Differential Methods for Assessing Sensitivity in Biological Models

Rachel Mester[1], Alfonso Landeros[1], Chris Rackauckas[4,5], Kenneth Lange[1,2,3]

Departments of Computational Medicine[1],
Human Genetics[2], and Statistics[3]
University of California Los Angeles, Los Angeles, CA 90095
Massachusetts Institute of Technology[4] and Pumas-AI[5]
Phone: 310-206-8076
E-mail klange@ucla.edu

October 18, 2021

Abstract

Differential sensitivity analysis is indispensable in fitting parameters, understanding uncertainty, and forecasting the results of both thought and lab experiments. Although there are many methods currently available for performing differential sensitivity analysis of biological models, it can be difficult to determine which method is best suited for a particular model. In this paper, we explain a variety of differential sensitivity methods and assess their value in some typical biological models. First, we explain the mathematical basis for three numerical methods: adjoint sensitivity analysis, complex-perturbation sensitivity analysis, and forward-mode sensitivity analysis. We then carry out four instructive case studies. (i) The CARRGO model for tumor-immune interaction highlights the additional information that differential sensitivity analysis provides beyond traditional naive sensitivity methods, (ii) the deterministic SIR model demonstrates the value of using second-order sensitivity in refining model predictions, (iii) the stochastic SIR model shows how differential sensitivity can be attacked in stochastic modeling, and (iv) a discrete birth-death-migration model illustrates how the complex-perturbation method of differential sensitivity can be generalized to a broader range of biological models. Finally, we compare the speed, accuracy, and ease of use of these methods. We find that forward-mode automatic differentiation has the quickest computation time, while the complex-perturbation method is the simplest to implement and the most generalizable.

Author Summary

Over the past few decades, mathematical modeling has become an indispensable tool in the biologist's toolbox. From deterministic to stochastic to statistical models, computational modeling is ubiquitous in almost every field of biology. Because model parameter estimates are often noisy or depend on poorly understood interactions, it is crucial to examine how both quantitative and qualitative predictions change as parameter estimates change, especially as the number of parameters increase. Sensitivity analysis is the process of understanding how a model's behavior depends on parameter values. Sensitivity analysis simultaneously quantifies prediction certainty and clarifies the underlying biological mechanisms that drive computational models. While sensitivity analysis is universally recognized to be an important step in modeling, it is often unclear how to best leverage the available differential sensitivity methods. In this manuscript we explain and compare various differential sensitivity methods in the hope that best practices will be widely adopted. In particular, we stress the relative advantages of existing software and their limitations. We also present a new numerical technique for computing differential sensitivity.

1 Introduction

In many mathematical models underlying parameters are poorly specified. This problem is particularly acute in biological and biomedical models. Model predictions can have profound implications for scientific understanding, further experimentation, and even public policy decisions. For instance, in an epidemic some model parameters can be tweaked by societal or scientific interventions to drive infection levels down. Differential sensitivity

can inform medical judgement about the steps to take with the greatest impact at the least cost. Similar considerations apply in economic modeling. Finally, model fitting usually involves maximum likelihood or least squares criteria. These optimization techniques depend heavily on gradients and and Hessians with respect to parameters.

In any case it is imperative to know how sensitive model predictions are to changes in parameter values. Unfortunately, assessment of model sensitivity can be time consuming, computationally intensive, inaccurate, and simply confusing. Most models are nonlinear and resistant to exact mathematical analysis. Understanding their behavior is only approachable by solving differential equations or intensive and noisy simulations. Sensitivity analysis is often conducted over an entire bundle of neighboring parameters to capture interactions. If the parameter space is large or high-dimensional, it is often unclear how to choose representative points from this bundle. Faced with this dilemma, it is common for modelers to fall back on varying just one or two parameters at a time. Model predictions also often take the form of time trajectories. In this setting, sensitivity analysis is based on lower and upper trajectories bounding the behavior of the dynamical system.

The differential sensitivity of a model quantity is measured by its gradient with respect to the underlying parameters at their estimated values. Given the gradient of a function $f(\boldsymbol{\beta})$, it is possible to approximate $f(\boldsymbol{\beta})$ by the linear function $f(\boldsymbol{\beta}_0) + \nabla f(\boldsymbol{\beta}_0)^t(\boldsymbol{\beta} - \boldsymbol{\beta})$ for all $\boldsymbol{\beta}$ near $\boldsymbol{\beta}_0$. The existing literature on differential sensitivity is summarized in the modern references [1, 2]. The Julia software DifferentialEquations.jl [3] makes sensitivity analysis fairly routine for many problems. Although the physical sciences have widely adopted the method of differential sensitivity [4, 5], the papers and

software generally focus on a single sensitivity analysis method rather than a comparison of the various approaches. For example, PESTO [6] is a current Matlab toolbox for parameter estimation that uses adjoint sensitivities implemented as part of the CVODES method from SUNDIALS [7]. Should the continuous sensitivity equations be used, or would direct automatic differentiation of the solvers be more efficient on biological models? On the types of models biologists generally explore, would implicit parallelism within the sensitivity equations be beneficial, or would the overhead cost of thread spawning overrule any benefits? How close do simpler methods based on complex step differentiation get to these techniques? The purpose of the current paper is to explore these questions on a variety of models of interest to computational biologists.

In the current paper we also suggest a more accurate method of approximating gradients for models involving analytic functions without discontinuities, maxima, minima, absolute values or any other excursion outside the universe of analytic functions. By definition an analytic function can be expanded in a locally convergent power series around every point of its domain. In the sections immediately following, we summarize known theory, including the important adjoint method for computing the sensitivity of functions of solutions [4, 5]. Then we illustrate sensitivity analysis for a few deterministic models and a few stochastic models. Our exposition includes some straightforward Julia code that readers can adapt to their own sensitivity needs. These examples are followed by an evaluation of the accuracy and speed of the suggested numerical methods. The concluding discussion summarizes our experience, indicates limitations of the methods, and suggests new potential applications.

For the record, here are some notational conventions used throughout the

paper. All functions that we differentiate have real or real vector arguments and real or real vector values. All vectors and matrices appear in boldface. The superscript ^{t} indicates a vector or matrix transpose. For a smooth real-valued function $f(\boldsymbol{x})$, we write its gradient (column vector of partial derivatives) as $\nabla f(\boldsymbol{x})$ and its differential (row vector of partial derivatives) as $df(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^t$. If $g(\boldsymbol{x})$ is vector-valued with ith component $g_i(\boldsymbol{x})$, then the differential (Jacobi matrix) $dg(\boldsymbol{x})$ has ith row $dg_i(\boldsymbol{x})$. The chain rule is expressed as the equality $d[f \circ g(\boldsymbol{x})] = df[g(\boldsymbol{x})]dg(\boldsymbol{x})$ of differentials. The transpose (adjoint) form of the chain rule is $\nabla f \circ g(\boldsymbol{x}) = dg(\boldsymbol{x})^t \nabla f[g(\boldsymbol{x})]$. For a twice differentiable function, the second differential (Hessian matrix) $d^2 f(\boldsymbol{x}) = d\nabla f(\boldsymbol{x})$ is the differential of the gradient. Finally, i will denote $\sqrt{-1}$.

2 Methods for Computing Sensitivity

The simplest dynamical models are governed by the linear constant coefficient differential equation $\frac{d}{dt}\boldsymbol{x}(t) = \boldsymbol{A}(\boldsymbol{\beta})\boldsymbol{x}(t)$ with solution $\boldsymbol{x}(t) = e^{t\boldsymbol{A}(\boldsymbol{\beta})}\boldsymbol{x}_0$. The directional derivative of the matrix exponential $e^{\boldsymbol{B}}$ in the direction $\boldsymbol{V}$ can be represented by the integral

$$d_{\boldsymbol{V}} e^{\boldsymbol{B}} = \int_0^1 e^{s\boldsymbol{B}} \boldsymbol{V} e^{(1-s)\boldsymbol{B}} ds.$$

A proof of this fact appears in Example 3.2.2 of reference [8]. Setting $\boldsymbol{B} = t\boldsymbol{A}(\boldsymbol{\beta})$ and applying the chain rule leads to the partial derivative

$$\begin{aligned} \frac{\partial}{\partial \beta_j} e^{t\boldsymbol{A}(\boldsymbol{\beta})}\boldsymbol{x}(0) &= \int_0^1 e^{st\boldsymbol{A}(\boldsymbol{\beta})} t \frac{\partial}{\partial \beta_j}\boldsymbol{A}(\boldsymbol{\beta}) e^{(1-s)t\boldsymbol{A}(\boldsymbol{\beta})} ds\, \boldsymbol{x}(0) \\ &= \int_0^t e^{s\boldsymbol{A}(\boldsymbol{\beta})} \frac{\partial}{\partial \beta_j}\boldsymbol{A}(\boldsymbol{\beta}) e^{(t-s)\boldsymbol{A}(\boldsymbol{\beta})} ds\, \boldsymbol{x}(0), \end{aligned}$$

which can be laboriously evaluated by numerical integration. Simplification into a sum of exponentials is possible if $\boldsymbol{A}(\boldsymbol{\beta})$ is uniformly diagonalizable across all $\boldsymbol{\beta}$ [9], but the details are messy.

