4. Results and Analysis. We evaluated SNAP on CPU architectures from three different vendors. The Intel platform hosts a Skylake-X Xeon Platinum 8160 CPU running at 2.1 GHz with 24 cores. The ARM platform is based on 28-core Cavium ThunderX2 CPUs clocked to 2 GHz. The IBM system is based on the Power8 architecture,

*The size of this array is tied to the number of Z-terms and not the number of beta-terms because the current implementation stores beta coefficients in an array of structs, also containing indexing information for Y-term computations. The factor of six corresponds to the number of 64-bit elements in each struct. Other indexing structures are not included in this assessment of working set size since they are not accessed as frequently on the innermost loops.

has 8 cores, and has 8-way SMT enabled. SNAP batched code was compiled with GCC 7.2 on all systems. We enabled architecture-specific flags and also used `-O3 --param vect-max-version-for-alias-checks=100 -ftree-vectorize -fopenmp -ffp-contract=fast` for all targets. In all tests, batched SNAP ran reference problems for $2J_{max} = 8$ and 14 with 2000 atoms. The outputs of each run were checked for correctness against reference forces. The latency-throughput product gives a lower bound for atom batch size in Y-term computation. The upper bound on batch size is given by the cache capacity limit. For $2J_{max} = 8$, the limit is based on the L2 cache since a unit batch size initially fits within the L2 cache. For $2J_{max} = 14$, the capacity limit used is the L3 cache slice size attached to a single core. However, in cases where the capacity bound is lower than the latency-throughput lower bound, we set that upper limit equal to the lower limit.

Fig. 4.1: Runtime Plots for batched SNAP with $2J_{max} = \{8, 14\}$ on Intel SKX.

In the performance plots, the per-phase and total runtime of SNAP is plotted on the left-hand vertical axis against a range of atom batch sizes along the horizontal axis. Superimposed on the performance bars are the lower and upper batch size bounds shown as vertical lines in red and blue respectively. The Y-term working set size and the L2 and L3 slice cache capacities are plotted against the right-side vertical axis, so that the cross-over points of WSS with respect to cache capacities can be seen. Note that SNAP is fully batched in the performance plots below. Therefore, changes in the atom batch size affect runtime of all three phases of computation.

The performance results for Intel Skylake-X on Blake reflect the tradeoffs between increasing batch size to fill FMA pipelines and increasing the working set size past cache

capacity limits (Fig. 4.1). Using a latency of 4, throughput of 2 vectors, and vector size of 8 atoms, the predicted minimum batch size is 32 [4]. Runtime for Y-term computation improves as the atom batch size approaches the predicted lower bound, and for $2J_{max} = 8$, the best performance for Y-term computation is achieved here (5.19x over the original). For $2J_{max} = 14$, the performance clearly improves further up to a batch of 64 atoms, beyond which runtime for Y-terms increases (3.19x speedup). End-to-end, performance improvements for the entire batched SNAP track similarly to the Y-term computation for $2J_{max} = 14$, but in the smaller test case the decrease in performance is due more to Phases I and III losing performance as their working set sizes increase. The number of Y-terms scales with the number of bispectrum components, so overall performance of the larger case is more strongly affected by performance changes in computation of Y-terms than in the other phases. Additionally, the working set sizes for Phases I and III are larger than that of Phase II. In the larger test case, the best performance is achieved for a batch size beyond the capacity upper bound. These dynamics suggest that further work in modeling the cache hierarchy and other phases of SNAP is necessary to understand SNAP on Intel processors.

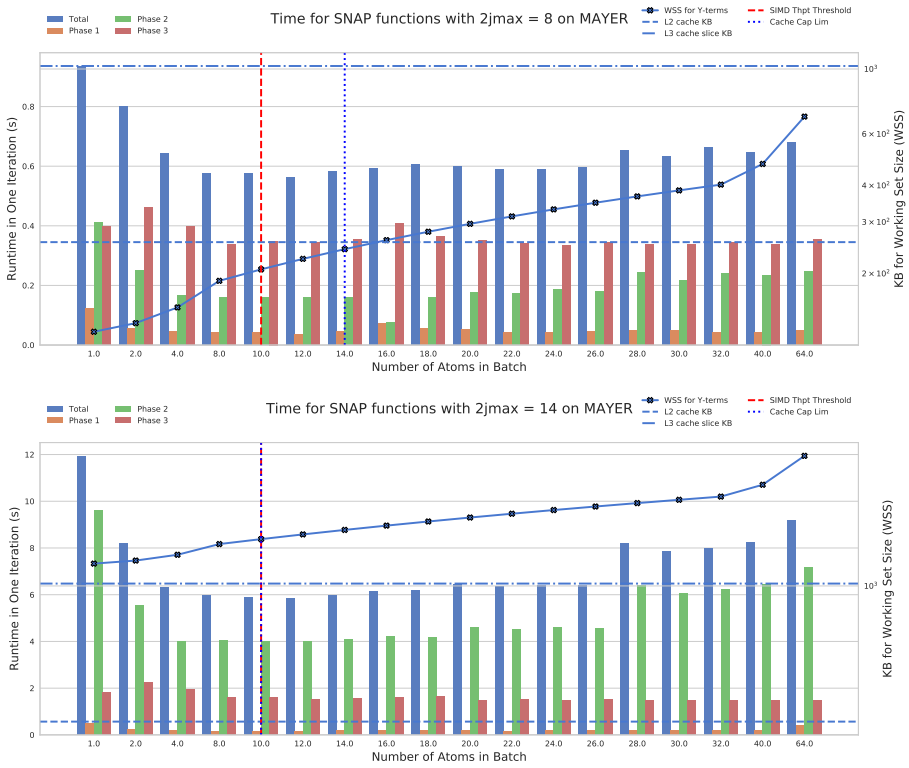


Fig. 4.2: Runtime Plots for batched SNAP with $2J_{max} = \{8, 14\}$ on the ARM Cavium ThunderX2 processor.

For the ARM-architecture ThunderX2 processor on Mayer, performance improvements follow a similar progression as on the Intel platform (Fig. 4.2). For Mayer, we estimate the latency-throughput product based on similar ARMv8.1 processors: 5 cycles for vector FMAs, two vector units, and two elements per vector gives a minimum of 10 elements per batch for peak throughput [1]. For $2J_{max} = 8$, improvement in computation of Y-terms appears to plateau in a range starting near the lower limit of 10 atoms. However, the

best performance is actually achieved for 16 atoms in an unexpected dip in runtime (5.49x speedup). This batch size is just past the cache capacity upper bound, suggesting that upper bound is too conservative. For the larger $2J_{max}$ case, the best performance is at the lower bound (2.4x speedup on Phase II). For this case, the computed upper bound was lower than the minimum bound, and was fixed to be equal to 10. Of note, the performance of Phase III improves to some extent with increased batch size, but does not appear to be as sensitive to batched computation compared to Phase II on this system. The performance results on this system suggest once again that further work in determining the true upper bound is needed for this system. The performance dip at 16 atoms for the smaller test case hints at another architectural aspect to study further.

Fig. 4.3: Runtime Plots for batched SNAP with $2J_{max} = \{8, 14\}$ on IBM Power8.

The IBM Power8 system (White) has a lower performance bound of 14 atoms per batch (7 cycles, 2 vector units, 2 elements per vector) [8, 2]. As expected, the best performance improvements are visible in Fig. 4.3 after this point. However, White exhibits the least negative sensitivity to large atom batches (Fig. 4.3). In fact, the best speedup for Phase II on the smaller test case is seen at 20 atoms, while the best performance for the larger case is at a larger batch of 28 atoms. For these batch sizes and test cases, the speedups are 2.74x and 2.14x, respectively. It may be that having a larger batch, coupled with the larger register file of the Power8 chip, allows more operations and therefore memory requests to be in flight, thereby hiding the latency of accesses to more distant caches. Nonetheless, further study of the memory system on this platform is needed. Additionally, the dip in performance from two to four atoms in a batch is unexpected. To understand this, further

work on understanding the pipelines in the Power8 processor should be done.

Overall, applying the batched computation model affords good speedups to SNAP. Our initial performance model can designate the start of a good region for batch sizes, but often cuts the region short when a larger batch may yield better performance.

5. Conclusions and Future Work. In this work, we examined existing code for Spectral Neighbor Analysis Potentials (SNAP), a recently developed molecular dynamics formulation that offers near-DFT accuracy with lower computational cost scaling. We identified the most expensive computations of the application, namely the computation of Y-terms, followed by the computation of gradient terms. Between the two accuracy levels normally used in SNAP computations ($2J_{max} = 8, 14$), we see that computation of Y-terms dominates gradient computation for the larger case. Focusing our performance analysis on computation of Y-terms, we identified that data-parallel hardware is more effective when atoms are computed in batches. When compiled with auto-vectorization, Y-term computation is accelerated between 2.14 and 5.49 times, depending on the host architecture and the number of bispectrum components being based on $2J_{max}$. When batching is applied to the entire application, SNAP as a whole is sped up between 1.66 and 3.22 times, before any thread-parallelism is applied.

The primary direction for future work is to refine our understanding of SNAP from a performance modeling perspective. The latency-throughput product gives a relatively accurate lower-bound on the atom batch size that should be selected for best performance. However, our suggested bound based on cache capacity is often too conservative and does not fully capture performance scaling of Y-term computation. Additionally, the achieved performances on each platform are lower than the computational peaks for SIMD Fused-Multiply-Accumulate would predict. These limitations indicate that some other performance choke-points must be identified and modeled to understand the performance characteristics of SNAP across architectures.

While we applied data-parallelism to only one iteration space (across atoms), future work could take advantage of multiple sources of data parallelism, including work across neighbors, similar to the approach taken by Höhnerbach et al. on Tersoff atomic potentials [5]. Finally, batching is a method that we expect to map well to thread groups on GPU hardware in addition to the SIMD hardware we have examined. Deriving a performance-portable analytical model for SNAP that applies to both CPU and GPU hardware is a promising direction for future work.

6. Acknowledgements. We would like to thank Aidan Thompson for detailed and informative discussions on SNAP mathematics and the reference code. We also thank Simon Hammond for sharing his time and expertise in architectural details and performance benchmarking for this project.

REFERENCES

[1] *ARM cortex-a series programmer's guide for ARMv8-a*, p. 296. https://developer.arm.com/documentation/den0024/a/preface.

[2] *POWER8 processor user's manual for the single-chip module*. https://openpowerfoundation.org/?resource_lib=power8-processor-users-manual.

[3] A. P. Bartók, M. C. Payne, R. Kondor, and G. Csányi, *Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons*, Phys. Rev. Lett., 104 (2010), p. 136403.

[4] A. Fog, *Lists of instruction latencies, thoughputs, and micro-operation breakdowns for intel, AMD, and VIA CPUs*.

[5] M. Höhnerbach, A. E. Ismail, and P. Bientinesi, *The vectorization of the tersoff multi-body potential: An exercise in performance portability*, in Proceedings of the International Conference for

High Performance Computing, Networking, Storage and Analysis, SC '16, IEEE Press, pp. 7:1–7:13. event-place: Salt Lake City, Utah.

[6] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, *Designing vector-friendly compact BLAS and LAPACK kernels*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, pp. 55:1–55:12. event-place: Denver, Colorado.

[7] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, *Analytical modeling is enough for high-performance BLIS*, ACM Trans. Math. Softw., 43, pp. 12:1–12:18.

[8] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, *IBM POWER8 processor core microarchitecture*, IBM Journal of Research and Development, 59, pp. 2:1–2:21.

