A Framework for Distributed Approximation of Moments with Higher-Order Derivatives through Automatic Differentiation

Michel Schanen, Daniel Adrian Maldonado, and Mihai Anitescu

Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
{mschanen,maldonadod,anitescu}@anl.gov

Keywords: Uncertainty, Method of Moments, Automatic Differentiation

Abstract. We present a framework for the distributed approximation of moments, enabling the evaluation of the uncertainty in a dynamical system. The first and second moment, mean, and variance are computed with up to third-order Taylor series expansion. The required derivatives for the expansion are generated automatically by automatic differentiation and propagated through an implicit time stepper. The computational kernels are the accumulation of the derivatives (Jacobian, Hessian, tensor) and the covariance matrix. We apply distributed parallelism to the Hessian or third-order tensor, and the user merely has to provide a function for the differential equation, thus achieving similar ease of use as Monte Carlo-based methods. We demonstrate our approach using with benchmarks on Theta, a KNL-based system at the Argonne Leadership Computing Facility.

1 Introduction

Mathematical models are an approximation of real life systems and their validity resides in how well the outputs of the model agree with measured data. Often, the input or parameters of the model are uncertain because data is unavailable or inaccurate; for these cases, one typically performs an uncertainty quantification (UQ) analysis to determine how much the outputs vary with the input parameters of the model. The uncertainty in the outputs can be quantified as a range of values, but also as a probability distribution function (pdf). Several methods, for example Monte Carlo computation and polynomial chaos, try to solve the problem of computing the probability distribution of the output given parameters defined as pdf's.

A field that has experienced renewed interest in these techniques is energy systems engineering. The electrical power grid with the adoption of renewable energy has tied its behavior to stochastic weather fluctuations requiring the use of UQ techniques to predict its performance. However, the scale of these problems is

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_19

such that conventional methods are not satisfactory from a computational point of view. Monte Carlo methods can be thought of as the first-line tools for UQ; with sufficient sampling they are able to quantify the uncertainty regardless of the input distribution or the nonlinearities of the system. However, Monte Carlo methods suffer from slow convergence, which has led to the search for alternative approximations [8]. Recently, the method of moments sparked new interest as one alternative [5].

The method of moments is an approximating technique that works with the moments of probability distributions instead of their density functions. The main idea is to use a Taylor expansion of the function and write the moments of the output distribution as a polynomial function of the moments of the input distribution. Depending on the characteristics of the function, only a few terms of the Taylor expansion might be enough to achieve enough precision. As noted in [9], one of the main issues with the method of moments is that although its accuracy increases with the degree of the Taylor polynomial, computing higher-order derivatives poses serious technical challenges, leading to mostly linearization techniques for acquiring sensititivies [3].

In this paper we present ADUPROP[1], a framework developed at Argonne National Laboratory that combines the automatic differentiation (Section 2), method of moments (Section 3), uncertainty quantification, and distributed parallelism (Section 4) into an easy to use tool that is able to quantify uncertainty of dynamical systems using the method of moments at an unprecedented scale. We use automatic differentiation (AD) by overloading through a C++ template library. This flexible technique allows a straightforward augmentation of C++ codes for computing higher-order derivatives. By exploiting the structure of this approach, we implement a scheme that parallelizes both the accumulation of the derivative information and the computation of the covariance based on the derivative values.

2 Algorithmic Differentiation

Automatic differentiation [2] allows one to differentiate computer programs by applying differential calculus at a program's statement level. It uses compilers or language-based approaches to transform an implementation of a multivariate vector function $y = g(x), \mathbb{R}^n \mapsto \mathbb{R}^m$ into Jacobian vector products $y^{(1)} = J(x) \cdot x^{(1)}$ (tangent-linear model) or transposed Jacobian vector products $x_{(1)} = J^T \cdot y_{(1)}$ (adjoint model), where $x^{(1)}$, $y^{(1)}$ denotes the tangents and $x_{(1)}$, $y_{(1)}$ denotes the adjoints. The tangent-linear mode is equivalent to the finite difference method with the additional advantage of providing derivative information up to machine precision with no truncation or cancellation errors.