In practice, it is simpler to differentiate the original ODE with respect to β_j, interchange the order of differentiation, and numerically integrate the system

$$\frac{d}{dt}\frac{\partial}{\partial\beta_j}\boldsymbol{x}(t,\boldsymbol{\beta}) = \frac{\partial}{\partial\beta_j}\boldsymbol{A}(\boldsymbol{\beta})\boldsymbol{x}(t,\boldsymbol{\beta}) + \boldsymbol{A}(\boldsymbol{\beta})\frac{\partial}{\partial\beta_j}\boldsymbol{x}(t,\boldsymbol{\beta})$$

from 0 to some final value of t. The initial condition $\boldsymbol{x}(0,\boldsymbol{\beta}) = \boldsymbol{x}(0)$ remains intact, and the new condition $\nabla_{\boldsymbol{\beta}}\boldsymbol{x}(0,\boldsymbol{\beta}) = 0$ is added. The second method has the advantage of giving the sensitivity along the entire trajectory. Another huge advantage is that the second method generalizes to nonlinear systems $\frac{d}{dt}\boldsymbol{x}(t,\boldsymbol{\beta}) = f[\boldsymbol{x}(t),\boldsymbol{\beta}]$ under the right circumstances. In particular the sensitivity of this more elaborate system can be evaluated by solving the differential equation

$$\frac{d}{dt}\frac{\partial}{\partial\beta_j}\boldsymbol{x}(t,\boldsymbol{\beta}) = \frac{\partial}{\partial\beta_j}f[\boldsymbol{x}(t),\boldsymbol{\beta}] + d_{\boldsymbol{x}}f[\boldsymbol{x}(t),\boldsymbol{\beta}]\frac{\partial}{\partial\beta_j}\boldsymbol{x}(t,\boldsymbol{\beta})$$

numerically. This sensitivity equation depends on knowing $\boldsymbol{x}(t,\boldsymbol{\beta})$. In practice, one solves the system

$$\frac{d}{dt}\begin{bmatrix}\boldsymbol{x}(t,\boldsymbol{\beta})\\ \nabla_{\boldsymbol{\beta}}\boldsymbol{x}(t,\boldsymbol{\beta})\end{bmatrix} = \begin{bmatrix}f[\boldsymbol{x}(t),\boldsymbol{\beta}]\\ \nabla_{\boldsymbol{\beta}}f[\boldsymbol{x}(t),\boldsymbol{\beta}] + d_{\boldsymbol{\beta}}\boldsymbol{x}(t,\boldsymbol{\beta})^t\nabla_{\boldsymbol{x}}f[\boldsymbol{x}(t),\boldsymbol{\beta}]\end{bmatrix} \quad (1)$$

jointly, where $d_{\boldsymbol{\beta}}\boldsymbol{x}[t,\boldsymbol{\beta}]$ is the Jacobi matrix of $\boldsymbol{x}(t,\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$. This is commonly referred to as forward sensitivity analysis and is carried out by software suites such as DifferentialEquations.jl [3] and SUNDIALS

CVODES [7]. We note in passing that a common implementation detail of sensitivity analysis is to base calculations on directional derivatives. Thus, the directional derivative

$$d_{\boldsymbol{\beta}}\boldsymbol{x}(t,\boldsymbol{\beta})^t\nabla_{\boldsymbol{x}}f[\boldsymbol{x}(t),\boldsymbol{\beta}] \quad = \quad \lim_{\epsilon\to 0}\frac{f\{\boldsymbol{x}(t)+\epsilon\nabla_{\boldsymbol{x}}f[\boldsymbol{x}(t),\boldsymbol{\beta}],\boldsymbol{\beta}\}-f[\boldsymbol{x}(t),\boldsymbol{\beta}]}{\epsilon}$$

version of the chain rule allows one to evolve dynamical systems without ever computing full Jacobians.

The well-known adjoint method, another variant of the sensitivity method, is incorporated in the biological parameter estimation software PESTO through CVODES [7]. The adjoint method [1, 2] is defined directly on a function $g[x(\boldsymbol{\beta}),\boldsymbol{\beta}]$ of the solution of the ODE. The adjoint method first numerically solves the ODE system forward in time over $[t_0,t_n]$, then solves the system

$$d_{\boldsymbol{\beta}}\lambda(\boldsymbol{\beta}) \quad = \quad d_{\boldsymbol{x}}f[\boldsymbol{x}(\boldsymbol{\beta}),\boldsymbol{\beta}]\lambda(\boldsymbol{\beta})+d_{\boldsymbol{\beta}}g[\boldsymbol{x}(\boldsymbol{\beta}),\boldsymbol{\beta}],$$

in reverse time, and finally computes derivatives via

$$d_{\boldsymbol{\beta}}g[x(\boldsymbol{\beta}),\boldsymbol{\beta}] \quad = \quad \int_{t_0}^{t_n}\lambda(t,\boldsymbol{\beta})d_{\boldsymbol{\beta}}\boldsymbol{x}(t,\boldsymbol{\beta})dt.$$

The second and third stages are commonly combined by appending the last equation to the set of ODEs being solved in reverse. This tactic achieves a lower computational complexity than other techniques, which require solving an n-dimensional ODE system p times for p parameters. In contrast, the adjoint method solves an n-dimensional ODE forwards and then solves an n-dimensional and a p-dimensional system in reverse, changing the computational complexity from $\mathcal{O}(np)$ to $\mathcal{O}(n+p)$. Whether such asymptotic

cost advantages lead to more efficiency on practical models is precisely one of the points studied in this paper.

Alternatively, one can find the partial derivatives purely numerically. The obvious method here is to compute a slightly perturbed trajectory $\boldsymbol{x}(t, \boldsymbol{\beta} + \Delta \boldsymbol{v})$ and form the forward differences

$$\frac{\boldsymbol{x}(t, \boldsymbol{\beta} + \Delta \boldsymbol{v}) - \boldsymbol{x}(t, \boldsymbol{\beta})}{\Delta}$$

at all specified time points as approximations to the forward directional derivatives of $\boldsymbol{x}(t, \boldsymbol{\beta})$ in the direction $\boldsymbol{v}$. Choosing $\boldsymbol{v}$ to be unit vectors along each coordinate axis gives ordinary partial derivatives. The accuracy of this crude method suffers from round-off error in subtracting two nearly equal function values. These round-off errors are in addition to the usual errors committed in integrating the differential equation numerically. Round-off errors can be ameliorated by using central differences rather than forward differences, but this doubles computation time and still leaves the difficult choice of the stepsize Δ.

There is a far more accurate way when the function $f[\boldsymbol{x}, \boldsymbol{\beta}]$ defining the ODE is analytic in the parameter vector $\boldsymbol{\beta}$ [10]. This implies that the trajectory $\boldsymbol{x}(t, \boldsymbol{\beta})$ is also analytic in $\boldsymbol{\beta}$. For a real analytic function $g(\beta)$ of a single variable β, the derivative approximation

$$g'(\beta) = \frac{\text{Imag}\, g(\beta + \Delta i)}{\Delta} + O(\Delta^2) \tag{2}$$

in the complex plane avoids roundoff and is highly accurate for $\Delta > 0$ very small [11, 12]. Thus, in calculating a directional derivative of $\boldsymbol{x}(t, \boldsymbol{\beta})$, it suffices to (a) solve the governing ODE $\frac{d}{dt}\boldsymbol{x}(t, \boldsymbol{\beta}) = f[\boldsymbol{x}(t), \boldsymbol{\beta}]$ with $\boldsymbol{\beta} + \Delta i \boldsymbol{v}$

replacing $\boldsymbol{\beta}$, (b) take the imaginary part of the result, and (c) divide by Δ. Of course to make these calculations feasible, the computer language implementing the calculations should support complex arithmetic and ideally have an automatic dispatching mechanism so that only one implementation of each function is required. In contrast to numerical integration of the joint system (1), the analytic method is embarrassingly parallelizable across parameters.

The following straightforward Julia routine for computing sensitivities

```
function differential(f::F, p, d) where F
  fvalue = real(f(p)) # function value
  df = zeros(length(fvalue), 0) # empty differential
  for j = 1:length(p)
    p[j] = p[j] + d * im # perturb parameter
    fj = f(p) # compute perturbed function value
    p[j] = complex(real(p[j]), 0.0) # reset parameter
    df  = [df imag(fj) ./ d] # concatenate jth partial
  end
  return (fvalue, df) # return the differential
end
```

requires a function $f(\boldsymbol{p}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ of a real vector $\boldsymbol{p}$ declared as complex. The perturbation scalar d should be small and real, say 10^{-10} to 10^{-12} in double precision. If the parameters p_j vary widely in magnitude, then a good heuristic is to perturb p_j by $p_j di$ instead of di. The returned value df is an $m \times n$ real matrix. The Julia commands `real` and `complex` effect conversions between real and complex numbers, and Julia substitutes im for $i = \sqrt{-1}$. We will employ these functions later in some case studies.

A recent extension [13] of the analytic method facilitates accurate ap-

proximation of second derivatives. The relevant formula is

$$\frac{\partial^2}{\partial\beta_j^2}g(\boldsymbol{\beta}) = \frac{\text{Imag}[g(\boldsymbol{\beta}+e^{\pi i/4}\Delta\boldsymbol{e}_j)+g(\boldsymbol{\beta}-e^{\pi i/4}\Delta\boldsymbol{e}_j)]}{\Delta^2}+O(\Delta^4), \quad (3)$$

where $e^{\pi i/4}=(1+i)/\sqrt{2}$. Roundoff errors can now occur, but are usually manageable. Recently we discovered how to extend the analytic method to approximate mixed partials. Our derivation is condensed into the following equations

$$\begin{aligned}
g[\boldsymbol{x}+e^{\pi i/4}\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)] &\approx g(\boldsymbol{x})+e^{\pi i/4}dg(\boldsymbol{x})\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)\\
&\quad +\frac{i}{2}d^2g(\boldsymbol{x})[\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]^2\\
&\quad +\frac{e^{\pi 3/4}}{6}d^3g(\boldsymbol{x})[\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]^3\\
g[\boldsymbol{x}-e^{\pi i/4}\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)] &\approx g(\boldsymbol{x})-e^{\pi i/4}dg(\boldsymbol{x})\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)\\
&\quad +\frac{i}{2}d^2g(\boldsymbol{x})[\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]^2\\
&\quad -\frac{e^{\pi 3/4}}{6}d^3g(\boldsymbol{x})[\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]^3.
\end{aligned}$$

This approximation is accurate to order $O(\Delta^6)$ and allows us to infer that

$$\begin{aligned}
\frac{\text{Imag}\{g[\boldsymbol{x}+e^{\pi i/4}\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]+g[\boldsymbol{x}-e^{\pi i/4}\Delta(\boldsymbol{e}_j+\boldsymbol{e}_k)]\}}{\Delta^2} &=\\
d^2g(\boldsymbol{x})[(\boldsymbol{e}_j+\boldsymbol{e}_k)]^2+O(\Delta^4) &= \quad (4)\\
\frac{\partial^2}{\partial\beta_j^2}g(\boldsymbol{\beta})+\frac{\partial^2}{\partial\beta_k^2}g(\boldsymbol{\beta})+2\frac{\partial^2}{\partial\beta_j\partial\beta_k}g(\boldsymbol{\beta})+O(\Delta^4) &
\end{aligned}$$

Since we can approximate $\frac{\partial^2}{\partial\beta_j^2}g(\boldsymbol{\beta})$ and $\frac{\partial^2}{\partial\beta_k^2}g(\boldsymbol{\beta})$, we can now approximate $\frac{\partial^2}{\partial\beta_j\partial\beta_k}g(\boldsymbol{\beta})$ to order $O(\Delta^4)$. These approximations are derived in the Appendix.