[9] A. P. Thompson, L. P. Swiler, C. R. Trott, S. M. Foiles, and G. J. Tucker, *Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials*, Journal of Computational Physics, 285, pp. 316–330.

[10] D. Varshalovich, A. Moskalev, and V. Khersonskii, *Quantum Theory Of Angular Momentum*, World Scientific, 1988.

MODULAR WEB APPLICATION DESIGN FOR THE MANAGEMENT, VISUALIZATION, AND ANALYSIS OF NUCLEAR DETECTOR CHARACTERIZATION AND INVENTORY DATA

MARTON DEMETER[†], BORIS KUDRYAVTSEV[‡], BELKIS CABRERA-PALMER[§], AND MATTHEW H. WONG[¶]

Abstract. The Instrument Characterization Catalog (CharCat) is an online reference tool to be used by the Department of Homeland Security and Countering Weapons of Mass Destruction (CWMD) test scientists as part of the Data Mining, Analysis, and Modeling Cell (DMAMC) tool suite. The purpose of the web application is to manage, visualize, and analyze characterization and inventory data for a variety of radiation detection instruments. Utilizing modular web design principles along with modern web frameworks in the creation of CharCat allowed for flexible, iterative development with scalable design solutions.

1. Introduction. The Data Mining, Analysis, and Modeling Cell (DMAMC) is a collaboration of subject matter experts from the radiation detection community responsible for leveraging the information from more than 100 tests conducted or sponsored by the U.S. Department of Homeland Security (DHS) Countering Weapons of Mass Destruction Office (CWMD). In order to facilitate the archival, access and reuse of previously collected radiation test data, DMAMC has developed a suite of web applications that specialize in handling specific aspects of radiological data and instruments. These applications have been developed by various teams of scientific staff members from most of the U.S. Department of Energy National Laboratories, Johns Hopkins University Applied Science Laboratory, U.S. Naval Research Laboratory, CWMD, and its contractors. The Characterization Catalog (CharCat), developed under the lead of Sandia National Laboratories, fits right into the DMAMC application ecosystem by providing users - CWMD test scientist and DMAMC members - with an easy way of managing and analyzing the characterization and inventory data of radiological instruments used and tested at CWMD test events. CharCat contents are specific to each detector unit identified by its serial number. The CharCat web application makes it easy to view, create, update, and manage data through the website's clean interface shown in Fig. 2.1 and Fig. 2.2.

2. System Design.

2.1. Technologies Used. The Characterization Catalog utilizes modern web frameworks to develop a modular web application. The application uses MongoDB[*], Express.js[†], React.js[‡], Node.js[§] (often called the *MERN* stack), and Redux[¶]. React.js is a JavaScript library used for building modular web user-interfaces by utilizing the component-focused approach it provides. It's especially useful in developing single-page applications. Redux is used for global component state management, and provides an easy way for multiple components to interface with each other. Express.js is a Node.js framework used to write simple webservers that serve the webpage, and provide an application programming interface (API)

[†]University of Southern California, M.S. Computer Science, mdemeter@alumni.usc.edu
[‡]University of Victoria, Computer Science Undergraduate, bkudryavtsev@uvic.ca
[§]Sandia National Laboratories, R&D S&E Physics, bcabrer@sandia.gov
[¶]Sandia National Laboratories, R&D S&E Computer Science, mhwong@sandia.gov
[*]https://www.mongodb.com/
[†]https://expressjs.com/
[‡]https://reactjs.org/
[§]https://nodejs.org/
[¶]https://redux.js.org/
[‖]The image is intentionally blurred to protect sensitive information

for the web application to interact with. Node.js provides the runtime environment for all the server-side code (including Express.js). Finally, MongoDB is the NoSQL database that the Characterization Catalog uses to persistently store data.

2.2. Redux State Management. Redux is commonly used in applications with complicated workflows as a solution for state management. The application uses Redux to centralize CharCat's application state and logic, enabling content persistence for the user. The Redux store structure is flat and avoids deep nesting, making it easy to update specific slices of the database using unique identifiers like instrument serial numbers efficiently. Despite storing de-normalized data, the application retains the ability to atomically update fields by removing the need for bidirectional references between one to many and many to one relationships. The key benefit in enforcing these rules is maintaining the advantages of de-normalized data: efficient lookups and infrequent joins; while avoiding its flaws: inefficient updates and creations. However, these advantages come at the cost of storing duplicate data and additional documentation. In addition to optimizing create, read, update, and delete (CRUD) operations with Redux, the application strictly uses immutable objects in the Redux store and relies on pure functions for mutating the Redux state. By using immutable objects, the application allows sophisticated change detection techniques to be implemented easily, which ensures that computationally expensive operations like updating the document object model (DOM) are only performed when absolutely necessary. The rationale behind mutating the state with pure functions stems from the fact that multiple React components may attempt to edit the same datum asynchronously. With a shared state and multiple parallel processes running, this quickly becomes a race between the processes that results in nondeterministic behavior. By forcing programmers to update the Redux state using pure functions, it is guaranteed that given the same input, functions always return the same output without any side effects. Removing side effects is essential because in JavaScript, all non-primitive objects are passed into functions as references. If our function directly mutates a property on an object, the object changes outside the function as well. Therefore, the only way to know the full effect a function has on an object is by knowing the full history of the object that is passed in. This produces nondeterministic behavior, which is difficult to debug, especially with applications with complex logic and state management. However, by following these design principles, the development team reduces the possibility for race conditions in our application and ensure fully deterministic behavior.

2.3. Regression Fit Algorithms. The development team created a custom library for the plot fit panel that uses common regression techniques, such as linear least squares. It also gives users the ability to customize settings like the number of degrees for polynomial fits, independently scale the X and Y axes, and fit using a multitude of fitting types. The panel draws the line of best fit on the chart and displays the fitting coefficients on the panel with their standard errors. The library was tested against the CERN ROOT[††] library to verify its accuracy.

2.4. Dataset Filtering. The development team created a waterfall style filtering system to give scientists a flexible method for manipulating datasets. Scientists initially filter instrument data by the dataset they belong to, which affects the data available to lower filters in the hierarchy. Our test case filters give the ability to filter available dataset entries by common traits like the test event, recorded dates, radionuclides, and source ID. Any filters selected in the secondary level updates the final layer, a list of filtered datasets that can further be selected from.

[**]The image is intentionally blurred to protect sensitive information

[††]https://root.cern.ch/

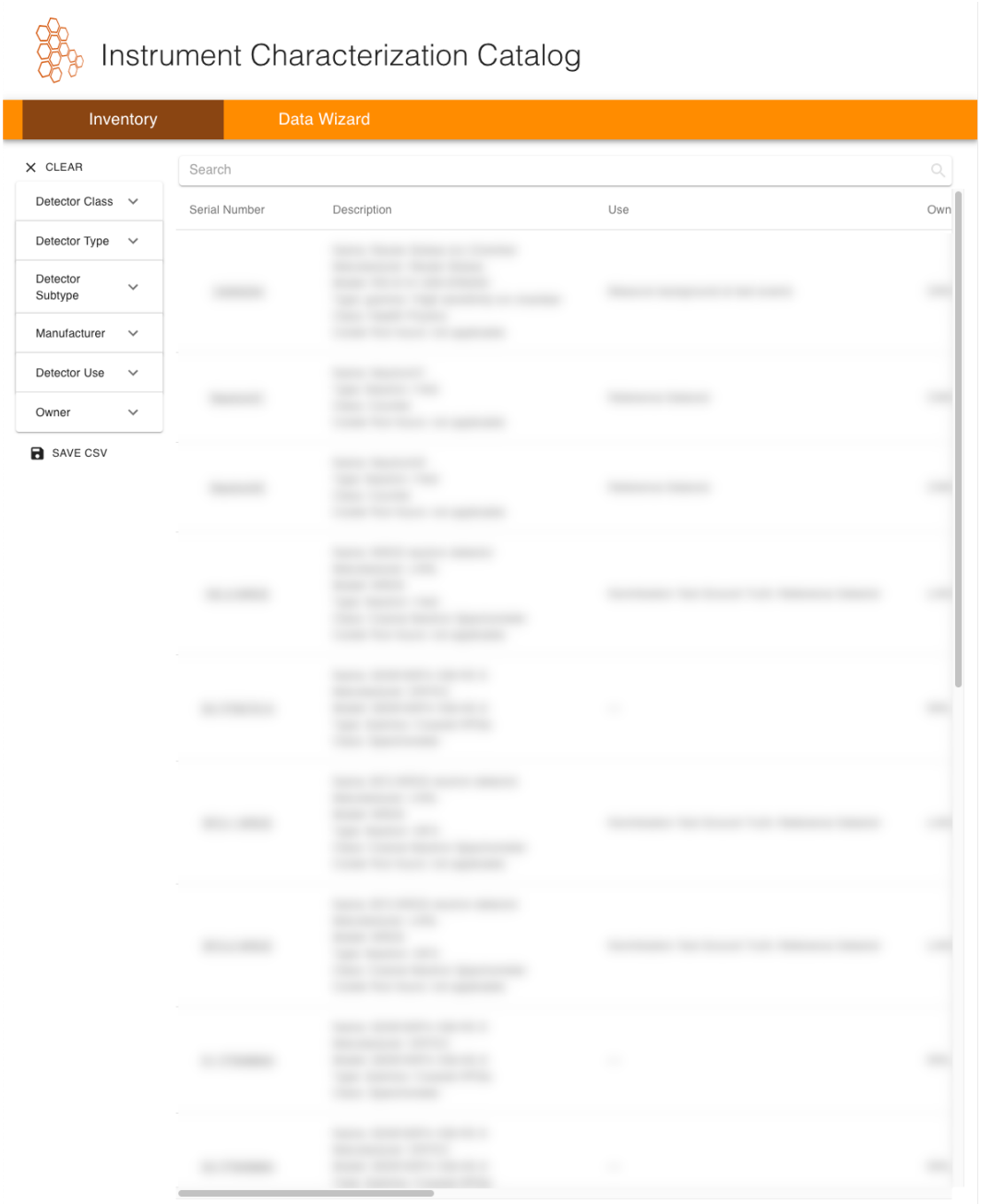


Fig. 2.1: CharCat User Interface - Inventory Management[‖]