In this paper we solely rely on the tangent-linear or forward mode where the transformed code computes the product of the Jacobian J at point x times a directional derivative $x^{(1)}$, yielding the output tangent. The directional derivatives, denoted with a superscript order of differentiation, are defined as a partial

[1] https://gitlab.com/aduprop/aduprop

derivative of y and x with respect to an auxiliary variable s. For readability we use Spivaks notation for derivatives $y^{(1)} = \frac{\partial y}{\partial s} = \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial s} = \mathrm{D}g(x) \cdot x^{(1)} \in \mathbb{R}^m$. Letting $x^{(1)}$ go over the Cartesian basis vectors of the implementation $J \cdot x^{(1)}$ yields, column by column, the entire Jacobian $J = \mathrm{D}g(x) \in \mathbb{R}^{m \times n}$. Thus, for the *accumulation* of the full Jacobian we need to rerun the tangent-linear code n (number of columns) times. For higher-order derivative models we use the inner product $<>$ notation introduced in [6] where the tangent-linear model is written as a projection of the Jacobian onto the tangent:

$$y = g(x),\, y^{(1)} =< \mathrm{D}g(x), x^{(1)} >= \mathrm{D}g(x) \cdot x^{(1)} \,. \tag{1}$$

Note that in general an implementation transformed by an automatic differentiation (AD) tool computes both $g(x)$ and the Jacobian vector product. Reapplying an AD tool to an already first-order differentiated code yields a second-order forward over forward (FoF) code computing (2):

$$\begin{aligned} y &= g(x), & y^{(2)} &=< \mathrm{D}g(x), x^{(2)} > \\ y^{(1)} &=< \mathrm{D}g(x), x^{(1)} >, & y^{(1,2)} &=< \mathrm{D}^2 g(x), x^{(1)}, x^{(2)} > + < \mathrm{D}g(x), x^{(1,2)} > \,. \end{aligned} \tag{2}$$

The superscript $^{(2)}$ denotes the second order of differentiation. Rerunning this FoF model and letting $x^{(1)}$ and $x^{(2)}$ each go over the Cartesian basis vectors, we obtain all the entries of the Hessian $\mathrm{D}^2 g \in \mathbb{R}^{m \times n \times n}$ evaluated at x. Here $< \mathrm{D}^2 g(x), x^{(1)}, x^{(2)} >$ is the projection of $x^{(1)}$ onto the Hessian followed by the projection of $x^{(2)}$; $x^{(1,2)}$ must be set to zero. (For a detailed definition of Jacobian, Hessian and tensor projections, please refer to [6]). Following this logic, we reapply the tangent-linear model to acquire third order derivatives using the forward over forward over forward model (FoFoF):

$$\begin{aligned} y &= g(x), & y^{(3)} &=< \mathrm{D}g(x), x^{(3)} > \\ y^{(2)} &=< \mathrm{D}g(x), x^{(2)} >, & y^{(2,3)} &=< \mathrm{D}^2 g(x), x^{(2)}, x^{(3)} > + < \mathrm{D}g(x), x^{(2,3)} > \\ y^{(1)} &=< \mathrm{D}g(x), x^{(1)} >, & y^{(1,3)} &=< \mathrm{D}^2 g(x), x^{(1)}, x^{(3)} > + < \mathrm{D}g(x), x^{(1,3)} >, \end{aligned}$$

and the last term capturing the third-order tensor D^3:

$$\begin{aligned} y^{(1,2)} &=< \mathrm{D}^2 g(x), x^{(1)}, x^{(2)} > + < \mathrm{D}g(x), x^{(1,2)} >, \\ y^{(1,2,3)} &=< \mathrm{D}^3 g(x), x^{(1)}, x^{(2)}, x^{(3)} > + < \mathrm{D}^2 g(x), x^{(1,3)}, x^{(2)} > \\ &\quad + < \mathrm{D}^2 g(x), x^{(1)}, x^{(2,3)} > + < \mathrm{D}^2 g(x), x^{(1,2)}, x^{(3)} > + < \mathrm{D}g(x), x^{(1,2,3)} > \,. \end{aligned} \tag{3}$$