The Julia code for computing second derivatives

```
function hessian(f::F, p, d) where F
  d2f = zeros(length(p), length(p)) # hessian
  dp = d * (1.0 + 1.0 * im) / sqrt(2)
  for j = 1:length(p) # compute diagonal entries of d2f
    p[j] = p[j] + dp
    fplus = f(p)
    p[j] = p[j] - 2 * dp
    fminus = f(p)
    p[j] = complex(real(p[j]), 0.0) # reset parameter
    d2f[j, j]  = imag(fplus + fminus) / d^2
  end
  for j = 2:length(p) # compute off diagonal entries
    for k = 1:(j - 1)
      (p[j], p[k]) = (p[j] + dp, p[k] + dp)
      fplus = f(p)
      (p[j], p[k]) = (p[j] - 2 * dp, p[k] - 2 * dp)
      fminus = f(p)
      (p[j], p[k]) = (complex(real(p[j]), 0.0), complex(real(p[k]), 0.0))
      d2f[j, k]  = imag(fplus + fminus) / d^2
      d2f[j, k] = (d2f[j, k] - d2f[j, j] - d2f[k, k]) / 2
      d2f[k, j] = d2f[j, k]
    end
  end
  return d2f
end
```

operates on a scalar-valued function $f(u)$ of a real vector $\boldsymbol{p}$ declared as complex. Because roundoff error is now a concern, the perturbation scalar d should be in the range 10^{-3} to 10^{-6} in double precision. The returned

value $d^2 f$ is a symmetric matrix.

Another technique one can use to calculate the derivatives of model solutions is to differentiate the numerical algorithm that calculates the solution. This can be done with computational tools collectively known as automatic differentiation [CITE]. Forward mode automatic differentiation is performed by carrying forward the Jacobian-vector product at each successive calculation. This is accomplished by defining higher-dimensional numbers, known as dual numbers [14], coupled to primitive functions $f(\boldsymbol{x})$ through the action

$$f(\boldsymbol{a} + \boldsymbol{b}\epsilon) \quad = \quad f(\boldsymbol{a}) + \epsilon df(\boldsymbol{a})\boldsymbol{b}.$$

Here ϵ is a dimensional marker, similar to the complex i, which is a two-dimensional number. For a composite function $f = f_2 \circ f_1$, the chain rule is $df(\boldsymbol{a})\boldsymbol{b} = df_2[f_1(\boldsymbol{a})][df_1(\boldsymbol{a})\boldsymbol{b}]$. The ith column of the Jacobian appears in the expression $f(\boldsymbol{x}+\boldsymbol{e}_i\epsilon) = f(\boldsymbol{x})+\epsilon\nabla_i f(\boldsymbol{x})$. Since computational algorithms can be interpreted as the composition of simpler functions, one need only define the algorithm on a small set of base cases (such as $+$, $*$, sin, and so forth, known as the primitives) and then apply the accepted rules in sequence to differentiate more elaborate functions. The ForwardDiff.jl package in Julia does this by defining dispatches for such primitives on a dual number type and provides convenience functions for easily extracting common objects like gradients, Jacobians, and Hessians. Hessians are calculated by layering automatic differentiation twice on the same algorithm to effectively take the derivative of a derivative.

In this form, forward mode automatic differentiation shares many similarities to the analytic methods described above without the requirement that the extension of $f(\boldsymbol{x})$ be complex analytic. Of course, at every stage of

the calculation $f(\boldsymbol{x})$ must be differentiable. The pros and cons of this weaker but still restrictive assumption will be discussed in the results section. Conveniently, automatic differentiation allows for arbitrarily many derivatives to be calculated simultaneously. By defining higher-dimensional dual numbers that act independently via $f(\boldsymbol{a}+\sum_i b_i\epsilon_i) = f(\boldsymbol{a})+\sum \epsilon_i df(\boldsymbol{a})\boldsymbol{b}_i$, one can calculate entire Jacobians in a single function call $f(\boldsymbol{a}+\sum_i \boldsymbol{e}_i\epsilon_i)$. This use of higher-dimensional dual numbers is a practice known as chunking. Chunking reduces the number of primal (non-derivative) calculations required for computing the Jacobian. Because the ForwardDiff.jl package uses chunking by default, we will investigate the extent to which this detail is applicable in biological models.

3 Case Studies

We now explore applications of differential sensitivity to a few core models in oncology and epidemiology.

3.1 CARRGO Model

The CARRGO model [15] was designed to capture the tumor-immune dynamics of CAR T-cell therapy in glioma. The CARRGO model generalizes to other tumor cell-immune cell interactions. Its governing system of ODEs

$$\begin{aligned} \frac{dx}{dt} &= \rho x\left(1-\frac{y}{\gamma}\right) - \kappa_1 xy \\ \frac{dy}{dt} &= \kappa_2 xy - \theta y \end{aligned}$$

follows cancer cells x as prey and CAR T-Cells y as predators. This model captures Lotka-Volterra dynamics with logistic growth of the cancer cells.

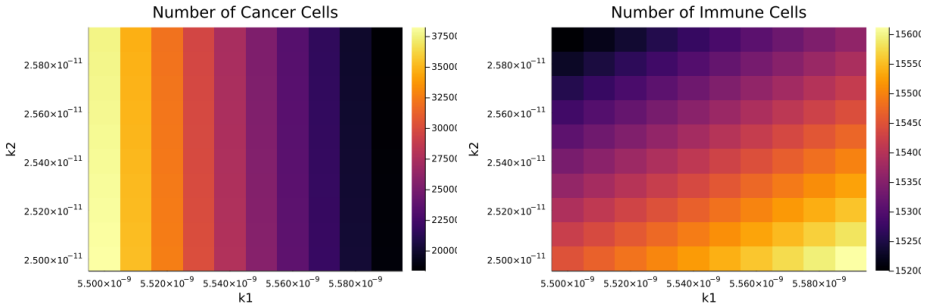

Figure 1: Sensitivity of Cancer Cells in the CARRGO Model

Our numerical experiments assume the parameter values and initial conditions

$$\begin{aligned} \kappa_1 &= 6 \times 10^{-9}/(\text{day} \times \text{cell}), \quad \kappa_2 = 3 \times 10^{-11}/(\text{day} \times \text{cell}), \\ \theta &= 1 \times 10^{-6}/\text{day}, \quad \rho = 6 \times 10^{-2}/\text{day}, \quad \gamma = 1 \times 10^{9} \text{ cells}, \\ x_0 &= 1.25 \times 10^{4} \text{ cells}, \quad y_0 = 6.25 \times 10^{2} \text{ cells} \end{aligned}$$

suggested by Sahoo et al. [15].

A traditional sensitivity analysis hinges on solving the system of ODEs for a variety of parameters and displaying the solution at a chosen future time across an interval or rectangle of parameter values. Figure 1 shows how $x(t)$ and $y(t)$ vary at $t = 1000$ days under joint changes of κ_1 and κ_2, where κ_1 is the rate at which cancer cells are destroyed in an interaction with an immune cell, and κ_2 is the rate at which immune cells are recruited after such an interaction. This type of analysis directly portrays how a change in one or two parameters impacts the outcome of the system. As might be expected, the number of cancer cells $x(t)$ strongly depends on κ_1 but weakly on κ_2. In contrast, the number of immune cells $y(t)$ depends comparably on both parameters, perhaps due to the fact that the initial population of

immune cells is much smaller than the initial population of cancer cells.

There are limitations to this type of sensitivity analysis. How many solution curves should be examined? What time is most informative in displaying system changes? Is it really necessary to compute sensitivity over such a large range of parameters when the trends are so clear? These ambiguities cloud our understanding and require far more computing than is necessary. Differential sensitivity successfully addresses these concerns. Gradients of solutions immediately yield approximate solutions in a neighborhood of postulated parameter values. The relative importance of different parameters in determining species levels is obvious from inspection of the gradient. Furthermore, modern software easily delivers the gradient along entire solution trajectories. There is no need to solve for an entire bundle of neighboring solutions.

Differential assessment is far more efficient. The required calculations involve solving an expanded system of ordinary differential equations just once under the partial derivative method or solving the system once for each parameter under the analytic method. Either way, the differential method is much less computationally intensive than the traditional method of solving the ODE system over an interval for each parameter or over a rectangle for each pair of parameters. Here is our brief Julia code for computing sensitivity via the analytic method.

```
using DifferentialEquations, Plots

function sensitivity(x0, p, d, tspan)
  problem = ODEProblem{true}(ODE, x0, tspan, p)
  sol = solve(problem, saveat = 1.0) # solve ODE
  (lp, ls, lx) = (length(p), length(sol), length(x0))
```

```
  solution = Dict{Int, Any}(i => zeros(ls, lp + 1) for i in 1:lx)
  for j = 1:lx # record solution for each species
    @views solution[j][:, 1] = sol[j, :]
  end
  for j = 1:lp
    p[j] = p[j] + d * im # perturb parameter
    problem = ODEProblem{true}(ODE, x0, tspan, p)
    sol = solve(problem, saveat = 1.0) # resolve ODE
    p[j] = complex(real(p[j]), 0.0) # reset parameter
    @views sol .= imag(sol) / d # compute partial
    for k = 1:lx # record partial for each species
      @views solution[k][:,j + 1] = sol[k, :]
    end
  end
  return solution
end

function ODE(dx, x, p, t) # CARRGO model
  dx[1] = p[4] * x[1] * (1 - x[1] / p[5]) - p[1] * x[1] * x[2]
  dx[2] = p[2]* x[1] * x[2] - p[3] * x[2]
end

p = complex([6.0e-9, 3.0e-11, 1.0e-6, 6.0e-2, 1.0e9]); # parameters
x0 = complex([1.25e4, 6.25e2]); # initial values
(d, tspan) = (1.0e-13, (0.0,1000.0)); # step size and time interval in days
solution = sensitivity(x0, p, d, tspan); # find solution and partials
CARRGO1 = plot(solution[1][:, 1], label = "x1", xlabel= "days",
ylabel = "cancer cells x1", xlims = (tspan[1],tspan[2]))
CARRGO2 = plot(solution[1][:, 2], label = "d1x1", xlabel= "days",
ylabel = "p1 sensitivity", xlims = (tspan[1],tspan[2]))
```

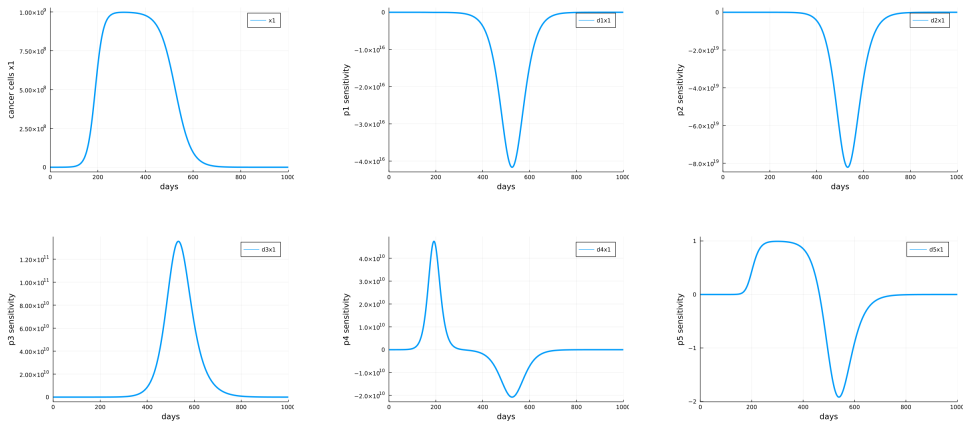

Figure 2: Sensitivity of Cancer Cells in the CARRGO Model

In the Julia code the parameters κ_1, κ_2, θ, ρ, and γ and the variables x and y exist as components of the vector $\boldsymbol{p}$ and $\boldsymbol{x}$, respectively. The two plot commands construct solution curves for cancer and its sensitivity to perturbations of κ_1.