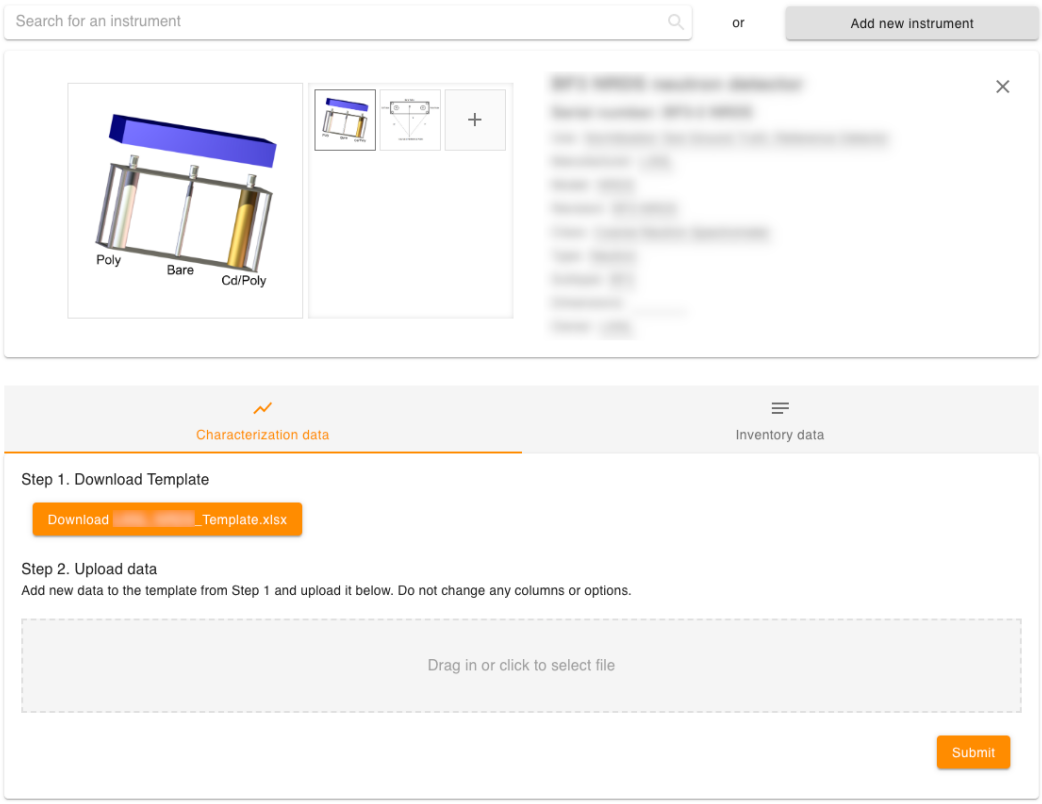


Fig. 2.2: CharCat User Interface - Data Upload**

3. Security Considerations. The Characterization Catalog web application is hosted on Sandia National Laboratories owned servers. The login page is public-facing, however only a few restricted users are able to access the whole application. Regular demonstrations and tests of the application are conducted by various parties interested in determining in which direction the product progresses. As such, it is of high priority to keep the system reasonably safe and secure, even while its architecture, design, and features might swiftly evolve. If a malicious actor were to gain access to the application, depending on how widespread that access is, they could have Official Use Only (OUO) level information, or the usernames of various system users leading to privacy concerns.

3.1. Security During Development. In software development, especially while employing rapidly iterating, agile release cycles - such as the ones encouraged by the Scrum framework - security can have a lower priority, creating some risk. Most of the focus is centered on delivering features and fixing bugs, making security a lower priority. The Characterization Catalog development team took several concrete steps to mitigate the risk of security being an issue during development:

- Employing and utilizing secure coding practices by all software developers
- Conducting regular architecture and source code reviews
- Considering security during the initial system design, as well as later architectural changes
- Paying close attention to online vulnerability lists (e.g. Open Web Application Security Project (OWASP)[*], MITRE Common Vulnerabilities and Exposures (CVE)[†])
- Testing the application before deploying a new version to be hosted publicly with restricted access

3.2. Security Architecture. The Characterization Catalog - under development - is only open to a restricted number of users. To protect the website, and release data only to authorized users, the application utilizes a login system. To authenticate, users utilize basic access authentication to provide their credentials to the server. Through basic access authentication, the credentials are sent in the request's "Authorization: Basic" header field as a Base64[‡] encoded string in the form of "username:password". Due to the constraints of working at Sandia National Laboratories, the team had limited access to hosting infrastructure. As a result, the packets are forwarded through an Apache web server before being handled by the application's Express web server. Both servers authenticate the request's validity. In order to avoid having users provide two separate sets of credentials at the time of login, the passwords are compared against the same master list. Basic access authentication doesn't provide any method of encryption for the packets. Encryption is necessary to protect the credentials, and other information from unauthorized eavesdroppers during their transfer from client to server. As such, all the communication is done over transport layer security (TLS)[§]. After receiving the credentials, the server uses the provided username as a key to look up the password hash associated with that username from the master list. The server then hashes the password received from the client and compares it to the password on file. The user is authorized to access the rest of the application if the password hashes match, however the user is denied access if they don't. The passwords are stored as salted hashes on the same physical system as the web server.

[*] https://www.owasp.org/
[†] https://cve.mitre.org/
[‡] https://www.ietf.org/rfc/rfc4648.txt
[§] https://www.ietf.org/rfc/rfc5246.txt

3.3. Security Evaluation & Vulnerability Assessment. It is important to be constantly aware of what potential vulnerabilities might exist due to how the application evolved. For example, at the start of development, the web server did not need to process any user-given input besides the username and password. However, as the application matured, the API server now has to handle user input for managing and modifying characterization and inventory data pertaining to the nuclear detector instruments in the system. To prevent any type of injection attack by a malicious actor, the input has to be sanitized and correctly stripped of troublesome characters.

3.3.1. Evaluation of MD5 for Password Hashing. The use of message digest algorithm version 5 (MD5)[¶] as a hashing algorithm is strongly discouraged, as its collision-resistance has been broken a long time ago [3, 2, 1] and can no longer be considered cryptographically secure. It's currently not included in the National Institute of Standards and Technology's (NIST) list of recommended hash algorithms for password storage. However, for password storage, pre-image resistance is more important than collision-resistance. Pre-image resistance means that given a hash, the original input that hashes to that output cannot easily be determined. Collision-resistance means that it is very difficult to find two inputs that result in the same hashed output. Since passwords are short, collisions should not be an issue. Consider, the output of MD5 is 128 bits in length, resulting in a total of

$$2^{128} = 3.4028237e^{38}$$

combinations. The number of passwords with an average length of 16 characters, given that there are 95 printable American Standard Code for Information Interchange (ASCII) characters, would be

$$95^{16} = 4.401266687e^{31}$$

combinations. Since an administrator also manages credentials for the Characterization Catalog (no availability of a sign-up system), potential collisions are a non-issue. A bigger issue is creating secure, high-entropy, long passwords by combining characters, numbers, and symbols. Given that strong passwords are enforced by the administrator, there is still an issue of using a computationally intensive hashing algorithm in case the list of passwords is compromised. Even by using Apache's implementation of MD5, which iterates a 1000 times before the final output, an adversary is not seriously limited by the use of marginally more resources. A more computationally resource intensive algorithm, such as bcrypt[‖] would be advisable for password hashing implementation over the MD5 hashing algorithm.

3.3.2. Evaluation of Basic Access Authentication. The current login system is centered around using basic access authentication[**]. Basic access authentication sends credentials with every request in the header of the packet. It is often used in practice with API calls, however it's not quite as useful for full-blown login systems. A token-based system, such as OAuth 2.0[††] would work better, as it provides key revocation options, and as a result: session management. Session management provides users with the ability to log out once they're done using the application, decreasing the likelihood that an authorized person will gain access to the application solely based on the fact that it was left open. The browser stores the basic access authentication credentials in a secure local storage, and only clears it once the browser window is closed. Token-based systems such as OAuth 2.0

[¶] https://www.ietf.org/rfc/rfc1321.txt
[‖] https://www.ietf.org/rfc/rfc7914.txt
[**] https://www.ietf.org/rfc/rfc7617.txt
[††] https://www.ietf.org/rfc/rfc6749.txt

provide the user with a token after authenticating an initial grant supplied by the user (in the simplest case this can be a username, password pair). Tokens act as the temporary key that users can utilize to access restricted resources. Tokens are more easily managed than the user's more permanent password, and thus can be granted and revoked at a relatively fast pace (e.g. daily).

4. Future Work. The Characterization Catalog is nearing the end of its development at Sandia National Laboratories; it still has some features that need to be implemented before releasing it for production. As the project is handed off to the Department of Homeland Security and is integrated with the rest of the DMAMC applications at Pacific Northwest National Laboratories, it is a top priority to take steps to facilitate the transfer with as much ease as possible. For this reason, the development team has decided to utilize Docker[‡‡] for the containerization of the application. Containerization makes sure the execution environment of the application is always static and never changing, regardless of the environment the docker container is running in. Furthermore, Dockerizing the application eliminates the need to install any libraries and dependencies on the host system, as those dependencies are part of the Docker container by default; after careful configuration of the environment.

5. Conclusion. In conclusion, the Characterization Catalog development team has successfully developed a web application that is due to become part of the Data Mining, Analysis, and Modeling Cell (DMAMC) ecosystem hosted on Pacific Northwest National Laboratories' infrastructure. The application keeps track of nuclear detector characterization data obtained by test scientists, as well as inventory data for unique radiological detector instruments. Users are able to easily analyze and interact with the data through the well-designed interface, facilitating the existence of a frequently updated and well-maintained central repository for characterization data and instrument inventory information.

REFERENCES

[1] J. Liang and X.-J. Lai, *Improved collision attack on hash function MD5*, Journal of Computer Science and Technology, 22 (2007), pp. 79–87.
[2] M. Stevens, *Fast Collision Attack on MD5*, IACR Cryptology ePrint Archive, 2006 (2006), p. 104.
[3] X. Wang and H. Yu, *How to break MD5 and other hash functions*, in Annual international conference on the theory and applications of cryptographic techniques, Springer, 2005, pp. 19–35.

[‡‡] https://www.docker.com/

Applications

Articles in this section discuss the utilization of computational techniques in specific codes and applications.

M. Powell
M.L. Parks

September 21, 2020

VERIFICATION TESTING OF THE NimbleSM SOLID MECHANICS FINITE ELEMENT CODE

ANN THOMPSON* AND DAVID LITTLEWOOD†

Abstract. This report documents a set of solution verification tests for the NimbleSM code. NimbleSM is a Lagrangian finite element code designed to produce force and displacement solutions to nonuniform, three-dimensional solid mechanics problems. The tests involved were various patch and beam tests. Our work has shown that NimbleSM functions correctly for the given set of verification tests.

1. Introduction. The Finite Element Method (FEM) is useful for solving partial differential equations and is the most prevalent approach for solving the balance of linear momentum for solid mechanics problems. Errors in FEM codes, however, can go unnoticed because the codes may produce correct results under simple circumstances. It may only be when the problems are sufficiently complex that accuracy is actually hindered, but these complex problems are often the ones that are the most difficult to check. Because of this, there have been many methods designed to test FEM codes for emergent errors. Of these tests, we chose to use various patch and beam tests due to their well designed accuracy and simplicity [5].

The focus of our testing, NimbleSM, is a Lagrangian FEM code used to find solutions to solid mechanics problems [4]. Primary applications include solving for displacement, strain, stress, and force for three-dimensional problems on nonuniform meshes. NimbleSM can use either explicit transient dynamic or implicit quasi-static time integration when executing simulations. The meshes inputted are made of eight-node hexahedral solid brick elements. Hexahedral elements provide a good balance between computational expense and accuracy, and, like all mainstream FEM element formulations, will produce increased accuracy under mesh refinement. Testing NimbleSM was particularly important because the code is designed to have performance portability across varying hardware architectures. A verification test suite is critical to ensuring proper code performance on multiple hardware architectures.

2. Finite-Element Method. FEM is one of multiple strategies used for solving partial differential equations. It is particularly useful for complex geometry and properties, which makes it well suited for solid mechanics. It works through a process of discretization, where an object is divided into numerous elements that are connected by nodes. By doing this, a set of simultaneous algebraic equations is produced, which can then be solved for values at desired locations. Since these elements are finite, the solutions provided are approximate. In solid mechanics, FEM is most often used to solve for displacement, velocity, stress, strain, and force. In the following tests, displacement is the primary solution variable.

In order to solve a solids problem using FEM we must define the geometry, element types, element connectivity, material properties, and boundary conditions. We used Cubit [2], a meshing software, to create the material geometry, element types, and element connectivity in a single mesh file. For the majority of the tests, the material properties are the same. Each material is a Neo-Hookean solid, meaning that they are hyperelastic, but do not have a linear stress-strain curve [7]. Their density is 7.8 g/cm^3, their bulk modulus is 1.6×10^{12} Pa, and their shear modulus is 0.8×10^{12} Pa. The boundary conditions of the tests were displaced surfaces and fixed surfaces [3, 6].