The original code using one variable x went up to two for the tangent-linear model, four for the FoF model and eight for the FoFoF model. To accumulate the third-order tensor $\mathrm{D}^3 g(x) \in \mathbb{R}^{m \times n \times n \times n}$ in $y^{(1,2,3)}$ we have to let $x^{(1)}$, $x^{(2)}$, and $x^{(3)}$ go over the Cartesian basis vectors, thus requiring n^3 reruns of the model. The remaining tangents of x must be set to zero. These properties will translate directly to the implementation described in Section 5.

3 Method of Moments

What is the distribution of y if we let $y = g(x)$ be a function of a random variable x with known properties? Computing this analytically is often difficult; and, in particular, obtaining the pdf of y is not possible in general. An alternative approach is to consider the moments of the distributions. Depending on the shape of the pdf, the first few moments of the pdf can be sufficient to capture relevant behavior. More concretely, consider $g(\boldsymbol{x})$ where $\boldsymbol{x}$ is a random variable with density $f(x)$. If $g(x)$ is sufficiently smooth, given the mean value theorem, we can write [7]

$$\mathbb{E}\left[g(\boldsymbol{x})\right] = \int_{\infty}^{\infty} g(x)f(x)dx \approx g(\mu)\int_{\infty}^{\infty} f(x)dx = g(\mu)\,, \tag{4}$$

where $\mu = \mathbb{E}\left[\boldsymbol{x}\right]$. Using a third order Taylor expansion for a function $g(x)$ around $\mu = \mathbb{E}\left[\boldsymbol{x}\right]$ and considering the expressions for the mean $\mu^g = \mathbb{E}\left[g(x)\right]$ and covariance
$c^g_{pg} = \mathbb{E}\left[(g_p(\boldsymbol{x}) - \mu_p)(g_q(\boldsymbol{x}) - \mu_q)\right] = \mathbb{E}\left[g_p(\boldsymbol{x})g_q(\boldsymbol{x})\right] - \mathbb{E}\left[g_p(\boldsymbol{x})\right]\mathbb{E}\left[g_q(\boldsymbol{x})\right]$, we obtain

$$\mu^g_p = g(\mu) + \frac{1}{2}\sum_{i,j=1}^{n} \mathrm{D}^2_{ij}g_p \cdot c_{ij} \tag{5}$$

and

$$\begin{aligned}
c^g_{pq} = \tfrac{1}{2!}\textstyle\sum_{i,j=1}^{n} \quad & \left(\mathrm{D}_j g_p \cdot \mathrm{D}_i g_q + \mathrm{D}_j g_q \cdot \mathrm{D}_i g_p\right) c_{ij} \\
+\tfrac{1}{4!}\textstyle\sum_{i,j,k,l=1}^{n} & \Big(\mathrm{D}_i g_p \cdot \mathrm{D}^3_{jkl} g_q + \mathrm{D}_j g_p \cdot \mathrm{D}^3_{ikl} g_q + \mathrm{D}_k g_p \cdot \mathrm{D}^3_{ijl} g_q \\
& +\mathrm{D}_l g_p \cdot \mathrm{D}^3_{ijk} g_q + \mathrm{D}^2_{ij} g_p \cdot \mathrm{D}^2_{kl} g_q + \mathrm{D}^2_{ik} g_p \cdot \mathrm{D}^2_{jl} g_q \\
& + \mathrm{D}^2_{il} g_p \cdot \mathrm{D}^2_{jk} g_q + \mathrm{D}^2_{jk} g_p \cdot \mathrm{D}^2_{il} g_q + \mathrm{D}^2_{jl} g_p \cdot \mathrm{D}^2_{ik} g_q \\
& + \mathrm{D}^2_{kl} g_p \cdot \mathrm{D}^2_{ij} g_q + \mathrm{D}^3_{jkl} g_p \cdot \mathrm{D}_i g_q + \mathrm{D}^3_{ikl} g_p \cdot \mathrm{D}_j g_q \\
& +\mathrm{D}^3_{ijl} g_p \cdot \mathrm{D}_k g_q + \mathrm{D}^3_{ijk} g_p \cdot \mathrm{D}_l g_q\Big) c_{ijkl} \\
-\tfrac{1}{2!}\textstyle\sum_{i,j,k,l=1}^{n} & \left(\mathrm{D}^2_{ij} g_p \cdot \mathrm{D}^2_{kl} g_q\right) c_{ij} c_{kl}\,.
\end{aligned} \tag{6}$$