Figure 2 reinforces the conclusions drawn from the heatmaps, but more clearly and quantitatively. In brief, the scaled sensitivity of cancer cells $x(t)$ is much more dependent on carrying capacity γ than on birth rate ρ, and even more so than on the destruction rate κ_1. Interestingly, the number of immune cells $y(t)$ is much more sensitive to the recruitment rate κ_2 than to the natural death rate θ.

3.2 Deterministic SIR Model

The deterministic SIR model follows the number of infectives $I(t)$, the number of susceptibles $S(t)$, and the number of recovereds $R(t)$ during the course

of an epidemic. These three subpopulations satisfy the ODE system

$$\begin{aligned} \frac{d}{dt}S &= -\eta I \frac{S}{N} \\ \frac{d}{dt}I &= \eta I \frac{S}{N} - \delta I \\ \frac{d}{dt}R &= \delta I, \end{aligned}$$

where η is the daily infection rate per encounter and δ is the daily rate of progression to immunity per person.

For SARS-CoV-2, current estimates [16] of η range from 0.0012 to 0.48, and estimates of δ range from 0.0417 to 0.0588 [17]. As an alternative to solving the extended set of differential equations, we again use the analytic method to evaluate parameter sensitivities.

The following Julia code for the analytic method incorporates the sensitivity function of the previous problem.

```
function ODE(dx, x, p, t) # Covid model
  N = 3.4e8 # US population size
  dx[1] = - p[1] * x[2] * x[1] / N
  dx[2] = p[1] * x[2] * x[1] / N - p[2] * x[2]
  dx[3] = p[2] * x[2]
end

p = complex([0.2, (0.0417 + 0.0588) / 2]); # parameters
x0 = complex([3.4e8, 100.0, 0.0]); # initial values
(d, tspan) = (1.0e-10, (0.0, 365.0)) # 365 days
solution = sensitivity(x0, p, d, tspan);
Covid = plot(solution[1][:, :], label = ["x1" "d1x1" "d2x1"],
        xlabel = "days", xlims = (tspan[1],tspan[2]))
```

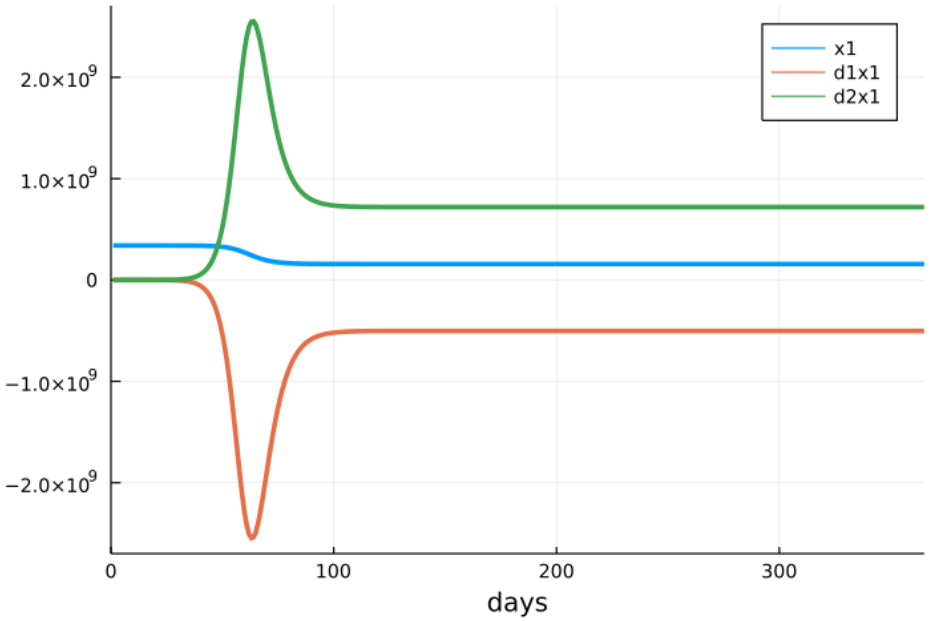

Figure 3: Sensitivities of Susceptibles in the Covid Model

Our parameter choices roughly capture the American experience. Figure 3 plots the susceptible curve and its sensitivities. In this case all three curves conveniently occur on comparable scales.

3.3 Second-Order Expansions of Solution Trajectories

In predicting nearby solution trajectories, the second-order Taylor expansion

$$f(\boldsymbol{x}+\boldsymbol{v}) \;\approx\; f(\boldsymbol{x})+df(\boldsymbol{x})\boldsymbol{v}+\frac{1}{2}\boldsymbol{v}^t d^2 f(\boldsymbol{x})\boldsymbol{v}$$

obviously improves accuracy over the first-order expansion

$$f(\boldsymbol{x}+\boldsymbol{v}) \;\approx\; f(\boldsymbol{x})+df(\boldsymbol{x})\boldsymbol{v}.$$

Figure 4 displays predicted trajectories for the SIR model when all parameters p_i are replaced by $p_i(1 + U_i)$, where each U_i is chosen uniformly from $(-0.25, 0.25)$. The figure vividly confirms the improvement in accuracy in passing from a first-order to a second-order approximation. The more nonlinear the solution trajectory becomes, the more improvement is evident.

For example, the middle panel of Figure 4 shows that the solution trajectory of infected individuals bends dramatically with a change in parameters. This behavior is much better reflected in the second-order prediction compared to the first-order prediction. The Euclidean distance between the actual and predicted trajectories at the sampled time points is about 30.0 in the first-order case and only about 9.82 in the second-order case, a reduction of over 65% in prediction error. By contrast, the trajectory of the recovered individuals steadily increases in a nearly linear fashion. The bottom panel of Figure 4 shows that the first-order prediction now remains reasonably accurate over a substantial period of time. Even so, the discrepancy between the predicted solutions grows so that by day 100 the absolute difference between the first-order prediction and the actual trajectory exceeds 140, compared to about 39.0 for the second-order prediction. Thus, calculating second-order sensitivity is helpful in both highly non-linear systems and systems with long time scales.

3.4 Stochastic SIR Model

We now illustrate sensitivity calculations in the stochastic SIR model. This model postulates an original population of size n with i infectives and s susceptibles. The parameters δ and η again capture the rate of progression to immunity and the infection rate per encounter. Since extinction of the infectives is certain, it makes sense to focus on the time to elimination of the

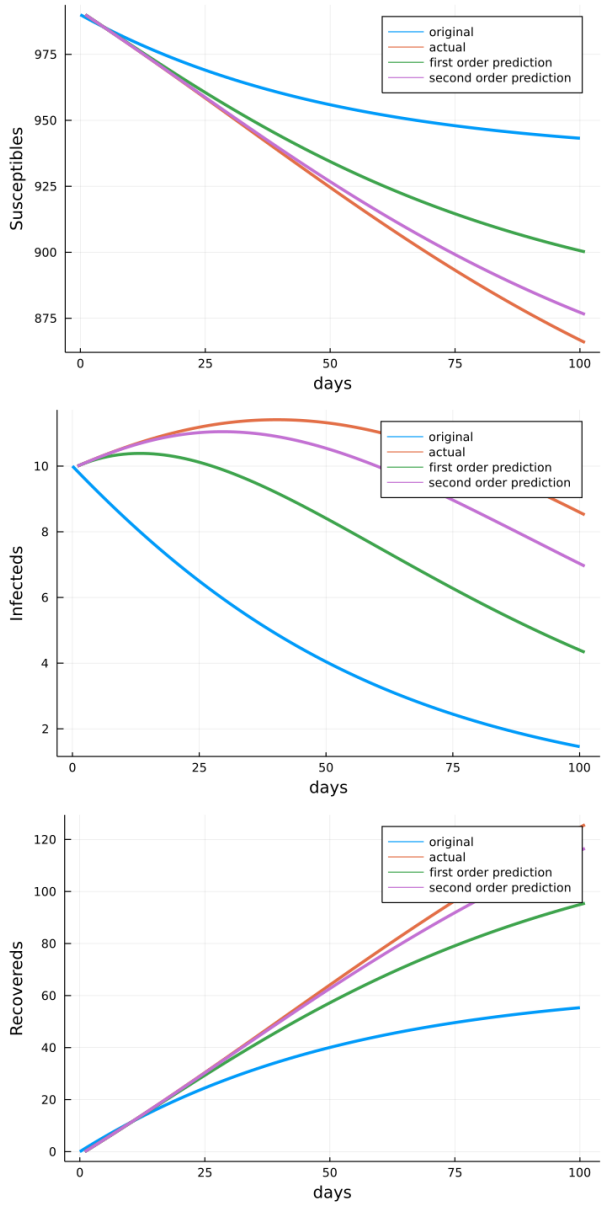

Figure 4: Model Trajectories for SIR Model Calculated Using First and Second Differentials

infectives. It is also convenient to follow the vector (i, n), where $n = i + s$ is the sum of the number of infectives i plus the number of susceptibles s. Elementary arguments now show that the mean time t_{in} to elimination of all infectives satisfies the recurrence

$$\begin{aligned} t_{in} &= \frac{1}{i\delta + i\left(\frac{n-i}{N}\right)\eta} + \frac{i\delta}{i\delta + i\left(\frac{n-i}{N}\right)\eta} t_{i-1,n-1} \\ &\quad + \frac{i\left(\frac{n-i}{N}\right)\eta}{i\delta + i\left(\frac{n-i}{N}\right)\eta} t_{i+1,n} \end{aligned} \tag{5}$$

for $0 < i < n$ together with the boundary conditions

$$t_{ii} = \sum_{j=1}^{i} \frac{1}{j\delta} \quad \text{and} \quad t_{0n} = 0.$$

The expression for t_{ii} stems from adding the expected time for the $i \to i-1$ transition, plus the expected time $i-1 \to i-2$, and so forth. This system of equations can be solved recursively for $i = n, n-1, \ldots 0$ starting with $n = 1$. Once the values for a given n are available, n can be incremented, and a new round is initiated. Ultimately the target size $n = N$ is reached. Taking partial derivatives of the recurrence (5) yields a new system of recurrences that can also be solved recursively in tandem with the original recurrence. The analytic method is easier to implement and comparable in accuracy to the partial derivative method.