Once these conditions are defined, a set of simultaneous equations is produced. This set of equations embodies the governing equations and the boundary conditions. The combined

*University of Texas at Austin, aat2676@utexas.edu
†Sandia National Laboratories, djlittl@sandia.gov

equations take the form

$$[K]\{u\} = \{f\}, \tag{2.1}$$

where $[K]$ is a given property, $\{u\}$ is a behavior that will be solved for, and $\{f\}$ is a given action. For example, in a typical solid mechanics problem, $[K]$ is stiffness, $\{u\}$ is displacement, and $\{f\}$ is force [3].

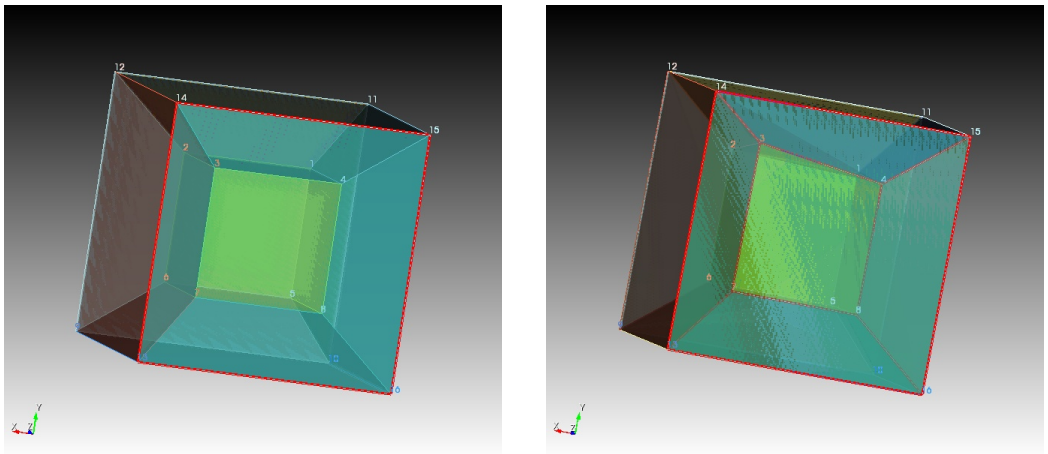
Once the simultaneous equations have been solved for the nodal values, the value for any point in the body can be found using piecewise polynomial interpolation. This means a piecewise function is produced that determines the solution value for any given point in the body. Determining the primary solution values, in our case displacement, is sufficient to find other quantities of interest, for example stress and strain [3, 6]. All of this can potentially be calculated by hand, but to apply it to complex problems, software like NimbleSM is needed. When using a program like NimbleSM, post-processing is generally required. We used a python script to sort and print our results. We then put our results into Gnuplot [9] to produce graphs.

The discretization, or meshing, involved in FEM can be created by using either Eulerian or Lagrangian coordinates. Eulerian coordinates are located in the space around an object, and do not move with the object. Lagrangian coordinates are located on an object and move with it. The Lagrangian approach is generally better for finite element problems with less sever mesh distortions. NimbleSM uses a Lagrangian finite element approach [8].

3. Approach to Testing. In order to test NimbleSM, running multiple problems with known solutions was necessary. The most suitable initial tests for NimbleSM's function are patch tests and beam tests for solids. Patch tests are particularly useful because correct results in a patch test are a necessary condition for solution convergence under mesh refinement, that is, an increase in accuracy as the number of elements is increased. The selected beam tests are useful because they are a simple way to test NimbleSM with distorted elements under uniform strain. The goal for these tests is to demonstrate correctness relative to a known analytic solution. We developed these tests by first creating the volumes and meshes in Cubit [2]. Then we executed the NimbleSM code on a corresponding FEM problem and checked the results by inspecting numerical values and visualizing the solution using Paraview [1]. Below are the tests we used. First is a patch test on a regular grid, followed by an irregular patch test, and finally by beams with varying element shapes.

4. Patch Tests. Our first patch test is a cube with 2.0m sides that has mesh nodes arranged in a smaller cube with 1.0m sides centered internally. In total it has seven elements and 16 nodes. This can be seen in Figure 4.1(a). The purpose of this patch test is to provide a basic solid that can be edited and also referenced by more complex solids. Because of its simplicity, it is likely that NimbleSM would be able to pass this test even if it could not pass others. We used this test by fixing one face, and displacing the opposite face in the X direction. Figure 4.4 is an external visualization of this displacement [1]. In response, all the nodes should be displaced to produce a linear relationship between X coordinate position and X displacement. In Figure 4.2 these exact results can be seen for .001m, .006m, and .01m displacements. They were correct to machine precision. This indicates that NimbleSM is accurate in determining displacement throughout this solid.

Our second patch test uses the first patch test as a template. Each internal node from the first has been moved in each of the three dimensions to a random position within 0.1m, as shown in Figure 4.1(b). The specific coordinates for each inner cube can be seen in table 4.1. This test provides a lot more information than the regular patch test because it is



(a) Patch Test 1 (b) Patch Test 2

Fig. 4.1: Meshes of Patch Tests

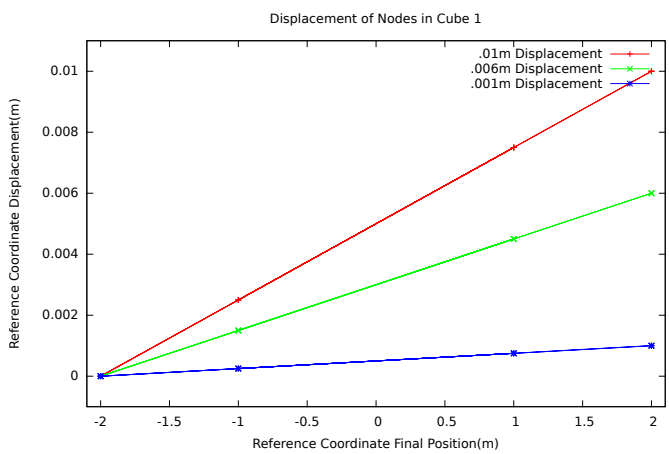


Fig. 4.2: Results of First Patch Test

practically impossible to pass by coincidence. Distorted elements are key to this. Again, one outer face of the cube was fixed while the opposite was displaced. We were looking for the same results in which there is a linear relationship between the nodal X coordinate position and the nodal X displacement. In addition, we want these new lines to match those of the first cube, despite having different coordinate positions. The ability of NimbleSM to correctly solve this classical patch test, in which a linear displacement field is recovered on a nonuniform mesh, is a necessary condition for other important properties of the FEM code, such as mesh convergence.

The results for the second patch test, shown in Figure 4.3, indicate the ability of NimbleSM to solve problems on a nonuniform mesh. All displacements are linear and are in agreement with the expected values. This test did, however, exhibited small but negligible errors in the the displacement solution. The use of a nonuniform mesh increases our confi-

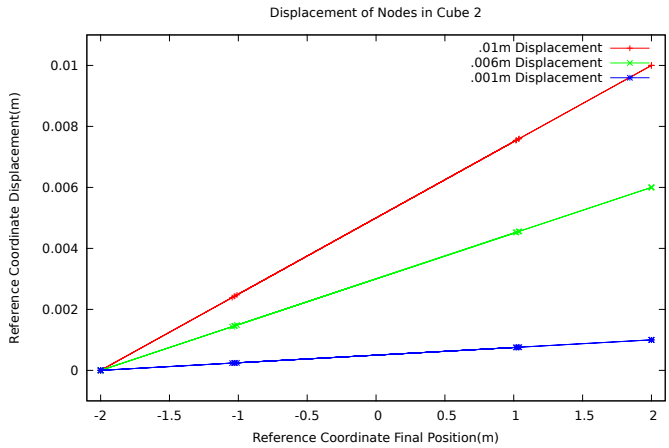


Fig. 4.3: Results of Second Patch Test

Node	Patch Test 1			Patch Test 2		
	X	Y	Z	X	Y	Z
1	-1.000	-1.000	1.000	-0.970	-1.030	1.035
2	1.000	-1.000	1.000	0.985	-1.010	1.015
3	1.000	-1.000	-1.000	0.990	-1.000	-1.001
4	-1.000	-1.000	-1.000	-0.970	-1.020	-1.0301
5	-1.000	1.000	1.000	-0.980	1.020	1.040
6	1.000	1.000	1.000	1.000	1.040	1.020
7	1.000	1.000	-1.000	0.990	1.030	-1.025
8	-1.000	1.000	-1.000	-0.975	1.010	-1.045

Table 4.1: Coordinate Positions of Internal Nodes in Mesh 1 and Mesh 2.

dence in the correctness of NimbleSM. The lines shown in Figures 4.2 and 4.3 for 0.001m, .006m, and .01m displacements are in agreement with the expected results, indicating that NimbleSM is functioning correctly for both regular and irregular patch tests.

The final patch test we used was developed by Richard H. MacNeal and Robert L. Harder [5]. It is a unit cube with an internal irregular cube of nodes. It is structured very similarly to the second patch test, but with different internal node locations. These are listed in table 4.2. In this specific test, our material parameters differ from the rest of the tests. The material is a linear elastic solid, the density is 1 g/m3, the bulk modulus is 0.667×10^6 MPa, and the shear modulus is 0.4×10^6 MPa. The biggest difference between this test and the others is that the displacement is a gradient. It follows the equations:

$$u = 10^{-3}(2x + y + z)/2, \tag{4.1}$$

$$v = 10^{-3}(x + 2y + z)/2, \tag{4.2}$$

$$w = 10^{-3}(x + y + 2z)/2, \tag{4.3}$$

This unique displacement is visualized in Figure 4.5.

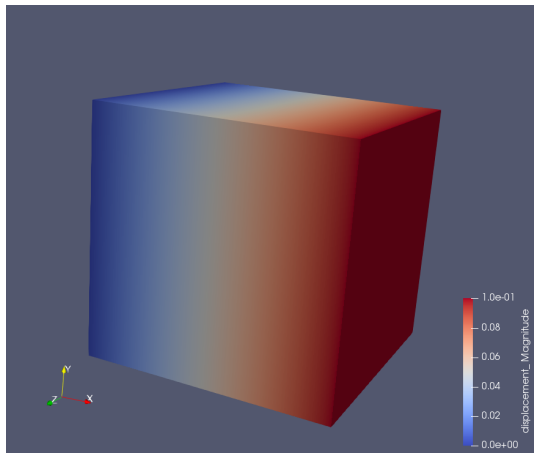


Fig. 4.4: Paraview Rendering of Displacement of Both Patch Tests

Node	X	Y	Z
1	0.249	0.342	192
2	0.826	0.288	0.288
3	0.850	0.649	0.263
4	0.273	0.750	0.230
5	0.320	0.186	0.643
6	0.677	0.305	0.683
7	0.788	0.693	0.644
8	0.165	0.745	0.702
9	0.0	0.0	0.0
10	1.0	0.0	0.0
11	1.0	1.0	0.0
12	0.0	1.0	0.0
13	0.0	0.0	1.0
14	1.0	0.0	1.0
15	1.0	1.0	1.0
16	0.0	1.0	1.0

Table 4.2: Node Locations for the MacNeal-Harder Patch Test

The exact solution for stress for this test is by (4.4),

$$\sigma_x = \sigma_y = \sigma_z = 10^{-3}, \quad \tau_{xy} = \tau_{yz} = \tau_{xz} = 400, \tag{4.4}$$

Our numerical results matched the expected solution within machine precision, again increasing our confidence in the correctness of NimbleSM.