In the derivation of these formulas we assume a Gaussian distribution, which results in the cancellation of the odd terms of the expansion.

4 Tensor Decomposition

In addition to combining the aforementioned methods and tools in a novel way to compute the moments, our main contribution is the distributed parallelization described in this section. The FoF model always computes one projection $y^{(1,2)}$ of $D^2 g \in \mathbb{R}^{m\times n\times n}$ onto $x^{(1)}, x^{(2)} \in \mathbb{R}^n$ (see Figure 1). Hence we cannot decompose the Hessian along the entries of $y^{(1,2)}$. By parallelizing over the entries of $x^{(1)}$ or $x^{(2)}$, we can restrict the Cartesian basis vectors (see Section 2) to the local indices and thus distribute the Hessian accumulation over all processes. The same holds

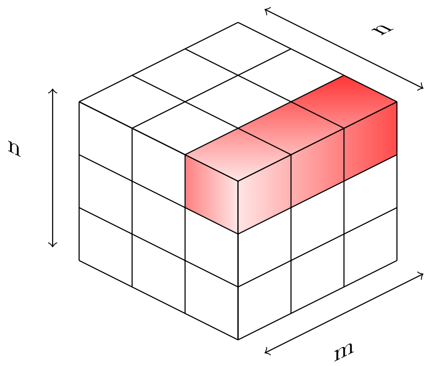


Fig. 1. Hessian $D^2 g$

true for the computation of the four-dimensional tensor $D^3 g$ (3) where we can parallelize over the entries of $x^{(1)}$, $x^{(2)}$, or $x^{(3)}$ and thus distribute the tensor over all processes.

The computation of the mean in (5) is then parallelized in a straightforward way over either the index i or j. Each process thus has a local copy of μ that needs to be *allreduced* at the end of the summation. We parallelize the computation of C over p or q and perform *allgather* to share C among all processes. Solving the linear system for the time stepper has a runtime complexity of at most $\mathcal{O}\left(n^3\right)$, or lower in case of sparsity in the inner Jacobian. Accumulating the tensor $D^3 g$ has a runtime complexity of $\mathcal{O}\left(n^2\right) \cdot cost(g)$. Thus, the total run-time complexity for accumulating the tensor is $\mathcal{O}\left(n^6/p\right)$, where p is the number of processes. The covariance computation in (6) yields the same complexity of $\mathcal{O}\left(n^6/p\right)$, which is our global runtime complexity.

5 Implementation

ADUPROP (AD for Uncertainty Propagation) is a C++ implementation of the propagation of moments. In particular, it implements the concepts of Section 3 in the context of differential equations. ADUPROP is a template-based code that allows easy computation of higher-order derivatives. The library provides vector, matrix, and tensor data structures using a template type `T` instead of `double`. For our implementation of ADUPROP we chose the AD tool CoDiPack, which is based on operator overloading. To create an $n + 1$ derivative type `t3s` we recursively apply differentiation on an n order type.

```
typedef RealForwardGen<RealForwardGen<RealForwardGen<double
    > > > t3s;
```

This maps exactly to the notation in Section 2. Accessing, for example, `x.gradient().gradient().value()` of the third-order type `t3s` of a variable `x` is equivalent to accessing $\boldsymbol{x}^{(2,3)}$. The variable type `T` in the implementation of f allows us to instantiate the function using the types `double`, `t1s`, `t2s`, and `t3s`.