Another important index of the SIR process is the mean number of infectives m_{in} ever generated starting with i initial infectives and n total

people. These expectations can be calculated via the recurrences

$$m_{in} = \frac{i\delta}{i\delta + i\left(\frac{n-i}{N}\right)\eta}(m_{i-1,n-1} + 1) + \frac{i\left(\frac{n-i}{N}\right)\eta}{i\delta + i\left(\frac{n-i}{N}\right)\eta}m_{i+1,n} \quad (6)$$

for $0 < i < n$ together with the boundary conditions

$$m_{ii} = i \quad \text{and} \quad m_{0n} = 0.$$

Obviously, one can compute the sensitivities of the m_{in} to parameter perturbations in almost exactly the same way as the t_{in}. Here is the Julia code for the two means and their sensitivities via the analytic method. Note how our earlier differential function plays a key role.

```
function SIRMeans(p)
  (delta, eta) = (p[1], p[2])
  M = Matrix(zeros(eltype(p), N + 1, N + 1)) # mean matrix
  T = similar(M) # time to extinction matrix
  for n = 1:N # recurrence relations loop
    for j = 0:(n - 1)
      i = n - j
      a = i * delta # immunity rate
      if i == n # initial conditions
        M[i + 1, n + 1] = i
        T[i + 1, n + 1] = T[i, i] + 1 / a
      else
        b = i * (n - i) * eta / N  # infection rate
        c = 1 / (a + b)
        M[i + 1, n + 1] = a * c * (M[i, n] + 1) + b * c * M[i + 2, n + 1]
        T[i + 1, n + 1] = c * (1 + a * T[i, n] + b * T[i + 2, n + 1])
```

```
      end
    end
  end
  return [M[:, N + 1]; T[:, N + 1]]
end

p = complex([0.2, (0.0417 + 0.0588) / 2]); # delta and beta
(N, d) = (100, 1.0e-10);
@time (f, df) = differential(SIRMeanTime, p, d);
```

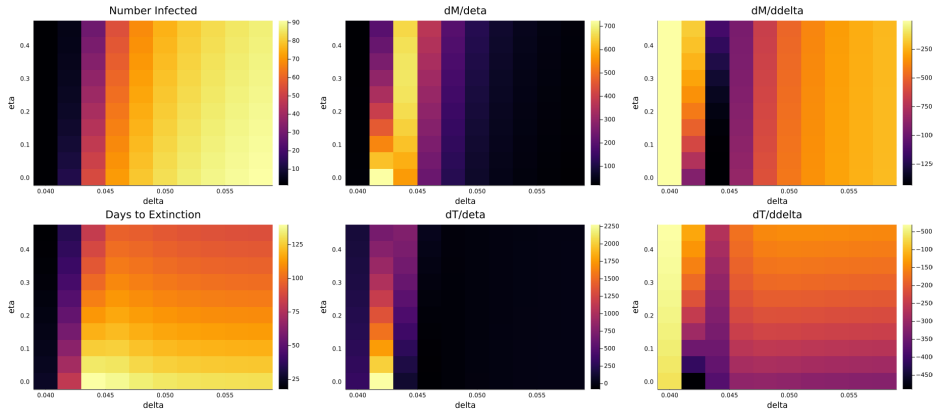

Figure 5: Sensitivity of Stochastic SIR Model

The left column of Figure 5 displays the expected total number of individuals infected (top) and the expected number of days to extinction (bottom) for the stochastic SIR model based on the parameters mentioned in Section 3.2. The middle column of Figure 5 depicts the derivatives of these quantities with respect to η (the infection rate), and the right column depicts the derivatives with respect to δ (the recovery rate).

It is interesting to compare our method's results to estimates from stochastic simulations. To see the difference in accuracy, we calculated the average number of individuals infected and the average time to extinction by

Table 1: SIR Model Outcomes

	Calculated Mean	Simulated Mean	Simulated Standard Error
Time to Extinction	2.792×10 days	3.074×10 days	4.153 days
Number Infected	5.484×10^3 people	5.838×10^3 people	8.551×10^2 people

Table 1 shows the comparison between the calculated and simulated means of SIR model outcomes with $N = 3.4 \times 10^4$ and $I_0 = 1$.

stochastic simulation using the software package Biosimulator.jl [18]. Table 1 records the analytic and simulated means of these outcomes under the parameter values pertinent to Figure 3. As Table 1 indicates, the simulated means over $r = 100$ runs are roughly comparable to the analytic means, but the standard errors of the simulated means are large. Because the standard errors decrease as $\frac{1}{\sqrt{r}}$, it is difficult to achieve much accuracy by simulation alone. In more complicated models, simulation is so computationally intensive and time consuming that it is nearly impossible to achieve accurate results. Of course, the analytic method is predicated on the existence of an exact solution or an algorithm for computing the same.

Parameter sensitivities inform our judgment in interesting and helpful ways. For example, derivatives of both the total number of infecteds and the time to extinction with respect to η are very small except in a narrow window of the δ parameter. This suggests that we focus further simulations, sensitivity analysis, and possible interventions on the region of parameter space where δ is small and derivatives with respect to η tend to be large. Derivatives with respect to δ depend mostly on η except at very small values of δ. These conclusions are harder to draw from noisy simulations alone.

3.5 Branching Processes

Branching process models offer an opportunity for checking the accuracy of sensitivity calculations. For simplicity we focus on birth-death-migration processes [19]. These are multi-type continuous-time processes [9, 12] that can be used to model the early stages of an epidemic over a finite graph with n nodes, where nodes represent cities or countries. On node i we initiate a branching process with birth rate $\beta_i > 0$ and death rate $\delta_i > 0$. The migration rate from node i to node j is $\lambda_{ij} \geq 0$. All rates are per person, and each person is labeled by a node. Let $\lambda_i = \sum_{j \neq i} \lambda_{ij}$ be the sum of the migration rates emanating from node i. Given this notation, the mean infinitesimal generator of the process is the matrix

$$\mathbf{\Omega} = \begin{pmatrix} \beta_1 - \delta_1 - \lambda_1 & \lambda_{12} & \cdots & \lambda_{1,n-1} & \lambda_{1n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \lambda_{n1} & \lambda_{n2} & \cdots & \lambda_{n,n-1} & \beta_n - \delta_n - \lambda_n \end{pmatrix}$$

The entries of the matrix $e^{t\mathbf{\Omega}} = [m_{ij}(t)]$ represent the expected number of people at node j at time t starting from a single person of type i at time 0. The process is irreducible when the pure migration process corresponding to choice $\beta_i = \delta_i = 0$ for all i is irreducible. Henceforth, we assume the process is irreducible and let $\mathbf{\Gamma}$ denote the mean infinitesimal generator of the pure migration process. The process is subcritical, critical, or supercritical depending on whether the dominant eigenvalue ρ of $\mathbf{\Omega}$ is negative, zero, or positive.

To determine the local sensitivity of ρ to a parameter θ [9, 20], suppose its left and right eigenvectors $\boldsymbol{v}$ and $\boldsymbol{w}$ are normalized so that $\boldsymbol{v}\boldsymbol{w} = 1$.

Differentiating the identity $\boldsymbol{\Omega}\boldsymbol{w} = \rho\boldsymbol{w}$ with respect to θ yields

$$\left(\frac{\partial}{\partial\theta}\boldsymbol{\Omega}\right)\boldsymbol{w} + \boldsymbol{\Omega}\frac{\partial}{\partial\theta}\boldsymbol{w} \;=\; \left(\frac{\partial}{\partial\theta}\rho\right)\boldsymbol{w} + \rho\frac{\partial}{\partial\theta}\boldsymbol{w}.$$

If we multiply this by $\boldsymbol{v}$ on the left and invoke the identities $\boldsymbol{v}\boldsymbol{\Omega} = \rho\boldsymbol{v}$ and $\boldsymbol{v}\boldsymbol{w} = 1$ we find that

$$\frac{\partial}{\partial\theta}\rho \;=\; \boldsymbol{v}\left(\frac{\partial}{\partial\theta}\boldsymbol{\Omega}\right)\boldsymbol{w}. \tag{7}$$

Because $\frac{\partial}{\partial\delta_i}\boldsymbol{\Omega} = -\frac{\partial}{\partial\beta_i}\boldsymbol{\Omega}$, it follows that an increase in δ_i has the same impact on ρ as the same decrease in β_i. The sensitivity of $\boldsymbol{v}$ and $\boldsymbol{w}$ can be determined by an extension of this reasoning [22]. The extinction probabilities e_i of the birth-death-migration satisfy the system of algebraic equations

$$e_i \;=\; \frac{\delta_i}{\beta_i + \delta_i + \lambda_i} + \frac{\beta_i}{\beta_i + \delta_i + \lambda_i}e_i^2 + \sum_{j\neq i}\frac{\lambda_{ij}}{\beta_i + \delta_i + \lambda_i}e_j \tag{8}$$

for all i. This is a special case of the vector extinction equation

$$\boldsymbol{e} \;=\; P(\boldsymbol{e}) \;=\; \begin{pmatrix} P_1(\boldsymbol{e}) \\ \vdots \\ P_n(\boldsymbol{e}) \end{pmatrix}$$

for a general branching process with offspring generating function $P_i(\boldsymbol{x})$ for a type i person [21]. For a subcritical or critical process, $\boldsymbol{e} = \boldsymbol{1}$. For a supercritical process all $e_i \in (0, 1)$. Iteration is the simplest way to find $\boldsymbol{e}$. Starting from $\boldsymbol{e}_0 = \boldsymbol{0}$, the vector sequence $\boldsymbol{e}_n = P(\boldsymbol{e}_{n-1})$ satisfies

$$\boldsymbol{0} \;\leq\; \boldsymbol{e}_n \;\leq\; \boldsymbol{e}_{n+1} \;\leq\; \boldsymbol{e}$$

and converges to a solution of the extinction equations. Here all inequalities apply component-wise.