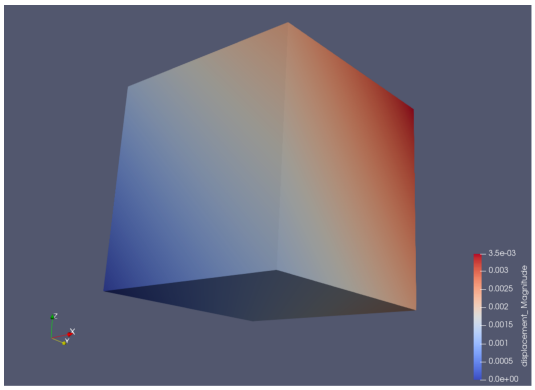


Fig. 4.5: Paraview Rendering of MacNeal-Harder Patch Test.

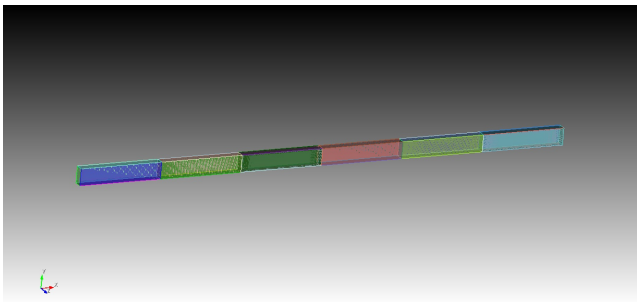


Fig. 5.1: Mesh of Beam with Rectangular Elements

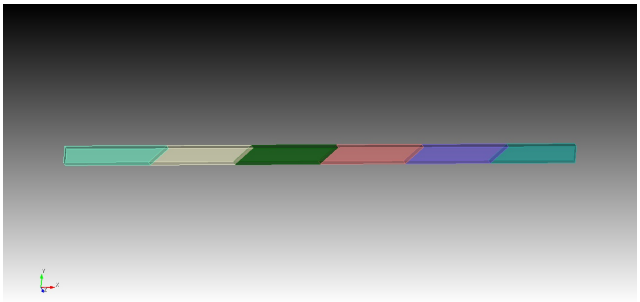


Fig. 5.2: Mesh of Beam with Parallelogram Elements

5. Beam Tests. The second set of tests applied to NimbleSM were beams under tension. Beam tests are useful because of their simplicity and their variability. We tested three identical beams that are 6.0m in length, 0.2m in width, and 0.1m in height. In this test, the beams have a fixed end and an end that is displaced. The ends are opposite each other and perpendicular to the x axis. This displacement is shown in Figure 5.4. Each of the three beams have six differently shaped elements. The first consists of rectangular elements, the second of parallelogram elements, and the third of trapezoidal elements. The nodes of the irregular elements have been moved 0.2 m away from their position in the rectangular beam.

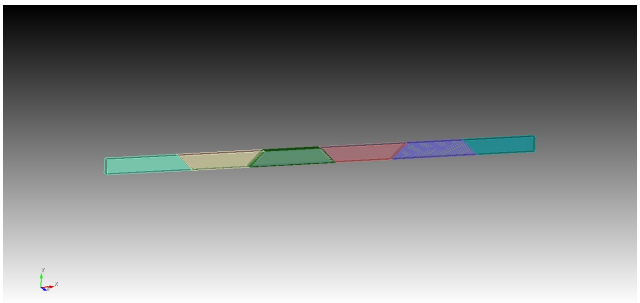


Fig. 5.3: Mesh of Beam with Trapezoid Elements

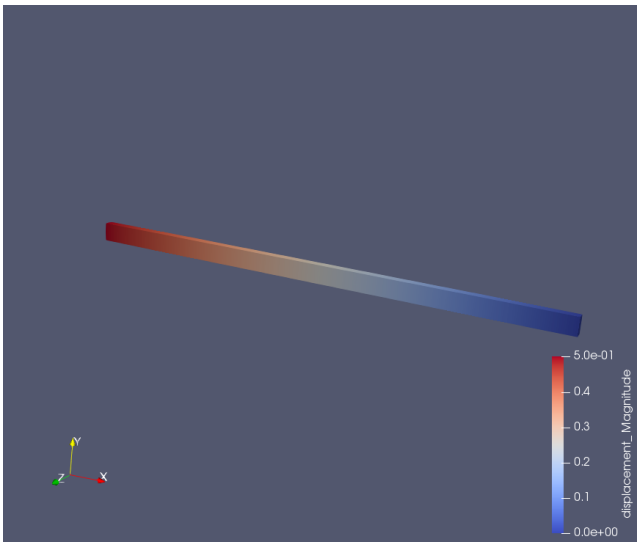


Fig. 5.4: Normal Beam Displacement

Each beam has 28 nodes. These solid meshes are shown in Figure 5.1, Figure 5.2, and Figure 5.3. Although simple, beams of this type could be used to test constant strains, linearly varying strains, constant curvatures, and linearly varying curvatures, which are principal element deformation modes. In this study, we examined constant strains. Changing the element shapes creates a more thorough test [5].

The expected results for the beam tests are also linear relationships between the node coordinates and node displacements. Again, we tested all three of the beams with .01m, .006m, and .001m end displacements. Although the points will be located differently, the lines for all three beams should be identical [5].

The results of this test were as expected. The linear relationship between the node coordinates and displacements are shown in Figure 5.5, Figure 5.6, and Figure 5.7. They each have their own point placement, but all show the same lines. The errors were within machine precision.

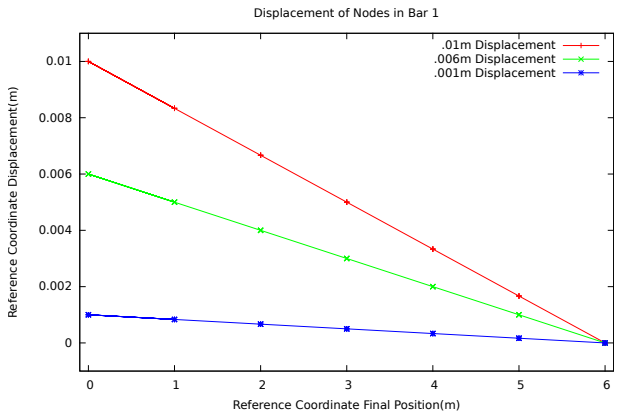


Fig. 5.5: Beam with Rectangular Elements Displacement Results

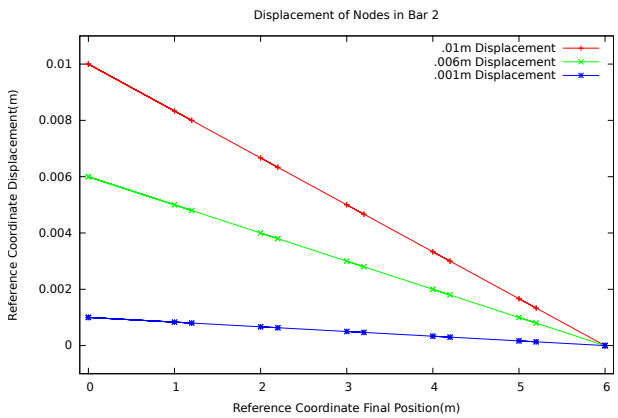


Fig. 5.6: Beam with Parallelogram Elements Displacement Results

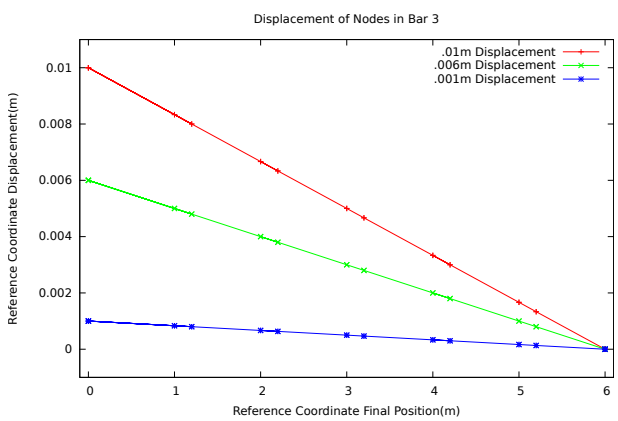


Fig. 5.7: Beam with Trapezoid Elements Displacement Results

6. Lessons Learned. During these tests we encountered multiple challenges that resulted in incorrect solutions. It is important to recognize that even if a finite element code is working properly, other problems in the workflow can create difficulties. Luckily, we rechecked our work to find that the errors were a result of earlier steps in our procedure. The two main errors encountered were due to an incorrect mesh and incorrect boundary conditions.

Our error in the mesh was easily caught. While meshing the patch test, one of the elements was not formed despite the nodes being in the correct position. This gave us strange results, but it did not take long to figure out the problem, due to the odd appearance of the mesh. In this case, the Cubit mesh generator was failing to create the desired elements for the prescribed node locations. This difficulty was overcome by modifying our Cubit workflow.

The error with boundary conditions was much more significant. While running the beam tests, we applied boundary conditions to the two ends of the beam. Boundary conditions were applied in the X direction, but not in the Y and Z directions, which resulted in unconstrained rigid-body modes that affected the ability of NimbleSM to converge to the expected results. The errors in these results are listed in table 6.1. Because of these results, we had to question the integrity of NimbleSM, but after significant searching, we found the problem in the boundary conditions. NimbleSM produced the expected results after the boundary-condition error was corrected.

Because there are many steps to creating and running these tests, there is a lot of room for error outside of the software. While the main purpose of the tests is to test the code on its own, the tests also provide information on how to prepare the problems. With this knowledge, possibility for error in any context is decreased.

Coordinate(m)	Bar 1	Bar 2	Bar 3
1.0	1.34%	1.07%	1.05 %
1.2		1.26%	1.29%
2.0	0.56%	0.98%	0.61%
2.2		0.66%	0.76%
3.0	0.00%	0.22%	0.32%
3.2		0.01%	0.01%
4.0	1.12%	0.79%	0.91%
4.2		1.61%	0.87%
5.0	6.71%	5.05%	5.18%
5.2		5.53%	5.38%

Table 6.1: The Initial Use of Incorrect Boundary Conditions Resulted in Significant Solution Errors for the Rectangular Beam with 0.01m end Displacement.

7. Conclusion. NimbleSM functioned correctly as a Lagrangian finite element code with these patch tests. This increases our confidence in the correctness of the code and provides a set of tests that can be used in the NimbleSM test suite. Patch tests, like those defined by MacNeal and Harder [5], are a cornerstone of verification testing for FEM codes. Further, the process of creating and executing the verification tests provided an opportunity to become familiar with the workflow of a FEM analysis for solid mechanics.

REFERENCES

[1] U. Ayachit, *The ParaView Guide*, Kitware Inc., 2019.

[2] Cubit Team, *Cubit 15.4 user documentation*, SAND Report 2019-3478 W, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2019.

[3] O. de Weck and I. Y. Kim, *Finite element method*, Jan 2004. http://web.mit.edu/16.810/www/16.810_L4_CAE.pdf.

[4] D. Littlewood, *NimbleSM*, 2018. https://github.com/NimbleSM/NimbleSM.

[5] R. H. Macneal and R. L. Harder, *A proposed standard set of problems to test finite element accuracy*, Finite Elements in Analysis and Design, 1 (1985), pp. 3–20.

[6] U. of Victoria Engineering, *Introduction to finite element analysis (fea) or finite element method (fem) lecture notes*, Mar 2016.