As an example, we implement UQ of a differential equation system with ADUPROP. When the system is discretized ($x_k = \phi(x_{k-1})$), the procedure is identical to the one described in Section 3. The default integration scheme that we use is backward Euler. One of the main advantages of using AD is that we can differentiate through functions, loops, or other complex functions in which obtaining explicit derivatives might be practically challenging and tedious. The integration loop is written as follows

```
pVector<T> xold(dim), yold(dim), y(dim), res(dim);
pMatrix<T> J(dim, dim), Jold(dim, dim);
xold = x;
sys->residual_beuler<T>(x, xold, y);
do {
  sys->jac_beuler<T>(x, xold, J);
  yold = y; Jold = J;
  adlinsolve<T>(J, y);
  res = Jold * y - yold;
  x = x - y;
  sys->residual_beuler<T>(x, xold, y);
} while (y.norm() > eps);
```

The object `sys`, defined by the user, has to contain the residual function `residual_beuler` and the Jacobian `jac_beuler`. The function `adlinsolve` provides an interface to linear solvers. Currently we support BLAS and Eigen for dense and sparse linear systems, respectively. For an order of differentiation k we differentiate $Ax = b$. Let $A(s) \in \mathbb{R}^{n\times n}$, $b(s) \in \mathbb{R}^n$, and $x(s) \in \mathbb{R}^n$ with s being some input dependency. We define $\frac{\partial^k A}{\partial s^k} = A_k$, $\frac{\partial^k b}{\partial s^k} = b_k$, and $\frac{\partial^k x}{\partial s^k} = x_k$. With $Ax = b$, we have

$$c_k \cdot A \cdot x^{(k)} = b^{(k)} - c_0 \cdot A^{(k)} \cdot x - c_1 \cdot A^{(k-1)} x^{(1)} - \ldots - c_{k-1} \cdot A^{(1)} \cdot x^{(k-1)}. \quad (7)$$

In summary, we have to solve 2, 4, and 8 linear systems for first-, second-, and third-order derivatives, respectively. With these three basic blocks in place, a time stepper, residual function, Jacobian, and linear system, we can compute the Jacobian, Hessian, and third order derivative tensor using the logic described in Section 2.

6 Scalability

A prototype sequential implementation of this method in Julia was tested on power system dynamics in [5]. To assess the scaling and computational capabilities of ADUPROP, we resort to a well-known dampened nonlinear dynamical system used in the weather simulation community and elsewhere [4, **?**,**?**]:

$$\dot{x}_i = x_{i-1}\left(x_{i+1} - x_{i-2}\right) - x_i + F,\ i = 1, \ldots, n > 3\,. \quad (8)$$

This system is known to show chaotic behavior when the *forcing term* $F \geq 8$, having an equilibrium $(F, \ldots, F)$ that becomes unstable for all $n \geq 4$. The system transitions from a damped, constant-valued system to a traveling wave with a periodic attractor, and eventually to chaotic behavior, all adjustable through the selection of F.