To find the differential [22] of the extinction vector $\boldsymbol{e}$ with respect to a vector $\boldsymbol{\theta}$ of parameters, we assume that the branching process is supercritical and resort to implicit differentiation of the equation $\boldsymbol{e}(\boldsymbol{\theta}) = P[\boldsymbol{e}(\boldsymbol{\theta}), \boldsymbol{\theta}]$. The chain rule gives

$$d_{\boldsymbol{\theta}}\boldsymbol{e} \;=\; d_{\boldsymbol{e}}P(\boldsymbol{e},\boldsymbol{\theta})d_{\boldsymbol{\theta}}\boldsymbol{e} + d_{\boldsymbol{\theta}}P(\boldsymbol{e},\boldsymbol{\theta}).$$

This equation has the obvious solution

$$d_{\boldsymbol{\theta}}\boldsymbol{e} \;=\; [\boldsymbol{I}_n - d_{\boldsymbol{e}}P(\boldsymbol{e},\boldsymbol{\theta})]^{-1}\, d_{\boldsymbol{\theta}}P(\boldsymbol{e},\boldsymbol{\theta}). \tag{9}$$

The indicated inverse does, in fact, exist, but the proof is a detour. Alternatively, one can compute an entire extinction curve $\boldsymbol{e}(t)$ whose component $e_i(t)$ supplies the probability of extinction before time t starting from a single person of type i [12]. This task reduces to solving a ODE $\frac{d}{dt}\boldsymbol{e}(t)$ by the methods previously discussed. We leave the details to interested readers.

The following Julia code computes the sensitivities of the extinction probability for a two-node process by the analytic method.

```
using LinearAlgebra

function extinction(p)
  types = Int(sqrt(1 + length(p)) - 1) # length(p) = 2 * types + types^2
  (x, y) = (zeros(Complex, types), zeros(Complex, types))
  for i = 1:500 # functional iteration
    y = P(x, p)
    if norm(x - y) < 1.0e-16 break end
```

```
    x = copy(y)
  end
  return y
end

function P(x, p) # progeny generating function
  types = Int(sqrt(1 + length(p)) - 1) # length(p) = 2 * types + types^2
  delta = p[1: types]
  beta = p[types + 1: 2 * types]
  lambda = reshape(p[2 * types + 1:end], (types, types))
  y = similar(x)
  t = delta[1] + beta[1] + lambda[1, 2]
  y[1] = (delta[1] + beta[1] * x[1]^2 + lambda[1, 2] * x[2]) / t
  t = delta[2] + beta[2] + lambda[2, 1]
  y[2] = (delta[2] + beta[2] * x[2]^2 + lambda[2, 1] * x[1]) / t
  return y
end

delta = complex([1.0, 1.75]); # death rates
beta = complex([1.5, 1.5]); # birth rates
lambda = complex([0.0 0.5; 1.0 0.0]); # migration rates
p = [delta; beta; vec(lambda)]; # package parameter vector
(types, d) = (2, 1.0e-10)
@time (e, de) = differential(extinction, p, d)
```

To adapt the code to a different branching process model, one simply supplies the appropriate progeny generating function and necessary parameters.

The average number a_{ij} of infected individuals of type j ultimately generated by a single initial infected individual of type i is also of interest. The matrix $\boldsymbol{A} = (a_{ij})$ of these expectations can be calculated via the matrix

equation

$$\boldsymbol{A} = (\boldsymbol{I}_n - \boldsymbol{F})^{-1}, \tag{10}$$

where $\boldsymbol{F}$ is the offspring matrix

$$\boldsymbol{F} = \begin{pmatrix} \frac{2\beta_1}{\beta_1+\delta_1+\lambda_1} & \frac{\lambda_{12}}{\beta_1+\delta_1+\lambda_1} & \cdots & \frac{\lambda_{1,n-1}}{\beta_1+\delta_1+\lambda_1} & \frac{\lambda_{1n}}{\beta_1+\delta_1+\lambda_1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\lambda_{n1}}{\beta_n+\delta_n+\lambda_n} & \frac{\lambda_{n2}}{\beta_n+\delta_n+\lambda_n} & \cdots & \frac{\lambda_{n,n-1}}{\beta_n+\delta_n+\lambda_n} & \frac{2\beta_n}{\beta_n+\delta_n+\lambda_n} \end{pmatrix}.$$

One can determine the local sensitivity of the expected numbers of total descendants by differentiating the equation $\boldsymbol{A} = (\boldsymbol{I}_n - \boldsymbol{F})^{-1}$. The result

$$d_{\boldsymbol{\theta}}\boldsymbol{A} = (\boldsymbol{I}_n - \boldsymbol{F})^{-1} d_{\boldsymbol{\theta}}\boldsymbol{F} (\boldsymbol{I} - \boldsymbol{F})^{-1}, \tag{11}$$

depends on the sensitivity of the expected offspring matrix $\boldsymbol{F}$. Julia code for the analytic method with two nodes follows.

```
function particles(p) # mean infected individuals generated
  types = Int(sqrt(1 + length(p)) - 1) # length(p) = 2 * types + types^2
  delta = p[1: types]
  beta = p[types + 1: 2 * types]
  lambda = reshape(p[2 * types + 1:end], (types, types))
  F = complex(zeros(types, types))
  t = delta[1] + beta[1] + lambda[1, 2]
  (F[1, 1], F[1, 2]) = (2 * beta[1] / t, lambda[1, 2] / t)
  t = delta[2] + beta[2] + lambda[2, 1]
  (F[2, 1], F[2, 2]) = (lambda[2, 1] / t, 2 * beta[2] / t)
  A = vec(inv(I - F)) # return as vector
end
```

```
delta = complex([1.0, 1.75]); # death rates
beta = complex([1.5, 1.5]); # birth rates
lambda = complex([0.0 0.5; 1.0 0.0]); # migration rates
p = [delta; beta; vec(lambda)]; # package parameter vector
(types, d) = (2, 1.0e-10)
@time (A, dA) = differential(particles, p, d)
```

4 Accuracy and Speed

We now assess the speed and accuracy of the computational schemes just described. In general, we measure accuracy by the Euclidean norms

$$\begin{aligned} \text{err}_1 &= \|f(\boldsymbol{x}+\boldsymbol{v}) - f(\boldsymbol{x}) - df(\boldsymbol{x})\boldsymbol{v}\| \\ \text{err}_2 &= \left\|f(\boldsymbol{x}+\boldsymbol{v}) - f(\boldsymbol{x}) - df(\boldsymbol{x})\boldsymbol{v} - \frac{1}{2}\boldsymbol{v}^t d^2 f(\boldsymbol{x})\boldsymbol{v}\right\|. \end{aligned}$$

Other norms, such as the ℓ_1 and ℓ_∞ norms, yield similar results. In the ODE models, $f(\boldsymbol{x})$ denotes a matrix trajectory so the Frobenius norm applies. This is scaled by $\frac{1}{n}$, where n is the number of sampled time points. We also compare the analytic derivatives to the automatic derivatives delivered by DiffEqSensitivity.jl (abbreviated DES.jl) via normed differences. Finally, our speed comparisons offer a first look at the efficiency gains possible with multithreading. All computations were done in Julia version 1.6.2 on a Windows operating system with an Intel(R) Core (TM) i7-8565U CPU.

Table 2 records benchmarking results for three differential equations models. The top table shows the deterministic SIR model. This model, previously introduced in Section 3.2, serves as a straightforward benchmark for comparing the various differential sensitivity methods summarized earlier. For first-order approximations, we compare the analytic method with the au-

tomatic differentiation method available in DES.jl [3]. The table shows that the analytic method offers comparable accuracy but slower compute times than automatic differentiation. It is noteworthy that more naive implementations of forward mode differentiation do not take advantage of internal parallelization (chunking) and are consequently much slower than the analytic method. Results for forward mode differentiation with and without internal parallelization as implemented in ForwardDiff.jl [14] can be found in Appendix C.

As we expect, second-order approximations are more accurate in prediction. For second-order methods, we compare the double Jacobian method found in ForwardDiff.jl with the second-order analytic method. In contrast to the analytic method, forward differentiation slows dramatically in calculating the Hessian directly, and only speeds up if we instead calculate the gradient of the gradient. The ForwardDiff.jl package currently does not offer this method natively, so the analytic method provides a simple way to improve the speed of computing second derivatives. While both of these methods allow for multi-threading, this model is too low dimensional to show much of an improvement.

The middle subtable in Table 2 records benchmarking results for the CARRGO model previously described in Section 3.1. This model serves as an example of a stiff system of differential equations. We can immediately see the impact of stiffness in the loss of accuracy at time points 100 and 1000. Stiffness highlights the added value of the second-order approximations. With stiff ODEs, high quality integrators are a must. In the analytic method, one must also choose the perturbation interval d carefully in computing second derivatives. Careful calibration of d improves the accuracy of the CARRGO second derivative analytic prediction to the level of automatic

differentiation.

To explore whether parallelization improves computational speed in high-dimensional models, we now turn a third model summarized in the bottom subtable of Table 2. This ODE model of the mammalian cell cycle (MCC) was constructed by Gerard and Goldbetor [23] and is explained in Appendix B. The model comprises 11 equations and 15 parameters and captures aspects of cell reproduction and cycling mediated by chemical signaling from cell-state dependent proteins such as through tumor repressors, transcription factors, and other DNA replication checkpoints. The model relies on cell state as opposed to cell mass and nicely replicates sequential progression along the cell cycle. The model clearly demonstrates the increase in speed from parallelization of both the analytic method and forward differentiation.

We now turn to the stochastic SIR model and compare our two suggested methods for computing the derivatives of the mean number of infected individuals (M) and the mean time to extinction (T). Both methods depend on the recurrences (5) and (6). The direct differentiation method relies on differentiating these recurrences, while the analytic method simply perturbs parameter values and recomputes solutions. Table 3 records the speed and accuracy of the two methods. As expected, computation speed varies quadratically with the number of individuals in the system N. The analytic method proves to be twice as fast as the direct method.