[7] Wikipedia, *Neo-hookean solid*, 2019.

[8] Wikiversity, *Nonlinear finite elements/lagrangian and eulerian descriptions*, 2017.

[9] T. Williams and C. Kelley, *Gnuplot 4.4: an interactive plotting program*, March 2011.

PREDICTING MOLECULAR TOXICITY FROM STRUCTURAL IDENTITY

CATHERINE M. WRIGHT* AND SIVA RAJAMANICKAM†

Abstract. Whenever a new chemical product is created it brings with it a risk of potential harm to human health that must be extensively tested. Conventional methods of pre-clinical toxicology testing involve expensive and ethically-questionable experiments typically performed on animals. Recent approaches posed by the consortium *Toxicology in the 21st Century* involve a series of high-throughput screening processes of a chemical compound that looks for possible toxic effects caused by interrupting biological pathways. We can model the data collected from these tests using machine learning methods to try to accurately predict whether an unknown compound is toxic to human health. Our approach trains a model based on data derived solely from chemical structure. Comparing two unsupervised machine learning algorithms, *struc2vec* and *node2vec*, we create representations of molecules by concatenating the low-dimensional feature embeddings formed by each molecule's atomic structure. Using these representations, we utilized ensemble decision trees to classify molecular toxicity in a set of compounds. Results achieved from this model indicate that this method could be a starting point for future work.

1. Introduction. Human interaction with chemical substance is unavoidable. Whether it is pesticides, food additives, medications, environmental chemicals, or more, it is imperative to understand which biomolecular substances exhibit toxic effects to humans. Conventional pre-clinical testing for these effects is typically expensive and time consuming, and usually rely on using animals. This brings forward many issues; it provides no guarantee to human safety during clinical tests, alongside ethical concerns. "Toxicology in the 21st Century", or *Tox21*, is a US federal research collaboration that provides a new-age method of toxicity testing, developed by conducting tests in human cells or cell lines *in vitro*. Analysis of cell response through individual toxicity pathway assays can be performed with the assistance of robotics to provide both time- and cost-efficient testing, while bypassing the need for animal experimentation [1].

Part of this development includes production of computational models that can accurately determine presence of toxins. Many computational models suffer from insufficient accuracy, so in 2014 a challenge was issued to the community to produce the best model for predicting compound's interference in biochemical pathways using only data pertaining to chemical structure. The best fit model could be then used by government agencies for determining compounds that cause toxic effects to human health. The Tox21 challenge dataset consisted of activity data from 12 independent assays for approximately 10,000 molecules, and resulted in a series of fairly-accurate classification models [6].

In this paper, we compare using either node neighborhood locality or global structural identity for classification of toxins present in molecular compounds. Node neighborhood locality here refers to the homophilic communities of nodes within networks. *node2vec* [4] is an unsupervised algorithmic framework that uses node and edge input of a graph to learn continuous feature representations of nodes. Projecting the network into a low-dimensional space, it works to preserve network neighborhoods of nodes. Structural identity of nodes refers to the relationship between both a node and the overarching network structure, as well as between a node and the identities of other nodes. *struc2vec* [11] is an unsupervised learning approach for building latent representations using only network structure to characterize the structural identity of individual nodes. By using molecular bond structure as input, *struc2vec* obtains high-dimensional feature representations of each molecule that can be used in supervised Machine Learning classification methods, and predicting the toxicity

*University of New Mexico Department of Computer Science, wrightc@unm.edu
†Sandia National Laboratories, srajama@sandia.gov

of new compounds. Another potential approach in line with the above methods is to utilize *graph2vec* [9], an unsupervised learning algorithm that looks at graphical substructures to create distributed representations of graphs. This paper focus' on frameworks that apply solely to atomic structure of individual molecules, therefore we do not consider *graph2vec*, although it could be utilized in future analysis.

We used ensemble decision trees to model and classify the embeddings created by *node2vec* and *struc2vec*. Due to a high volume of inactive compounds, this classification method was not as competitive as the methods used in related works, but did provide a quantification of the usefulness of a latent space representation using molecular structure alone.

The main contributions of this work is an evaluation of two node-level embeddings in the context of a compound classification problem using only molecular structure as input. We also compare a classifier using these embeddings with state-of-the-art classifiers for toxicology of compounds.

2. Related Work. Unsupervised learning approaches to build representational molecular models has been a subject of interest in both Machine Learning communities as well as biomolecular communities. Modeling the presence of toxins within compounds directly affects human safety on a large scale, therefore the Tox21 dataset has been leveraged multiple times as a benchmark for predictability.

The Grand Challenge winners of the 2014 Tox21 challenge produced DeepTox [8], which computes chemical features to describe the chemical compounds, then feeds these features into a Deep Neural Network (DNN) accompanied with complementary Machine Learning models to best predict toxicity of new compounds. DeepTox produced a final averaged AUC score of 0.846.

In 2017 mol2vec [7], a method of creating vector representations from molecular substructures using unsupervised learning was proposed. By making use of Morgan Fingerprints, which contain substructure information about a molecule, mol2vec uses a Natural Language Processing inspired method of treating compound substructures as words and complete compounds as sentences. In doing so, a modified word2vec unsupervised learning approach is used to obtain high-dimensional embeddings of substructures, clustering chemically related substructures in latent space. This feature representation learning produces dense vector embeddings as compound features that they tested through several supervised learning classification algorithms. Overall, best results were produced by using a Random Forest (RF) method, which produced an AUC score of 0.83 ± 0.05 for the 12 bioassays.

In this paper we use *node2vec* [4], which is an algorithmic framework for an unsupervised method of learning continuous feature representations for individual nodes within graphs. It preserves node neighborhood locality, and has flexible parameterization that can influence how representations are built and what they represent. *node2vec* samples a neighborhood using a biased random walk, which based on the hyperparameters will search according to a breadth-first search algorithm or a depth-first search algorithm. For finding node neighborhood locality, a breadth-first search is used in conjunction with stochastic gradient descent similar to many natural language processing algorithms to generate an embedding.

Our approach compares the embeddings created by *node2vec* to the structural embeddings of *struc2vec* [11], which generates learned representations of molecular graphs to capture atomic structural identity. Designed with intent to form latent representations based solely on structural graph composition, *struc2vec* uses only presence of edges within the graph and no information regarding node or edge attributes. Ideally the representations place nodes in high-dimensional coordinate space such that distance between nodes is correlated to structural similarity. The approach utilizes Dynamic Time Warping (DTW)

to impose a hierarchy for measuring structural similarity, before constructing a multilayer weighed graph that represents the structural similarity of every node at increasing hop distance away. *struc2vec* also uses biased random walks to traverse this graph and build representations using natural language processing techniques.

3. Datasets. The complete dataset that was used in testing comprised of the training dataset provided by the National Center for Advancing Translational Sciences (NCATS) for the 2014 Tox 21 challenge. It contains the results from high-throughput screening assay measurements of 12 toxic effects. Seven effects were part of the Nuclear Receptor (NR) pathways, while 5 were part of the Stress Response (SR) pathways. NRs are a superfamily of transcription factors that cause many human diseases and play a key role in regulation of biologic processes [10]. SR pathways work to resist the effects of stress and restore cell and tissue homeostasis. Both NR and SR pathway effects are important to understand and pinpoint due to their direct implications on human health. Activation of NR effects can disrupt the human endocrine system, while activation of SR effects can cause liver failure and cancer [8].

The NR signaling panel is comprised of seven datasets, or bioassays, which are an Androgen receptor (NR-AR), Androgen Receptor Ligand Binding Domain (NR-AR-LBD), Aryl Hydrocarbon Receptor (NR-AhR), Estrogen Receptor (NR-ER), Estrogen Receptor Ligand Binding Domain (NR-ER-LBD), Aromatase (NR-Aromatase), and Peroxisome Proliferator-Activated Receptor gamma (NR-PPAR-gamma). The SR signaling panel is comprised of five bioassays, including Antioxidant Responsive Element (SR-ARE), ATP-ase family AAA Domain containing 5 (SR-ATAD5), Heat Shock Factor responsive element (SR-HSE), Mitochondrial Membrane Potential (SR-MMP), and p53 (SR-p53). Table 3.1 contains information about the number of molecules in each bioassay, along with the ratio of active to total molecules.

Table 3.1: Tox21 Training Data Points per Assay

Inhibitor	Total	Active	Ratio of Active/Total
NR-AR	9349	380	0.04
NR-AR-LBD	8589	301	0.04
NR-AhR	8159	948	0.12
NR-ER	7688	937	0.12
NR-ER-LBD	8742	444	0.05
NR-Aromatase	7217	360	0.05
NR-PPAR-gamma	8172	219	0.03
SR-ARE	7161	1098	0.15
SR-ATAD5	9081	337	0.04
SR-HSE	8138	423	0.05
SR-MMP	7311	1141	0.16
SR-p53	8623	535	0.06

4. Approach. The available dataset consisted of SMILES (Simplified Molecular-Input Line-Entry System) strings alongside a MOL chemical table file and bioassay results. The provided MOL file contained all molecular information regarding atom and bond structure for which we parsed into individual files containing the bond structure of each molecule. In Figure 4.1 we illustrate how the bond list could be extracted from the structure and used as a graph edgelist, which was run as input through *struc2vec* and *node2vec* to produce a 2-dimensional embedding containing latent space coordinates for each atom. To create our dataset we input all molecule bond information into both *struc2vec* and *node2vec*, using 16-dimension embeddings to create dense compound features derived from the atoms of every molecule. A single run of *struc2vec* and *node2vec* on every molecule was used to compile the subdatasets corresponding to each bioassay.

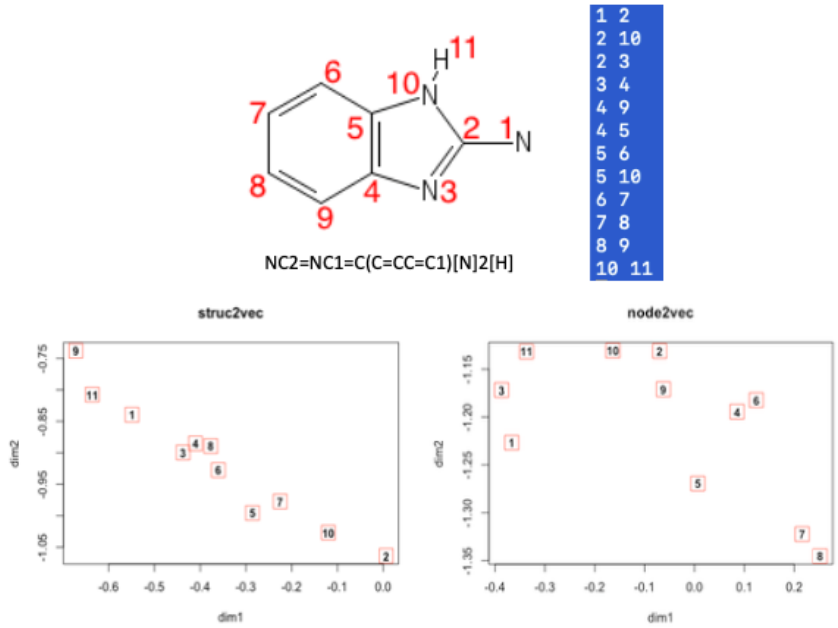


Fig. 4.1: Latent space representation of a molecular compound. SMILES string and 2-D representation (top left), molecular edgelist used as input to *struc2vec* and *node2vec* (top right), 2-D embedding in *struc2vec* (bottom left), and 2-D embedding in *node2vec* (bottom right).