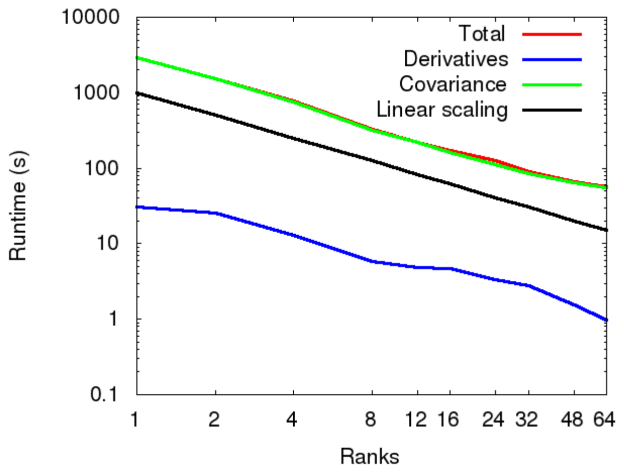
The scalability study was done on Theta at the Argonne Leadership Computing Facility. Theta is composed of 1.3 GHz Intel Xeon Phi 7230 SKU nodes with 64 cores each. Our goal was to achieve strong scaling on a single node with up to 64 MPI processes. Our focus is on the strong scaling of the covariance computation using Hessians and third-order derivative tensors. We use the same $F = 4.4$ forcing but increase the dimension to $N = 64$ for third-order derivatives and $N = 512$ for second-order derivatives. The time horizon is irrelevant to the scaling, since there is no parallelization in time. With the timestep set to 1 we achieved the strong-scaling results in Figure 2. The black line serves as a reference point for linear scaling. We show that our implementation scales nearly linearly with up to 64 cores, with both second- and third-order derivatives. As anticipated by our complexity analysis in Section 4, the covariance computation dominates the runtime with second-order derivatives, whereas with third-order derivatives both the tensor accumulation and covariance computation are much closer.

Each KNL node has 64 cores, limiting the strong scaling to one row or projection of the derivative tensor. Our code is able to run beyond a single node, but the runtime cost of computing the derivative tensor becomes too high. In the future we will investigate low-rank approximations to compress the derivative tensor and decrease its computational cost [1].

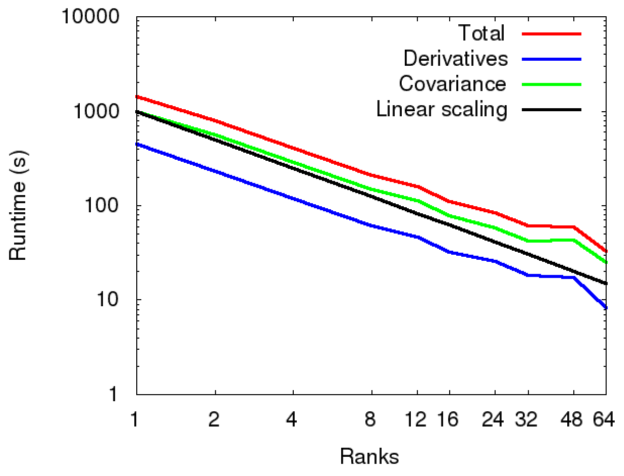
We validate the approximation of the variance propagation in Figure 3 with dimension $N = 10$ and in a nonlinear regime with $F = 5$. The higher-order derivatives allow us to better capture the effects of nonlinearity in power systems [5]. The numerical aspects of this research will be subject of future research.

7 Conclusion

This paper describes a distributed parallel framework for using the method of moments backed with AD. The extension to third-order derivatives and the parallelization over the derivative and covariance accumulation is unprecedented at this scale and speed. The distribution of the Hessian and tensor is chosen such that AD and the covariance computation benefit from the parallelism. Nonetheless, the computational cost grows at a factor of $\mathcal{O}\left(N^6\right)$, N being the dimension, times the original simulation evaluation using third-order derivatives. This factor is reduced to $\mathcal{O}\left(N^4\right)$ while using only the Hessian. In both cases we have shown the scalability on the current Intel KNL architecture. As opposed to Monte Carlo, this algorithm provides an analytical propagation workflow that showed promising results in [5]. To scale beyond a single KNL node, future research will focus on exploiting Hessian, tensor and covariance matrix structure in order to apply sampling methods. This approach has shown promising results in machine learning [10] and we plan to integrate this approach in our software. While being highly problem dependent, it has the potential to significantly reduce the com-