The branching process model for the spread of COVID-19 is similar to the SIR model. Table 4 compares the direct method of calculating $\frac{d\boldsymbol{A}}{d\beta}$ based on equation (11) to the analytic method based on equation (10). It also compares the direct method of calculating $\frac{de}{d\beta}$ based on differentiating the recurrence (8) to the analytic method based on the same recurrence. In this case, the analytic method proves to be just as accurate as the direct

Table 2: Time and Accuracy per Time Point for ODE System Derivatives over the Interval $[0, t_{end}]$

SIR Model

	Time (μs)			Accuracy		
	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 100$	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 1000$
Forward DES.jl ∇F	8.02×10^1	2.30×10^2	1.77×10^3	1.19	2.75×10^2	9.86×10^3
Analytic ∇F	2.44×10^2	9.94×10^2	8.82×10^3	1.19	2.75×10^2	9.86×10^3
Multi-Threaded Analytic ∇F	2.49×10^2	1.03×10^3	1.22×10^4	1.19	2.75×10^2	9.86×10^3
Parallel FD.jl $\nabla(\nabla F)$	2.07×10^2	4.28×10^2	3.92×10^3	1.15×10^{-1}	7.24×10^1	6.78×10^3
Multi-Threaded Parallel FD.jl $\nabla(\nabla F)$	1.71×10^2	4.29×10^2	3.95×10^3	1.15×10^{-1}	7.24×10^1	6.78×10^3
Analytic D^2F	8.86×10^2	3.76×10^3	5.20×10^4	1.15×10^{-1}	7.24×10^1	6.78×10^3
Multi-Threaded Analytic D^2F	8.77×10^2	3.26×10^3	5.57×10^4	1.15×10^{-1}	7.24×10^1	6.78×10^3

CARRGO Model

	Time (μs)			Accuracy		
	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 100$	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 1000$
Forward DES.jl ∇F	1.21×10^2	3.69×10^2	2.30×10^3	6.19	2.80×10^5	1.46×10^8
Analytic ∇F	4.41×10^2	2.14×10^3	1.76×10^4	6.19	2.80×10^5	1.46×10^8
Multi-Threaded Analytic ∇F	4.43×10^2	2.17×10^3	2.58×10^4	6.19	2.80×10^5	1.46×10^8
Parallel FD.jl $\nabla(\nabla F)$	1.23×10^3	4.48×10^3	4.38×10^4	1.13×10^{-1}	5.01×10^4	4.23×10^7
Multi-Threaded Parallel FD.jl $\nabla(\nabla F)$	1.20×10^3	4.60×10^3	4.66×10^4	1.13×10^{-1}	5.01×10^4	4.23×10^7
Analytic D^2F	2.38×10^3	1.43×10^4	1.22×10^5	1.12×10^{-1}	4.96×10^4	7.93×10^7
Multi-Thread Analytic D^2F	2.00×10^3	1.14×10^4	1.25×10^5	1.12×10^{-1}	4.96×10^4	7.93×10^7

MCC Model

	Time (μs)			Accuracy		
	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 100$	$t_{end} = 10$	$t_{end} = 100$	$t_{end} = 1000$
Forward DES.jl ∇F	6.12×10^2	9.35×10^3	2.51×10^4	3.47×10^{-3}	7.56×10^{-3}	1.54×10^{-1}
Analytic ∇F	2.71×10^3	1.51×10^4	1.08×10^5	3.47×10^{-3}	7.56×10^{-3}	1.54×10^{-1}
Multi-Threaded Analytic ∇F	2.56×10^3	1.51×10^4	1.08×10^5	3.47×10^{-3}	7.56×10^{-3}	1.54×10^{-1}
Parallel FD.jl $\nabla(\nabla F)$	1.84×10^4	7.44×10^4	2.04×10^5	1.26×10^{-3}	2.07×10^{-3}	4.21×10^{-2}
Multi-Threaded FD.jl $\nabla(\nabla F)$	1.76×10^4	7.77×10^4	2.04×10^5	1.26×10^{-3}	2.07×10^{-3}	4.21×10^{-2}
Analytic D^2F	4.78×10^4	2.81×10^5	1.00×10^6	1.27×10^{-3}	1.92×10^{-3}	3.89×10^{-2}
Multi-Threaded Analytic D^2F	2.56×10^4	1.74×10^5	1.36×10^6	1.27×10^{-3}	1.92×10^{-3}	3.89×10^{-2}

Table 3: Time and Accuracy in the Stochastic SIR Model

	Time (μs)			Accuracy		
	$N = 10$	$N = 100$	$N = 100$	$N = 10$	$N = 100$	$N = 1000$
$\frac{d\boldsymbol{M}}{d\eta}$ direct	3.21×10^1	3.39×10^3	4.33×10^5	3.47×10^{-3}	2.95×10^{-1}	3.05×10^{-1}
$\frac{d\boldsymbol{M}}{d\eta}$ analytic	1.72×10^1	1.51×10^3	1.70×10^5	3.47×10^{-3}	2.95×10^{-1}	3.05×10^{-1}
$\frac{d\boldsymbol{T}}{d\eta}$ direct	3.78×10^1	3.71×10^3	4.48×10^5	2.04×10^{-2}	4.86×10^{-1}	4.97×10^{-1}
$\frac{d\boldsymbol{T}}{d\eta}$ analytic	1.65×10^1	1.51×10^3	2.45×10^5	2.04×10^{-2}	4.86×10^{-1}	4.97×10^{-1}

Table 4: Time and Accuracy in the Branching Process Model

	Time (μs)			Accuracy		
	$N = 10$	$N = 100$	$N = 100$	$N = 10$	$N = 100$	$N = 1000$
$\frac{d\boldsymbol{A}}{d\beta}$ direct	4.28	2.46×10^3	1.09×10^5	1.02×10^{-2}	7.41×10^{-4}	1.43×10^{-5}
$\frac{d\boldsymbol{A}}{d\beta}$ analytic	1.46×10^2	4.28×10^3	1.68×10^5	1.02×10^{-2}	7.41×10^{-4}	1.43×10^{-5}
$\frac{d\boldsymbol{e}}{d\beta}$ direct	1.29×10^3	8.29×10^4	7.75×10^6	9.23×10^{-5}	9.25×10^{-7}	1.85×10^{-15}
$\frac{d\boldsymbol{e}}{d\beta}$ analytic	9.69×10^2	5.64×10^4	1.20×10^7	9.23×10^{-5}	9.25×10^{-7}	1.85×10^{-15}

methods, albeit slower in some cases. Other evidence not shown suggests that the analytic method can reliably evaluate sensitivities where solutions depend on linear algebra and/or recurrence relations. Unless derivatives are quite complicated, direct methods will generally be faster. In computing second derivatives, we expect the tables will be turned.

5 Discussion

Our purpose throughout has been to demonstrate the ease and utility of incorporating differential sensitivity analysis in dynamical modeling. Because models are always approximate, and parameters are measured imprecisely, uncertainty plagues all of dynamical modeling. Improving models is incremental and domain specific. Sensitivity analysis provides a handle on local parameter uncertainty. Assessing global parameter sensitivity is more challenging. It can be attacked by techniques such as Latin square hypercube sampling or Sobel quasi-random sampling, but these become infeasible in high dimensions [24]. Given the availability of appropriate software, differential sensitivity is computationally feasible, even for high dimensional systems.

In the case of stochastic models, traditional methods require costly and inaccurate simulation over a bundle of parameter values. Differential sensitivity is often out of the question. This limits the ability of researchers to

understand a biological system and how it responds to parameter changes. If a system index such as a mean, variance, extinction probability, or extinction time can be computed by a reasonable algorithm, then differential parameter sensitivity analysis can be undertaken. We have indicated in a handful of examples how this can be accomplished.

In summary, across many models representative of computational biology, we have found the following results to hold:

(a) Chunked forward mode automatic differentiation and forward mode sensitivity analysis tend to be the most efficient on the tested models.

(b) The analytic methods described in this manuscript are competitive and often outperform the unchunked version of forward mode automatic differentiation.

(c) Shared memory multithreading of the analytic and forward mode automatic differentiation methods provides a performance gain in high-dimensional systems.

(d) Adjoint methods tend to have better scalability to very large systems with more than 100 ODEs [25]. However, on our test problems the adjoint-based methods were not competitive.

(e) Forward mode automatic differentiation method requires that each step of a calculation is differentiatiable. This renders it unusable for calculating the derivative of ensemble means of discrete state models, such as birth-death processes. For these cases, the analytic method outperforms direct numerical differentiation.

These conclusions are tentative, but supported by our limited number of biological case studies.

We note that the performance differences may change depending on the efficiency of the implementations. The Julia DifferentialEquations.jl library and its DiffEqSensitivity.jl package have been shown to be highly efficient, outperforming other libraries in both equation solving and derivative calculations in Python, MATLAB, C, and Fortran [25, 26] (Details on the current state of performance can be found at https://github.com/SciML/SciMLBenchmarks.jl.) The automatic differentiation implementations in machine learning libraries optimize array operations much more than scalar operations. This can work to the detriment of forward mode AD. Similarly, forward mode AD sensitivity analysis is more amendable to MATLAB or Python style vectorization that improves performance by reducing interpreter overhead. Therefore, our conclusions can serve as guidelines for the case where all implementations are well-optimized. However, when using programming languages with high overheads or without compile-time optimization of the automatic differentiation passes, the balance in efficiency shifts more favorably towards the analytic method.

One last point worth making is on the coding effort required by the various methods. Both automatic differentiation and the analytic method have comparable accuracy when applied to systems of ODEs, with automatic differentiation having the advantage in speed when it is implemented with an additional level of parallelization. However, the analytic method can easily be generalized to other kinds of objective functions and may be more straightforward to implement for those less sophisticated in computer science. While automatic differentiation is the basis of many large scientific packages, the code required for the analytic methods is fully contained within this manuscript and is easily transferable to other programming languages with similar dispatching on complex numbers. This hard to measure

benefit should not be ignored by practicing biologists who simply wish to quickly arrive at reasonably fast code.

Acknowledgments

We wish to thank Chris Elrod for assistance in adding multithreading to parts of our software. We wish to thank Janet Sinsheimer, Mary Sehl, and Xiang Ji for helpful comments on the manuscript and biological applications.

References

[1] Cao Y, Li S, Petzold L, Serban R. Adjoint sensitivity analysis for differential-algebraic equations: the adjoint DAE system and its numerical solution. SIAM Journal on Scientific Computing. 2003; 24:1076–108

[2] Strang G. Computational Science and Engineering. Wellesley-Cambridge Press. 2007.

[3] Rackauckas C, Nie Q. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. Journal of Open Research Software. 2017; 5

[4] Errico RM. What is an adjoint model? Bulletin of the American Meteorological Society. 1997; 78:2577–2592

[5] Granzow GD. A tutorial on adjoint methods and their use for data assimilation in glaciology. Journal of Glaciology. 2014; 60:440–446

[6] Stapor, Paul, et al. PESTO: parameter estimation toolbox. Bioinformatics. 2018; 34.4:705-707.

774 [7] Hindmarsh et al. SUNDIALS Suite of nonlinear and differen-
775 tial/algebraic equation solvers. ACM Transactions on Mathematical
776 Software (TOMS). 2020; 31(3):363–396

777 [8] Lange K. MM Optimization Algorithms. SIAM. 2016

778 [9] Dorman K, Sinsheimer JS, Lange K. In the garden of branching pro-
779 cesses. SIAM Review. 20014; 46:202–229

780 [10] Yakovenko S, Ilyashenko Y. Lectures on Analytic Differential Equa-
781 tions. American Mathematical Society. 2008.

782 [11] Henrici P. Fast Fourier transform methods in computational complex
783 analysis. SIAM Review. 1979; 21:481–527

784 [12] Lange, K . Applied Probability, 2nd ed. Springer. 2010.