Looking at training data for a particular assay, we can compile the set of molecules M for which we have data regarding the activation of the compound. This data is a binary classifier that is 0 if the molecule is inactive, and 1 if active. All molecules in M have latent space embeddings generated by *struc2vec* and *node2vec* with 16 dimensions. To compile molecular embeddings into a single input vector X for each assay we performed truncation and ordering.

Truncation provided a way to limit the amount of padding added to each molecule while maintaining as much embedding data as possible. Figure 4.2 contains a histogram for the number of atoms per molecule, with each assay overlaid. Since 98% of molecules are composed of 50 atoms or less, we truncate the maximum number of atoms in X to be 50 as opposed to the true maximum of 136. Ordering the atoms by element proved to be useful to avoid features being limited to the ordering set by the SMILES string. Atoms in X were rearranged with elements of highest occurrence in the dataset ordered first.

For each input matrix X used as input in our binary classifier, we started with a set of molecules M for which toxicity testing for the assay was provided. Within M, there exists a given molecule mol_i, such that each row of mol_i embedding will give the latent space coordinates of a specific atom. Because not all compounds in a particular bioassay's dataset have equal number of atoms, we used 0-padding to ensure matrices would be of equal size. An example of the padded *struc2vec* molecular embedding of mol_i is shown in Table 4.1, where n represents the number of atoms within mol_i, and N represents the maximum atoms in the set of molecules. If truncation is applied, N will be equal to 50 atoms.

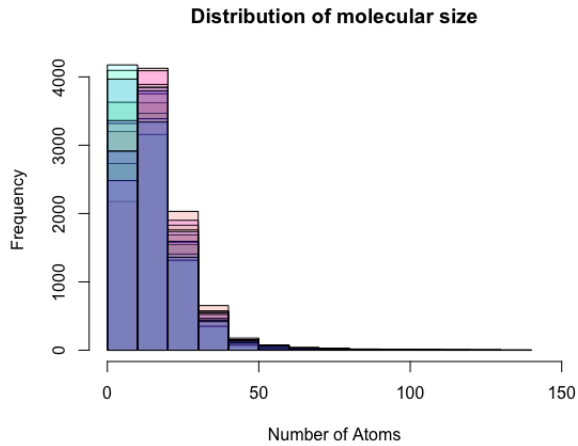


Fig. 4.2: Distribution of Molecule Size

Table 4.1: Example of Molecular Latent Space Representation with 0-Padding

atom	dim1	...	dim16
1	val	...	val
2	val	...	val
$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	val	...	val
$n+1$	0	...	0
$\vdots$	$\vdots$	$\ddots$	$\vdots$
N	0	...	0

From here we flattened each molecule by row to produce a single vector size 1X$(16 * N)$. For molecule mol_i, this is represented as follows.

$$mol_i = \{val_{d1}, \dots val_{d16}, \cdots, val_{d1}, \dots val_{d16}, 0, \dots 0\}$$

By flattening the embedding data of every molecule we can then combine all mol_i for $1 \leq i \leq |M|$ to produce our input feature matrix X_A, and the corresponding label vector y_A, which contains the classifier for every molecule in bioassay set A, such that $A_i \in \{0, 1\}$.

$$X_A = \begin{bmatrix} mol_1 \\ mol_2 \\ \vdots \\ mol_i \\ \vdots \\ mol_{|M|} \end{bmatrix} \quad y_A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_i \\ \vdots \\ A_{|M|} \end{bmatrix}$$

Creating X_A for each bioassay yields a pre-processed molecular embedding produced by *struc2vec* or *node2vec* and a corresponding vector of classification labels. Each X_A was used as an input to an ensemble decision trees algorithm in order to perform the binary classification task of predicting the toxicity y_A to be either active or inactive. Data were separated into 80/20 training/test stratified random split. For classification we used AVATAR: Adaptive Visualization Aid for Touring and Recovery [5]. Avatar provides a set of tools for classifying large datasets using ensembles of C4.5 decision trees created from the dataset [3].

5. Results. The output probabilities determined by classifying each test set in AVATAR are quantitatively measured using the area under the Receiver-Operator Characteristics curve (ROC-AUC) performance metric. This metric was preferred because in the case of bioassays, overall accuracy is not a telling metric due to a low ratio of active molecules within the test set. The ROC-AUC score of a binary classifier is the probability that the model will rank a chosen positive instance higher than a negative instance [2]. This value represents the expected performance of the model, where a score of 0.5 signifies the model performing no better than random choice. Using variations in the above approach for embeddings created by both *struc2vec* and *node2vec*, resulting averaged AUC scores of 10 tests are shown in Table 5.1, denoting **O** to imply elements were ordered by occurrence, and **T** implying that the molecules were truncated to 50 atoms.

Best classification came from using the full *node2vec* embedding with atoms ordered by occurrence. We believe that further ordering of elements by molecular features would improve the overall score, leading to further exploration into this approach. The ROC curve of bioassay NR-AR-LBD is shown in Figure 5.1, which achieved the highest area under the curve of any assay.

Fig. 5.1: ROC Curves for NR-AR-LBD Inhibitor

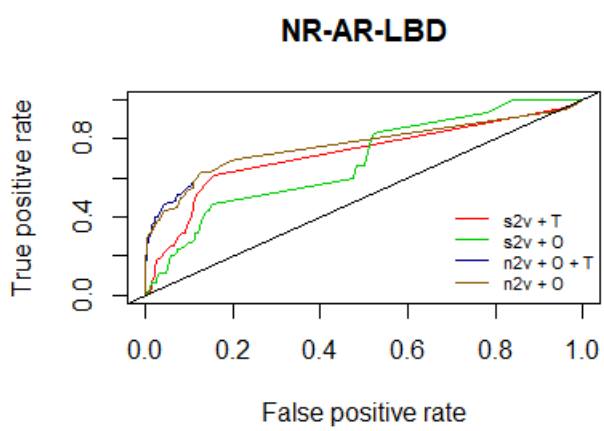


Table 5.2 compares our results to the two leading papers to classify the *Tox21* dataset: DeepTox [8] and mol2vec [7]. Preliminary results show that using structural information at the atomistic level may not be enough to be competitive in this dataset. We plan to study molecular level approaches with embedding of the molecules.

Table 5.1: ROC AUC scores comparing embedding mechanisms using struc2vec (stv), node2vec (ntv) with truncation (T) and/or atomic ordering (O)

Inhibitor	s2v + T	s2v + O	n2v + O + T	n2v + O
NR-AR	0.663	0.62	0.708	0.77
NR-AhR	0.628	0.602	0.653	0.658
NR-AR-LBD	0.723	0.683	0.773	0.77
NR-ER	0.603	0.587	0.605	0.605
NR-ER-LBD	0.673	0.633	0.602	0.726
NR-Aromatase	0.665	0.695	0.759	0.758
NR-PPAR-gamma	0.641	0.587	0.577	0.539
SR-ARE	0.587	0.593	0.567	0.573
SR-ATAD5	0.627	0.619	0.65	0.646
SR-HSE	0.562	0.572	0.539	0.533
SR-MMP	0.582	0.63	0.671	0.721
SR-p53	0.61	0.654	0.62	0.653
Average	0.63	0.623	0.644	0.663

Table 5.2: ROC AUC scores compared to leading algorithms

Inhibitor	DeepTox (DNN)	mol2vec (RF)	n2v + O (RF)
NR-AR	0.849	0.89	0.775
NR-AhR	0.376	0.79	0.63
NR-AR-LBD	0.88	0.87	0.777
NR-ER	0.666	0.73	0.63
NR-ER-LBD	0.653	0.82	0.597
NR-Aromatase	0.752	0.85	0.716
NR-PPAR-gamma	0.637	0.81	0.551
SR-ARE	0.792	0.82	0.643
SR-ATAD5	0.797	0.85	0.581
SR-HSE	0.735	0.78	0.551
SR-MMP	0.849	0.9	0.674
SR-p53	0.696	0.84	0.606

6. Conclusion. Performing toxicology testing on chemical compounds both in circulation as well as newly released is necessary for the overall well-being of living creatures. The high-throughput screenings performed robotically that produced the Tox21 dataset provide a new age method to toxicology risk assessment that is more efficient and ethical than previous methods. Our approach focused on the core structure of a molecule, using *struc2vec* [11] and *node2vec* [4] to build a high-dimensional feature embeddings for each molecule based on structure of the global network or of the homophilic community. Classification of molecular toxicity was performed by utilizing ensemble decision trees. The results demonstrate that it is not enough to use just the atomic structure as part of the embeddings. We plan to evaluate molecular (graph) embeddings as opposed to node embeddings.

REFERENCES

[1] M. E. Andersen and D. Krewski, *Toxicity testing in the 21st century: bringing the vision to life*, Toxicological sciences, 107 (2008), pp. 324–330.

[2] A. P. Bradley, *The use of the area under the roc curve in the evaluation of machine learning algorithms*, Pattern recognition, 30 (1997), pp. 1145–1159.

[3] N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer, *Learning ensembles from bites: A scalable and accurate approach*, Journal of Machine Learning Research, 5 (2004), pp. 421–451.

[4] A. Grover and J. Leskovec, *node2vec: Scalable feature learning for networks*, in Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2016, pp. 855–864.

[5] L. Hall, K. Bowyer, N. Chawla, T. Moore Jr, and W. Kegelmeyer, *Avatar–adaptive visualization aid for touring and recovery*, tech. rep., Sandia National Labs., Albuquerque, NM (US); Sandia National Labs, 2000.

[6] https://tripod.nih.gov/tox21/challenge, *Tox21 data challenge 2014*.

[7] S. Jaeger, S. Fulle, and S. Turk, *Mol2vec: unsupervised machine learning approach with chemical intuition*, Journal of chemical information and modeling, 58 (2018), pp. 27–35.

[8] A. Mayr, G. Klambauer, T. Unterthiner, and S. Hochreiter, *Deeptox: toxicity prediction using deep learning*, Frontiers in Environmental Science, 3 (2016), p. 80.

[9] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, *graph2vec: Learning distributed representations of graphs*, arXiv preprint arXiv:1707.05005, (2017).

[10] J. M. Olefsky, *Nuclear receptor minireview series*, Journal of Biological Chemistry, 276 (2001), pp. 36863–36864.

[11] L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo, *struc2vec: Learning node representations from structural identity*, in Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 385–394.

SUPPLEMENTING THE DAMERAU-LEVENSHTEIN MINIMUM EDIT DISTANCE ALGORITHM WITH PHONETIC MATCHING

LOKRAJ SRINIVASAN* AND EDWARD J. WALSH†

Abstract. Spell checking is a common feature of most word-based applications. Popular examples include Google search spell check and Microsoft Word autocorrect. Most spell checkers use well developed mathematical concepts like the Levenshtein edit distance and statistics to find possible corrections. This research explores the addition of phonetics into the Damerau-Levenshtein distance algorithm and its effect upon the algorithm's accuracy. Two spell-checking algorithms are compared, and their accuracies examined. Testing of the phonetic implementation and the Damerau-Levenshtein Distance algorithm reveals that use of phonetics provides improved accuracy.

1. Introduction. With the progression of time and advancements in technology, people have grown to increasingly rely upon spell check when using their devices and machines. They often let auto correct catch any spelling errors. Even with Google searches, a safety net is cast underneath the request with an auto corrected version of the user input.

The secret to most spell checkers lies within natural language processing. An algorithm analyzes a user's misspellings to deduce the correct spelling. Some spell-checking algorithms integrate Vladimir Levenshtein's concept of the Levenshtein distance to create several correctly-spelled candidates. Over the years, these algorithms have grown to include a variety of other procedures and principles to boost the chances of finding the correct spelling.