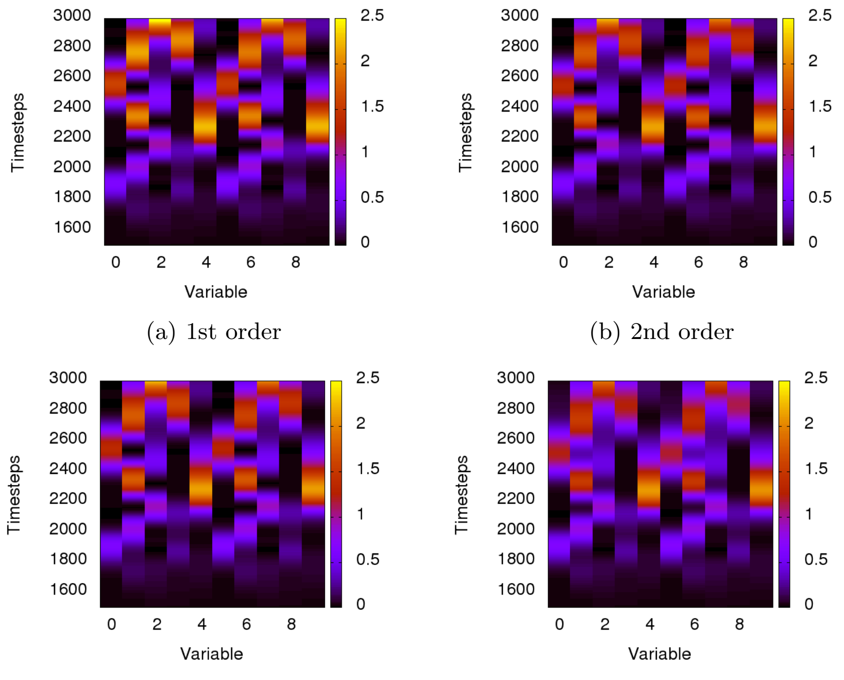


(a) Second order derivatives $N = 512$, $F = 4.4$



(b) Third order derivatives $N = 64$, $F = 4.4$

Fig. 2. Strong Scaling



(a) 1st order (b) 2nd order

(c) 3rd order (d) Monte Carlo

Fig. 3. Approximation of the variance propagation from timestep 1,500 to 3,000 (h=0.001) for $N = 10$ $F = 5$ using first-order, second-order and third-order derivatives in addition to fully converged Monte Carlo at 1,000 samples.

plexity of the covariance computation. In particular, at higher dimensions, this method of propagating uncertainties may become a valuable alternative to the Monte Carlo-based sampling methods.

References

1. Abdel-Khalik, H.S., Hovland, P.D., Lyons, A., Stover, T.E., Utke, J.: A low rank approach to automatic differentiation. In: Advances in Automatic Differentiation, pp. 55–65. Springer (2008)
2. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 105, SIAM, Philadelphia, PA, 2nd edn. (2008)
3. Hiskens, I.A., Alseddiqui, J.: Sensitivity, approximation, and uncertainty in power system dynamic simulation. IEEE Transactions on Power Systems **21**(4), 1808–1820 (Nov 2006). https://doi.org/10.1109/TPWRS.2006.882460
4. Lorenz, E.: Predictability: a problem partly solved. In: Seminar on Predictability, 4-8 September 1995. vol. 1, pp. 1–18. ECMWF, Shinfield Park, Reading (1995)
5. Maldonado, D.A., Schanen, M., Anitescu, M.: Uncertainty propagation in power system dynamics with the method of moments. In: 2018 IEEE Power Energy Society General Meeting (PESGM). pp. 1–5 (Aug 2018). https://doi.org/10.1109/PESGM.2018.8586023
6. Naumann, U.: The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation, vol. 24. SIAM (2012)
7. Papoulis, A.: Probability, Random Variables and Stochastic Processes. McGraw-Hill (1965)
8. Preece, R., Milanovi, J.V.: Efficient estimation of the probability of small-disturbance instability of large uncertain power systems. IEEE Transactions on Power Systems **31**(2), 1063–1072 (March 2016). https://doi.org/10.1109/TPWRS.2015.2417204
9. Smith, R.C.: Uncertainty Quantification: Theory, Implementation, and Applications, vol. 12. SIAM (2013)
10. Xu, P., Roosta-Khorasan, F., Mahoney, M.W.: Second-order optimization for non-convex machine learning: An empirical study. arXiv preprint arXiv:1708.07827 (2017)

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11347.