785 [13] Lai K-L Crassidis J, Cheng,Y, Kim J. New complex-step derivative ap-
786 proximations with application to second-order Kalman filtering. AIAA
787 Guidance, Navigation, and Control Conference and Exhibit 2005. 2005;
788 :5944

789 [14] Revels, J et al. Forward-mode automatic differentiation in Julia. arXiv
790 preprint. 2016; 1607.07892.

791 [15] Sahoo P et al. Mathematical deconvolution of CAR T-cell proliferation
792 and exhaustion from real-time killing assay data. Journal Royal Society
793 Interface. 2020; 17: 20190734.

794 [16] Toda, AA. Susceptible-infected (SIR) dynamics of COVID-19 and eco-
795 nomic impact. arXiv preprint. 2020; 2003.11221

[17] Zhou F et al. Clinical course and risk factors for mortality of adult inpatients with COVID-19 in Wuhan, China: a retrospective cohort study. The Lancet. 2020; 395:1054–1062

[18] Landeros A, Stutz, T, Keys, KL, Alekseyenko, A, Sinsheimer, JS, Lange, K, & Sehl, ME. BioSimulator. jl: Stochastic simulation in Julia. Computer Methods and Programs in Biomedicine. 2018; 167, 23-35.

[19] Renshaw E. Birth, death and migration processes. Biometrika. 1972; 59:49–60

[20] Hautphenne S, Krings G, Delvenne J-C, Blondel VD. Sensitivity analysis of a branching process evolving on a network with application in epidemiology. Journal of Complex Networks. 2015; 3:606–641

[21] Athreya KB, Ney PE. Branching Processes. Springer. 1972.

[22] Magnus JR, Neudecker H. Matrix Differential Calculus with Applications in Statistics and Econometrics. Wiley, New York. 1988; pp. 30–31,158–165

[23] Gérard C, Goldbeter A. Temporal self-organization of the cyclin/Cdk network driving the mammalian cell cycle. Proceedings of the National Academy of Sciences of the United States of America. 2009; 106:51:21643-21648

[24] Qian G, Mahdi A. Sensitivity analysis methods in the biomedical sciences. Mathematical Biosciences. 2020; 323:108306

[25] Rackauckas, Christopher, et al. A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. arXiv preprint. 2018; arXiv:1812.01892

[26] Rackauckas, Christopher, et al. Universal differential equations for scientific machine learning. arXiv preprint. 2020; arXiv:2001.04385

Appendix A: Derivation of Second Derivative Analytic Method

To prove the formulas for approximating partial derivatives stated in the text, we first note that any analytic function $f(\boldsymbol{z})$ of several variables can be expanded in a locally convergent power series about every point $\boldsymbol{z}$ of its open domain of definition. If we choose a real direction vector $\boldsymbol{v}$, then the function $g(w) = f(\boldsymbol{z} + w\boldsymbol{v})$ is locally analytic in the complex plane $\{\boldsymbol{z} + w\boldsymbol{v} : w \in \mathbb{C}\}$ and can be expanded in a power series around $w = 0$. Thus,

$$g(w) = \sum_{j=0}^{d} \frac{d^j}{dw^j} g(\mathbf{0}) \frac{w^j}{j!} + O(|w|^{d+1}) \tag{12}$$

for any integer $d \geq 0$. Now consider the setting where $\boldsymbol{z}$ has real components. If $f(\boldsymbol{z})$ is real valued, then the derivatives $\frac{d^j}{dw^j} g(\mathbf{0})$ will be real as well. One can exploit this fact in approximating the derivatives. For example, if $w = i$, then w^j rotates among the four values 1, i, -1, and $-i$. Because the terms of the expansion (12) alternate between real and imaginary values, the first partial derivative formula (2) holds. For the choice $w = e^{\pi i/4}$, the powers w^d rotate among the eight values 1, $e^{\pi i/4}$, i, $ie^{\pi i/4}$, -1, $-e^{\pi i/4}$, $-i$, and $-ie^{\pi i/4}$. The powers $(-w)^j = (-1)^j w^j$ agree in this regard except for sign. Hence, the terms in the expansion of the sum

$$g[\boldsymbol{x} + e^{\pi i/4}\Delta(\boldsymbol{e}_j + \boldsymbol{e}_k)] + g[\boldsymbol{x} - e^{\pi i/4}\Delta(\boldsymbol{e}_j + \boldsymbol{e}_k)]$$

alternately cancel and reinforce. Thus, the first five terms of the expansion are real, 0, imaginary, 0, real, 0. It follows that the imaginary part of the sum is accurate to order $O(\Delta^6)$ and that the approximations (3) and (4) are accurate to order $O(\Delta^4)$.

Appendix B: Mammalian Cell Cycle Model

The Mammalian Cell Cycle Model presented in [23] reduces to a system of ordinary differential equations representing the interaction of cyclin-dependent kinases (Cdk) with Cdk inhibitors, growth factors, and other proteins that regulate the development of mammalian cells. The model includes characteristics such as cell cycling, tumor repressor initiated progression control,

and cell cycle completion. The ODE system representing the model is

$$
\begin{aligned}
\frac{dpRBc1}{dt} &= kpc1 * pRB * E2F \\
\frac{dpRBc2}{dt} &= kpc3 * pRBp * E2F \\
\frac{dCd}{dt} &= kcd1 * AP1 + kdecom1 * Mdi \\
&\quad - kcom1 * Cd * (Cdk4_{tot} - (Mdi + Md + Mdp27)) \\
&\quad + kcd2 * E2F * \frac{Ki7}{Ki7 + pRB} * \frac{Ki8}{Ki8 + pRBp} \\
\frac{dMdi}{dt} &= Vm2d * \frac{Md}{k2d + Md} + 2 * kcom1 * Cd * (Cdk4_{tot} - (Mdi + Md + Mdp27)) \\
\frac{dMd}{dt} &= Vm1d * \frac{Mdi}{k1d + Mdi} + kcom1 * Cd * (Cdk4_{tot} - (Mdi + Md + Mdp27)) \\
\frac{dpRB}{dt} &= kcd2 * E2F * \frac{Ki7}{Ki7 + pRB} * \frac{Ki8}{Ki8 + pRBp} \\
\frac{dE2F}{dt} &= kcd2 * E2F * \frac{Ki7}{Ki7 + pRB} * \frac{Ki8}{Ki8 + pRBp} \\
\frac{dpRBp}{dt} &= kcd2 * E2F * \frac{Ki7}{Ki7 + pRB} * \frac{Ki8}{Ki8 + pRBp} \\
\frac{dAP1}{dt} &= kcd1 * AP1 \\
\frac{dp27}{dt} &= 0 \\
\frac{dMdp27}{dt} &= kc1 * Md * p27 + kcom1 * cd * (Cdk4_{tot} - (Mdi + Md + Mdp27))
\end{aligned}
$$

The initial values of each compartment and parameter are defined as

Compartment	Initial Value (μ mol)	Parameter	Value
pRBc1	0.1	kpc1	0.05
pRBc2	0.05	kpc3	0.025
Cd	0.01	kcd1	0.4
Mdi	0.01	Ki8	2.0
Md	0.01	Ki7	0.1
pRB	0.0	kcd2	0.005
E2F	0.0	kdecom1	0.1
pRBp	0.0	k2d	0.1
AP1	0.0	$Cdk4_{tot}$	1.5
p27	0.0	kcom1	0.175
Mdp27	0.0	Vm2d	0.2
–	–	k1d	0.1
–	–	Vm1d	1.0
–	–	kc1	0.15

Appendix C: Supplementary Benchmarking Data

In addition to the methods for conducting the differential sensitivity analysis for systems of ODEs displayed in Table 2 (Section 4), we conducted benchmarking tests on some variations of these methods as well. Tables 5 and 6 summarize these benchmarks for the SIR Model. Additional first-order methods include using the ForwardDiff.jl [14] package to calculate the Jacobian with and without chunking and with and without multi-threading, and using a constant value for $\epsilon = 1 \times 10^{-12}$ instead of one scaled by the parameters. Additional second order methods include using DES.jl

Table 5: Time and Accuracy per Time Point for ODE System Derivatives over the Interval $[0, t_{end}]$ - Additional First Order Methods

SIR Model Time in μs

t_{end}	Parallel FD.jl Gradient	Parallel FD.jl Multi-Threaded Gradient	FD.jl Muti-Threaded Gradient	FD.jl Gradient	Constant Analytic Gradient
10	1.40×10^2	1.47×10^2	3.41×10^2	3.59×10^2	3.19×10^2
100	4.33×10^2	4.39×10^2	1.10×10^3	1.12×10^3	1.40×10^3
1000	3.20×10^3	4.79×10^3	8.73×10^3	1.15×10^4	1.28×10^4

SIR Model Accuracy

t_{end}	Parallel FD.jl Gradient	Parallel FD.jl Multi-Threaded Gradient	FD.jl Muti-Threaded Gradient	FD.jl Gradient	Constant Analytic Gradient
10	1.19	1.19	1.19	1.19	1.19
100	2.75×10^2	2.75×10^2	2.75×10^2	2.75×10^2	2.75×10^2
1000	9.86×10^3	9.86×10^3	9.86×10^3	9.86×10^3	9.86×10^3

Table 6: Time (ns) and Accuracy per Time Point for ODE System Derivatives over the Interval $[0, t_{end}]$ - Additional Second Order Methods

SIR Model Time in μs

t_{end}	DES.jl Adjoint Hessian	DES.jl Non-Adjoint Hessian	FD.jl Double Jacobian	FD.jl Multi-Threaded Double Jacobian	Constant Analytic Hessian
10	3.72×10^4	1.18×10^4	1.00×10^3	9.81×10^2	1.16×10^3
100	4.23×10^5	1.30×10^5	3.27×10^3	3.19×10^3	5.28×10^3
1000	8.67×10^6	2.37×10^6	4.22×10^4	4.26×10^4	5.61×10^4

SIR Model Accuracy

t_{end}	DES.jl Adjoint Hessian	DES.jl Non-Adjoint Hessian	FD.jl Double Jacobian	FD.jl Multi-Threaded Double Jacobian	Constant Analytic Hessian
10	1.15×10^{-1}	1.15×10^{-1}	1.15×10^{-1}	1.15×10^{-1}	7.53×10^{-1}
100	7.24×10^1	7.24×10^1	7.24×10^1	7.24×10^1	1.33×10^2
1000	6.78×10^3	6.78×10^3	6.78×10^3	6.78×10^3	7.15×10^3

(both adjoint and non-adjoint) to calculate the Hessian, using the ForwardDiff.Hessian method, using the ForwardDiff.jacobian(ForwardDiff.jacobian) method without chunking (with and without multithreading), and using the constant value $\epsilon = 1 \times 10^{-6}$ in the analytic method.