Spell checkers often have a list of correctly-spelled words that the computer will compare against. Often, there's additional information coupled alongside this list to increase the probability of finding a more accurate group of candidate corrections. This paper provides the approach of one of the algorithms taken to spell check and describes the use of phonetic matching to improve accuracy

2. Background. The Levenshtein distance refers to the number of single character edits (deletion, insertion, and substitution) that must be made to one sequence of characters to transform it into another. This is more colloquially known as the minimum edit distance. Frederick J. Damerau modified this string manipulation algorithm by adding the transposition of adjacent characters. According to Damerau, the combination of these four edit processes account for more than 80% of all human spelling errors[1]. With this new process, Damerau reduced the number of character edits required to correct a word

The implementation of the Leventhstein Minimum edit distance is known as the Wagner-Fischer algorithm. The algorithm computes the minimum number of edits to transform one string to another. The first step is to break down each string into a sequence of substrings. For example, "dolphin" is broken down into these substrings: d, do, dol, dolp, dolph, dolphi, and dolphin. "dolfun" is broken down into these substrings: d, do, dol, dolf, dolfu, and dolfun. Figure 1 depicts the minimum edit distances of all combinations of substring-transformations for "dolfun" to "dolphin." It takes 2 edits to convert "dolf" to "dolph." The algorithm uses previous smaller-substring computations to derive the number of edits for larger substring transformations. Converting "dolf" to "dolphi" requires 3 edits (2 edits to convert dolf to dolph plus one insertion). The cell in the bottom right-hand corner of figure 1 is the minimum number of edits to convert "dolfun" to "dolphin."

*University of Michigan Computer Science, lokrajs@university.edu
†Sandia National Laboratories, ejwalsh@sandia.gov

		D	O	L	P	H	I	N
	0	1	2	3	4	5	6	7
D	1	0	1	2	3	4	5	6
O	2	1	0	1	2	3	4	5
L	3	2	1	0	1	2	3	4
F	4	3	2	1	1	2	3	4
U	5	4	3	2	2	2	3	4
N	6	5	4	3	3	3	3	3

Figure 1. Figure 1: A matrix depicting the Levenshtein distance matrix between the correctly spelled word "Dolphin" and the improperly spelled word "Dolfun." Each cell contains the minimum number of edits that must be made to get the substring in the first column to become the top row. The 3rd box on the primary diagonal is a 0 since 0 edits are required to transform "Do" into "Do." Transforming "Dolf" into "Dolph" takes 2 edits (1 substitution and 1 insertion).

3. Phonetics.

3.1. Phonetic integration into the Damerau-Levenshtein Algorithm. To improve the Damerau-Levenshtein algorithm's accuracy, phonetic matching was integrated. Experiments were conducted to test its feasibility and effect. Many spellcheckers use popular libraries like Metaphone and Soundex as the sole form of correction; this requires the user to get the structure of the word mostly correct. Combining the minimum edit distance calculation and the Metaphone library increases the probability of success in finding accurate suggestions.

The phonetic implementation first uses the Damerau-Levenshtein minimum edit distance to generate several strings by using a sequence of single-character edits (insertions, deletions, transpositions, and substitutions). Next, the Metaphone library finds matching sounds within the list of correctly spelled words–for example "Foto" and "Photo." Vowels are removed from the user input string and from all candidate corrections; this allows a match

to be found when the user accidentally alters the number of vowels or their respective placements (examples: "Calender" – "Calendar", "liesure" – "leisure", "potatos" – "potatoes"). Comparisons, with a list of correctly spelled words, are used to generate suggestions.

The phonetics aspect of this algorithm serves to eliminate potential words that could be considered a false correction. It relies on the user's attempt at properly sounding and spelling the word out. Once the single character edits are generated, the Metaphone library filters out the ones that do not sound similar to any correctly-spelled word. This eliminates obscure suggestions.

3.2. The Fuzzy Logic Underlying Phonetic Matching. The underlying assumption and foundational idea as to why this phonetic algorithm works, when coupled with the Damerau-Levenshtein distance, comes from Fuzzy String Matching. The minimum edit distance is a comparison of two strings and a numerical quantification as to how similar they are. Fuzzy logic was put forth by Dr. Lotfi Zadeh[7] and states that there are several degrees of 'truth' and there is no simple true or false when it comes to deducing something or deciding. Fuzzy logic is based off of Jan Lukasiewicz's work with "many-valued" logic. The same holds true and is applied to string matching with the minimum edit distance. There are several string matches that can arise solely from the Damerau-Levenshtein distance; the phonetic implementation can eliminate improbable truths. For instance, the string "rime" has a minimum edit distance of 3 for both "Rhyme" and "Seize." The Damerau-Levenshtein algorithm provides no contextual information to deduce which of the two suggestions is correct. The phonetic library can eliminate "seize" because rime and seize sound nothing alike. Fuzzy logic enables the Damerau-Levenshtein algorithm to find multiple different possible string corrections, but the addition of phonics aids in reducing the number of corrections.

4. Efficacy Testing.

4.1. Comparison Testing. A list of misspelled words was sent to an application that uses the Damerau-Levenshtein distance and to an application that uses the phonetic algorithm. The outputs of both applications are compared to determine the effectiveness of the phonetic algorithm.

4.2. Testing the Damerau-Levenshtein Algorithm. A common spell checking library is the Apache Common's Lucene Spell Checker. This spell checker works by using index-based spell checking – which is a method of tracking the frequency of words. Using a text file such as a book, dictionary, or other document, the spell checker tracks how many times each word appears in the file and maps the word to a number (the frequency). The Lucene Spell Checker uses the Damerau-Levenshtein minimum edit distance to find potential corrections and then checks to see if that corrected word appears frequently. The higher the rate of appearance in the file, the more probable that word is the right correction.

A text file containing a list of misspelled words, a text file containing a list of correctly spelled words, and the Lucene Spell Checker were used to test the Damerau-Levenshtein algorithm.

4.3. Testing the Phonetic Algorithm. An application, that utilizes the phonetic algorithm, finds corrected-suggestions. The same text file of misspelled words was used as input.

4.4. Misspelled Words for Testing. The list used in testing was taken from Wikipedia. That group of words was compiled and aggregated from other documented lists of "the top 100, 200, or 400 most commonly misspelled words in all variants of the English language"[6]. The 187-word list is representative of a variety of spelling errors. Some of these mistakes include but are not limited to:

- Non-word Errors: When the user accidentally hits another key or two to create a word that has no meaning or does not exist within the intended language. Example: "Hurry" being misspelled to "Hurryu."

- Real-word Errors: When the user misspells their intended word and unintentionally spells another word correctly such that a spell-checking algorithm will find nothing wrong with the spelling. Example: "Buckled" and "Bucked."

- Phonetic Errors: When the user misspells their intended word due to issues with a misconstrued sounding of the word in their minds. Example: "Proof" being misspelled to "Prufe."

It is expected that the Damerau-Levenshtein algorithm and the phonetic algorithm can both handle non-word and phonetic errors well. The phonetic implementation should fare better with phonetic errors due to its use of sound matching. Both, however, are expected to suffer when it comes to real-word errors since these words are already spelled correctly.

5. Results. The phonetic algorithm passed 185/187 of the tests and the Damerau-Levenshtein algorithm passed 184/187 of the tests.

The phonetic based spell checker seems to suffer from lack of knowledge in differentiating between homophones and true intention. "Fourth" and "Forth" are essentially equivalent to this implementation and there's no real way of figuring out what applies best without the use of contextual evidence.

It performs best with errors that arise from spelling but generally correct-sounding attempts. An interesting example demonstrating this concept can be found with misspelling "Proof" as "Prufe." The Metaphone library, which registers similar sounds, enables the algorithm to accurately deduce that the intended word is "Proof." For generic misspellings, in which the user generally sounds the word out correctly, the phonetic addition in the implementation improves the accuracy of the spell checker.

The Lucene Spell checker is reputable and well-tested implementation of the Damerau-Levenshtein algorithm; it fares well in testing common misspellings of words. The degree of severity of a misspelling is quite subjective but still relatively uniform across most users and implementations. When the misspelling becomes quite erroneous, the spell checker struggles to find the right match.

Because the phonetic algorithm allowed only one edit, the algorithm fails when two or more edits are required. An instance of this comes from the correctly spelled word "Hygiene" and "Hiygeine." From the pure point of view of editing, there is a decent amount of work that must be done to correct the spelling of Hiygeine. Most spellchecker algorithms fail when too many edits are required.

There is a more global and long-term benefit of a non-phonetic algorithm in that it's ap-

plicable to different languages. One could theoretically pass in a Spanish dictionary into the non-phonetic version of Damerau-Levenshtein algorithm and have it correct words in Spanish. The phonetic implementation is solely restricted to English.

The phonetic version of the Damerau-Levenshtein distance algorithm holds an advantage over the non-phonetic version due to simplicity and efficiency. Once the Levenshtein distance is used to create potential edits, the Metaphone library filters through potential candidates to find similar-sounding matches. This is the reason "Prufe" was successfully transformed to "Proof" on the phonetic implementation but not in the non-phonetic. The phonetic checking greatly aids the user by using their attempts at sounding out the word and so it does have a substantial impact on improving the accuracy and efficiency of spell checking when no contextual information is present.

However, the phonetic matching algorithm does lack a certain specificity when words are homophones. A non-phonetic algorithm has a higher chance of deducing the intended spelling.

6. Conclusions. There are multiple different spell-checking mechanisms available for use in open source libraries, large scale search engines, and in text editors. Several employ an implementation of the Damerau-Levenshtein distance to find the correct word. While proven accurate, this paper shows the methodology can be improved by coupling phonetic-based libraries with the standard minimum edit distance algorithm. The results of testing highlight how the phonetic checking helps to mitigate concerns with poor spelling by capitalizing on the users' attempts to spell out the word. Often, the consonants appear correctly in the string, but the vowels are the ones which are misplaced, switched, or missing and so the algorithm ignores vowels. As a result, words that sound like correctly spelled words such as "Prufe," can be easily corrected. Implementations do exist that solely rely on the results of the Metaphone or Soundex library; this impedes the accuracy of their results. By pairing the Damerau-Levenshtein distance with the Metaphone library, the best of both worlds is attained, and the results are more accurate for it.

Further steps to improve spellchecking might include providing contextual information to the spell checker to enable probability calculations. Such an approach might may resolve issues differentiating between homophones as the algorithm can analyze surrounding words to determine what is the most fitting word to utilize. With additional data, the algorithm can adopt some form of Bayesian inference or statistical machine learning to find the correct spelling suggestions.

REFERENCES

[1] P. Miller Frederic, et al., *Damerau-Levenshtein Distance.*,Alphascript Publishing, 2010.
[2] H. Larry, *Longstanding problem put to rest.*, news.mit.edu. 10 June 2015. 16 July 2019
[3] *Edit distance*, Edit distance. nlp.stanford.edu. 2008. 16 July 2019
[4] P. Norvig, *How to Write a Spelling Corrector*, 16 July 2019
[5] *Commonly misspelled English words.*, Wikipedia. 11 July 2019. Wikimedia Foundation. 16 July 2019
[6] R. Wagner, et al., *The String-to-String Correction Problem.*, Journal of the ACM (JACM). 1974. ACM. 16 July 2019
[7] J. Tregoat, *An Introduction to Fuzzy String Matching*, Medium. 09 Jan. 2018. Medium. 16 July 2019